

THIRTY-FOURTH ANNUAL  
PACIFIC NORTHWEST  
SOFTWARE QUALITY  
CONFERENCE

P N S Q C <sup>TM</sup>

October 17-19, 2016

World Trade Center Portland  
Portland, Oregon

Permission to copy without fee all or part of this material, except copyrighted material as noted, is granted  
provided that the copies are not made or distributed for commercial use.



## Welcome

The theme “Cultivating Quality in Software” has been very exciting for us as we have gone about planning this year’s conference. For me personally as a Software Engineering Manager and gentleman farmer, I have had a lot of fun with this idea of looking at all the ways we can cultivate quality within our own spheres. As you participate this year, we hope that you will see how this theme plays out in our lunch programs, evening events, talks and presentations. Our goal is for you to be able to take skills, tools, and traits back from PNSQC and use them to cultivate your own organizations.

This year we are pleased to have Peter Khoury kick-off our technical conference. Peter started his career as an engineer and found out that critical speaking, conflict management, and leadership are vital qualities for success in our industry. He is now a national speaker, executive presentation coach, and author of “Self-Leadership Guide”. Peter, who is also the founder of Magnetic Speaking, is going to help us bridge the gap “From Software Tester to Leader: How to Take a Radical Leap Forward at Work” so we can communicate like a VP but think like an engineer.

On Tuesday we welcome the return of Rex Black whose company is a worldwide leader in testing services. Rex is the author of 12 books and dozens of articles and possesses has vast experience in the collection and use of metrics. Although facts and measures are the foundation of true understanding, the misuse of metrics can also be the cause of much confusion. Rex will discuss with us some of his favorite “Stupid Metrics Tricks – and How to Avoid Them”.

If you have been to our conference in the past you will see that we have kept the format of multiple tracks, which (we hope) gives you plenty of choices throughout the two-day conference. We have also maintained the 45-minute presentation format with 10 minutes between sessions to ensure presenters and attendees have plenty of time to get settled and ready for the next topic.

Monday night will include a reception with poster papers on the mezzanine level along with hors d’oeuvres and beverages. This will be followed by hosted networking and discussion opportunities as part of our effort to cultivate everyone’s experiences and strengthen your organizations. One night we will be trying something a bit different – the program committee has worked with local establishments to create locations where you can gather, relax, and share your conference experiences with colleagues.

This year we have again broken away from our traditional birds of a feather luncheon discussions. Monday, we will be doing a grab and go lunch where you can choose between cultivating quality demonstrations and participating in thought-provoking games. For Tuesday, we will be having a plated lunch program where you can meet the speakers and participate in a tabletop challenge.

Our closing keynote address will be given by Darlene Greene. Darlene was a VP at McAfee where she served as the “face” of IT. Additionally, Darlene is the former Dean of the prestigious Culver Girls Academy and a retired U.S. Navy Commander. Darlene is going to help us in “Cultivating a Champion Mindset and Skills to Dramatically Improve Your Life”.

This conference is full of practical, useful, and valuable information. With everyone’s help, software quality continues to move forward. We must all play an active part! I am glad you are here and hope to associate and network with each of you as we strive to accomplish our mission, which is to “Enable knowledge exchange to produce higher quality software”.

Doug Reynolds  
President, PNSQC 2016



# TABLE OF CONTENTS

<b>Welcome</b> .....	iii
<b>PNSQC™ BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS</b> .....	viii
<b>PNSQC™ VOLUNTEERS</b> .....	ix
<b>PNSQC™ Call for Volunteers</b> .....	x
<b>Opportunities to Get Involved:</b> .....	x
<b>PNSQC™ 2017 Call for Presentations</b> .....	x
<b>Keynote Addresses</b>	
<i>From Software Tester To Leader: How To Take a Radical Leap Forward at Work</i> .....	1
Peter Khoury, <i>Magnetic Speaking</i>	
<i>Stupid Metrics Tricks—and How to Avoid Them</i> .....	3
Rex Black, <i>RBCS</i>	
<i>Cultivate a CHAMPION Mindset and Skills to Dramatically Improve Your Life</i> .....	5
Darlene Bennett Greene, <i>Consultant</i>	
<b>Invited Speakers</b>	
<i>Deliver Quality with Agile and Lean</i> .....	7
Johanna Rothman, <i>Rothman Consulting Group, Inc.</i>	
<i>How Testing Strategy can Increase Developer Efficiency and Effectiveness</i> .....	19
Brian Okken, <i>Test &amp; Code Podcast</i>	
<i>Social Engineering – How to Avoid Being a Victim</i> .....	21
Bhushan Gupta, <i>Gupta Consulting, LLC.</i>	
<i>Proactive Software Quality Assurance™ Overcomes Traditional “Traffic Cop” SQA Resistance.</i> .....	37
Robin Goldsmith, <i>Go Pro Management, Inc.</i>	
<i>Quality: 2020</i> .....	57
Brian Gaudreau, <i>Consultant</i>	

## Development Track

<b><i>Risk Management in Software and Hardware Development</i></b> .....	65
Christopher Alexander, <i>AGLX Consulting</i>	
<b><i>Third-party library mismanagement: How it can derail your plans</i></b> .....	79
Ruchir Garg, <i>Intel</i>	
<b><i>Innersource in Test Automation</i></b> .....	79
James Knowlton, <i>NAVEX Global</i>	
<b><i>Defect Prediction Model for estimating Project Schedule</i></b> .....	93
Rajesh Yawantkar, Debashish Barua, Anupriya Vatta, <i>Intel</i>	
<b><i>Infrastructure Orchestration to Optimize Testing</i></b> .....	107
Rohit Naraparaju, Sajeed Mayabba Kunhi, <i>Intel</i>	
<b><i>Cultivating Software Performance in Cloud Computing</i></b> .....	121
Li Chen, Colin Cunningham, Pooja Jain, <i>Intel</i> Chenggang Qin, Kingsum Chow, <i>Alibaba Inc.</i>	
<b><i>When Continuous Integration meets Application Security</i></b> .....	131
Vasantharaju MS, Harish Krishnan, <i>Intel</i>	
<b><i>How Architecture Decisions Solve Quality Aspects</i></b> .....	141
Anil Z Chakravarthy, Charulatha Dhandapani, <i>Intel</i>	
<b><i>*Strategies for Minimizing Continuous Integration Response Time</i></b> .....	149
Adrian Cook, Krishna Nattanmai, John Liao, <i>Intel</i>	

## Management Track

<b><i>Ensuring Quality with a Quality Team of One</i></b> .....	163
Kim Janik, <i>Intel</i>	
<b><i>Transforming Organizations to Achieve TMMi Certification</i></b> .....	173
Suresh Chandra Bose, Ganesh Bose, <i>Cognizant Business Consulting</i>	
<b><i>Measurement System Analysis with Attribute Data</i></b> .....	183
Nagesh NM, Praveen S, <i>Siemens</i>	
<b><i>Better Together</i></b> .....	193
Linda Wilkinson, <i>Evisions</i>	
<b><i>Breaching Barriers to Continuous Delivery with Automated Test Gates</i></b> .....	203
Chris Struble, <i>Vertafore</i>	

*\*May not be presented at conference.*

<b><i>Automated Testing in DevOps</i></b> .....	215
Rajesh Sachdeva, <i>Optum Technology Inc.</i>	
<b><i>*DITA and Evolution of Documentation to Empower Software</i></b> .....	233
Madhuri Karanth, <i>Intel</i> ; Sandeep Karanth, <i>DataPhi Labs</i>	
<b><i>*Product Documentation in Agile Software Development</i></b> .....	243
Madhuri Karanth, <i>Intel</i>	
<b><i>*Agile Fatigue: What is it and Why Should We Care?</i></b> .....	253
Heather M. Wilcox, <i>NWEA</i>	

## Testing Track

<b><i>Building Stakeholder Confidence Through an Automated Testing Solution</i></b> .....	263
Rebecca Long, <i>Spokane Teachers Credit Union</i>	
<b><i>Winning with Flaky Test Automation</i></b> .....	273
Wayne Matthias Roseberry, <i>Microsoft Corporation</i>	
<b><i>Automated Testing for Continuous Delivery Pipelines</i></b> .....	291
Justin Wolf, Scott Yoon, <i>Cisco</i>	
<b><i>Agile Testing Without Automation</i></b> .....	305
John Duarte, Eric Thompson, <i>Puppet</i>	
<b><i>Logic Programming to Generate Complex and Meaningful Test Data</i></b> .....	311
Donald Maffly, <i>Huron Consulting Group</i>	
<b><i>Testing Cloud Hosted Solution in Production</i></b> .....	331
Arvind Srinivasa Babu, Sajeed Mayabba Kunhi, Joshy A. Jose, <i>Intel</i>	
<b><i>*Test Estimation</i></b> .....	347
Shyam Sunder, <i>Sidra Medical &amp; Research Center</i>	
<b><i>*Unified Test Library and Fault-Tolerant Automatic Test Script Creation</i></b> .....	353
Chandrashekhar M V; Manini Sharma, <i>Intel</i>	

*\*May not be presented at conference.*

## **PNSQC™ BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS**

**Doug Reynolds** – Board Member & President  
*Tektronix, Inc.*

**Tim Farley** – Board Member, Program Chair & Vice President  
*Insitu*

**Rachel Eggers** – Board Member & Secretary  
*CDK Global*

**Bill Baker** – Board Member & Treasurer

**Brian Gaudreau** – Board Member, Audit & Marketing Chair

**Lloyd Bell** – Board Member & Operations Chair  
*Crowd Compass*

**Phil Lew** – Board Member & Social Media Chair  
*XBOSoft*

**Srilu Pinjala** – Program Co-chair  
*Lowes*

**Shauna Gonzales** – Conference Format Chair  
*Nike, Inc.*

**Hong Ye** – Volunteer Chair  
*Tektronix, Inc.*

## **PNSQC™ VOLUNTEERS**

**Rick Anderson**

**Ove Armbrust**

**Sue Bartlett**

**Ganesh Bose**

**Suresh Chandra Bose**

**Carol Brands**

**Laura Bright**

**Anil Z Chakravarthy**

**Robert Cohn**

**Moss Drake**

**Dan Grover**

**Sion Heaney**

**Aaron Hockley**

**Simon Howlett**

**Kathy Iberle**

**Jeremy Kuelh**

**Elena Lang**

**Michael Larsen**

**Jeyashekar Marimuthu**

**Launi Mead**

**Ashish Nirgudkar**

**Bill Opsal**

**Shivanand Parappa**

**Dave Patterson**

**Randy Pelligrini**

**Srilu Pinjala**

**Amith Pulla**

**Jan Ralston**

**S. Shashidhar**

**Jeanette Schadler**

**Anurag Sharma**

**Keith Stobie**

**Vadiraj Thayur**

**Patt Thomasson**

**Richard Vireday**

**Heather Wilcox**

## PNSQC™ Call for Volunteers

PNSQC is a non-profit organization managed by volunteers passionate about software quality. We need your help to meet our mission of enabling knowledge exchange to produce higher quality software. Please step up and volunteer at PNSQC.

**Benefits of Volunteering:** Professional Development, Contribution & Recognition

### Opportunities to Get Involved:

- **Program Committee** — Manages the submission, selection and scheduling of program content, and coordinates the efforts of authors and reviewers to ensure excellent conference content.
- **Invited Speakers Committee** — Collaborates with the Program Committee to identify and invite leaders in the global quality community to provide conference speakers.
- **Marketing Communications Committee** — Ensures the software community is aware of PNSQC events via electronic and print media; coordinates and collaborates with co-sponsors and other organizations to get the word out. Identifies potential exhibitors and solicits their participation.
- **Operations & Infrastructure Committee** — Develops techniques to enhance communications with PNSQC board and committee members as well as the PNSQC community at large. Responsible for the PNSQC website, SharePoint, and speaker recording.
- **Community & Networking Committee** — Implements networking opportunities for the software community. Manages the social networking channels of communication. Recruits and works with incoming volunteers to place them in committees. Provides programming for the networking opportunities at the conference. This includes the lunch time format and evening sessions. Responsibilities are recruitment of lead participants, compelling topics and titles and structure of the program.

**Contact Us:** by submitting your name in the conference survey which is a link sent to all PNSQC attendees or complete the contact form at [www.pnsqc.org/About/Contact](http://www.pnsqc.org/About/Contact) and address it to the PNSQC Volunteer Coordinator.

## PNSQC™ 2017 Call for Presentations

Inspired? Got an idea you want to tell us about? Consider submitting your ideas for PNSQC 2017. All it takes is a paragraph or two and selected authors receive waived fees.

Get more information at [www.pnsqc.org](http://www.pnsqc.org), link to the Call for Abstracts page for more details. Become part of the program by submitting an abstract.

# From Software Tester To Leader: How To Take a Radical Leap Forward at Work

## Peter Khoury, Magnetic Speaking

In this continuously shifting world of software development, technical leaders must demonstrate an unprecedented level of technical ability, confidence, and communication skills. The latter is often overlooked, either altogether, or until it is too late in one's career. But increasingly, public speaking skills represent an essential element in the career of any engineer who aspires to rise high within the corporation. To claim your right to the C-Suite, one must communicate one's concepts and imperatives effectively.

In this presentation, Peter Khoury, a former biotech engineer turned leadership speaker, will demonstrate the ways in which mastery of public speaking can empower one's career. He will discuss his EPIC framework – Expertise, Experience, Identity and Conditioning–the foundation for excellence in public speaking. When you attend his presentation, you will discover how to:

- Communicate like a VP yet think like an Engineer
- Build your confidence to be able to tackle any project or any position
- Systematically craft your proposals or project updates for maximum impact

Your technical skills will only take you so high in an organization. Soon you will hit a Glass Ceiling: Your ability to communicate, lead and persuade will take you to the next level.

***Peter Khoury** is the founder of Magnetic Speaking and an Ex-Engineer, turned author, national speaker and executive presentation coach. In addition to Public Speaking training, Peter is a regular speaker on the topics of Negotiations, Conflict Management and Leadership. He is the author of the book "Self-Leadership Guide."*



# Stupid Metrics Tricks— and How to Avoid Them

Rex Black, RBCS

If you have been in software engineering for a while—or in fact just in the working world in general for a while—you’ve probably seen someone do something stupid with metrics. Such mistakes raise a whole bunch of interesting questions. What are the most common metrics mistakes? Why are they mistakes? Why do people make these mistakes? Are you making these mistakes? Why use metrics at all, when there are so many mistakes? In this talk, Rex will give real-world examples of these mistakes, explain the management and economic theories behind metrics, and help you find ways to implement metrics that aren’t stupid.

***Rex Black** is President of RBCS ([www.rbc-us.com](http://www.rbc-us.com)), a worldwide leader in testing services, including consulting, outsourcing, assessment, and training. RBCS has over 100 clients spanning twenty countries on six continents. Rex has written 12 books and dozens of articles, and his best-seller, *Managing the Testing Process*, now in its third edition, has reached over 100,000 test professionals in the almost 20 years since it was first published.*



# Cultivate a CHAMPION Mindset and Skills to Dramatically Improve Your Life

Darlene Bennett Greene, Consultant

We say we want to prioritize quality—and we mean it—until we hit our first shortage of nearly any kind of resource. Then testing all too often bears the brunt of project schedule slips, personnel cuts, funding for tools, or even extra environments. Sometimes, after identifying significant quality mistakes, we fail to voice them, prioritize them, or correct them.

The research is clear: the top factors that bring success to people and companies are rooted in communication and leadership. Let's learn to communicate, think, and lead as a quality CHAMPION to prevent the "us" vs. "them" mentality and elevate quality across development, quality assurance, testing and then the entire company.

Become a leader that transforms a divisive culture to a teaming, transparent one. Reduce frustration, increase alignment, improve efficiency and effectiveness and bridge the divide that finding defects and promoting quality naturally creates. You can be the spark in creating a culture in which people feel valued, are optimized, and work together to promote quality.

***Darlene Bennett Greene** demonstrates that a toolkit with strong leadership, communication, and people skills delivers outstanding results across corporate, military, and educational domains. While a VP at McAfee, she served as the "face" of IT, delivering application programs and projects to business partners. In another IT role she led senior data, IPT, and security architects and project managers supporting an enterprise carrier class network, optical fiber ring, and centralized security architecture for 140 state agencies. She is a former Dean of the prestigious Culver Girls Academy and a retired U. S. Navy Commander who held three Commanding Officer positions. Recently, Darlene offered executive coaching and personalized leadership, communication, quality, culture, and employee engagement training to a health and technology company in the San Francisco area.*



# Deliver Quality with Agile and Lean

Johanna Rothman, Rothman Consulting Group, Inc.

We want to deliver quality products. Sometimes, it seems as if the world is against us: the developers can't finish their design and code in time, the testers can't finish their tests in time, or we're surprised by the number of problems at the end of the project or even an iteration. We need to see our quality as well as our code or tests. Agile and lean can help.

In this talk, Johanna will discuss how agile and lean limit the batch size by either limiting scope or work in progress, and how each can help. You'll learn how measurements change when you use agile or lean and how you can catch problems earlier. And, you'll see that some more traditional measurements are just as useful in an agile and lean project.

Bring your quality to the next level with agile and lean.

***Johanna Rothman**, known as the “Pragmatic Manager,” provides frank advice for your tough problems. She helps leaders and teams see problems and resolve risks and manage their product development.*

*Johanna was the Agile 2009 conference chair. She is the current [agileconnection.com](http://agileconnection.com) technical editor. Her most recent book is “Agile and Lean Program Management: Scaling Collaboration Across the Organization.” She writes columns for [techwell.com](http://techwell.com) and [projectmanagment.com](http://projectmanagment.com), and writes two blogs on her web site, [jrothman.com](http://jrothman.com), as well as a blog on [createadaptablelife.com](http://createadaptablelife.com).*

## Deliver Quality with Agile and Lean

# Deliver Quality with Agile and Lean

Johanna Rothman  
*Agile and Lean Program Management: Scaling Collaboration Across the Organization*  
@johannarothman  
www.jrothman.com  
jr@jrothman.com  
781-641-4046



## Is Your Agile Approach Working for You?

- Easy releasing is a problem:
  - QA doesn't "finish" the stories before the iteration ends
  - Integration testing has no home (no one's responsibility)
  - Quality is not clear for a story, iteration, or release
  - You need to fix defects past the iteration or feature-done
- You don't see that agile is helping your quality or speed of release

@johannarothman

2

© 2016 Johanna Rothman

## Our Agenda

- Agile and Lean and why they work
- Quality concerns
- What to measure
- Role of the QA person/group

@johannarothman

3

© 2016 Johanna Rothman

## Johanna's General Agile Picture



@johannarothman

4

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### The Principles Behind the Agile Manifesto

- Satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

@johannarothman

5

© 2016 Johanna Rothman

### Lean Principles

1. Eliminate waste.
2. Amplify learning.
3. Decide as late as possible.
4. Deliver as fast as possible.
5. Empower the team.
6. Build integrity in.
7. See the whole.

— Mary and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*

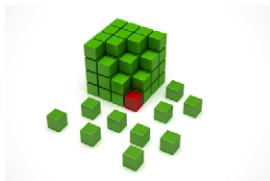
@johannarothman

6

© 2016 Johanna Rothman

### Why Agile/Lean Works

- Small batch size
- Collaboration among everyone
- Frequent feedback
- See working product
- Change is an effect



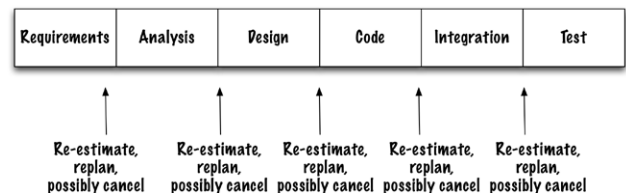
@johannarothman

7

© 2016 Johanna Rothman

### Waterfall: Illusion of Control

- But when could you tell when anything was ready?



@johannarothman

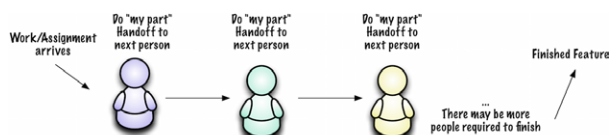
8

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### Resource Efficiency

- Is anyone concerned with utilization or expertise?
- Resource efficiency is based on experts, not throughput



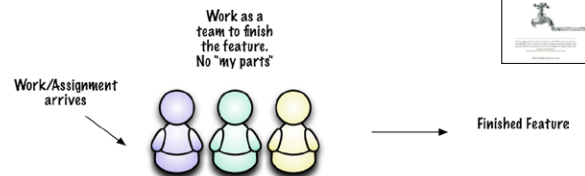
@johannarothman

9

© 2016 Johanna Rothman

### Flow Efficiency

- Based on throughput
- Throughput provides project/delivery momentum



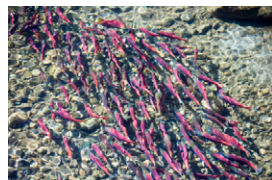
@johannarothman

10

© 2016 Johanna Rothman

### Consider Pairing, Swarming, Mobbing

- Pair: two people, one story
- Swarm: Entire team works on a story, not necessarily together
- Mob: Entire team works on one story all together



@johannarothman

11

© 2016 Johanna Rothman

### Questions for You to Ponder

- Are you working by design using resource efficiency?
- If yes, what would it take for you to work in flow efficiency?
- If no, do you fall into resource efficiency?
  - What would it take to collaborate on stories?
- What prevents you from working as a team?

@johannarothman

12

© 2016 Johanna Rothman

### Our Agenda

- Agile and Lean and why they work
- **Quality concerns**
- What to measure
- Role of the QA person/group

@johannarothman

13

© 2016 Johanna Rothman

### Where's Quality?

- Agile and lean are supposed to be better, faster, cheaper
- Cause vs. effect



@johannarothman

14

© 2016 Johanna Rothman

“Quality is value to  
someone”  
— Jerry Weinberg

@johannarothman

15

© 2016 Johanna Rothman

### Your Someones

- Product Owner
- Customer(s)
- Developers
- Testers
- Anyone else?



@johannarothman

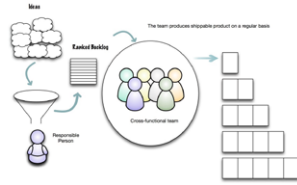
16

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### The PO Decides on Quality

- Acceptance criteria
- Frequent review of work in progress and finished work
- Boards help everyone see what's going on: done and not done

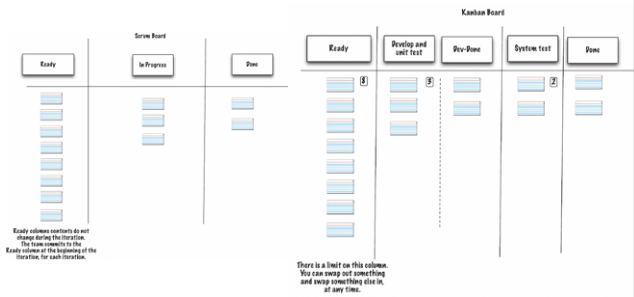


@johannarothman

17

© 2016 Johanna Rothman

### Boards Visualize the Work



@johannarothman

18

© 2016 Johanna Rothman

### Two Major Quality Concerns

- You gave me what I asked for. It's not what I need.
- It's not done.



@johannarothman

19

© 2016 Johanna Rothman

### Not What I Asked For Problem

- Use acceptance criteria
- Rapid and frequent feedback
- Keep stories small and build incrementally



@johannarothman

20

© 2016 Johanna Rothman

### Iterative and Incremental Creation

- Start with simplest thing possible
  - Add in more parts until you have a whole
    - Iterate on features
    - Incremental on details
- Example: Secure login for shopping cart
    1. Already-existing user can log in.
    2. Already-existing user can add too cart
    3. AE user can check out.
    4. Add new user. (Add cart and check-out stories...)
    5. Add security for users...

@johannarothman

21

© 2016 Johanna Rothman

### Quality Criteria

- What done means to the team
- Acceptance criteria on a story
- Release criteria for a release



@johannarothman

22

© 2016 Johanna Rothman

### Let's Discuss "Done"

- You've heard of "done-done" and "done-done-done"
- What do people mean when they say that?



@johannarothman

23

© 2016 Johanna Rothman

Done means the feature  
is releasable.

@johannarothman

24

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### What Do You Need for Releasable Features?

- Possible list:
  - Collaboration on the feature so more than one person understands the issues
  - Multiple eyes on the code
  - Sufficient test automation
  - Exploratory tests
  - Code & tests checked in
  - Accepted by PO
  - Documentation completed
- Maybe more for your environment



@johannarothman

25

© 2016 Johanna Rothman

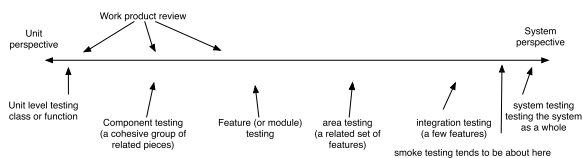
### Releasable Features Means Release is a Business Decision

@johannarothman

26

© 2016 Johanna Rothman

### What Kind of Testing Does Your Product Need?

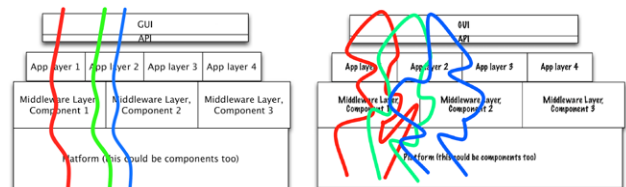


@johannarothman

27

© 2016 Johanna Rothman

### What Kind of Features Do You Have?



@johannarothman

28

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### Role of Automation

- If you create iteratively and incrementally, you will:
- Add features and their tests (unit, integration, system)
- Change features and their tests (unit, integration, system)
- Exploration & Automation!



@johannarothman

29

© 2016 Johanna Rothman

### Questions for You to Ponder

- Do you have areas of testing your team “ignores”?
- What could you do to address that?
- Does someone ask you to write test cases?
- Do you have automation debt?
- What can you do about test debt?

@johannarothman

30

© 2016 Johanna Rothman

### Our Agenda

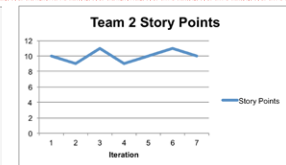
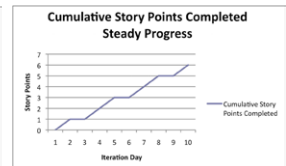
- Agile and Lean and why they work
- Quality concerns
- What to measure
- Role of the QA person/group

@johannarothman

31

© 2016 Johanna Rothman

### Several Velocity Stories



@johannarothman

32

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### “Problems” with Velocity

- Velocity is a measure of capacity, not productivity
- Not always predictable
- Individual to each team, and can vary with domain



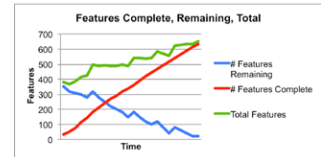
@johannarothman

33

© 2016 Johanna Rothman

### Measure Completed Features

- Completed features (running, tested features):
  - Your customers use them
  - You can release them
  - They are valuable
- Include total and remaining features so we have a sense of where we are
- Depends on deliverables, not epics or themes



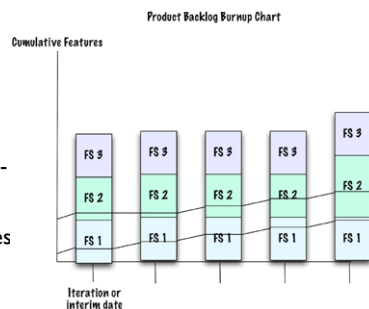
@johannarothman

34

© 2016 Johanna Rothman

### Product Backlog Burnup

- Real earned value
- Partial answer to “Where are we?”
- Shows value feature-by-feature
- Shows when features grow



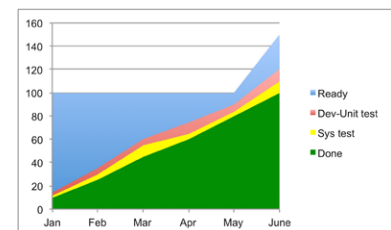
@johannarothman

35

© 2016 Johanna Rothman

### What Do You Want Less of?

- Work In Progress (across project)
- Defects
- Other “Less of”:
  - Multitasking
  - ?



@johannarothman

36

© 2016 Johanna Rothman

### What is Productivity in an Agile Environment?

- Features over time
- Teams take features, people don't take features
  - If people collaborate, swarm, or mob, personal "productivity" is irrelevant
- We don't normalize features between teams

@johannarothman

37

© 2016 Johanna Rothman

### Some Other Measures

- Time between internal/external releases (trend)
- Build time
- Quality scenarios (performance, reliability, functionality) as trends from build to build

@johannarothman

38

© 2016 Johanna Rothman

### Questions for You to Ponder

- Do you measure snapshots or trends of defects?
- Does your team try to release features even if you have found defects?

@johannarothman

39

© 2016 Johanna Rothman

### Role of QA

- Consider changing the name from QA to test
- Provide information about the product under test, not its goodness or badness



@johannarothman

40

© 2016 Johanna Rothman

## Deliver Quality with Agile and Lean

### The Agile/Lean Product Development Team Owns Quality

- PO defines quality
- Team implements it as a team
- QA/Tester provides information about the product as the team develops and tests it



@johannarothman

41

© 2016 Johanna Rothman

### Questions for You to Ponder

- Do you think of yourself as a “product development team”?
- If not, what would it take for you to do so?
- Does someone have a vested interest in you/your group owning quality?

@johannarothman

42

© 2016 Johanna Rothman

### Let's Stay in Touch

- Please link with me on LinkedIn:
  - [www.linkedin.com/in/johannarothman](http://www.linkedin.com/in/johannarothman)
- Subscribe to the Pragmatic Manager newsletter:
  - <http://www.jrothman.com/pragmaticmanager/>



@johannarothman

43

© 2016 Johanna Rothman

# How Testing Strategy can Increase Developer Efficiency and Effectiveness

## Brian Okken, Test & Code Podcast

We understand the benefits of automated testing and how this strategy directly improves developer efficiency and effectiveness. However, a poor test strategy also has costs that can grow over time and wipe out much of the gains. During this presentation we will explore the benefits and costs of automated testing, and describe a strategy that can maximize the benefits and minimize the costs. This strategy also intuitively makes sense, and should scale well to different sized teams. We will also explore system architecture, team organization, and tool resources that impact the effectiveness of automated tests, and what teams can do to modify their strategy to meet their situation.

*Brian Okken is a lead software engineer and team lead at Rohde & Schwarz USA, and has worked in test and measurement since 1996. Outside of work, Brian is actively involved in the Python community. He writes articles about software testing, development, and Python test frameworks at [pythontesting.net](http://pythontesting.net), and hosts a podcast about the same topics called “Test and Code.” Brian received a MSCS from the University of Oregon.*



# Social Engineering – How to Avoid Being a Victim

Bhushan Gupta, Gupta Consulting, LLC.

Social engineering (an act of exploiting people instead of computers) is one of the most dangerous tools in the hacker's toolkit to breach internet security. The Ubiquiti Networks fell victim to a \$39.1 M fraud as one of its staff members was hit by a fraudulent "Business Email Compromise" attack. Thousands of grandmas and grandpas are victim of phishing emails and are forced to pay ransom to have their data released.

In this new millennium, the cyber security game has changed significantly from annoying harmless viruses to stealing vital personal data, causing negative financial impact, demanding ransom, and spreading international political feud. Anyone with presence in the Cyber space has to protect himself/herself, the infrastructure, customers, and also deal with the legal repercussions in the event of a breach. In this talk, Bhushan will present the different types of social engineering practices, including use of social networks such as Facebook, Twitter, LinkedIn, the bad guys successfully use. The victims can range from the "C" levels (CEO, CFO, CTO), down to the individual contributors in an organization, or even to a grandparent on their laptop. The presentation will also discuss a variety of ordinary but effective measures such as awareness campaigns that organizations can take to minimize the risk of breach.

*Proven champion for quality and well-versed with software quality engineering, measurements, metrics, **Bhushan Gupta** is the principal consultant at Gupta Consulting, LLC. A Certified Six Sigma Black Belt (ASQ, HP), he possesses deep and broad experience in solving complex technical problems, change management, and coaching and mentoring. As a member of Open Web Application Security Project (OWASP) Portland Chapter, he is dedicated to the integration of Web application security into Agile software development life cycle.*

*Bhushan has a MS in Computer Science (1985) from New Mexico Tech and has worked at Hewlett-Packard and Nike Inc. in various roles. He was a faculty member at Oregon Institute of Technology, Software Engineering, from 1985 to 1995.*

# SOCIAL ENGINEERING - HOW NOT TO BE A VICTIM!

BHUSHAN GUPTA  
GUPTA CONSULTING, LLC.  
WWW.BGUPTA.COM

WHAT IS YOUR PASSWORD?

Jimmy Kimmel Live

@Gupta Consulting, LLC. www.bgupta.com

2

## JIMMY KIMMEL LIVE - OBSERVATIONS

- No hesitation to answer password specific questions
- Only one person realized that he was being asked to reveal his password
- Only one person realized that he has given away his password.
- Common Password – password123

Twitter Hack on June 9, 2016

120,000 Users opted for password - 123456

Its fair to assume that these people work somewhere and can be victims of serious social engineering attacks.

@Gupta Consulting, LLC. www.bgupta.com

3



## SOCIAL ENGINEERING INGREDIENT



@Gupta Consulting, LLC. www.bgupta.com

5

## WHAT IS SOCIAL ENGINEERING?

**Social engineering** is an attack vector that relies heavily on human interaction and often involves tricking people in breaking normal security procedures.

(WhatIs.com)

@Gupta Consulting, LLC. www.bgupta.com

6

## WHAT IS SOCIAL ENGINEERING?

**Social engineering**, in the context of information security, refers to psychological manipulation of people into performing actions or divulging confidential information.

(Wikipedia)

@Gupta Consulting, LLC. www.bgupta.com

7

## UBIQUITI NETWORKS ATTACK



- Ubiquiti Network Inc.  
San Jose, CA (Hong Kong Subsidiary)
- June 5, 2015 (after 7 months)
- CEO Fraud/BEC Attack
- Cost \$46.7M

- 2015 FBI Warning - \$215M

BEC – Business Email Compromise

@Gupta Consulting, LLC. www.bgupta.com

8

SOCIAL ENGINEERING - TROJAN HORSE  
(GREECE & TROY WAR)



@Gupta Consulting, LLC. www.bgupta.com

9

THE BIGGEST THREAT



"The biggest threat to the security of a company is not a computer virus, an unpatched hole in a key program or a badly installed firewall. In fact, the biggest threat could be you."

- Kevin Mitnick – Computer Security Consultant, Author

@Gupta Consulting, LLC. www.bgupta.com

10

Why?



@Gupta Consulting, LLC. www.bgupta.com

11

WE GROW UP WITH SOME CORE VALUES!



@Gupta Consulting, LLC. www.bgupta.com

12

## SCOUT LAW

A SCOUT IS:

★ TRUSTWORTHY	OBEDIENT
★ LOYAL	CHEERFUL
★ HELPFUL	THRIFTY
★ FRIENDLY	BRAVE
★ COURTEOUS	CLEAN
★ KIND	REVERENT


## THE GIRL SCOUT LAW

I will do my best to be  
honest and fair, ★  
friendly and helpful, ★★  
considerate and caring, ★  
courageous and strong, and  
responsible for what I say and do,  
and to  
respect myself and others, ★  
respect authority, ★  
use resources wisely,  
make the world a better place, and  
be a sister to every Girl Scout.

OUR FOUNDATION – TEACHINGS FROM CHILDHOOD

@Gupta Consulting, LLC. www.bgupta.com 13

## REWARDED BASED UPON - TEAM WORK, WIN-WIN, HELPFUL, LOYALTY, OBEDIENCE!




@Gupta Consulting, LLC. www.bgupta.com 14



@Gupta Consulting, LLC. www.bgupta.com 15

## ATTACK VECTORS AND LURES



@Gupta Consulting, LLC. www.bgupta.com 16

## ATTACK VECTORS - TACTICS

### Physical Vectors

- Shoulder Surfing
- Baiting – promise of goods to entice victim  
Infected USBs, CD
- Quid Pro Quo – service  
IT Impersonator trying to fix your system, obtaining a password and convincing you to load a utility (malware) on your system

@Gupta Consulting, LLC. www.bgupta.com

17

## ATTACK VECTORS - TACTICS

### Physical Vectors Cont..

- Tailgating – impersonation as a courier/messenger/friend of an employee
- Impersonating as an authority - Ubiquity
- Dumpster Diving
- Pretexting – scammer creating false trust

@Gupta Consulting, LLC. www.bgupta.com

18

## ATTACK VECTORS - TACTICS

### Digital Vectors:

- Social Media – Obtaining information from Facebook or Instagram and create fake profiles  
Admiral James Stavridis (NATO Supreme Allied Commander Europe) Facebook Profile

@Gupta Consulting, LLC. www.bgupta.com

19

## WHAT IS PHISHING?

**Phishing** is the attempt to obtain [sensitive information](#) such as usernames, passwords, and [credit card](#) details (and sometimes, indirectly, [money](#)), often for malicious reasons, by masquerading as a trustworthy entity in an [electronic communication](#).

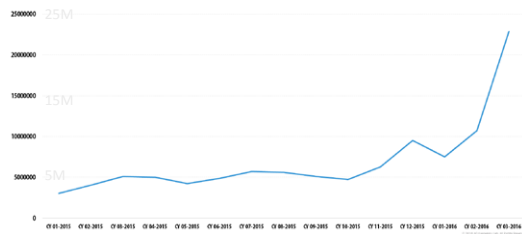
While Phishing, an attacker casts a wider net hoping that someone will be tricked.

Source: Wikipedia

@Gupta Consulting, LLC. www.bgupta.com

20

## PHISHING TREND – JANUARY 2015 TO MARCH 2016



Source: Securitylist.com

@Gupta Consulting, LLC. www.bgupta.com

21

## ATTACK VECTORS - PHISHING

### Types of phishing:

- Phishing – attacker casts a broad net
- Spear Phishing – somewhat more specific to one person
- Whaling – phishing targeted at a person with specific valuable information

@Gupta Consulting, LLC. www.bgupta.com

22

## PHISHING TARGETS - ALMOST ANYONE

- Individuals (elderly, common people, security experts)
- Influential people
- Corporations

@Gupta Consulting, LLC. www.bgupta.com

23

## MECHANICS OF A PHISHING ATTACK

- Attacker sends out an email that appears to be legitimate
- Email either directs the receiver to click on a link or perform an action
- When sent to the linked site, the attacker intends to:
  - Gather personal and confidential information
  - Install malware to get access to the system
- If the email requires an action it is either to collect confidential information or take an action (such as transferring money) to benefit the attacker
- The email often has threats - harmful consequences

@Gupta Consulting, LLC. www.bgupta.com

24

## A PHISHING EXAMPLE

**From:** "Catherine Arnold" <Catherine\_Arnold@comcast.net>  
**To:** "Zafar Haq" <Zafarpx@gmail.com>, "amanda.radclyffe" <amanda.radclyffe@canopyonline.com>, "andrea.mayrose" <andrea.mayrose@gmail.com>, "anelson" <anelson@cemins.com>, "bhushan.gupta" <bhushan.gupta@comcast.net>  
**Sent:** Wednesday, June 29, 2016 12:40:10 AM  
**Subject:** just wanted to say hi

Hi,

We haven't talked for a while so I just wanted to say hi and share with you some information, read more here please <http://trecyfrizi.carhandle.com/ineat>

Catherine\_Arnold@comcast.net

@Gupta Consulting, LLC. [www.bgupta.com](http://www.bgupta.com)

25

## A SPEAR PHISHING EXAMPLE

**Important Banking Alert**  
**Sent By:** Bank Of America **On:** Jun 06/29/16 11:05 AM

"Bank Of America"  
+ Add to Address Book

**Dear Valued Customer,**

Unfortunately, there has been a problem processing your statement information for this month. Please review our information requirements. You will be able to update your information quickly and easily using our secure server web form. Please understand that without promptly updating your Online Account, service may be discontinued. To update your billing information:

please visit our secure server web form by clicking: <http://www.bankofamerica.com>

Sincerely,

Bank Of America Customer Centre

### ABOUT THIS MESSAGE:

This is a service email from Bank of America. Please note that you may receive service emails in accordance with your Bank of America service agreements, whether or not you elect to receive promotional email. Read our Privacy Notice.

Please don't reply directly to this automatically generated email message.

Bank of America Email, NC1-002-08-25, 101 N College St., Charlotte, NC 28255

Bank of America, N.A. Member FDIC. Equal Housing Lender

© 2010 Bank of America Corporation. All rights reserved.

Bank of America Email, 8th Floor-NC1-002-08-25, 101 South Tryon St., Charlotte, NC 28255-0001

@Gupta Consulting, LLC. [www.bgupta.com](http://www.bgupta.com)

26

## WHALING ATTACK

Focused on an individual who has highly valuable information – C Levels

For a successful whaling attack

- Attacker performs significant research ahead of attack
- Takes place when the C-Level executive admin forwards an email to the finance department

@Gupta Consulting, LLC. [www.bgupta.com](http://www.bgupta.com)

27

## CHARACTERISTICS OF A PHISHING EMAIL

Not as polished as a legitimate email

- Grammar
- Salutation (generic)
- Wrong Information

@Gupta Consulting, LLC. [www.bgupta.com](http://www.bgupta.com)

28

## ATTACK LURES – MOST POPULAR IN 2014

- Big News – Malaysian Airline Flight 370
- Celebrity Gossip – Death of Robin Williams in 2012
  - Link to site to view the video, WORM\_GAMARUE.WSTQ
- Movies – Annie, Hobbit: Battle of the five armies
  - Malicious links, Adware
- Tech Games – Flappy Bird
- Social Media Scams – LinkedIn
- Scare Tactics – Ebola Outbreak

@Gupta Consulting, LLC. www.bgupta.com

29

## PHISHING - HOW TO PROTECT YOURSELF?



@Gupta Consulting, LLC. www.bgupta.com

30



31

## THE BIGGEST THREAT

Educate/  
Train

"The biggest threat to the security of a company is not a computer virus, a hacker, a hole in a key program or a badly installed firewall. The biggest threat could be you."

- Kevin Mitnick – Computer Security Consultant, Author



@Gupta Consulting, LLC. www.bgupta.com

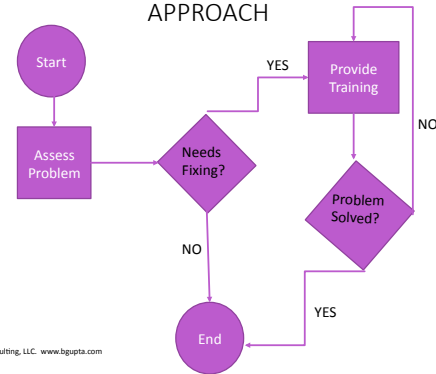
32

## HUMAN TRAITS

Big Five Trait	Trait Description
<i>Openness to experience</i>	"People scoring high on the openness scale are] characterized by such attributes as open-mindedness, active imagination, preference for variety, and independence of judgment."
<i>Conscientiousness</i>	"People [scoring] high on the conscientiousness scale tend to distinguish themselves for their trustworthiness and their sense of purposefulness and of responsibility. They tend to be strong-willed, task-focused, and achievement-oriented."
<i>Extraversion</i>	"People scoring high on the extraversion scale tend to be sociable and assertive, and they prefer to work with other people."
<i>Agreeableness</i>	"People [scoring] high on the agreeableness scale tend to be tolerant, trusting, accepting, and they value and respect other people's beliefs and conventions."
<i>Neuroticism</i>	"People [scoring] high on the [neuroticism] scale tend to experience such negative feelings as emotional instability, embarrassment, guilt, pessimism, and low self-esteem"

33

## APPROACH



@Gupta Consulting, LLC. www.bgupta.com

34

## ASSESS YOUR VULNERABILITY

Assess your vulnerability by simulating controlled experiments:

- Baiting – promise of goods to entice victim
- Quid Pro Quo – service
- Tailgating – impersonation as a courier/messenger/ friend of an employee
- Impersonating as an authority
- Dumpster Diving

**Phishing, spear phishing, whaling**

@Gupta Consulting, LLC. www.bgupta.com

35

## SIMULATING A PHISHING EXPERIMENT

### Considerations

- Support
- Experiment Logistics
- Metrics
- Outcome

@Gupta Consulting, LLC. www.bgupta.com

36

## SIMULATE A PHISHING EXPERIMENT – SUPPORT

### Build Support

- Company ethics
- Championship from upper management
- Support from participating managers

@Gupta Consulting, LLC. www.bgupta.com

37

## PHISHING EXPERIMENT – SAMPLE

### Characteristics

- Statistically viable Sample size - 20 to 30% of population
- Appropriate representation – high value targets
  - System Administrators – Level of Privilege
  - C-Levels (CEO, CFO, CTO)- Decision Makers
  - Finance – Revenue
  - Sales/Marketing - Extroverts
  - Human Resources – Access to personal information
  - Design Groups – seeking new ideas

@Gupta Consulting, LLC. www.bgupta.com

38

## SIMULATE PHISHING EXPERIMENT - LOGISTICS

### Crafting a Phishing Email

- Appear to come from a legitimate source
- Compelling reason or enticing element to take an action
  - Incentive
  - Threat / consequence
  - Entice user to read and take action multiple times
  - Address interest of entire population

@Gupta Consulting, LLC. www.bgupta.com

39

## SIMULATE PHISHING EXPERIMENT - LOGISTICS

### Data collection – time period

- Optimal time
- Follow the schedule

### Resources – Security operations + QA

- Account for tools – price and training
- Creation of experiments
- Data analysis

@Gupta Consulting, LLC. www.bgupta.com

40

## SIMULATE PHISHING EXPERIMENT - ASSESS PROBLEM MAGNITUDE

### Metrics:

- Percentage of population fell victim
- Victim Rate by Time: % by time – 24, 48, 72 hours, beyond
- Percentage of people falling victim multiple times
- Group affinity - # of victims by Group(s)
- Rate of Proactive Reporting

@Gupta Consulting, LLC. www.bgupta.com

41

## SIMULATE PHISHING EXPERIMENT - OUTCOME

### Analysis Outcome:

- Is it a problem that needs attention based upon security policy?
- How wide spread the problem is?
  - Particular group(s) - HR, Finance, Marketing
  - Entire Organization

@Gupta Consulting, LLC. www.bgupta.com

42

## COMBAT THE PROBLEM

- Assess problem magnitude
- Validate against your organization security objectives
- Set goals for social engineering vulnerability
  - Overall Victims Rate <10% in next 6 months
  - X Group victim Rate < 5% in next 6 months
- Design and provide Training
- Assess impact and follow up

@Gupta Consulting, LLC. www.bgupta.com

43

## PROVIDING EDUCATION/TRAINING



@Gupta Consulting, LLC. www.bgupta.com

44

## TRAINING – DESIGN/CONTENT

### Identification of email legitimacy:

- Source validation
- Evaluation of email content
  - Grammatical Errors
  - Professional vs. crude email
  - Source of origin
  - Disclaimers if any
  - Getting basic link information – IP Address

@Gupta Consulting, LLC. www.bhgupta.com

45

**From:** "Bank Of America" <onlinebanking.support1@verizon.net>

**Sent:** Wednesday, June 29, 2016 11:05:37 AM

**Subject:** Important Banking Alert

**Dear Valued Customer,**

@Gupta Consulting, LLC. www.bhgupta.com

46

### Important Banking Alert

**Sent By:** Bank Of America **On:** Jun 06/29/16 11:05 AM

"Bank Of America"  
Add to Address Book

**Dear Valued Customer,**

Unfortunately, we have had a problem processing your statement information for this month. Please review your information, requirements. You will be able to update your information quickly and easily using our secure server web form. Please understand that without promptly updating your Online Account service may be discontinued. To update your billing information

please visit our secure server web form by clicking <http://www.bankofamerica.com>

Sincerely,

Bank Of America Customer Centre

#### ABOUT THIS MESSAGE :

This is a service email from Bank of America. Please note that you may receive service emails in accordance with your Bank of America service agreements, whether or not you elect to receive promotional email.

Read our Privacy Notice

Please don't reply directly to this automatically generated email message.

Bank of America Email, NC1522-09-01, 180 N College St., Charlotte, NC 28255

Bank of America, N.A. Member FDIC. Equal Housing Lender

© 2016 Bank of America Corporation. All rights reserved.

Bank of America Email, 8th Floor-NC1-002-08-25, 101 South Tryon St., Charlotte, NC 28255-0001

@Gupta Consulting, LLC. www.bhgupta.com

47

**Fwd: Hello Bhushan Gupta**

**Sent By:** rosy gupta **On:** Aug 08/08/16 12:41 PM

**To:** bhushan gupta

FYI

**From:** "cass4021" <cass4021@p3nlhg939.shr.prod.phx3.secureserver.net>

**To:** "rosy gupta" <rosy.gupta@comcast.net>

**Sent:** Monday, August 8, 2016 9:16:21 AM

**Subject:** Hello Bhushan Gupta

Bhushan Gupta, you are approved for your Loan from \$200 to \$1000

Click to Apply Now I

[http://cassicarver.com/della-diverted.php?](http://cassicarver.com/della-diverted.php?lqolez=aHR0cDovL2Vhc3JjYXNoc2VhcmNoLmNvbS8_bD1OSFp6ckFzZW5hQVZZL)

[lqolez=aHR0cDovL2Vhc3JjYXNoc2VhcmNoLmNvbS8\\_bD1OSFp6ckFzZW5hQVZZL](http://cassicarver.com/della-diverted.php?lqolez=aHR0cDovL2Vhc3JjYXNoc2VhcmNoLmNvbS8_bD1OSFp6ckFzZW5hQVZZL)

Oregon 97007, USA <Aug Mon 12:16:20 08 2016

@Gupta Consulting, LLC. www.bhgupta.com

48

## TRAINING – DESIGN/CONTENT

Importance of not deleting the email

- Forensics
- Build threat data
- Example for future training

@Gupta Consulting, LLC. www.bgupta.com

49

## WHAT TO DO IF:

A phishing email has been identified: Provide Documented process for :

- Contact Person – phone and email
- Safe ways to deal with the email - forward to security group

The link has been followed

- Capturing the web site
- Basic system monitoring techniques

@Gupta Consulting, LLC. www.bgupta.com

50

## POST-TRAINING ACTIVITIES

- Monitor phishing email rate
- Run another experiment to measure the impact
- Train new employees on a regular basis
- Offer supplemental training as necessary

@Gupta Consulting, LLC. www.bgupta.com

51

## ASSESS YOUR OVERALL VULNERABILITY

Other vectors:

- Tailgating – impersonation as a courier/messenger/ friend of an employee
- Dumpster Diving
- Baiting – promise of goods to entice victim
- Quid Pro Quo – service
- Impersonating as an authority

@Gupta Consulting, LLC. www.bgupta.com

52

## SIMULATE PHISHING EXPERIMENT - TOOLS

### Industry Tools:

- Wombat Security Technologies – wide range of utilities
- phishingbox (phishingbox.com)
  - Capability to execute an attack and gather results
- Kali Linux – Social Engineering Module – Open Source

@Gupta Consulting, LLC. www.bgupta.com

53

## HUMAN HARDWARE BUGS

- High level of trust
- Believing in simplicity
  - Password
- Losing passion of something important
- Failure to take a timely action
  - Apply patches
  - Provide documentation
  - Review requests for action

@Gupta Consulting, LLC. www.bgupta.com

54

## BEST PRACTICES

- Develop an enterprise security plan
- Put right team in place
- Deploy defenses
- Perform threat analysis
- Continued training and awareness
- Respond to incidence

@Gupta Consulting, LLC. www.bgupta.com

55

## CYBERSECURITY IS A BUSINESS ISSUE!!

- Support from Management
- Appropriate Budget
- Multiple forms of protection

@Gupta Consulting, LLC. www.bgupta.com

56

WANT TO KNOW MORE?

White Paper from SANS

<https://www.sans.org/reading-room/whitepapers/engineering/methods-understanding-reducing-social-engineering-attacks-36972>

@Gupta Consulting, LLC. www.bgupta.com

57

Copyright 2008 by Randy Glasbergen.  
www.glasbergen.com



"Instead of waiting for someone to steal my identity, I'm going to auction it on eBay!"

58



59



@Gupta Consulting, LLC. www.bgupta.com

60

# Proactive Software Quality Assurance™ Overcomes Traditional “Traffic Cop” SQA Resistance

Robin Goldsmith, Go Pro Management, Inc.

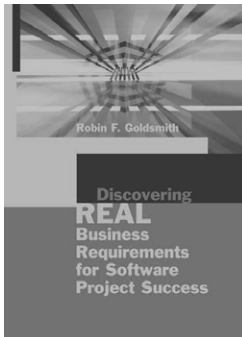
Proactive SQA™ is a key basis of significant value-enhancing revisions to IEEE SQA Std. 730's often-resisted “traffic cop” enforcement of procedural compliance. Although most of what is called SQA today actually is just testing, true SQA is much different from quality control (QC) testing. SQA can and should do far more, contributing proactively to assure the software process in fact does the right things well so it truly produces high quality cheaper, preventing errors or catching them earlier when they can be fixed more easily. This interactive workshop positions SQA and explains the six proactive functions it should perform to provide far greater value.

- Why traditional reactive software quality assurance is resisted and fails
- Working definition of quality that overcomes common definition issues
- The six proactive functions effective SQA should perform

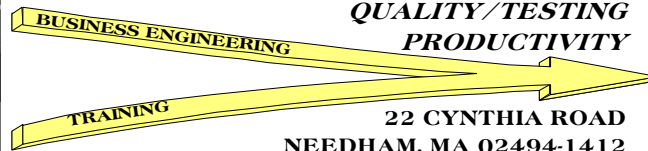
**Robin F. Goldsmith, JD** is President of the Software Quality Group of New England and Needham, MA consultancy Go Pro Management, Inc. He advises and trains business and systems professional on risk-based Proactive Software Quality Assurance and Testing™, REAL requirements, REAL ROI™, metrics, outsourcing, project and process management. Subject expert for IIBA's BABOK v2 and TechTarget.com, he is author of the book, *Discovering REAL Business Requirements for Software Project Success*, and the forthcoming book, *Cut Creep — Put Business Back in Business Analysis to Discover REAL Business Requirements for Agile, ATDD, and Other Project Success*.

# Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

*Robin F. Goldsmith, JD*



**GO PRO MANAGEMENT, INC.**  
SYSTEM ACQUISITION & DEVELOPMENT  
QUALITY/TESTING  
PRODUCTIVITY



22 CYNTHIA ROAD  
NEEDHAM, MA 02494-1412  
INFO@GOPROMANAGEMENT.COM  
WWW.GOPROMANAGEMENT.COM  
(781) 444-5753 VOICE/FAX

©2016 GO PRO MANAGEMENT, INC.

- 1 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## Are You Familiar with QA as 'Traffic Cop'



www.123rf.com/photo\_12808921

- Enforcing compliance
  - Document formats
  - Following procedures
- Obstacle to
  - Progress
  - Delivery
- Understandable  
**RESISTANCE**

©2016 GO PRO MANAGEMENT, INC.

- 2 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## Objectives

- Distinguish system/software quality, quality assurance (SQA), and quality control (SQC).
- Analyze conventional SQA/standards and why they so often are resisted, ignored, and/or fail.
- Describe the six functions Proactive SQA™ performs so
  - Involved parties understand and willingly participate in meaningful methods to assure software quality
  - Resisted practices are reduced, such as being a 'traffic cop'
  - Higher quality software truly is delivered quicker and cheaper.

***Proactive SQA™ is a key basis of significant value-enhancing revisions to IEEE SQA Std. 730-2014***

## Exercise: What is System Quality?

***System Quality***

***Software Quality***

# Exercise: What is SQA?

## *System Quality Assurance*

### *Software Quality Assurance (SQA)*

©2016 GO PRO MANAGEMENT, INC. - 5 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## Software Quality Assurance (SQA)

## - 5 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

# System vs. Software Quality

## Relevance to SQC/SQA

- At which life cycle phase is it decided whether solution includes hardware?
  - Requirements
  - Design
  - Build and test
- What impact on quality activities
  - If system vs. software initially misidentified?
  - If system vs. software subsequently changes?

***Is system vs. software distinction relevant, useful?***

---

---

©2016 GO PRO MANAGEMENT, INC. - 6 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

- 

- 6 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## *Quality Is Key to Delivering Quicker and Cheaper*

- “Quality is free”
- Cost of (poor) quality
  - Assessment (appraisal)
  - Prevention
  - Failure
    - » Internal
    - » External

-- Philip Crosby

## *Some Common Definitions of Quality*

- Customer satisfaction
- Meets or exceeds customer expectations
- Optimization, value
- Conformance to requirements (Philip Crosby)
- Percent of (a sample of) products passing inspection for defects; lack of defects (~Deming)
- Minimal variation within specification (Six Sigma)
- Fitness for use (Joseph Juran)

***Any problems with these definitions? Relation to systems?***



## *What We Mean By System Quality*

- Fits system specs
- Runs efficiently
- Doesn't blow up
- Follows standards
- Current technology
- Modern techniques
- Easily modified
  - without code change
  - when code changes

©2016 GO PRO MANAGEMENT, INC.

- 9 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality



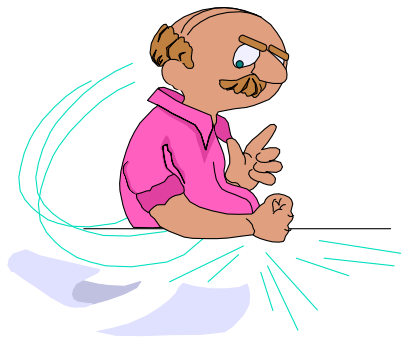
## *What Others Mean By System Quality*

- Does what needs to be done correctly
- Performs adequately
- Reliable/consistent
- Easy to use
- Supported quickly and correctly
- On-time, in budget

©2016 GO PRO MANAGEMENT, INC.

- 10 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## Until We Share a Common Definition of System Quality...



- ✓ Users, managers, developers, and Quality professionals will continue to disappoint each other
- ✓ Each has a different idea of what to deliver and how to tell whether it has been delivered adequately
- ✓ Each thinks the others don't care about Quality

©2016 GO PRO MANAGEMENT, INC.

- 11 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## ★ Quality Dimension: Quality of Design (What's it need to do)

- Required functions, capabilities, and performance levels defined appropriately
  - needs of all stakeholders identified
  - definitions accurate and complete
  - meaningful common understanding
- Design suitably meets requirements
- Costs/benefits/schedules are accurate
- Trade-offs based on adequate information

©2016 GO PRO MANAGEMENT, INC.

- 12 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality



### ★ *Quality Dimension: Quality of **Conformance** (How it's produced)*

- Products conform to design
- Products apply standards/conventions
- Workers use expected skill and care
- Workers apply defined methods, tools
- Management uses appropriate practices
- Product is delivered on-time, in-budget



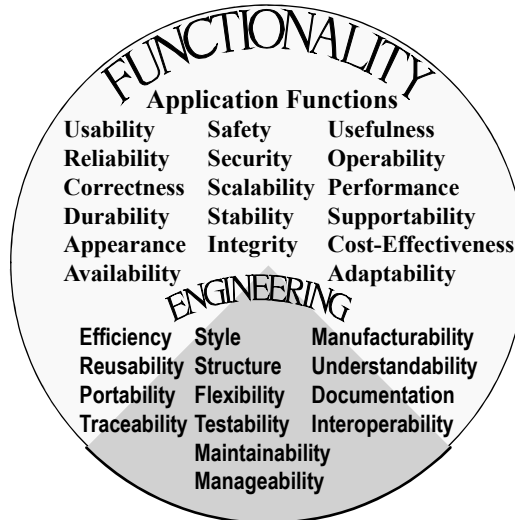
### ★ *Quality Dimension: Quality of **Performance** (How it's delivered)*

- Product is available as needed for use
- Product works in intended manner
- Product works reliably and accurately
- Product handles workload adequately
- Product is supported and maintained  
responsively



## ★ Addressing Quality Factors

Factors:  
Exterior  
Interior  
Future

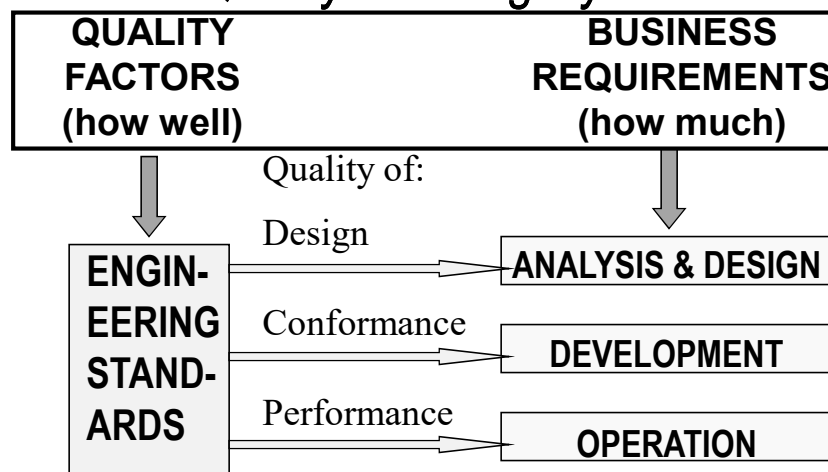


©2016 GO PRO MANAGEMENT, INC.

- 15 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality



## Turning Requirements Into a Quality Working System



©2016 GO PRO MANAGEMENT, INC.

- 16 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## **Our Working Definition of System Quality**

The **extent** to which **it meets** weighted stated and implied exterior, interior, and future **REAL business requirements** of all affected internal and external stakeholders **consistent with standards** of design, workmanship, and performance.

The more of the relevant requirements which are met, and the more demanding the standards are with respect to meeting those requirements, the higher the quality.

**Quality** is absolute. The amount of quality one receives is governed by available resources, priorities, and other constraints.

**Value** is the perceived benefit of quality received relative to the costs of producing and receiving it.

## **Engineered Deliverable Quality™**

### **How Much**

### **How Well**

Deliverable Capability	Weight/ Priority	Minimum	Desirable	Ideal

## Quality Assurance (QA) vs. Quality Control (QC)/Testing

Dynamic  
Code  
Execution

- QC/Testing examines end products, typically for conformance to specifications (but which often are referred to as 'requirements')

- QA assures the processes producing the end products produce quality products

Static Reviews  
of Requirements  
and Designs

- To some, that means examining intermediate products within the development process
- Often checks compliance of documents/procedures to standards/guidelines ("traffic cop")

***These are QC too—examining products***

## IEEE Std 12207-2008 Systems and software engineering —Software life cycle processes 1/4

### 7.2.3 Software Quality Assurance Process

#### 7.2.3.1 Purpose

The purpose of the Software Quality Assurance Process is to provide assurance that work products and processes comply with predefined provisions and plans.

***Starting point for revision of IEEE Std. 730 for SQA***

## IEEE Std 12207-2008 Systems and software engineering —Software life cycle processes 2/4

### 7.2.3.2 Outcomes

As a result of successful implementation of the Software Quality Assurance Process:

- a) a strategy for conducting quality assurance is developed;
- b) evidence of software quality assurance is produced and maintained;
- c) problems and/or non-conformance with requirements are identified and recorded; and
- d) adherence of products, processes and activities to the applicable standards, procedures and requirements are verified.

### 7.2.3.3 Activities and tasks

The project shall implement the following activities in accordance with applicable organization policies and procedures with respect to the Software Quality Assurance Process.

## IEEE Std 12207-2008 Systems and software engineering —Software life cycle processes 3/4

**7.2.3.3.1 Process implementation.** This activity consists of the following tasks:

**7.2.3.3.1.1** A quality assurance process suited to the project shall be established. The objectives of the quality assurance process shall be to assure that the software products and the processes employed for providing those software products comply with their established requirements and adhere to their established plans.

**7.2.3.3.1.2** The quality assurance process should be coordinated with the related Software Verification (subclause 7.2.4), Software Validation (subclause 7.2.5), Software Review (subclause 7.2.6), and Software Audit (subclause 7.2.7) Processes.

**7.2.3.3.1.3** A plan for conducting the quality assurance process activities and tasks shall be developed, documented, implemented, and maintained for the life of the contract. The plan shall include the following:

- a) Quality standards, methodologies, procedures, and tools for performing the quality assurance activities (or their references in organization's official documentation).
- b) Procedures for contract review and coordination thereof.
- c) Procedures for identification, collection, filing, maintenance, and disposition of quality records.
- d) Resources, schedule, and responsibilities for conducting the quality assurance activities.

Original Std. 730 scope

## IEEE Std 12207-2008 Systems and software engineering —Software life cycle processes 4/4

- e) Selected activities and tasks from supporting processes, such as Software Verification (subclause 7.2.4), Software Validation (subclause 7.2.5), Software Review (subclause 7.2.6), Software Audit (subclause 7.2.7), and Software Problem Resolution (subclause 7.2.8).

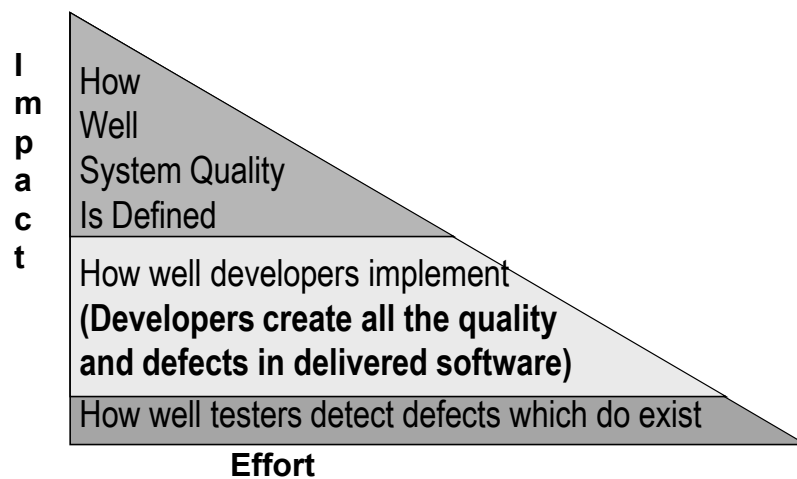
**7.2.3.3.1.4** Scheduled and on-going quality assurance activities and tasks shall be executed. When problems or non-conformances with contract requirements are detected, they shall be documented and serve as input to the Problem Resolution Process (subclause 7.2.8). Records of these activities and tasks, their execution, problems, and problem resolutions shall be prepared and maintained.

**7.2.3.3.1.5** Records of quality assurance activities and tasks shall be made available to the acquirer as specified in the contract.

**7.2.3.3.1.6** It shall be assured that persons responsible for assuring compliance with the contract requirements have the organizational freedom, resources, and authority to permit objective evaluations and to initiate, effect, resolve, and verify problem resolutions.

***How similar is this to what your organization does?***

## System Quality Results From

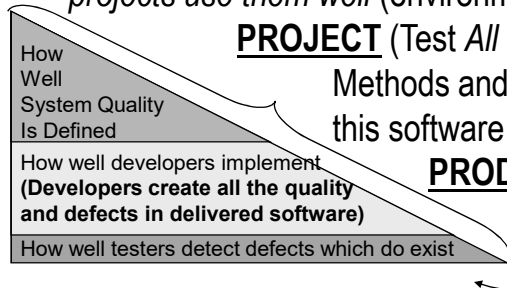


## Proactive System Quality Assurance (SQA)<sup>TM</sup> Direction of New IEEE Std. 730



### PROCESS

**Define** appropriate methods and techniques and **assure** all projects use them well (environment that promotes quality)



### PROJECT (Test All Development Deliverables)

Methods and techniques used to create this software product were appropriate

### PRODUCT (Testing the Code)

Delivered software works properly

©2016 GO PRO MANAGEMENT, INC.

- 25 Proactive SQA<sup>TM</sup> Overcomes the 'Traffic Cop' Mentality

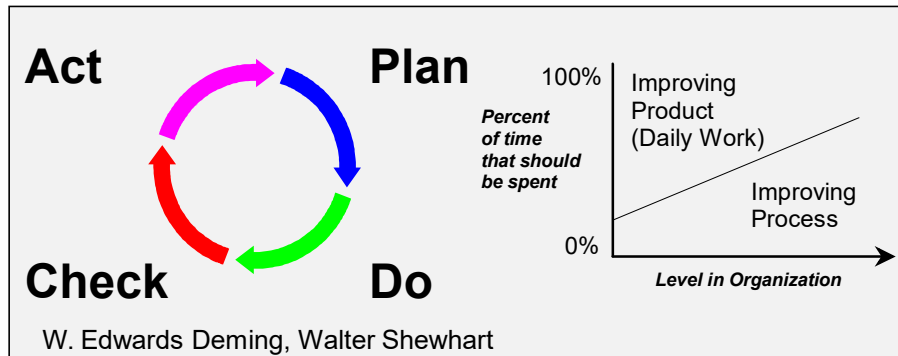
## Proactive SQA<sup>TM</sup>: Establishes an Environment that Promotes Quality



©2016 GO PRO MANAGEMENT, INC.

- 26 Proactive SQA<sup>TM</sup> Overcomes the 'Traffic Cop' Mentality

## Key Quality Environment Approaches



Use data to fix the problem and the source of the problem

©2016 GO PRO MANAGEMENT, INC.

- 27 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## The 6 Functions of SQA

**Actually  
HELP**

**A  
S  
S  
U  
R  
E  
V  
S.  
D  
O**

- ① Define Quality Assurance Plans (What to do)
- ② Define, methods, practices, and standards (How to do it well)
- ③ Assure systematic quality controls of processes and products (Make sure it gets done right)
- ④ Maintain quality records (Keep track of it)
- ⑤ Analyze and report on quality (Learn from it)
- ⑥ Direct attention to improving quality (Encourage it)

©2016 GO PRO MANAGEMENT, INC.

- 28 Proactive SQA™ Overcomes the 'Traffic Cop' Mentality

## 1 Define Quality Assurance Plans

- The project plan for QA, becomes part of overall project plan—tasks, resources, budget, schedule
- Identifies every task and other information needed to assure software product quality
  - Templates, common to all projects
  - Tasks unique to project
  - Balanced with risk, needs, and constraints
- Used to monitor/control progress

***Entire focus of IEEE Std. 730 until current revision***

## QA Plan, Very Detailed Deliverables Mil. Std. 2167 Requirements Analysis Phase

Computer Software Configuration Item  
Functional Requirements  
Performance Requirements  
Interface Requirements  
Qualification Requirements  
Software Requirements Specification  
Interface Requirements Specification  
Software Development Plan  
Software Standards and Procedures Manual  
Software Configuration Management Plan  
Software Quality Evaluation Plan  
Operational Concept Document  
Software Specification Revue  
Allocated Baselines for each CSCI  
Authenticated SRS  
Authenticated IRS(s)  
Ongoing Internal Reviews Verification

## **QA Plan Deliverables & Checklist** **Generic Quality Checkpoints**

	<u>Date Completed</u>
Feasibility Analysis Report	
Business/User Requirements	
System Requirements Spec.	
System Design	
Conversion Plan	
Technical Test Plans	
Acceptance Test Plans	
User Documentation	
Operations Documentation	
Technical Testing Completion	
Production Turnover	
Acceptance Testing Sign-off	
Post-Implementation Review	

## **QA Plan Deliverables & Action Plan** **Generic Quality Checkpoints**

	<u>Applicable Standards</u>	<u>Resp</u>	<u>Budget Hours</u>	<u>Actual Hours</u>	<u>Date Due</u>	<u>Date Done</u>
Feasibility Analysis Report						
Business/User Requirements						
System Requirements Spec.						
System Design						
Conversion Plan						
Technical Test Plans						
Acceptance Test Plans						
User Documentation						
Operations Documentation						
Technical Testing Completion						
Production Turnover						
Acceptance Testing Sign-off						
Post-Implementation Review						



## QA Plan Deliverables, QA Action Plan Generic Quality Checkpoints

	Development				Quality Assurance Review				
	Std	Rsp	Hrs	Date	Resp	Budg	Act	Due	Done
Feasibility Analysis Report									
Business/User Requirements									
System Requirements Spec.									
System Design									
Conversion Plan									
Technical Test Plans									
Acceptance Test Plans									
User Documentation									
Operations Documentation									
Technical Testing Completion									
Production Turnover									
Acceptance Testing Sign-off									
Post-Implementation Review									



## Exercise: Managing SQA Tasks, Resources

*How would you handle and account for?*

***Development deliverable is delivered after SQA review was scheduled to begin***

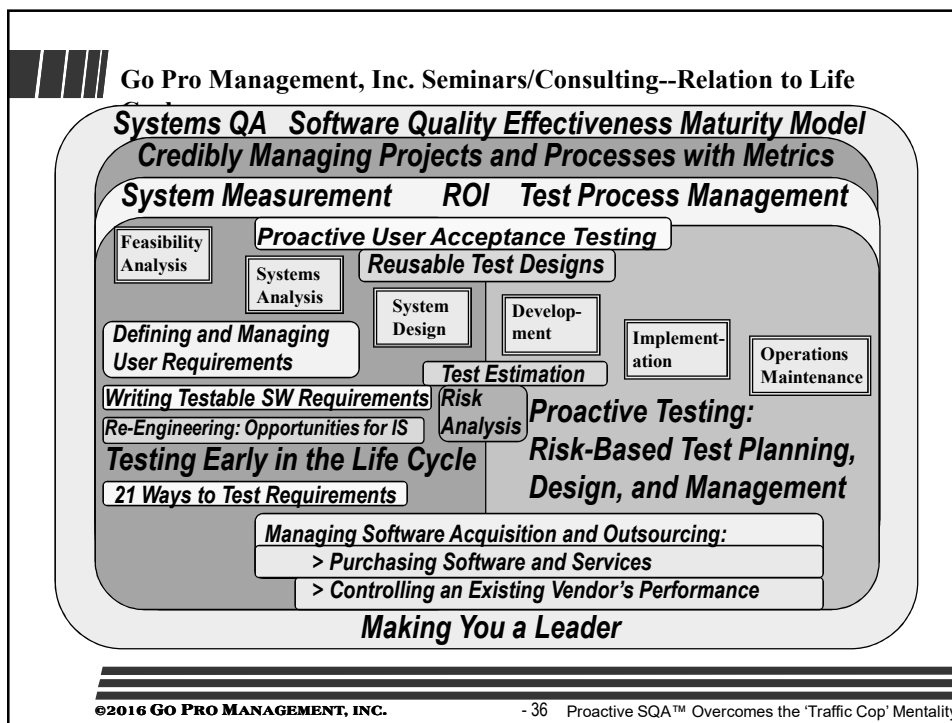
***SQA review finds a development deliverable inadequate and needs the deliverable to be corrected and re-reviewed***

***The SQA review takes longer and/or more effort than planned***

## Objectives

- Distinguish system/software quality, quality assurance (SQA), and quality control (SQC).
- Analyze conventional SQA/standards and why they so often are resisted, ignored, and/or fail.
- Describe the six functions Proactive SQA™ performs so
  - Involved parties understand and willingly participate in meaningful methods to assure software quality
  - Resisted practices are reduced, such as being a 'traffic cop'
  - Higher quality software truly is delivered quicker and cheaper.

***Proactive SQA™ is a key basis of significant value-enhancing revisions to IEEE SQA Std. 730-2014***





## Robin F. Goldsmith, JD

robin@gopromanagement.com (781) 444-5753 [www.gopromanagement.com](http://www.gopromanagement.com)

- President of Go Pro Management, Inc. consultancy since 1982, working directly with and training professionals in business engineering, requirements analysis, software acquisition, project management, quality and testing.
- Partner with ProvelT.net in REAL ROI™ and ROI Value Modeling™.
- Previously a developer, systems programmer/DBA/QA, and project leader with the City of Cleveland, leading financial institutions, and a "Big 4" consulting firm.
- Degrees: Kenyon College, A.B.; Pennsylvania State University, M.S. in Psychology; Suffolk University, J.D.; Boston University, LL.M. in Tax Law.
- Published author and frequent speaker at leading professional conferences.
- Formerly International Vice President of the Association for Systems Management and Executive Editor of the *Journal of Systems Management*.
- Founding Chairman of the New England Center for Organizational Effectiveness.
- Member of the Boston SPIN and SEPG'95 Planning and Program Committees.
- Chair of record-setting BOSCON 2000 and 2001, ASQ Boston Section's Annual Quality Conferences.
- TechTarget, SearchSoftwareQuality requirements and testing subject expert.
- Member IEEE Std. 829-2008 for Software Test Documentation Standard Revision Committee.
- Member IEEE 730-2014 Working Group rewriting IEEE Std. 730-2002 for Software Quality Assurance Plans.
- International Institute of Business Analysis (IIBA) Business Analysis Body of Knowledge (BABOK) subject expert.
- Admitted to the Massachusetts Bar and licensed to practice law in Massachusetts.
- Author of book: ***Discovering REAL Business Requirements for Software Project Success***
- Author of forthcoming book: ***Cut Creep—Put Business Back in Business Analysis to Discover REAL Business Requirements for Agile, ATDD, and Other Projects***

# Quality: 2020

## Brian Gaudreau, Consultant

Why this presentation? Quality in software delivery and sustainment will always have competing pressures between capability desired and the budget/time/quality delivered. As we approach the year 2020 technology landscape there are additional competing pressures we need to understand to be most effective with quality going forward. We will take a closer look and discuss these new challenges – for example:

- Client/Customer expectations and expected results will change and impact product roadmaps in new ways – what does this mean to the QA role?
- What does risk look like going forward?
- What is meaningful to measure?
- How can a Quality organization or methodology position itself to add even greater value?

Solution: This presentation will introduce new perspectives on how to plan and be ready for the future of software quality. Like how to expand DevOps culture with greater involvement by QA and adding customer/product owners to the model. In addition we will discuss how to identify, manage and loop data and metrics back into the requirements and continuously improve the client/customer experience. Takeaways will be:

- Diagrams and flows to help participants share the presentation topics with their peers and teams
- Steps to becoming quality initiators to the enterprise instead of maintaining the traditional quality assurance role
- A fresh perspective on test data management to ensure metrics and real time customer experience is measured in a meaningful way
- New ways of moving towards a DevOps culture where quality is a key driver

**Brian Gaudreau** is a seasoned quality professional, most recently working a Avanade a Group Manager – Cloud Solutions Quality. He brings to PNSQC over 20 years of software testing and quality assurance experience relevant to telecom, enterprise metadata, government, multimedia, health care and marketing CRM platforms. In addition to local involvement in software quality assurance interest groups, Brian's other interests include playing piano/synthesizers and supporting animal rescue shelters in the Pacific NW.

# Quality: 2020

Brian Gaudreau, [bg2066@outlook.com](mailto:bg2066@outlook.com)

## Abstract:

Quality in software delivery and sustainment will always have competing pressures between capability desired and the cost incurred, time taken and quality delivered.

As we approach the year 2020 the technology landscape will have additional competing pressures we need to understand to be most effective keeping quality high. For example:

- Client/Customer expectations will change and impact product roadmaps in new ways
- There will be new risks and assumptions to identify
- What is meaningful to measure in a Quality program will be more varied and dynamic than ever before
- Quality Organizations and Programs will need to re-assert their position to add even greater value

This paper will introduce new perspectives on how to plan and be ready for the future of software quality. Expanding DevOps culture with greater involvement by QA and adding customer/product owners to the model is one perspective. I will also highlight how to identify, manage and loop data and metrics back into the requirements and continuously improve the client/customer experience.

## Biography:

*Brian was most recently Group Manager of Quality at Avanade on the Cloud Solutions team. Brian brings to PNSQC over 20 years of software testing and quality assurance experience relevant to telecom, enterprise metadata, government, multimedia, health care and CRM platforms. In addition to local involvement in software quality assurance interest groups, Brian's other interests include playing live music and supporting animal rescue shelters in the Pacific NW.*

# Introduction

Whether it is the steadfast construction of an ocean vessel or producing enormous pyramids and temples, Quality has been part of our thinking and delivery for thousands of years. Software quality is relatively new adjective to our industrial journey. Moreover, most technologists would agree that the 1972 Trident submarine project involving over a million lines of code was the first time an iterative quality feedback approach was used in large scale software delivery. Software as a focus of quality and the rate which we need to iterate and change to be most effective is overwhelming and exciting at the same time.

As technology advances, ways in which requirements and business value are determined and measured also changes. For most people in developed nations acquiring specific goods or services relies on the Internet. It is easier than asking your neighbor or a good friend, and even more accessible wherever you are. The data points that allow a transaction acquiring items of value to be possible are increasing- even the way we seek and store information has changed.

## Strategy: Understanding Each Piece

Building the right thing and building it right the first time means you have to embed quality into every part of the software development life cycle. By integrating a quality focus from planning, requirements, design, release and beyond, testing can deliver more than simply quality assurance. The resulting framework delivers a high-value quality roadmap that will support and sustain the product or program.

Einstein's theory of relativity can help us put the new quality landscape into perspective:

*Two events, simultaneous for one observer, may not be simultaneous for another observer if the observers are in relative motion.*

*[Einstein, Albert (2009), Relativity - The Special and General Theory]*

What does this have to do with software quality? Cloud computing, Internet of Things (IoT) and connected Devices as a Service (CDaaS) are expanding technical possibilities similar to the big bang theory. Accelerating outward on multiple tangents and exponentially growing in size and impact with every moment. And these new components exist on top of the technology infrastructure already established. As professionals immersed in technology we may not be able to see how all these elements impact each other, but to add value and be successful with Quality going forward we must learn this skill.

To understand and implement a relevant quality management strategy with the future's ever growing scope, we should understand smaller pieces of the puzzle in more detail:

## Testing

The term "testing" (and with that Quality) is getting lost in today's framework and methodology. Creating test assets and then monitoring the health of those tests seems to be accepted by the majority nowadays as the best case scenario, but testing is so much more than that. Make sure Testing and Quality are defined and effective, even as they apply to nonfunctional requirements.

## Strategy

Don't always assume that when test or quality teams are invited to strategy or planning meetings, that is the time to tell everyone that a concept cannot succeed or has too many missing criteria to be sustainable. Even though the test and the quality effort may be a separate team, the effectiveness of our input may require follow up after team meetings.

## Areas of Quality & Test coverage

We enjoy looking at coverage models and pie graphs that remind us as quality professionals all the potential areas to prioritize and account for in our plan. Now it is becoming harder to segregate these coverage areas based on platform interdependencies. App testing and service testing need to be part of the same equation. Extensibility, Live Service Agreements and Capability are rising as new coverage areas that rank high in priority for planned coverage and reporting.

## People

People continue to be the glue that makes delivering products and solutions rewarding and scalable. More than ever we need to invest and train staff to be T-shaped in their thinking. A T shaped person has depth of related skills and expertise in a single role, while at the same time are able to collaborate across roles and platform experts and contribute in areas of expertise other than their own. The “T” represents this simultaneous depth and breadth concept with its perpendicular lines.

## Test Driven Development

Hours could be spent digging deeper into this one practice: the technique of having teams write tests that fail before new code is written. One key benefit is self-documenting the product or platform under test. Even with refactoring and working iteratively towards a correct final outcome, the resulting test suite will increase in value as a source of record. This allows our understanding of how Production currently works and enables planning for future capability.

## Data

Data is another area that is massive and a person could devote days to the ways Data and Quality intersect. More than ever, we need to be intentional about the ways data impacts our need to have the product or platform function as well as understanding how it is being used (versus how it was designed or planned for). There are four key data areas that need to be understood and planned for:

- **Determine and Classify.** It is amazing even in this day how often this step is glossed over for the mechanics of creating the data itself. Not only sources, types and variants but security policy and data flow need to have expectations set and understood.
- **Develop and Get.** Quantity of data, what qualifies for mock & stub, conversion and encryption strategies are all important. Don't forget researching the performance of extracts and acquisition even before the first validation run is done.
- **Populate and Use.** How loaded, who owns it and storage needs. So many people organizations miss the disconnect between identified data and dependencies to store, migrate and host data. Make sure you take a step back and ensure scenario E2E flows are represented.
- **Maintain.** Refresh, retention, baselining and reporting all are needed to maintain data quality. Without reusability there is very little value in test data management. Consider a self-service model where data can be refreshed on demand as well as how to measure effectiveness of data cleansing.

## Metrics

Metrics are one of the keys to evidence decision making, a core principal of Quality Management. We need to have the courage to re-examine metrics we are collecting and how applicable they are going

forward, even if we have developed months or years of trends based on the current criteria. Here are metrics I see needing the most attention for organizations to be ready for the Quality agenda:

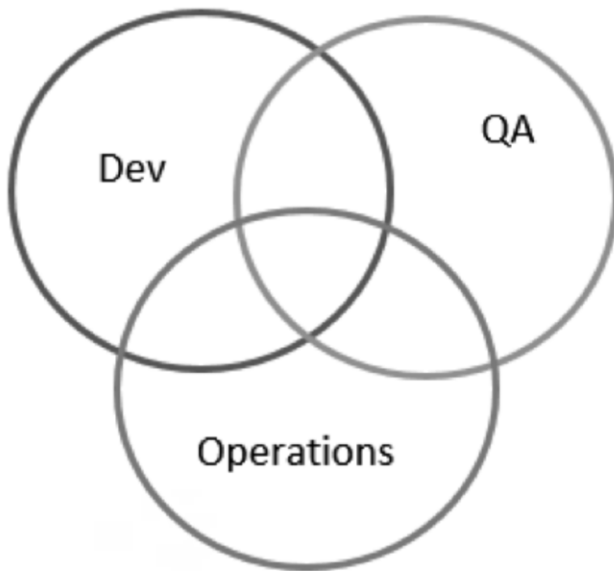
- Define a definition of done (“doneness” relative to the bigger picture). Even when using waterfall. Make sure this is understood, agreed to and relevant to the delivery model and number of concurrent teams developing features.
- Develop metrics that show the cost of test including rework, test data delivery and usage.
- Metrics that show Production incidents by feature as well as metrics that show effectiveness of delivery. And for added value, measure the manual effort for processes associated to features and delivery teams.

### **Tying It All Together:**

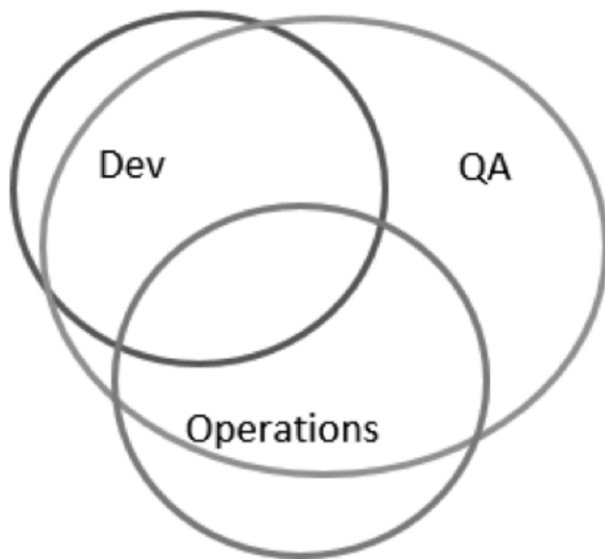
We should remind ourselves that clients and customers focus on the experience, not the features. Features in concert influence and drive a positive experience. The following strategies are key to having a successful quality management program in the future:

#### **Take DevOps to the next Level**

Most DevOps teams operate in a 3 circle model sharing Development, QA and Operations:



Having a data driven DevOps approach will result a larger Quality impact:



So how is this wider circle achieved? Developing quality heuristics that measure how the product or solution is *actually* being used in both preproduction and production can dramatically reduce critical defects and ongoing customer support issues surrounding usability and data compatibility. This evolution of DevOps culture allows more direct feedback to influence and understand what the solution and product is actually delivering (not just what the specifications say it should deliver)

### Test Data Management 2.0

Another strategy that needs attention is data management. The idea of Test Data Management (TDM) has been around for years and applied in many different ways. To be most effective in the future TDM will need to evolve even more.

The Quality Program of the future will need a matrixed perspective on the data it is using to create the maximum value for the effort:

PEOPLE – The user community and providers using the product or platform

TECHNOLOGY – The expectations and dependencies of the product or platform

PROCESS – Delivery and Service of the product or platform

By deliberately understanding production data delivery teams can spend less time justifying what data to use and more time validating a delivered solution. Dependencies within test and production environments become intentional and not blocking or reactive. Having the Quality program lead this effort reinforces the value of this discipline to the overall product or solution.

In future Quality programs data management metrics will distinguish departments that document defects as a reactive response and reinforce being a cost center ...from those that proactively detect flow/data errors and are a value add to the organization. We need to get Quality teams comfortable with exploring how customers are using the delivered product rather than measuring clients using the system only in the way we expected them to.

## Removing the Label of Quality as a Cost Center

Time to market expectations and delivery complexity is not making it easier to sustain a quality program. To be best prepared we need to assert the value of quality planning and tasks whether they are part of a standalone team or embedded into the development cycle. Here are four areas I believe are a solid start to meeting that goal:

- 1.Reduce handoff costs and measure effectiveness of continuous delivery. Having the process established is a great milestone and I do not want to minimize this, but measuring the effectiveness of this process takes value and efficiency to the next level
- 2.When the Quality program understands and proactively supports the product roadmap - more products/services will be sold. Use the strategy planning, data and metrics elements discussed earlier to allow execs and product owners to make informed prioritization choices that will position your features best in the marketplace
- 3.Modify attributes and codes so a Quality event gets traced back to providing value and not just absorbed into the Engineering delivery effort
- 4.There will always be bugs, people will make mistakes and providing solutions using cloud will introduce scaling and service level challenges never before experienced. Automating repeatable processes identified by a collaborative concert of Architects, Developers, Test and Operations resources will be of paramount importance going forward. As I discussed earlier, remember to track progress towards automating key manual tasks that remain necessary for deployment, support and extensibility

## Conclusion:

Whether we are delivering a fully developed system in a consulting model or our packaged software is playing a role in a larger platform ecosystem, we need to broaden our understanding of the impact a Quality program can have.

The very definition of quality is having a grade of excellence. The more distinctive and essential something is, the better able it is to satisfy stated or implied needs. Read that part again: *implied needs*. We are moving to an era when we will develop and implement an idea, test how the customers respond to using the idea based on initial conditions and then evaluate how close we were to building the right thing. Facebook and other service companies have been doing continuous delivery and deployment in Production for years – but the next years of delivering technology solutions are going to expand feasibility of who can use this approach and the speed in which results will be assessed and prioritized for delivery. Quality teams can either choose to embrace and be influencers of this new approach, or risk being seen as less agile with higher cost, but without matching benefits in return.

The value of this approach should be crystal clear by 2020.



# Risk Management in Software and Hardware Development

Christopher Alexander

chris@aglx.consulting

## Abstract

Quality in software and hardware development is increasingly proving a critical, if not decisive factor in delivering value, ensuring customer satisfaction, and increasing business longevity. Yet many technology companies lack a basic vocabulary and framework with which they can communicate, respond to, and mitigate the risks inherent in complex knowledge work. Especially in organizations employing agile practices, risks are introduced through development practices unfamiliar with risk management fundamentals, the results of which impact business in the form of escaped defects or errors (bugs) which slow internal production and affect end-customer utility and value.

Applying lessons learned from years of military service using Operational Risk Management (Office of the Chief of Naval Operations, 2010), we have developed a scalable framework providing a model for companies, teams, and individuals to apply to their risk management systems. Our approach focuses on prevention and detection while also addressing mitigation and response. We provide methods to continuously improve the system across multiple levels, increasing overall robustness, resiliency, and responsiveness.

Consideration is also applied to ways in which Risk Management can be used to improve not only the quality of the work itself, but also the processes and systems in which the work is being done, as well. In this way, Risk Management practices become active contributors to the continuous improvement cycle and can be used to drive change not only within software and hardware development, but across teams, departments, and workplaces to improve engineering and organizational practices.

## Biography

*Chris Alexander is a former Naval Officer who flew and instructed in the F-14 Tomcat and served abroad as a foreign policy desk officer for several European countries. He is also a web developer, agile coach, Scrum Master, and the co-founder of AGLX Consulting, where he co-developed High-Performance Teaming – a training methodology focused on teaching individuals and teams the social, interactive skills necessary to help them achieve high-performance. He currently works as an agile coach at Qumulo, Inc. in Seattle, Washington.*

# 1. Introduction

Risk Management has traditionally existed as a specialized domain of practice within leadership and project management circles. As a silo of concern, many other areas of business consider risk management to be “someone else’s problem,” if considered at all.

Yet in complex knowledge work such as software and hardware development, considerable risk exists at multiple levels throughout the system, and those most capable of detecting and preventing risks from becoming actualized have little to no part to play in risk prevention, analysis, and mitigation. What these individuals, teams, and the companies employing them lack is not a specific skill or ability available to only a select few, but rather a fundamental framework with which they can conceptualize and conduct risk management.

Given a central reference enabled by a common vocabulary and shared mental model (Mathieu and others, 2000), individuals and teams throughout an organization are transformed from passive, risk-unaware bystanders to active participants in risk detection, prevention, and mitigation. Additionally, they are able to assume active, intervening roles in improving the systems and strategies available to the company to manage and eliminate risk.

That last point – that Risk Management can be employed across co-located or distributed teams, departments, and entire enterprise organizations as a catalyst for organizational change – is the crux of this paper. Rather than relegating quality to a silo within a team, division, or department, individuals and teams can use risk management practices to improve the entire system within which they operate, across team and organizational boundaries.

## 2. Operational Risk Management

Risk Management in Software and Hardware Development is based on the application of Operational Risk Management (ORM) to companies developing software and hardware. Operational Risk Management is the name of the formalized process of risk management matured by the military and derived from routine human practices and habits.

Every day we awaken, review factors of our environment such as weather and traffic, and make decisions about how we plan to commute and what clothing to wear. We unconsciously make trade-offs every day of our lives between the things want to do (our goals), and what will be required to accomplish them, often weighted against the expected benefit or cost involved with our endeavor.

### 2.1 The Climber

As a practical example, imagine a rock climber. Rock climbing is a relatively risky endeavor. Yet statistically speaking a climber is much more likely to be injured or killed while driving to or from a crag or climbing gym than while actually climbing. When the climber awakens on a weekend morning, they review the weather, the anticipated climbing site, judge their personal desire to be climbing that day, and decide whether or not to face potential accidents, inclement weather, wild animals, etc., just to go rock climbing.

The risk and reward evaluative process designed to reach a decision about whether or not to undertake the activity in consideration is a risk management system. In the case of the climber, they are deciding how much risk they’re willing to accept in pursuing their goal of rock climbing.

Typically, for most of us, the impact and severity of risk in our daily activities is relatively low. In decent weather, when well rested, in a well-maintained car, the climber would accept the risks involved with driving to a climbing site. The training and climbing instruction they’ve undergone, combined with years of experience rock climbing, lowers their risk of being involved in an accident while climbing.

The scenario described above is the essence of risk management in an everyday life. In a business environment, the primary focus of risk management is analyzing and reducing or mitigating business risk. Business risk comes in a variety of shapes and sizes, and can affect individuals, teams, projects, programs, or the security, safety, and integrity of business data, and in many instances can threaten the survival of the business itself.

Despite the generally accepted importance of risk analysis and mitigation in business, many technology companies, departments, and teams view risk as a business problem requiring a business solution.

With the advent of agile frameworks and methodologies, and particularly in the rise of DevOps, risk can no longer be accepted as a purely business problem, addressed by business people.

Indeed, in many cases, risks not analyzed or mitigated by software and hardware development teams are indeed passed on to the business, but in an opaque, informal, and latent manner. When individuals and teams fail to appropriately analyze, plan for, and react to risks, those risks endure and can impact business operations. Such impacts are generally unnecessary, as mitigations often could have been applied to these risks well before their effects became realized.

### 3. The Swiss Cheese Model and Defense-in-Depth

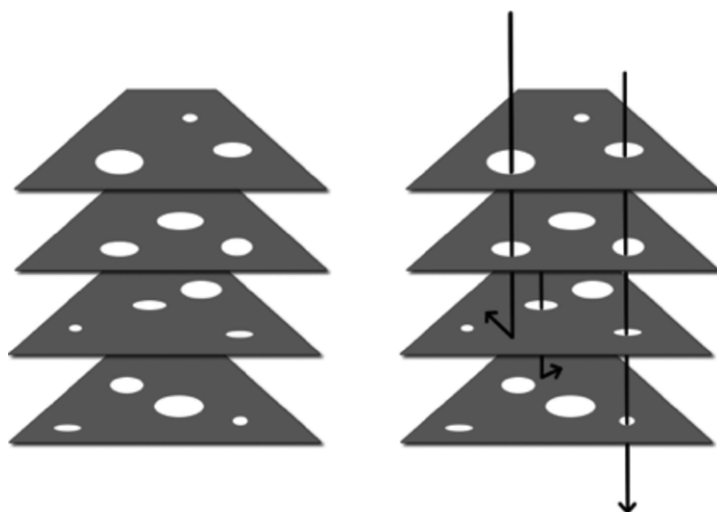
Any risk management system can be seen as a series of layers designed to employ a variety of means to stop errors from progressing further through or escaping from the system. This is known as error trapping. Each layer represents one slice of a larger system. The ability to trap errors with different layers in a system is known across industries as defense-in-depth.

Defense-in-depth reflects the simple idea that rather than employing a single, catch-all solution for trapping all errors and eliminating all risk, a layered approach which employs both latent and active methods for trapping errors at different points throughout the system will be far more effective in both detecting and preventing errors from escaping into production environments.

These layers are often envisioned as “slices” of Swiss cheese (Reason, 1990), with each slice representing a different part of the larger system of protection. As an error progresses through holes in those protective layers, it should eventually be caught by (unable to pass through) one of the layers. It is therefore only when “all the holes line up” that a bug “escapes” into production.

There are two basic classifications of traps (or layers) in any system: latent and active. Latent traps are those characteristics and features of existing systems and tools. Objective-C, for example, contains many built-in features for handling memory allocation, designed to trap many different types of potential memory errors which coding in traditional C occasionally produced. These memory management features exist already within the language and we use them almost literally without ever worrying about it.

Active traps, on the other hand, are active and purposeful measures taken to guard against errors in a proactive way. Running a set of manufacturing tests against a newly assembled product or running acceptance tests for a newly developed feature are examples of active traps. They are purposeful steps designed to find errors.



In daily life, latent traps are things such as the tires on your car or the surface of the road. All-weather, mud and snow tires are risk mitigation steps intended to help guard against the potential for slipping on wet or snowy roads and having an accident. Active traps, conversely, are things such as checking the weather, putting on safety gear, wearing a helmet, or deciding to mitigate the risks present by not driving in bad weather to begin with.

Latent traps in software and hardware development may be things such as the original (legacy) code base, the chassis fabrication line, development language(s), and system architecture and design. Active traps might include development practices like Test-Driven Development or Acceptance-Test-Driven Development, automated test suites, pair programming, code reviews, manufacturing tests, or acceptance verification tests.

### 3.1 A Fractal, Self-Similar Model

The risk management models which follow can be used by individuals, teams, and organizations, with adjustments and adaptations as necessary, to inform the way the entire company holistically thinks about and manages risk. The benefit of a self-similar, fractal system is that it builds cohesion, understanding, and alignment through the application of a shared mental model. Shared mental models have been instrumental in aligning organizations and teams, and are critical in improving quality and performance in any domain.

At individual levels, and to a lesser extent team levels, this model may be extremely abstract and much less formalized. However understanding its structure, components, and functioning enables the abstraction of key concepts for use in informal, yet still useful ways

### 3.2 Separation of Concerns in Defense-in-Depth Layers

To better focus on dealing with the most appropriate work at the appropriate time (in reflection of the agile principle of simplicity) in responding to error trapping, bug detection, triage, and risk mitigation, we separate our risk management concerns into the following areas:

---

#### *Software & hardware development: focus on trapping errors*

**Prevention** - the practices, procedures, and techniques we undertake to help ensure we do not release bugs into our codebase or introduce defects during assembly and build.

**Detection** - the methods available to us as individuals, teams, departments, and a company to find and deal with errors or bugs (which includes reporting and tracking).

---

#### *Production environments: focus on risk analysis and mitigation*

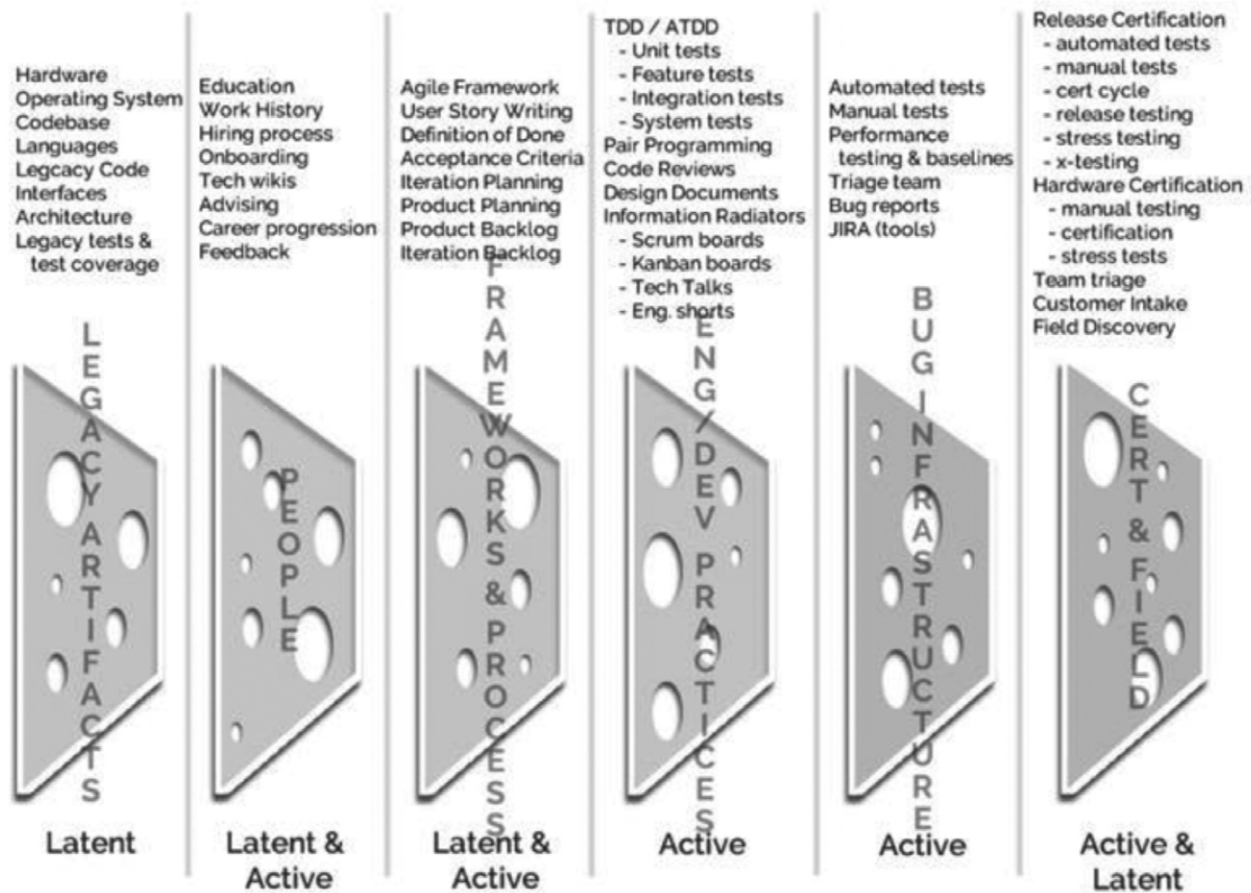
**Risk Analysis** - the steps required to analyze the severity and impact of an error or bug.

**Risk Decision-making** - the process for ensuring decisions about risk avoidance, acceptance, or mitigation are made at appropriate levels with full transparency.

---

#### *Continuous Improvement in every case*

**Improvement** - the process of improving workflows and practices through shared knowledge and experience supported by learning in order to improve development and production practices and further build resiliency and robustness. This step uses causal analysis to help close the metaphorical holes in our Swiss cheese model.



*Prevention and Detection* deals with those layers, systems, and practices designed to either prevent or find bugs in a system prior to their release into a production environment. Once a bug has been discovered in a production environment, the follow-on processes of *Risk Analysis* and *Risk Decision-making* occur. In either case, *6gb 5* completes the loop in the continuous improvement cycle by identifying root causes and lessons learned, which are used to feed improvements back into the system, increasing its robustness and resiliency.

## Error Causality, Detection & Prevention

Errors occurring during software and hardware development can be classified into two broad categories: skill-based errors, and knowledge-based errors.

### Skill-based errors.

Skill-based errors are those errors which emerge through the application of previously acquired knowledge and experience. They are differentiated from knowledge-based errors in that they arise not from a lack of knowing what to do, but instead from either misapplication of, or an overall failure to apply, what is known.

There are two types of skill-based errors: errors of commission, and errors of omission.

*Errors of commission* are the mis-application of a previously learned behavior (knowledge or experience). Returning to the climbing metaphor which opened this paper, if the climber tied the rope into their harness using an incorrect knot, they would be committing an error of commission. They know they need a knot, know which knot to use, and know how to tie the correct knot. They simply did not accomplish tying the knot correctly. In software development, one example of an error of commission might be an engineer providing the wrong variable to a function call, as in:

```
var x = 1;           // variable to call
var y = false;      // variable not to call
public function callVariable(x) {
    return x;
}
// whole bunch of code... code... code...
callVariable(y); // should have provided "x" but gave "y" instead
```

*Errors of omission* are the total failure to apply previously learned behaviors (knowledge and/or experience). Returning to our climbing metaphor, an experienced climber not tying the rope into their harness before beginning a climb is an example of an error of omission (and yes, this actually does happen). In software development, an example of an error of omission would be an engineer forgetting to provide a variable to a function call (or possibly forgetting to add the function call at all), as in:

```
var x = 1;           // variable to call
var y = false;      // variable not to call
public function callVariable(x) {
    return x;
}
// whole bunch of code... code... code...
callVariable(); // should have provided "x" but left empty
```

## 4.2 Knowledge-based errors

Knowledge-based errors, in contrast to skill-based errors, arise from the failure to know the correct behavior to apply (if any). An example of a knowledge-based error would be a developer checking in code but not having written or run any tests for that code. If the developer is new and has never been indoctrinated on the requirements for code check-in as including unit, integration, and system tests, this is an error caused by a lack of knowledge (as opposed to omission, where the developer had been informed of the need to build and run the tests, but failed to do so).

## 4.3 Preventing and Detecting Errors

*Prevention* comprises the layers, systems, tools, and processes employed to trap bugs and stop them from escaping development environments and entering into build, staging, testing, certification, or eventually production environments. In envisioning our Swiss cheese model, we need to understand that the layers include both latent and active types of error traps, and are designed to mitigate against certain types of errors.

The following are examples of some of the methods, tools, and processes available to aid in preventing bugs.

*To mitigate against Skill-based errors in bug prevention:*

- Code base and architecture [latent]
- Automated test coverage [active]
- Manual test coverage [active]
- Unit, feature, integration, system, and story tests [active]
- TDD / ATDD practices [active]
- Code reviews [active]
- Pair Programming [active]
- Performance testing [active]
- Software development framework / methodology (ie, Scrum, Kanban, DevOps, etc.) [latent]

*To mitigate against Knowledge-based errors in bug prevention:*

- Employee education & background [latent]
- Recruiting and hiring practices [active]
- Employee Onboarding [active]
- Performance feedback & professional development [active]
- Design documents [active]
- Definition of Done [active]
- User Story Acceptance Criteria [active]
- Code reviews [active]
- Pair Programming [active]
- Information Radiators [latent]

*Detection* is the term for the ways in which we find bugs, hopefully in the development environment but this phase also includes continuous integration, testing, and/or certification stages as well. The primary focus of detection methods is to ensure no bugs escape into production. As such, the entire software integration, deployment, and testing (or certification) system itself may be considered one, large, active layer of error trapping. In fact, in many enterprise companies, the certification team is literally the *last line of defense*.

The following are examples of some of the methods, tools, and processes available to aid in detecting bugs:

*To mitigate against Skill-based errors in detecting bugs:*

- Automated test coverage [active]
- Manual test coverage [active]
- Unit, feature, integration, system, and story tests [active]
- TDD / ATDD practices [active]
- Release integration / certification testing [active]
- Performance testing [active]
- User Story Acceptance Criteria [active]
- User Story “Done” Criteria [active]
- Bug tracking software [active]
- Triage reports [active]

*To mitigate against Knowledge-based errors in detecting bugs:*

- Employee education & background [latent]
- Hiring discipline and practices [active]
- Employee Onboarding [active]
- Continuous Improvement activities [active]

## 5. Risk Analysis

Should existing preventative measures fail to stop a bug from escaping our development environment and entering into a production system, an analysis of the level of risk needs to be explicitly completed. This

analysis is almost always accomplished, but in an implicit way. The analysis of the level of risk derives from two areas (note that the severity and probability metrics introduced below are in no way meant to be proscriptive, but rather to serve as examples from which to build or deviate).

## 5.1 Risk Severity

Risk Severity is the degree of impact the bug can be expected to have to the data, operations, or functionality of affected entities (the company, vendors, clients, customers, etc.).

<b>Blocker</b>	A bug the effects of which are so bad, or a feature that is so important, that we would not ship the next release or additional units until it is remedied. Could also signify a bug that is currently blocking customer workflows, or one that is blocking our own internal submits/builds, etc.
<b>Critical</b>	A bug that needs to be resolved ASAP, but for which we would not hold a release or execute a special release. Depending on its probability, a team might need to be interrupted during their work to remedy the error.
<b>Major</b>	Best judgment should be used to determine the priority of this issue against other work in a Sprint. It needs to be resolved, but is not a “drop everything and fix this” issue. If a bug sits in major for too long (more than a Sprint or a few weeks), it should be upgraded to critical or downgraded to Minor.
<b>Minor</b>	A bug that is known, but that has been explicitly deprioritized. Often, these bugs should be either addressed as part of a Sprint or downgraded to Trivial.
<b>Trivial</b>	We should consider closing this. At best these should be put into a backlog for tracking and remedy as work capacity becomes available.

## 5.2 Risk Probability

Risk Probability is the anticipated percentage of all customers potentially affected by the bug who will likely experience it (for example: always, only if they have a power outage, most likely not, etc.).

<b>Definite</b>	100% - <i>issue will occur in every case</i>
<b>Probable</b>	60-99% - <i>issue will occur in most cases</i>
<b>Possible</b>	30-60% - <i>coin-flip; issue may or may not occur</i>
<b>Unlikely</b>	2-30% - <i>issue will occur in less than 50% of cases</i>
<b>Won't</b>	1% - <i>occurrence of the issue will be exceptionally rare</i>

## 5.3 Risk Assessment Code

Given Risk Severity and Probability, the risk can be assessed according to a risk assessment matrix (example below), and assigned the corollary Risk Assessment Code (RAC).

The Risk Assessment Codes are a significant factor in Risk decision-making. Based on your organization's culture and sensitivity to risk, which may need to account for things such as regulatory environment, compliance and reporting, sensitivity of customer data, reputation, etc., your decision-making on risk may need to be significantly altered from the examples here.

Risk Assessment Matrix		Probability				
		<i>Definite</i>	<i>Probable</i>	<i>Possible</i>	<i>Unlikely</i>	<i>Won't</i>
Severity	<i>Blocker</i>	1	1	1	2	3
	<i>Critical</i>	1	1	2	2	3
	<i>Major</i>	2	2	2	3	4
	<i>Minor</i>	3	3	3	4	5
	<i>Trivial</i>	3	4	4	5	5
<b>Risk Assessment Codes</b> 1 - Strategic    2 - Significant    3 - Moderate    4 - Low    5 - Negligible						

The example here is intended to inform discussion and act as a guide, not to prescribe levels or constrain a company's need to adopt and adapt scales more responsive to its needs.

**Strategic** - the risk to the business is significant enough that its realization could threaten business operations, basic functioning, regulatory / compliance requirements, and/or professional reputation to the point that the basic survival of the business could be in jeopardy.

**Significant** - the risk poses considerable, but not life-threatening, challenges for the business. If left unchecked, these risks may elevate to strategic levels.

**Moderate** - the risk to business operations, continuity, and/or reputation is significant enough to warrant consideration against other business priorities and issues, but not significant enough to trigger higher responses.

**Low** - the risk to the business is not significant enough to warrant special consideration of the risk against other priorities. Issues should be dealt with in routine, predictable, and business-as-usual ways.

**Negligible** - the risk to the business is not significant enough to warrant further consideration.

## 5.4 The Risk Decision

The risk decision is the point at which a decision is made regarding actions to take or not take based on the risk. Typically, risk decisions take the form of:

**Accept** - accept the risk as it is and do not mitigate or take additional steps.

**Delay** - for less critical issues or dependencies, a decision about whether to accept or mitigate a risk may be delayed until additional information, research, or steps are completed.

**Mitigate** - establish a mitigation strategy and deal with the risk.

For risk mitigation, feasible courses of action (COAs) should be developed to assist in formulating the mitigation plan. These potential actions comprise the mitigation and or reaction plan. Specifically, given risk severity, probability, and the resulting RAC, the courses of action are the possible mitigate solutions for the risk. Examples include:

### *Pre-release*

- Apply software fix / patch
- Code refactor
- Code rewrite
- Release without the code integrated (re-build)
- Hold the release and await code fix
- Cancel the release

### *In production*

- Add to normal backlog and prioritize with normal workflow
- Pull / create a team to triage and fix
- Swarm multiple teams on fix
- Pull back / recall release
- Release an additional fix as a micro-release

Mitigation strategies at the corporate level may also need to include deliverables from and outputs for multiple departments, such as marketing, PR, operations, IT, and legal. For serious issues a marketing and PR campaign may be required to mitigate impacts from a production error on the company's reputation. For legal and compliance related issues, self-reporting is often a swift path to avoiding investigations, penalties, and fines.

For all risk decisions, those decisions should be shared with stakeholders, recorded, and for those which remain active, tracked. There are many methods available for logging and tracking risk decisions, from simple spreadsheets to entire software platforms expressly designed to track and monitor risk status and decisions.

Decisions to delay risk mitigations are the most important to track, as they typically require future action, and at the speed most business move today, another risk exists in potentially losing track of risk-delayed decisions. Therefore a Risk Log or Review should be used to routinely review the status of pending risk decisions and reevaluate them, not to mention using simple calendar reminders or other "low-tech" methods to keep the delayed decision visible.

Risk changes constantly, and risks may significantly change in severity and probability as quickly as overnight. In reviewing risk decisions regularly, individuals, teams, and leaders are able to simultaneously ensure both that emerging risks are mitigated and that effort is not wasted unnecessarily.

## **6. Process Improvement**

Once a bug has been discovered and risk analysis / decision-making has been completed, a retrospective-style analysis on the circumstances surrounding the development or engineering practices which failed to effectively trap the bug completes the cycle.

The purpose of the retrospective or debrief is not to assign blame or find fault, but rather to understand the cause of the failure to trap the bug, inspect the layers of the system, and determine if any additional tools, methods, procedures, or process changes could effectively improve collective practice and help to prevent future bugs.

### **6.1 Methodology**

**Review** sequence of events that led to the anomaly / bug.

Determine **root cause**.

**Map** the root cause against the layer model.

**Decide** if there are remediation efforts or improvements which would be effective in supporting or restructuring the system to increase its effectiveness at error trapping.

**Implement** any changes identified, sharing them publicly to ensure everyone understands the changes and the reasoning behind them.

**Monitor** the changes, adjusting as necessary.

### 6.1.1 Review sequence of events

With appropriate representatives from engineering teams, hardware, operations, customer support, etc., review the discovery path which found the bug. The point is to understand the processes used, which layers or guards worked, and which ones let the error pass through.

### 6.1.2 Determine *root cause* and analyze the optimum layers for improvement

What caused the bug? There are many enablers and contributing factors, but typically only one or two root causes. The root cause is one or a possible combination of Organization, Communication, Knowledge, Experience, Discipline, Teamwork, or Leadership.

**Organization** - typically latent, organizational root causes include things like existing processes, tools, practices, habits, customs, culture, etc. These are factors of the company or organization as a whole.

**Communication** - a failure to convey necessary, important, or vital information to an individual or team who required it for the successful accomplishment of their work.

**Knowledge** - an individual, team, or organization did not possess the knowledge necessary to succeed. This is the root cause for knowledge-based errors.

**Experience** - an individual, team, or organization did not possess the experience necessary to accomplish a task (as opposed to the knowledge about what to do). Experience is often a root cause in skill-based errors of omission.

**Discipline** - an individual, team, or organization did not possess the discipline necessary to apply their knowledge and experience to solving a problem. Discipline is often a root cause in skill-based errors of commission.

**Teamwork** - individuals, possibly at multiple levels, failed to work together as a team. Additional root causes may be knowledge, experience, communication, or discipline.

**Leadership** - less often seen at smaller organizations, a Leadership failure is typically a root cause when a leader and/or manager has not effectively communicated expectations or empowered team execution regarding those expectations. Leadership has, either directly or indirectly, hindered the individuals', teams', or organization's ability to deliver effectively.

### 6.1.3 Map the root cause to the layer(s) which should have trapped the error.

Given the root cause analysis, determine where in the system (which layer or layers) the bug / error should have or could have been trapped. The best location to identify is the one which most closely corresponds to the root cause of the bug and which occurs earliest in the system or process flow.

The earlier an error can be prevented or caught, the less costly it is in terms of both time (to find, fix, and eliminate the bug) and effort (a bug in production requires more effort from more people than a developer discovering a bug while checking their own unit test).

While we should seek to apply fixes at the locations best suited for them, the earliest point at which a bug could have been caught and prevented will often be the *optimum place* to improve the system.

For example, if a bug was traced back to a team's discipline in writing and using tests (root cause: discipline and experience), then it would map to layers dealing with testing practices (TDD/ATDD), pair programming, acceptance criteria, User Story writing, definition of "Done," etc. Those layers where the team can most readily apply improvements and which will trap the error sooner, should be the focus for improvement efforts.

#### **6.1.4 *Decide on improvements to increase system effectiveness.***

Based on the knowledge gained through analyzing and mapping the root cause, decisions are made on how to improve the effectiveness of the system at the layers identified. Using the testing example above, a team could decide that they need to adjust their definition of Done to include listing which tests a story has been tested against and their pass/fail conditions.

#### **6.1.5 *Implement the changes identified.***

Once changes have been identified, those changes must be clearly communicated at appropriate levels so that everyone involved understands their individual and collective responsibilities in implementing those changes. This is also a good place to share lessons learned about the process and the improvements discovered so that everyone may benefit from the shared knowledge.

#### **6.1.6 *Monitor the changes and adjust as necessary.***

In an agile environment, where teams are able to continuously learn and adapt, as new insights or options become available to either reinforce or replace existing aspects of a defense-in-depth system, the effectiveness of methods and layers should be evaluated against new ideas and improvements to continue building the optimum configuration for error trapping. The goal is to build and maintain a robust and resilient system.

## **7 Application of Risk Management in Planning**

When undertaking backlog grooming, Sprint planning, or any type of work breakdown activity, we tend to estimate work items / User Stories in terms of the size and amount of work required to complete them, often including consideration of the degree of complexity (unknowns) inherent in the work. Some items are low in complexity and tend to be estimated lower, while others are higher in complexity and tend to be estimated somewhat larger.

Once the backlog of work has been broken down and estimated according to effort and complexity, the work is prioritized according to business value (which is typically, but not always, the value of the work to potential or actual customers). In many instances, this is where estimation and prioritization ceases and the most valuable (highest priority) items are pulled to kick-off work.

This is a mistake in most environments, however. The backlog needs to also be analyzed for risk, risks need to be estimated, and the backlog needs to be prioritized again to account for both risk and value.

For example, the physical assembly and testing of a new hardware platform may be a relatively low-value story or work item, however the risk of not being able to deliver the platform in a sufficient amount of time due to incomplete testing, coupled with the risk that the build and shipment of the platform's physical chassis itself requires three months, may mean that this work item becomes the highest priority item, today.

Without adjusting our estimation and prioritization for risk, we may miss important dependencies, durations, challenges, opportunities, etc. which could be easily mitigated or capitalized upon if recognized early enough, helping to avert a potential disaster later in our product development or release.

## 7.1 Adjusting and prioritizing for Risk

With the product or work backlog now estimated and prioritized according to value, the team looks again at each item and evaluates the risk(s) inherent to it. Complexity is a risk, for example, present in many larger software development User Stories and relates directly to the amount of unknown (unknowable) work involved in completing the story.

Other types of risk might be dependencies (either internal or external), critical components, availability of hardware/infrastructure/etc., outcomes of experiments, etc. Such risks need be evaluated for their degree of impact and likelihood of occurrence, just as business risk analysis was described previously.

Again, as an example, an external dependency (vendor) on a critical component (chassis) which will entirely block future work must be mitigated by prioritizing the work item appropriately in order to ensure lead time is met. If we are at the start of an estimated four month project and we have a two month wait time for a vendor delivery of a critical component, we need to prioritize this work item now.

Having reviewed backlog or work items, we are now aware of risks inherent in the backlog, and can reprioritize it again, adjusting the balance of work to account for both value and risk. This new prioritization also has the added benefit of potentially revealing risks which we cannot mitigate or overcome through prioritization of work alone - we may need help from sources external to the team or we may need to add additional risk-mitigating work to the backlog / work inventory.

These discussions regarding how to think about and prioritize risk may seem pedantic and overly process-focused, even intuitive, but if such is the case then you are among the blessed minority of teams who have a good methodology for risk management. The depressing reality is that the overwhelming majority of teams have no framework, process, tool, or skillset to help them analyze and manage risk in any explicit or useful way.

Through a mindful analysis of potential risks and a purposeful strategy to manage them, teams can improve their chances of successful outcomes as they build and deliver products and features, in any domain.

## 8. Summary

Individuals, teams, and companies engaged in software and hardware development have typically regarded risk in indirect, assumptive ways. In increasingly competitive environments in which quality and development time can prove to be critical market differentiators, companies can no longer afford to view risk as a project- or management-level factor. Risk management needs to be understood and employed at every level within an organization to achieve effectiveness.

Risk Management is an organic activity which we undertake every day of our lives in some form. As such, leveraging our innate ability to analyze risk and make risk-based decisions, our strength in building robust systems lies in extricating our intrinsic risk management knowledge and formalizing it into a purposeful and conscious system.

In this paper we have provided a framework for individuals, teams, and organizations to conceptualize, analyze, and evaluate risk, based on a common vocabulary and using shared mental models. These structures (common vocabulary and shared mental models) have been proven to be instrumental in ensuring alignment and effectiveness in teams and organizations (DeChurch and Mesmer-Magnus, 2010; Mathieu and others, 2000).

Understanding how a company's processes and practices form a Defense-in-Depth system for containing errors enables the system to be purposefully improved over time based on lessons learned from root cause analysis of defect or error occurrences. The improvement of multiple layers of the system creates

a more robust and resilient system, in turn increasing the capacity for higher levels of quality and productivity.

Every organization operates some form of this Defense-in-Depth system, which can be conceptualized as layers of Swiss cheese. Understanding that the majority of preventable errors fall into one of several categories enables teams and individuals to root cause those errors and apply improvements to the system, optimally in multiple places, thus increasing the organization's robustness and resilience.

In cases of escaped defects, risk decisions must occasionally be taken and should be based on a clear understanding of the risk's severity and probability of occurrence. In these instances, a risk decision log should be created, updated, and reviewed.

Individuals, teams, and companies should not seek to apply this framework in a prescriptive manner, but rather adapt them to fit the unique circumstances and requirements of their situation.

## References

Office of the Chief of Naval Operations. "OPNAV Instruction 3500.39C, Operational Risk Management, (dated 02 Jul 2010)." United States Navy. <http://www.public.navy.mil/airfor/nalo/Documents/SAFETY/OPNAVINST%203500.39C%20OPERATIONAL%20RISK%20MANEGEMENT.pdf> (accessed July 12, 2016).

Reason, James. 1990. "The Contribution of Latent Human Failures to the Breakdown of Complex Systems." *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 327 (1241): 475–84.

DeChurch, Leslie A. and Jessica R. Mesmer-Magnus. 2010. "Measuring Shared Team Mental Models: A Meta-Analysis." *Group Dynamics: Theory, Research, and Practice* 14 (1): 1-14.

Mathieu, John E and others. 2000. "The Influence of Shared Mental Models on Team Process and Performance." *Journal of Applied Psychology* 85 (2): 273-283.

# Third-party library mismanagement: How it can derail your plans

Ruchir Garg

Ruchir.Garg@Intel.com

## Abstract

Third-party libraries are synonymous with most software applications as they enable faster development. While third-party libraries provide value, but if not managed properly, they could later become a major source of project delays or unhappy customers. In this paper, we discuss multiple suggestions that can help alleviate issues arising from third-party library mismanagement. Adopting these suggestions will help you mitigate these risks to a large extent, while allowing to you build a more predictable release strategy.

## Biography

*Ruchir Garg is a Security Architect at Intel Security, where he is working on securing enterprise class applications. He has also worked as an automation architect, developing brand new test automation frameworks. Ruchir has spent more than 14 years of industry experience developing, testing and supporting software products in various domains like embedded, mobile, wireless and desktop applications.*

*Ruchir holds a degree in Electrical Engineering from Nagpur University, India.*

# 1 Introduction

Most decent sized software products use third-party libraries for faster development. The topic we'd discuss here is something that is usually given lesser importance than it deserves i.e. the management of third-party libraries. We describe processes to minimize the negative impact of using third-party libraries to your release plans. We identified and adopted best-practices that helped us manage third-party libraries in a more predictive manner. We also observed that adopting these best-practices reduced the number of hotfixes we shipped.

In this paper, we'll discuss various issues like managing different library versions, its compatibility– API or platform support, maintenance, and security vulnerability management.

## 2 Problem/Opportunity

Along with the obvious benefits, using third-party libraries also bring along its own share of possible challenges and complexities. When third-party libraries are not managed in an organized manner, developers use and include numerous external libraries, without giving attention to their maintenance.

This may lead to a situation where the products used by the customers include obsolete and possibly, vulnerable libraries. If a vulnerability introduced by an installed product gets exploited, not just Intellectual Property (IP) is at risk, but the organization that owns the vulnerable product may even face a litigation.

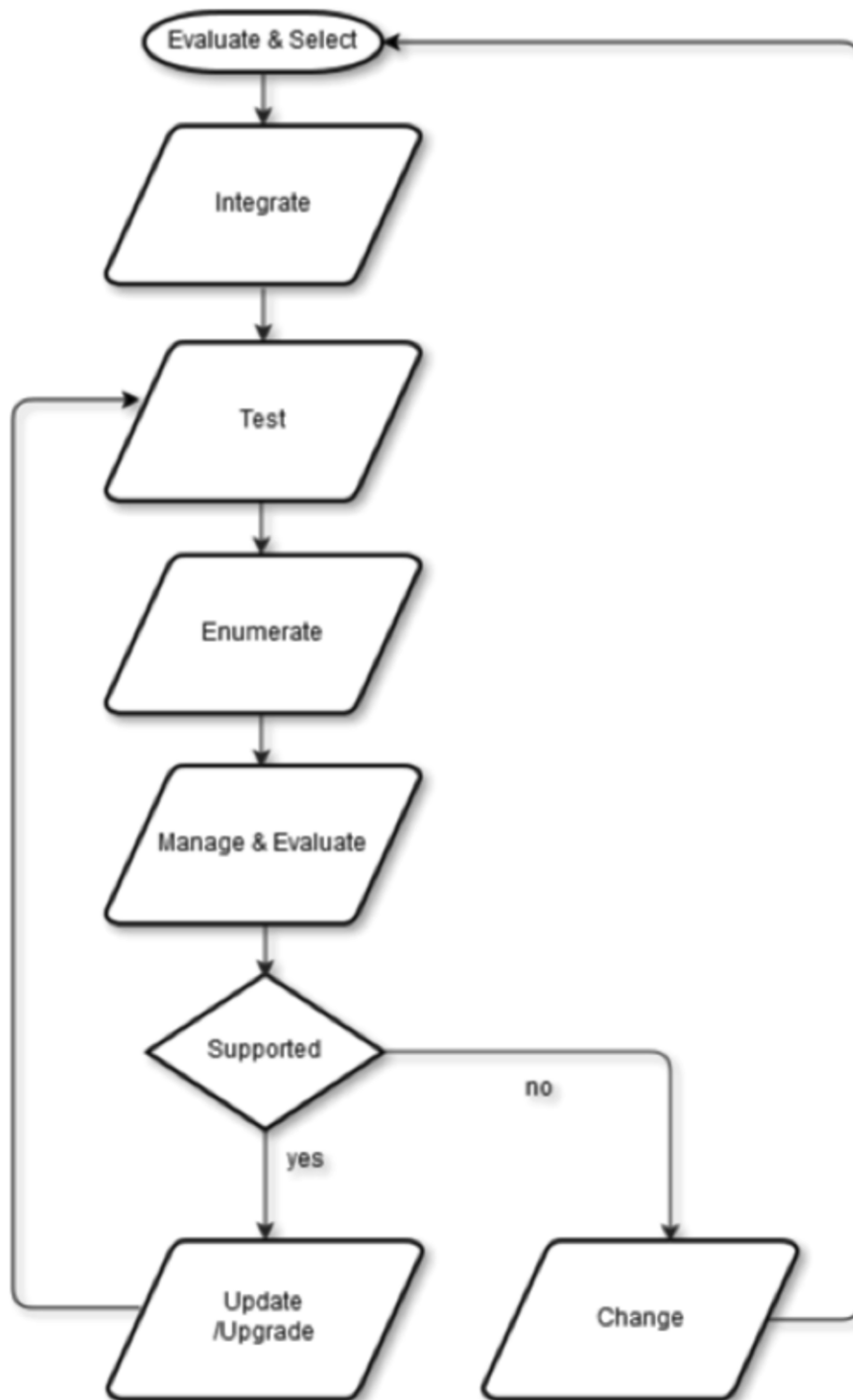
These challenges may create unwarranted situations where the ongoing product releases get affected. For example, consider a situation where a security researcher discloses a critical vulnerability in a third-party library version that you are using and your customers are demanding an immediate fix within a week. Now when you try to upgrade to the newest version of the library, which has the fix, but you realize that an important OS platform support has been dropped. This may affect a large customer base and definitely not a good news for the organization. Worse, if you lost track, the library might have gone end-of-support and the fix for the vulnerability is simply not available. This is a nightmare scenarios for any organization.

Project delays can hurt any organization, especially when faster delivery to market is the norm and completion is breathing down the neck. Delays are inherent to most software projects, but a delay due to external dependencies is something on which we've least amount of control, and has to be managed very efficiently.

The problem described here will resonate with all software professionals, who are dealing with third-party libraries in their code, as they are challenged with these issues every day. Participants can expect to learn from the best practices described in this paper and apply them to minimize the unexpected delays to their releases.

### 3 Solution

The diagram <sup>1</sup>given below represents a typical lifecycle of a third-party library:



<sup>1</sup> Figure 1:- Lifecycle of a third-party library

### 3.1 Libraries enumeration

The recommended process starts with building an active list of all such libraries, including the version used and the latest version available in the field. Keep an eye on the date when the library was last updated.

### 3.2 Vulnerability tracking

Actively search and track all security vulnerabilities published against the third-party libraries used in the project. A simple search in the [National Vulnerability Database<sup>2</sup>](#) or tracking the product specific security bulletins will help. This should give you a good indication of your current position. In an ideal scenario, a library should have vulnerabilities listed against it and those should have been fixed too. Finding no vulnerabilities either means not many are using it, or no one has actively researched it yet. What it definitely does **not** indicate that it's a secure library.

### 3.3 Keeping things updated

Keeping all libraries up-to-date at frequent intervals will help in eliminating the risk of a major version upgrade on a short notice. A major version upgrade may possibly introduce a risk of API changes or platform deprecation.

A frequency of once every quarter to review and update the libraries is reasonable. Any greater frequency could be an overhead and a lesser frequency poses a risk of taking in too many changes together, which may increase the quality risks. Retaining source and packages of all versions of the libraries used in your source control repository is also good practice, which helps in debugging the field issues.

### 3.4 Library selection

Selection process of new libraries is a big deal as various factors that need to be carefully looked into. Factors like their active maintenance schedules, platform support, number of vulnerabilities reported and fixed, EOS schedule, and third-party libraries (if any) used within. We can get the list of any third-party used within by looking up its licensing documentation.

Organization must setup a whitelisting team, which evaluates third-party libraries based on these factors and then recommends them for general use by all teams. This will ensure uniformity and compliance.

I'd reiterate that having no reported vulnerability for a library is not a sign of a secure product.

### 3.5 Support considerations

There could be libraries in your product that are either reaching their End-Of-Support (EOS) or are already EOS. It's strongly recommended to migrate away from such libraries to their alternatives. As such libraries will not receive any security fixes, continuing to use them pose a danger to our customers and also to our organization's reputation.

Keep an eye on the extended support fine-print to ensure that extended support is not limited to critical issues or technical assistance only, but not fixes.

---

<sup>2</sup> National Vulnerability Database is a good resource to search vulnerabilities in software: <https://web.nvd.nist.gov/view/vuln/search>

### 3.6 Quality matters

We can ensure the quality of our code by employing rigorous validation, static analysis, vulnerability scanning, fuzzing and penetration testing. However, we have no control over the quality processes followed during the development of the third-party libraries we consume. Such libraries could be either open-source or closed source, but we should definitely apply all **applicable** quality checks described above to ensure that the libraries meet the minimum quality criteria. We must always remember that a chain is only as strong as its weakest link.

### 3.7 Management commitment

The factor of management support cannot be overlooked, if we've to make any of these changes discussed above. All these process and technical changes require time and investments, before they start to show the results. Hence, it's important that management invests in this strategy by allowing additional time in the projects for the reduction of technical debt, and prioritizes relevant stories from the backlog to enable library upgrades.

Libraries that are feature-rich and are also maintained efficiently often come at a cost, so the management must be prepared to pay for it as its going to be more cost-effective in the long run compared to a cheap, but poorly written library. Using open-source is always an option, which has its own pros and cons, but that's another discussion, outside the scope of this paper. Even for open-source libraries, all these learnings would certainly apply.

## 4 Results

Teams generally follow their own processes of managing third-party libraries. We also had our own process that we followed, but frequently ran into issues with that. Adopting these best-practices reduced the probability of manifestation of risks associated with external library usage to a large extent, if not completely. We don't claim that our process is the best, but we can definitely claim that it works.

We've seen that following these best practices minimized the risk of running into project risks, if not to totally avoid them. Moving away from current practices to the ones described here is not an easy task and will require significant investments. My team has realized that adopting these best-practices saved us from the problems of unplanned security hotfixes and sudden platform support deprecation. It also means that our existing projects will no longer suffer from unwanted interruptions and we are better compliant with security incident response SLAs. We can now afford to have a more predictable release cycles. This is good news for our support team and our customers, which will eventually means a higher Net Promoter Score (NPS)<sup>3</sup>.

There are software tools available in the field that could be used for automatic patch and update management, but those are limited to only well-known applications. Until a generic product becomes available that can automate this process, library management process as described in this paper is a possible solution.

---

<sup>3</sup> Net Promoter Score is a customer loyalty metric: [https://en.wikipedia.org/wiki/Net\\_Promoter](https://en.wikipedia.org/wiki/Net_Promoter)

## 5 Final words

The changes suggested in this paper are both technical and process related. Developers need to work closely with the security architects while selecting external libraries, existing libraries need to be evaluated and risk need to be addressed. Product Managers also need to prioritize such tasks so that it gets into the release. It's a shift in the mindset and all stakeholders must support to get it addressed. Finally, this template once applied to all products in the organization will ensure process uniformity and compliance, bring efficiency into project releases, and increase customer confidence into your products.

# Innersource in Test Automation

James Knowlton

jknowlton@navexglobal.com

## Abstract

In implementing test automation at your organization...does any of this sound familiar?

- Organizational silos, each implementing their own automation framework and processes (or not implementing automation at all)
- “Cathedral”-based automation framework, developed by one or two engineers, understood only by them
- Highly customized framework, which requires in-house training and support

This is the situation we encountered when I was hired as QA Architect at NAVEX Global. NAVEX is an enterprise ethics and compliance software vendor based in Lake Oswego, OR. They were assembled as a company via a series of acquisitions, so they are geographically diverse, with a wide disparity in technical and development ability/experience. As will be discussed, we had an automation framework build by one person (who then left the company) which was highly customized. This made it difficult to train and support QA Engineers, and also to implement improvements.

How we addressed the challenge of implementing a common automation framework is the subject of this paper. The paper will also discuss the results we were able to produce, and next steps in our process.

## Biography

*James Knowlton is a QA Architect at NAVEX Global, where he oversees quality assurance processes including defining standards, methodologies, procedures and metrics used across NAVEX Global's engineering organization. His previous experience includes stints at Symantec, McAfee, ADP and eBay. He has presented on software quality at multiple conferences and is the author of the book “Python: Create, Modify, Reuse” (Wrox, 2008).*

Copyright James Knowlton 2016

# 1 Introduction

## 1.1 Description of NAVEX Global

NAVEX Global is a developer of cloud-based ethics and compliance software. They offer hotline reporting, case management software, policy compliance software, vendor management software, online compliance training, and other tools. The company headquarters are in Lake Oswego, Oregon, with several offices in the US and an office in London, England.

### 1.1.1 History of NAVEX

- July 2012: The Riverside Group, a private equity firm, purchases EthicsPoint and merges it with Global Compliance (a Charlotte, North Carolina based competitor). The new company will be called NAVEX Global.
- June 2012: NAVEX Global acquires PolicyTech, a Rexburg, Idaho based policy compliance software provider.
- October 2014: Vista Equity Partners acquires NAVEX Global from the Riverside Group.
- August 2015: NAVEX Global acquires The Network, an Atlanta, Georgia based competitor, doubling the size of the company (to about 1000 employees).

Bottom line: there's been a lot of churn.

## 1.2 Overview of the QA team

NAVEX had introduced a company-wide rollout of Scrum/agile about a year before I arrived. It had varying levels of adoption and support. One of the challenges generally was that since the company had been cobbled together by combining several different companies, there were significant variations in culture and experience.

### 1.2.1 Engineering team structure

At NAVEX, the engineering teams (about 10-20 engineers per team) were vertically aligned, with the following:

- Product owner - the engineering leader (usually a manager/director) who is responsible for the overall success of the product from an engineering perspective. Some of the product owners came from an engineering background, but some actually came from project management, so they lacked the engineering background to provide guidance, especially concerning QA processes.
- DevOps engineers – this position existed for some teams. When present, the DevOps engineers managed the engineering infrastructure, including the build system and QA servers.
- Development engineers
- QA engineers

### 1.2.2. QA teams' knowledge/experience

There was significant variation in knowledge and experience concerning QA engineers. Some of the teams had a good mix of very experienced QA engineers with some engineers who were less

experienced in testing but who had significant coding/development expertise. However, other teams had QA engineers who knew the product but were lacking not only coding knowledge (for automation), but even general technical expertise. The fact that teams were geographically dispersed made training these people difficult.

## **2 Challenges**

### **2.1 Prior automation framework**

#### **2.1.1 Overview of pre-existing automation**

I did not actually work with the pre-existing framework, but my understanding is that there were many limiting factors. It was developed by a single QA Engineer. It used significant customization to try to make it as easy as possible for people to write tests (“push button automation”). It was used in some teams, but other teams did their own thing for automation (for example, one team had a framework based on Java/Selenium).

#### **2.1.2 Problems with pre-existing framework**

- Because only one person maintained the framework:
  - Other engineers were prevented from contributing, thus preventing them from enhancing their morale and sense of empowerment.
  - There was a bottleneck when it came to suggesting or implementing improvements.
- Because the framework was highly customized:
  - The framework was very brittle, and not very adaptable to unforeseen conditions.
  - There were no public training materials available. This was especially problematic due to the geographic dispersal of the company and the lack of experience of some engineers (see 1.2.2 above).
- The framework was developed in Ruby and Watir (a Ruby-based web automation framework). Since Ruby wasn’t actually used anywhere else in the company, there was no help available from development engineers.
- The one person who developed it...quit. The company was left with no one to maintain the framework.

### **2.2 Review of the challenge**

We had a QA organization that:

- Was dispersed all over the company (making training more of a challenge)
- Had significant variation in the level of QA and coding knowledge/experience
- Had management (sometimes) that was willing but not able to provide experienced technical direction

We had a framework that was:

- In a language not widely used at the company

FLOWED IN KEEPING FORMATTING FROM (WORD, THEN RICH, AND THEN GENTLY) broken)

Not understood enough to apply some of the new styles I made (bt indented). But you can search for Arial italic and apply the Arial italic character style.

- Not widely used, so public training was not available

Also can delete some of the imported “styles” and replace with the styles I created. Also replaced “emphasis” character style with Arial italic style...

Any grays will need to be replaced with real gray (they’re rgb).

## 3 What is Innersource?

### 3.1 Definition

Danese Cooper of PayPal defines Innersource as “The application of best practices, processes, culture and methodologies taken from the open source world and applied to internal software development and innovation efforts”.

In other words, it consists of using the benefits of open source (collaboration, collective code ownership, documentation of code reviews and checkins) in the context of an organization’s internal projects.

### 3.2 Open source principles

- Engineers share their work with a wide audience, instead of just with a manager or team
- Everyone can contribute to the project
- Distributed version control is used (such as GitHub) so that collaboration and code review can happen more easily
- New code repositories (branches) can be made freely
- People at large geographical distances, at separate times, can work on the same code or contribute different files of code to the same project.
- Communication tends to be written and posted to public sites instead of shared informally by word of mouth.

### 3.3 Applied internally at an organization

- Innersource differs from classic open source by remaining within the view and control of a single organization.
- The “openness” of the project extends across many teams within the organization. Ideally, code is shared freely within the entire organization.

### 3.4 Benefits of Innersource

- Code reuse across the organization grows immensely
- Cross-team collaboration becomes relatively frictionless
- Written comments exchanged among team members, although taking up some time, more than pay for themselves by helping new engineers learn the system faster.

- Engineers learn to document their code better, both formally (as in-code comments and documentation) and informally (on discussion lists and chat channels).

## 4 How we applied Innersource to the NAVEX Global challenge

### 4.1 Project Overview – NOVA Framework

#### 4.1.1 What is it?

The NOVA Framework is a test automation framework for UI-based testing of web applications. It is Selenium-based and makes use of Page Object Model, a Selenium design pattern designed to encourage code maintainability.

*Nova* is a word form which (in various languages and spellings) means “new”. So, we felt it would be a good name for our new framework.

It makes use of C#/.NET, Selenium, and NUnit.

#### 4.1.2 Who created it?

It was initially developed by a partnership of development and QA engineers on the Ethicspoint Incident Manager product, after watching a training video on Page Object Model. After I was brought in as QA Architect, I designed the Innersource process to develop and improve NOVA companywide.

### 4.2 NOVA framework architecture

Component	Owned by *	Composed of	Use
<b>Tests</b>	Product team	Ummm...tests	Implementation of NOVA framework/product framework
<b>Product framework</b>	Product team	<ul style="list-style-type: none"> <li>• Page objects for application under test</li> <li>• Helpers/code specific to application</li> </ul>	Inherits from (and makes use of) NOVA framework
<b>NOVA framework</b>	Engineering org	<ul style="list-style-type: none"> <li>• Common objects and helper methods</li> <li>• Product-independent</li> </ul>	Foundation layer

\* Development & QA

The intention is that just as with an open source project, individual product teams can code their own solutions in the product framework as they see fit, and if they have something they think can be generally useful, they can submit it for consideration in the NOVA framework

### 4.3 NOVA project team organization

Role	Membership	Details
<b>Architect/Project Lead</b>	QA Architect	<ul style="list-style-type: none"><li>• Provide overall direction and leadership</li><li>• Chair of NOVA steering committee</li></ul>
<b>Steering Committee</b>	<ul style="list-style-type: none"><li>• 3-4 people</li><li>• Dev &amp; QA</li><li>• Multiple products</li></ul>	<ul style="list-style-type: none"><li>• Plan and direct NOVA releases</li><li>• Groom NOVA backlog regularly</li><li>• Communicate with engineers on NOVA tasks assigned</li></ul>
<b>Engineering cross-product team</b>	Engineering org	<ul style="list-style-type: none"><li>• Anyone interested in contributing</li><li>• Sign up for stories per sprint, based on interest and availability</li></ul>

#### 4.3.1 The Steering Committee

The steering committee is represented by as many cross sections as possible in the engineering organization. For example, there were both dev & QA and also multiple products (and geographical sites) represented.

### 4.4 NOVA schedule

NOVA was implemented and maintained in a fairly typical Scrum schedule. One difference is the team did not engage in story pointing or specific estimating. They basically just signed up for stories until it felt like a good amount of work for the duration of the sprint. If we ran out of time for a story, it just got moved to the next sprint.

#### 4.4.1 Releases

- Every six weeks
- In sync with product release schedule
- Release is comprised of two 3-week sprints (also in sync with product schedule)
- Planning day is held on a Wednesday (to prevent conflict with product planning, which is on Mondays)

#### 4.4.2 Sprint planning

- Everyone interested is invited
- We do the following:
  - Complete current sprint, including reviewing work completed and moving in-progress work to the next sprint
  - Pull tasks from the backlog until we have what we feel is a good amount of work for three weeks
- It's "low-maintenance" planning (no time estimates, story points, etc.)
- When someone signs up for a story/task, they clone it to their product's JIRA project
  - This is so the product team can track their work

- When they complete the task/story, they close it in both projects

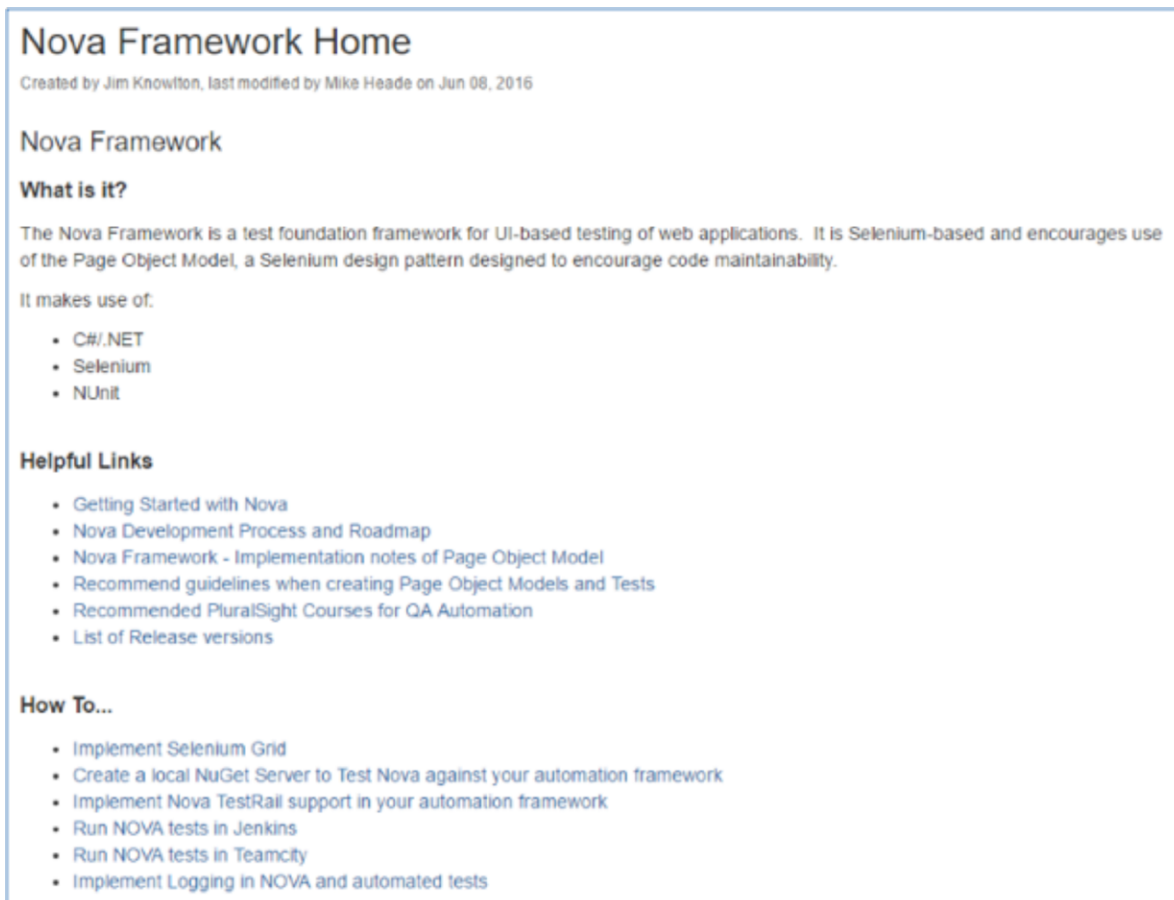
#### 4.4.3 Backlog grooming

- Responsibility of the steering committee
- Performed weekly

### 4.5 NOVA training/documentation

#### 4.5.1 Documentation

- Extensive how-to documents built in Confluence – example:



The screenshot shows a Confluence page titled "Nova Framework Home". Below the title, it says "Created by Jim Knowlton, last modified by Mike Heade on Jun 08, 2016". The main heading is "Nova Framework". Underneath, there is a section "What is it?" which describes the Nova Framework as a test foundation framework for UI-based testing of web applications, Selenium-based, and encourages the use of the Page Object Model. It lists technologies it makes use of: C#.NET, Selenium, and NUnit. There is a "Helpful Links" section with links to getting started, development process, implementation notes, guidelines, PluralSight courses, and release versions. Finally, there is a "How To..." section with links to implementing Selenium Grid, creating a local NuGet server, implementing TestRail support, running tests in Jenkins and Teamcity, and implementing logging.

**Nova Framework Home**  
Created by Jim Knowlton, last modified by Mike Heade on Jun 08, 2016

## Nova Framework

### What is it?

The Nova Framework is a test foundation framework for UI-based testing of web applications. It is Selenium-based and encourages use of the Page Object Model, a Selenium design pattern designed to encourage code maintainability.

It makes use of:

- C#.NET
- Selenium
- NUnit

### Helpful Links

- [Getting Started with Nova](#)
- [Nova Development Process and Roadmap](#)
- [Nova Framework - Implementation notes of Page Object Model](#)
- [Recommend guidelines when creating Page Object Models and Tests](#)
- [Recommended PluralSight Courses for QA Automation](#)
- [List of Release versions](#)

### How To...

- [Implement Selenium Grid](#)
- [Create a local NuGet Server to Test Nova against your automation framework](#)
- [Implement Nova TestRail support in your automation framework](#)
- [Run NOVA tests in Jenkins](#)
- [Run NOVA tests in Teamcity](#)
- [Implement Logging in NOVA and automated tests](#)

- API documentation automatically generated (via Doxygen) with every build

## 5 Benefits and Lessons Learned

### 5.1 Benefits

We saw many benefits to our implementation:

- Collective code ownership spread knowledge (and accountability) across the organization
- Using a standard framework/design pattern (Selenium, Page Object) allowed us to more easily build and collaborate across the organization.
- This also made training much easier, as there are significant publicly available training materials on Selenium and Page Object Model.

### 5.2 Lessons learned

We learned several things as we used the open source model to implement our automation:

- We found constant collaboration to be critical both in troubleshooting issues and training people on the framework. As NAVEX uses Slack, we built Slack channels specifically for NOVA collaboration. This was a great help, especially as Slack has great integrations for GitHub.
- As each team was different in its ability to jump into implementing NOVA, we were very flexible in terms of implementation schedule. This allowed teams to plan for training, transitions from other frameworks, etc.
- One thing we intentionally planned for from the beginning is that we wanted the framework to be lightweight: the less there was in it, the less to go wrong. We communicated from the beginning that if a product team wanted some functionality that wasn't in NOVA, and we didn't feel we should add it, they were always welcome to add whatever functionality they wanted to their own product framework.

## 6 Conclusion

Although not a “one size fits all” solution, our experience was that Innersource can be a very effective way to collaboratively manage development of a test framework inside an organization.

## References

Web Sites:

1. Portland Business Journal. “EthicsPoint acquired, to merge with rivals”. <http://www.bizjournals.com/portland/news/2012/02/01/ethicspoint-acquired-to-merge-with.html> (accessed June 20, 2016)
2. Oregonlive. “EthicsPoint changes its name to NAVEX Global, buys Idaho-based PolicyTech”. [http://www.oregonlive.com/silicon-forest/index.ssf/2012/06/ethicspoint\\_changes\\_its\\_name\\_t.html](http://www.oregonlive.com/silicon-forest/index.ssf/2012/06/ethicspoint_changes_its_name_t.html) (accessed June 20, 2016)
3. Danese Cooper, PayPal, “Getting Started in Innersource”, (<https://www.youtube.com/watch?v=r4QU1WJn9f8> ), 2015.

# Defect Prediction Model for estimating Project Schedule

Rajesh Yawantikar, Debashish Barua, Anupriya Vatta

{rajesh.yawantikar, debashish.barua, anupriya.vatta}@intel.com

## Abstract

Many organizations want to estimate the number of defects prior to a product's launch in order to estimate the product quality and maintenance effort. Another serious concern for the Project management is about defining the project milestone schedules. Some of problems that needs to be addressed are: What if Alpha, Beta, and Production Release milestones are planned way too early or too late? How does the number of open defects impact the program schedule? Are there any existing tools which can aid for scheduling project milestones (Alpha, Beta, and PRQ i.e. Product Release Qualification)?

This paper addresses the above problems and we demonstrate this by applying our Defect Prediction Model in estimating project milestones. We'll take a closer look at identifying the common mistakes made by the Project Management in declaring the milestones. Using this model, we can identify the risk in the existing program schedule. Additionally, we can monitor in-process quality of products and take preventive actions on the unforeseen defects/issues/risks. We have successfully applied our model to approximately 50 different platforms across different business segments and as a result taken preventive measures in upcoming projects and releasing the products on time.

## Biography

*Rajesh Yawantikar is Senior Software Quality Manager at Intel Corporation, he is responsible for all aspects of software quality and reliability across mobility products this includes tablets and phones. With over all experience of 19+ years in software Quality, of which 10 years with Intel. Rajesh won 5 Intel Software Quality Awards (highest award in Software at Intel), now he mentors several teams for Intel Software Quality Award. Rajesh leads the external industry collaborations for Intel Software Quality group, this includes working with various industry bodies and academia. Rajesh is Board member of IEEE Computer Society- Professional Education and Activities Board.*

*Debashish Barua is a Software Quality Engineer at Intel. Debashish has a Master's degree in Electrical and Computer Engineering from University of California, Santa Barbara. He is well versed with Statistical Analysis and Machine Learning and has over 4 years of work experience in the software industry.*

*Anupriya Vatta joined Intel as Software Quality Engineer 3 years ago with 5+ years of QA and automation experience, including companies like Blackberry, Ream Networks and Good etc. Anupriya has a Master's degree in Computer Science from University of Texas at Dallas. As a SQE she is responsible for software initiatives and process improvements and helps multiple teams to adhere the best software practices. She is also always looking for innovative ideas and methodologies to improve the overall software lifecycle.*

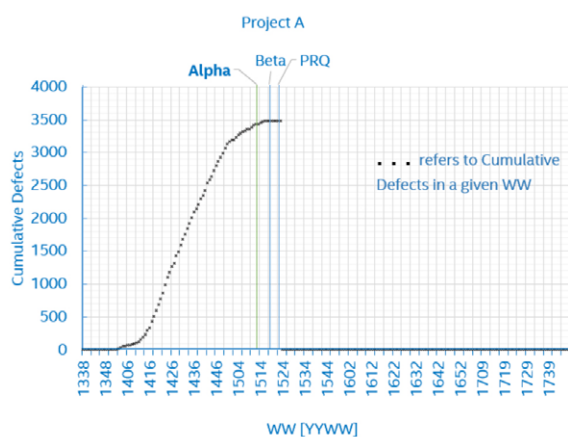
# 1 Introduction

On one hand, we have to predict the number of Software Defects in an upcoming project. On the other hand, we need to schedule Project Milestones in such a way that the number of Defects are reduced with subsequent releases and thereby improving the quality of the product. There are many papers which predict Software defects using statistical models and metrics. The paper by Fenton and Neil (Fenton & Neil, 1999) critiqued on a wide range of Prediction Models. It covered complexity and size metrics, the testing process, the design and development process, and multivariate studies. Milestones are used in project management to mark specific points along a project timeline. These points may signal anchors such as a project start and end date, a need for external review or input and budget checks, among others. Project milestones focus on major progress points in the Product Life Cycle (PLC). We want to tackle these problems by correlating the number of Defects to the Project Milestones.

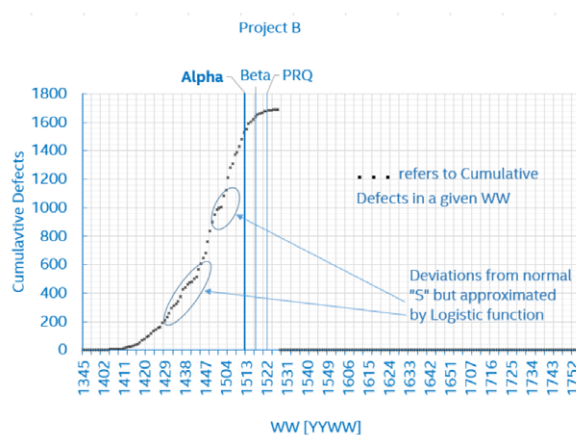
With this in mind, we developed a Defect Prediction Model which integrates the Project Milestones and provides defects trends along with improved schedule for upcoming projects. In this paper, we cover Building a Prediction Model (Section 2), Defect Prediction Model using Logistic function (Section 3), Project Milestones (Section 4), Generate Reference Parameters from Past Projects (Section 5), and Predicting Defects and defining Project Milestones(Section 6).

## 2 Building a Prediction Model

Our main objective is to build a model which is simple, easy to understand, and can be readily used by the Program / Project Managers. We have a huge database of defects containing historical data of previous projects. In our projects, we have used defect tracking systems like Jira (Jira Software, n.d.), Bugzilla (Bugzilla, n.d.), etc. to collect the defect data. At first, we plotted all the defects found in a project for the entire product lifecycle. These plots were random in nature, with many crests and troughs in the curve for different projects, making it difficult to perform analysis. In our next iteration, we chose to plot the Cumulative Defects instead of standalone ones. We observed a steady increasing trend for all the projects. In our third and final iteration, we chose to aggregate all the defects within a Work Week (5 business days from Monday to Friday) and plot the Cumulative Defects with respect to a Work Week (WW).



**Figure 1:** Plot of Cumulative Defects Vs WW for Project A



**Figure 2:** Plot of Cumulative Defects Vs WW for Project B

We observe an “S” shaped pattern for both the projects A and B. We decided to group the days into Work Weeks for the following reasons:

1. Agile based projects have typically 1 or 2 WWs as Sprints. Each sprint starts with a sprint planning event that aims to define a sprint backlog, identify the work for the sprint, and make an estimated

commitment for the sprint goal.

2. Project Managers plan with respect to WWs. All the milestones are also defined in WW terminology.

Figure 1 and Figure 2 trends show that they follow the “S” trend. Figure 1 almost looks perfect whereas Figure 2 deviates a little but overall we can make a good assumption for an “S” curve. Also, we have highlighted the specific project milestones (Alpha, Beta, and PRQ) which will be discussed in detail in Project Milestones (Section 4). By carefully analyzing the defect trends of approximately 50 projects, we conclude that the projects when plotted by Cumulative Defects Vs WWs resemble the slanted version of the letter “S”.

## 2.1 The significance of S-curve in Project Management

Wideman (Wideman, May 2001) defined S-curve as follows: “A display of cumulative costs, labor hours or other quantities plotted against time. The name derives from the S-like shape of the curve, flatter at the beginning and end and steeper in the middle, which is typical of most projects. The beginning represents a slow, deliberate but accelerating start, while the end represents a deceleration as the work runs out.”

Garland's (Garland, The Mysterious S-curve, 2009) article about the mysterious S-curves says, “S-curves are an important project management tool. They allow the progress of a project to be tracked visually over time, and form a historical record of what has happened till date. Analyses of S-curves allow project managers to quickly identify project growth, slippage, and potential problems that could adversely impact the project schedule if no remedial action is taken.” Figure 3, Figure 4, and Figure 5 are from (Garland, The Mysterious S-curve, 2009).

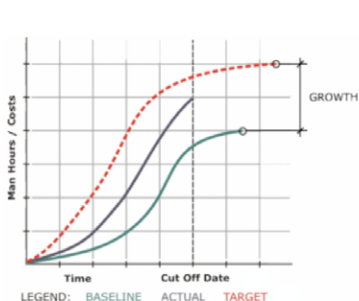


Figure 3: Project Growth

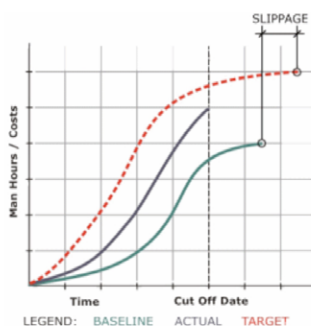


Figure 4: Project Slippage

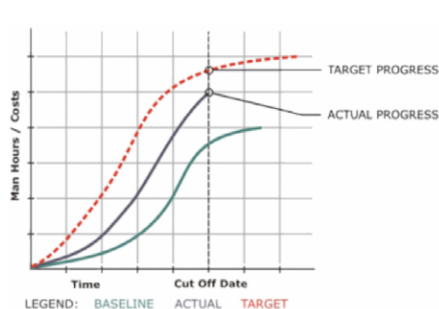


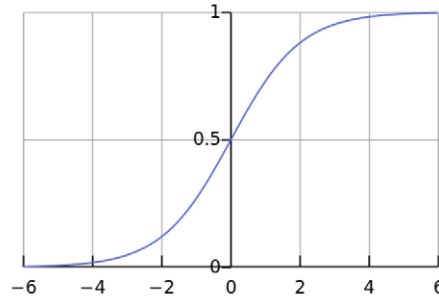
Figure 5: Project Progress

## 2.2 Basic S-curve Mathematical Formula (Sigmoidal function)

The simplest S-curve is a Sigmoidal function (Sigmoid function, 2016) which is defined by the following expression:

where,

- $e$  = the natural logarithm base (also known as Euler's number)



**Figure 6:** Sigmoid Function (Sigmoid function, 2016)

We notice that, the Sigmoid curve ranges between 0 and 1 with the corresponding x values as  $-\infty$  and  $+\infty$  respectively. Also, the mid-point is 0.5 at  $x = 0$ .

Even though the basic S-curve is good representation of our model, we cannot use it for following reasons:

1. As we are bounded between 0 and 1.
2. The midpoint is fixed and occurs at  $x = 0$ .
3. The steepness of the curve is constant which needs to be adjustable.

This can be solved by using the generic version of the sigmoidal function i.e. the Logistic function.

### 3 Defect Prediction Model using Logistic function

A logistic function (Logistic function, 2016) or logistic curve is a common “S” shape (sigmoid curve), with equation:

where,

- $e$  = the natural logarithm base (also known as Euler’s number),
- $x_0$  = the x-value of the sigmoid’s midpoint,
- $L$  = the curve’s maximum value, and
- $k$  = the steepness of the curve

Now that we are at a liberty to change the curve’s maximum, midpoint, and the steepness, we can use the Logistic function to model our Defect trends.

We are applying the Logistic 3P model (Logistic regression) using the JMP® Pro 11.1.1 (64-bit) software (JMP Pro, n.d.). The following equation best describes our Model which is exactly the same as the above logistic function with different naming convention.

where,

- $e$  = the natural logarithm base (also known as Euler’s number),
- Inflection Point = the Work Week at which the Cumulative Defects is exactly half of the total,
- Asymptote = the total Cumulative Defects, and
- Growth Rate = the rate at which the Defects grow with respect to Work Week

Let's apply Logistic regression to our projects mentioned earlier, Project A (Table 1) and B (Table 2), and calculate the parameters (Growth Rate, Inflection Point, Asymptote) Table 3 & Table 4. (Logistic 3P - Fit Curve Options, n.d.)

Our Defect Data consists of 3 columns: Week #, WW, and Cumulative Defects.

1. Week # is used for referencing the sequential order of Work Week (WW). The Inflection Point is one of these values.
2. WW is the actual Work Week
3. Cumulative Defects is the aggregated total number of defects as of that WW.

An example with reference to Table 1:

1. Week # 1 corresponds to WW 38 of Year 2013 with '1' as Cumulative Defects which implies that we found only '1' defect in WW 38.
2. Week # 2 corresponds to WW 39 of Year 2013 with '1' as Cumulative Defects. This implies that we found '0' defects on WW 39 and the '1' is carry forwarded from WW 38.
3. Week # 31 corresponds to WW 16 of Year 2014 with '335' as Cumulative Defects. This implies that on WW 16 we found 43 more defects as of WW 15, where the Cumulated Defects were 292.

**Table 1: Cumulative Defects Vs WW for Project A**

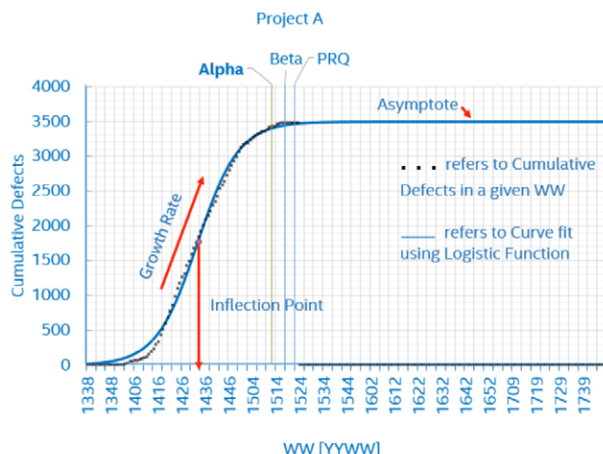
Week #	WW	Cumulative Defects
1	201338	1
2	201339	1
3	201340	1
4	201341	3
5	201342	3
6	201343	3
7	201344	3
8	201345	4
9	201346	5
10	201347	5
...	...	...
30	201415	292
31	201416	335
32	201417	432
...	...	...
50	201435	2005
51	201436	2100
52	201437	2153
...	...	...
89	201522	3484
90	201523	3486

**Table 2: Cumulative Defects Vs WW for Project B**

Week #	WW	Cumulative Defects
1	201345	1
2	201346	1
3	201347	1
4	201348	1
5	201349	3
6	201350	4
7	201351	4
8	201352	4
9	201353	4
10	201402	4
...	...	...
30	201422	101
31	201423	115
32	201424	135
...	...	...
50	201442	501
51	201443	516
52	201444	565
...	...	...
86	201526	1689
87	201527	1690

**Table 3: Generated Parameters for Project A**

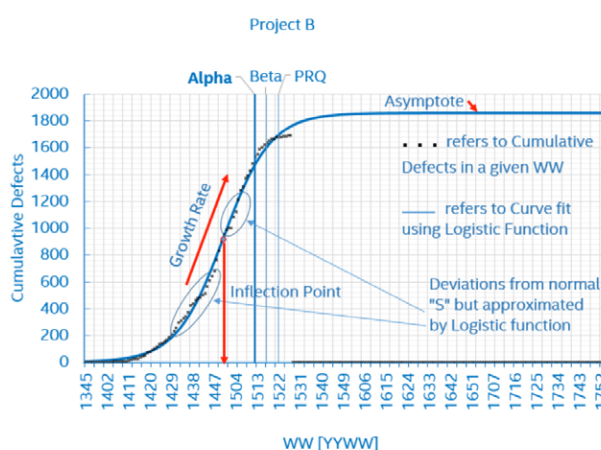
Growth Rate	Inflection Point	Asymptote
0.11	48	3492



**Figure 7: Curve fit response for Project A**

**Table 4: Generated Parameters for Project B**

Growth Rate	Inflection Point	Asymptote
0.10	59	1856



**Figure 8: Curve fit response for Project B**

### 3.1 Accuracy of the Logistic model

Now that we are using the Logistic model, the question that arises is how much do we deviate with the curve and what's the error margin. We show our results in Table 5 by estimating the R-Square value (Fit Curve Report, n.d.).

**Table 5: Summary of fit showing R-Square values**

Summary of fit	Project A	Project B
R-Square	0.9972143	0.9973997

#### 3.1.1 R-Square

Estimates the proportion of variation in the response that can be attributed to the model rather than to random error. R2 is calculated as follows:

where,

- 'Sum of Squares (Cumulative Total)' is the sum of squared distances of each response from the sample mean.
- 'Sum of Squares (Error)' is the sum of squared differences between the fitted values and the actual values. This corresponds to the unexplained variability after fitting the model.
- 'Sum of Squares (Model)' is the difference between the Sum of Squares (Cumulative Total) and the Sum of Squares (Error).

An  $R^2$  closer to 1 indicates a better fit. An  $R^2$  closer to 0 indicates that the fit predicts the response no better than the overall response mean. On an average, we have an R-square of 0.997 accuracy, which tells that the Logistic model is almost a perfect fit.

### 3.1.2 Comparison of Sigmoid functions

Besides the logistic function, sigmoid functions include the ordinary arctangent, the hyperbolic tangent, the Gudermannian function, and the error function, but also the generalized logistic function, and algebraic functions like

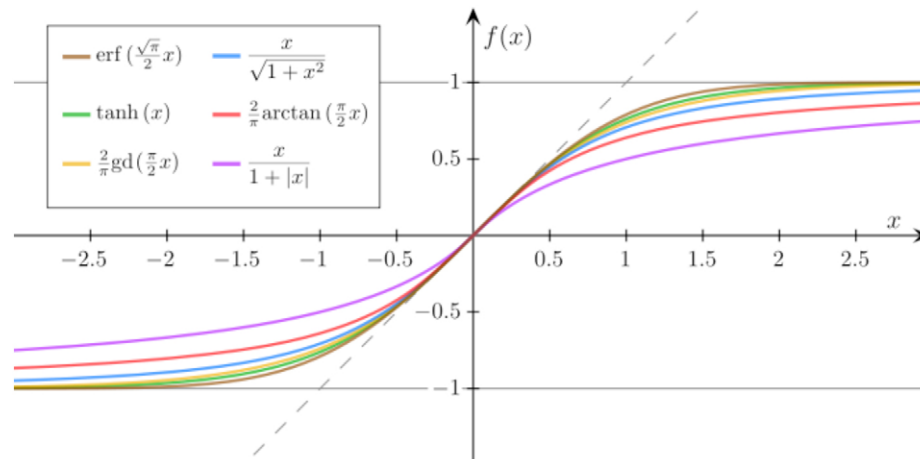


Figure 9: Comparison of Sigmoid functions (Sigmoid function, 2016)

We see that different Sigmoid functions vary only in one parameter (Steepness of the curve), we went with Logistic function where we can change all the three parameters.

## 3.2 Can the Logistic Model be used for Prediction?

Now that we know that the Logistic function is fairly accurate when it comes to Modelling the Cumulative Defects against Work Weeks, can it be used for Prediction? Precisely, for a new project, all we need to do is figure out a way to predict the 3 Parameters: Growth Rate, Inflection Point, and Asymptote. This may sound simple but there are a lot of steps involved in this process which will be explained in the next sections. From this point on, we will be referring to the combination of Logistic function + milestones as our Defect Prediction Model.

## 4 Project Milestones

In Section 2 and 3, we have seen how good our model works with the existing Defect Data. We also referred to the combination of Logistic function with the milestones as our Prediction Model. In this section, we will be discussing about some of the milestones in a Product Life Cycle (PLC) [7] specifically focusing on Alpha, Beta, and PRQ. Out of several milestones, we have shortlisted these 4 as they hugely impact the decision making and widely used by Project Managers.

### 4.1 Alpha

By the time we hit Alpha, all the Plan of Record (POR) features is completed from the development point of view. Validation is still in progress. Therefore, we observe that about 20-30% of the total defects are seen by Alpha.

### 4.2 Beta

We declare a product to be Feature Complete by Beta. During the Alpha to Beta timeframe, we mainly

focus on full validation, bug fixes and implementation of new features. We observe around 60-70% of the total defects by Beta. In other words, we expect to have around 30-40% open defects after Beta.

### 4.3 PRQ

PRQ or Product Release Qualification is the final milestone in the PLC. It is the final qualified release product ready to ship to the customers. During the Beta to PRQ timeframe, we mainly focus on bug fixes and implement any new Change Request (CR) from customers. Beta and PRQ timelines could be very close to each other depending on the number of new requests. We observe that around 70-80% of the total defects by PRQ which leaves us to an estimated 20-30% open defects after PRQ.

## 5 Generate Reference Parameters from Past Projects

Let's apply this Prediction Model to previous projects and calculate the 3 Parameters: Growth Rate, Inflection Point, and Asymptote. These parameters are taken as a reference for estimations performed on new and upcoming projects. The following Table 6 shows a list of projects with their respective parameters and milestones. The term WW 0215 represents the 2nd Work Week in the year 2015 and Alpha was declared at that time. All the 4 projects are roughly 1 year long.

**Table 6:** *Project A, B, C, and D with their parameters and milestones*

Project Name	Growth Rate	Inflection Point	Asymptote	Alpha	Beta	PRQ
Project A	0.18	25	3037	WW 0215	WW 1915	WW 2615
Project B	0.15	37	4147	WW 1015	WW 1415	WW 2215
Project C	0.04	78	260	WW 4813	WW 1014	WW 2614
Project D	0.05	78	309	WW 4214	WW 0815	WW 2615

We observe a lot of difference:

1. The Growth Rate varies from 0.04 to 0.18
2. The Inflection Point ranges from 25 to 78
3. A huge difference in Asymptote from 260 to 4147 defects

This raises a question as to why do we have such huge differences in the Parameters. Additionally, what if I were to make a prediction for a completely new project? Will I be able to predict in the first place? The answer to this question is both 'Yes' and 'No'. In this case, we haven't provided any details about the projects apart from the fact that they were of approximately of 1 year duration. What if Project A & B were similar type? What if Project B was derived from Project A with extra features? Do the parameters of A & B make sense this time around? Yes, it does. In fact, Project B is an extension of A which is the reason the parameters are similar. The same goes for Project C and D. Therefore, if we look at Project A & B separately from C & D, we can make good estimates for a new project which falls under one of them. In other words, we can form multiple groups each having their own reference parameters.

Another way of generating parameters is to look at a Project in more details and break it down in different parts. Some of them are listed below:

1. If a project contains multiple components like Hardware, Software, Firmware, etc., we can calculate the parameters of each separately. Separating Software from the Hardware bugs can be extremely beneficial when we have future projects having incremental updates in one of the disciplines while the other remains the same. For example, newer projects can have frequent software updates but

may use the same hardware from the previous ones. In this case, we can have the analysis ready for the software side.

2. This applies to projects with multiple ingredients. Some projects can exclude certain components from the previous ones. Analyzing with respect to individual ingredients can be helpful in this case.
3. Depending on the type of Defects: Critical, High, Major, and Minor. We may only be interested in the Critical and High defects. Analyzing those trends can come in handy.
4. One project can support different platforms such as Operating Systems, Architecture type, etc. We can observe the trends for each of them.
5. We can analyze the trends from the Defects that were reported internally and from the customers. By doing this, we can plan accordingly to reduce the number of defects for upcoming projects before releasing it to the customer.

**Table 7:** *Reference Parameters calculated with respect to the Type of Defects*

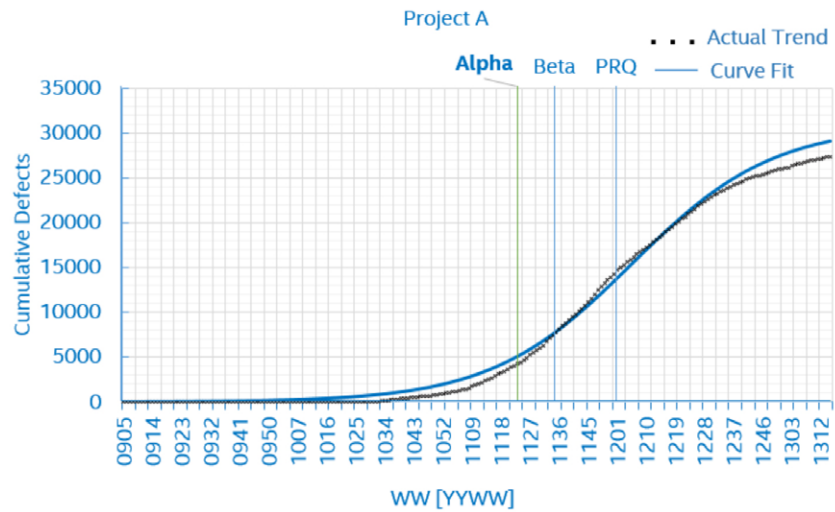
Projects	Growth Rate	Inflection Point	Asymptote
Project A Critical Issues	0.056	51	260
Project A High Issues	0.062	74	345
Project A Major Issues	0.046	77	114
Project B Critical Issues	0.057	83	821
Project B High Issues	0.038	114	1028
Project B Major Issues	0.044	96	250
Project C Critical Issues	0.043	80	280
Project C High Issues	0.044	78	260
Project C Major Issues	0.056	69	69

## 6 Predicting Defects and defining Project Milestones

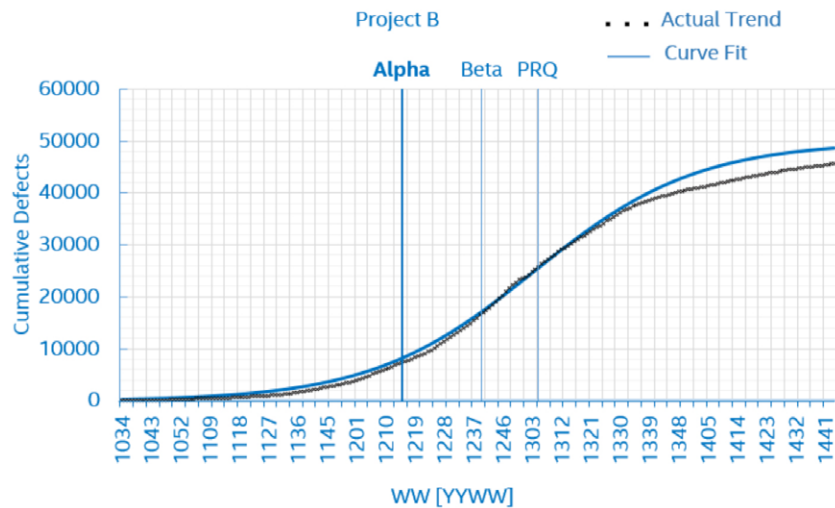
After we calculate the reference parameters from the previous projects, we need to understand how the new projects are related. Can we infer any similarities?

### 6.1 Discuss Parameters and Dependencies

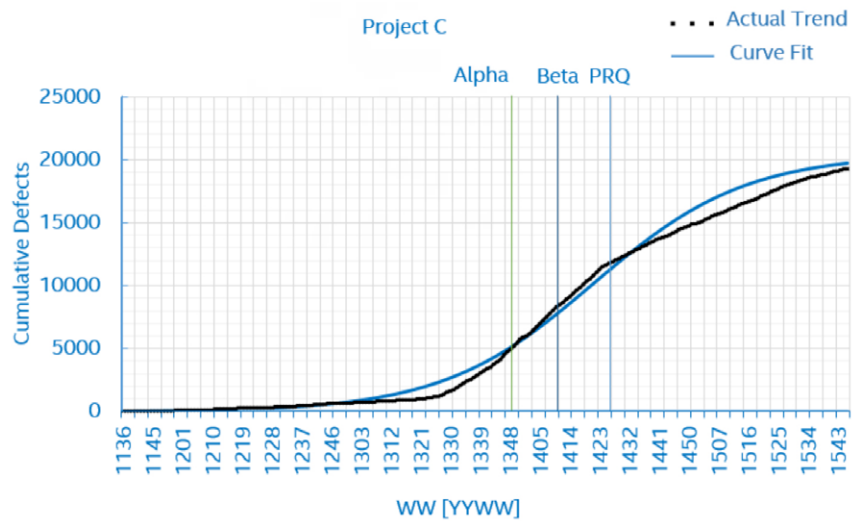
Let's take an example and look at the following 3 previous projects A, B, and C and their respective milestones. These projects are big and are about 3 years long. Also, we observe that the following projects have a lot of Open Defects after the PRQ milestone (around 50%), which is something we want to avoid for a new project. For a new project on-going project (say D), how do we reduce the number of defects before declaring the milestone PRQ? Let us discuss in detail.



**Figure 10:** Project 'A' Actual and Approximated trend



**Figure 11:** Project 'B' Actual and Approximated trend



**Figure 12:** Project 'C' Actual and Approximated trend

The parameters (Growth Rate, Inflection Point, and Asymptote) are calculated are shown below:

**Table 8:** *Reference parameters of Project A, B, and C*

<b>Project Milestones / Parameters</b>	<b>Project A</b>	<b>Project B</b>	<b>Project C</b>
Alpha	WW 2311	WW 1512	WW 4813
Beta	WW 3511	WW 4012	WW 1014
PRQ	WW 0112	WW 0513	WW 2516
<b>Defect Growth Rate</b>	<b>0.046</b>	<b>0.04</b>	<b>0.044</b>
<b>Inflection Point</b>	<b>159</b>	<b>128</b>	<b>143</b>
<b>Asymptote</b>	<b>30831</b>	<b>49897</b>	<b>20462</b>

In order to estimate the parameters of a new project (say Project D), we must understand the following dependencies such as Architectural complexity, Validation methodology, Resources, Number of Features, and Project Timelines. We will try to correlate the new Project D to any or all of the previous projects (A, B, C). We will perform Delta analysis between the previous projects and the new project. Also, note that we are not quantifying any of these dependencies. It is up to the user to make estimations based on their assumptions. Let's take a look at few of the dependencies.

#### **6.1.1 Architectural complexity**

How complex is the architecture of Project A, B, and C? How are they different? Are they related? Is one the improvement of the other? Did we modify a particular architecture to a point that it is completely different? These questions need to be addressed by the Program Architects by looking at various platforms. We just assign the terms "Not Complex", "Complex", and "High Complexity" to those projects.

Based on our observation, the complexity of a project is directly proportional to the Defect Growth Rate and Asymptote. This implies that, the higher complexity of the project, the more the Growth Rate as compared to the previous ones. In case of Inflection Point, we found that it is directly proportional to the Milestones but independent of the complexity.

#### **6.1.2 More of Less Features**

How many features were part of Project A, B, and C? Are some of the features similar to a new project? Are these features planned to be released for the first time? What about the number of programmers? Were there changes in Development methodology, Project culture, and development / verification technology? These questions need to be addressed by the Project Management for the previous projects. Based on our observations, the Number of Features in a project is directly proportional to all the 3 parameters: Defect Growth Rate, Inflection Point, and Asymptote. For example: A completely new feature never tested before will definitely increase the bug count.

#### **6.1.3 Project Timelines**

As timelines become cramped i.e. there isn't much duration between any two milestones (Alpha – Beta, Beta – PRQ), we observe a lower Defect trend. As a result, we tend to miss out lot of defects before the final release (PRQ). In order to avoid this issue altogether, we need to have a significant duration between any two milestones. This will increase the bug count before the PRQ and we'll see a higher Growth Rate and Asymptote.

## 6.2 Estimation of Parameters and adjusting Milestone

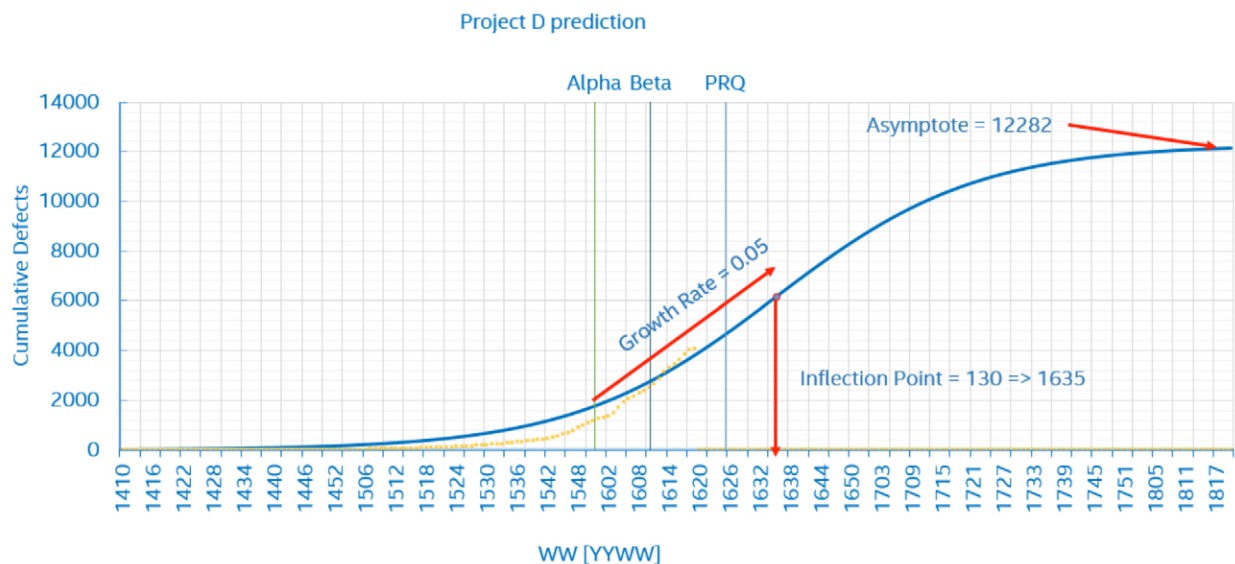
Once we have the details of Architectural Complexity, Number of Features, and Project Timelines, we adjust the Defect Growth Rate, Inflection Point, and Asymptote. Please note that, since we have a proportionality criteria and nothing is actually quantified, it is up to the individual to make necessary assumptions in estimating for newer projects.

In this case, based on our discussions with the Program Managers, we came to an agreement that the parameters for Growth Rate, Inflection Point, and Asymptote are 0.05, 130, and 12282 respectively.

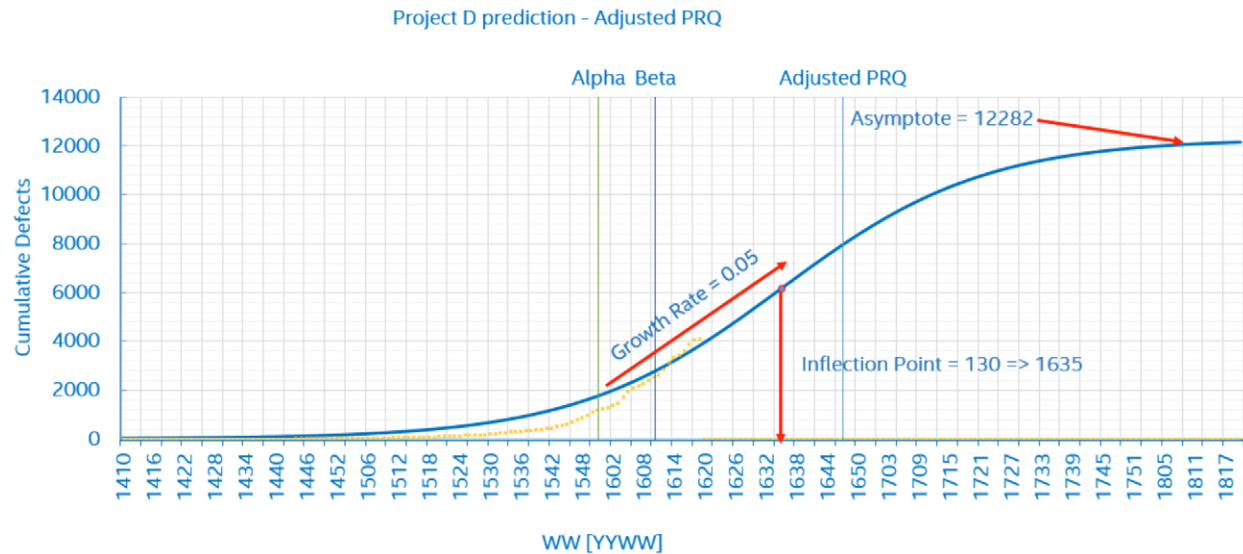
**Table 9: Estimated Parameters and PRQ change for Project D**

Project Milestones / Parameters	Project D with Old PRQ	Project D with New PRQ
Alpha	12/14/2015	12/14/2015
Beta	2/29/2016	2/29/2016
PRQ	6/13/2016	11/13/2016
Estimated Defect Growth Rate	0.05	0.05
Estimated Inflection Point	130	130
Estimated Asymptote	12282	12282

When we plot the actual trend of the Defects with the milestones, we observe that our PRQ is declared a bit too early. Approximately 55% of the unknown Defects are expected by Beta.



**Figure 13: Predicted trend of Project D Vs the Actual Trend**



**Figure 14: Adjusted PRQ**

With the help of this model, we have pushed the PRQ milestone by 5 months and now we account for around 30 – 35 % of the open defects. During this time, we can focus more on the validation such that we further decrease from the 30% mark and come down to 10% open defects. Program Managers can always revisit their assumptions and make necessary changes to the 3 parameters as the project progresses and monitor the S curve trends i.e. actual vs predicted curve.

## 7 Conclusion

Planning a Project schedule and implementing it to the finest is an age old problem faced by Project Management Teams. Our Defect Prediction Model not only eliminates the uncertainty in declaring milestones but also suggests corrective actions during an on-going project. This tool helps the management by predicting the future defects and providing a robust Project schedule. Additionally, we monitor in-process quality of products and take preventive actions on the unforeseen defects/issues/risks. To conclude, we have successfully applied our model to approximately 50 different platforms across different business segments and as a result taken preventive measures in upcoming projects.

## References

*Bugzilla*. (n.d.). (Bugzilla) Retrieved from <https://www.bugzilla.org/>

Fenton, N., & Neil, M. (1999, Sep/Oct). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5), 675-689. doi:10.1109/32.815326

*Fit Curve Report*. (n.d.). (JMP) Retrieved from [http://www.jmp.com/support/help/The\\_Fit\\_Curve\\_Report.shtml](http://www.jmp.com/support/help/The_Fit_Curve_Report.shtml)

Garland, D. (2009, May). *The Mysterious S-curve*. Retrieved from <http://www.maxwideman.com/guests/s-curve/intro.htm>

Garland, D. (2009, May). *The Mysterious S-curve*. Retrieved from <http://www.maxwideman.com/guests/s-curve/using.htm>

*Jira Software*. (n.d.). (Atlassian) Retrieved from <https://www.atlassian.com/software/jira>

*JMP Pro*. (n.d.). (JMP) Retrieved from [http://www.jmp.com/en\\_us/software/jmp-pro.html](http://www.jmp.com/en_us/software/jmp-pro.html)

*Logistic 3P - Fit Curve Options*. (n.d.). (JMP) Retrieved from [http://www.jmp.com/support/help/Fit\\_Curve\\_Options.shtml](http://www.jmp.com/support/help/Fit_Curve_Options.shtml)

*Logistic function*. (2016, July 22). (Wikipedia) Retrieved July 27, 2016, from Wikipedia, The Free Encyclopedia.: [https://en.wikipedia.org/w/index.php?title=Logistic\\_function&oldid=731037380](https://en.wikipedia.org/w/index.php?title=Logistic_function&oldid=731037380)

*Sigmoid function*. (2016, July 25). Retrieved July 27, 2016, from Wikipedia, The Free Encyclopedia.: [https://en.wikipedia.org/w/index.php?title=Sigmoid\\_function&oldid=731468779](https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=731468779)

Wideman, M. (May 2001). Wideman Comparative Glossary of Common Project Management Terms v2.1.

# Infrastructure Orchestration to Optimize Testing

**Rohit Naraparaju**

[rohit.naraparaju@intel.com](mailto:rohit.naraparaju@intel.com)

**Sajeed Mayabba Kunhi**

[sajeed.mayabba.kunhi@intel.com](mailto:sajeed.mayabba.kunhi@intel.com)

## Abstract

The advent of new automated testing frameworks and tools has brought about a new revolution in the testing landscape. When used effectively, these tools and frameworks can considerably reduce manual efforts during testing. However, there still remains an area which deserves some attention and has sufficient scope for optimization. This is the setting up of the test environments.

With virtualization gaining popularity, a new dimension has opened as to how testing infrastructure can be configured. The world has moved from setting up a room full of test servers to virtualized environments. Infrastructure Orchestration is the ability to use this advancement in virtualization to spin up complex multi-node, multi-network test infrastructures at will, perform the test, and remove the infrastructure when done. The ability to do this as part of a Continuous Integration or Continuous Delivery pipeline adds vast value to testing and brings down manual effort tremendously. A team which would otherwise spend days or weeks to setup infrastructure can now quickly add value to testing the product, thereby contributing to product quality.

This paper details how Infrastructure Orchestration and Configuration Management can be used to automate test infrastructure setup to reduce the manual effort and optimizing the test cycle overall.

## Biography

*Sajeed Kunhi is a senior engineering manager with Intel with nearly twenty years of software engineering experience. He is highly passionate about process automation, product quality and DevOps principles. His expertise in software development, technology domain and managerial experience has helped in building optimized software development lifecycle.*

*Rohit Naraparaju is software builds/tools engineer with Intel with over eight years of software development experience. He is an ardent believer in DevOps and has dedicated a good amount of his career working on Infrastructure Orchestration, Continuous Integration, Configuration Management, Test Automation, and Release Management. Rohit is currently working on advancing the usage of Configuration Management in various other products within his group.*

# 1 Introduction

Testing plays a vital role in the Software Development Life Cycle (SDLC) that helps in improving the quality and reliability of the software thus ensuring that software does what it is supposed to do. In this new competitive era, a lot of companies are looking at shortening their release cycles from years to months to days. To support this new trend of release, product teams are looking at implementing smaller feature sets which need to be tested thoroughly on a variety of environments, performing various kinds of tests, before the software makes it to the successful release. Therefore, testing plays an important role, and should be introduced in the early stages of SDLC as fixing a bug in critical stages is very expensive.

To support testing from the early stages of development, product teams need to test the new features on various testing environments as soon as they have them developed. Traditionally, the test environments are pre-created and configured as per the product team guidelines. To set up the infrastructure for each test environment could take anywhere from a week to more than month, depending on the infrastructure requirements from the product teams. This delay can cause significant impact to testing activities and delay the release of the product.

In this paper, we present the issues faced with the traditional approach of pre-creating the environments for testing by requesting the infrastructure from IT, and how we had to use the same environments for different kinds of tests based on our experience and learnings from earlier projects.

Our aim is to present how we are solving this problem by following a new approach of automating the Infrastructure Provisioning of test environments at will using various tools. We also present how we have integrated this process as part of our Continuous Integration pipelines to test the product from code check-in to running unit tests, spinning of a new testing environment, deploying the product using the Configuration Management tool, running integration tests, and sending the results of the tests as an email to the subscribers.

## 2 Problem Statement

Testing an application that involves multiple systems is a cumbersome task. To deliver a reliable product to the customer, we need to perform detailed testing of the independent components, systems, and other systems that integrate with the core product. The first step in performing testing is to come up with the infrastructure needs and configuration of test environments. If the product team requires a basic web server and database server for the set up, then it is easy to set up the environment. However, as the product starts integrating with other systems, the set up becomes complex and setting up an environment manually will take lot of manual effort, time, and human errors when setting up configurations.

## 3 Problems with the Fixed Environments and how it impacts Testing

In our earlier project, to set up a single testing environment, we required two web servers and one database server. We had to then finalize the number of environments required for the project, as well as deciding on the computing resources required in total for setting up the environments. Once we had all these details, we had to engage with IT to set up a project on their existing Team Foundation Server (TFS) lab environment and get the things finalized.

The process we followed to set up the environment was as follows:

1. System preparation for Web server:
  - a. Decide on the OS for the test environment

- b. Create a Virtual Machine (VM) required with specified OS, Network LAN, RAM, CPU Cores, and Disk Space for Web server
  - c. Configure Internet Information Services (IIS) for the Windows server
  - d. Install the required software for the product
  - e. Install all Windows update
  - f. Disable the firewall settings
  - g. Convert this VM as a template for Web server
2. System preparation for Database server:
  - a. Decide on the OS for the test environment
  - b. Decide on the MS SQL Server Version and Flavor
  - c. Create a VM required with specified OS, RAM, CPU Cores, and Disk Space for Database server
  - d. Install the MS SQL Server and Configure the necessary settings required for the project
  - e. Install the required software for the product
  - f. Install all Windows update
  - g. Disable the firewall settings
  - h. Convert this VM as a template for Database server
3. Clone two VMs from the Web server template
4. Clone a VM from the Database server template
5. Create an Environment Template by adding the two web server VMs of Web server and the database server VM of Database server
6. Give a unique name to the Environment Template
7. Create an environment from the Environment Template
8. Give a unique name for each environment
9. Login to each VM and change the name of the machines
10. Install Visual Studio Test Agents on each VM and configure them to talk to the TFS Test Controller
11. Take a clean snapshot of VM
12. Repeat the steps 3-8 for each planned environment
13. Deploy the product by running shell scripts from TFS Build Deployment definition as part of the CI process which execute the automated tests.

The above process worked great in the initial stages where the code base was small and few changes were required in configuring the environments. However as the product matured and integrated with multiple systems, we came across changes to versions of software, changing IIS settings, adding new components, etc. for the environments which became very difficult to update over a period of time, often spending a day or two in getting all the environments to the new set of changes.

Disadvantages with pre-created environments:

1. Requires lot of manual configuration and can be error prone
2. Making a small change to the environment configuration would require us to login in to all environments, propagate the change, and take the snapshots

3. Restricted to work with the same configuration settings on all the environments which may vary with the production environments
4. A change in the OS version or SQL Server version meant that we should have to repeat this entire process, which would delay making the test environments available for testing new features
5. If the TFS lab goes down, all the environments go down
6. Maintenance of these environment overtime became time consuming
7. Faced a lot of connectivity issues multiple times with the TFS lab environment affecting testing
8. Upgrade to a new TFS Lab environment version means recreating the entire set up of testing environments as there was no easy way to migrate the existing environments to new versions
9. Need a dedicated resource to take care of these issues, work with IT, and investigate them
10. Space limitations on the storage in the Lab environments
11. Setting up a new environment means provisioning the servers, configuring the servers, and getting the applications to run

## 4 Approach

Considering the issues that arise during setting up a fixed environment for testing, choosing a right approach for setting up the testing environment at will depends on following process:.

1. Determine if the team will adapt to the new technologies and tool set used to set up the environment.
2. Evaluate a cloud services platform (such as: AWS, VMware, Azure, Openstack, Private cloud) by understanding its capabilities, advantages, costs, documentation, pricing, and support
3. Choose the right Configuration Management tool, Infrastructure Provisioning tool, and Continuous Integration tool by evaluating the tool, team's adoption to new tool, documentation, training, licensing and support.
4. When we understand and realize the importance of Infrastructure Provisioning, Configuration Management, and Continuous Integration the next challenge is to select the appropriate scripting language that supports making calls to the API's exposed by the cloud services platform in order to set up the testing environment.

Based on above discussed points, we give a small overview of Infrastructure Orchestration and Configuration Management and then give a picture of how all the roles need to come together to make this solution successful.

### 4.1 Infrastructure Orchestration

Orchestration mean arranging, coordinating, provisioning, and managing the complex computer systems, middleware, and services. For example, if a team may not only want to spin up a server, but install the software, configure the database, provision load balancers, and get all these things orchestrated for testing or production purpose, then that is described as Infrastructure Orchestration.

Infrastructure Orchestration addresses the problem of getting infrastructure components to work together as documented in code or in a structured document. In the compute realm, it is often used to spin off a specific OS or a custom image and its resources to get a service up and running. It also addresses the problem at a layer above the configuration management system. Configuration Management systems are good at describing the details of nodes but less good at coordination amongst clusters and complex provisioning tasks. Examples of Infrastructure Orchestration tools are: AWS CloudFormation and Terraform. See details in the "Tools" section below.

From the Developers/Testers perspective, they will use the resources intended for their purpose and release them after they have completed their task. Therefore Infrastructure Orchestration must support assigning and removing the environments from the shared pool. The main goals of Infrastructure Orchestration are to manage the resources, avoid conflicts, and enable user productivity.

Advantages:

1. Cost Savings
2. Manageability
3. With the Infrastructure Orchestration in place, IT can collaborate with product team and come up with new infrastructure topologies which can be later be converted to standardized templates for other teams to consume/integrate with.
4. Product team can spin up multiple environments on an as-needed basis
5. Reduces manual effort of setting up the testing environment
6. Testing environments will be consistent
7. The setup process can be integrated into the Continuous Integration and Delivery processes

## 4.2 Configuration Management

Configuration Management can be defined as a process that deals with maintaining the hardware and software for a product. It records information about the computer system and updates that information as needed. It is a declarative system for managing the configuration of an instance. These details can be installation of packages, starting/stopping services, and editing config files on the instance. This simplifies engineering efforts in determining which programs are installed and which systems need to be upgraded.

A Configuration Management system allows you to describe the infrastructure pieces in a high-level, Domain Specific Language (DSL) and often provides facilities that abstract the underlying actions being taken. This allows you to track changes to the systems over time through source control, easily move between OS/platforms, quickly modify the state of the infrastructure, and test the changes. In a nutshell, Configuration Management systems do the detail work of making one or many instances perform their roles without the operator needing to specify the exact commands needed to configure the system under test.

Deployment of applications, and maintaining infrastructures are very critical and delicate tasks which were traditionally managed by hand. If the above tasks could be handled by a tool then it would save lot of time and also remove the element of human error. Therefore, this led to the rise of Configuration Management tools which handle the automation of configuration states for the application. Like any other tool, this requires effort to learn, and depends on the skills of the person putting them together to work.

The two most popular Configuration Management tools are Chef and Puppet. Both these tools serve the same purpose and are designed to accomplish the Configuration Management for an application.

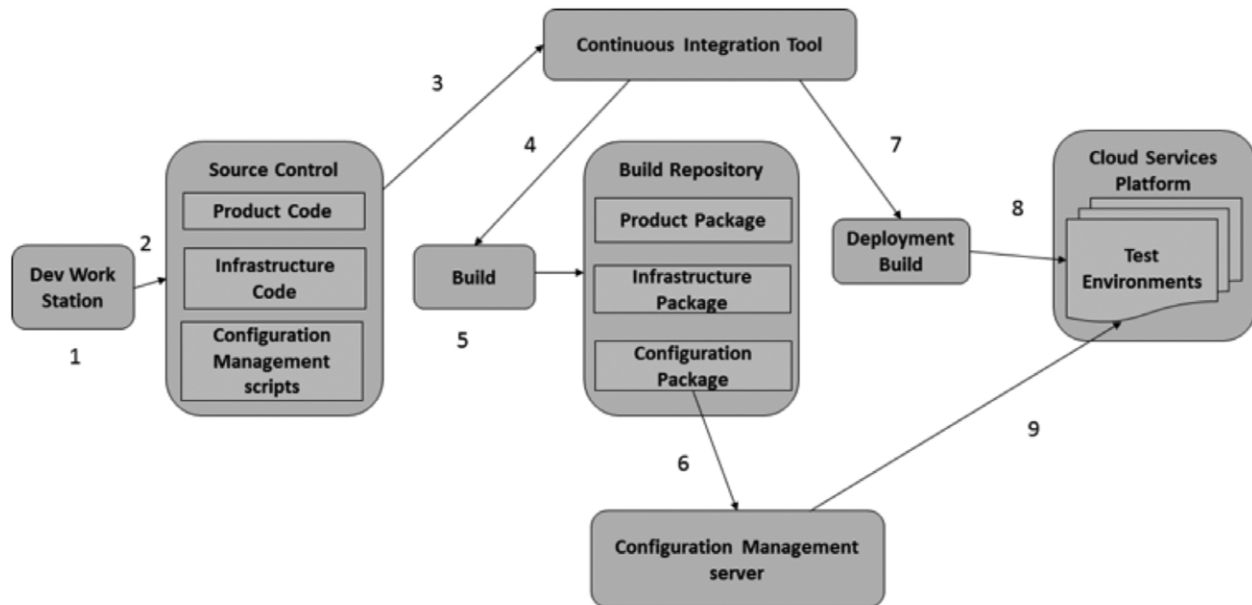
Advantages:

1. Faster restoration of the application
2. Increased efficiency of configuring the system
3. Cost reduction
4. Strong security

### 4.3 Automated Infrastructure Orchestration Workflow

Once we have a clear understanding of the terms Infrastructure Orchestration and Configuration Management, the next step is to establish a workflow around setting up a fully automated infrastructure orchestrated solution.

The diagram below represents the workflow and the jobs performed by the roles.



**Fig 4.1a:** Continuous Deployment Process

1. Write a structured Infrastructure Provisioning and Configuration Management scripts required for setting up the test environment
2. Check-in these scripts to the source control
3. Source control sends a notification to the Continuous Integration tool
4. A Continuous Integration tool kicks off the compile build
5. Compiled build copies the packages to the Build Repository
6. Get the latest Configuration Management scripts from Build Repository and upload to the Configuration Management server
7. Continuous Integration tool kicks off a deployment build
8. Deployment build is responsible for spinning up a test environment dynamically in the cloud service platform provider space
9. After the test infrastructure is up, the deployment build runs the build step where the Configuration Management scripts bring the test environment to the desired state

## 5 Tools

Based on the points described in the approach section, we have classified the tools required for setting up the infrastructure automatically into four categories:

1. Cloud Services Platforms
2. Infrastructure Provisioning Tools

3. Configuration Management Tools

4. Continuous Integration Tools

Below, we give a brief overview of the most popular tools in the market for each category.

## **5.1 Cloud Services Platforms**

### **5.1.1 Amazon Elastic Compute Cloud (EC2)**

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides scalable computing capacity in the Amazon Web Services (AWS) cloud. It is designed to make cloud computing easier, develop and deploy applications faster thus eliminating the need to invest in hardware upfront.

EC2 web service interface allows you to launch virtual servers, configure capacity, security and network, and manage storage that you need as well as providing complete control of the computing resources. It enables you to scale the capacity both up and down depending on changes in computing requirements thereby allowing you to pay just for the capacity that is used. It also provides developers a lot of tools to handle build failure applications and isolate the common failure scenarios.

Benefits of Amazon Elastic Compute Cloud:

1. Elastic web-scale computing
2. Completely controlled
3. Flexible cloud hosting services
4. Designed for use with other Amazon web services
5. Reliable
6. Secure
7. Inexpensive
8. Easy to start

### **5.1.2 Microsoft Azure**

Microsoft Azure is a public cloud computing platform and infrastructure created by Microsoft for developing, deploying, and managing the applications on a global network of data centers managed by Microsoft. It is widely considered as both Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) which offers cloud services such as compute, analytics, storage, and network thus supplementing the use of on-premise servers.

Facts about Azure:

Flexible – Scale compute resources up and down as needed

Open – Supports almost any language, OS, tool, or framework

Reliable – 99.95% availability and 24 X 7 tech support

Global – Data housed in geo-synchronous data centers

Economical – Only pay for what you use

Microsoft Azure categorizes its services into 11 types:

1. Compute

2. Web and Mobile
3. Data Storage
4. Analytics
5. Networking
6. Media and Content Delivery Network
7. Hybrid Integration
8. Identity and Access Management
9. Internet of Things
10. Development
11. Management and Security

### **5.1.3 VMware VCloud Air Virtual Private Cloud**

VMware VCloud Air Virtual Private Cloud is a multitenant, logically isolated cloud Infrastructure as a Service (IaaS) compute solution that lives on shared infrastructure. It is architected on VMware vSphere design elements that delivers high availability and fully private networking capabilities. It is also a secure, flexible, capable, and expandable IaaS environment that helps to launch new applications onto the cloud.

Benefits:

1. Compatible with an On-Premises vSphere
2. Flexible VM sizing
3. Application Portability
4. Advanced Networking
5. High Availability
6. Cost effective starting point

Features offered by Virtual Private Cloud are:

1. Trusted Hypervisor
2. High Performance
3. Custom VM sizing
4. Tiered storage options
5. Enterprise-grade network virtualization
6. VM lease and quota management
7. Direct console access

## **5.2 Infrastructure Provisioning Tools**

### **5.2.1 Terraform**

Terraform is a tool to build, change, and version the infrastructure such as VMs, network switches, etc safely and efficiently. It can manage existing service providers as well as the custom built in-house providers

Terraform uses a declarative language to describe the resource configurations. The main advantage of Terraform is that it is cloud agnostic and enables multiple providers and services to be combined and composed.

A configuration file tells Terraform the components needed to set up the application. Terraform generates an execution plan that describes the desired state to reach and later executes it to build the desired infrastructure described in the plan. If the configuration changes, Terraform has the capability to detect the changes and create incremental execution plans to be applied to the set up the infrastructure.

Terraform operates at a lower level than the Configuration Management tools. While Configuration Management tools work on the existing resources, Terraform actually builds those nodes by managing the components such as storage, compute instances, networking, DNS entries, etc.

Key features of Terraform are:

1. Infrastructure as code
2. Resource graphs
3. Execution Plans
4. Change Automation

### **5.2.2 AWS CloudFormation**

AWS CloudFormation is a service to model and set up the Amazon Web Services resources. It gives the developers a means to create and manage the resources, provision, and update them in less time and helps them to focus more on developing/testing the applications.

Developers can use the sample templates offered by CloudFormation or create their own templates to describe all the AWS resources like instances, associated dependencies, and runtime parameters required to run the application. Developers don't need to worry about the order in which the provisioning takes place as CloudFormation takes care of this for them. After the AWS resources are deployed, they can modify and update them in a controlled way.

Features of CloudFormation are:

1. Supports a wide range of AWS resources
2. Easy to use
3. Declarative and Flexible
4. Infrastructure as code
5. Customized via parameters
6. Visualize and edit with drag-drop interface
7. Integration ready

### **5.2.3 VMware vRealize Automation**

VMware vRealize Automation is designed to improve the deployment, application management, personalized infrastructure, and compute services thus improving the operational efficiency. It is a software product for unified cloud management that provides the administrators the ability to provision and configure the storage, network, compute resources across private and public cloud platforms. Its key benefit is that we can deploy an automated, on-demand cloud infrastructure thereby giving flexibility. It is also very well integrated with the VMware VCloud Air Virtual Private Cloud platform. It helps in automating the infrastructure set up by leveraging the variety on CLI/APIs exposed by the cloud platform for requesting and configuring the VMs.

## 5.3 Configuration Management Tools

### 5.3.1 Chef

Chef is a ruby based Configuration Management tool which can be easily installed and configured to specific needs with little bit of programming knowledge. It is available as a free open source tool as well as a paid enterprise subscription. Chef utilizes a master-agent model. Apart from the master server, it requires a workstation setup that controls the master server. A Chef client would be running on all the nodes which have to be configured.

Chef has the unique concept of cookbooks. A cookbook is the fundamental unit of configuration in Chef. Cookbook is a collection of recipes and associated attributes. Recipes are collections of resources that are written in ruby and Attributes provide specific details of the node. The Chef client is responsible for pulling the cookbooks from master server and running the specific configurations on respective nodes to bring them to a desired state. Chef has a lot of features such as text-based search and support for multiple test environments. Its command line interface, testing mode, and large database make it ideal for companies that need to store records for a large number of computers. The capability to install or even create different modules makes this one of the most customizable Configuration Management options.

### 5.3.2 Puppet

Puppet is a ruby based Configuration Management tool like Chef. It is also available as free open source and paid enterprise versions. The configuration code is written using Puppet's Domain Specific Language (DSL) and wrapped in modules. Puppet also runs an agent on all the nodes to be configured and it pulls the compiled module from the Puppet server and install required software packages specified in the module. Role-based access, orchestration, automation, and event inspector are some important features on Puppet enterprise. It is also designed to work on a variety of platforms.

### 5.3.3 Ansible

Ansible is relatively a new tool in the market when compared to Chef and Puppet. It is an agentless tool that use Secure Shell (SSH) to provide a simple Configuration Management tool yet with strong security capabilities. Configuration modules in Ansible are call playbooks which are written in Yet Another Markup Language (YAML). It is relatively easy to learn and write. Besides Configuration Management, it offers workflow monitoring and automating application deployment for updates like an orchestration tool. Ansible focuses on five principles:

1. Small learning curve
2. Ease of use
3. Automation everything
4. Efficiency
5. Strong security

## 5.4 Continuous Integration Tools

### 5.4.1 Jenkins

Jenkins is an open source, cross-platform, Continuous Integration and Continuous Delivery application that increases the productivity. Continuous Integration is the practice of running the automated tests on a non-developer machine every time new code is pushed into a source repository.

It has tremendous advantage of always knowing if all tests work and getting fast feedback. The fast feedback is important so you always know right after you broke the build (introduced changes that made either the compile/build cycle or the tests fail) what you did that failed and how to revert it.

If you only run your tests occasionally the problem is that a lot of code changes may have happened since the last time and it is rather hard to figure out which change introduced the problem. When it is run automatically on every push then it is always pretty obvious what and who introduced the problem. Some important features of Jenkins include easy to install, and configure, extensibility, and support for a large number of plugins.

#### **5.4.2 TeamCity**

TeamCity is a Java-based Continuous Integration server package from JetBrains. It is easy to install and configure. Though it is a java based, it supports a lot of .NET development shops. TeamCity server is the main component and comes with a browser hosted interface. The interface is designed to be the primary source to administer TeamCity users, agents, projects, and build configurations.

TeamCity provides project status and reporting dashboards which can be subscribed by the interested users and project stakeholders. It also provides build progress, drill down detail, and history information on the projects and configurations.

TeamCity installation also comes with a system tray utility to get the information of build status and progress. Build notifications such as passed/failed are received in this tray instead of email.

Some important features of TeamCity include easy setup, use, configurability, and being well-documented. The tool also has the ability to integrate with wide range of tools and technologies.

## **6 Solution**

Our new project is based on the Big Data platform technologies which usually require a large infrastructure cluster to be up and running for supporting the deployment of an application. The main aim of this new project is to store different types of threat events coming from McAfee agent into a scalable distributed database and move away from storing into Microsoft SQL Database Server.

In the new project we had to use the Cloudera Distribution including Apache Hadoop (CDH) platform that provides the Apache Hadoop based software, support, and services. Apache Hadoop is an open-source software for reliable, scalable, distributing computing. It is a framework that supports distributed processing of large data sets across clusters of computers. It is mainly designed to scale up from a single machine to thousands of machines, offer local computation, storage, handle failures, and deliver highly-available service on top of a cluster of computers.

As Cloudera has numerous products in the offering, we handpicked a few of the products that were useful for designing our solution. In order to make the solution more reliable and robust, we needed a fairly big infrastructure set up for deployment and testing.

For setting up a single testing environment, we required about 12 VMs with a higher configuration specification per VM (such as higher memory, CPU cores, and larger disk spaces). If we had taken a traditional approach of setting up this environment by going through IT channel, it could have taken anywhere from a day to weeks just to get approved for infrastructure. It could have taken another few days to configure and deploy the application on the environment manually. Once we had the environment configured manually, validating the environment and fixing the issues could take anywhere from few hours to days due to the number of VM's involved.

Since setting up a test environment such as this was very time consuming, and required a lot of manual

intervention, and expense, we decided to adopt the approach of Infrastructure Provisioning combined with a Configuration Management tool to set up the environment of testing.

We selected the following technologies and tools to develop the Infrastructure Provisioning solution that helps in spinning up a testing environment for our project based on the technical expertise within the team.

1. In-house VMware VCloud Virtual Private Cloud as Cloud Services Platform
2. Chef for Configuration Management
3. Jenkins for Continuous Integration
4. VMware vRealize Automation (vRA) for Infrastructure Provisioning
5. Python scripting to bundle the vRA API calls, Chef knife bootstrap commands, and running automated tests

To set up and configure this infrastructure automatically, we followed these steps:

1. Create a job in Jenkins for provisioning the infrastructure
2. Write Chef cookbooks and check-in to source control
3. Write a python script for provisioning the VMs using vRA API's and bootstrapping the Chef commands
4. Check-in the python script to the source control
5. Configure the Jenkins job to call this python script that provisions, configures the infrastructure, and run automated tests
6. Configure a post build deployment step in Jenkins to publish the test results and send out an email notification to the interested subscribers.

#### Python script layout

We chose python because of the features it offers, its simplicity, and ease of maintenance. Also, most of the members in the team were comfortable with this scripting language. This script handles the core functionality of setting up the testing environment. Below is the layout of the script

1. Read the VM configurations specified in the config file
2. Call the vRA API's exposed by the Private Cloud for requesting the VMs
3. Check the provisioning status for the VMs
4. Store the result of the provisioned VM in a JSON file
5. Call the Chef knife bootstrap command to configure the respective application by passing the Chef's recipe list for the VM

This python script can be invoked by passing it as a build step in Jenkins as part of continuous deployment or from a developer work station with the required parameters for setting up the infrastructure required for the testing environment.

#### Advantages

With the above tools, technologies, and scripts in place and configured to the needs of the product team, the team was able to

1. Spin up environments on an as-needed basis by developers/testers
2. Integrate Infrastructure Provisioning, Configuration Management, and Automated Tests into the Continuous Integration process

3. Spin up environments with different configurations in less time
4. Enable immediate developer testing, rather than waiting for a specific stable build
5. Run multiple builds performing various tests on various environments
6. Save time that would have been spent on manual intervention
7. Create environment configurations consistent across all environments

## 7 Conclusion

This paper presents our experience with how fixed environments can cause delay and impact testing activities, and how we wanted to address this problem in our new project by moving to Infrastructure Provisioning combined with Configuration Management tools to set up the environments for different kinds of tests on an as-needed basis. Testing is one of the key components of Software Development Life Cycle (SDLC) and it needs to be started from the beginning of a project, and not left to the end.

In order to support testing activities, many factors need to be considered, and the one that tops the list is infrastructure availability. Once we have this issue resolved, then the product teams can spin up multiple testing environments as needed, to perform various tests (i.e. BVT, Regression, Integration, System tests, etc), optimize the testing phase, and qualify the product for production grade. When we get the entire process correct, a product team may have the capacity to release product to production more often, and in shorter periods of time, leading to increased customer satisfaction.

## References

Upguard, “7 Configuration Management (CM) Tools You Need to Know About”, <https://www.upguard.com/articles/the-7-configuration-management-tools-you-need-to-know>

DEVOPSCUBE, “10 Devops Tools for Infrastructure Automation”, <http://devopscube.com/devops-tools-for-infrastructure-automation/>

Adam Bertram, June 24, 2015, “Choosing The Right Configuration Management Tool”, <http://www.tomsitpro.com/articles/configuration-management-tools,2-920.html>

Quora, “Infrastructure Orchestration and Configuration Management”, <https://www.quora.com/How-do-I-learn-orchestration-and-configuration-management>

“Microsoft Azure”, <http://searchcloudcomputing.techtarget.com/definition/Windows-Azure>

Spencer Towle, August 31, 2015, “Microsoft Azure Explained: What It Is and Why It Matter”, <http://ccbtechnology.com/what-microsoft-azure-is-and-why-it-matters/>

“What is Python? Executive Summary”, <https://www.python.org/doc/essays/blurb/>

“VMware Virtual Private Cloud”, <http://vcloud.vmware.com/service-offering/virtual-private-cloud>

“VMware vCloud Air Virtual Private Cloud”, <http://www.vmware.com/cloud-services/infrastructure/vcloud-air-virtual-private-cloud/>

Joao Miranda, August 4, 2014, “Terraform Infrastructure”, <https://www.infoq.com/news/2014/08/terraform>

“Terraform”, <https://www.terraform.io/intro/>

“Cloud Formation”, <https://aws.amazon.com/cloudformation/>

“Cloud Formation”, <http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html>

“VMware vRealize Automation”, <http://searchservvirtualization.techtarget.com/definition/VMware-vCloud-Automation-Center-vCAC>

“VMware vRealize Automation”, <http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/product/vcloud/vmware-vrealize-automation-datasheet.pdf>

Stephen Ritchie, November 20, 2012, “TeamCity vs Jenkins: Which is the Better Continuous Integration (CI) Server for .NET Software Development?”, <https://www.excella.com/insights/teamcity-vs-jenkins-better-continuous-integration-server>

“Amazon Elastic Compute 2”, <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>

“Amazon Elastic Compute 2”, <https://aws.amazon.com/ec2/>

“FountainHead, March 16, 2009”, <http://fountnhead.blogspot.com/2009/03/what-is-infrastructure-orchestration.html>

<https://en.wikipedia.org/>

<http://www.cloudera.com/>

<http://hadoop.apache.org/>

# Cultivating Software Performance in Cloud Computing

Li Chen, Colin Cunningham, Pooja Jain, Chenggang Qin  
and Kingsum Chow

{li.chen, cunningham.colin, pooja.jain}@intel.com,  
{chenggang.qcg, kingsum.kc@alibaba-inc.com

## Abstract

There exist multitudes of cloud performance metrics, including workload performance, application placement, software/hardware optimization, scalability, capacity, reliability, agility and so on. In this paper, we consider jointly optimizing the performance of the software applications in the cloud. The challenges lie in bringing a diversity of raw data into tidy data format, unifying performance data from multiple systems based on timestamps, and assessing the quality of the processed performance data. Even after verifying the quality of cloud performance data, additional challenges block optimizing cloud computing. In this paper, we identify the challenges of cloud computing from the perspectives of computing environment, data collection, performance analytics and production environment.

## Biography

*Li Chen is a data scientist at Intel. She received her PhD degree in Applied Mathematics and Statistics from the Johns Hopkins University in 2015. She specializes in applying analytics to system performance data.*

*Colin Cunningham is a data scientist at Intel. He received his Master's in Statistics from Pennsylvania State University in 1994.*

*Pooja Jain received a MS Degree in Computer Science and Engineering from University at Texas at Dallas in December, 2015. Pooja's curiosity about performance analysis, got her to Intel as a Server Performance Engineer. Pooja has hands-on experience enabling, analyzing and optimizing applications in a cloud environment. Pooja has 4 technical papers, including one at PNSQC 2015.*

*Chenggang Qin is a senior technical expert at Alibaba Inc. He received his PhD degree in Computer Science from the University of Chinese Academy of Sciences in 2012. He specializes in developing performance analysis system.*

*Kingsum Chow is currently a Chief Scientist at Alibaba Infrastructure Services. Before joining Alibaba in May 2016, he was a Principal Engineer and Chief Data Scientist in the System Technology and Optimization (STO) division of the Intel Software and Services Group (SSG). He joined Intel in 1996 after receiving Ph.D. in Computer Science and Engineering from the University of Washington. Since then, he has been working on performance, modeling and analysis of software applications. At the Oracle OpenWorld in October 2015, Intel and Oracle CEO's announced the joint Cloud lab called project Apollo led by Kingsum in the opening keynote in front of tens of thousands of software developers. He has published more than 20 patents and presented more than 80 technical papers. In his spare time, he volunteers to coach multiple robotics teams to bring the joy of learning Science, Technology, Engineering and Mathematics to the K-12 students in his community.*

# 1 Introduction

Cloud computing has become increasingly important. Deploying diverse applications upon cloud infrastructure introduces layers of complexity. To gain understanding and design a more efficient cloud, system and software performance data over compute, memory, storage and network are collected from virtual and physical machines. Examining these immense cloud performance datasets and uncovering patterns presents a daunting challenge. The model of computation is such that hundreds of compute nodes host thousands of virtual machines to deliver a diverse range of software over the internet. The cloud meets storage needs with a combination of block and object storage interacting with application and database virtual machines to provide data management functionality. Cloud management requires a set of administration nodes and hypervisors distributed on physical compute server resources to orchestrate the cloud.

To gain understanding and propose more efficient cloud management solutions, performance data is collected from virtual machines, administration nodes, computing nodes, and so on. Unlike enterprise applications, cloud performance optimization has many aspects under consideration. First, from a macro perspective, multiple applications in the cloud data center are running at the same time. Optimizing for a single software application is likely to prove suboptimal for the entire cloud computing environment. Second, applications running on virtual machines are tenants on the same computing nodes, and consequently share memory bandwidth, cache, and network. Mixing of workloads and using the same shared resources causes the problem of noisy neighbors and performance degradation. Third, from a micro-scope perspective, characterizing individual applications running on each virtual machine or individual applications running on multiple virtual machines requires processing and analyzing performance from multiple systems. Within an individual application, analyzing the correlation of events happening on one system versus another adds another layer of complexity. Last but not least, with the complicated relationship between all the components in the cloud, making a rapid data-driven cloud management decision is almost impossible without processing and deriving useful information from a large cloud performance dataset. To meet industry standards for providing a rapid and efficient management decision, a data-driven approach to deliver immediate analysis of the cloud and providing analytical intelligence of the cloud metrics is needed. Examining large cloud datasets and uncovering hidden patterns is not easy. Yet, we hope to influence the cloud computing industry should it be able to discover such patterns, identify solutions to satisfy user experience and translate these learnings into blueprints that enable more efficient cloud management solutions.

In [1], we describe a process, implemented in software, to assess the quality of cloud performance data. This process combines performance data from multiple machines, spanning across user experience data, workload performance metrics, and readily available system performance data. The data collection and processing procedure across the system is able to generate concrete data for posterior analytics. Principled statistical analysis on cloud performance data will serve as a valuable tool to assess cloud performance. One challenge of dealing with performance data sets is bringing them into a tidy data format [10] for analysis. In [1], the process we proposed addresses this challenge and prepares the collected cloud performance data readily for data-driven analytics.

In this paper, we discuss additional challenges for cloud computing. We describe these challenges from the aspects of computing environment, data collection, performance analytics and production environment. The value of understanding the challenges helps us seek better solutions to improve cloud performance analysis.

In Section 2, we review different types of cloud performance data and how our previous process [1] enables cloud performance data cleansing. In Section 3, we discuss the challenges for cloud computing from the analytics perspective and production perspective. Section 4 summarizes our paper and discusses future directions for this area of investigation.

## 2 Cloud Data Diversity

In this section, we describe the diverse range of cloud performance data, acknowledge the challenges for processing the data, and review our proposed approach [1] on assessing the quality of cloud performance data.

User experience data, such as throughput and response time, can be obtained from load driver systems typically used in software testing. One such load driver, Faban [4], is a driver development framework used in SPECjEnterprise [5], SPECvirt [6] and SPECsip [7] benchmarks, and can control a number of load generation nodes. Such a framework defines operations, transactions and associated statistics collection and reporting. Faban is one of the tools that can be used for cloud computing workloads.

System Activity Report (SAR) [8] on Linux systems can save system performance data such as CPU activity, memory/paging, device load, network, etc. SAR logs the contents of selected cumulative activity counters in the operating system. The accounting system, based on the values in the count and interval parameters, writes information a specified number of times spaced at the specified intervals in seconds. Performance Counters for Linux (Perf) [9], a Linux profiling tool for performance counters, can add additional detailed performance counters for software processes and microarchitectures.

### 2.1 The challenges in preparing cloud performance data

One challenge of dealing with performance data sets is bringing them into a tidy data format [10] for analysis. Before discussing these challenges, we first offer motivation for why such an action is desired. Cloud workloads are much more complex than enterprise workloads. Manually examining the cloud performance data is not feasible. The workload data must be restructured to allow statistical modelling. Dynamic analysis on time series cloud performance data provides new insights for cloud performance optimization. Techniques such as model selection, non-stationary time series analysis, and stochastic processes can be adopted to analyze the cloud workload. Furthermore, one could also borrow the techniques such as spectral clustering [11], sparse representation classification [12] [13], vertex nomination [14] [15] and graph matching [16] [17] in the field of random graph inference to model the user behavior graph of the cloud workload. All these inference frameworks and methodologies would benefit from a tidy data format. Only then can data analysis yield insights on the dynamics of cloud performance. As different performance tools generate data in different formats, parsing the data into a simple and coherent format is just as essential as collecting and analyzing the data. Moreover, validating the quality of the data processing procedure is necessary; otherwise all subsequent work can be suspect. In this section, we first start with a primitive and idealized scenario for quality assessment the data processing. We then define the challenges in identifying performance bottlenecks in the cloud, and present our insights on dealing with and working around these challenges.

There exists no standard format in which the data are collected. Still, we posit that parsing the data into a simple and coherent format is equal in importance to the effort involved in collecting and analyzing the data. Data munging gets tedious when dealing with data sets collected from different sources. Merging these files across the date and time spectrum is a particular challenge. Elaborating on this, various data sets might have different formats of time in which the data are collected, namely the a) world clock time and b) epoch (UNIX time), which might further differ in the following criterion: i) time zones, ii) units of measurements (seconds or milliseconds), iii) the time interval at which the data is collected and iv) the frequency at which data samples are collected. Cloud performance data sets are simply too huge to manually check consistency of results produced by the software components.

## 2.2 Review of our previously proposed approach

Here we present a brief review of our previously proposed approach [1] for assessing the quality of cloud performance data. To overcome these challenges, our approach incorporates multiple layers of quality assessments. Our goal is to assess the quality of the processed data for further analysis. In a nutshell, we first add a set of basic queries and scripts to check the correctness. As data can vary greatly, we then add a set of performance models to check the quality of the performance assessment. By using ensembles of performance models, we minimize potential errors in the software performance assessment tool chains. Figure 1 depicts the flow of our proposed software. In the following subsections, we describe in detail how our software works.

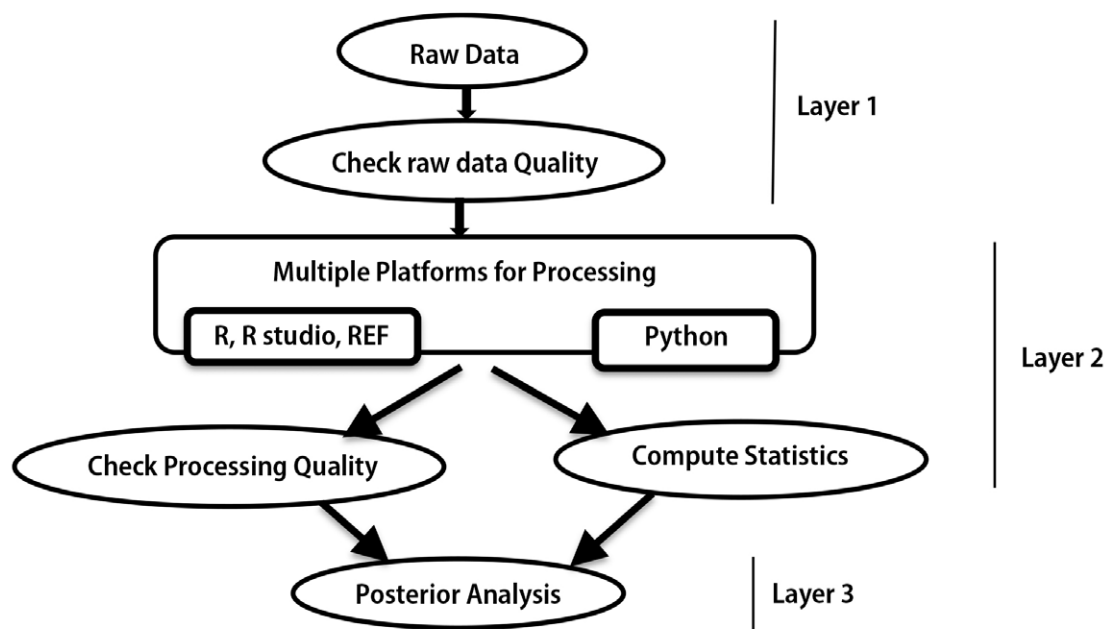


Figure 1. A quality model for Cloud performance analytics

## 3 Identifying the Challenges for Cloud Performance

### 3.1 Challenges of Cloud Computing Performance Guarantee

Cloud computing offers the promise to efficiently share resources among tenants and their workloads but verifying this aspiration and searching for improvement and optimization opportunities for the cloud administrator is not simple. The resources we first focus upon might be categorized into compute, memory, network and storage. In the case of Infrastructure-as-a-Service (IaaS), the cloud service provider may dedicate an amount of virtual CPUs (vCPUs) along with access to a fixed amount of physical memory and temporary working disk, based on the customer service request. Storage needs are often serviced a la carte.

One might think that the pure compute resources are immune to noisy neighbors but today's virtualized infrastructure employ multi-core processors potentially execute multiple threads per core and allow the hypervisor great flexibility in provisioning VCPUs. Many cloud service providers (CSPs) use Intel Xeon CPUs with Hyper-Threading Technology which allows two threads to process simultaneously. To the operating system each processor core has two virtual CPUs (vCPUs). The hypervisor can create a virtual machine (VM) with a configurable number of VCPUs. Because many VMs are often placed on a single server, the VCPUs for a single VM will often reside next to another instance or perhaps none at all.

The cloud performance analyst must be cognizant of the VCPU placement and adjacent VM and VCPU activity as compute resources on any individual VCPU can flex higher if the potentially adjacent VCPU is absent or idle and available. This complexity will manifest in slightly variable compute resources to the VM one level up.

For each physical server, the memory subsystem is shared among administrative and customer instances with some complexity. By default, memory cache and memory bandwidth are shared among virtual machines within a socket. Workloads differ greatly in the cache capacity required and the frequency of physical memory access. As such, VM instances memory caching and bandwidth requirements dynamically modulate memory resources available to neighboring instances while themselves being impacted by neighbor demands. We believe that being able to monitor and perhaps control memory activity will yield understandings that can positively influence cloud orchestration and VM density by way of smarter VM placement and memory control. It should also yield insight into the system memory requirement most appropriate to workload types.

The computing instance also attaches generally to networking services that are shared by other instances at various levels as you move up in the network hierarchy from the NIC to the initial switch and onward. Networking is typically less described to the customer and is most often shared among other instances on the server, within the data center and beyond onto the Internet. As to the network, a primary concern for the cloud operator to ensure that the network does not impede the quality of service or degrade the customer experience. But, to know if and when the network is a bottleneck to workload performance, and to understand where in the network the constraint lies, a rich and thoughtful data collection across the data center is required. The cloud analyst must be aware of the network topology and its capabilities. We may then instrument the monitoring of the local virtual and physical NICs as well as individual and aggregate use of switches throughout the datacenter.

Because cloud storage is very often delivered over the network, storage is not immune to noisy neighbor effects. A VMs I/O performance is subject first to all the networking forces mentioned above. The I/O devices a VM sees are likely to be virtual devices that route to a physical device over the network. The cloud analyst must be able to map physical devices to the VMs in use. Physical devices space will also be shared among users so concurrent use of the physical device also injects noisy neighbor concerns.

We should not close this section before bringing attention to another shared infrastructure common to virtualized systems are those resources set aside or used by the virtual machine manager and the hypervisor. It is common for a CSP to reserve a fixed number of VCPUs per physical server for local virtualization management purposes. These VCPUs may well be pinned to a set of physical cores perhaps not available for customer subscription. Clearly a CSP would want to minimize the non-revenue compute available yet provide enough resources for local orchestration and execution. In the case of Xen, the initial domain, known as Dom0 is a privileged domain that starts and manages those hosted on the server. Cloud operations will need to monitor the compute, memory, networking and I/O use on this VM instance along with the customer VMs. This is especially important when the burden of hosted VM networking or I/O puts demand on compute processing or if networking is transparent to customer VM as might be the case when for I/O (e.g., for paravirtualized guests in Xen 4.3). Monitoring here can help cloud operations as they strive to expose the indirect resource demands the customers place in order to improve both billing and architecture practices.

## **3.2 Challenges of Collecting Performance Data in a Cloud environment**

Collecting performance data in a distributed environment is challenging in many ways, a few of which we will highlight in this section. Firstly, in a cloud environment, we have number of virtual machines residing on the same or different hosts talking to each other over a common network.

Synchronous data collection in such environments is very difficult. It would require configuring and deploying a complex software application which would enable synchronized automation of performance

monitoring tools on all the machines. Enabling of such software systems requires automation of secure communication in the cluster. Setting up secure connections over networks involves much tedious effort like creating secure keys, users, and links, unique subnets and IP address, bridges on VMs, etc. Even one error in the setup can break the connection and will result in incomplete and inaccurate data.

Secondly, the version of the tools we use have a lot of dependencies on the operating systems, kernel, hardware specifications etc. So to call out in simple words, selecting a single performance monitoring tool that can collect all the data we need from all machines in our cluster could be challenging. For example, the open source performance monitoring tools, like sar and perf are exclusive to UNIX/LINUX-based operating systems, and not for WINDOWS. This compels us to use multiple performance monitoring tools, adding to the complexity of the VM environment before starting performance monitoring tools.

The real challenge comes in the phase of data interpretation. Every tool, has its own way of computing results—keeping aside the data formats (time notation, units, frequencies, etc.), the way in which each value/data point is calculated could vary vastly from tool to tool. By example, suppose tools A, B and C report data every 10 seconds on three different machines performing inter-dependent functions, Tool A takes an average of the data observed in the last 10 seconds and gives a value, while tool B, reports whatever it measures at 10<sup>th</sup> second and Tool C reports data that it measured for the 5<sup>th</sup> second of the 10s interval. Now when we put the data together after an experiment, in a time-series sheet to analyze the performance and resource utilization of our machines during our experiment, the data could not support consistent analysis or comparison.

The right approach would be to take in account the internal computation of every counter of every tool we use to monitor data and adjust our data accordingly before doing further analysis. This in itself is a huge challenge and lot of hard work and planning.

### 3.3 Challenges of Analytics on Cloud Performance Data

The cloud performance data include both event and sample-based data logs. Putting the time series of performance data back to the same scale becomes challenging. Even though the sampling intervals for different performance monitoring tools may be the same, the actual timestamp for each type of performance data might not lie on the exact same timestamp as another. Therefore, by merging the time series data as described in [2], a lot of missing values introduced for unmatched points in time. The presence of missing data can cause challenges for correlation analysis, because the mathematical formulation requires each variable hold a value for each observation to be included in the correlation. We have found that appropriate missing value imputation can address this challenge and comes to rival in importance any subsequent statistical analysis methodology. There is a vast of literature describing different techniques of missing value imputation. For instance, one can choose to impute the missing values by the mean, median, maximum or minimum. Or one can also impute the missing values by interpolation over the previous and subsequent observed values. In cloud performance analysis, the choice of missing value imputation techniques depends on the performance metrics. For example, garbage collection (GC) are event-based and occurs when resources need to be reclaimed, while system activity report (SAR) data are sample-based and occurs at fixed length of time intervals. When merging garbage collection logs and system activity report data, blocks of missing values are introduced. However imputing the missing values for different performance metrics requires contextual knowledge. When imputing the heap size of GC logs, imputing the missing values by zero makes sense, because at the missing timestamps, no GC activity occurred. On the other hand, one could use linear interpolation for imputing many SAR performance metrics. If taken during the steady state of a workload, one could use the mean or median to impute the missing values for SAR performance metrics. It is indeed a challenge to identify the optimal missing value imputation techniques for performance analysis.

Another challenge for drawing inference on merged performance data lies in data transformation. The need for data transformation becomes essential when we want to spot patterns and detect changes across performance metric types. CPU utilization lies between 0-100%, network activity is measured by

kilobytes per second often the order of  $10^6$ , while cache misses per thousand instructions could be in the order of  $10^{-3}$ . Plotting such metrics on a common scale cannot expose the behavior changes, because the very low valued performance metrics will display as almost a straight line. There are many ways of data transformation such as centering, data normalization, log transformation, inverse transformation, etc. Different types of data transformation will illuminate (or potentially mask) different patterns of behavior changes. Furthermore, it will result in a different result of correlation analysis. Therefore, providing a principled methodology of transformation is a necessary if difficult task.

The complexity of cloud computing environment is beyond manual inspection. When merging cloud performance data, the number of performance metrics, or statistically speaking, the number of variables, is usually in the thousands. Manually inspecting which performance metrics are more relevant to each other is virtually impossible; to infer causal relationship among the performance metrics is doubly so. Therefore, we find examination of correlation analysis results is an efficient way in the early journey to identify relevant metrics. Directly calculating the correlation between two performance metrics is not sufficient, because the data has inherent temporal effects. We propose using cross-correlation analysis, which considers the time lagging effect to assess the correlation between performance metrics. Although this approach is more suitable to identify correlated time series, it is challenging to determine how wide or narrow the time windows should be for identify the temporal correlation.

## 4 Related Work

Bianque is a software quality performance monitoring system deployed in Alibaba. It enables performance data collection from thousands of servers. As such, it can be viewed as a software quality performance analysis platform. To aid software quality analysis, it can extract performance profiles from the servers. From those profiles, developers can identify source code that may be performance bound. Operators and software testers can also benefit from such performance monitoring process.

Performance profiling presents the time spent by the software application. It breaks the usage into components such as processes, modules and functions. Hotspots are regions of source code that consume significant amount of system resources. Thus, identifying hotspots can help developers locate the regions of code (bottlenecks) that they can work on to optimize the software application.

Linux executables following the Executable and Linkable Format (ELF) contains a symbol table known as “symtab”. This section contains the description of each method, including its length and starting instruction address. Thus, it provides sufficient information to translate an instruction pointer to the function. If we also have “Debug info” in ELF, we can translate it to the source code too.

Using statistical sampling, we can estimate the utilizations of each function in the software application. Statistical sampling is also used by other tools such as VTune. To efficiently deploy Bianque into thousands of servers in a data center, we separate it into three roles.

1. Daemon Server – to collect samples and provide storage for the data. It is important this task has low overhead as it is running all the time.
2. Binary Server – to resolve symbols and analyze data. This task is triggered only when a user queries the system. Also this task does not run on the customer facing server. So the overhead can be a higher than that of the daemon server.
3. Builder Server – to store symbol tables in ELF files. This is essentially a storage server keeping tracks of different OS versions and symbol tables running in the servers. It provides storage for both the daemon and the binary servers described above.

## 5 Summary and Discussion

In this paper, we identified challenges to understanding cloud computing performance from the aspects of computing infrastructure, data collection, data analytics and production environment. While solving all these challenges is not currently feasible, understanding these challenges provides us directions in terms of proposing frameworks for cloud computing characterization. An analytics framework is in need to discover the hidden patterns within cloud performance data and thus extract insights for improving software performance in cloud computing. This framework shall utilize time series performance data including user experience data, software performance metrics and system performance data. Needless to say, due to high complexity in cloud computing environment, characterizing the cloud data center becomes fundamentally different from characterizing stand-alone enterprise workloads. Moreover, there are many aspects of cloud computing open to further optimization, and optimizing with respect to one single objective is not realistic. A joint optimization solution for software performance in cloud computing must be sought. When looking into the intersection of cloud computing, performance analysis, mathematical programming and machine learning, we notice a systematic solution for cloud performance enhancement could be born from there. A cloud computing characterization approach that combines performance engineering domain expertise and the rich data attainable from the cloud with advanced data science techniques is essential if we hope to generate robust conclusions.

## Acknowledgments

The authors wish to express their gratitude to their reviewers, Rick Anderson and Sue Bartlett, whose assistance and feedback have been invaluable.

## References

- [1] Chen, Li, et al. "Brewing Analytics Quality for Cloud Performance." The Proceedings of 33rd Annual Pacific Northwest Software Quality Conference (PNSQC) 2015.
- [2] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg*. (2011).
- [3] Khun Ban, Robert Scott, Huijun Yan and Kingsum Chow, "Delivering Quality in Software Performance and Scalability Testing", *Pacific Northwest Software Quality Conference*, October 2011, Portland, OR. [http://www.uploads.pnsrc.org/2011/papers/T-40\\_Ban\\_et\\_al\\_paper.pdf](http://www.uploads.pnsrc.org/2011/papers/T-40_Ban_et_al_paper.pdf)
- [4] Zhidong Yu, Hongjie Wu, Da Teng, Zhengzhu Xu, Ethan Huang, Peng-fei Chuang, Tony Wu and Kingsum Chow, "Classifying Enterprise Applications using Computer System Performance Statistics", *The 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*, Shanghai, China (July 11-12, 2011)
- [5] <https://java.net/projects/faban/>
- [6] <http://www.spec.org/jEnterprise2010/>
- [7] [http://www.spec.org/virt\\_sc2013/](http://www.spec.org/virt_sc2013/)
- [8] <http://www.spec.org/sipinf2011/>
- [9] [https://en.wikipedia.org/wiki/Sar\\_\(Unix\)](https://en.wikipedia.org/wiki/Sar_(Unix))
- [10] [https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page)
- [11] Wickham, Hadley, "Tidy Data", under review. 2014. <http://courses.had.co.nz.s3-website-us-east-1.amazonaws.com/12-rice-bdsi/slides/07-tidy-data.pdf>
- [12] Ng, Andrew Y., Michael I. Jordan, and Yair Weiss. "On spectral clustering: Analysis and an algorithm." *Advances in neural information processing systems* 2 (2002): 849-856.
- [13] Chen, Li, Cencheng Shen, Vogelstein, Joshua, and Priebe, Carey. "Robust Vertex Classification." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, accepted for publication 2015.
- [14] Shen, Cencheng, Li Chen, and Carey E. Priebe. "Sparse Representation Classification Beyond L1 Minimization and the Subspace Assumption." *arXiv preprint arXiv:1502.01368 (2015)*. Under review in the *Journal of the American Statistical Association*.
- [15] Fishkind, Donniell E., Vince Lyzinski, Henry Pao, Li Chen, and Carey E. Priebe. "Vertex Nomination Schemes for Membership Prediction." *arXiv preprint arXiv: 1312.2638. Annals of Applied Statistics* (2015).
- [16] Chen, Li. "Pattern Recognition on Random Graphs." *PhD Thesis, Johns Hopkins University* 2015.
- [17] Conte, Donatello, Pasquale Foggia, Carlo Sansone, and Mario Vento. "Thirty years of graph matching in pattern recognition." *International journal of pattern recognition and artificial intelligence* 18, no. 03 (2004): 265-298.
- [18] Lyzinski, Vince, Daniel L. Sussman, Donniell E. Fishkind, Henry Pao, Li Chen, Joshua T. Vogelstein, Youngser Park, and Carey E. Priebe. "Spectral clustering for divide-and-conquer graph matching." *arXiv preprint arXiv: 1310.1297. Parallel Computing* (2015).
- [19] R Core Team (2015). R: A language and environment for statistical computing. *R Foundation for Statistical Computing*, Vienna, Austria. URL <http://www.R-project.org/>.
- [20] Studio, R. "R Studio: integrated development environment for R." (2012).



# When Continuous Integration meets Application Security

Vasantharaju MS, Harish Krishnan

vasantharaju.ms@intel.com, harish.krishnan@intel.com

## Abstract

Continuous Integration (CI) is not specific to extreme programming anymore, and most organizations which follow any form of Agile practice for product development adopt CI. The two major benefits that organizations derive from CI are build verification, and test automation. Test automation in CI usually covers Unit Testing, Functional Testing, Integration Testing, etc. Every check-in a developer makes is tested automatically to make sure the build is verified and functional. But what about the security of the product?

In most cases, security testing is not automated, but performed manually on a milestone build or, worse, at the end of project. As the number of companies offering Software as a Service (SaaS) grows, and reliance on a Continuous Deployment or Continuous Delivery pipeline increases, every change can be potentially deployed to production and there is an increased risk to companies in terms of application security. There is a pressing need to do security testing more often or, better yet, on every change.

What if you could run security test cases, security scanners and other security related build verifications during CI? This is where Continuous Integration meets Application Security. For example, static code analysis with security checkers or a defense to specific vulnerabilities like Blind SQL Injection, Cross Site Scripting (XSS) and other vulnerabilities can be verified in integration testing or an audit for vulnerable 3rd party libraries can be conducted on top of build created by CI. Automating security testing leverages the integration testing platform provided by CI to ensure application security. Of course, we can only deliver secure solutions as fast as we can test them, adding security testing to your existing CI capabilities will help in achieving this goal.

This paper covers the processes and tools required to automate security testing in Continuous Integration based on our learning and experience.

## Biography

*Harish Krishnan is a Product Security Champion at Intel Security Group with over 10 years of experience in Quality and Security domain. His primary responsibility is Product Security. His daily duties include performing security audits, architecture design review through Threat Model, Privacy review, Penetration testing, analyzing vulnerabilities and writing security Bulletins. He has also authored an in-house proprietary web application security scanner tool which many Intel security enterprise products run regularly to uncover any vulnerabilities.*

*Vasantharaju M.S. is a Senior Software Development Engineer at Intel Security Group with over 8 years of experience in developing enterprise-class software. His past experiences include; designing and implementing back-office solution to address a Dodd-Frank clause, enhancing web crawler and vulnerability detection in SaaS based vulnerability scanner and executing multiple Proof-of-Concepts for enhancing security management platform. He has a keen interest in defensive programming.*

# 1 Introduction

Continuous Integration (CI) is a software development practice of merging all developer working copies to a shared mainline several times a day. Each iteration is verified by an automated build along with running several automated product feature tests to detect integration errors as quickly as possible.

There are several CI platforms available out there. In this paper we will be sharing our learning and experience working with JetBrains TeamCity [1] CI server.

Application security, in simple terms, is testing the security of an application, which can be achieved to some extent by running tests using a few security tools that are available. Our experiences that we are sharing in this paper include both commercial and free security tools. We use the following tools, which are described in much detail in later sections, to ensure the security of our Java based Web Application.

1. Static Analysis: Coverity [2] and FindBugs [3]
2. Network and Dynamic Analysis: Nessus [4] and some proprietary scanning tools
3. 3rd party component security audit: OWASP Dependency Check [5]

Though a few of the tools mentioned above are specifically for use with Java based applications, the process we describe can still be leveraged using alternative tools applicable to software applications built using a different technology stack.

When such tools are automated and are run as part of CI, then this is where Continuous Integration meets Application Security, detecting security issues as early as possible in the product development life cycle. When any security issue is found as part of CI tests run, then the build in TeamCity will be marked as failed indicating that the new code that was merged introduced this issue.

The benefit of having such CI setup for security testing is that many vulnerabilities are found and fixed early in the product development life cycle. Fixing issues earlier in the product life cycle is much easier and less costly in terms of rework needed and the number of engineers involved, than towards the end of project thereby saving us lot of time. Also during the product release, this gives us more confidence on the overall security of the product.

The paper is structure as follows. Section 2 talks about Static analysis, how to enable only security checkers in the FindBugs static analysis tool and how to run the tool. It also provides typical build steps needed for CI. Section 3 describes the Nessus scanner, a few examples on Nessus automation and again the typical build steps needed for CI. In Section 4, we mention some of the generic Web Application vulnerability scanners and the general build steps needed to integrate with CI. Lastly, in Section 5 we talk about how we perform the third party component security review.

## 2 Static Code Analysis with Security Checkers

Static code analysis is done using automated tools to examine source code that is instrumented and built to some form of object code without actually executing the program. Static code analysis tools are available for different programming languages like Java, C++ and also provide plugins for many IDEs (Integrated Development Environment), so that developers have a chance to run static code analysis during the development phase itself before the code is checked-in into the source repository. These tools find both quality and security issues in the code, but our focus here is only on security issues.

It is highly recommended to have a separate build configuration that runs static code analysis dedicated to running security checks. This separate build configuration can assure the software development team that for every check-in, the code is free from security flaws or malicious code. Additionally, when many

such security flaws are identified during this CI build, it will eventually encourage the developers to use plugin version of static code analysis tools to help them find such issues before the code is checked-in.

We use two static code analysis tools. Coverity, one of the commercial static code analysis tools and the other one is FindBugs, a very popular free tool for java static code analysis.

## 2.1 FindBugs

FindBugs is a free program which uses static analysis to look for bugs in Java code.

### 2.1.1 Configuring FindBugs to report only security issues

There is a useful feature in FindBugs called 'Filter files' using which the user can configure to include or exclude a particular bug instance/category in the report. A filter file is an xml document and here is an example filter file to report only items that match an issue category of 'security' when FindBugs static analysis is run:

```
<?xml version="1.0" encoding="UTF-8"?>
<FindBugsFilter>
  <Match>
    <Bug category="SECURITY"/>
  </Match>
</FindBugsFilter>
```

### 2.1.2 Running FindBugs

FindBugs static analysis tool can be run in multiple ways, either using the application GUI, the command line interface or using an Ant [6] task.

We run this using an Ant task method because TeamCity supports the Ant scripts as one of the build steps. Also, most of our other TeamCity build steps are Ant scripts so this way of running was our first choice to maintain the same pattern. Following is a sample Ant build script for running FindBugs static analysis:

```
<project name="FindBugs" default="findbugs" basedir=".">
  <taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.
  FindBugsTask"/>
  <property name="findbugs.home" value="${basedir}/findbugs-3.0.1" />
  <target name="findbugs">
    <findbugs home="${findbugs.home}"
      output="xml"
      outputFile="result.xml"
      includeFilter="${basedir}/secFilter.xml" >      <!--this is the above
      created filter file to include only security issues ->
      <sourcePath path="${basedir}/src" />
      <class location="${basedir}/lib/*.jar" />
    </findbugs>
  </target>
</project>
```

Store the above xml in 'build.xml' file and run the following command on the command prompt from the directory where this 'build.xml' is stored:

```
C:\> ant findbugs
```

## 2.2 Typical build steps needed for CI

- Checkout the code and build.
- Run Static code Analysis with security checkers enabled as shown above.
- Automatically analyze the scan result. Following are the details on how we achieve this -
  - The scan report is generated in xml format as we instructed in the ant build.xml file above. Also the report includes only security issues, if any.
  - Programmatically we parse this xml file and look for any bugs being reported. A typical FindBugs report includes a <BugInstance> element for every issue it finds. Following is an example from the report:

```
<BugInstance category="SECURITY">
  <Class classname="">
    <SourceLine classname="" sourcepath="" sourcefile="" end="" start=""/>
  </Class>
</BugInstance>
```

- Finally, mark the build PASS/FAIL based on the analysis done in the previous step i.e. fail the build if any bug instances found in the report file.

## 3 Network Scanners – Nessus

Nessus is a popular and widely used vulnerability assessment solution. This tool has a wide variety of tests to choose from and we mainly use it for Web Application testing. Writing a basic framework to integrate with Nessus API's to automate Nessus scanning provides an abstraction layer making it easy to perform scanning related operations. For our application, we used nessrest [7] framework to integrate with Nessus REST API's.

### 3.1 Example Usage of this python framework

Here are few examples using the APIs provided by the nessrest framework on how to connect to the Nessus server, how to run the scan and how to download the report. You can see with this framework now, how easy it is to connect to a Nessus server.

```
# Connecting to Nessus Server
from nessrest import ness6rest as nes
URL = 'https://' + NESSUS_HOST + ':' + NESSUS_PORT
log("Connecting to Nessus Server: %s" % URL)
try:
    # Insecure is set to True for allowing self-signed certificates.
    scan = nes.Scanner(url=URL,
                       api_akey=NESSUS_ACCESSKEY,
                       api_skey=NESSUS_SECRETKEY,
                       insecure=True)
    log("Successfully connected to Nessus")
```

```

except Exception, e:
    log("Could not connect to Nessus Server: %s" % str(e))
    sys.exit(1)

# Running the scan
scan.policy_set(POLICY)
scan.scan_add(targets=TARGETS, name=scan_name)
log("Running the scan....")
scan.scan_run()

# Downloading the scan report
content = scan.download_scan()
uid = scan.scan_uid
output_path = os.path.join(os.getcwd(), 'reports', uid + '.nessus')
f = open(output_path, 'w')
f.write(content)
f.close()
log("Download complete - %s" % output_path)

```

## 2.2 Typical build steps needed for CI

- Checkout the code, build and deploy the application.
- Run the automation scripts written on top of the above python framework to trigger the Nessus scan.
- Once the scan is complete, automatically download the reports using the automation scripts.  
(Note: We store these reports in TeamCity server under Artifacts)
- Automatically analyze the report. Following are the details on how we achieve this -
  - Other than providing the APIs to download the scan report, the framework also provides an API for parsing the downloaded report.
  - The Nessus report lists vulnerabilities with following severities – Critical, High, Major, Low and Informational. You need to decide what type of severity issues you are interested in, so based on that you can PASS/FAIL the build.
  - We ignore Informational issues. Examples of Informational issues reported are, Hyper Text Transfer Protocol (HTTP) information, Self-Signed SSL (Secure Socket Layer) certificates are used, etc.
  - Here is an example snippet (not an exact working code) on how we achieve this –

```

report = PyNessusFramework.Report()

# parsing the downloaded report.
report.parse(downloadedReportFile)

# getting the target on which the scan was run.
target = report.targets[0]

# getting the list of vulnerabilities found on that target.
vulnerabilities = target.vulns

# creating empty list to store severities of the vulnerabilities.
severityOfVulnerabilities = []

# populating the list with severities of the vulnerabilities
for i in vulnerabilities:
    severityOfVulnerabilities.append(i.get('severity'))

# initial status

```

```

vulnerabilityFound = False

# If any vulnerability with severity >= Medium (i.e. med (2), high (3),
critical (4))
if ('2' in severityOfVulnerabilities) or ('3' in
severityOfVulnerabilities) or ('4' in severityOfVulnerabilities):
    vulnerabilityFound = True
    log("VULNERABILITY FOUND = %s" % vulnerabilityFound)
else:
    log("VULNERABILITY FOUND = %s" % vulnerabilityFound)

```

- Finally, mark the build PASS/FAIL based on the analysis done in the previous step, i.e. fail the build if “VULNERABILITY FOUND = True” message is logged in the build logs.

## 4 Web Application Vulnerability Scanners

Web Application Vulnerability Scanners are tools that automatically scan web applications for known security vulnerabilities like cross-site scripting, SQL injection, cross-site request forgery, directory traversal, etc. Web Application Vulnerability Scanners can be configured to automatically crawl the web application whilst authenticated or unauthenticated and scan all discovered links and/or forms for vulnerabilities. There are many Web Application scanners that are available both in commercial and free versions.

Some of the well-known commercial web security scanners are Rapid7 [8], Burp Suite Pro [9], HP WebInspect [10] and Acunetix [11]. Some of the free utilities also available are OWASP ZAP [12], Nikto [13] etc.

We have not integrated any of the above tools; instead, we use a proprietary web application security scanner to test our product. Like many other tools, even our proprietary scanner provides APIs so that scanning operations like triggering a new scan, downloading of reports, etc. can be automated using scripts.

We have automated the security tool in the PowerShell scripting language, which is supported by the TeamCity build steps.

Once we have an automated security-scanning tool, as mentioned in previous section, the following are the typical build steps for CI.

### 4.1. Typical build steps needed for CI

- Checkout the code, build and deploy the application
- Run the automated security scanning tool which will crawl the application links and analyze for vulnerabilities
- Automatically analyze the report similar to the previous section.

Finally, mark the build PASS/FAIL based on the analysis done in the previous step.

## 5 Third Party Component Security Review

Third party component security review involves identifying project dependencies and reviewing known or publicly disclosed vulnerabilities for identified project dependencies. Reviewing the software for third party components for known or publicly disclosed vulnerabilities is typically done in a milestone build and

removing a third party library or framework after triaging published vulnerabilities involves a considerable amount of time and effort. The major challenges in reviewing the third party components for known vulnerabilities are;

1. Identifying third party libraries, modules and frameworks and their corresponding version included in the software
2. Triaging published vulnerabilities for identified third party components

OWASP Dependency-Check tool automatically identifies project dependencies and reports any known or publicly disclosed vulnerabilities for the identified project dependencies. Currently Java, .NET, Ruby, Node.js, and Python projects are all supported by OWASP Dependency-Check. Having a build configuration in CI to run OWASP Dependency-Check helps to review project dependencies for known or publicly disclosed vulnerabilities and also helps in preventing vulnerable third party component to be added as project dependency.

Kicking off the build for every check-in would not be necessary since 3rd party libraries are not checked in frequently so kicking off a build for a check-in which includes 3rd party components will be more efficient and CI tools have provisions to trigger builds based on specific files based on Regex like \*.jar in case of java project. Scheduling a nightly build also helps in finding recently published vulnerabilities for third party components which are already white listed, since there can be new CVE added for a third party component.

Triaging vulnerabilities published for known components certainly needs application security expertise to determine possible attack vectors and take a call to suppress the finding or remove the third party component and looking for alternatives. In the case of suppressing findings due to false positive results or ignoring for a valid reason after triaging, Suppression.xml can be used as below – this suppresses specific CVE (Common Vulnerability and Exposures) for a tomcat library catalina-ha.jar

```
<?xml version="1.0" encoding="UTF-8"?>
<suppressions xmlns="https://www.owasp.org/index.php/OWASP_Dependency_Check_Suppression">
  <suppress>
    <notes><![CDATA[
      file name: catalina-ha.jar
    ]]></notes>
    <sha1>d68cfd77bd369975cf4e51b07b0d66707a1af656</sha1>
    <cve>CVE-2013-2185</cve>
  </suppress>
</suppressions>
```

OWASP Dependency Check reports have provisions to create the suppression xml dynamically based on the requirements so there is no need to create the suppression xml manually. After creation, this suppression xml is used to suppress findings in subsequent scans.

## 5.1 Typical build steps needed for CI

- Checkout the code, build and package.
- Explode the package and copy third party libraries like Jar/DLL into separate directory

- Run Dependency Checker tool to analyze third party libraries for published vulnerabilities.
  - Dependency-Check tool can be run in many different ways, one way is to run OWASP Dependency-Check Ant task as part of ant build script.
- Analyze the build report for failure conditions. For example OWASP Dependency-Check ant task can be configured to fail the ant task call if CVSS of finding is more than 4 or any number ranging 0-11.
- Finally, mark the build PASS/FAIL based on the analysis done in the previous step.

## 6 Conclusion

Most of the time security testing is performed manually on a milestone build or towards end of the project. If any vulnerabilities are uncovered later in the project, then it is very expensive to fix the issue as the product is already built to some extent, thereby consuming more time to resolve the issue.

As one can see from reading the information presented, automating security testing and then running those tests as part of a continuous integration workflow, many vulnerabilities can be found and fixed early in the product development life cycle, which is much easier than towards the end of project, thereby saving us lot of time and expense.

With around 10 - 12 weeks of efforts from 2 persons, we were able to build this entire CI system for security testing. Some of the immediate benefits we saw by having this setup were:

- a) By performing a 3rd party security audit regularly as part of CI, new CVEs (Common Vulnerability and Exposures) that get published were tested almost every day. Around a dozen libraries that we were using was found to have some vulnerabilities and we fixed them immediately before it was too late.
- b) In the past, we used to get many customer escalations on some security issues, which they found by running similar, or the same security tools on our product. This has been reduced by 95% as we are also proactively running these tools and fixing all the legitimate issues.
- c) As part of performing static analysis, we have uncovered and fixed hundreds of resource leaks in our product, which when present, can lead to even DOS (Denial of Service) vulnerability in the product when all system memory is used up.

The trickiest part of running security tools in CI environment was how to automatically analyze the scan report. Even though we did our best in automatically analyzing the report and marking the build status accordingly, we have seen around 1/25 times, manual assessment was needed to make sure the build was marked accurately.

Overall, by performing the Application Security testing as part of CI, we are confident that we are releasing a more secure product than before

# References

## Web Sites

- [1] TeamCity : <https://www.jetbrains.com/teamcity/>
- [2] Coverity : <http://www.coverity.com/>
- [3] FindBugs : <http://findbugs.sourceforge.net/>
- [4] Nessus : <https://www.tenable.com/products/nessus-vulnerability-scanner>
- [5] OWASP Dependency Check: [https://www.owasp.org/index.php/OWASP\\_Dependency\\_Check](https://www.owasp.org/index.php/OWASP_Dependency_Check)
- [6] Ant : <http://ant.apache.org/>
- [7] Nessus Frameworks
  - : <https://github.com/tenable/nessrest>
  - : <https://python-nessus.readthedocs.io/en/latest/>
  - : <https://github.com/Adastra-thw/pynessus-rest>
- [8] Rapid7 : <https://www.rapid7.com/products/appspider/>
- [9] Burp Suite pro: <https://portswigger.net/burp/>
- [10] HP Web Inspect: <http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/>
- [11] Acunetix : <http://www.acunetix.com/vulnerability-scanner/>
- [12] OWASP ZAP: [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- [13] Nikto : <https://cirt.net/Nikto2>



# How Architecture Decisions Solve Quality Aspects

**Anil Z Chakravarthy**

(Anil.Z.Chakravarthy@Intel.com)

**Charulatha Dhandapani**

(charulatha.dhandapani@gmail.com)

## Abstract

Software architecture is about making fundamental and crucial decisions on individual elements which could prove to be costly to change, once they are implemented. Solving quality aspects such as performance, reliability, and security as part of the software design can be daunting and challenging.

In this paper, we will dwell on some software architecture decisions and their impact on achieving quality software products.

- Single threaded vs multi-threaded programming
- Blocking calls vs non-blocking calls
- Reuse of the components (no need to reinvent the wheel)
- The technology we choose to solve the bigger problem
- Phoenix- kill the current design and redesign

## Biography

*Anil Z Chakravarthy is an Engineering Manager at Intel (McAfee now part of Intel Security), with more than eleven years of software development experience. Highly passionate about quality, he strives to lookout for ways and methods to improve software reliability and usability. As an inventor of key technology solutions, his interests include security management, content distribution and updating.*

*Charulatha Dhandapani is Software Engineer at Intel (McAfee now part of Intel Security), with more than ten years of software development experience. She is a very passionate about working on emerging and innovative technologies and always strives to find innovative ways to improve quality and reliability.*

*Copyright Anil Z Chakravarthy 10-Apr-2016*

# 1 Introduction

Quality aspects have a system-wide impact and hence their implementation should also be system-wide. Quality aspects are the factors which have impact on the runtime behavior, user experience, and security. These can be further classified into modularity, testability, maintainability, robustness, scalability, throughput, etc. One of the most challenging aspects of the software design is to create a system which has built-in quality aspects needed by the end user.

This demands that an architect have a great deal of understanding on the quality aspects, their impact on the system and approaches to satisfy these in the architecture of the system. Architecture patterns addresses various issues in software engineering with an impact on many quality aspects of the system.

In this paper we present our experiences from our architecture design decisions we had made to resolve some of the above quality aspects.

## 2 Architectural Decisions & Quality Aspects

### 2.1 Architectural Decisions

Architecture is the blueprint of software. Depending on the constraints and operational environment of the software, various choices about the software's functionality, about the implementations, and about capabilities are made by architects. Every application/software has different requirements and goals which need to be considered in the design phase. One of which is to identify the actions that the application is going to do in its lifetime and also most importantly identify the resources that the application would need to perform its action/job. Each of these choices has pros and cons which need to be considered to make a right choice at the design phase that would create the best possible outcome.

In this section, we list out some of these design decisions that software architects make. For each of these design decisions, we will provide a description of the decision and then dwell into identifying the quality aspects of the decision. We will finally share the decision we have made and the quality aspect we tried to address.

### 2.2 Quality aspects

In this section, we will list the quality aspects that can be addressed by the architectural decisions that are made. There are many trade-offs for each approach which deals directly with some of the quality aspects like

- Performance & concurrency
- Generating code
- Code maintenance
- Validation
- Porting
- scalability

The following sections talk about each architectural decision and the quality aspects it impacts.

## 2.3 Single threaded vs multi-threaded programming

Based on the requirements, an architect has an option to choose one of the below options in his / her architecture:

- single-threaded,
- large thread-pool or
- fixed thread count

### 2.3.1 Quality Aspect

If an application/software is meant to perform one action at a given time or if it is acceptable to queue the actions to be sequential, then we would ideally go for a **single threaded** application. This would by design resolve the traditional **synchronization/concurrency** problems that come with multithreaded applications/software.

Another quality aspect that we need to consider is **Generating code**. To have a cross platform application/software we need to write code to either have a wrapper which would abstract the threading APIs (platform specific code) and then start using these in the code. But, if an application is a single threaded application, we would avoid use of any threading modules, making it easier to develop platform agnostic code.

### 2.3.2 Example

In one of our Projects, we developed a single code base which runs on a variety of platforms. This presented a challenge to ensure we chose the right architecture and the language that can be built across these heterogeneous platforms. One of the areas that we identified was that our software didn't need multiple threads and all the actions can be performed sequentially. We adopted the single threaded programming approach and have seen in our experience that most of the issues related to concurrency, synchronization as already addressed by design. We also have benefited by not generating code for cross platform threading module, or using/including any 3rd party code which is common across platforms.

## 2.4 Blocking calls vs non-blocking calls

Traditionally in multi-threading programming we use locks to synchronize access to shared resources. The thread that attempts to acquire a lock that is already held by another thread, it will be blocked until the lock is released. Blocking a thread is undesirable for some of the below reasons:

- A blocked thread cannot accomplish anything.
- If the blocked thread had been performing a resource intensive task, those resources are now blocked from being used by other process threads
- Possibilities for deadlock; the app or process comes to a halt (effectively crashes)

### 2.4.1 Quality Aspect

Unlike the above, non-blocking calls do not suffer from these downsides. Non-blocking in conjunction with single threaded programming allows us to resolve these issues by design.

Asynchronous programming allows more parallelism. Such a programming model relies on mechanisms such as callbacks and events to trigger/initiate a task or signal task completion.

### 2.4.2 Example

An asynchronous programming model allows an application to respond to service requests that don't need to be entirely processed immediately. In this scenario, it is best suited for the user to just initiate the operation and get notified when completed. This resolves many of the quality aspects like **resource utilization**, **deadlock**, and unnecessary **wait**.

In a network application, where the user has to be notified when there is data available, we will need a mechanism to register for such a notification and go ahead with other operations. As and when the data is available the callback will be called and an appropriate action can be taken.

## 2.5 Reuse of the components (no need to reinvent the wheel)

In most business software applications, the workflow or the actual solution implementation can be the business logic and rest of the application can be implemented using standard infrastructure components.

### 2.5.1 Quality Aspect

Generating, building, or validating such common infrastructure components pose a lot of inherent quality aspects that must be considered. Below are a few:

- Need to build the expertise in generating/building/validating these components
- This is not the core area of the solution one would like to offer to their customers
- Stability issues as part of build these components in house
- Adhering to the regular updates that are published to these components
- Ensuring these components adhere to security and performance constraints

Instead when we reuse infrastructure components, we realize these benefits:

- Assured quality, as the reused component is written and delivered by SMEs (Subject Matter Experts)
- Adhering to standards is a given as these are developed by SMEs as per open standards
- Increased velocity in developing the core business logic one would want to deliver to their customer

### 2.5.2 Example

As a multiple platform software/application, we had multiple requirements like:

- Cross platform support (e.g.: Windows, Linux, Mac OSX, etc.)
- Need for a network module to perform network operations such as secure communication
- Packing/unpacking module to perform zipping/unzipping operations
- And so on

In order to meet these requirements, we either had to

- start designing/developing/validating these modules
  - This is not our core competency and also not the actual solution that we would want to deliver to our customers
- (or) re-use these infrastructure components developed by 3rd party in order
  - **Increase the velocity** of development of the product and also be **assured of the quality** which is already tried and tested in the field by many others.

Reuse of components helped **lower our costs (smaller team)** and **faster development** to deliver the product to the customer/market **on-time**.

For example: if we have to develop a network library, which handles HTTP, HTTPS, FTP, Proxy with all the authentication mechanisms like NTLM, Digest, Kerberos, etc., below will be the challenges and costs.

- Additional resources
  - Will need a SME in the network domain
  - Need a SME in a crypto domain
- Time
  - Development of these modules will take time, before we venture into the business logic (which is the core competency)
  - Validation towards functionality of these modules
  - Validation towards complying to the standards
  - Different network topologies to be setup for validation
- Risks
  - This is a new 1.0 source which needs field time (feedback) to stabilize over a period of time
  - Will be engaged with customers on these issues than the feedback on the actual business logic or workflows

Versus, if we choose cURL 3rd party open source, all of the above are a given.

cURL is developed and supported by open source,

- No need to hire SMEs
- No need to validate this library
- Focus on the business logic
- Feedback from customers will be more on the business logic

## **2.6 Phoenix- kill the current design and redesign**

As a successful product, customers might be happy with the current version\design which is stable and is good enough for the customers to resolve their needs. But, the product development cycle and team would know best when they think the time is right to redesign \ re-implement \ re-architect a legacy product which has been in the field.

### **2.6.1 Quality Aspect**

During the development and maintenance of a software/application, there are changes/fixes/improvements for the betterment of the product in the field. Most of these changes\fixes are due to one or more of the following:

- Feedback on the workflow
- Performance (in-the-field adoption)
- High Priority / Severity issues
- And so on

As part of customer feedback over a period of time, we also learn how customers intend to use certain features of the product. We also learn about the workflow of the entire solution stack. This is valuable feedback, which can be considered for our future implementations/design.

### 2.6.2 Example

In our experience, we have an existing product which has been successful in the field for more than a decade. Product had evolved over this period of time based on the feedback from the customers and also was enhanced with new features based on requests from multiple customers.

There was a need for the product to evolve further to serve customer needs with new infrastructure components and also resolve a list of design quality aspects. The re-design / re-architecture of the current product updated workflows which also considered the seamless transition between the legacy and next generation versions.

For example: we had a product/application which was mainly supporting Windows platform and was designed and developed over couple of decades ago with the technology which was prominent in that time period.

The product had served best in the field with great quality and with more net promoters. Having said that, when we envisioned the future possibilities of the product we realized that the current/legacy design/architecture/implementation is not going to serve or with stand another decade or more.

We realized that we had to re-design/re-architect the application to meet our vision and future possibilities. This was accomplished and the results were highly positive.

Along with the above mentioned architectural decisions, we also had few more considerations (like, choice of language, technology, etc.,) that added more value in terms of improved quality and led to a remarkable reduction in delivery time.

## 3 Conclusion

We had a unique opportunity to **re-invent** our product/solution which had been in the field for more than a decade. In this process, we considered the design patterns or architectural decisions which would influence and impact the quality of the product/solution/software. This paper presents a few of those concepts that we had considered and implemented in our project. Basing our redesign on all of the quality aspects discussed in this paper, we saw a **significant improvement** in the **quality** and the **velocity of the development**.

These architectural decisions helped us achieve our vision without impacting the quality and time aspects that are critical to meet customer requirements and to be in the market at the right time.

## References

### Web Sites:

IASA SPAIN – An association for all IT architects. [http://www.tutorialspoint.com/software\\_architecture\\_design/introduction.htm](http://www.tutorialspoint.com/software_architecture_design/introduction.htm) (accessed July 7, 2016).

Wikipedia – Architectural Patterns. [https://en.wikipedia.org/wiki/Architectural\\_pattern](https://en.wikipedia.org/wiki/Architectural_pattern) (accessed July 7, 2016).

Tutorials Points – Software Architecture and Design Tutorials. [http://www.tutorialspoint.com/software\\_architecture\\_design/introduction.htm](http://www.tutorialspoint.com/software_architecture_design/introduction.htm) (accessed July 7, 2016).

Anil Z Chakravarthy 2012. How Design Trade-offs Influence Software Testing. [http://www.uploads.pnsqc.org/2012/papers/t-79\\_Chakravarthy\\_paper.pdf](http://www.uploads.pnsqc.org/2012/papers/t-79_Chakravarthy_paper.pdf) (accessed July 7, 2016).



# Strategies for Minimizing Continuous Integration Response Time

Adrian Cook, Krishna Nattanmai, John Liao

*adrian.cook@intel.com, krishnakumar.nattanmai@intel.com, john.liao@intel.com*

## Abstract

Continuous integration (CI) is integral to the modern software development process. As with the majority of development teams today, our organization uses CI with the goal of improving development throughput and quality. In the past, our CI had a 90 minute turnaround time while running every test for each patch submitted to our project. The reality is that our long CI response time was creating a bottleneck in the development process rather than accelerating it. Ideally, CI builds and tests only the code modified by a patch, which theoretically allows us to get feedback within minutes. To move toward this ideal state, we have employed several strategies including: a redesign of our major software components, splitting our software repositories, and targeted testing. These strategies helped to reduce our CI response time by up to 75%. The reduced response time increased our development throughput by up to 33%. The same strategies can be employed by any software development project to improve its development throughput.

## Biography

*Adrian is a Software Engineer at Intel. He holds a B.S. in Computer Engineering and a B.S. in Electrical Engineering from the University of Florida. He has been at Intel for about 2 years and currently works on a software QA team within a validation engineering organization in Portland, OR.*

*Krishna is a Software Engineer at Intel. He holds a Master's degree in Computer Applications from Madurai Kamaraj University, India. He previously worked at IBM. He has been at Intel for 19 years and, for the past 2 years, has been working on a software QA team within a validation engineering organization in Portland, OR.*

*John is an Engineering Manager at Intel. He holds a B.S. in Electrical Engineering from the University of Arizona and an M.S. in Electrical Engineering from the University of Illinois at Urbana Champaign. He has been at Intel for 14 years and currently manages a software QA team within a validation engineering organization in Portland, OR.*

# 1 Introduction

Continuous Integration (CI) improves software quality by providing immediate feedback to developers as they make incremental changes to a project's code base. As a project grows, the source code expands, which, causes compilation time to increase. Additionally, more source code means more unit and integration tests, resulting in an increased testing time. Thus, as a project grows, the CI response time to a developer's change grows proportionally. Eventually teams reach a threshold where the CI response time becomes a detriment because the feedback loop is so long that it discourages developers from making frequent commits.

Our team manages two git repositories. One contains code for our custom compiler while the other contains user developed code intended to be built using the custom compiler. Both repositories use a combination of Google's Gerrit Code Review tool and JetBrains' Team City to create the CI workflow you see in Figure 1. In this workflow, a developer clones the repository and proceeds to create their incremental change on their local machine. When ready, the developer pushes the change to Gerrit making it available for code review, and also triggering our Continuous Integration Gate testing managed by Teamcity. This gate testing occurs pre-merge and ensures all changes that get merged to our code bases conform to our coding standards, build in all environments, and pass 100% of the unit and integration tests. If all CI servers available to Teamcity are busy, then the change is queued, and run as resources become available. If the patch fails the gate testing, a notice is sent to the developer so they can amend their change, and push a new patch to Gerrit. If the gate passes, the developer and code reviewers are notified the patch is ready for review. As in a typical code review process, the code reviewers will leave comments for the developer to fix, until they approve of the code, then the code is merged into the repo. Once merged, our CI regression testing is triggered. Our regression testing is a much deeper level of testing than what is run pre-merge and is where all testing that takes a substantial amount of time is done i.e. functional testing. If the commit fails regression testing, the developer is notified of the bug that is now in the code base, and expected to create a new commit to fix the issue. If regression testing passes, then this commit is ready to be deployed to customers.

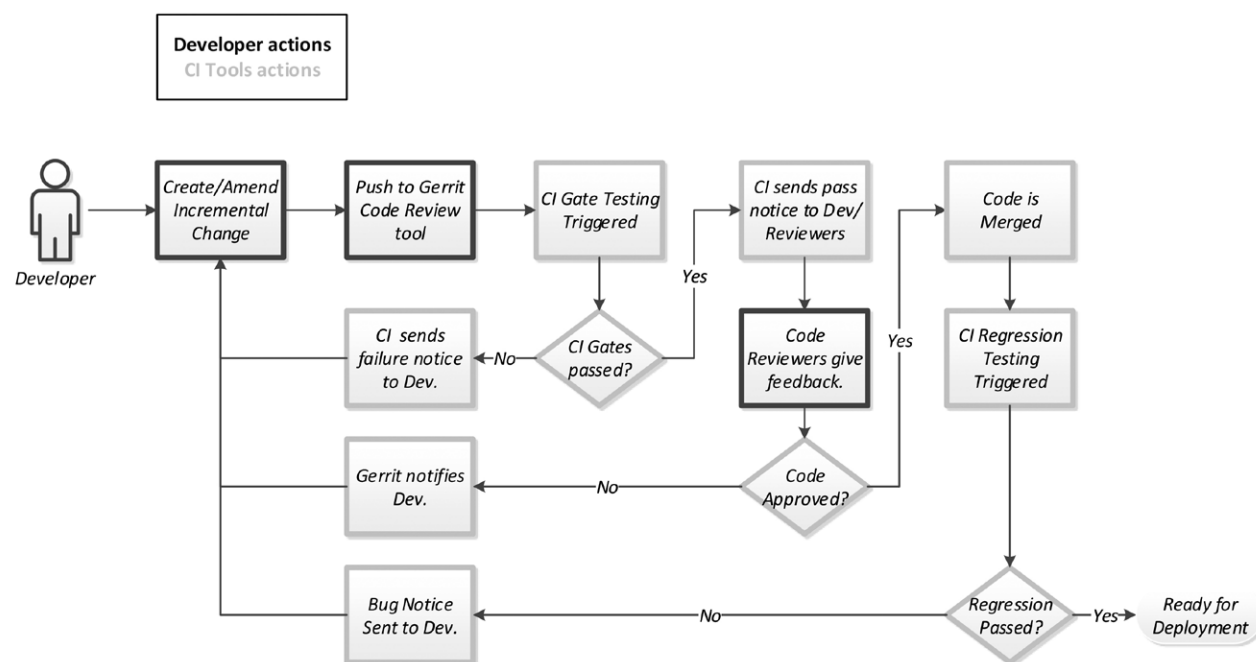


Figure 1: Our Environment's CI Workflow

As you can see in this process the Gate testing is run very frequently. It is run with every new commit pushed to Gerrit as well as with each amendment made to a commit prompted by either a gate test failure, or code reviewer's comments. For us, it is critical to minimize the CI response time of the gate testing so developers can quickly get their incremental changes merged. As our projects grew, we observed the response time of our gate testing growing proportionally, and consequently a drop off in development throughput.

This paper will detail strategies we applied to minimize CI response time in each of our repos. We will also provide a retrospective detailing the lessons we learned while applying these strategies, and a look at our plans for future improvements.

## **2 Compiler Repository**

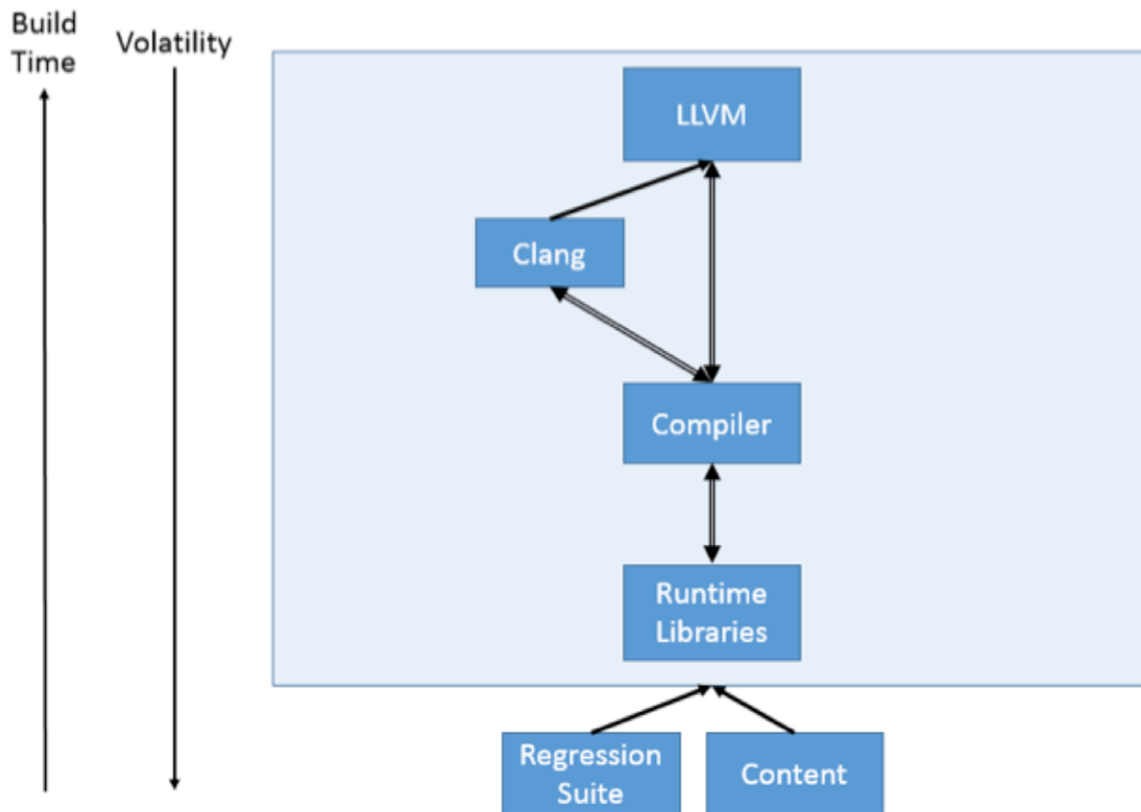
### **2.1 Background**

Our compiler repository contains a C++-based tool built around an open source compiler framework called **LLVM**. We began our tool by expanding upon the optimization infrastructure provided by LLVM, and during this initial phase a design decision was made to integrate LLVM with our repo. This integration allowed developers to easily add customizations to, build, test, and debug LLVM along with their own code. About one year into our project, we were maintaining an acceptable gate testing median response time of 45 minutes during which time we performed coding standards checks, compiled our tool, and ran ~1400 unit+integration tests on 3 different OS for each patch submitted to our repo. Over the course of the second year, added features caused the median response time to double to 90 minutes. At this point, development was being severely hampered by the slow CI response. Since our automation was already taking advantage of some typical strategies of using powerful build servers as well as parallel builds, we began to look at how we could apply some other strategies for minimizing CI response time.

### **2.2 Strategies Implemented**

#### **2.2.1 Staged Builds**

While the close coupling between our compiler and LLVM increased initial development speed, this led many developers to view LLVM as component of our repo as opposed to a third party set of libraries. In some cases, this perspective caused developers to create dependencies from LLVM into our code generating the dependency graph that you see in Figure 2.



**Figure 2:** *Project Dependency Tree*

The cyclical dependencies within our project meant that almost any change to our compiler tool required that you also rebuild most parts of LLVM, making incremental building very costly. As our project matured, we found fewer issues were being traced back to LLVM code and that most of our compiler developers actually needed little to no knowledge of LLVM internals. As can be seen in table 1, we were seeing that LLVM which made up 2,486,642 lines of our 6,118,464 line project saw an average of only 1.91 commits per month compared to the 77.64 commits going into the project as a whole.

**Table 1:** *LLVM vs. the Rest of the System*

	LLVM	Rest of Our Code	Together
<b>Approximate Size<sup>[1]</sup></b>	2,486,642	3,631,822	6,118,464
<b>Avg. # commits/month</b>	1.91	75.74	77.64
<b>CI Build time Windows</b>	5m:14s	3m:38s	10m:41s
<b>CI Build time Linux</b>	4m:44s	3m:0s	8m:40s
<b>Unit+Integration Test time</b>	5m:0s	6m:0s	10m:15s

As continuous integration response time increased to 90 minutes, we identified LLVM as the largest source of waste. LLVM was a large, static component of our system which took a long time to build and test during the gate, but doing this as part of every patch's testing provided very little value. The original reasons for including LLVM in our repo were gone since now most developers worked within our code base and very few developers needed an understanding of LLVM. In order to bring our continuous integration times down, we decided that LLVM needed to be decoupled from our compiler.

<sup>1</sup> Size was measured using the open source tool CLOC and gives a fair approximation of the lines of code (not including blanks, comments, etc.) in each project however is not an exact number.

The first step that we took was to remove the cyclical dependencies that had been created between our compiler and LLVM through the lifetime of our project. One of the main ways this was achieved was through dependency inversion<sup>1</sup>. Eliminating these dependencies decreased our incremental build time significantly, but the clean build time was still very high. In order to get improvement in the clean build time as well, we decided to split LLVM into a separate repo. This made sense since developers were no longer tracing through LLVM code, and it also would prevent any cyclical dependencies from being created again. Maintaining multiple repositories can often incur a large cost, but since our LLVM component was very stable this was not a large concern for us.

In order to split the repositories, we needed our compiler tool to compile with LLVM packaged as a set of libraries. However, we still wanted the compilation experience to be seamless for developers. To facilitate this, we setup automation for continuous deployment<sup>2</sup> of our custom branch of LLVM as well as a dependency manager. Since LLVM saw relatively few commits, each commit now triggers a release which then updates a configuration file used by the dependency manager so that our compiler points to the newly released version of LLVM. While the separation of repos caused developers modifying LLVM code to need to do some extra steps in order to commit changes, this strategy yielded a 66% decrease in compilation time for the common case.

## 2.2.2 Dead Code Removal

The staged builds yielded the most significant returns for us, but we still employed some other strategies in order to reduce our CI response time. The second strategy we employed was removing dead code. Identifying dead code is often very difficult. In order to find dead code, we gathered code coverage data for our tool by building our tool using coverage flags, running all user-generated content, then using `gcov` to generate a coverage report. We then brought the results to the development community and discussed each module showing 0% coverage. The question was whether these modules just hadn't been utilized by users yet or if they were actually dead code. Additionally, we undertook a large effort to remove code and libraries that were in the repo, but weren't being built. This was a much more manual effort and required a close analysis of our build system. Through these efforts we were able to remove approximately 2,000,000 lines of dead code from our repository. Although only a small portion of this code was actually being built as part of our CI, this removal still helped to improve our repository clone time, and decreased time for coding standard analysis scripts to run over our repo.

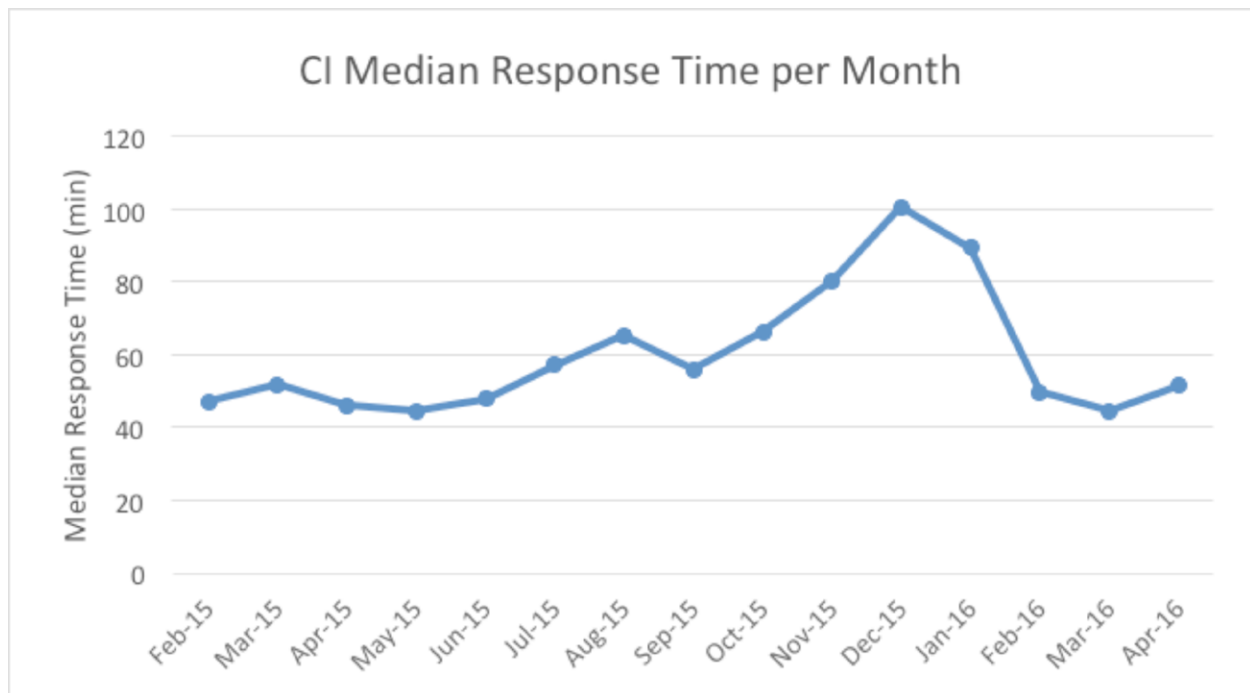
## 2.3 Results

As can be seen in Figure 3, there was a steady increase in the CI median response time throughout 2015 from around 45 minutes then culminating at 90 minutes. We implemented the staged builds and dead code removal in February-March 2016. Figure 3 shows that within this time, there was a dramatic impact to the CI median response time with a drop of about 50%.

---

<sup>1</sup> Dependency inversion is a form of decoupling of software modules by having both high and low level modules depend on abstractions.

<sup>2</sup> Continuous deployment is an approach where the process for deploying a software project is automated so that each successful integration that meets the release criteria is immediately deployed to users.



**Figure 3:** *CI Median Response Time per Month*

## 2.4 Retrospective

Ultimately, we learned while trying to optimize the CI response time for the compiler repository that the fundamental design assumptions you begin a project with can slowly shift over time. The needs of your developers and the demands of your customers are always in flux, so from time to time you need to reassess even your most basic assumptions. At the beginning of the project, it was thought every developer would be tightly coupled to LLVM, however at this point, very few developers have any LLVM expertise at all. This kind of evolution can be seen in almost any project, and while the shift may occur within the development community, the infrastructure can lag behind until it truly begins to hinder development. Large amounts of dead code or areas with very little activity can be a strong indication that a shift such as this has occurred. It implies the development/user community has moved away from a previous paradigm and has new requirements. It can also imply that a planned feature was never fully implemented or utilized by the user base, but the feature didn't get removed. While eliminating dead code doesn't always give large gains to the CI response, monitoring your repository for it can lead you to discover when tectonic shifts have occurred and through this when to reassess your design assumptions.

We also learned that though there may seem to be compelling reasons for tightly coupling your project with an external dependency, the cost generally outweighs the benefits in the long run. Many other companies have realized this as well and have made the shift to a microservices model. In a microservices model an application is built in a distributed fashion by integrating many small, modular services with the expectation that each service practices both CI and Continuous Deployment (CD). While our project is not fully utilizing a microservices model, and it may not be feasible to fully switch yours to that model either, we recommend at least using a dependency manager, and relying on your build system to tie your libraries to your dependency's libraries. This loose coupling provides a much more maintainable, extensible solution.

## 2.5 Future Improvements

When running a test suite, ideally, you should only run tests that target areas affected by your patch. If done correctly, targeted testing provides your change with the same confidence as running your entire test suite in a fraction of the time. In the compiler repository, our goal is to map the relationship between integration tests and the modules they cover in order to implement this kind of targeted testing.

## 3 User Repository

### 3.1 Background

The compiler our software development team creates is used by hundreds of engineers across the world to compile a set of common libraries and shared tests used to validate various hardware platforms. These libraries and tests are created by isolated validation teams all with a hardware engineering background. It is our team's responsibility to ensure tests developed by these teams can be shared among all our platform validation teams. Our continuous integration workflow is the main tool which ensures quality and reusability is maintained even under this open source like development model.

After the first year of our project, the user repo contained approximately 250 tests mostly used by a single platform, and took approximately 3 minutes to build. Over the course of the second year, we had an influx of adoption of our tool to the point where each checkin submitted to the user repo required a build of ~2400 tests taking approximately 30 minutes. We found that with this 30 minute CI response time, development throughput in the user repository began to be impeded. Each new platform adopted for the tool, multiplied the ways we needed each test to build to ensure reusability. Therefore, we realized the current gate testing would not scale for the further growth we were expecting in the near future.

### 3.2 Strategies Implemented

#### 3.2.1 Eliminate Build Inefficiencies

Due to the open source nature of our user repository, its build system was developed over the past couple years by many different people with varying levels of expertise in the overall system. Upon close inspection, we found some libraries were re-built for every test that included them instead of being built once and subsequently linked into. There were also some inefficient design choices made to make project-level development convenient, but made system level verification difficult. For instance, there was no single build command to make all projects. The lack of system oversight meant that no one was watching when checkins went in that drastically affected the overall build time. In order to resolve this, we dedicated an engineer to oversee the system and to coordinate the project teams. This system engineer cleaned up these inefficiencies by redesigning the build system to optimize for the CI build as well as reuse.

#### 3.2.2 Eliminate Code Duplication

The majority of our user base was hardware developers who were now being asked to create tests in an object-oriented environment. We found that many of these users lacked understanding of good software design principles especially at a system level. Some project teams were duplicating large amounts of code within their tests instead of creating libraries for their common validation flows. There was also duplication between projects as teams desiring reuse would copy desired code into their own area of the repository. We had believed code review would catch bugs like these, but evidently they were escaping. Upon finding this rampant duplication, we set out to 1) prevent further introduction of it into the system, and 2) eliminate the existing instances of it.

In order to prevent further introduction of it into the system, we created scripts to be run as part of our gate testing. The script drives the open source code duplication detection tool SIM. There was some tuning needed to find the correct thresholds, but SIM is now run as part of the gate testing and fails users who submit new files which duplicate too much code within existing source files. We define too much code duplication by setting the threshold to 100% token similarity where tokens are defined by SIM as the text of code blocks not including comments, identifiers, layout, etc. In this way, we can steer our users toward greater code reuse. Though new developers still fail this gate rather often, we have found that as they gain experience in our repo, their coding habits actually change and the process encourages better practice going forward. While SIM was helping us prevent more duplicated code, we also used it to track to the existing instances of duplicated code and eliminate them one by one. Through this method we were quickly able to cleanup redundant tests and libraries especially between projects.

### 3.2.3 Support for Targeted Gate Testing

Though the above fixes improved the build time marginally, we didn't get the exponential benefits needed to maintain our median response time while scaling the number of platforms the user content was intended to be reused for. We observed that in most incremental changes, a user only touches their own area of expertise. In many cases, they make changes which affect only a very small portion of the tests. In these cases, building all tests in the user repository for all platforms is a waste of time and resources.

We decided to add intelligence to the build system so that when building the user repository tests as part of our Gate Testing we could provide a list of affected resources/tests for a given incremental change. When running the gate testing, we are able to extract a change list from Git, and map resources/libraries/tests added, modified, or deleted to a list of affected resources/libraries/tests. The key to this mapping from the changed list to the affected list has been created by hand, and only achieves a file level granularity, but has worked to cover the majority of check-ins. From the affected list, the gate can then create recipes for building each member of the affected list. This essentially amounts to an incremental build, but can be done even when the build servers are forced to do a clean checkout of the source due to failures. Though this added a small extra step in the gate testing, once it is identified that the targeted testing is possible for the given changes, the build time is considerably reduced. In the best case, when it identified only one test needs to be built out of the ~2400, it reduces the build time overall by 75%.

At times the targeted gate testing is not possible. For instance, when an affected list grows beyond 100 tests for the given change set. At that point, the cost of discovery of affected resources/libraries/tests begins to outweigh the benefit of the targeted gate. In this case, the gate testing just builds all 100% of the tests. Figure 4 shows the effect that the targeted gate had on the number of tests. The targeted testing was introduced at around build 600, and thereafter we see more than 80% of the check-ins from the users are processed with targeted testing.

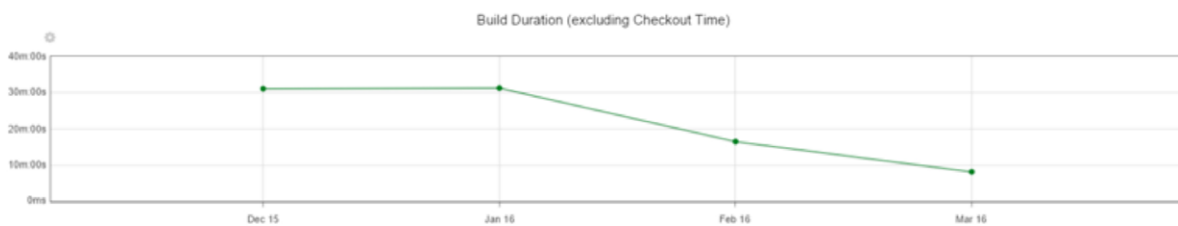


**Figure 4: Effect of Targeted Build on Test Count**

To protect the integrity of the system, regression testing has been added to the system which clean builds all the tests for each check-in instead of using the targeted build.

### 3.3 Results

As can be seen in Figure 5, this shows the average duration of the Gate Testing from December 2015 through March 2016. While the build system and automation improvements were gradual, we implemented the Targeted Gate testing in February-March 2016. It can be seen that within this time, there was a dramatic impact to the average build duration with a drop of about 75% which directly correlates to the drop in CI response time. The reduced response time also, increased our development throughput by 33% from 30 average incremental changes per day to 40. While changes per day is a limited measurement for development throughput because it doesn't take into account code quality, size of commits, code complexity, etc., it does serve as a good measure here to show the increase in the number of changes that our CI was able to respond to. After implementing these solutions, we found that our CI was no longer the bottleneck to the incremental changes, since we rarely saw Gate Testing queues building up, and developer feedback on the gate was generally positive. The challenge was now to maintain this level of response time as new features continue to be introduced.



**Figure 5:** Average Build Duration of User Repository Gate Testing per Month

### 3.4 Retrospective

We mistakenly believed that things like code duplications, or build system inefficiencies could be identified in code review. The problem with this is that in most cases humans are bad at noticing errors in the details. It's boring and generally our brain will simply fill in the context with what we expect to see. This along with the fact that our users were generally under tight project deadlines meant that even when an error like this was noticed in code review it was often felt to be too late to hold up the commit to change. You can imagine that this attitude quickly spirals out of control. We strongly recommend for formulaic checking tasks like code duplication checks a tool or script such as SIM is used. This is much more error proof, and can generally give the user feedback much more quickly and accurately than a code reviewer.

Seek the simple solutions first that give high return for low effort such as upgrading your servers, utilizing multithreading, and incremental building. After these are exhausted more creative, project specific solutions need to be found.

In software development, there is no nirvana, no attainable ideal state where improvement is no longer necessary. Within 3 months of our implementation of the above solutions, we have already caught our CI response time dramatically increasing due to a new feature's implementation and an escaped performance bug. New requirements and project growth, always introduce new bottlenecks to CI testing. Therefore, we are continuously monitoring for regressions as well as improvements to make.

### 3.5 Future Improvements

SIM does a very good job at stopping users from copying exact code within their tests and libraries, however something it can't do is detect redundant features. One major problem we are currently facing is deprecating redundant API's. We have had the process where when creating a new API, you must also support the existing one while tests and libraries are transitioned. This creates little disruption for the

user, however means that we often end up having multiple libraries implementing the same feature, which bloats the code base. Our goal is to systematically eliminate these redundant API's in much the same way we did with the duplicate code. While this creates some risk for users who are working against versioned API's, since all our users are internal to Intel, we are able to work directly with them and help them migrate to the latest version of our tool. Eliminating the redundant API's will once again help in reducing the CI response time in addition to reducing the overall infrastructure requirements.

## **4 Both Repositories**

### **4.1 Background**

One of the main reasons for our response time being so long is that often many developers will submit patches at the same time. This usually occurs at deadlines such as when we are approaching a release branch cutoff. This means that although a single patch submitted to our CI with all machines available could take 35 minutes to run, patches will be queued waiting for machine time in order to run. Our original CI implementation simply intercepted patches committed to our repository, and built/tested our tool with that patch in all necessary environments. We saw the opportunity to optimize our gate for three common development cases that were applicable to both repositories.

### **4.2 Strategies Implemented**

#### **4.2.1 Acknowledgement of a Commit**

We observed that a developer will often submit a patch to retrigger CI if he/she doesn't see a pass/fail response within an hour. When surveyed, many developers responded that they thought the original job may have been lost when, in reality, the job is simply waiting in the queue. This behavior often caused our CI gate testing queues to get longer, leading other developers to submit more patches with the problem quickly spiraling out of control. We implemented a simple solution to address this problem. As soon as a job is submitted, we send a response to the developer acknowledging the receipt of the commit and its place in the queue. This way the developer knows the job is in the queue and approximately how long he/she would need to wait for it to complete. For updates, the developer can always access our CI web interface to get an estimation from Team City for an up-to-date estimation of how much longer their job will be in the queue.

#### **4.2.2 Deletion of Duplicate Patches**

We noticed that developers would frequently push several patches in quick succession (within <5 minutes of each other). They would do this when they realized that they had made a mistake in the original patch, or wanted to adjust the wording of a comment, etc. For our CI this meant that there were now two or more almost identical patches running on our machines, some of which were actually obsolete already. We call patches like this redundant patches in our CI. In order to fix this redundant patch issue we wrote a script which runs before any other jobs are started on the build machines. This script checks to see if there is a previous version of the patch we are about to run already in the gate. If a redundant patch is found, then all jobs currently running that redundant patch are cancelled. This means that one developer submitting many patches would never clog the queue for other developers causing a long CI response time for everyone.

#### **4.2.3 Staged Gate Testing**

There were many cases where a patch would fail a simple job such as a coding standard check which takes under 1 minute to run, but the developer wasn't being informed of that failure until all jobs

completed. Our longest job could take up to 35 minutes to run! The initial rationale to only report failures once all jobs had finished was that developers wanted to see and fix all problems associated with a patch together. From a QA perspective however, we knew that the patch was bad within a minute, and then that bad patch would take up machine time while not necessarily providing any more information to the developer than we knew after a single minute. We wanted to shift toward the popular “Fail Fast” paradigm. The compromise we found with developers was to introduce stages so that certain failures get reported early, and cancel jobs in later stages that take more machine time. Our first stage of testing consists of all jobs that take less than 10 minutes, and the second stage contains the rest. This way a developer gets feedback within a short time, doesn’t waste machine time with a bad patch, and still gets information from multiple jobs.

## 4.3 Results

Since we cannot measure how many patches were avoided as a result of adding an acknowledgement of commit, we cannot accurately gauge its effect. However, we can measure the effect of deleting duplicate patches as well as the effects of staged testing. On average, 15% of the commits in the queue are deleted because they are older patches of the same commit. That means the average queue time is reduced by 15% as a result of this strategy. After implementing Staged testing, we observed approximately 19% of our commits failing the first stage of gate testing, so in those cases we are now saving a significant amount of machine time by skipping our second stage of testing.

Figure 6 shows the effects of the both the user repository specific strategies and the strategies detailed in this section on the average time each checkin spent in the queue. You can see that in December 2015 our average queue time was at around 17 minutes, however as these changes were made it was reduced to only 3 minutes.



**Figure 6:** Average Time Each Checkin Spent in Queue per Month in the User Repository

## 4.4 Retrospective

Improving the gate testing time of a single patch in an ideal scenario is great, but optimizing your CI workflow for multiple patches provides larger benefits when your resources are limited. In general, these strategies probably took us the least amount of time to implement, but provided some of our largest benefits. These strategies were mostly developed from developer feedback to our QA team. For instance, our commit acknowledgement strategy was prompted by developer feedback in scrum meetings that they felt their incremental changes were being dropped by the automation system frequently. In reality, their commits were often just queued behind many other developer’s commits. While waiting through long queue times developers would often realize a mistake in their code as well and want to fix it. This prompted many to ask us how to remove their old patch from the queue. This was something that was done manually at first, but when we realized many developers were running into this, we automated it. Some of these strategies may not be generally applicable, but we would highly recommend soliciting feedback from your development community on their interactions with your CI system. We survey developers at the end of each sprint/quarter, and listen to their feedback on how their interactions with the CI system are going.

## 4.5 Future Improvements

One consistent piece of feedback we have been getting from developers is that they want to be able to run all of our coding standard checks within their local repository before even pushing to their incremental change for gate testing. They are frustrated when they need to push an additional patch for a formatting change, comment fix, etc. The staged testing was good in that it gave them much faster feedback, but we would like to break down the stage testing even further to the point where our fastest checks can be run locally on a developer's machine as either pre-commit or pre-push hooks.

## 5 Conclusion

While at the birth of a project, CI often gives a near instantaneous response, as the project grows, CI response time grows proportionally. Initially, the CI response time can often be kept under control through horizontal scaling, such as upgrading your build servers and parallelizing builds, and through incremental builds. However, once these simple measures have already been implemented, it can become necessary to pursue more creative strategies to prevent CI response time growth. It takes a more in-depth understanding of the project, but many effective strategies can be found for reducing CI response time while maintaining confidence in the quality of a patch. In our process of optimizing our CI response time, we learned several key lessons: question our project's most basic design assumptions, direct our optimizations by listening to developer feedback, and finally that continuous improvement must be part of continuous integration.

## References

1. Shamash, A. 2013. "Evolution from Quality Assurance to Test Engineering." Google Test Automation Conference 2013, <https://developers.google.com/google-test-automation-conference/2013/presentations> (accessed March 24, 2015).
2. Duvall, P. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007.
3. Paliotta, J. 2015. "How to Develop High Quality Software." Proceedings of the Pacific Northwest Software Quality Conference 2015, <http://www.pnsqc.org/how-to-develop-high-quality-software/> (accessed July 18, 2016).
4. "Gerrit Code Review - A Quick Introduction." <https://review.openstack.org/Documentation/intro-quick.html> (accessed August 2, 2016).
5. "TeamCity." <https://www.jetbrains.com/teamcity/> (accessed August 2, 2016).
6. "The LLVM Compiler Infrastructure." <http://llvm.org/> (accessed August 2, 2016).
7. "gcov—a Test Coverage Program." <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov> (accessed August 2, 2016).
8. Grune, D. "The software and text similarity tester SIM." [http://dickgrune.com/Programs/similarity\\_tester/](http://dickgrune.com/Programs/similarity_tester/) (accessed August 2, 2016).

### (Footnotes)

- <sup>1</sup> Size was measured using the open source tool CLOC and gives a fair approximation of the lines of code (not including blanks, comments, etc.) in each project however is not an exact number.



# Ensuring Quality with a Quality Team of One

Kim Janik

kim.janik@Intel.com

## Abstract

You are brought in at the start of a project as the quality lead, tasked with bringing up the quality team, standards and processes for the project. The development team is already fully staffed and churning out code like crazy. Surprise! The quality team is...you. How do you proceed when you are the only quality engineer on the team? How can you meet management expectations of a quality product? Learn five methods for improving quality on your team even if you are the only dedicated quality resource for your project.

## Biography

*Kim is a Sr. SDET at Intel Security, advocating quality, process improvement and teamwork. She received her M.S. degree in computer engineering from Portland State University. Kim credits her success to her passion for learning new things as well as her diverse background. Her varied roles have included ScrumMaster, Quality Champion, Validation Engineer, Software Engineer, Hardware Design Engineer, Computer Science Instructor and Zombie Apocalypse Survival Instructor.*

*Copyright Kim Janik 2016*

# 1 Introduction

You promised yourself you would never again be the only quality engineer on the team. You asked the right questions during the job interview. You got to hear from the team members how happy they were working there. The hiring manager guaranteed you could hire a rock star validation team to meet all of the project goals. Little did you know that the budget would be cut between the time you were hired and the time you started. Does this sound familiar?

Unfortunately, this scenario isn't uncommon. Talking with engineers, they will grudgingly agree it has happened to them before. So why do we sweep it under the rug instead of addressing it? Has it become so common of a problem that we ignore it? Why are there not dozens of papers on the topic? Having a one-person quality team happens to the best of us. It is important to note that this is not the ideal situation. This paper is in no way advocating anyone should actually run a team with a single quality member as that would be taking the "small and scrappy" concept too far. In the long run, it is more beneficial to try to change the underlying mindset that often leads to this situation instead.

However, it is important to acknowledge that this problem does actually happen. As the newly hired quality lead, this now becomes the reality you need to face. In the worst cases, upper management still has the same expectations except you have less resources than you originally anticipated. Your first deliverable may be to provide a schedule of when quality milestones will be achieved. So, rather than give up, how can you still provide the best quality possible giving the lack of expected resources?

Before deciding on a course of action, you need to do some pre-work. Early in your arrival to the project, take the time to do a good, honest assessment of the current quality situation. Sometimes, there genuinely are no processes, procedures or tools in place. Take careful note of the people identified as part of your project. Next, list where you want to be in the future. This may or may not align with management's requirements so note any discrepancies. Next, organize your list from least to most important. Then, put estimates on each line item regarding how long it will take to implement with the current team (i.e. you). Be brutally honest. You are not doing yourself any favors by underestimating your time. At the same time, don't pad the work, either. If there are tools, licenses or other hurdles in the way, note those.

The rest of the paper presumes you have already evaluated the present quality situation and are familiar with the work being requested of you. This paper also expects that you are using Scrum or similar Agile methodologies. If that is not true in your specific case then adapt the suggestions to your situation where possible.

## 2 Ask for More

Sometimes, the first steps are the most obvious. When you don't have enough people to do the work you've been given, ask for more people. This is a tough scenario as it's likely that something has changed in the organization since you've joined. Perhaps the budget got downsized. Perhaps the expected team members left. Perhaps other extended job offers fell through. But, it doesn't matter if the original goal was to have a team of one hundred engineers working under you. Currently, you have one. It will never hurt to ask the question:

*"Can more people be hired?"*

If the answer is "no" because there's a hiring freeze, find out if there is an end date for when it will be lifted. If you are told "no" then do not hesitate to ask when you should follow up on the topic again. Also, consider asking if it's easier to hire in a contractor or consultant on a temporary basis to help jumpstart the work. While you may not want to give the impression of being a nuisance when you're brand new, you also don't want to leave anything on the table, either.

In many cases, it helps your case to state facts. The easiest way to do this would be to point to an existing document as a reference. You can tailor this to your particular situation, but an example might be:

*"I was reading an article on improving quality that suggests the ideal developer to tester ratio is 1:1. It seems we are behind the industry curve. Is this something we can address?"*

In the best case, this will be sufficient to point out the imbalance. However, engineers as a whole are data-driven and more persuasion might be required. You can start by evaluating your team's efficiency. If you already have historical data then use it as a basis for your calculations. If not, use the most recent information you do have.

For example, let's suppose you have a fairly new team where the definition of done (DOD) was readily adopted and is being consistently followed. Each sprint is two weeks. On your team, the average developer has a velocity ( $V_{dev}$ ) of two small user stories per sprint. Given that you have dedicated half of your time to test infrastructure development, you have a velocity ( $V_{qa}$ ) of four small user stories per sprint. Start by working out deficit on a per sprint basis ( $D_{sprint}$ ) based on the number of developers ( $N_{dev}$ ) and testers ( $N_{qa}$ ) you currently have:

$$(V_{dev} * N_{dev}) - (V_{qa} * N_{qa}) = D_{sprint}$$

Independent of the resulting numbers, this will give you a rough idea of the work facing you. You can extrapolate this out to an entire quarter ( $D_{quarter}$ ) or even the next year ( $D_{year}$ ).

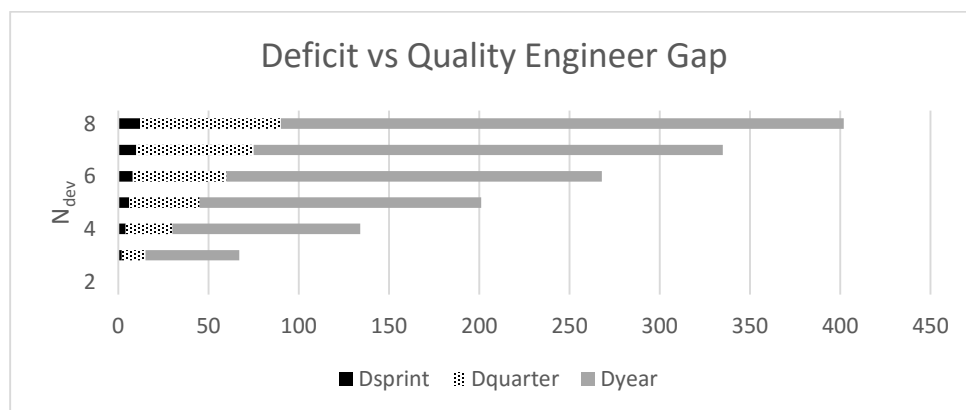
$$D_{sprint} * 6.5 = D_{quarter}$$

$$D_{quarter} * 4 = D_{year}$$

The following table gives you an idea of the impact as the number of developers increases.

$N_{dev}$	$N_{qa}$	$V_{dev}$	$V_{qa}$	$D_{sprint}$	$D_{quarter}$	$D_{year}$
2	1	2	4	0	0	0
5	1	10	4	6	39	156
8	1	16	4	12	78	312

You may find it useful to graph this data.



This is a painful trend to see, however it's a good example of how you can use real data to make your case. You can explain how many quality engineers you still need, given the team's current state. Is this the only equation you can use? Of course not! Only you understand what data is most likely to sway your company. Don't hesitate to bring in 'soft' figures as well.

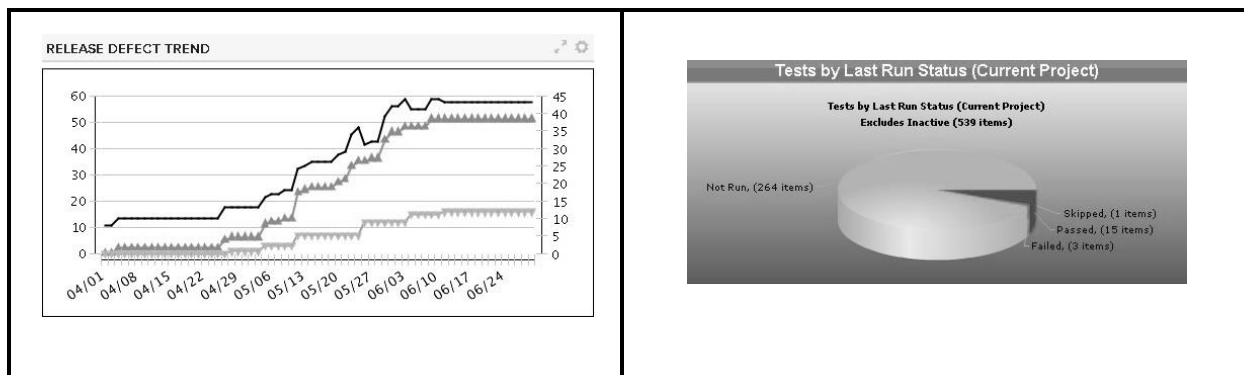
### 3 Expose the Deficit

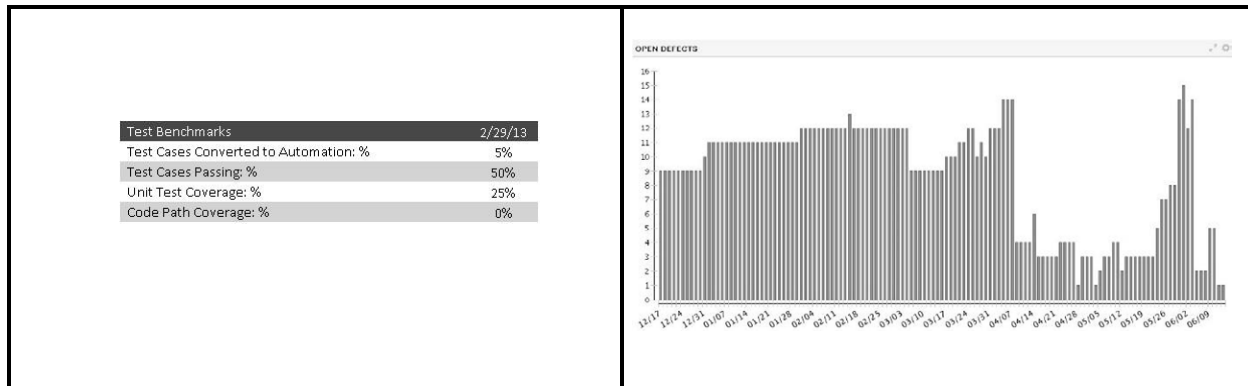
At the heart of this section is one important word: communication. You are in a difficult position. A good team will understand this and appreciate knowing what you are accomplishing for them. It is also possible that your team will need to be educated on the current situation. No matter what, you want to ensure open communication with your team, management and stakeholders. Your goal is to make as many people as possible aware of the work you are engaged in as well as what you are trying to accomplish going forward.

Similar to what was discussed in the last section, there is another obvious step to take when you don't have enough people to do the work you've been given. Reduce the amount of work. In most situations you can negotiate the amount of work that needs to be accomplished. Perhaps that complicated regression suite can come off the table early on. Maybe the formal UI testing can be de-prioritized until resources are available to complete it. At the end of these negotiations, you are likely still going to have an overwhelming amount of work to accomplish "sometime".

You're probably doing this already, but start publicly documenting the amount of work to be done. This is useful for two reasons. First, it can be very informative in showing why some work is not being completed. Second, it puts the focus on everything you've accomplished for your team. You want full transparency in the state of quality for your product. Not only is this a great work habit but it's also laying an information foundation that you can build on when you get to your other options in the sections below.

Quality reports can be an effective tool for communication. First, select a cadence. Emailing once a week, bi-weekly or monthly are all solid choices. Whatever you choose, be consistent in your mailings. Next, select your indicators. Only you can determine what is most critical to your team's needs. Some common indicators can be open bugs, test cases written, tool completion countdown, percentage of test cases automated. A few examples are show below.





It is important to note that this is not supposed to be a chance to complain or be negative. This is about facts. Yes, point out the deficit. But, don't forget to include the positive trends, too! Include your manager on these. Consider including your second-level manager as well.

If your team is employing Agile methodologies, you may be able to leverage them to help bring attention to quality. This can provide a baseline for user stories going forward, but it has an additional benefit. If the quality metrics you've laid out in the DOD are not met, your product owner will be faced with a decision – ignore the quality metrics to finish the work or delay the new feature to ensure quality. If the team doesn't already have a DOD or exit agreements, consider volunteering to help put those into place. You can include line items that will help the team align with the quality goals.

Last, there is one more skill to master in your communications. Learn to love “No.” In your calculations from the above section remember you included time for test development. That is critical. To make progress, you need to keep developing better processes and procedures. You will hear, “can't you just skip this one week?” You might get a request to reinstate previously removed quality items. Learn and use the important one-word phrase. As these type of instances come up, simply say, “No” and point them back to the data if necessary. This does NOT imply rigidity; instead, it means that each future conversation becomes a trade-off to ensure your team is able to meet the commitments they agreed to.

## 4 Enlist Existing Help

By now, you might be feeling pretty discouraged. Your assignment has gotten off to a rough start. The amount of work hasn't really gone down. Additionally, you've established that reinforcements aren't coming. However, have you considered the possibility that perhaps they're already here?

First, this may be an uphill battle if you're starting fresh at a new company. Nevertheless, it's still possible. Talk to everyone. Have coffee with them and learn what they do. Learn what they enjoy. If you already know people in your company, ask for networking suggestions. You will find that there are people that may be able to help you.

Consider the following scenarios:

*Peyton has just been hired as the ScrumMaster for three teams, including yours. After evaluating user story throughput, he comes to you to brainstorm possible improvements. However, in the course of these discussions, he offers to roll user acceptance tests into the definition of done, a feature which you had previously determined was below your cut line.*

*Emanuela has worked in finance for over 10 years. She loves her work but has reached a career plateau. She hears about a position overseeing the creation of a beneficial new financial tool that her team will be working on. The only qualification she's missing is PHP programming. Through mutual coworkers, she hears about the quality reporting webpage you want to implement. She agrees to work overtime for the next two months to write the webpage for you so she can acquire the new skillset.*

*Over lunch Joon shares her excitement about finishing her new build tool that she has developed for her team. Unfortunately, since she's ahead of schedule, she laments that the team won't have time for a first run for another two weeks. Since build tools are not yet a reality on your team, you offer your product as a beta test environment. Joon happily agrees.*

*Stefan is a seasoned product owner newly assigned to your product. His last product suffered from numerous sustaining defects originating from customer calls so he's concerned with early testing efforts. After discussing various options, he agrees to hold all new features for two Sprints in order to give you time to implement the unit test framework.*

*Lindsay is an intern in a neighboring group. Their project is ahead of schedule and Lindsay is looking for a short project to fill the remainder of the internship before going back to school. With their manager's permission, Lindsay agrees to spend 30% of their time working on your project.*

If you weren't looking for these opportunities, you would have missed some of your best free resources. Some companies actually have programs in place to help employees find each other. These systems are generally a win-win scenario as they get more skills, exposure or beta testers and you get a new, albeit temporary, resource. This is also where including the second-level manager on quality issues can come back to help you. They might have someone interested in the work you're doing, leading to another possible source of help.

## 5 Pick the Low Hanging Fruit

Right about now, you might be questioning if the point of this paper isn't to just wiggle out of doing your job. No; that couldn't be farther from the truth. At our core, we are quality professionals. This section is all about how to make a meaningful impression in a tough situation. And, it is possible.

By now, you should have a good understanding for the quality requirements as well as the other team factors. Now, evaluate your list. Look for opportunities. We've already identified DOD as one potential prospect. These are going to be the quality goals that have the highest priority and yet can be accomplished in the shortest time. You may hear these referred to as the "low-hanging fruit".

Take the specific example of defects. It is a common problem for projects to have absolutely no defect process. However, the defect process as a whole is full of quick-hitting possibilities. You could get the team started on an existing defect tracking tool. You can write up a quick-reference defect process cheatsheet. You can provide defect training including what fields are important for resolution. You can start sending out defect reports. You can start a bug scrub meeting.

In addition to the defect process suggestions above, the following table will give you a few ideas for some additional low-hanging fruit. Again, the focus is on where you can make a biggest impact in a minimal amount of time. These specific examples were selected because they could be implemented inside of a single week.

Process	Description	Why
Test Plan Overview	A presentation that outlines the quality goals	Different from a formal test plan, the idea is to level-set quality goals with the team.
Team Webpage	A centralized web-based location that the team can add documents	This is a quick and effective way to share information with the broader team.
Version Control Tools	Tracks the changes to the source code during development	Manage multiple developers and allow source code recovery.
Coding Standards	Set of guidelines to standardize the source code's programming style	It's often easy to get team buy-in. Additionally, it can increase readability and reduce defects.

Code reviews	Peers review each other's code before it is checked in to find mistakes early	Mistakes can be caught earlier in development. Additionally, enables software architecture cross-training.
--------------	---	--

Another technique you can try is to bound quality tasks. This may work well in cases where your quality vision does not directly correlate with management's requests. Reach an agreement about how long should be spent on a given task. Make as much progress towards this goal as possible in the allotted time. Then stop. Review it with management. If they want more than what was completed, propose a trade-off with other planned improvements.

Don't forget that quality processes have a lot in common across projects. You do not have to re-invent the wheel. In fact, you can often quickly establish some standardized tools and processes by adopting them, in whole or in part, from other teams. For example, if your team does webpage development for the company's widgets, can you talk with the gizmos web development team and establish a standard look and feel? The more creative you can get in your collaborations, the more opportunities you will start to see.

## 6 Convert Others

When you've tried all the normal methods for getting your quality improved it's time to fight (a little) dirty. In the previous sections, we talked about finding the willing converts. This section is about securing the unwilling ~~victims~~ volunteers. This will go a lot smoother if you have complete management buy-in on what you're about to do. If needed, you can re-visit the equations you developed above to help with justification. If you feel it would help, extrapolate the number of unproductive hours. Having developers working on QA might be a more palatable choice over developers doing busy work. Even if you don't have complete manager buy-in, a few of these proposed avenues will work even with a skeptical manager.

As a first step, consider converting your own developers. They are already part of your team and know the software well. This is going to be hard for a wide variety of reasons, but it's worth it because they have the programming skills you need. First, some engineers do not enjoy aspects of quality engineering. If you ask a developer to write a process document, they may consider it akin to chewing their own leg off. That brings us to another hurdle you'll face: the mindset. In some companies, there is the idea that quality engineers are somehow less skilled than developers. On one occasion, a seasoned quality engineer was asked by a developer, "You can program? Why aren't you a developer then?"

Start using the tools at your disposal. Education is an important aspect of your job. To address the mindset gap, consider listing out the technical skills needed to be a developer vs a quality engineer. In most cases, the underlying skillset is essentially the same. Scrum is an excellent example of this. In a true scrum team, you have only the product owner, the ScrumMaster and the team. There is no line between developers and testers. If your team has a strong Agile inclination, consider proposing that the team move to a 'true scrum' model and start erasing the line between the two roles. Offer to provide training to ease the transition.

At the end of the day, however, you may not be successful in fully converting your developers into quality assets. You can still get them to help in other ways, however. Find work where the developers have something to gain for themselves. Unit testing is a great example. They may not have come up with the idea but may take it over once suggested. It will identify early defects and the developers can manage it themselves, which may be more appealing than having the quality team own it. Last, have the developers complete tests on each other's work. If the quality team is unable to test in a timely manner, consider suggesting a developer.

*"I won't have time to look at that user story until tomorrow. However, Reilley is really familiar with that interface. Have you considered having Reilley take a look at it?"*

This has the added benefit of reducing friction in a group. Instead of being viewed as a blocker, you're offering them a solution to their current problem.

In your quest to convert the developers, don't forget about your other team members. ScrumMasters may be more sympathetic about achieving quality on a new product. Additionally, your goals may align with some of theirs. A technical writer with a few hours free may be able to help with reviewing early process documents. Even your manager isn't off limits here, especially if they were previously in a role with skillsets that align with your team's quality needs. It is an opportunity for them to keep their skills fresh and stay in closer contact with the team.

## 7 Conclusion

All of the techniques in this paper have been used in an industry setting over the last ten years. Communication and transparency will serve you well in any situation. Some of the ideas presented work better in particular environments. The manager's personality will influence your choices. Coworkers' temperaments come into play in many cases. You will need to be creative and flexible to reach your goals.

At the end of the day, you have a rough job ahead of you. Yet they hired you for a reason. You are in the unique position of solely owning the quality of the product. You can make a big difference even with your limited resources. Do not be afraid of being the first person to ask the question. Do not be afraid to hear 'no' repeatedly. And do not be afraid of making a few waves if it will result in a quality product.

## References

Koopman, Phil. 2015. "Tester to Developer Ratio Should be 1:1 For A Typical Embedded Project", Better Embedded System SW Blog, entry posted February 2, <http://betterembsw.blogspot.com/2015/02/tester-to-developer-ratio-should-be-11.html> (accessed July 6, 2016)

McConnell, Steve. 2007-2008. "Software Development's Low Hanging Fruit", <http://www.construx.com/LHF/> (accessed July 6, 2016)

Vroomans, Remko. 2014. "User Acceptance Test in Scrum", Agile/Scrum blog, entry posted July 28, <http://www.scrum.nl/prowareness/website/scrumblog.nsf/dx/user-acceptance-test-in-scrum> (accessed July 6, 2016)

Whittaker, James. 2011. "The 10 Minute Test Plan", Google Testing Blog, entry posted September 1, <http://googletesting.blogspot.com/2011/09/10-minute-test-plan.html> (accessed July 6, 2016)

# Transforming Organizations to Achieve TMMi Certification

Suresh Chandra Bose, Ganesh Bose

[SureshChandra.GaneshBose@cognizant.com](mailto:SureshChandra.GaneshBose@cognizant.com)

## Abstract

As the saying goes “Rome wasn’t built in a day”, it takes a significant amount of effort for organizations to improve their testing practices to achieve various maturity levels. It requires rigor to follow these practices consistently across all programs. And even though testing consumes a significant part of the overall project cost, less focus is paid to testing.

To bridge the gap for organizations to become TMMi certified at particular levels, Suresh Chandra Bose, an **Accredited TMMi** Lead Assessor, provides insight into how to implement the specific and generic practices required. This paper will explain: *the initial assessment of current testing practices and processes against the desired TMMi maturity level, build a recommendations roadmap, point out the organizational change management required to address gaps between the current and target end state, all while taking into consideration tactical and strategic opportunities to achieve TMMi and process improvement journey.*

In conclusion, the results of the proposed solution roadmap show how an organization which was once ad hoc - can transform into a managed, defined, measured, and optimized one. This improves the effectiveness of testing through benchmarking of its test practices, while achieving the only recognized test maturity model certification available in the industry.

## Biography

**Suresh Chandra Bose, Ganesh Bose** is a Manager - Consulting at Cognizant Business Consulting practice. Suresh is an accredited Lead Assessor from TMMi Foundation and has been in the IT Industry for more than 17 years with vast consulting experience in various industries. He has executed strategic initiatives for many Fortune 100 companies in the areas of PMO, PPM, Process Consulting, Program Management, TMMi Assessment/Implementation, Organization Strategy, Test Consulting and CIO/ Governance Dashboard/Metrics across the globe.

Suresh holds 17 International certifications in IT and speaks at numerous international conferences, such as the American Software Testing Qualifications Board (ASTQB) and the Pacific Northwest Software Quality Conference (PNSQC). He is also on the selection and review committee for one of the leading testing conferences.

Copyright **Suresh Chandra Bose, Ganesh Bose**

# 1 Introduction to TMMi

- TMMi is “Test Maturity Model integration”
- Developed by the TMMi Foundation
- Contains guidelines and framework for test process improvement
- Has five maturity levels for process evaluation in systems and software engineering
- Addresses cornerstones of structured testing: lifecycle, techniques, infrastructure and organization
- Provides a detailed model for test process improvement
- Positioned as being complementary to CMMi
- Performs assessment considering the level at which an organization wants to be appraised
- By far the most popular model in the industry for benchmarking testing practices

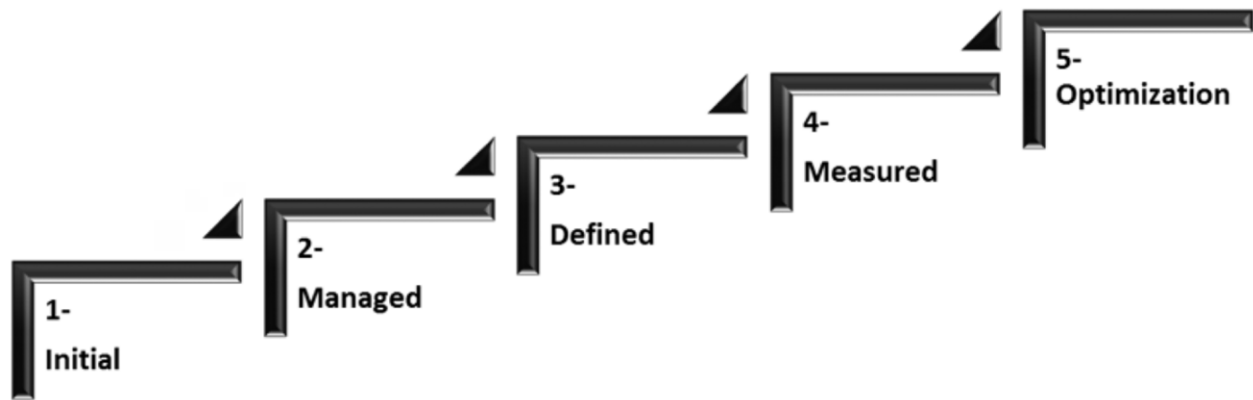


Figure 1: TMMi Maturity Levels

## 2 Need for TMMi Assessment

The reasons why organizations can benefit from a TMMi assessment are many and can be broadly grouped as follows:

- Integration of industry best practices with the testing process to achieve business objectives
- As IT is getting more complex, there is a need to show value to our customers
- Evaluating the current testing process against Industry standards to enhance credibility in market place (benchmarking)
- Standardizing and optimizing the testing process along with higher degree of process compliance

- Assess efficiency of support functions like Environment, Test Data, Build Management, Non-Functional Testing, etc. ,for testing activities
- Effective monitoring and control mechanism to mitigate schedule and cost overruns and quality issues

### 3 Comparison - CMMi and TMMi

Though TMMi is positioned as being complementary to CMMi and follows the same structure outlined by CMMi, there are some significant differences between the two. Figure 2 shows the key differences between both the process improvement frameworks.”

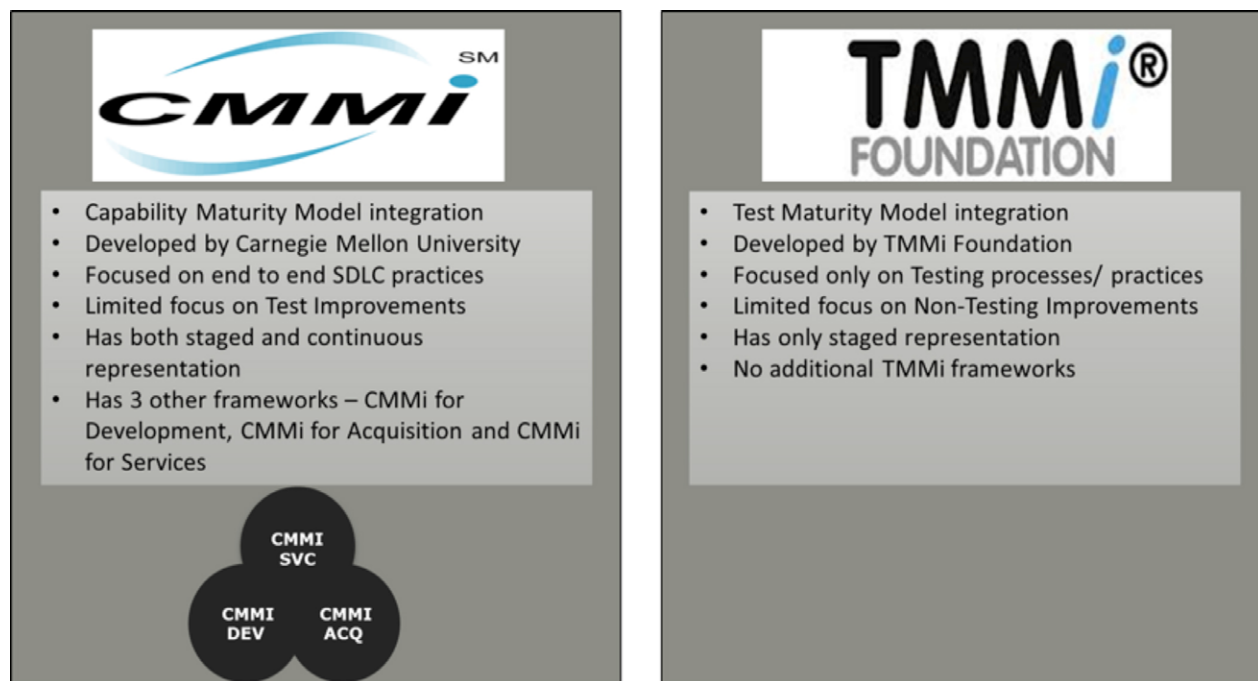


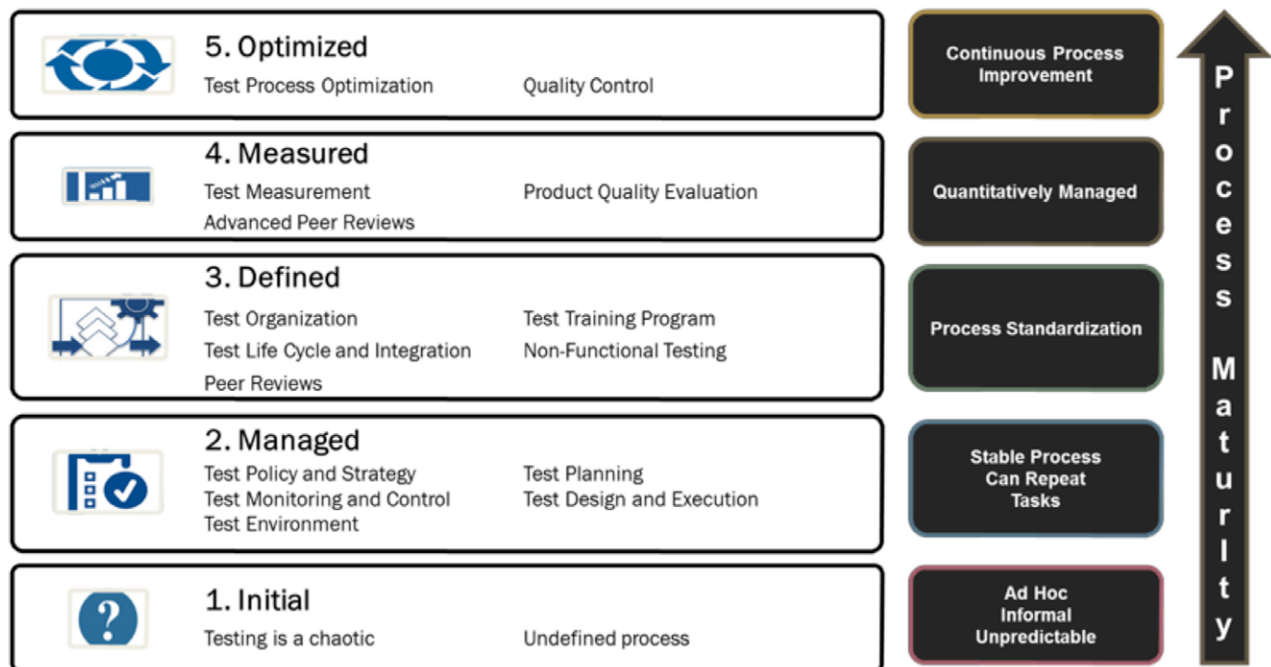
Figure 2: TMMi vs CMMi

### 4 Improvement Roadmap for Implementation

There are five levels of maturity in TMMi: starting from a level of being ad hoc and unmanaged, to improving in maturity to one which is managed, then defined, measured, and finally optimized. The process areas for each TMMi maturity level are shown below in figure 3.

- Level 1 - Initial: There are no defined process areas to be considered level 1. This means any organization regardless of whether they have any testing process can be considered at level 1. The actual maturity rating starts from level 2 upwards.
- Level 2 - Managed: This level has 5 process areas starting from Test policy and Strategy. The process here is considered stable and can repeat the tasks at the project level.
- Level 3 - Defined: Level 3 is standardization of processes at the organization level and also has 5 process areas.

- Level 4 – Measured: This level is quantitatively managed with focus on measurement, product quality and advanced peer reviews.
- Level 5 – Optimized: The organization is capable of continually improving its processes based on a quantitative understanding of statistically controlled processes. The testing techniques are optimized with continuous focus on fine tuning and process improvement.



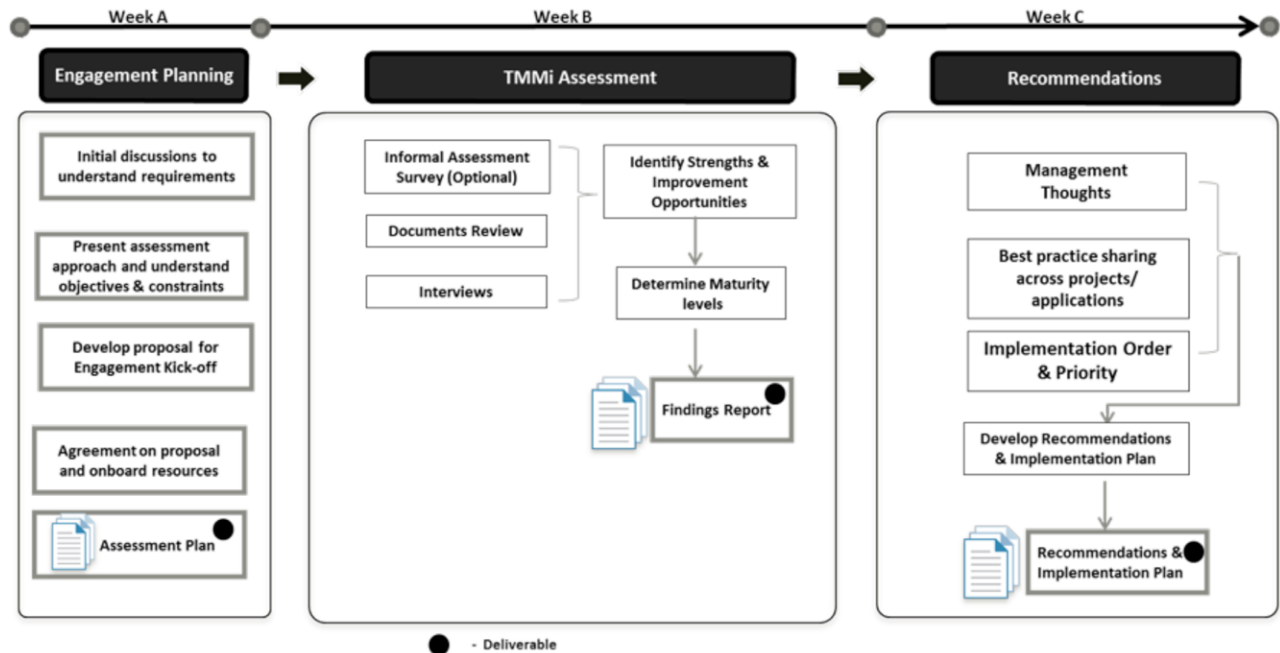
**Figure 3: TMMi Process areas**

## 5 Assessment Approach

Assessment is the first step in benchmarking the progress in test process improvement with three main phases as shown in figure 4.

As part of the planning phase, we (refer my team as “we” since it may not be done by a single person) understand the scope of the assessment with focus on the business units, geographical locations, sample projects representing the organization getting assessed covering all lifecycle methodologies and project size.

Once the interviews are scheduled, we talk to the various stakeholders to understand the current landscape of the organization. We also perform documentary evidence reviews. The interviews are to ensure adequate coverage of the generic and specific goals and practices across various process areas of TMMi. Timelines change based on the maturity assessment level chosen by the organization to be assessed.



**Figure 4: Assessment Phases**

Based on the outcome of the findings report, recommendations are developed for every gap identified and they are prioritized based on the inputs from the organization. A detailed Implementation roadmap is finally developed with timelines for all the prioritized recommendations.

Suresh Chandra Bose the only TMMi Lead Assessor from North America, using similar approach successfully transformed a technology giant to become the first organization in the United States to be formally certified at TMMi Level 3.

<http://en.community.dell.com/dell-blogs/dell4enterprise/b/dell4enterprise/archive/2016/04/26/dell-enterprise-validation-first-in-north-america-to-achieve-tmimi-certification>

## 5.1 TMMi Assessment Maturity Rating Methodology

The rating process strictly adheres to what is laid out in TMMi Assessment Method Application Requirements (TAMAR) which defines the requirements considered essential to Assessment methods intended for use with the TMMi framework.

There are 16 process areas, 77 goals and 345 practices including the specific and generic practices from level 2 to level 5. The interviews and document evidence are rated for all the specific and generic practices which are rolled up to the goals, then to process area, and finally to the maturity level. See Figure 5 for an overview of the TMMi Rating Mechanism.

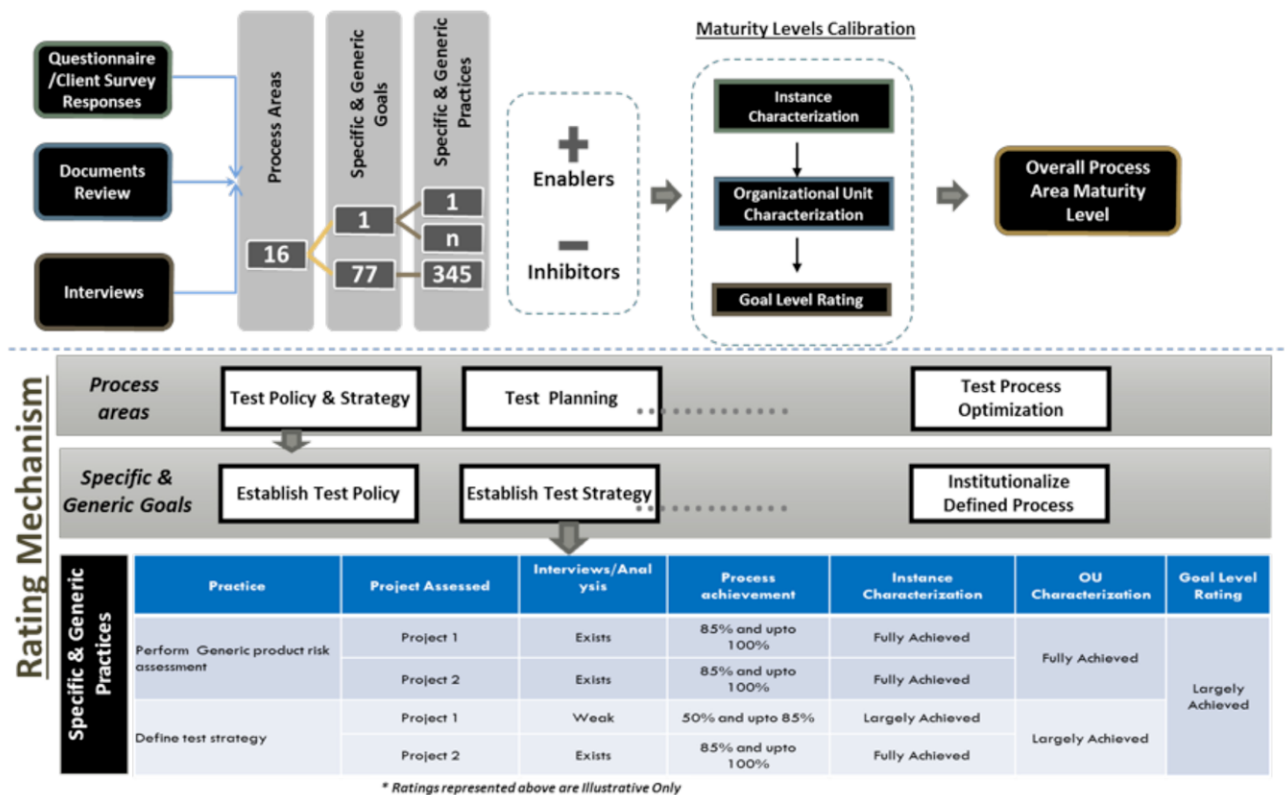


Figure 5: TMMi Rating Mechanism

## 5.2 TMMi Maturity Level Calibration Guidelines

Process Areas are designated the appropriate maturity levels based on the following guidelines

- **Fully Achieved -**
  - Convincing evidence of process compliance
  - Systematic and widespread implementation of process
  - No obvious weakness in distribution, application and results of this process exists
  - Process achievement is between 85% and up to 100%
- **Largely Achieved –**
  - Significant evidence of process compliance
  - Minor weakness in distribution, application and results of this process exists
- **Partially Achieved –**
  - Some evidence of process found
  - Process exhibits significant weaknesses, is incomplete, not widespread, or inconsistent in application or results.
  - Process achievement is between 15% and up to 50%

- **Not Achieved** -
  - Little or no evidence of process
  - Process achievement is between 0% and up to 15%
- **Not Rated** - Any supporting goal that cannot be rated based on the current phase of the project must be “Not Rated”
- **Not Applicable** - The process area is considered not to be in the scope of the assessment or applicable to the organizational unit by the Lead Assessor

### 5.3 Example

Let's take the process area “Test Policy and Strategy” from level 2 as an example to clearly explain how the sub-practice contributes to the roll up of rating to the process area level. Sub-practice is an informative model component that provides guidance for interpreting and implementing a specific or generic practice. See Figure 6 which shows the complete flow.

Specific goals (SG) are a required model component that describes the unique characteristics that must be present to satisfy the process area. The 3 goals within the process area “Test Policy and Strategy” are

- SG 1 Establish a Test Policy
- SG 2 Establish a Test Strategy
- SG 3 Establish Test Performance Indicators

Specific practice (SP) is an expected model component that is considered important in achieving the associated specific goal. The specific practices describe the activities expected to result in achievement of the specific goals of a process area.

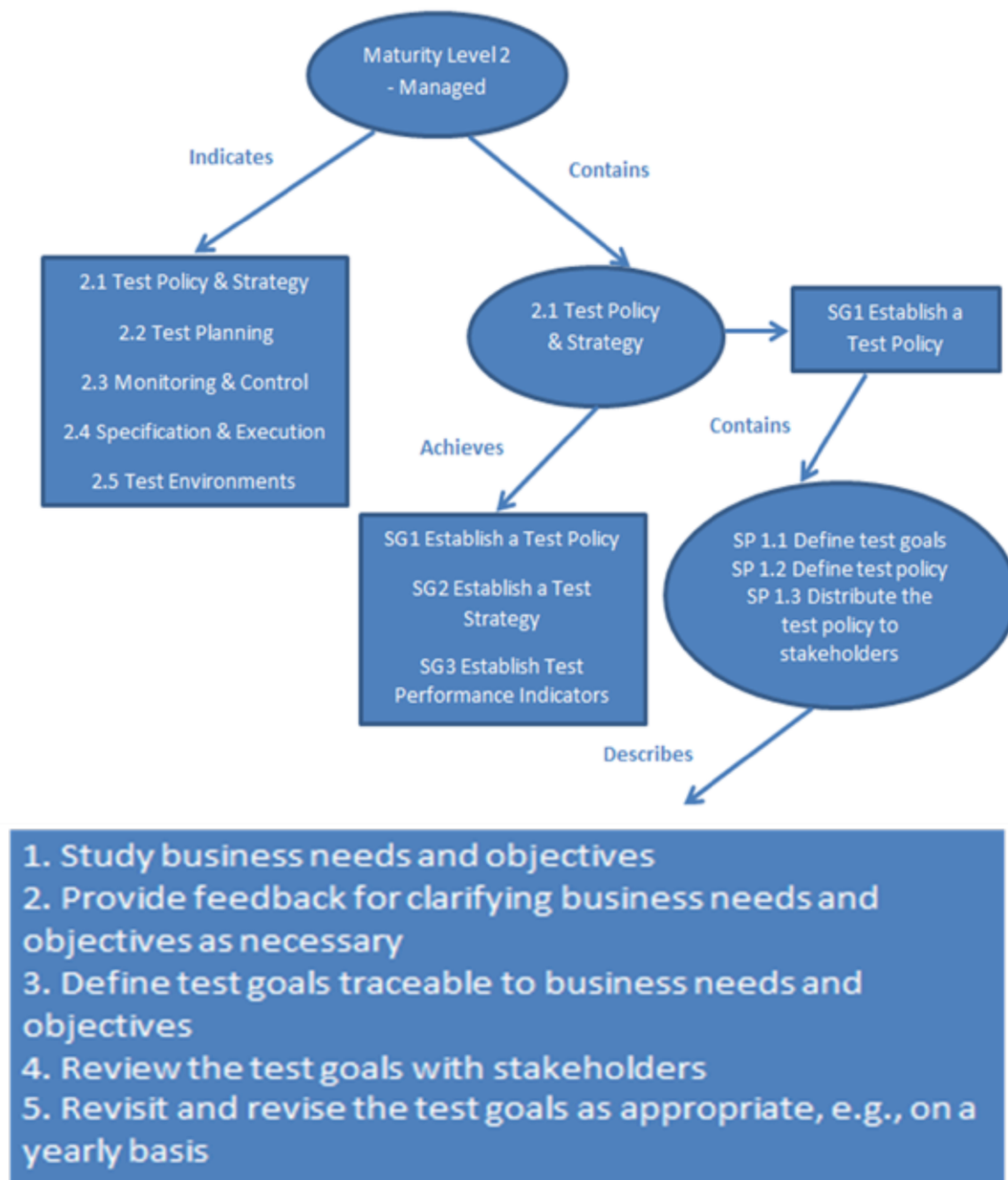
The SG 1 which is “Establish a Test Policy” contains the following 3 specific practices

- SP 1.1 Define test goals
- SP 1.2 Define test policy
- SP 1.3 Distribute the test policy to stakeholders

The SP 1.1 which is “Define test goals” contains the following sub-practices:

- Study business needs and objectives
- Provide feedback for clarifying business needs and objectives as necessary
- Define test goals traceable to business needs and objectives
- Review the test goals with stakeholders
- Revisit and revise the test goals as appropriate, e.g., on a yearly basis

Once all the above sub-practices are satisfied, the SP 1.1 will be fully implemented and likewise once SP 1.1 to SP 1.3 is satisfied, SG 1 will be fully implemented. Then the roll-up happens to the process area “Test Policy and Strategy” when all the SG 1 to 3 are evidenced.



**Figure 6: Example**

## 6 Benefits and Projected Savings

The improvements indicated below in Figure 7 are determined based on a number of factors from some sample projects and also could vary for different organizations.

TMMi completely focuses on addressing Testing practices and enables not only enhancement in quality through excellence in Testing process once these recommendations are implemented, but most importantly provides tangible business benefits to the organization given the fact testing phase of project accounts for nearly 40 to 50 percent of project efforts and related cost.



	Level	1. Initial	2. Managed	3. Defined	4. Measured	5. Optimized
Savings & ROI	Total Return on Investment %	0.0%	26.1%	55.7%	61.8%	64.9%
	ROI over previous Maturity Level	0.0%	26.1%	40.1%	13.8%	8.2%

**Figure 7: TMMi Benefits**

## 7 Summary

There has been a lot of traction in the recent years with so many organizations attempting to achieve TMMi maturity at various levels. Organizations want to decrease the leakage of defects to production, reduce the testing rework, provide quality outcomes and reduce cost of testing by having standardized TMMi practices, thus improving overall customer satisfaction.

This paper explains how organization that starts out as ad hoc can identify their weaknesses and remediates the gaps to transform to higher maturity levels using the only recognized Test maturity model available in the industry and benchmark their test practices.

## 11 References

Suresh Chandra Bose, Ganesh Bose. "How to Take Organizations to Higher Testing Maturity" ASTQB Conference, 2015

<https://www.astqb.org/certified-tester-resources/astqb-software-testing-conference/conference-agenda/how-to-take-organizations-to-higher-testing-maturity/>

TMMi Foundation Reference Model

<http://www.tmmi.org/pdf/TMMi.Framework.pdf>

"Dell Enterprise Validation First in North America to achieve TMMi Certification"

<http://en.community.dell.com/dell-blogs/dell4enterprise/b/dell4enterprise/archive/2016/04/26/dell-enterprise-validation-first-in-north-america-to-achieve-tmmi-certification>

# Measurement System Analysis with Attribute Data

Nagesh NM, Praveen S

Siemens Technology and Services Pvt Ltd., CT DD DS AA DF-PD  
[nm.nagesh@siemens.com](mailto:nm.nagesh@siemens.com), [praveen.shivaswamy@siemens.com](mailto:praveen.shivaswamy@siemens.com)

## Abstract

Every project captures defects which are classified based on multiple attributes like Severity, Defect Type, Injection & Detection Phase, and are used for Root Cause Analysis (RCA), based on which action items are derived to improve defect occurrence. However, we do not explore to see if different developers (who are also peer reviewers during reviews) who do this classification have same understanding of attributes. If the understanding is different among developers, the classified data used for RCA may provide a different picture & actions planned will not lead to improvement in defects thereby making it an ineffective RCA.

**Measurement System Analysis (MSA)** is scientific and objective method of analyzing the validity of a measurement system for use by quantifying its accuracy, precision, and stability. **Gage R&R**, an MSA methodology widely used for attribute data was piloted in a project with the following sample:

- 14 defects across Code Review, IT & ST which were classified by Subject Matter Experts
- 6 developers with varying experiences across the project selected to classify these defects in 2 trials
- The data was analyzed using Minitab tool in following steps:
  - o Attribute Agreement Analysis
  - o Comparing Appraisers against themselves
  - o Comparing Appraisers against a known Standard
  - o Comparing between Appraisers
  - o Agreement between all Appraisers together and the Standard

Gage R&R result needs be drawn based on overall Kappa values from the above 5 analysis steps. The result from sample study was that we need to improve the ability of developers to make better decision in categorizing the defects by providing training & bringing in more clarity on attributes.

Attribute Gage R&R was successfully implemented for the first time in CT DD DS AA DF-PD. The study helped us to device trainings specific to individuals & also teams. We now intend to spread this best practice across projects within & outside DF-PD.

## Biography

**Nagesh NM:** Lead Engineer - Quality, CT DD DS AA DF-PD. Having 14 years of experience in Quality & with Siemens for 11 years. Bachelor of Engineering & Post Graduate Diploma in Operations Management. Interested in Quality Assurance, SPC, Lean, Six Sigma, Minitab.

**Praveen S:** Lead Engineer – Quality, CT DD DS AA DF-PD. Having 12 years of experience in Quality & with Siemens for 6 years. Bachelor of Engineering. Interested in Quality Assurance, SPC, Lean, Six Sigma, Minitab.

# 1 Introduction

Every project uses a defect tracking tool and eventually data gets accumulated over the period of time. These defect data are further classified based on multiple attributes like Severity, Defect Type, Defect Injection phase & Defect Detection Phase as per organization requirements or QMS. This data is then used for conducting Root Cause Analysis (RCA) based on which action items are taken to improve the defect occurrence. However, we do not explore to see if different developers who do this classification have the same understanding of the attributes. If the understanding is different across developers, the classified data which is used for RCA may provide a different picture & the actions planned will not lead to the improvement in defects thereby making it an ineffective RCA. Here is one such a method that reveals lots of insights on the data that will be of good use to the Project Manager.

**Measurement System Analysis (MSA)** is scientific and objective method of analyzing the validity of a measurement system. Measurement System Analysis is often a “**project within a project**”. It is a “tool” which quantifies:

- Equipment Variation
- Appraiser (Developer/Tester) Variation
- The Total Variation of a Measurement System

Measurement error is unavoidable. There will always be some measurement variation that is due to the measurement system itself. Main sources of variation are due to: Materials, Methods, Machines, People, Environment & Measures.

## 2 Objectives of MSA

- Quantify the amount of variation
  - How BIG is the measurement error?
  - How much uncertainty should be attached to a measurement?
- Separate the contributions of variability
  - What are the sources of measurement error?
- Determine criteria to Accept/Reject a Measurement System
  - Is the measurement system “capable” for this study?
  - Are the measurements being made with measurement units which are small enough to properly reflect the variation present?
  - Can we detect process improvement if and when it happens?
  - Is the measurement system stable over time?
- Allow for the comparison of measurements systems
- Determine actions to improve the measurement system

## 3 Tools of MSA

Common tools and techniques of MSA include: Calibration studies, Fixed effect ANOVA, Components of variance, Attribute Gage R&R, ANOVA Gage R&R, Destructive testing analysis and others. The tool

selected is usually determined by characteristics of the measurement system itself. The methodology that is widely used for attribute data is Gage Repeatability and Reproducibility (Gage R&R).

### 3.1 Gage Repeatability and Reproducibility

Gage Repeatability and Reproducibility (Gage R&R) is measured as below:

$$\sigma^2_{\text{total}} = \sigma^2_{\text{product / process}} + \sigma^2_{\text{repeatability}} + \sigma^2_{\text{reproducibility}}$$

where  $\sigma^2_{\text{total}}$  = Total Measurement Variation

R&R stands for Repeatability and Reproducibility. Repeatability is the ability of a given appraiser (appraiser is a developer or tester who provides the data for the attributes selected during the study) to assign the same value or attribute multiple times under the same conditions. Reproducibility is the ability of multiple appraisers to agree among themselves for a given set of circumstances.

Typical Gage R&R analysis requires response variable to be Quantitative (Continuous). However, appraisers must frequently use subjective assessments that are Qualitative (Attribute). Most problematic measurement system issues come from measuring attribute data (data that can be classified and counted) in terms that rely on human judgment such as good/bad, pass/fail, etc. This is because it is very difficult for all testers to apply the same operational definition of what is “good” and what is “bad.” Hence, it is important to quantify how well such measurement systems are working.

Attribute Gage R&R reveals two important findings – percentage of repeatability and percentage of reproducibility. Ideally, both percentages should be 100 percent, but generally, the rule of thumb is anything above 90% is quite adequate.

Since measurements are used to guide decisions, it follows logically that the more error there is in the measurements, the more error there will be in the decisions based on those measurements. The purpose of Measurement System Analysis is to qualify a measurement system for use by quantifying its accuracy, precision, and stability.

## 4 Implementation of Attribute Gage R&R for Defect Databases

An Attribute Gage R&R is designed to simultaneously evaluate the impact of Repeatability and Reproducibility on accuracy. It allows the analyst to examine the responses from multiple reviewers as they look at several scenarios multiple times. It produces statistics that evaluate the ability of the appraisers to agree with themselves (repeatability), with each other (reproducibility), and with a known master or correct value (overall accuracy) for each characteristic – over and over again.

A defect database that tracks errors in processes (or even products) – can provide powerful information. This is one of the best examples on how Gage R&R can be done for attribute data & it can be quite helpful to the Project Managers (PM) & Project Quality Managers (PQM) in scoping and prioritizing potential improvement opportunities.

For this, first, the analyst (PQM/PM) should firmly establish that there is, in fact, attribute data. It is reasonable to assume that defects are classified into a category – a decision that characterizes the defect with attributes. A defect can have multiple attributes like Severity, Defect Type, Defect Injection phase & Defect Detection Phase as per organization requirement or QMS. Either a defect is properly assigned a category or it is not. Likewise, the correct slippage location is either assigned to the defect or it is not. These are “Yes” or “No” and “Correct assignment” or “Incorrect assignment” answers.

There are 2 types of analysis for Attribute data:

### **Fleiss Kappa:**

A statistical measure for assessing the reliability of agreement between a fixed number of raters when assigning categorical ratings to a number of items or classifying items. The measure calculates the degree of agreement in classification over that which would be expected by chance.

This method is used only when the data is nominal. Examples of nominal data are defect types & languages.

Kappa values range from -1 to +1. The higher the value of kappa, the stronger the agreement.

When:

- Kappa = 1, perfect agreement exists
- Kappa = 0, agreement is the same as would be expected by chance
- Kappa < 0, agreement is weaker than expected by chance; this rarely occurs

Usually a kappa value of at least 0.70 is required, but kappa values close to 0.90 are preferred.

### **Kendall's co-efficient of Concordance:**

Kendall's coefficient of concordance indicates the degree of association of ordinal assessments made by multiple appraisers when assessing the same samples.

This method is used when the data is ordinal. Examples of ordinal data are: severity levels, survey choices.

Kendall's coefficient values can range from 0 to 1. The higher the value of Kendall's, the stronger the association.

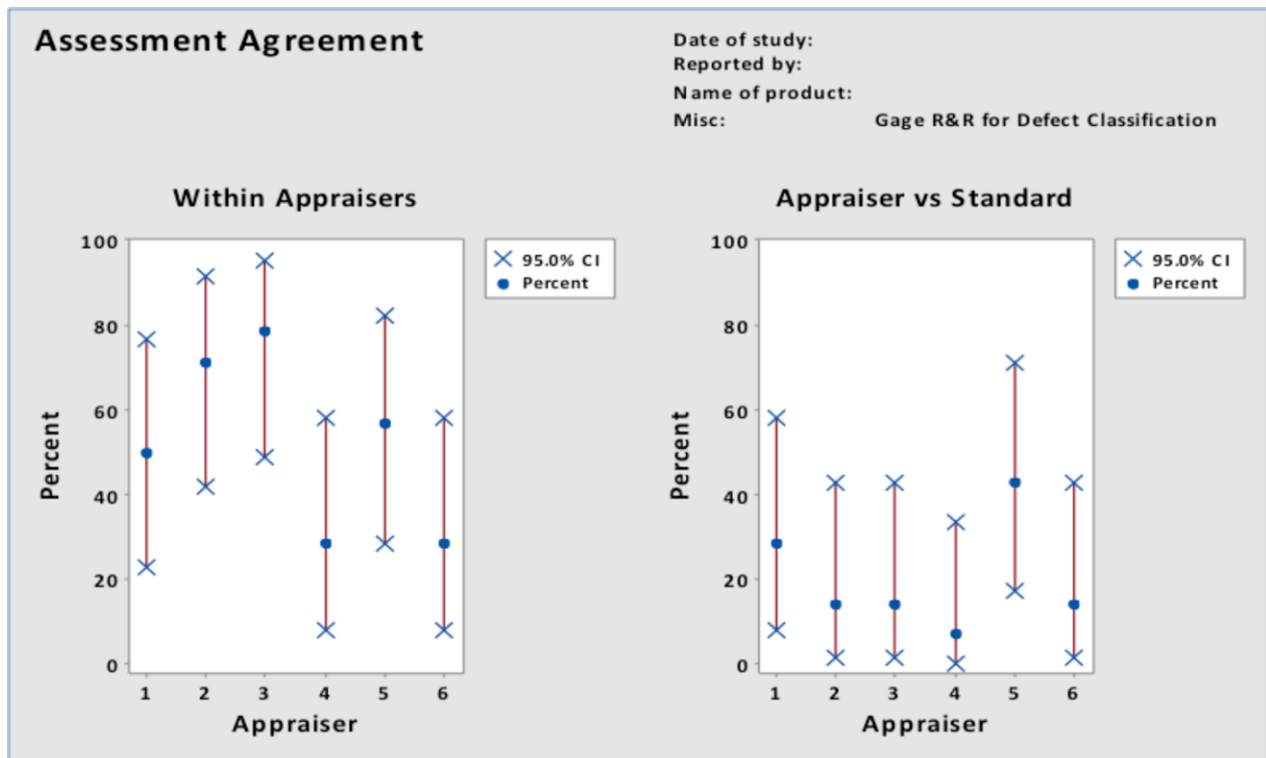
Usually Kendall's coefficients of 0.9 or higher are considered very good. A high or significant Kendall's coefficient means that the appraisers are applying essentially the same standard when assessing the samples.

Minitab has a module called Attribute Agreement Analysis which provides both the reports based on the data. Below is an example of a report with Fleiss Kappa analysis.

## 4.1 Assessment Agreement (Attribute Agreement Analysis)

This provides a chart as shown below to assess the consistency and correctness of subjective ratings:

- Within appraisers, when each appraiser rates each item more than once
- Between appraisers
- Against the standard, when you have a known rating for each part



### Interpretation from the Graph:

- The graph on the left side (within Appraisers) of the above figure shows how much an appraiser agrees with their own earlier decisions across successive trials
- This graph indicates that we may have a training issue with all the appraisers regarding their understanding of the criteria for the decision
- The graph on the right side (Appraiser vs Standard) indicates a percentage of agreement compared with the standard
- This graph indicates that none of the appraisers agree 100% with the standard & appear to be somewhat confused about the standard

## 4.2 Within Appraisers (Comparing Appraisers against themselves)

This provides a statistical summary assessment report within appraisers. If each appraiser provides two or more attribute values for the same defect, you can assess the consistency of each appraiser's ratings across the trials.

# Within Appraisers

Assessment Agreement				
Appraiser	#Inspected	#Matched	Percent	95% CI
1	14	7	50.00	(23.04, 76.96)
2	14	10	71.43	(41.90, 91.61)
3	14	11	78.57	(49.20, 95.34)
4	14	4	28.57	(8.39, 58.10)
5	14	8	57.14	(28.86, 82.34)
6	14	4	28.57	(8.39, 58.10)

# Matched: Appraiser agrees with him/herself across trials.

## Sample Data

None of the appraisers have agreed 100% between their two trials with themselves. Only appraisers #2 & #3 have agreed to >70% whereas #4 & #6 have the least agreement of around 28%.

## Fleiss' Kappa Statistics

Appraiser	Response	Kappa	SE Kappa	Z	P(vs > 0)
1	Access and Handling	0.26957	0.267261	1.00862	0.1566
	Algorithm	-0.07692	0.267261	-0.28782	0.6133
	Assignment Error	-0.03704	0.267261	-0.13858	0.5551
	Coding Standard	0.42857	0.267261	1.60357	0.0544
	Interface	0.62667	0.267261	2.34477	0.0095
	Performance Errors	0.57576	0.267261	2.15429	0.0156
	Variable Declaration	0.41667	0.267261	1.55902	0.0595
	Overall	0.39130	0.121317	3.22547	0.0006
2	Access and Handling	*	*	*	*
	Algorithm	0.71429	0.267261	2.67261	0.0038
	Assignment Error	*	*	*	*
	Coding Standard	0.42857	0.267261	1.60357	0.0544
	Interface	0.62667	0.267261	2.34477	0.0095
	Performance Errors	1.00000	0.267261	3.74166	0.0001
	Variable Declaration	-0.07692	0.267261	-0.28782	0.6133
	Overall	0.57088	0.160910	3.54783	0.0002

None of the overall Kappa values are less than zero which indicates that the responses are less likely the result of random chance.

None of the appraisers have agreed 100% between their two trials with themselves. Only appraisers #2 & #3 have agreed to >70% whereas #4 & #6 have the least agreement of around 28%.

None of the overall Kappa values are less than zero which indicates that the responses are less likely the result of random chance.

### Interpretation:

- None of the appraisers have agreed 100% between their two trials with themselves. However, only appraisers #2 & #3 have agreed to >70% whereas #4 & #6 have the least agreement of around 28%
- None of the Overall Kappa values are less than zero which indicates that the responses are less likely the result of random chance
- $P < 0.05$  for all except appraisers #4 & #6
- Algorithm Errors, Access & Handling Errors & Assignment Errors are having negative kappa values which indicate that appraisers have some confusions in these classifications
- **Overall Kappa < 0.7 across appraisers indicates training need**

### 4.3 Each Appraiser vs Standard (Comparing Appraisers against a known Standard)

This provides a statistical summary assessment report across trials with the known standard against attribute value to assess the correctness of each appraiser's ratings.

## Each Appraiser vs Standard

Assessment Agreement				
Appraiser	#Inspected	#Matched	Percent	95% CI
1	14	4	28.57	(8.39, 58.10)
2	14	2	14.29	(1.78, 42.81)
3	14	2	14.29	(1.78, 42.81)
4	14	1	7.14	(0.18, 33.87)
5	14	6	42.86	(17.66, 71.14)
6	14	2	14.29	(1.78, 42.81)

# Matched: Appraiser's assessment across trials agrees with the standard.

### Fleiss' Kappa Statistics

Appraiser	Response	Kappa	SE Kappa	Z	P(vs > 0)
1	Access and Handling	0.01576	0.188982	0.08338	0.4668
	Algorithm	-0.12179	0.188982	-0.64448	0.7404
	Assignment Error	0.27487	0.188982	1.45448	0.0729
	Coding Standard	0.42262	0.188982	2.23629	0.0127
	Interface	0.52167	0.188982	2.76040	0.0029
	Performance Errors	0.75652	0.188982	4.00314	0.0000
	Variable Declaration	-0.16667	0.188982	-0.88192	0.8111
	Overall	0.27749	0.080764	3.43576	0.0003
2	Access and Handling	-0.07692	0.188982	-0.40704	0.6580
	Algorithm	-0.14722	0.188982	-0.77903	0.7820
	Assignment Error	-0.07692	0.188982	-0.40704	0.6580
	Coding Standard	0.13095	0.188982	0.69293	0.2442
	Interface	0.81333	0.188982	4.30376	0.0000
	Performance Errors	-0.12000	0.188982	-0.63498	0.7373
	Variable Declaration	-0.12179	0.188982	-0.64448	0.7404
	Overall	0.06578	0.086810	0.75778	0.2243

### Sample Data

None of the appraisers have agreed 100% with the standard.

None of the Kappa values are less than zero which indicates that the responses are less likely the result of random chance as it gets closer to 1 & the answer changes depending on category.

#### Interpretation:

- None of the appraisers have agreed 100% with the known standard
- None of the Kappa values are less than zero which indicates that the responses are less likely the result of random chance as it gets closer to 1 & the answer changes depending on category
- $P < 0.05$  for all except appraisers #2 & #3
- Algorithm Errors, Access & Handling Errors, Assignment Errors, Variable Declaration & Performance Errors are having negative kappa values which indicate that appraisers have some confusions in these classifications wrt the Standard definitions
- **Overall Kappa < 0.7 across appraisers indicates training need**

## 4.4 Between Appraisers (Comparing between Appraisers)

This provides a statistical summary assessment report which tells about the appraiser's responses agreement with each other. Notable point is that, between-appraisers statistics do not compare the appraiser's ratings to the standard. Although the appraiser's ratings may be consistent, they are not necessarily correct.

# Between Appraisers

Assessment Agreement

#Inspected	#Matched	Percent	95% CI
14	0	0.00	(0.00, 19.26)

# Matched: All appraisers' assessments agree with each other.

Fleiss' Kappa Statistics

Response	Kappa	SE Kappa	Z	P (vs > 0)
Access and Handling	0.058531	0.0328976	1.7792	0.0376
Algorithm	0.128977	0.0328976	3.9206	0.0000
Assignment Error	-0.024390	0.0328976	-0.7414	0.7708
Coding Standard	0.375247	0.0328976	11.4065	0.0000
Interface	0.558788	0.0328976	16.9857	0.0000
Performance Errors	0.153808	0.0328976	4.6754	0.0000
Variable Declaration	0.180264	0.0328976	5.4796	0.0000
Overall	0.230766	0.0149072	15.4802	0.0000

Sample Data

All Appraisers treated as a single entity, did not agree well across categories with their replicates. This response is not by chance.

None of the Kappa values are less than zero except Assignment Errors which indicates that the responses are less likely the result of random chance as it gets closer to 1 & the answer changes depending on category.

### Interpretation:

- All appraisers treated as a single entity, did not agree well across categories with their replicates. This response is not by chance
- None of the Kappa values are less than zero except Assignment Errors which indicates that the responses are less likely the result of random chance as it gets closer to 1 & the answer changes depending on category
- $P < 0.05$  for all except Assignment Errors
- Overall Kappa < 0.7 indicates training need

## 4.5 All Appraisers vs Standard (Agreement between all Appraisers together and the Standard)

This provides a statistical summary assessment report which tells if the appraiser's responses are correct & do they agree with the known standard.

All Appraisers vs Standard

Assessment Agreement

# Inspected	# Matched	Percent	95% CI	
14	0	0.00	(0.00, 19.26)	

# Matched: All appraisers' assessments agree with the known standard.

Fleiss' Kappa Statistics

Response	Kappa	SE Kappa	Z	P(vs > 0)
Access and Handling	0.204803	0.0771517	2.65455	0.0040
Algorithm	0.027237	0.0771517	0.35303	0.3620
Assignment Error	0.019252	0.0771517	0.24954	0.4015
Coding Standard	0.244880	0.0771517	3.17401	0.0008
Interface	0.685800	0.0771517	8.88898	0.0000
Performance Errors	0.213302	0.0771517	2.76471	0.0028
Variable Declaration	0.068481	0.0771517	0.88762	0.1874
Overall	0.228400	0.0339050	6.73646	0.0000

### Interpretation:

- All appraisers treated as a single entity, did not agree well across categories with their replicates. This response is not by chance
- None of the Kappa values are less than zero which indicates that the responses are less likely the result of random chance as it gets closer to 1 & the answer changes depending on category
- $P < 0.05$  for all except Assignment Errors, Algorithm Errors & Variable Declaration
- **Overall Kappa < 0.7 indicates training need**

# 11 Conclusion

Conclusion needs to be drawn based on the overall Kappa values from all the above 5 analysis steps (4.1 to 4.5).

The final conclusion from the sample study is that we need to improve the ability of developers to make a better decision in Categorizing the Defects by providing training & bringing in more clarity of the attributes.

For example, if repeatability is the primary problem, then the appraisers are confused or indecisive about certain criteria. If reproducibility is the issue, then appraisers have strong opinions about certain conditions, but those opinions differ. Clearly, if the problems are exhibited by several appraisers, then the issues are systematic or process related. If the problems relate to just a few appraisers, then the issues could simply require a little personal attention. In either case, the training or job aids could either be tailored to specific individuals or to all appraisers, depending on how many appraisers are guilty of inaccurate assignment of attributes.

Attribute Gage R&R was successfully implemented for the first time in CT DD DS AA in some of the projects of DF-PD. Gage R&R was conducted on developers for classifying the Defect Type & on testers for Severity Classification of the defect. The study helped us in understanding Repeatability & Reproducibility issues within & across the appraisers which in turn helped us to device trainings specific to individuals & also the teams. The trainings were devised in such a way that trainings were provided by the developers themselves to the whole team. The defect attributes were shared among developers & trainings were provided with examples with the help of subject matter experts. By this way of training, it increased the knowledge of the developers on the defect attributes. The names of the participants during the study were not revealed anywhere & were used only for training purposes. The study also helped us to see if our RCA's were effective & this can also be called as "RCA on RCA". The study was showcased as a High Maturity Practice during the recently concluded Siemens Business Oriented Process Assessment for DF-PD & the Assessors have highlighted the Gage R&R Study as a Best practice followed in some projects - ["Most projects have performed a Gauge R&R analysis to identify the reliability of the RCA measurement system (e.g. defect classification reliability by various individuals)"].

Based on this study, we also conducted Gage R&R on Integration & System Testers who would classify the defect severity while logging the study. Even here, we had similar findings as that of the previous study & hence trainings were developed on similar lines. We now intend to spread this best practice across projects within & outside DF-PD.

## References

### Web Sites:

<https://www.isixsigma.com/tools-templates/measurement-systems-analysis-msa-gage-rr/making-sense-attribute-gage-rr-calculations/> by Chew Jian Chieh

<http://support.minitab.com/en-us/minitab/17/topic-library/quality-tools/measurement-system-analysis/attribute-agreement-analysis/kappa-statistics-and-kendall-s-coefficients/> from Minitab

<http://www.processexcellencenetwork.com/lean-six-sigma-business-transformation/articles/measuring-the-unmeasurable-gage-r-r-for-transaction/> by James Wells

### Tool Used for Analysis:

Minitab 17

# Better Together

Linda Wilkinson

[linda.wilkinson@evisions.com](mailto:linda.wilkinson@evisions.com)

## Abstract

So you're working on or with agile teams – but are you really agile? Or are your sprints more like little waterfalls, with testing **still** at the end of the process, **still** dependent on one or two people doing the testing, and **still** generating too many bugs too late to fix on time – forcing painful choices between quality and schedule? Too many bugs found in Production in spite of your spiffy TDD processes? Are you automating a test set that wasn't that great to begin with? Have a test team using exploratory testing when they don't know how to explore?

Part of our mandate as testing experts is to raise the quality bar. How do we help make that happen? How do we improve and educate ourselves and our companies in regards to testing?

One of our big challenges as testers is recognizing that maybe, just maybe, we aren't "all that". Our testing is by necessity limited in scope, perspective, and experience. We often do not play roles as the developer, end user, or devops. If testing is solely up to us, we will miss things. Important things. The purpose of this paper is to talk about ways we can collaborate to build (and therefore automate) better test sets. How to negotiate who does what and make testing a team activity. How to move towards or improve TDD (Test Driven Development) efforts. And above all, how to do that quickly, collaboratively, and with an eye towards improving quality.

## Biography

*Linda Wilkinson is Director of Quality Engineering for Evisions, Inc. She has been in the QA/QC field for over 30 years. She holds many certifications (some on tools that no longer exist!), has been a speaker and author, is PMP trained, and has acted on expert panels for process improvement and quality metrics. She is passionate about the quality field, and is dedicated to solving complex problems with simple solutions, mentoring, and learning/brainstorming about new ways to work.*

# 1 Introduction

Agile methodology is no longer new. Many of the test professionals working in the field right now have never used anything **but** Agile technique. As with all project methodologies, however, the definition and methods of implementation for *agile* varies. So what, exactly, is “Agile”?

Agile project methodologies are specifically designed to allow project teams the latitude, independence, empowerment, and authority to create products quickly and responsively through a variety of collaborative techniques. It doesn't matter if it's Kanban, Scrum, Lean, TDD (Test-Driven Development), Iterative, or something homegrown that falls in-between, agile methodology involves working together to get something done efficiently and effectively.

What sets successful projects apart from unsuccessful projects is the team, not the methodology. Specifically, highly collaborative teams are more successful than those who are not collaborative. Many would argue that agile methods require collaboration, and it's true that agile methodology encourages collaboration and therefore tends to push project efforts toward success. At the same time, many agile projects miss their deadlines, do not produce what they were hoping to produce for their sprints, and produce code with varying levels of quality. Some agile teams are superb, some are mediocre. Yet they can be following exactly the same methodology. How can that be? Well, collaboration is, to a large degree, an intellectual concept. While it can be encouraged, it is difficult to demand or measure. Collaboration is not a process; it's a mind-set. A positive, shared mind-set working towards a mutual goal.

While project methodology has been encouraging and educating agile teams to become increasingly collaborative, the testing processes for agile projects have not evolved at the same rate as other agile practices. The purpose of this paper is to present the concept of collaborative testing. The techniques themselves are methodology-agnostic and could be used successfully on any type of project, but this paper will focus on incorporation into agile methodology, as the level of collaboration required is complimentary in concept and thus more easily absorbed into agile teams.

## 2 What problems are we trying to solve?

In a truly agile team, the lines between roles becomes blurred. Members of the team can and do act in any capacity in order to get a given task completed. Oddly, that often doesn't apply to the testing tasks. In many organizations, the development staff turn their completed code over to a testing resource, who develops tests, performs the testing, notes defects, and returns results back to development staff. What's wrong with that picture? Well, it's not really *agile*, is it? It's actually a mini-waterfall, with testing taking place at the end of a sprint (or with its own dedicated sprint), reporting errors at the end of the process, and often delaying either the sprint underway or impacting the start of the next sprint.

In some teams, the QA resource can handle the test automation for the team, automating the gamut of tests from unit through acceptance testing. In this case, the QA resource is truly an *SDET* (Software Developer in Test), with all of the issues in regards to time and perspective experienced by any other developer on the team. Every minute spent automating a test is a minute that is not spent thinking about, exploring, or designing a test. In other words, where this process often fails is in the analysis of what is needed to adequately perform the testing, not the actual automation or testing of the change. The test set itself is incomplete. This can lead to “passed” code that doesn't perform well in the field.

One individual, no matter how talented, and whether a developer, QA resource, or BA (Business Analyst)/PO (Product Owner), has one perspective. In addition, we all have our individual biases. Teams often have issues where (some) testing performed downstream resulted in unexpected error because the code had not previously been used, thought of, or tested quite that way. Again, the test set the team was working with was incomplete.

Overall, the problems we are trying to solve are those that have challenged us for many years. Too many problems found too late in the process.

### 3 Roadblocks

People often don't like change and generally they don't appreciate anything they particularly value challenged. These ideas will challenge what is considered *typical* about some agile team practices, particularly in the area of acceptance criteria. Part of my mandate is to support **any** project with testing services that are both efficient and effective. For that reason, I am completely indifferent to the methodology chosen. Others are incredibly passionate about their methodology, are extremely attached to their ceremonies and current methods for doing work, and it will require some significant negotiation on your part to get them on board with trying something new. What I like best about agile methodology, personally, is that in its purest form, it encourages experimentation and creative thinking. Few companies actually practice that, however. It can be very difficult to get people to think about something they've done one way (perhaps for their entire careers) in new and different ways. They **love** their established way of working. It's comfortable, like an old shoe. What we are trying to do here is to encourage them to throw their old shoes away. The ease at which this is possible will depend on how comfortable that shoe really is and whether it is actually falling apart.

It helps that collaborative testing is complimentary to other accepted agile practices and encourages collaborative behaviors. At the same time, it is a new way to work and it has yet to go through the growth and change of other agile processes. Collaborative testing is an advanced topic for several reasons. First, it requires someone in a position to act as a change agent to get everyone on board and make it happen. Second, the collaborative, intellectual nature of the technique requires a level of sophistication that some organizations may simply not possess. Yet. If your company works with a variety of structured brainstorming techniques and practices, this process will come very easily to you. If not, the process can be like pulling teeth at first because you'll be teaching people to be collaborative in a group. If your company is not accustomed to structured brainstorming technique, it might be helpful to start with something simple, like introducing the idea of "lean coffee" (Modus Cooperandi 2015).

### 4 Let's rethink "acceptance criteria"

.What is acceptance criteria? This definition was pulled from [scrumalliance.org](http://scrumalliance.org):

**Microsoft Press** defines Acceptance Criteria as "Conditions that a software product must satisfy to be accepted by a user, customer or other stakeholder." Google defines them as "Pre-established standards or requirements a product or project must meet."

*Acceptance Criteria are a set of statements, each with a clear pass/fail result, that specify both functional (e.g., minimal marketable functionality) and non-functional (e.g., minimal quality) requirements applicable at the current stage of project integration. These requirements represent "conditions of satisfaction." There is no partial acceptance: either a criterion is met or it is not.*

*These criteria define the boundaries and parameters of a User Story or feature and determine when a story is completed and working as expected. They add certainty to what the team is building."*

*It's generally agreed that acceptance criteria are not as detailed as tests. The criteria are often written by the author of the user story. Often the author wants to keep the criteria to 3-5 points; more might indicate a story that may not fit into a given sprint.*

So what is wrong with this process? Nothing, if it works for you. It has never worked particularly well for me. As a tester, I have found acceptance criteria vague and largely useless to my testing efforts, making

the task of test design a separate process to be performed after the story is placed into a sprint. That means I ask all pertinent, probing questions during that analysis process and the information I gather may or may not be known or shared with the rest of the team, most of whom are concentrating on development, blissfully unaware of the tests I'm going to execute against their code. That means they are not coding to pass those tests. I find this process very waterfall-like and non-collaborative.

In addition, whenever you introduce a constraint into a process, it becomes a potential problem. If more than 3-5 criteria mean something won't fit into a sprint and as a PO I really, really want this feature in the next sprint, I am likely to keep my criteria in the acceptable range whether it really belongs there or not. Wow; what a terrible statement, right? None of you know someone who has bent the rules a tad to get something they passionately feel needs to be a priority done ASAP? I've seen it often, in fact, I've seen it in every company I've ever worked with.

There is also a problem in terms of perspective. The acceptance criteria is supposed to act as gauge of "done". I have never found that to be particularly true. For example, acceptance criteria, written from a user perspective, rarely says "All my existing stuff still works after this change is implemented". Yet I am confident end users don't want a change to break everything else it touches. Perhaps a story cannot truly be estimated with any degree of accuracy without considering the size of the entire suite of tests that need to be run against the feature.

What I am proposing is that acceptance criteria be re-imagined into something more robust, realistic, and useful for creating quality software quickly. There is no point in producing software that meets user expectation, but cannot be maintained or breaks something else. The important and edifying information that comes naturally as part of the test design process needs to be driven out as early in the process as possible and the development staff need to know what their code needs to pass before they start their actual code development. I believe it is time to replace the acceptance test criteria concept with that of a collaborative test set model.

## 5 Why collaborate on testing?

Let's say that we are agreed, in principal, that a more robust definition of acceptance criteria might be in order. Why can't that be done by one person on the team? The PO, the developer, the BA, the SDET? It all has to do with perspective.

The developer arguably knows their own code better than anyone else on the team, and several development team members together might understand the technical aspects of a given feature better than anyone else. They might also have insight in terms of impact to associated or downstream applications that are not widely known by others on the team. Their perspective on what needs to be tested is unique because of their technical knowledge base.

There is, however, a phenomenon that is part of the human condition that tends to make us blind to our own errors. Consider that if I write a report and read it 10 times, I will likely find and correct many errors. The minute I pass it on to someone else, however, they find 5 more. Almost instantly. Some of the errors will be obvious. This is true of developers as well. It is difficult to find error in your own work. The prevailing theory is that we subconsciously do not want to find errors in our own work. In addition, development focus and training rarely centers around "how to find software errors", whereas much of a tester's entire career revolves around finding and reporting error. Books have been written, classes taught, and specific techniques implemented all specifically targeted at finding error in code. Where a QA or SDET resource tends to shine is in taking whatever information is available and specifically analyzing it and asking "what if?" to drive out error conditions or to validate error handling. The ability to not only recognize valid conditions (correct input yields correct output), but to ensure error conditions are handled gracefully, makes the perspective of the testing resource valuable to a collaborative testing effort.

A QA resource is not an end user. QA resources often push back at that statement, as they strongly

feel they represent and support the end user. Not so. They may test like an end user, if their focus is on functional testing, but they cannot represent the end user – they simply do not understand end user workflows and constraints well enough to represent them. I would also argue that it is the development organization that is the primary client in terms of testing efforts. Testing provides information to development in order to help them meet their goals. The resource most likely to represent the end user is likely either the end user themselves, or the person who grooms the backlog or writes the user stories for the team. Most of us have experienced issues with a problem the team considered minor, only to find it had major impact on their user community. That makes the perspective of the end user critical to a good collaborative testing exercise.

These three perspectives form the triumvirate of what is minimally required to have a good collaborative test set. Does that mean no other perspectives are important? No. But this what is minimally required. When training others about collaborative testing, I always discuss the concept of *oracles*, which I define at a basic level as “anybody or anything that knows more about a given topic than you do”. I encourage inviting oracles to the collaboration process. That may be members of another team, a DBA, someone from devops, etc. Whomever you feel would have valuable testing ideas for the feature under discussion should be invited to participate.

When you collaborate on a test set containing multiple perspectives, your team aligns on what needs to be tested in order to pronounce a story done, you push test design to the start of a sprint, and you drive out anomalies in understanding early in the process.

## 6 Getting started

I started working on collaborative testing over 10 years ago after being inspired by a class on Just-In-Time Testing by Rob Sabourin. I adapted what I learned to fit into my environment at the time and that process has continued to evolve as agile methodologies matured and my employers and the teams with whom I worked became more sophisticated.

The basic concepts of collaborative testing are simple. Quality is the responsibility of the team. Testing is the responsibility of the team. In order to be effective and have a good testing effort, multiple perspectives need to be considered. Rather than operating like an old waterfall technique (build code, hand it over to QA), the testing effort needs to begin early in the process and the test, refactor, retest cycle needs to reiterate throughout the entire sprint. Decisions as to who tests what, when, and how are made by the team and negotiated early in the process with no single individual responsible for everything associated with the word test.

This differs from what is done on many agile teams in several important ways. First, test design is not a separate process done after a story is pulled into a sprint by one person, who is also going to be responsible for executing the test set. The test set is built collaboratively and replaces the acceptance criteria on a user story. This reinforces *quality is the responsibility of the team* and *testing is the responsibility of the team*. It ensures important perspectives are part of the base test set.

Collaborative testing consists of two parts. The first part is designing the collaborative test set. The second part is negotiating who does what.

Let's start at the very beginning. If you work on an agile team, it's likely you have a Product owner (PO) or Business Analyst (BA) in charge of a backlog. A backlog is a list of features and bugs that need to be coded. The PO will take of grooming the backlog, which is going through each item and determining priority based on user needs. Depending on your organization, the PO may target what sprint those items go into. Things can get a murky at this juncture, because the process used to size and estimate work varies from company to company. In my current company, once the work is prioritized and the PO has determined a desirable target sprint, a meeting is held to review the proposed work and design collaborative test sets (acceptance criteria). The work is then estimated and negotiation takes place as

to what moves forward into the sprint. Collaborative test sets built for items that return to the backlog do not represent wasted effort, as those were high priority items that are likely to make it into a subsequent sprint, with the test set and estimation work already done. There is no longer a reason for any separate test design process; the base is complete. This saves time on the part of the tester. The development staff know what tests need to pass in order to consider a user story done – before they start writing code. The entire team is aligned on what constitutes *done*. This sounds simple, but the concepts and actually making them happen can be a very complex issue for many agile organizations. Although agile methodology encourages testing early and often, and I don't believe I've ever read an agile treatise that did not recommend QA be involved at the story stage, actually making it so escapes many teams. They know it should be done, but don't know how to do that effectively.

The collaborative test sets are built using a modified Just-In-Time and lean coffee type of structuring brainstorming session.

## 7 Collaborative test sessions

The way I utilize what I learned about Just-In-Time testing over 10 years ago and the way I use it today are very different. I've simplified the process to make it easy for any participant to understand. The original process was taught to a bunch of people who were all testers. So they already understood invalid input, operational considerations, user-specific testing, etc. There were many categories of tests. I've reduced that to two. Valid input (which should yield valid output) and Invalid input (which should be handled gracefully by the system). What do you need to make it happen? A room, the proper attendees (which is anyone with a stake in the game, but minimally the PO, tester, and developer(s)), and a pile of sticky notes along with something to write with.

Collaborative test sessions are fun with the right group of people; very fast-paced and engaging. There are a few rules for the sessions, but just a few (in keeping with agile mindsets):

1. Everyone has to be prepared; if anyone has not read the user story or use case in advance, they are asked to leave (see "shunning" below)
2. The only person with a laptop is the person coordinating the session. This technique does not work effectively if people are on their phones or laptops – they are very focused sessions and require full engagement
3. There are no stupid ideas. If there is a test idea that does not make sense, it can be removed later on. The object is to participate and generate as many test ideas as possible in a short amount of time
4. The sessions can be timed (like lean coffee), but they cannot go over one hour without a break. If your stories are broken down to their smallest common denominator, the team may be able to build a test set in 10 or 15 minutes, which means they can get through 4-6 stories in an hour. Look at your throughput – how many stories does your team typically get done in a sprint? On our teams, it's typically 25 stories in a 2-week sprint. That means we put aside 4-6 hours during our planning to build collaborative test sets.
5. Anyone who is not prepared, does not participate, or sidetracks or torpedoes the session gets "shunned". That is, they are either asked to leave, or not invited to future sessions. The process is dependent on everyone being present in mind and body, focused on the task at hand. One of my peers, a development director, skyped me a few weeks ago to tell me he had been shunned from a collaborative test session for being unprepared. And his staff really enjoyed chivvying him from the room. We both laughed about it, but he learned from it and his team enjoyed themselves, felt empowered, and got down to business.

The session coordinator is usually the testing resource on the team, but this role can be assumed by anyone. It is up to the coordinator to invite pertinent players. Again, the players minimally include

developers associated with the user story or use case, the PO or user representative, and the testing resource.

The session coordinator establishes wall space labeled VALID, INVALID and QUESTIONS. VALID scenarios are those where valid information is input with an expectation that some valid system response or transformation will occur. Often these are referred to as *happy path* tests. INVALID scenarios are those that validate error handling. Invalid input is handled in some graceful way, such as returning an error message. "QUESTIONS are things that come up that no one in the room understands or can explain in enough detail to formulate tests. If, as part of the user story or use case a bank of regression tests need to be executed, that is specified under VALID conditions. The statement might be "I need to verify all of the existing X functionality still works once this change is made". There is no need to select tests or get into further detail during the session. The team builds out their test set by having the coordinator read off the user story or use case and generating test ideas. They do this by stating their test idea out loud, writing it on a sticky note, and placing it on the board as either a VALID test, an INVALID test, or a QUESTION. With a good, invested team this process is actually fun and little hectic. The idea behind stating the idea out loud is that one idea often spawns another idea. The coordinator ensures both valid and invalid cases are getting attention. For example, if there are many valid tests and only one invalid test, they might ask "What can we do to break this?". It is normal, particularly from a tester's perspective, to have many more invalid than valid tests. It is also up to the coordinator to ensure everyone is participating and that everyone's voice is heard ("Hey, John, what do you think?"). There's a difference between not participating and not being able to get a word in edgewise. The session ends when test ideas start trickling to a close; again, it's up to the coordinator to call a halt when it makes sense. The coordinator then gathers the stickies, puts together a list, and sends it to the participants, asking them to review and update it (if they wish) by X date. Any questions generated by the team are also put on the list and the coordinator takes point in getting those questions answered and updating the team with answers.

This, then, represents a test base that contains a variety of perspectives and that forms the base for all testing to be done against that user story or use case. In my particular company, the testing resource stores the base in a centralized test management tool and translates the scenarios to test stubs that can be automated or arranged into logical charters for session-based test management. This also allows us to pull reports on demand and metrics as requested. Our test management tool interfaces with JIRA (a common agile task management tool) to further enhance management of testing functions.

This process takes the analysis tasks normally performed by the developer, test resource, and BA/PO and performs them in concert, collaboratively, driving out questions and establishing the test base before code is written. It adds minimal time up front, but that time is recovered (plus) as the sprint progresses. For agile shops that would like to move to TDD at some future date, this is a very good first step, as it establishes the habit of designing tests prior to writing code.

Companies heavily invested in innovation like brainstorming techniques such as this. At my current company, all of our agile teams, POs, and testing staff have been trained to use this process. It has been so popular and saved so much time that teams are now using the technique to collaboratively write user stories. Because all of the pertinent consumers of the user stories are together in the same room, all questions get identified or answered up front, allowing for better, more consistent information and common understanding of what needs to be developed and tested. Overall, this technique is just a quasi-structured brainstorming session.

## 8 Collaborative testing

To my mind, this is where the true nature of agile kicks in. The team, as one entity, negotiates who will test what and when. Normally, the nature of the tests themselves makes division of duties obvious. If, for example, I have a highly technical test that wants to validate a new service has registered itself and can be consumed, it's likely that will be a unit test performed by a developer. An end-user would be unlikely to understand what that even means, let alone have the technical ability to verify it. In some cases, after

division of duty has been made, the tester (whether developer, QA resource, or end user) will further drill down the tests in their pieces of the acceptance criteria until it makes sense for what they are doing. This is especially common in TDD shops. The developer may have a number of technical tests they want to code and run based on user story sub-tasks that need to be completed and that feed the test in the acceptance criteria. In fact, some TDD shops take their share of the acceptance criteria tests and run another collaborative test session with tech staff only to further drive out unit-specific tests. Remember that a collaborative test session can take as few as 10 minutes.

The testing resource, if an automation framework exists, can perform the same tasks as TDD development staff – that is, code their tests prior to actually receiving the code. In those agile organizations where testing staff perform all of the testing or automation tasks, their testing base is complete and they can proceed as normal.

The end user (or representative) now has a set of tests they agree constitutes *done* and adequately covers what they feel is acceptable – after all, they were part of the process. What does the term *done* actually mean? The definition of *done* is determined by the team. A simplistic definition would be code is complete, testing is complete, and the user has accepted it. So the user representative can either execute those tests they consider important themselves as code is made available, or ask to see test results from the team. This results in less thrash at the end of the process, as there are no surprises for the team as their user representative executes scenarios they missed.

In all cases, regardless of who does what, the definition of done is more concrete. Even in those environments that utilize solely manual test technique, analysis time is saved, the test base is more complete, and the development staff have the opportunity and advantage they need to produce higher quality code, simply through the expedient of understanding what tests will be run against that code.

## 9 What can you expect from this process?

Before you decide to invest time and effort into any new process, you have to be sold on the idea that it has some benefit to you, your coworkers, and your company as a whole. I'd like to talk about some of the benefits I've experienced and encourage you to experiment with collaborative testing yourselves.

I have had the great good fortune to work with fantastic testers and every team I've managed and worked with has kicked butt in terms of quality. Maybe not at first, but with hard work and through heavy collaboration and a willingness and strong desire to "raise the bar". It's up to you and your own company to determine what that bar is. It differs from company to company. Whether the team was manual testing only, automated testing only, or (more commonly) a combination thereof, the testing efforts have always been robust and the error rates in the field have typically run at 3% or lower. That number comes from dividing the total number of errors found in the field by the total number of errors found during the sprint (or testing effort, if you don't use agile technique). In addition, our "measure of goodness" is that none of the errors found in production require immediate code changes or a "dot-one" (.1) release.

That number might tell you that I believe in and support metrics for process improvement and you would be right. Metrics, pulled the right way and for the right reasons, can tell a story. Whenever we've implemented and tried a new technique to see if we could raise our own quality bar, we have had common metrics to help us figure out if we're being successful or need to try something else.

With collaborative testing, since we were using some of the time and efforts of other teams outside our control, it was particularly important to management staff to determine if collaborative testing was a good investment of time and resources (which equates to money). What we found was this:

1. In cases where the error rate was already gratifyingly low in production, the primary benefit of collaborative testing was the savings in test design and execution time. We were able to cut 3-week sprints to 2-week sprints with no loss in production quality.

2. In companies without a robust testing process, using collaborative testing decreased the error rates in production by up to 20%.
3. In every case, whether robust or not, regardless of project methodologies, the error rates of code submitted by development staff decreased by up to 40% or more. That means using the same measure as above, we measured the number of errors found in the QA environment divided by the total number of errors found during the project life of the product under test. In the case of one team, it decreased from 400% (an average of 4 errors per test were found) to 20%. Within 6 weeks. This has been a huge win in every company I've introduced to this technique; the lower error rates allow for code to move through the testing process much more quickly and involve much less development time for coding fixes. In every case, it allowed the agile teams to condense and create working code ready for production in less time.
4. User sign-off has been obtained more quickly. Although agile process requires a definition of done up front, many companies flounder with this, particularly if their test set is not part of the user story contract (e.g. acceptance criteria). Acceptance criteria can pass and the product might not be ready for prime time (such as breaking major features elsewhere, negatively impacting response time, etc.). I worked for one firm where getting an approval from the user organization took months. The collaborative process reduced that to days, and the user organization chose to stop testing altogether (which saved them time and money), merely asking to be copied on test results. Having a more definitive definition of *done* helps the entire team focus on making that happen.
5. It allows development staff an opportunity to morph into TDD more painlessly. In my current company, we started with collaborative test tests and have had two teams move into TDD with relatively few hiccups and two more that are anxious to join the parade.
6. Teams start to use collaborative techniques for other efforts. The agile teams I work with have found value in the targeted brainstorming learned through collaborative testing and have applied it to many of their other tasks – from forming temporary SWAT teams to address specific problems, to documenting user stories, to coming up with discussion points for innovation.

## 10 Learning more

I can't refer you to any books; there aren't any on this subject. But I've included a few websites that address brainstorming technique if you've never tried it. The way I learn is to pick a team, a place, and a time, and experiment. This particular set of experiments started over 10 years ago.

I always start collaborative test efforts by training all of the staff. All of our agile team members have been trained in this technique. I try to make that fun and very "outside the box", and it's been very successful. For example, in late November, I took what is arguably one of the Worst Movies Ever Made – Santa Claus vs the Martians, and made one of the premises into an epic. That is, "I, as Santa Claus, want to deliver toys to the Martians". It was up to the team to brainstorm on what that would entail. I let them choose their own coordinator and halfway into the process, visited them as an oracle – Santa Claus – to answer any questions. A lot of candy canes were flung about. Overall, it was fun and the team learned to use the process in a painless, non-threatening way. The test sets were pages long (which was great, as it showed the team how much can be covered in a short time) and we were all entertained reading the collaborative criteria results.

Silly as that exercise was, everyone learned what they needed to learn and the results have been everything we had hoped and more. The nature of our work requires change and constant improvement. I hope the potential and possibilities of this technique resonate with you and you do some experimenting of your own!

# References

## Web Sites:

Lean Coffee Organization. <http://LeanCoffee.org> (accessed July 21, 2016)

Brainstorming methods. <http://mindtools.com> (accessed July 21, 2016)

Information on testing biases: Kilby, Richard Aug 27 2014. Paragon blog Aug 29, 2014, [How We Miss Bugs](#).

<https://www.paragon-inc.com/resources/blogs-posts/~media/Files/How%20We%20Miss%20Bugs.ashx>  
(accessed July 21, 2016)

Rob Sabourin classes: [http://lets-test.com/wp-content/uploads/2011/10/2012\\_05\\_06\\_Just\\_In\\_Time\\_Testing.pdf](http://lets-test.com/wp-content/uploads/2011/10/2012_05_06_Just_In_Time_Testing.pdf)

# Breaching Barriers to Continuous Delivery with Automated Test Gates

Chris Struble

cstruble@vertafore.com

## Abstract

Continuous Delivery may have come out of companies that were “born on the web”, but what if you are in a regulated, mature industry with sensitive customer data, where every change to production must be approved by a review board? How can you design, deploy, and execute automated tests that will work in environments that cannot be directly accessed by your team? How can you breach barriers to Continuous Delivery in a business context where the walls between development and operations are rigorously maintained?

Vertafore is an intermediary in the insurance industry, providing information solutions for insurance agencies, carriers, managing general agents, and state agencies. When our team was tasked with enabling Continuous Delivery, we faced all of the above challenges. We overcame them using Jenkins, Artifactory, and Chef to build a Continuous Delivery system with automated test gates for promoting product and test code from Development through Staging to Production.

Our Continuous Delivery system supports automatic promotion based on test results, or manual approval. Because of this flexibility, we were able to get adoption from product teams at each of the different stages of Continuous Delivery process maturity. It also provides a common way to deploy and test that our developers, testers, and release engineers can use in the same way in each environment. The system has enabled our company to increase the size and frequency of releases.

This paper will cover the key patterns that our team developed or discovered in our journey, which can help you in creating your own Continuous Delivery system, regardless of the tool set.

## Biography

*Chris Struble is a Senior Software Engineer in Test at Vertafore in Bothell, Washington. He has 20 years of experience in software testing and infrastructure, build, deployment, and test automation. He holds Masters Degrees in computer science from Walden University and mechanical engineering from the University of Houston. Currently, he is focused on improving software quality and Continuous Delivery. He lives in Renton, Washington, with his wife, two teenagers, and three cats.*

*Copyright Chris Struble 2016*

# 1 Introduction

The ability to deliver software changes to users quickly and reliably has become as important as the features of the software itself. Today's users have smart phones, smart televisions, tablets or other devices with applications that update continuously, accessing web applications hosted on cloud environments that themselves are updated multiple times a day without the end user even being aware.

These changes came about because many leading companies have implemented Continuous Delivery practices. Jez Humble defined Continuous Delivery as "a set of principles and practices to reduce the cost, time, and risk of delivering incremental changes to users". Organizations doing Continuous Delivery typically have automated deployments, delivery pipelines, and extensive automation of infrastructure, deployment and testing across all environments, including Production.

Vertafore is an intermediary in the insurance industry, providing information solutions for insurance agencies, carriers, managing general agents, and state agencies. The company's products touch every point of the distribution channel. Their engineers support dozens of products that combine to make the broadest and most complete portfolio of software in the insurance industry. Technology is rapidly transforming the insurance industry, inciting companies to move away from large monolithic applications and toward web services that can be released more quickly and reliably. Speed and reliability are critical in an industry-- venture capitalists have invested more than \$2.2 billion dollars in the industry over the last five years and they anticipate a return for their investments.

What has not changed are the strict rules about handling of confidential customer information in the insurance industry. An example is the Health Insurance Portability and Accountability Act (HIPAA), which includes network security rules that require companies providing solutions to limit access to personal health information (PHI) of individuals.

To comply with such rules, Vertafore limits access to its production environment to essential personnel, such as system administrators and release engineers. Any changes to production must be approved and documented by a change review board and must be initiated by an operations release engineer during approved maintenance periods only. Most of our software development engineers have no access to production and our network rules do not allow updates to be pushed into production by non-operations staff.

These constraints might appear to be a significant barrier to doing Continuous Delivery. However, in reality, Vertafore has been doing Continuous Delivery for several years. We just do it differently.

## 2 Case Study: Continuous Delivery at Vertafore

Our journey toward Continuous Delivery started in 2012, when Vertafore formed a team called Continuous Quality in our Bothell, Washington office. This team, which included the author, was initially focused on improving test automation for three of our largest management systems: AMS360, BenefitPoint, and Sagitta.

Agile development already was well established at Vertafore, with our product development teams already using a Scrum process, and most using Microsoft Team Foundation Server (TFS) to track work items, run manual tests, and store and build source code continuously. Many of our development teams also used behavior-driven development (BDD) to create automated acceptance tests with SpecFlow.

Our team's first project was to develop new UI tests in C# with Selenium WebDriver for .NET, based on SpecFlow scenarios provided to us by the product development teams. We set up a Jenkins continuous integration build server with Windows test agents to run the test cases on demand.

As we improved the test cases to make them more stable, we realized we could abstract much of

our code to a framework. We called the framework “Firefly” and shared it internally with our product development teams as a NuGet package.

Our team's first NuGet repository was a file share. As more product development teams began to use our framework and began to create their own NuGet packages to share .NET code with each other, we needed a repository manager that could scale with the growing demand.

We decided on Artifactory Pro. One factor driving that decision was that our team's charter had just expanded. We were tasked to implement an automated Continuous Delivery pipeline to support an Agile Release Train for Vertafore Agency Platform, a suite of new products designed to work together in a services-centric architecture. The pipeline needed to work across multiple environments, some of which are isolated.

A key feature of Artifactory Pro that would enable our pipeline was the ability to replicate artifacts between multiple Artifactory instances. Artifactory Pro had good integration with Jenkins, NuGet support, and a REST API, each of which was an attractive feature for us. We set up an Artifactory Pro server and were soon using it to host NuGet packages and other binary artifacts built on our Jenkins server.

Our Agile Release Train is a release management process that is part of the Scaled Agile Framework. Agile release trains occur on a regular schedule, with high quality, but with variable scope. In the release train metaphor, releases are equivalent to trains, while individual software products are equivalent to freight cars. If the product isn't ready, the release continues without it, but, no worries, the product can always go out with the next release.

Our team held sessions to visualize how a fully automated Continuous Delivery pipeline would work. Our design used Jenkins and Artifactory as the foundation and made extensive use of available plugins to implement parts of the pipeline and custom tools to implement any missing pieces. We called our delivery pipeline concept the Automated Deployment System (ADS) and the name stuck.

Figure 1 shows how ADS works to deliver our software through the development environment.

The ADS delivery pipeline begins when a software engineer checks in code to TFS. This triggers a Jenkins build job, which builds the code, runs unit tests, and publishes the artifacts to Artifactory.

Our Artifactory Pro server has artifact repositories, including a Development repository and a Release repository. Artifacts are always published to the Development repository, never to the Release repository.

Once the code is published, a “gate evaluation” Jenkins job is run to analyze the test results file from the unit tests. The gate evaluation job takes several parameters, such as the build being evaluated and the pass score, i.e., the percentage of test cases that must pass for the build to pass the unit test gate.

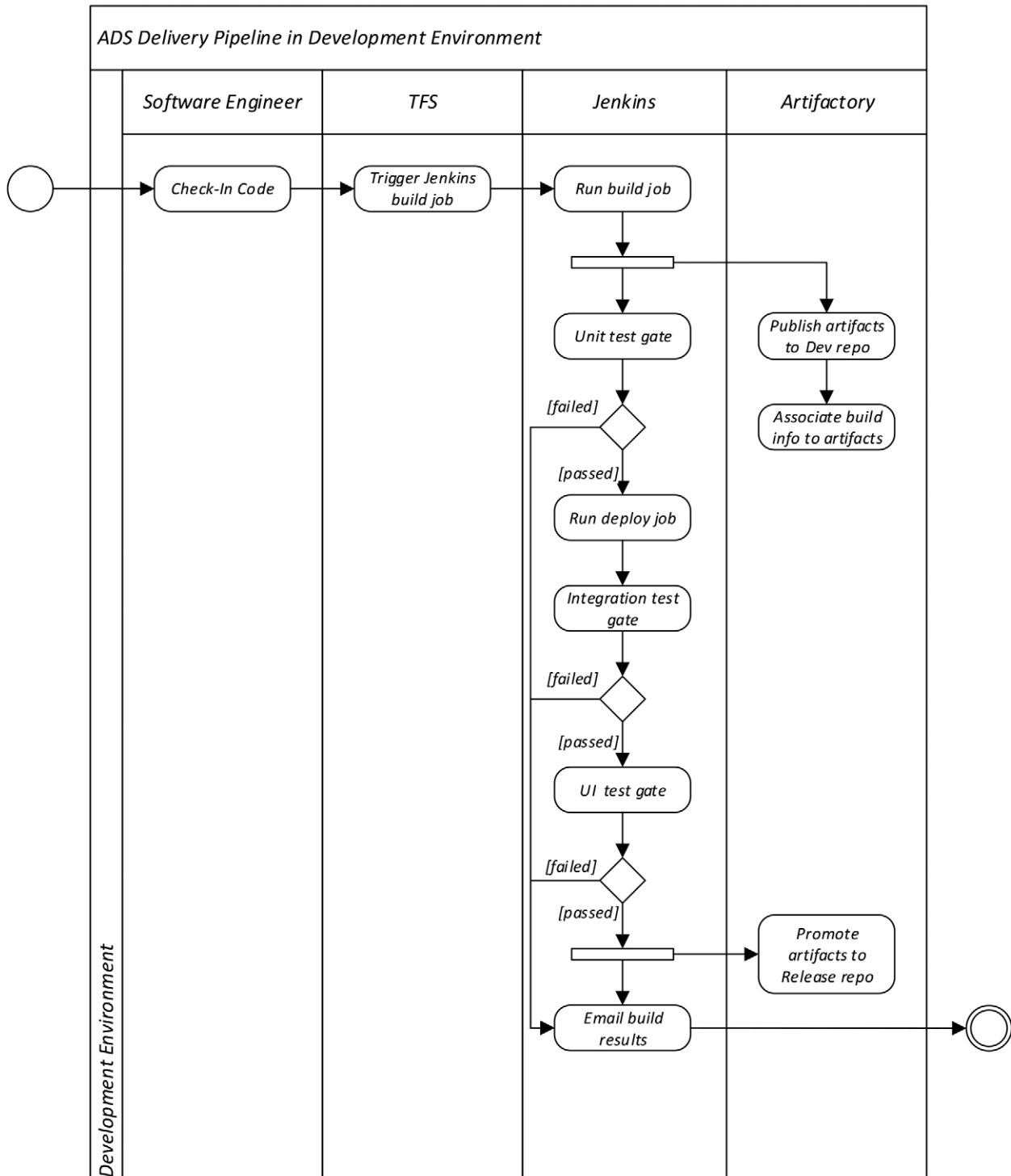
If the build passes the unit test gate, a deploy job is run to deploy the product into the development environment. After deployment, additional automated tests may be run (such as web service tests, integration tests, endpoint validation tests, and user interface tests), each of which may have test gates run to evaluate the test results.

If the build passes all of the test gates, the artifacts are automatically promoted to the Release repository. If any of the test gates fail, the build fails and the artifacts remain in the Development repository.

Product development teams are able to specify which test gates to include in their workflow and the required pass score for each gate. Typically teams require a pass score of 100%.

The development teams also are able to use a manual gate in their workflow. Providing support for manual gates allowed us to support development teams who were not as far along in their automated testing, or who were not comfortable with an automated process making release decisions.

To deliver our software all the way to Production, we had to use a different process than we used in Development. To explain why we had to do that, I need to digress and tell a short story.



**Figure 1 – ADS Delivery Pipeline for Vertafore Agency Platform in the Development Environment**

### 3 Story: A Game of Gates

Imagine for a moment that you are in Europe, sometime in the Middle Ages.

You are a merchant, the owner of a cart filled with trade goods. You want to get your goods to the market. The market is in the courtyard of a huge castle, surrounded by a series of stone walls and gates, each larger and stronger than the one outside it. You have a letter from a buyer who lives inside the castle, but you have never met her or previously been to this castle.

On a perfect day, all the gates of the castle would be open and you could just drive your cart into the market, deliver your goods, get paid, and go home.

When you arrive at the castle, the first gate is wide open. The guards barely look at you as you enter the outer courtyard.

Then you notice that the second gate is closed. The guards at the gatehouse are not smiling. You ask the captain of the guards to open the gate. He refuses. It seems there are rumors of assassins. He doesn't know you and won't let you in. Nothing personal. Orders from the king and all that.

For a moment you contemplate knocking in the gates with a battering ram. Then you remember that these people are not your enemies. You want to do business with them, not fight them.

Eventually you get an idea. "If you won't let me in, will you take a message to my buyer?" you ask the captain of the guards. "Tell her to come out of the gate once each day. If she wants to buy my goods, she can take them inside. Otherwise, she can leave them here."

The captain considers your proposal and eventually agrees. But he warns you. "Behind this wall is the inner gate, the largest of all. The captain of that gate isn't as friendly as I am. He only opens the gate at certain times of the day and he personally inspects everything. He won't let anything through unless there is paperwork listing every single item and every person that has come into contact with it. Can you do that?"

You agree. Each morning you bring your goods to the castle in your cart and sometime later your buyer comes out with her own cart. Sometimes she isn't able to get your goods past the inner gate, and sometimes she doesn't want them at all. But enough of your goods get through to make it worth the effort. And though you never meet your buyer face to face, you both get very rich.

### 4 Getting to Production

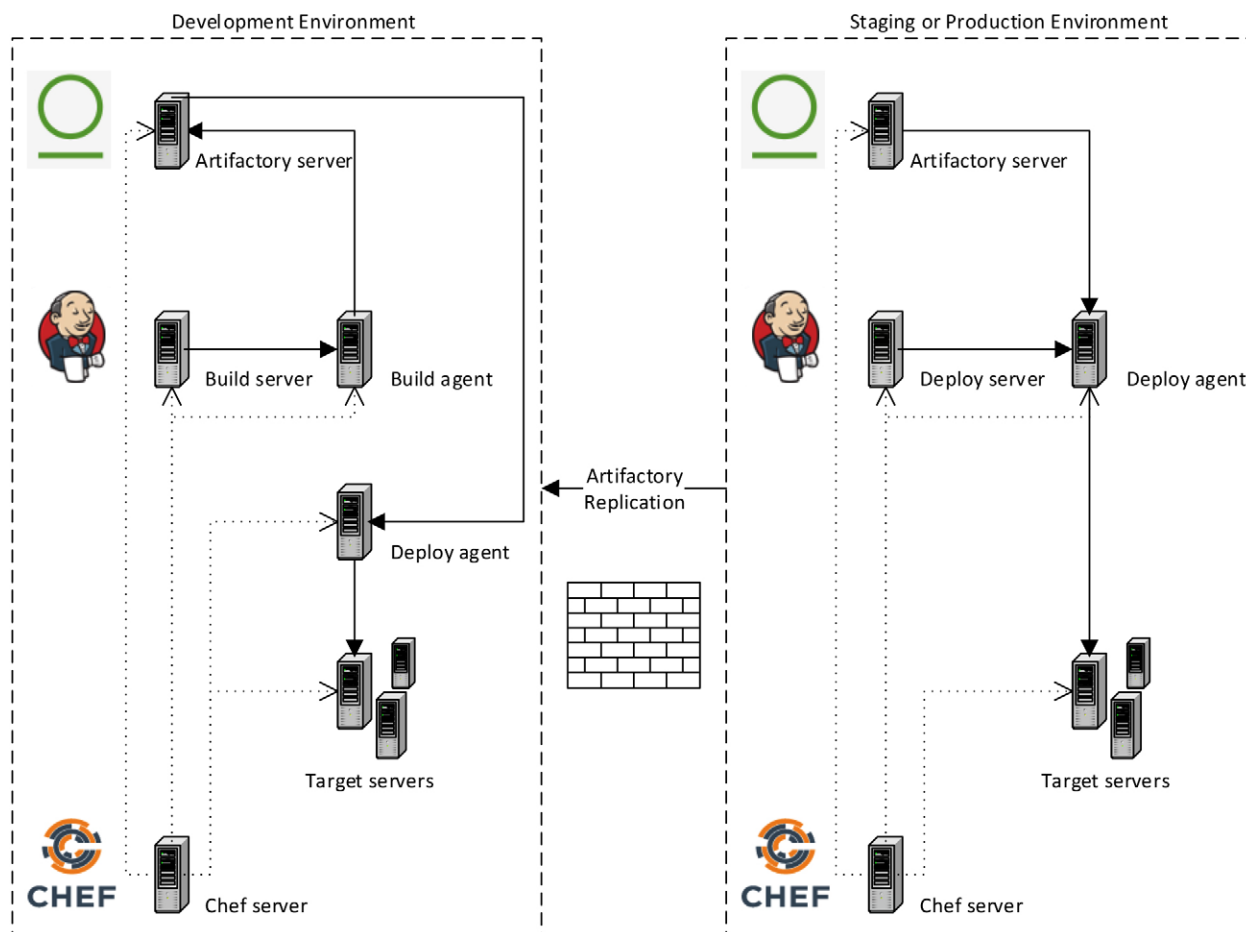
To extend the ADS delivery pipeline to Production, we had to have conversations with our security and operations teams. Vertafore has a Staging environment that our release engineers use to practice releases and identify deployment issues before releasing to the Production environment. Staging and Production are both isolated. Jenkins does not deploy directly into those environments, and a security exception was required

Like the merchant's request to the captain of the guards to open the castle gates in the story, our request to open a security exception for Jenkins was initially denied. For numerous security and business reasons, push-button deployments into Staging and Production are not allowed. But we were told that a deployment initiated from inside Staging or Production would be acceptable.

Our solution was to place an Artifactory Pro server inside Staging and Production and use the replication feature to request artifacts from our main Artifactory server in Development. Because replication requests are initiated from inside Staging or Production, we could meet Vertafore's security requirements. Figures 2 and 3 show the architecture and activity diagrams for our solution.

Every 10 minutes, the Artifactory Pro servers in Staging and Production initiate requests to our main Artifactory Pro server in Development for any new artifacts. Only the Release repository is replicated, to ensure that only promoted artifacts are available to be deployed in Staging or Production.

#### Continuous Delivery Pipeline Architecture

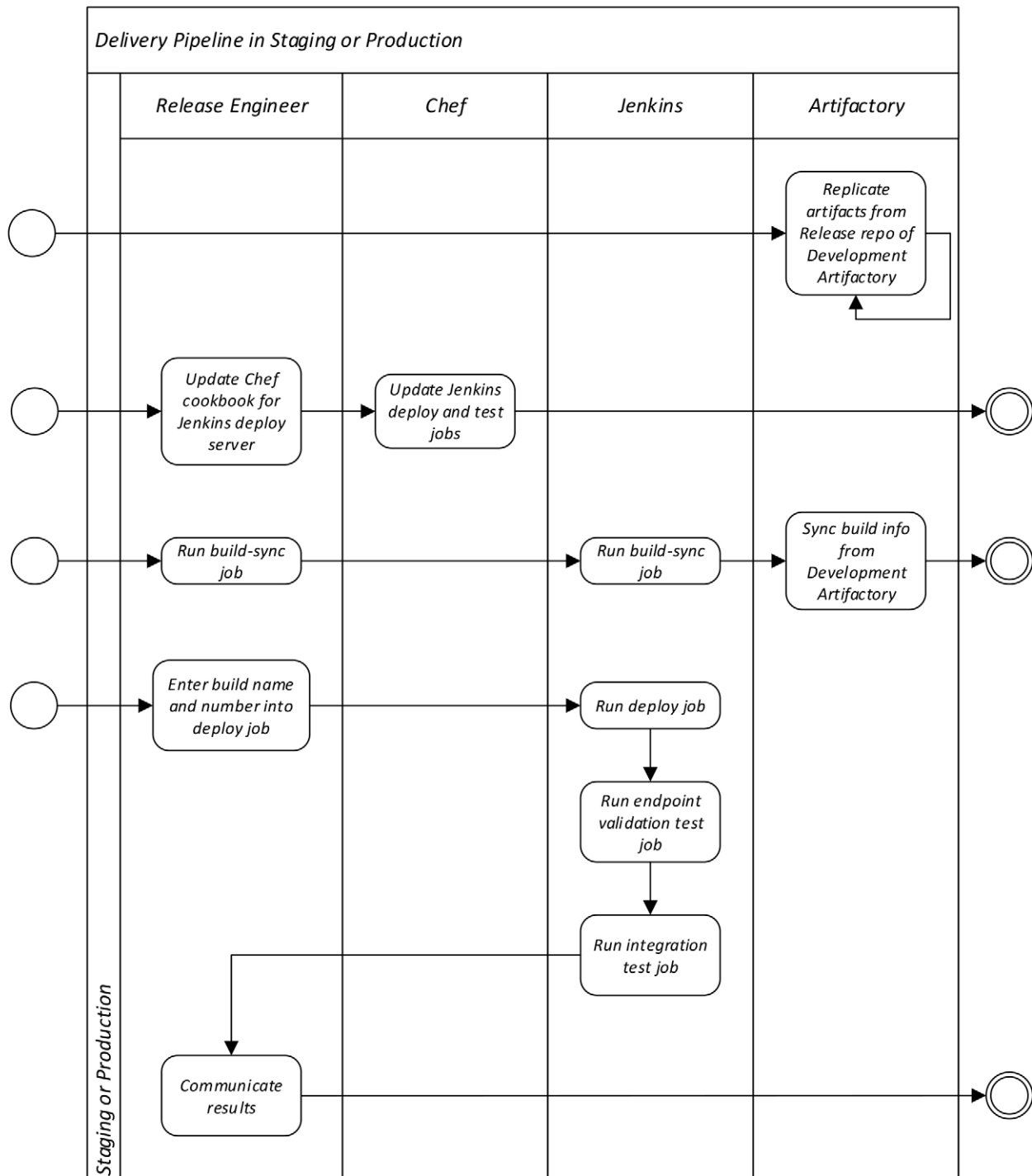


**Figure 2 – Architecture of ADS Delivery Pipeline for Vertafore Agency Platform**

The first product to use ADS to deliver to Production was contained in the December 2013 release of Vertafore Agency Platform. Six months later, 90% of the products on the Agile Release Train were using ADS to deliver all the way to Production. The only exceptions are legacy products that still require manual deployment.

As more products used ADS, a new problem emerged. Small differences between the Jenkins and Artifactory servers in Development, Staging, and Production, and between the versions of the Jenkins jobs in each environment, were causing deployment failures. These occurred because machines had been configured manually, often by different people.

We turned to Chef, an infrastructure automation tool, to help us solve this configuration problem. Chef uses scripts called “recipes” to ensure that machines are in a desired state. We created Chef recipes to configure our Artifactory and Jenkins servers, agents, and jobs, so that deployments and tests would work the same across all environments. Prior to each release, we update the Chef recipes and apply the updates to the Artifactory and Jenkins servers and agents in each environment.



**Figure 3 – Activity Diagram of ADS Delivery Pipeline in Staging and Production**

Since we started using Chef to configure the machines in the ADS delivery pipeline, deployment failures in Staging and Production have declined significantly.

By 2016, ADS grew to more than 800 Jenkins jobs, with more than 8000 artifacts being published to Artifactory each month. It has made releases more predictable and reliable. The Agile Release Train has consistently delivered at least one release a month for over two years, with an average of 10 products per release, with 100% of packages delivered by the development teams being released successfully.

Monthly releases may not seem very fast, especially compared to large web companies that release multiple times per day. But when needed, one of our software development teams can build, publish, test, promote, and replicate a fix to make it available for release engineers to deploy in Production very quickly, in less than 30 minutes in most cases.

## 5 Patterns

The Continuous Delivery system we built is very specific to Vertafore. It had to be. When we started, there were few commercial or open source tools available for Continuous Delivery and none that we found could work with isolated environments. More tools are available now, but many still are designed for building source code in the cloud and deploying to production servers in the cloud, both of which are not options for us.

What about your company? If your company has strict business and security rules, as ours does, you may have to create a customized system. Below are a few patterns we found helpful in this effort.

### 5.1 Separate Build and Deployment Code

Building and deploying software are distinct tasks. The code that does these tasks should be separated as much as possible.

If you have to rebuild your product code because of a change in the deployment process, a change in an environment, or an incorrect target server name, your deployment code and product code are too closely coupled.

Getting this wrong is not a catastrophe, but it does mean your product will build more often than necessary, causing delays and wasting machine time and artifact storage. Product code can be very large, while deployment code is typically very small. The less code you have to rebuild the better.

### 5.2 Standardize Machine Types

Machines in a delivery pipeline need to do different types of work, such as building, deploying, and testing. It is essential to standardize machine types based on the work they need to do and to use software to ensure they have the tools needed to do that work. Without standardization, the default is to configure machines by hand, on the fly, leading to uncontrolled differences between machines that grow over time and to unexpected failures.

We have a few Jenkins machine types: build servers, deploy servers' build agents' deploy agents, generic test agents, and UI test agents. Each of these machine types is defined by different Chef recipes. When we need to update one of the machine types, we update the corresponding Chef recipe and apply it to all the machines of that type at the same time. This approach enables resilience.

Your list of machine types will surely be different, but the point here is to have a list, not to let every machine be unique in an uncontrolled manner.

### 5.3 Separate and Promote Artifacts

Any delivery pipeline needs a mechanism to identify artifacts that are approved for release and to prevent unapproved artifacts from being released. A simple way to do this is to keep artifacts in separate repositories, such as Development and Release.

Promotion consists of moving artifacts from Development to Release. The promotion process should automatically tag the artifacts with information about who promoted the artifacts and by what process.

Repository separation and promotion can be used with any repository manager to ensure that artifacts cannot be released unless they have passed through the steps of the delivery workflow.

### 5.4 Automate Promotion with Test Gates

In waterfall software development, quality gates are project milestones that exist between phases of a software lifecycle. Validation of a quality gate typically involves a high-level review of checklists to determine if the project can move on to the next phase.

Test gates have a similar purpose, but are automated. A test gate evaluates test results or other output from a build and determines if a promotion criteria was satisfied. If the criteria are satisfied, the build and its artifacts are promoted through the pipeline, automatically.

What test gates would you want your software to have to pass through before being approved for release? Unit test gates, functional test gates, security gates, code coverage gates, and performance test gates are just a few of the possible gate considerations.

Any pipeline will need to have a manual gate process as well. Test gates can fail like any other automation, and they depend on automated test cases or other assets that might not exist for every product you have to move through your pipeline. Even when they exist, automated test cases may produce false failures due to incorrect assumptions in the test criteria, causing a test gate to fail even though it would have passed if the test case had been fixed. Only humans can make these type of decisions.

### 5.5 Artifact Retention

Any delivery pipeline will need to include a strategy for retention and cleanup of artifacts. This is driven by storage limitations, but also by performance. Some questions you will need to answer:

- How long do we keep artifacts that were never promoted?
- Should we delete artifacts that failed a test gate?
- How long do we keep artifacts that were promoted, but not deployed to Production?
- How long do we keep artifacts that were deployed to Production?

There is no right answer to any of these questions, and you probably don't need to answer them right away, but unless you have an unlimited storage budget, you will have to answer them eventually.

Our pipeline discards un-promoted artifacts after three months. That frees up disk space while giving development teams enough time to get their artifacts tested and promoted. We keep promoted artifacts indefinitely, at least for now. If we get to the point where we have to clean those up as well, we will probably retain them for several years.

## 5.6 Automated Rule Compliance

In your delivery pipeline, you may want to discourage your users from doing things that your pipeline tools do not prevent, but that nevertheless are a bad idea. A simple way to do this is to have an automated script check for rule violations.

For example, we want Jenkins users to configure their jobs such that builds are cleaned up. Otherwise, our build agents get filled up with old builds. So we created a job that checks all the other jobs on our build server and sends an email with a list of jobs that are not being cleaned up properly.

We have a similar job that checks to make sure all the jobs that publish artifacts to Artifactory clean up published artifacts for jobs that have been discarded.

Another job checks to see if anyone has configured their Jenkins job to publish directly to the Release repository without going through the promotion process.

All of these rules were created because people did things that actually caused problems. Once we had automated jobs running daily to check for violations, all the violations were eliminated, with new violations appearing only occasionally, mainly due to new users not being aware of the rules.

## 5.7 User Engagement

When building a customized delivery pipeline, it is essential to engage the software development teams that will be using your pipeline, explaining the benefits of using it and how to use it effectively.

Our team has a series of wiki pages that explain how to use our tools, and we give periodic training sessions to new users, including classes at Vertafore's annual developer training conference. We use email lists for support requests and announcements of maintenance windows and technology upgrades.

Finally, we have governance bodies for best practices, including a Chef developers' group and a Center of Excellence for Release Engineering processes and tools.

## 6 Final thoughts

Delivering software to customers is not an optional function: it is the reason we are all in this business. You have to do it, so you might as well do it well. Continuous Delivery, at its core, is a commitment to continuously improving the velocity of software releases, while reducing the cost and risk of delivering software. If there are barriers in your business that prevent you from improving every part of your delivery pipeline right now, don't let it stop you from being creative and transforming what you can transform. Tomorrow, when better tools are available, you will be in a better position to understand how to deliver to your customers in the future.

## 7 Acknowledgments

I wish to acknowledge the contributions of everyone who has been part of our Continuous Delivery journey: Raul Alvarez, Cody Dockens, Saratha Prabu, Rajani Tadanki, David Echols, Rasheed Usman, Lowell Young, Denzil Dwelle, Cameron Straka, Ebencilin Chandradhas, Venkat Chilakala, Tom Sawin, Rita McCann, and George Tsang.

I also acknowledge our colleagues from Ernest Mecham's group at our East Lansing, Michigan office. They have been on their own Continuous Delivery journey, using Bamboo instead of Jenkins, but otherwise overcoming many of the same challenges we have, with excellent results.

## References

Jez Humble: The Case for Continuous Delivery, 13 Feb 2014  
<https://www.thoughtworks.com/insights/blog/case-continuous-delivery>

Scaled Agile Framework  
<http://www.scaledagileframework.com/>

Artifactory Build Sync User Plugin  
<https://github.com/JFrogDev/artifactory-user-plugins/tree/master/build/buildSync>



# Automated Testing in DevOps

Rajesh Sachdeva

Optum technology Inc.  
Rajesh\_sachdeva@optum.com

## Abstract

Can you test your application without it being deployed to a static environment like development, test, stage, etc. and execute all your tests (no matter what the size) as many times you want in a day? **The answer is Yes.** In a DevOps or Continuous Delivery Model (CDM), automated tests (unit, smoke, regression, API, business logic tests, E2E, etc.) are hooked up with an application build deployment pipeline and are executed in the background by a Continuous Integration (CI) server. The developers or quality engineers write a test first, but are not forced to execute the tests manually. The tests are run automatically by a continuous integration tool. If builds are happening 50 times a day, then the CI server can execute your regression suite of any size 50 times a day, and promote the code to higher environments automatically.

At Optum Technology Inc., for one of our programs we execute more than 1.5 million tests per day. To enable this, our QC (Quality Control) team has transformed into a **quality engineering** team. We are now capable of more frequent releases to production with no testing phase. This paper details how our testing pyramid differs from the legacy waterfall model and what advanced technology and tools are available in the market that helped us to create the test environment on the fly, in the cloud or a transient environment, before deploying to static environments. Docker and Mesos are such examples which provide this flexibility; they can spin up an app server, DB server and deploy the code to the transient environment within a few seconds. You can run thousands of tests in minutes for each build. This technique potentially reduces the time wasted resulting from manual test execution in physical environments, by eliminating the entire testing phase in a software development cycle in scaled agile delivery. The testing feedback is given to the developer in the same build cycle, without having to deploy the code to the development environment.

## Biography

*Rajesh is working as a Senior Project Manager at Optum Technology Inc and is responsible for real-time test automation in DevOps, Microservices API Certification and testing in a transient environment (Docker, Mesos, Selenium Grid), Agile Transformation, Acceptance Test Driven Development (ATDD) ATDD, and 100% Real Test Automation (Spock, Geb, and Groovy). He is involved in designing solutions/strategies for testing consultancy and services, QA cost optimization and quality Index Implementations, test strategy development, regulatory compliance roadmap formulation, and current state testing assessments. He has extensive experience in testing of Healthcare Payer Systems applications & Products, Web Portal Testing, Regression Optimization, Test Automation; Test Optimization Technique Implementations like OATS, RBT, DTT, etc., and Test Automation of Mobile and Cross Browser Testing. His initiatives around real-time automation reducing QA cost, improving productivity, and optimizing resources, has greatly helped quality improvements in United Health Group IT deliveries.*

*Rajesh has a Bachelor degree in Economics (Honors) in 1998 and Masters in Computer Applications from Kurukshetra University, Kurukshetra, India (2001). He has 14 Years of professional experience in software quality assurance and engineering for banking and healthcare domain organizations.*

# 1 Introduction

IT in the healthcare industry is complex, especially when you are working on claims adjudication systems. Currently, most of the IT delivery is done in releases (monthly, quarterly, or longer) due to the nature of complex healthcare systems. Our application team is tasked with a mission to develop a new healthcare administration platform with advanced software capabilities. On Day 1, the goal for the project team is to support daily releases. In addition, the project should follow a Continuous Integration & Continuous Delivery (CI/CD) model, move away from silo divisions of development, QA, SA & Operations and implement DevOps.

For this program, there is no separate testing phase. Testing effort is spread across the entire development cycle, not isolated to a single test phase. Our testing is done at build level on a cloud environment using automated build/deploy/test; with 100% automated testing of all test phases, and regression testing around the clock. A healthy test pyramid is developed with no testing on static environments.

Our quality assurance team transformed itself by adapting engineering practices (Figure 1) over the last 18 months. To achieve the business goals, the IT team had to completely redefine our current software engineering techniques, process, and skill sets.

Below are the key business objectives met by our software development team:

- 1) Innovate and modernize the technology
- 2) Increase speed to market through frequent releases (quarterly to bi-weekly)
- 3) Reduce technology cost by almost 50%
- 4) Improve customer experience

## Vision & Goal: Cultural Evolution

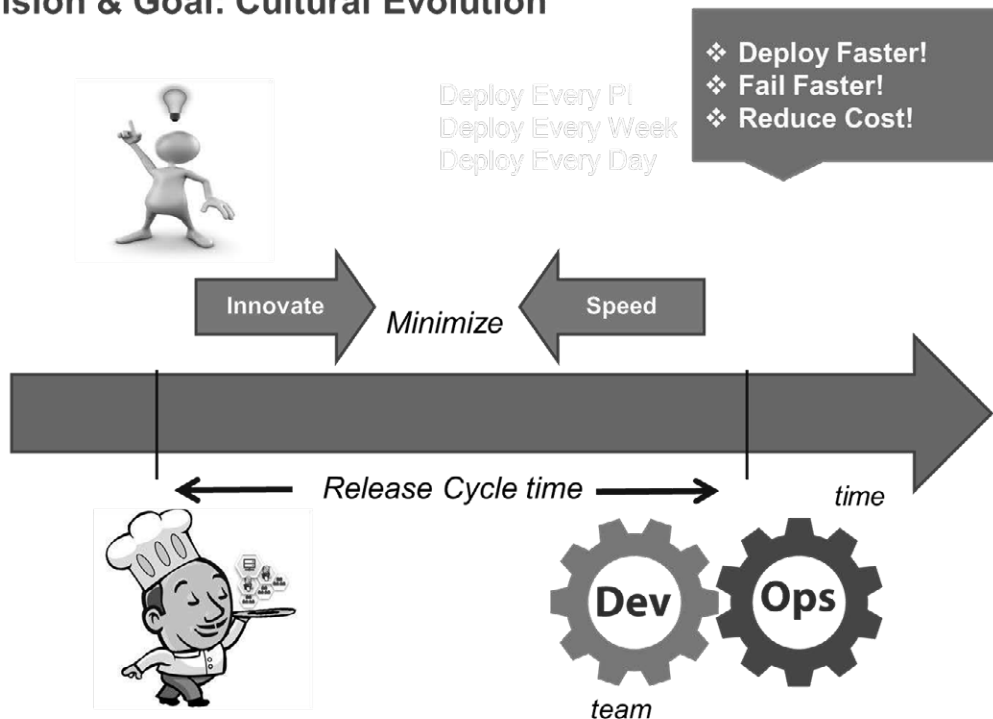


Figure: 1

## 2 Transformation Journey

We started the DevOps journey in early 2015. The Quality Assurance (QA) team members had experience in automated UI testing in the healthcare domain. Keeping in mind the goal, the first thing we changed was our hiring strategy. We started looking for *quality engineers* (rather than manual testers), who possessed programming and coding skills, and understand continuous integration process. Acceptance Testing Driven Development (ATDD) was chosen as a delivery model. The existing team started evaluating the automated testing frameworks available in the organization. The initial assessment criterion was that the automated test framework should support CI/CD and any testing artifacts (usually scripts) should be well integrated with application code. This kind of framework allows tests to be executed with each build and deployment.

Having manual testers on the team, we decided first to choose an automated test framework making script writing easy. FitNesse with Selenium was the test framework selected for this project for UI testing. It is Open Source, and integrates well with CI servers. The team developed almost 1000 tests and we were able to execute them with each build. But due to certain limitations in the framework such as an inability to write conditional statements, inability to loop, we decided to retire this framework and look for alternate code driven frameworks to enable writing more robust tests and provide CI/CD delivery support.

The development team was already using the Spock framework for unit tests which lent itself well for business logic and UI testing. A POC was done with GEB (a browser automation solution) + Selenium + Spock framework, designed to support our fast delivery requirements.

In the next six months, the team was able to create more than 40K+ unit tests, 8K+ UI tests, and 1000+ API tests. We moved to bi-weekly releases to production and were able to execute all the tests in the Dockerize environment by using Mesos in under 30 minutes. But still we were missing a thick layer of business logic integration tests. With more than 200 engineers constantly changing the application code, the corresponding UI tests were very difficult to maintain.

As a team, we decided to move away from UI tests and focus more on building the integration tests framework to test complex business scenarios which added much value to the project compared to the UI focused testing. At the same time, a non-functional testing strategy was also finalized and we started in-sprint automation and performance testing.

Our software development team continued to evolve, adapting new software techniques and maintaining agility. This has enabled our business to compete well in the market place.

The below Figure- 2 represents our transformation journey to CI/CD and DevOps.

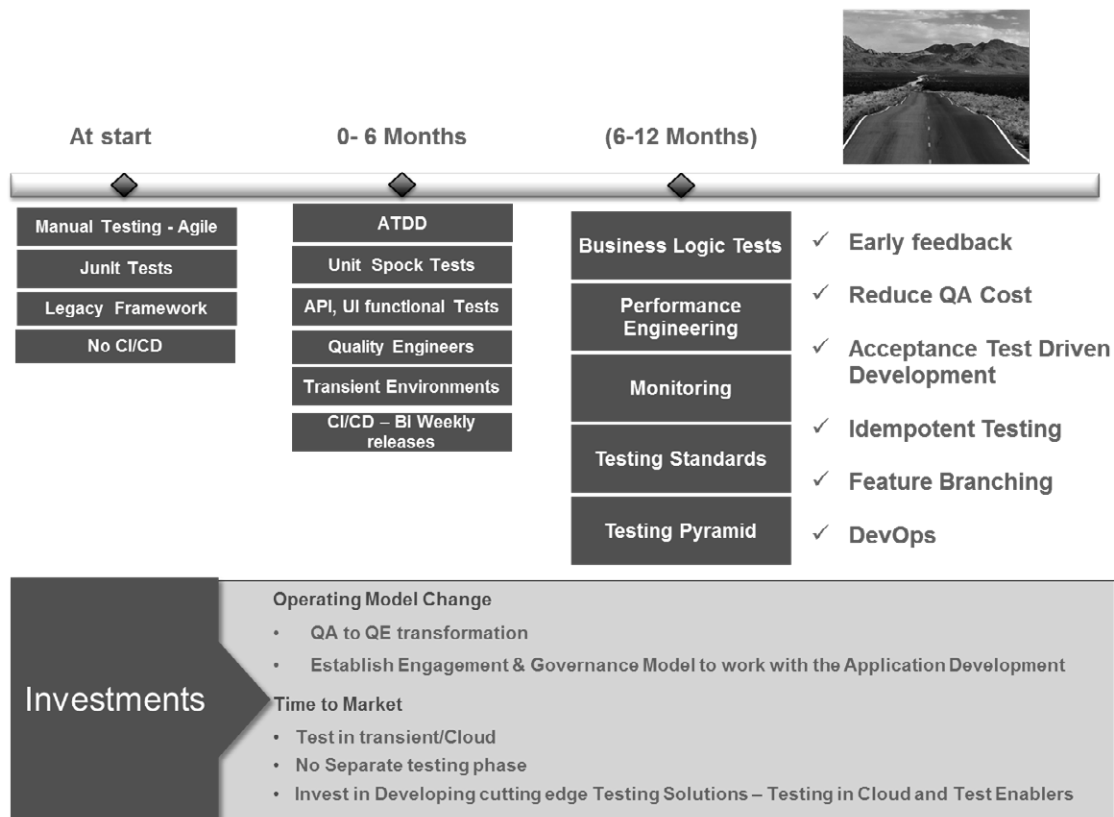


Figure: 2

### 3 Quality Control to Quality Engineering

A software development team cannot attain the DevOps delivery model without reliable test automation. Test automation done in the legacy world is completely different from the CI/CD model. In the latter model, automated tests have to be written at the same time code is developed. The testing artifacts need to not only be part of the application code, but be well integrated into it. Code coverage is measured at each build level and we used the Cobertura tool to measure it.

We established an engagement and governance model to work with the application development team. We had to change the culture and mindset of our existing team. The goal for the Quality Engineering (QE) team was to start writing the automated tests within the same sprint. Figure 3 explains the key reasons behind our team's move from a quality control mindset to adapting engineering best practices.

## QC to QE Transformation

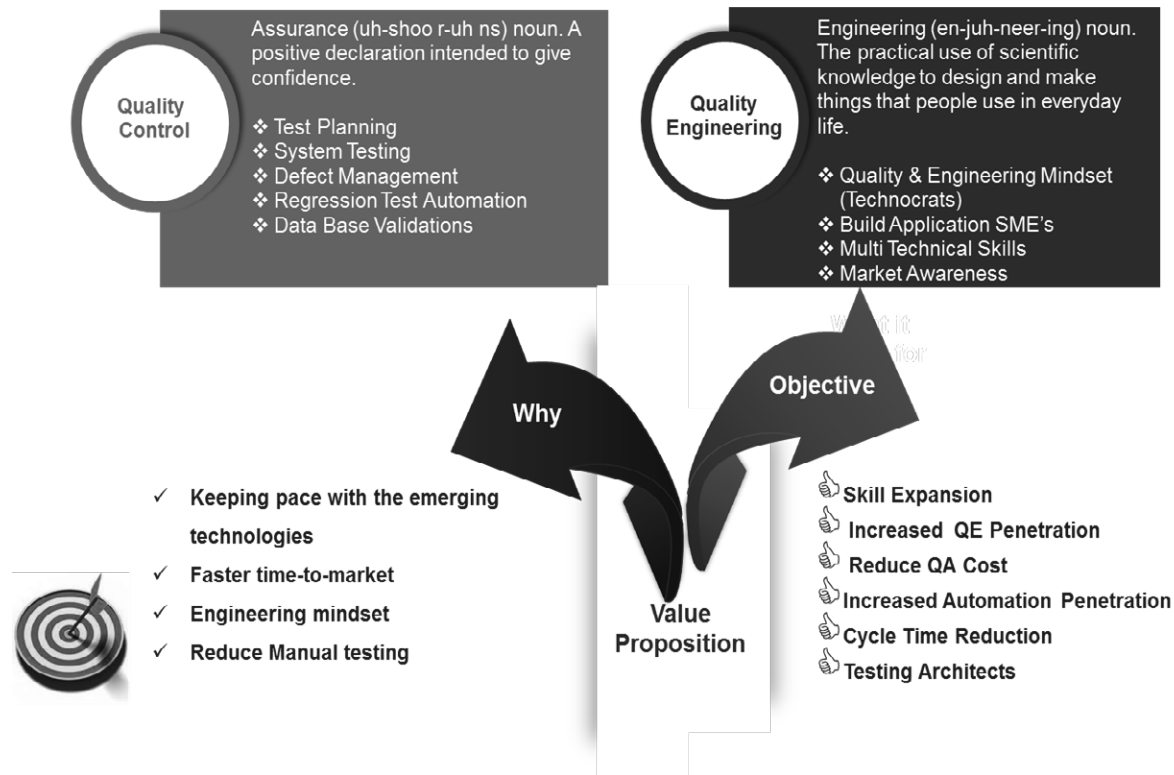


Figure: 3

## 4 Role Transformation

In a DevOps delivery model, the QA roles need to be redefined. The QA team has to partner with the development and business organizations, leveraging people, process and technology to drive speed and efficiency through a new role and with a new automation framework.

ATDD CI adoption has introduced a need to reinvent the QA role from quality assurance tester to quality engineer. This paves the way to introduce the new Quality Engineer/Software Development Engineer in Test (QE/SDET) role into the mix making QA empowered, enriched, and deeply ingrained in the software development process. The QE/SDET role is vital to deliver maximum value and customer satisfaction more efficiently and effectively.

SDETs are development testers that can drive better collaboration with the developers and business stakeholders within scrum teams. They can code, design, develop, and maintain test code and automation frameworks. The SDETs develop code and build tools to test product code and are instrumental in driving continuous integration with ATDD.

SDETs help organizations shift QA to the left in the development cycle. They also enable fully automated testing from day one, by adding development expertise to complement existing Subject Matter Experts

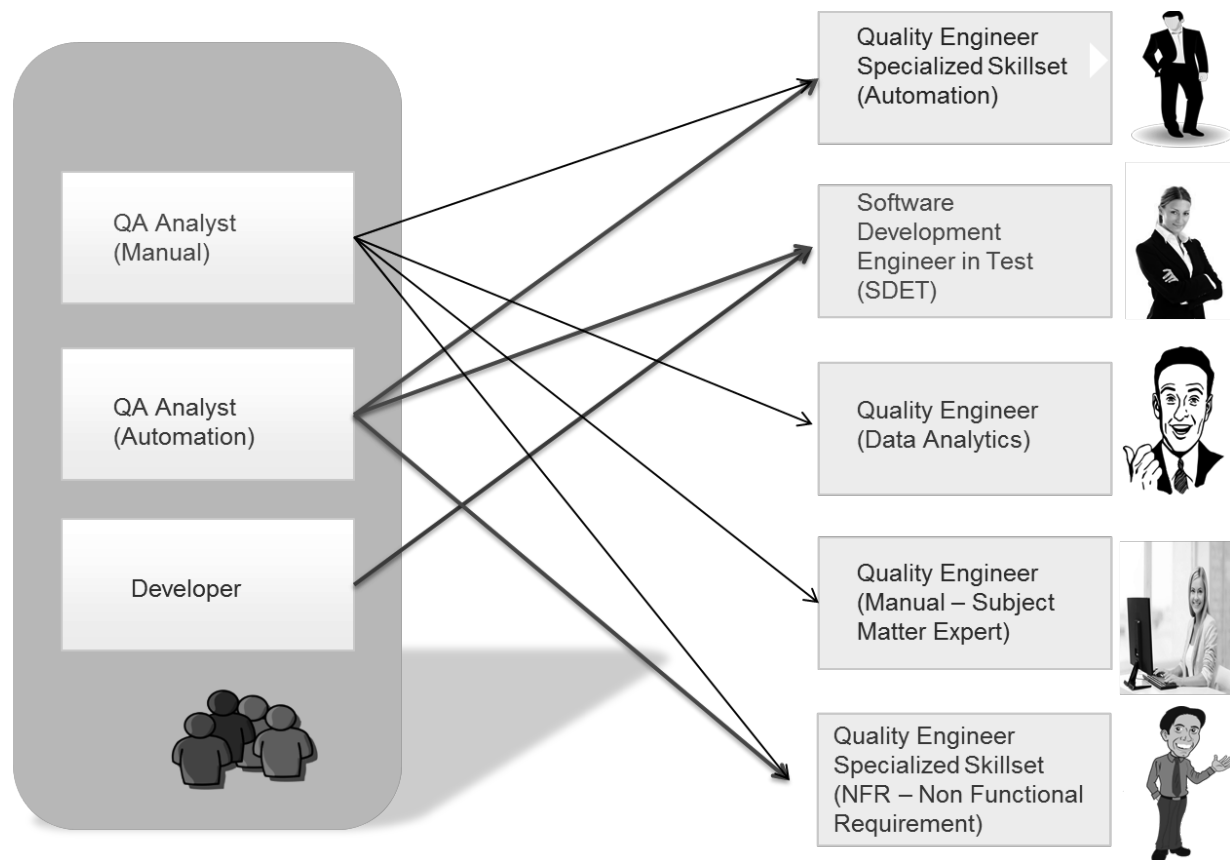
(SMEs) in QA. They accomplish this by partnering with the developers in driving test automation at the UI and API levels.

Testing roles are categorized into four areas:

- Automation Engineer - an agile, entry level role, academy (university) resourced providing automated testing under the guidance of senior engineers
- Software Developer in Test - review any automation frameworks that may already exist, setup acceptance test frameworks, and work closely with the SME and developer in support of ATDD
- Quality Engineer/Subject Matter Expert – understands business goals, application constraints, integrations, E2E scenarios; works with SDETs and Business Analyst (BAs) on achieving quality goals
- Quality Engineer in Non Functional Testing (NFR) addresses non-functional requirements and works with SME and SDET to integrate within ATDD-CI framework to validate NFRs.

We followed a multi-pronged approach to develop the community of Quality Engineers:

- ✓ Initiated a monthly Quality Engineers “Open Forum” meeting, open to all QEs in the organization to share knowledge and best practices
- ✓ We hired Quality Engineers for most open requisitions to inorganically seed this talent into high visibility programs implementing ATDD
- ✓ Identified existing resources who were already playing the role of QEs and formalized their role under an QE CoE.
- ✓ Trained over 200 QA resources, from our existing staff, on Quality Engineering skills such as programming languages, Selenium, Gatling, Groovy, CI/CD concepts, data analytics and skills required to become SDETs.



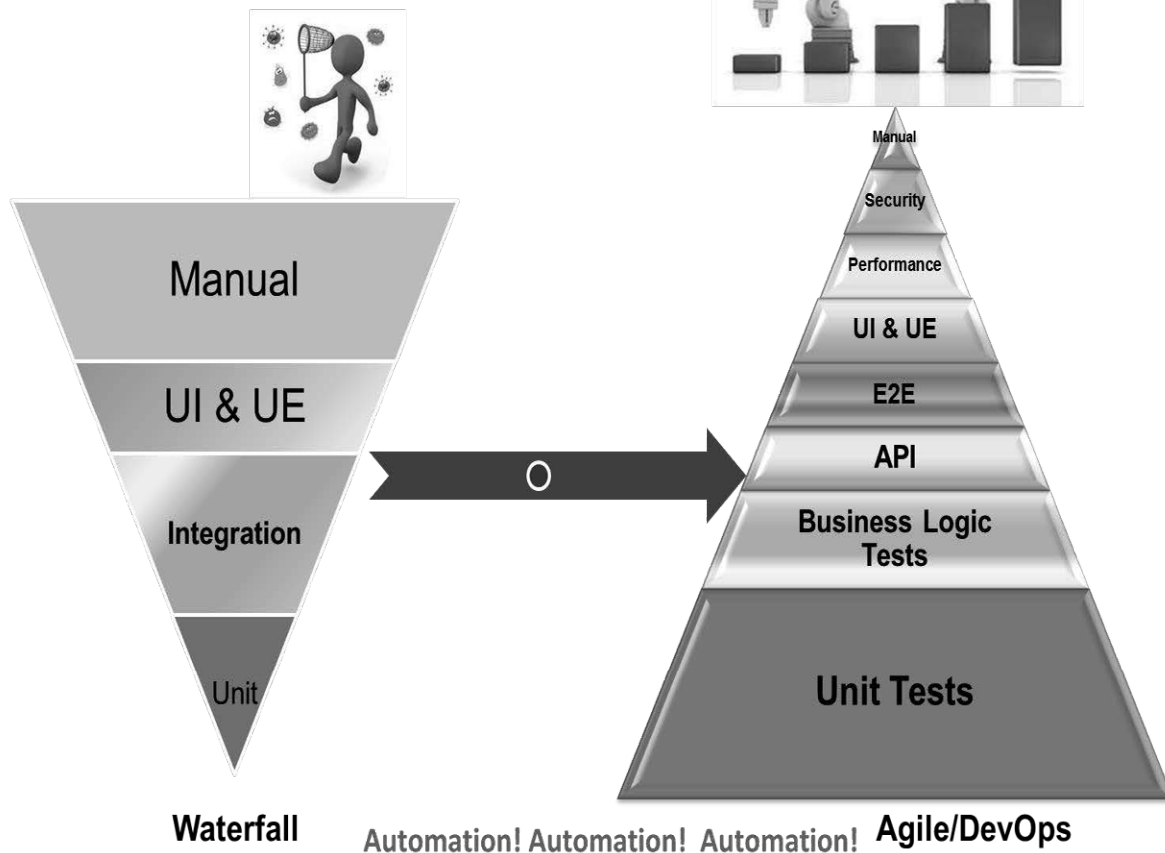
**Figure: 4**

## 5 Testing Pyramid

In order to do real continuous delivery, our applications must have fully automated black box functional / integration / system tests which cover everything necessary to verify that a particular build is ready for release to customers. All of those tests must pass in order for the build to proceed to the next stage in the continuous delivery pipeline. From that standpoint, the order of test case execution is irrelevant. However, from a practical standpoint, it helps to divide test cases into separate suites based on the likelihood of finding defects. In other words, if a recent code commit could have potentially introduced a defect, you want that failure to be detected as early in the testing cycle as possible so that the build system can alert the developers rather than waiting hours for the entire test suite to finish. This helps to compress the development cycle and prevent building up a long queue of commits.

The tests should be arranged in a pyramid with just a few tests in the first suite at the top. That way if a build is completely broken, the developers will find out within minutes. The lower levels of the pyramid contain progressively more test cases and take longer to execute. For example, it might look something like this:

## Testing Pyramid



5

Figure: 5

## 6 Testing types

Gone are the days of ginormous test cycles running thousands of large, multifaceted test cases spanning days to weeks to complete in a single run. In are the "speed of agile", targeted sampling by OS, platform, etc., parallel execution, multi-technique tests.



In the legacy world, more focus is on manual functional testing and if automation is there, it is mainly for functional regression, UI or a few select web-services tests. But that approach does not work in CI/CD delivery.

In CI/CD delivery, each test phase needs to be defined with goals and purpose of testing. As there are different layers like unit, component, integration, regression, NFR, etc., it is very important to draw the lines and define what should be covered in each layer. The overall motive should be to reduce the duplicate effort spent in subsequent layers. Suppose if a unit test covered the basic level of tests or validations, that validation should not be covered in subsequent layers. UI tests should test only the UI part; all functional testing should be moved to the component or integration level.

UI tests are very slow to execute and are error prone. The more UI tests there are, the more waiting time for developers to receive feedback. The turn-around-time is increased when you have more UI tests and these are very brutal because of slow execution, are seen as a low value addition and can delay the project delivery of the branches and builds.

The integration tests should cover all the vital business functions and complex engine scenarios; which are very fast to execute if done through services like REST (Representational State Transfer).

The NFR, performance, scalability and security types of testing need to be done in sprints with clear goals and defined purpose.

		
<b>Unit</b>	Testing small pieces of code, typically individual functions, alone and isolated. Executed for each build	Developer, Scrum team
<b>Smoke</b>	Smoke Tests make sure that build is stable No Environmental issues	Test Automation
<b>Integration Business Logic</b>	Test the correct inter-operation of multiple subsystems ( Example: integration between two classes) Business Logic tests	Scrum team (Dev and QE)
<b>API</b>	Test the API transformation logic Component Level tests Internal Functional Tests	API Scrum team
<b>UI</b>	Test only the UI navigation, validations and inconsistencies on the pages Use VOs to load test data	Scrum team
<b>E2E</b>	E2E testing from APIs to Core or Vice Versa Use real template and data	E2E and test automation team
<b>NFR</b>	Performance testing (API and UI) Availability monitoring & Logging Infra Monitoring	NFR Team

**Figure: 6**

## 7 Build Life Cycle

In a typical waterfall project, the “build cycle” means: a set of developers ready with code changes, merge these with the master repository, write and execute few unit tests, and deploy the code to the static environments. Feedback is provided to developers in the system or regression test phase which could be days after the code is committed.

On the other hand, in CI/CD delivery, the “build cycle” means: develop functionality and at the same time, write and run all types of automated tests (unit, UI, component, integration, regression, NFR etc.), providing feedback to the developers in minutes rather than days.

If any of the tests fail in building the software, these need to be fixed immediately, before deploying the code to the master branch. All tests must pass. This is a mandatory step in the build pipeline cycle.

The below figure shows the build life cycle in DevOps delivery:

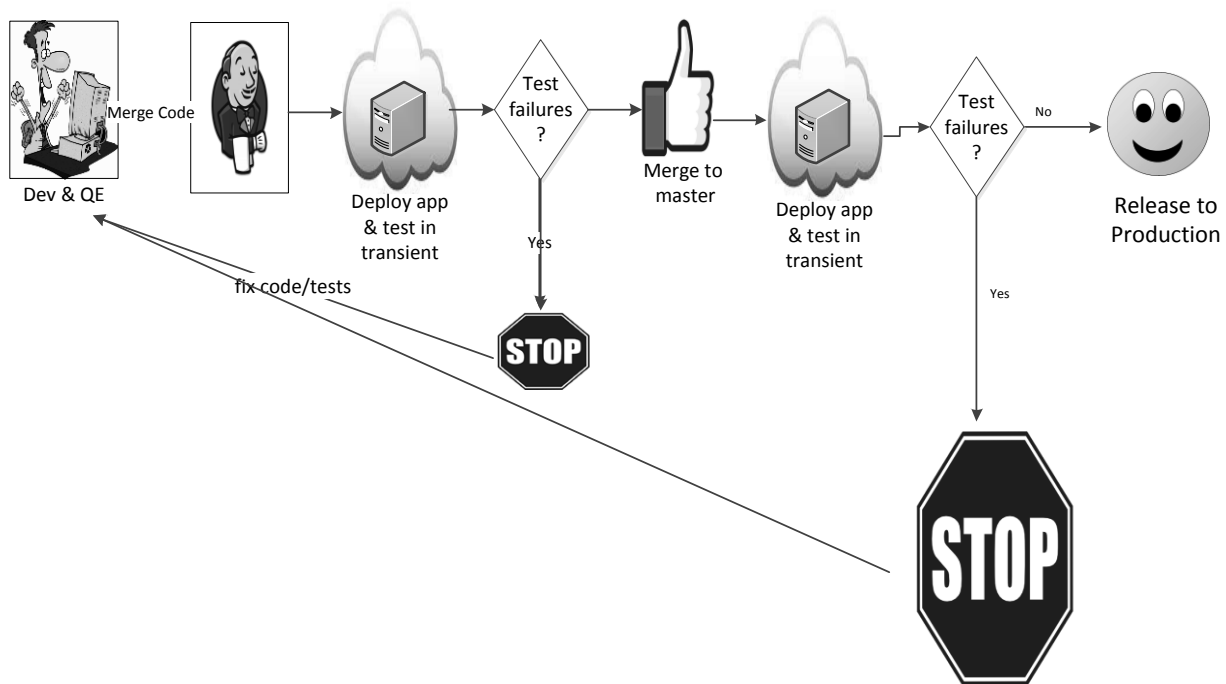


Figure: 7

# Build Life Cycle

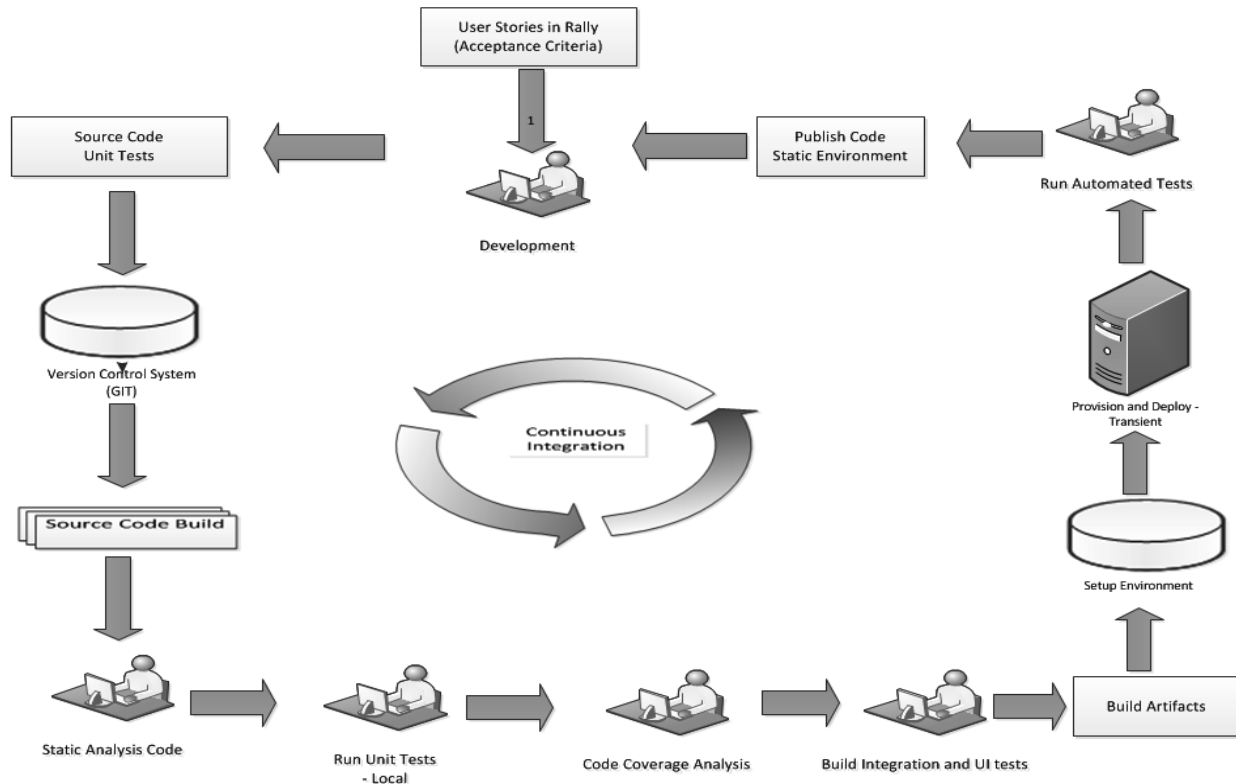


Figure: 8

## 8 Test Development

In a typical CI/CD environment and where ATDD is implemented, the Product Owner writes the requirements in a tool such as Rally. The Quality Engineers work together with the product owners and define the acceptance criteria for a user story or set of features. As soon as the acceptance criteria and timelines are defined for implementation, developers take the user story and start building the functionality at the same time that Quality Engineers start writing the automated tests. Developers are more focused on functionality development and writing unit tests to cover over 80% of the code. While Quality Engineers are more focused on writing integration and UI tests. As both are done with their development, they test the functionality in the same code branch by executing all unit, integration and UI tests. If all the tests pass for the user story, the code is checked into the main branch for build deployment.

From there, continuous servers like Jenkins/Anthipro take the latest code and merge it, building the software and deploying it to transient environments or any other static environment defined by the project build pipeline, according to the implemented technical stack.

The intent is to develop all test types combining unit, integration and UI testing in advance to be included in the build process.

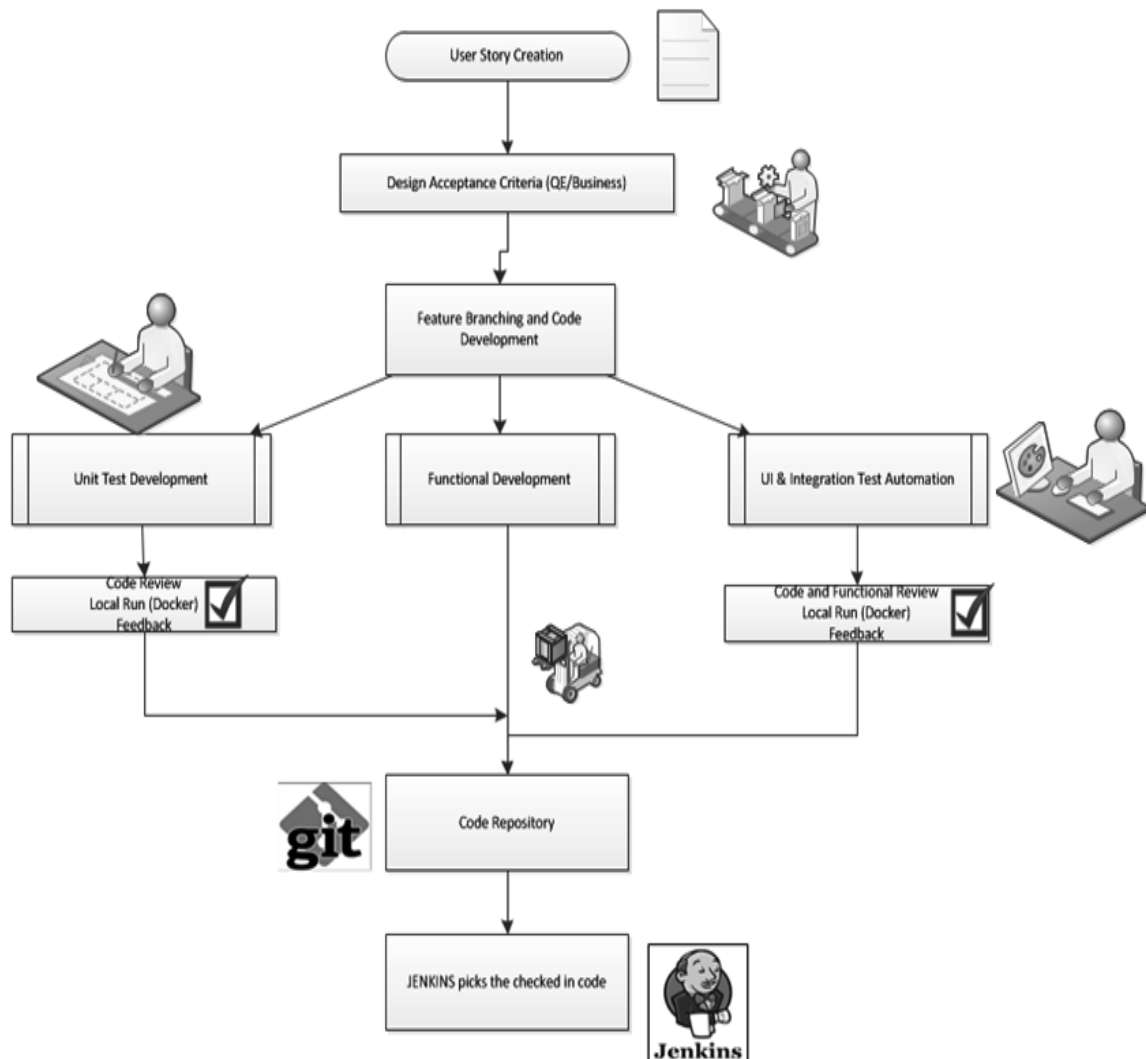


Figure: 9

## 9 Test Environments

Inception of the Cloud has given flexibility to application development teams to build and test application software in transient environments. The software is available in the market to spin up environments in a few minutes, allowing developers to write and test software without it being deployed to static environments.

We have used Open Shift Environment (OSE), Docker and Mesos in our technology stack to build the environment on the fly and test the software. The code is not moved to the static environment until it passes all the toll gates of different types of testing. All tests must be passed before code moves to the static environments. Only smoke and E2E tests should be run on the static environments in a large complex program or project.

A transient environment gives the opportunity to design the idempotent and repeatable tests and make sure every run gives the same results.

The below figure shows the test environment grid in the CI/CD type of software delivery model.

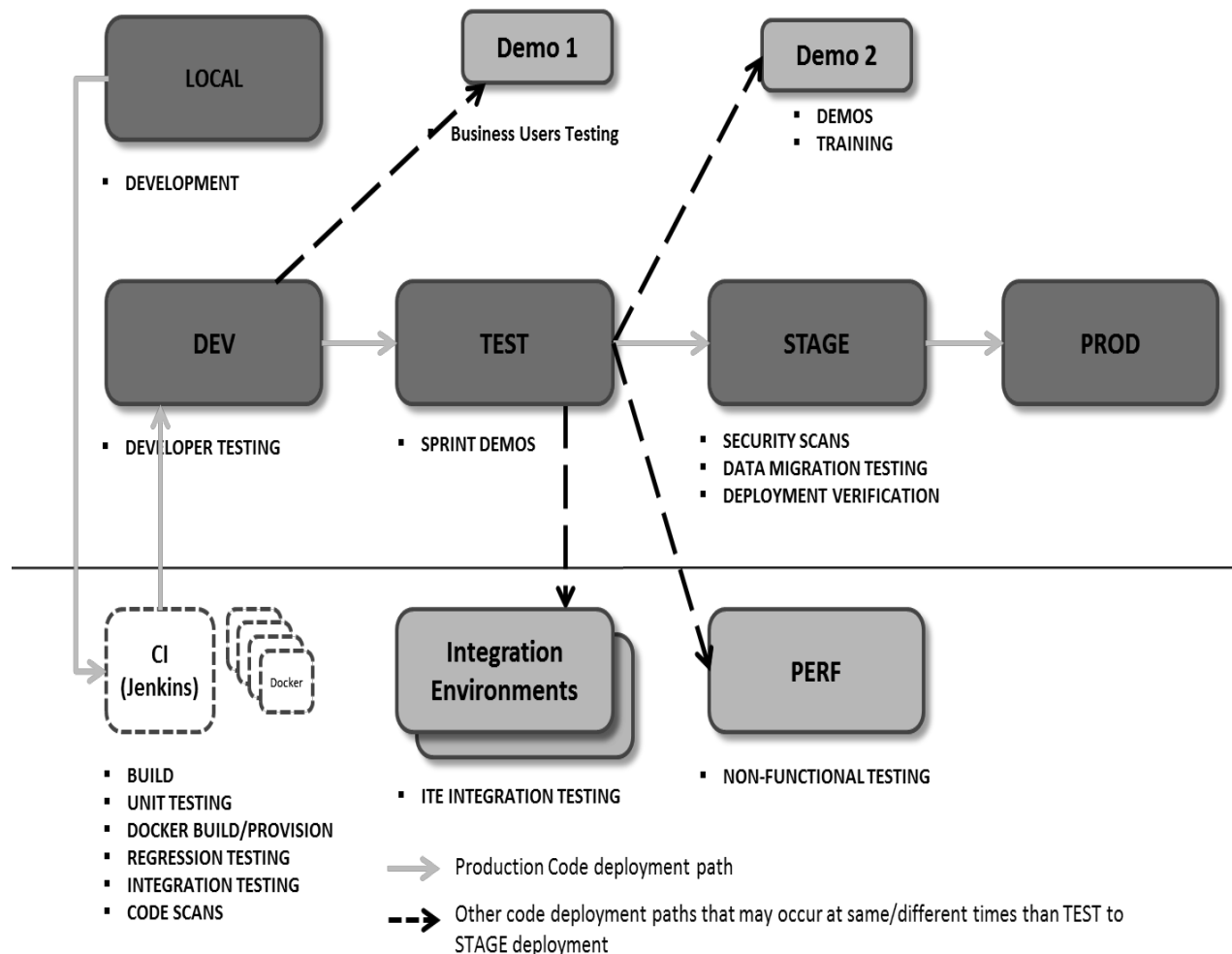


Figure: 10

## 10 Test Execution

In the legacy development life cycle, test execution is done on the static environment. In a DevOps or Continuous Delivery model, the automated tests (unit, smoke, regression, API, business logic tests & E2E, etc.) are hooked up with an application build deployment pipeline and executed in the background by the CI servers. The developer or Quality Engineer writes tests first but they are not expected to execute the tests manually.

For test execution purposes, Docker and Mesos can be used to create the environment and run all the tests as needed. We are executing almost 50K+ tests in under 30 minutes with 32 nodes in the Mesos environment. The entire build cycle finishes in 25 to 30 minutes. A normal day would see us complete on an average 30 builds

The test results can be obtained from the CI servers with a screenshot facility.

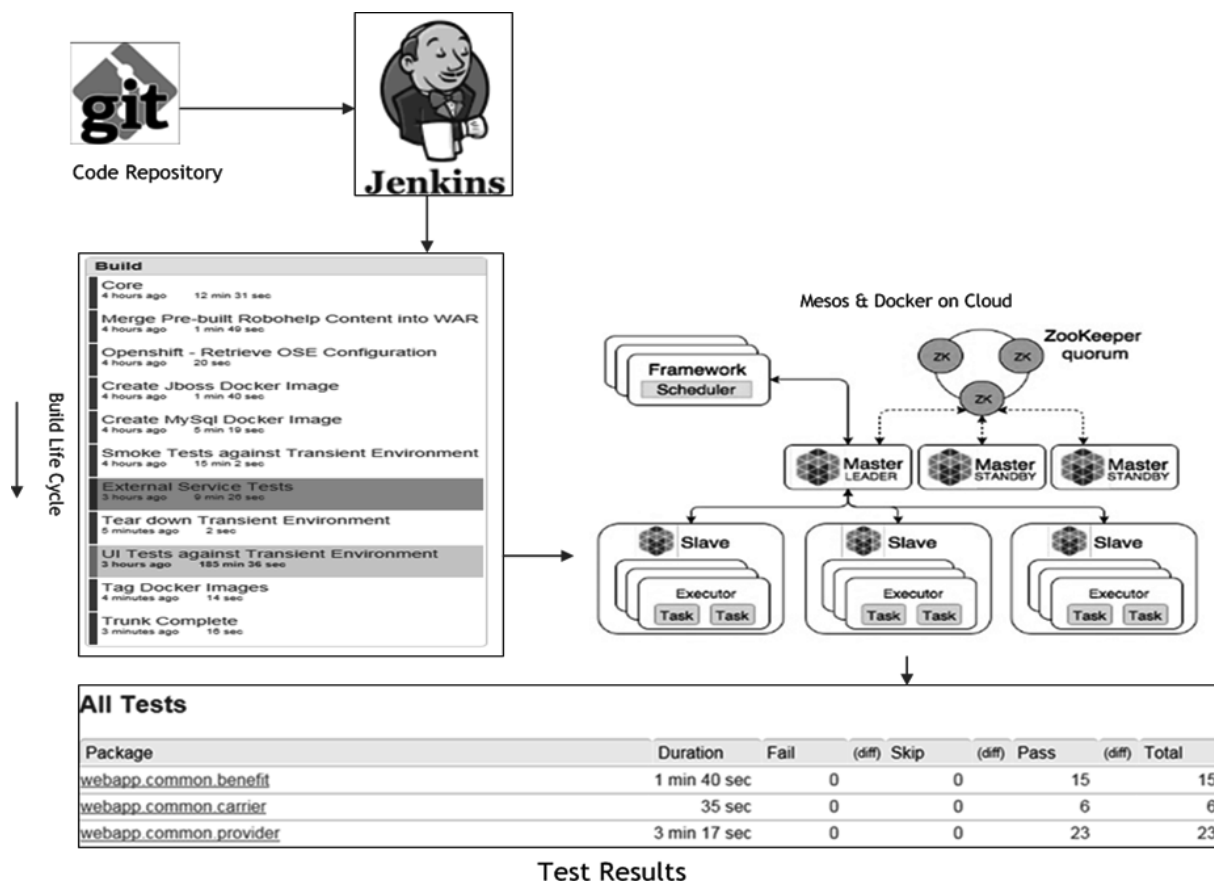


Figure: 11

# 11 Test Data Management

Parallel test execution is required in DevOps delivery for a faster feedback loop to the developers. But managing test data in parallel to test execution is difficult. The team has to be very careful in sharing the data among various tests being executed simultaneously. One test can create the data but another can update or delete it and later the same data is of no use to other tests.

To address this issue, we have adopted a different approach in managing test data. An automated test should be capable enough to create the data for its own need. We are deploying application code to a transient environment seeded with static and reference data that is never going to change. Also, the automated tests have been designed in such a way, that they create data for the functional scenario itself, use the data, and later neatly dispose of it. All tests are isolated from each other, from the test *data* point of view. No tests use test data created by other tests.

Hence, this approach helps in reducing test data and environment related defects. The following figure depicts a successful test data management technique which supports parallel and idempotent testing.

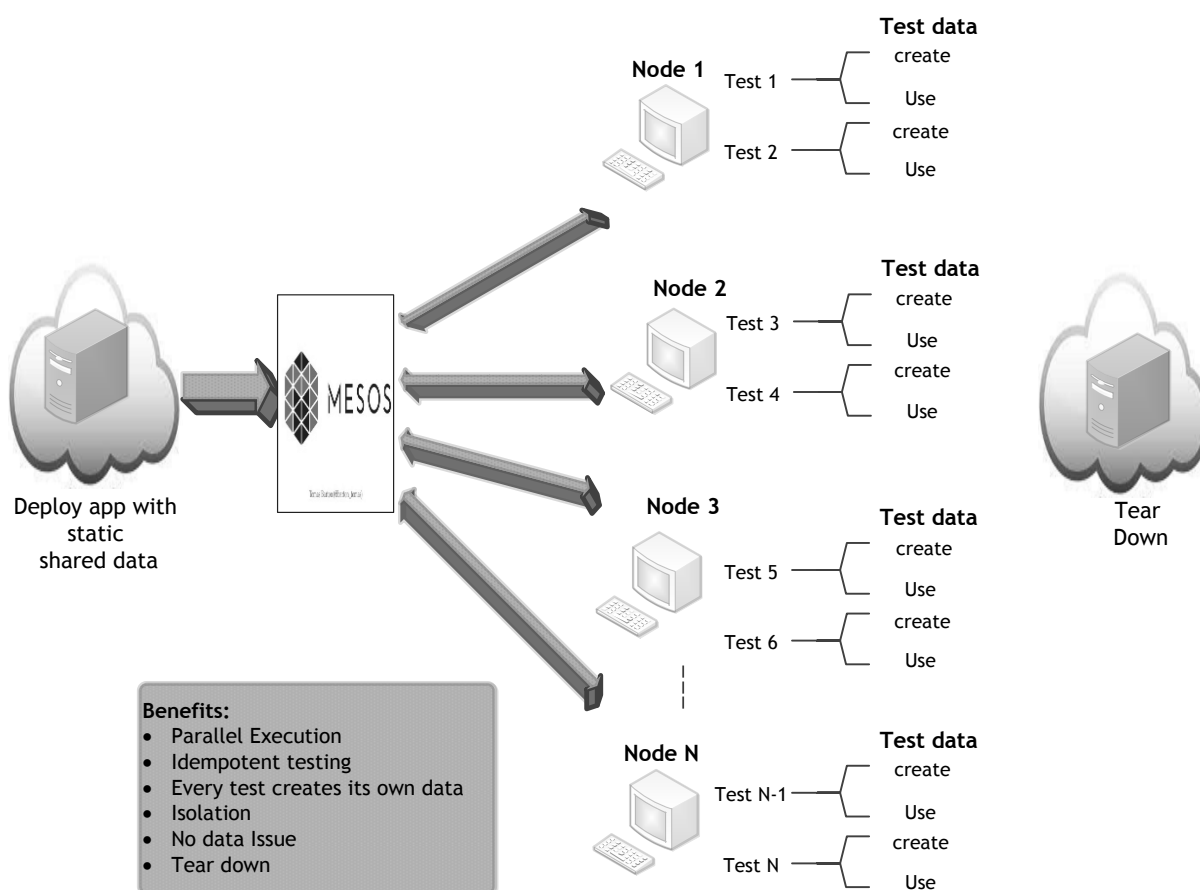


Figure: 12

## 12 Benefits: Transient Environment testing

There are several benefits of using a transient environment in the software project development cycle. The project team can achieve real time automation and up to 100% penetration of test automation.

With cloud infrastructure costs being very cheap (compared to physical servers) and application teams able to create the environment on the fly, this gives the flexibility to use the latest technology stack providing the fastest test execution! QA costs have been observed to be much lower in comparison to legacy testing where separate phases are needed to test the software.

With this implementation, we are able to maintain project QA costs less than 10% and automation penetration at 95%.

The below figure shows the QA cost difference and benefits of automated testing in the DevOps delivery cycle.

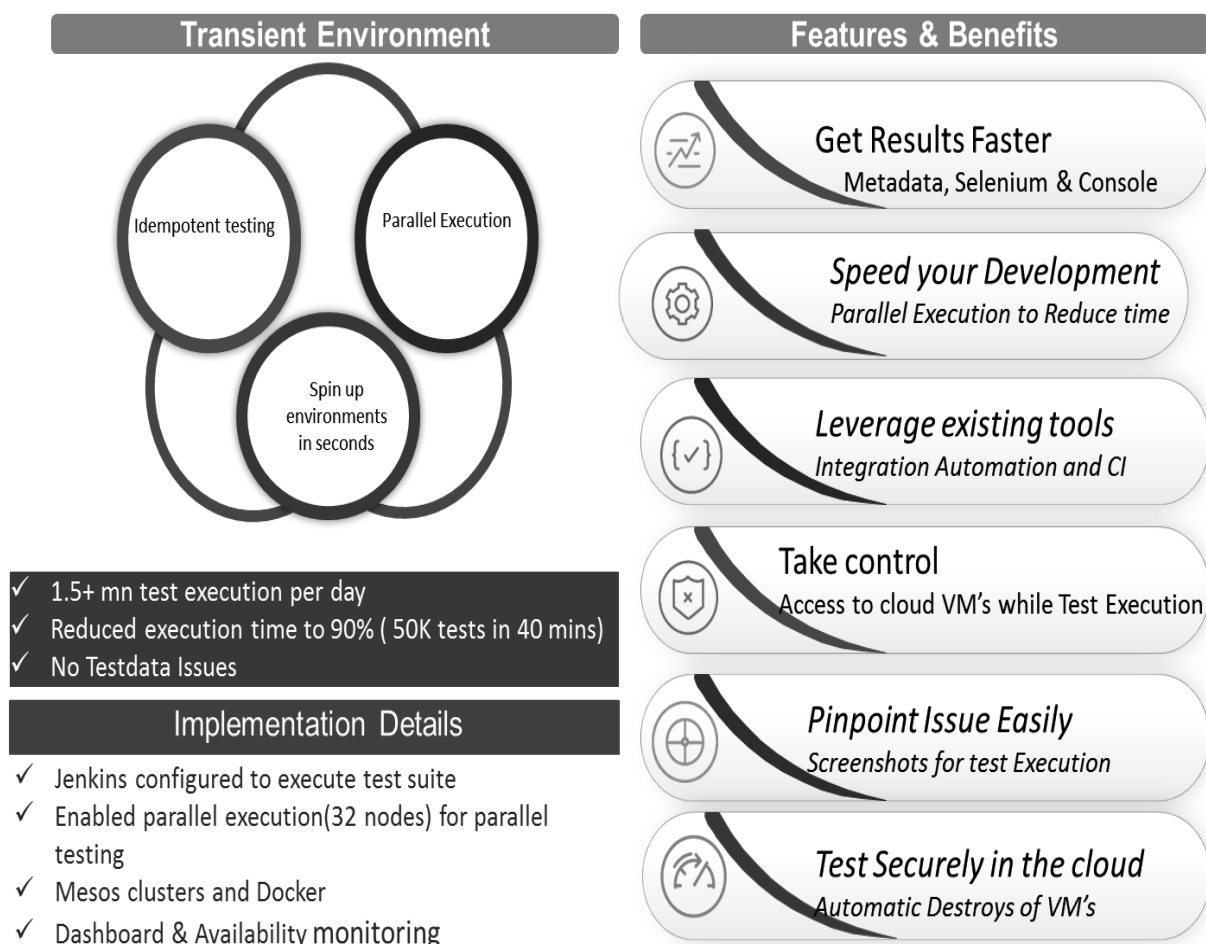


Figure: 13

## References

This is all based on practical experience and under the guidance of the program leadership team.

- 1) <https://www.docker.com/>
- 2) <http://mesos.apache.org/>
- 3) <http://www.gebish.org/>
- 4) <https://jenkins.io/>



# DITA and Evolution of Documentation to Empower Software

Madhuri Karanth and Sandeep Karanth

Madhuri.Karanth@Intel.com  
Sandeep.Karanth@gmail.com

## Abstract

In this fast paced technology savvy world, we have less time to understand and effectively use complex systems, products, and processes. For a good learning experience, the product development teams must provide high-quality product documentation in a timely and cost efficient manner. This makes product documentation an important, customer-facing artifact.

Authoring has come a long way since the traditional typing on Microsoft Word days. It is no longer about dumping heaps of information at one place and spending long hours formatting the same.

Structured documenting brings in more value. It is easy to understand, locate, and assimilate information. This paper explains how we have used DITA (Darwin Information Typing Architecture) to author, publish, and deliver business and technical content to achieve:

- Simplified content creation
- Easy to understand documentation
- Separate content from formatting
- Significantly reduce manual errors
- Automate reuse of content
- Save cost

## Biography

*Madhuri Karanth is a technical writer at Intel Security with eight years of experience. She works with the product engineering teams and makes sure that the highly complex enterprise technology is effectively translated into a simple instruction set for the users. She is highly passionate about content quality and advocates for an innovative, concise, simple, and a customer friendly instruction-based approach to drive the point home.*

*Sandeep Karanth is a technical architect who specializes in building and operationalizing software systems. He has more than 14 years of experience in the software industry, working on a gamut of products ranging from enterprise data applications to newer-generation mobile applications. He has primarily worked at Microsoft Corporation in Redmond, Microsoft Research in India and is currently chief Architect and Cofounder at DataPhi Labs.*

# 1 Introduction

Well written and reliable documentation is the backbone of software development. A well written document publicizes your code, highlights your software features, assists the users to learn the product quickly, and helps cut costs.

The production of end-user documentation is constantly evolving as new technologies become available. Users can access information in many different ways, such as pdfs, live documents, websites, support forums, KnowledgeBase systems, FAQs, and many more. DITA can help writers to meet the challenges of producing technical documents in an increasingly complex global market.

## 2 About DITA

The Darwin Information Typing Architecture (DITA) is an XML based authoring and publishing standard.

### 2.1 Information typing

In DITA, information is classified into different topics. A topic is a unit of information.

DITA topics have a title and some information that could be understood in isolation, or they could be used at different places.

Topic Types in DITA:

- Concept – Concepts give descriptive information to understand the background and context of the software feature.
- Task – Tasks are used to describe how to perform a procedure.
- Reference – Reference topics contain additional information to supplement other topics.

DITA defines how each topic type is structured. Every topic type has some common elements like Title, Prolog (for metadata like audience, category, and keywords), Short Description, and some that are also unique.

#### 2.1.1 Task

Tasks, for instance, have a series of <step>s contained inside a <taskbody> element, which also contains tags to define prerequisites, context and results.

Task topics answer questions like:

- How do I do this?
- What do I need before I start?
- What happens when I finish?

Task topics should contain:

- Steps to complete a specific operation for example:
  - Replace the printer cartridge
  - Set up the database

- Complete the application form
- Prerequisites
- Brief explanation of the context in which the steps are performed
- Examples (optional)
- Results (optional)
- Next steps (optional)

### **2.1.2 Reference**

Reference topics should contain facts.

Examples:

- A set of options or command parameters
- Meanings of fields that are displayed on the screen
- Drawings showing the controls on a lawn mower
- Schematic diagrams

They answer the questions;

- What does this thing (input field, hardware) do?
- What are the valid choices, and which one do I want?

### **2.1.3 Concepts**

DITA concept topics answer "What is..." questions. Use the concept topic to introduce the background or overview information for tasks or reference topics.

Concept topics answer questions like:

- Why am I doing this?
- What principles do I need to keep in mind?
- What's the big picture?
- What other things might be affected by the choices I make?

Concept topics:

- Set the context for a task or tasks
- Develop knowledge that the user needs to perform the tasks, for example: "What is storage management?"
- Describe how tasks fit together

### **2.1.4 Maps**

Maps are containers for topics. The topics can be arranged in a sequential and structural way to form publications or output formats.

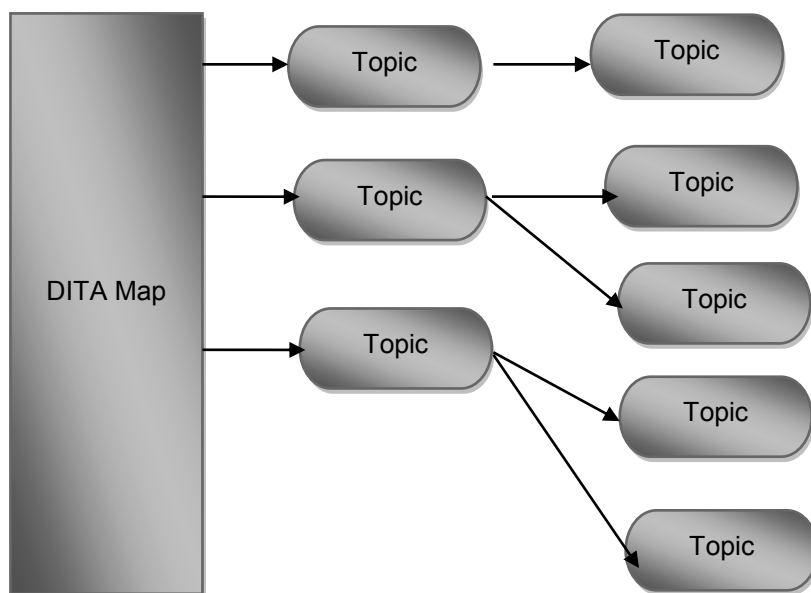
Same topics can be arranged in multiple ways to form different output types.

### 2.1.5 Assembling DITA topics into maps

There are many ways to create topics and maps. You can start by first creating topics and then assembling your finished topics into one or more documents by creating one or more maps. Or you can start by creating a map and then add new topics to it as you work.

The topics-first approach is useful if you intend to do much of content reuse, as it encourages you to think of each topic as an independent unit that can be combined with other topics in various ways.

A DITA map organizes content hierarchically, so you can add a topic as a child of the map root or as a child or sibling of any item already in the map. So, the first step to adding a topic to a map is always to choose the place it is inserted into the map.



### 2.1.6 Specialized topic types in DITA

You can create your own topic types tailored to suit your requirements. DITA allows you to make your own topic types for different output formats.

Apart from the standard topic types like task, concept, and reference, you could have customized topic types like:

- UI help
- Overview topic
- API information
- Messages

These could be standardized to achieve your ideal representation of your content.

## 2.2 Attributes and element types

Like XML, DITA has many attributes and element types which can be used to represent data in a topic or a map.

DITA also provides attributes like Conrefs and variables.

Following the DITA specification and best practices, all standard text that can be reused in many places, such as Product titles, company names, or disclaimers can be handled by making conrefs.

Variables can also be used to differentiate the text and content for different vendors without having to write the whole text again.

Using Conref and variables ensure that all translatable text can be handled with one XML translation process. The Conrefs and variables are efficient enough to allow for the simple substitution of product names and other variables; allowing for the runtime resolution of parameters.

Using different attributes and elements in individual topics and maps makes DITA flexible. They can be used to make different output formats for different vendors.

## 2.3 Output formats

DITA supports different output formats like

- Live content
- PDF
- CHM
- Webhelp
- Release notes

There are many tools available on the market which help you configure output formats for any defined types, according to your product requirement.

## 2.4 Localization and translation

DITA can be optimized for translation of documents into required languages. By using DITA, you can reduce the translation costs and also market your content globally.

DITA supports consistent terminology and phrasing; which aid a lot in the translation process.

You can differentiate between new and old topics, new and old content in topics. Topics are also reused in multiple contexts, reducing duplication.

## 2.5 Minimalism and DITA

Before minimalism, paper-based manuals were created where writers were encouraged to write anything and everything which made for increasingly thick volumes.

DITA embraces the concepts of minimalism. Minimalism is an approach that presents the reader with the smallest amount of information needed to achieve the reader's goals. The needs of the reader (or the learner), and not the system being documented, guide the information architecture and the writing style.

Instead of being in book form, topics are self-contained and standalone bits of information.

They are saved in the form of flat files or as records in a database like a content management system.

This way, we can have multiple authors for one product.

This is also suited for modular or small iteration development products, like how products are developed using agile or scrum.

## **2.6 Content Management systems with DITA**

A structured content management system is essential to manage and maintain large volumes of product documentation.

Content management systems allow users to create and manage document components, or just put individual ideas and not necessarily the whole documents. This creates greater flexibility and more effective production, reuse, translation, and adaptation of documents.

There are many solutions available, so adopting and using an ideal content management system enhances the flexibility and scalability of documentation.

## **2.7 Intelligent content**

By using DITA to carefully define content topics, you can ensure that your deliverables are always contextually relevant, according to audience, product, configuration, use-case, and task type.

DITA's intelligent content capabilities enable:

- Automated localization
- Multiple deliverables generated from one source
- Dynamic publishing to multiple formats: Word, PDF, HTML, and mobile
- Personalized documentation for each customer

With a DITA-based system, your content is easily available, at any time, on any device, in any language.

## **2.8 Reuse**

DITA allows you to do reuse and apply conditions at a granular level, for example individual words or even single characters.

- You can do a content reuse, in which a source topic or part of a topic is written once and used in many locations.

For example, you might reference the same concept topic as publishing the DITA output in both the processing and the troubleshooting maps.

Another example might be to use the DITA content reference (Conref or variable) mechanism to reuse content once like the name of a Company or a Component or a product; or many times like, repeating a short warning, danger, or a hazard statement about the proper use of a hardware unit.

- Information design reuse (specialization), in which you extend the definition of an existing DITA element to be used in a special way. Specialization makes use of the fact that DITA is based on the principle of inheritance.
- Processing reuse, in which you override stylesheet processing to customize your output.

## 2.9 Separate content from formatting and vendor

How much time do you spend formatting your content? While writing this paper, I spent much of time formatting my content. How much more could I benefit if I didn't have to worry about formatting, spacing, headers, footers, line spacing, font, and many such things. I would concentrate only on my content and this would make my job much easier because I don't have to deal with torturous mechanical work.



In DITA, the formatting of the content is in a separate file from the content. This allows the writer to focus on writing content rather than focusing on the formatting of the content. Since the formatting lies in separate files, the output is consistent, even when there are different authors.

## 2.10 Flexibility

DITA comes with a set of predefined elements, but it offers the ability to customize document structures for specific uses. Specialization is a process where you can define new elements and attributes that are based on existing DITA elements. This ability to extend the base elements to other types of documents makes DITA a flexible system that can adapt to the needs of different documentation projects.

The DITA DTD ships with the usual types of elements you might typically use in topics, such as paragraphs, tables, or lists. But, if you need to customize the structure for your specific needs, you can do that. For example, if you wanted to add an element that contains best practices, you can create an element called <Best Practices>. This ability to customize makes the system flexible.

# 3 Best Practices

## 3.1 Making documentation that considers its audience

Knowing the audience helps to make decisions what information to include, how to arrange that information, what supporting material is necessary and the complexity level. All these details are needed for the reader or user to understand what the writer is presenting. It will also decide the tone and structure of the document. For example, while writing a user manual, it is important to know if the users are administrators, developers, security officers, reviewers, and so on.

## 3.2 Choosing the suitable delivery format

To give a quick overview of a topic, a template like quick start guide may be suited. For a huge application which has a lot of blades and features, a delivery format like a PDF product guide and an

online help guide, which gives the user, a context-specific help would be more suitable. For a marketing audience, presentations might be more ideal.

For ever-evolving and developing products with many iterations and development, a digital format like live documentation might be more suited,

So it is important to choose a suitable delivery format for an application and the user.

### **3.3 Supporting visualization of content**

Diagrams, flowcharts, or screenshots provide a visual way for the audience to connect with the product or feature. Incorporating more diagrammatic representations of complex workflows helps users to connect with the content quickly.

### **3.4 Focus on User Goals, rather than product functions**

Many of the times, the engineering teams focus on documenting all product functions. But, the focus of the end user facing user-documentation is mainly to concentrate on what the user wants to achieve. It is not like marketing material where all aspects of the product are highlighted. Let us consider what the user needs this software to do and how he can achieve that and what he needs to take care of to get there.

## **4 Conclusion**

Single source technologies such as DITA allow technical writers to use standard processes for creating and publishing technical content. These processes also help writers to be more efficient in meeting the complex documentation needs of today and the future.

Using DITA tools and the documentation best practices, has a good impact on the ways in which the writers can plan and write documentation. While moving to DITA streamlines the process of creating and managing documents, using best practices improve the quality of documentation.

## References

### Web Sites:

DITA ORG – Optimizing the DITA authoring experience (accessed July 7, 2016).

<http://dita.xml.org/>

Tutorial points – DITA for the impatient (accessed June 20, 2016).

<http://www.xmlmind.com/tutorials/DITA>

*XML - Technical resources on XML standards and technologies*

<http://www.ibm.com/developerworks/xml/> (accessed March 12, 2016).

Wikipedia – Audience analysis (accessed March 12, 2016)

[https://en.wikipedia.org/wiki/Audience\\_analysis](https://en.wikipedia.org/wiki/Audience_analysis)



# Product Documentation in Agile Software Development

Madhuri Karanth

## Abstract

With more and more businesses and practices adapting agile methodologies, it has become the way to move forward in terms of software process management. With chunks of working software being delivered in short iterations of time, it has become important that the documentation should also be delivered in chunks, with that working software, for those incremental features. When we have intermediate milestones like technical previews, betas or any demonstrations to customers, having documentation on par with the working software increases the value of the product to the prospective customers and all stake holders. In addition to this, this approach will give a more holistic view of the product and it makes the whole product lifecycle truly agile.

It could get challenging to deliver documentation on par with the working software due to various reasons like lack of visibility into the engineering practices, changing scope in product features and in-depth review cycles of documents. However, there are ways to get past these challenges and many advantages of using agile methodologies in delivering technical product documentation.

This paper describes the ways and means to achieve growing and evolving quality documentation which can change with the evolving product. As a result we will have short iterative technical documentation deliverables along with the growing software.

## Biography

*Madhuri Karanth is a technical writer at Intel Security with eight years of experience. She works with the product engineering teams and makes sure that the highly complex enterprise technology is effectively translated into a simple instruction set for the users. She is highly passionate about content quality and advocates for an innovative, concise, simple, and a customer friendly instruction-based approach to drive the point home.*

Copyright Madhuri Karanth 07-Apr-2016

# 1 Introduction

End-user product documentation is the one of the key artifacts of a product that gets delivered to the customer with the software. It is an efficient medium of communication between the engineering teams and the customer. A healthy, high-quality document can enable the customer to be delighted with the quality of artifacts delivered to the customer.

The main aim of adopting agile is to develop software incrementally in short iterations. It facilitates for a rapid and continuous delivery which can evolve and change over feedback. Agile Manifesto says “Working software over comprehensive documentation” which means that the internal documents like design, requirements, elaborate use case documents are either minimal or do not exist. Technical writers usually rely on these documentations to give them a high-level overview of the product. So in agile, technical writers and documentation teams end up struggling because they don’t have their usual material or prototypes to rely on. So let’s take a look at different scenarios and how to efficiently handle them to create quality evolving documentation which evolves with the product.

## 2 Effect of Agile on Writing

- The iterative changes in the product can lead to changing documentation and the need to re write, modify, or delete what is already been written.
- The writers mostly have to create documentation on word of mouth of developers due to the lack of prototypes and minimal design, requirement specification, or use cases. This might not be accurate. If the writer has to wait until the product is complete it means that the writer has little or no time to document the product features.
- The writer many times does not have the visibility into the product features delivered until the very end of the development iteration.
- Varying scope and small changes in the sprint sometimes means that the feature set being developed can keep changing at every step of the sprint. This means that we keep doing and undoing the feature set in documentation progress.
- Technical writing itself has evolved with practices like DITA, and structured writing. That means, now making a user guide is not merely typing into Microsoft word. It has its own methods and structures which have to be prepared and structured before starting the whole process of documenting the features.
- The documents have to be technically and editorially reviewed. This might get cumbersome in the sprint cycle with documents coming up for reviews.
- If translation of documentation is involved, it can create quite a few complexities.

## 3 Solutions

### 3.1 Master Agile and scrum nomenclature

Sometimes not knowing the terminology used in Scrum could hinder you from getting a grasp on things.

A good starting point is to start familiarizing yourself with some common scrum terminology like epics, sprints, story points, scrum master, product backlog, sprint backlog, release planning, iteration, burn down, stand up meetings, release planning meetings, sprint retrospective.

Getting trained in scrum methods, process, and procedures helps us to better plan our work.

Many organizations have scrum trainings for their engineering teams, it helps a lot if as technical writers, you could also take the same trainings.

### **3.2 Adopting documentation methodology that best suits the agile methodology**

The best suited documentation methodologies are

- Information mapping
- Topic based/ structured writing using documentation architecture like DITA.
- Live documentation

These mostly deal with the fact where you can separate content from formatting. You can focus only on developing the content. This decision has to be taken at a higher organizational level. These methods have tools of their own and also document repositories where information is stored and used.

- Information mapping
  - Information mapping is a technique focused on creating clear and concise information based on user's needs. This has several tools to analyze, organize, and present information.
  - This method focuses on categorizing information into units of information.
- Topic based/structured writing
  - Topic is unit of information with a title and content. It represents one idea. Structured topics contain only the information needed to understand one concept, perform one task or procedure or look up one set of reference information.
  - It is a nonlinear approach to document information. It is not narrative. Here you don't have to follow a linear structure to understand something. You can look up at a small unit of information as to what you need and move on.
  - Topics can be picked up to make up different deliverable types. It also facilitates for reuse and cost effectiveness to maintain documentation.

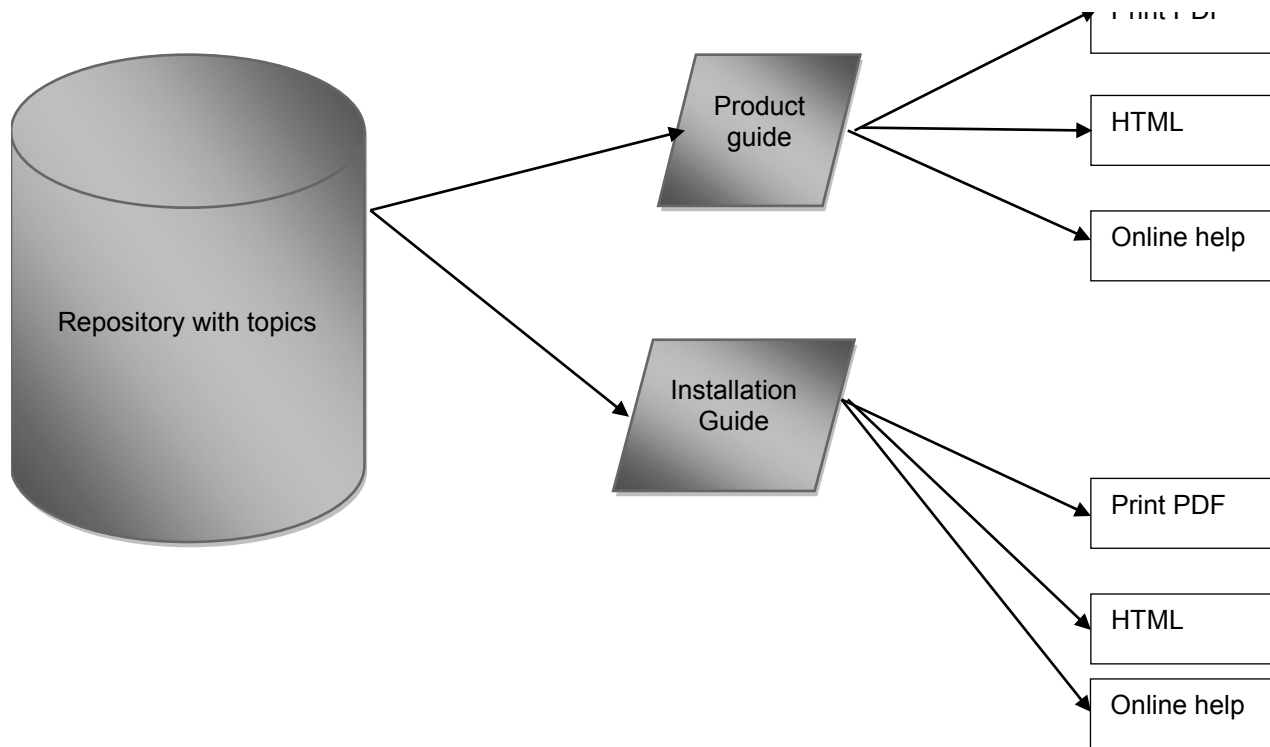


Fig: Topic based writing

- Live documentation
  - A live document is continuously edited and updated. As opposed to static pdfs, the users can provide feedback in the form of comments on the documentation page or article.
  - The customers can provide instant feedback. This also serves as a forum for continuous feedback which is a main principle in Agile and scrum. The feedback could be on the product which the writers can pass on to the engineering and product development teams.

### 3.3 The n+1 Module

This simply means that you document Sprint n features in sprint n+1. This is a very effective way of scrum implementation in technical documentation. By the end of the sprint, the features are ready and the writer has access to all information.

- You document the features of sprint 1 in sprint 2, and so on. This has the documentation developed at a lag or delay of one sprint. But in many projects, last sprint is usually a hardening sprint. In a hardening sprint, there are no new features developed, instead the focus is on fixing bugs, providing performance improvements and other code strengthening activities.
- This gives the writer ample time and opportunity to finish up the last feature sprint's documentation and wrap up the whole guide/document.

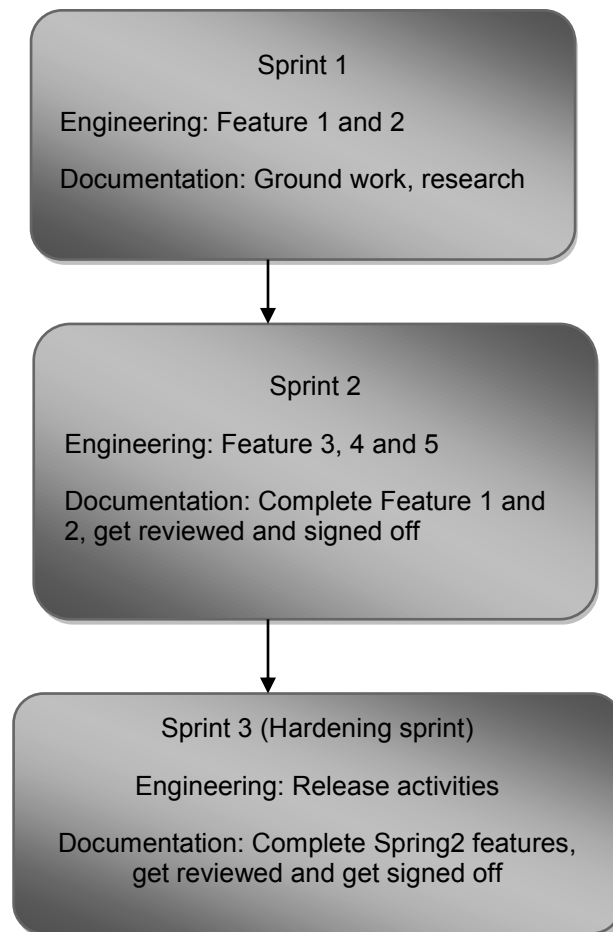


Fig: Module n+1

### 3.4 Involvement in stand-ups/meetings/design discussions

- The writer needs to participate in all the stand-ups , requirements discussions, planning meetings, and design discussions. This gives the writer a holistic view of the product and its features. This also enables the writer to keep up with the backlog, feature lists in turn deliver incremental documents.
- The writer has to have ready the documentation and review stories and their respective tasks. Make sure that you have review stories on developers and add them to the sprint planning schedule. A big advantage here is that you as a writer can define the "DONE" for these stories and mark them closed only when you think that they are done. This way you can track your documentation progress and the review progress.

### 3.5 Have an agreement with the engineering team as to how the documentation stories are to be included

Approach1: Would you want to include the work as tasks under the development team's user stories?

<b>Sprint 1</b>	
-----------------	--

<b>Documentation user story</b>	Technical publications documentation tasks
<b>Tasks under documentation user story</b>	<ul style="list-style-type: none"> <li>• Research on the product</li> <li>• Create Product guide outline</li> <li>• Review Product guide outline</li> <li>• Incorporate review comments into product guide outline</li> </ul>
<b>Sprint 2</b>	
<b>Documentation user story</b>	Technical publications documentation tasks
<b>Tasks under documentation user story</b>	<ul style="list-style-type: none"> <li>• Create content for user profiles feature</li> <li>• Review user profile feature</li> <li>• Incorporate feedback on user profile feature</li> <li>• Get the feature finalized and approved by the engineering team.</li> </ul>

Approach 2: Would you want to have a separate documentation user story and track your progress with tasks under them?

<b>Sprint 1</b>	
<b>Engineering feature</b>	As an admin, I should be able to access user profile and grant appropriate permissions to the users.
<b>Developer's story</b>	Admin can manage user profiles and grant permissions.
<b>Developer's tasks</b>	<ul style="list-style-type: none"> <li>• Code for the feature, "manages user profiles and grant permissions".</li> <li>• Write unit test cases</li> <li>• Self-review of code</li> <li>• Peer review of code</li> </ul>
<b>Writer's tasks</b>	<ul style="list-style-type: none"> <li>• Understand the feature</li> <li>• Incorporate the feature in the documentation outline</li> <li>• Create content for the feature</li> <li>• Technical review of the feature</li> <li>• Incorporate feedback for the feature content</li> <li>• Approvals for the feature content</li> </ul>

- For every sprint write user stories which are in sync with the development stories that are being planned in that sprint.

To successfully accomplish this, you must have access to an Agile planning tool or software that the engineering team is using such as VersionOne. You must be an active member of this tool with developers, QA, and other folks. This will also make sure that the documentation stories and tasks have same visibility as the development and QA tasks.

### **3.6 Volunteer yourself to be a scrum master for your sprint team.**

Well, why not? Technical writers can be good scrum masters. This is a type of role where your organizational skills and communication skills are put to good use. A scrum master resolves conflicts, and helps all members in the scrum team to achieve their goals. A scrum master also tries to remove any impediments that anyone has.

This coordination will keep you keep up the pace with the progress of the features for that iteration. It will give you good exposure into the different processes that go on in an engineering team.

The obvious advantage is that you are up to date and all caught up with the features and it very much simplifies your documenting process.

### **3.7 Have a working setup refreshed with latest builds of the product**

You can write about a product much better if you have used it yourself, played with it and if you have hands on experience.

Working with the latest builds of the product, keeps the writer up to date. It can also help the writer to be on par with changes and keep track of things.

### **3.8 Document what you learn as you go**

It is easier to adapt to agile when you focus on its benefits rather than on its challenges. So it's time to spend more time out of our cubicles, talking and interacting with development team members.

Pen down what you learn daily and keep refining the information.

### **3.9 Staying close to the engineering team helps**

Communication and co-ordination can go a long way in scrum. Find a place which is close to your engineering team and the relevant developer. From my experience, I see that this helps a lot. They see you around and you are very much a part of the team. They even toss ideas and the new changes when they pass by. You can get information faster. If they see you around more, they will keep you in mind when there are any unplanned meetings and they would reach out and inform you.

Make sure that you are part of all main distribution lists for the project so that you can receive updates first hand.

### **3.10 Sprint demos**

In Agile typically, the features implemented in every sprint are demonstrated to key stake holders at the end of every sprint which are called sprint demos. This is to adhere to the principle of agile "Get continuous feedback".

Sprint demos are done by engineering or the software development teams. Being technical writers, we would be equipped to give these demos. We know the functionality well, we can articulate this to a wider audience. Being a part of sprint demos or rather hosting a sprint demo will definitely show case the product knowledge and thus there is no question that it would also be documented well.

### **3.11 Planning**

Planning the types of documents, estimation, user needs, and review timelines for every iteration and for the whole product lifecycle can ease out whole process.

Have templates for documentation plan and schedules.

### 3.12 Be part of sprint retrospective meetings and give feedback

Agile and scrums are built on the fundamental fact “Fail fast and recover fast”, “Get continuous feedback”.

The retrospectives give an opportunity to do exactly that.

So it is very important to be a part of sprint retrospective meetings and voice your opinion on any changes that you might need to facilitate faster and efficient delivery of documentation artifacts.

### 3.13 Translation of documents

Planning ahead plays an important role when translation of documentation is involved. The documents have to be delivered to the localization team or translation vendors and hence after a decided date, any more changes cannot be incorporated. Usually vendors are hired for translation and it would prove to be costly affair to go back on a decided date because of changes.

- Keep the core team committee, engineering team involved and informed about the drop dates.
- Plan a schedule which gives you a slight flexibility to incorporate some changes.
- Remind the team about the drop dates and make sure to put this date in the project schedule and important release dates for the product.

## 4 Advantages

- The writers work as early testers of the product, giving quality feedback to the engineering teams and functioning as bridge between customers and developers.
- The writers with their experience and early involvement can sometimes spot issues in early stages of the product development life cycle and thereby enabling quality.
- Many user interface design related issues can be spotted in early stages. Writers are also known to review terminologies on screen and give right feedback to the engineering teams.

## 5 Summary

### 5.1 Workflow

Here is a basic structure workflow to summarize the information presented in this paper.

Attend Sprint planning and release planning meetings, user Interface discussions
Look at the epics and as the engineering team breaks them in to stories, Include documentation stories
Make documentation tasks for the stories and include what comes in every sprint

Attend daily scrum standup
Make sure that you are part of the agile planning tool, put your stories, tasks, and daily effort there and track your progress.
Talk to the scrum master about any challenges or impediments
As the product evolves and the features get developed, get a working setup so that you can test and document them
Review the documentation set every sprint
Give feedback in the retrospective

## 5.2 Conclusion

In traditional waterfall models, the writers get involved mostly late in the product development process, struggle to understand what everything was about, and then hurry to document the final product with no chance whatsoever of intelligently contributing to the development activities. With Scrum and agile, the role of technical writers as part of the development group, can be remarkably more important. In addition to the actual product documentation our experience can support developers and testers in delivering better project documentation, that is, software analysis and design papers.

Just as engineering teams are adopting agile methodologies to build a robust, user-friendly and evolving software, technical writers too can use complimentary methods to be an integral part of the product development process.

Based on our experience, we have suggested some ways, techniques, best practices, and ideas to make this a far easier process. When used efficiently, you realize that working as part of agile leads to a more streamlined process and is beneficial in many ways.

Once we learned how to use agile, we found that benefits it provides by far outweighs the struggles we might have to face while adopting this process.

# References

## Websites:

Manifesto for Agile Software Development (accessed July 7, 2016).  
<http://www.agilemanifesto.org/>

Agile documentation (accessed July 12, 2016).  
[http://writersua.com/articles/Agile\\_doc/](http://writersua.com/articles/Agile_doc/)

Agile teams and documentation (accessed July 12, 2016).  
<http://techbeacon.com/why-agile-teams-should-care-about-documentation>

Agile documentation best practices (accessed June 21, 2016).  
<http://www.agilemodeling.com/essays/agileDocumentationBestPractices.htm>

## Book:

A Practical Guide to Distributed Scrum (IBM Press)

# Agile Fatigue: What is it and Why Should We Care?

Heather M. Wilcox

Heather.wilcox@NWEA.org

## Abstract

Agile is a very effective method for developing high quality applications quickly and accurately. However, as time goes on, interesting side effects develop: Marketing and Sales teams become addicted to a constant stream of new features, fixes, and upgrades. Customers ask for more and better software and begin to expect near-instant gratification. In response to the demand, Engineering teams crank out release after release which slowly and quietly takes a toll that often goes unrecognized and can have very destructive results.

Fatigue is only just now being recognized as a problem within Agile organizations and the greater software community. It is something that may be experienced by many Agile-based teams, but very few recognize it for what it is. Even fewer groups are able to respond to it in a useful way.

This paper uses anecdotal and published evidence to explore the signs and symptoms of Agile Fatigue and suggests multiple strategies (both proven and unproven) for fighting it.

## Biography

*Heather has spent over 20 years working and learning in the software industry, choosing to focus primarily on start-up and small companies. As a result, Heather has had a broad range of job descriptions which include, but are not limited to: Tech Support Engineer, IS Manager, Technical Writer, QA Engineer, QA Manager, and Configuration Management Engineer. This has given her a wide range of experiences to draw from in her current role as a Senior Quality Assurance engineer.*

Copyright Heather M. Wilcox, 2016

# 1 Introduction

*"I think People think Waterfall is great because you have a lot more time to not do stuff."*

*-Heather M. Wilcox  
PNSQC 2014*

Of all the statements I made during my talk at the 2014 PNSQC, this one tidbit apparently resonated with people in the room. It was tweeted and re-tweeted, which was a first for me. Those words were spoken as a response to a question about why I thought some people liked Waterfall development better than Scrum. At the point that I made this statement, I felt in my gut that it was *\*right\**. However, since that time, I've come to realize that there are many more layers to it than the most obvious interpretation.

I didn't understand it then, but I now understand that I was indirectly talking about Agile Fatigue. This is a condition that is the result of months upon months of steadily churning out code that has customer value. This is a great thing for a company – the steady output of new software and updates that partners can use and benefit from. However, if that is all a team ever does, there can be severe long term consequences including burnout, frustration, boredom and team attrition.

## 2 Back to Basics: SDLC Differences – Agile vs. Waterfall

### 2.1 Waterfall Development Methodology

Waterfall development is the foundation upon which our industry is built. Business carefully defines the structure and features of a product. Then development specifies how it's going to build the thing and how long it will take to create each aspect. QA proposes a plan and estimates for testing. There is a negotiation over what features can and will be included within the development window and, once everyone is satisfied, the work begins. The product marches through the process in a strictly regimented manner and, if everything works to plan, is ready to ship on the projected release date.

#### 2.1.1 Pros of Waterfall

Waterfall has some big advantages in the right situations. The output from a Waterfall project should be very predictable. As long as things go well, everyone knows almost exactly what they are going to get and exactly when they are going to get it. Each team in the process knows exactly what they need to do, has time to plan their work, time to execute, and time to reorganize, clean up, and prepare to begin the next project. Any situation where customers don't want to upgrade often or don't expect a constant stream of new features (e.g. telecom or government agencies ) is a good one for the consistency and stability of Waterfall development.

#### 2.1.2 Cons of Waterfall

The down side of Waterfall development is its monolithic nature. Changing the feature set part way through the development process can cause catastrophic levels of thrash. Once set on the path, it's difficult to modify anything that's been committed to. In a more dynamic marketplace, such as web based applications, Waterfall is too clunky and slow. Mid-process course correction, which is often necessary in web development, is painful and difficult. Additionally, it's easy for large projects to turn into a "Death March" - never-ending projects which suck money and other company resources until they are ultimately either killed or released, often in a state of partial completion.

## 2.2 Agile Development Methodology

If Waterfall is the foundation of the Software Industry, Agile is the frosting. Light and a bit whimsical, Agile rolls with the punches and can turn on a dime to meet last minute requests and unexpected but vital requirements. One of the most commonly used Agile techniques, Scrum, is built around short sprint intervals (usually around 2 weeks) that contain small stories which can be developed, tested and built within the course of the sprint. A Product Owner (PO) represents the business and works to ensure that the work brought into each sprint is properly prioritized and sized to fit. At the end of each sprint, the team releases and demos whatever software is complete. The PO can approve or deny each new feature depending on whether it meets the acceptance criteria that were included in the original story.

### 2.2.1 Pros of Scrum and other Agile Processes

As an Agile process, Scrum is VERY flexible and responsive. New software is (ideally) available at the end of every sprint. Since sprints tend to vary between 1 and 3 weeks, that means that a reasonably sized feature can, in most cases, be requested and delivered within that 1 – 3 week time period. A sudden market shift can be quickly accommodated without a lot of lost work and redirection. Teams aren't stuck working on the same monolithic code for months on end. Agile processes have driven the development of concepts such as micro-services, highly modularized software, and other techniques that encourage highly transformable code.

### 2.2.2 Cons of Scrum and other Agile Processes

Although Scrum is very effective for small projects or the addition of features to an existing product, it can be difficult for large, complicated products and multiple teams. As the size and complexity of a project increases, so does the need for careful coordination. Scrum in this kind of environment has a tendency to turn into a mini-waterfall or "scrumfall" style process.

In addition to being inappropriate, or at difficult, for large-scale projects, other problems have been identified since the creation of Agile and its varied methods. For instance, it is easy to go down the wrong path very quickly when using a process that's meant to be fast and flexible. A small misunderstanding can quickly turn into an inappropriate feature that has to be removed or discarded.

Additionally, Scrum, XP and other methodologies can be easy to abuse. Documentation, formal testing, best practices and other important development and test processes are sometimes abandoned under the guise of being "fast and flexible". The result can be complete chaos for incoming developers or anyone trying to track a team's work. Without careful effort to document and communicate, it is easy for Agile teams to segregate themselves from other groups and create silos of tribal knowledge.

At the end of the day, both methodologies have advantages and disadvantages. The key is determining which one is right for your software, customers, and organization.

## 3 The Face of Agile Fatigue

Once an organization has made a commitment to Scrum or any other Agile Methodology, the usual procedure is to dive in head first. Within a few weeks of making the conversion, software starts rolling out. As time goes on, the Business team gets used to being able to ask for and receive small features, modifications, and bug fixes within a sprint or two. Larger requests take more time, but the completed feature is often still delivered months ahead of when it would have been available in the Waterfall world. In a reasonably healthy organization, it only takes a few months to establish a regular release cadence. Once that happens, the train is rolling and becomes essentially unstoppable. This is also the point where Fatigue comes into play.

There really is no downtime in Agile. Stories are small and are completed quickly. QA is writing test cases and testing while Dev is coding. All the members of the team are writing stories, coding, testing, doing builds, or other work related to the software at all times. With Scrum, at the end of each sprint is a day or so of Ceremony which, although it is not coding, is also hard work. Then, the next Sprint starts and software work begins anew.

As the weeks and months go on, teams become tired and burned out. No one can perform at a consistently high level forever. Because there is no scheduled downtime described in the Agile process, it is not unusual for teams to fail to create any down time for themselves. There's not a lot of time for personal development, long term planning, process improvement, or even just desk cleaning.

Additionally, by the time fatigue starts to set in, the Business team has fallen in love with the unending stream of new features and bug fixes. This makes the idea of slowing or stopping the process even more difficult.

The challenge then becomes how to give teams some much-deserved down time without disturbing the release process, quality, or quantity of output.

### **3.1 Individual Symptoms**

Everyone is different, so fatigue will show itself at different times and different amounts across the members of the team. However, symptoms are reasonably consistent. A decrease in individual productivity is not uncommon. "Meandering" becomes more common. That is, starting on tasks or stories and not finishing them in favor of other, less critical but more interesting things. (Wasson, 2014) Additionally, symptoms of general fatigue start to become apparent: irritability, tiredness, lack of focus, disaffection, and boredom.

Above and beyond the mental symptoms, there are other problems and characteristics of agile fatigue that can be damaging. Since there is rarely every any down time, it becomes difficult to schedule enrichment activities like voluntary trainings, classes, and conferences. These sorts of events help keep people inspired, enervated and thinking which. Rituals that used to be carried out at the end of a project like desk cleaning, archiving files, backing up data, etc. have to be purposefully scheduled or else they just don't happen.

### **3.2 Team Symptoms**

Teams can also suffer from Agile Fatigue. One of the first things that organizations do when they make the leap to Agile is that they co-locate the teams to facilitate communication. This tends to work extremely well – everyone has nearly instant access to everyone else on their team. Questions are asked and answered in seconds. The flip side of co-location is the same concern that NASA has in isolating astronauts together in space. Space Agencies test their candidates very carefully to make sure that they will behave well when contained in a small space with only a few other people for months on end. Most companies, however don't or can't do that kind of testing. As time wear on, teammates can get tired of and frustrated with each other.

Another interesting effect is that the team can bond through near-continuous expulsion of negative thoughts and emotions, eventually declining into an "Us vs. Them" state. "Them" tends to be the Business team, or whoever else is feeding the endless stream of stories. As a group, the team can become defensive and argumentative, sometimes kicking back stories that they don't like or that aren't interesting enough.

In addition to the symptoms described here, Robert Galen offers more great information in both his 2013 and 2014 papers that focus specifically on Agile "Burnout".

## 4 Cost of Ignoring “It”

Care must be taken to stop Agile fatigue before it has a chance to take root. Once the signs of fatigue begin to appear, it will be too late for some employees, as one of the most unfortunate side effects of this problem is attrition. In a tight job market, allowing engineers to burn out means that they'll make a quick exit to the next interesting company. If employee retention is a priority within your organization, this will be considered a management failure.

Another cost of burnout is reduced productivity at both the individual and team levels. For the team, no matter how full the backlog is, as time goes on, they group will complete less and less work. Alternatively, they'll settle on a velocity that is much lower than what the team is actually capable of. Essentially, they'll build their own free time into the schedule, but it will likely be more time than they would need if there were measures in place to actively deter burnout.

On an individual level, the need for “coffee breaks” increases along with the time spent standing at the coffee machine discussing the unending supply of stories. Workers are more easily distracted by petty things and, when they do return to work, it takes longer to achieve focus and start making actual progress. Productivity is money, so having a team, or even some individuals, working at reduced capacity means lost income.

Lack of Creativity will also become a problem over time. Teams that consistently crank out code week after week will eventually be hampered in their ability to innovate. Creativity relies on energy and a tired team, by definition, has no energy to spare.

A final, and really interesting side effect of fatigue, is a perceived failure of the Agile process. Tired, frustrated teams will complain that whatever process they are using just isn't working. Even if they are regularly shipping meaningful software, teams tend to believe that the technique is failing. Realistically, the process is actually working as intended, but because they teams are unhappy, they don't feel successful.

## 5 How to Battle Fatigue and Win

The fight against fatigue needs to start with the decision to adopt an Agile development process. Procedures and activities to fend off burnout need to be built into the rules of engagement for your teams and need to start as soon as the switch to Agile. Here are some suggested strategies for fighting fatigue that still result in a reasonable benefit to the organization that goes beyond just rejuvenating the employee.

### 5.1 Rotation

Once a team is established and has a known velocity, start rotating team members out to other teams. Moving more than one team member, or moving people constantly will disrupt team velocity, but moving someone out and moving a new person in every 6 months or so shouldn't severely impact a group. This can be done on a voluntary or assigned basis, but getting team members out to do a couple months of work with another team has several advantages.

#### 5.1.1 Building Connections

Spending time with another team allows people to make connections, develop relationships, and share ideas with other employees. They can also look at how another group implements what is, theoretically, the same development strategy. It is not common for all teams to implement an Agile strategy in exactly the same way. So embedding with another team may foster the spread of more effective techniques. It can also be educational to “see how the other half lives”. If a team has some sort of reputation, spending

time on that team may help spread the truth. E.g., Team X isn't lazy, but what they're working on is really complex and time-consuming.

### **5.1.2 Mentoring Opportunities**

Rotating more experienced workers onto a team with more junior engineers may provide a good avenue for challenging the senior by giving them an opportunity to mentor. In the end, the senior engineer stays engaged and the junior engineers can learn and improve their skills under the tutelage of the more advanced developer. Additionally, code quality may be improved from the additional scrutiny of the senior.

### **5.1.3 Easing the Transition**

Not every team member will welcome the opportunity to rotate to another group. Change can be disturbing for some people. Others enjoy what they're working on and don't want to be disturbed. Rotation should be handled on a case-by-case basis and may require special exceptions and handling for some workers. It's important to stress that rotations aren't permanent (unless that is a desired result) and to talk about the benefits. Additionally, taking the time to discover if there are projects that an employee is interested in working on will help ease the transition.

Another way to smooth the way is to do an incremental transition. Have the employee work part time on the new team for a sprint, then full time for a few sprints, then move back to the old team. This gives the person time to ramp up on the new team while still remaining functional in the old group. Incrementally transitioning a person allows them to feel like they're still contributing to the old team while learning and, when they start their first full week on the new team, they'll be ramped and ready to contribute 100%.

Not every engineer will be willing to rotate. Forcing the issue will likely have a negative result. Hopefully, the unwilling people will see how much others enjoy rotating and will eventually be willing to participate. Ultimately, there is only so much that can be done, so managers need to be flexible and work with their reports to determine the best solution for each individual.

## **5.2 Off-Cycles**

Taking a team member off of all teams for some number of weeks may also yield big rewards. Almost every organization has projects that need to be done that aren't necessarily appropriate for an Agile team. Giving an interested person a chance to work on a task outside of the Agile environment for a week or two will give that person a needed break from the pressure to constantly release code, but also provides benefit for the company in that needed projects or tech debt are getting taken care of.

## **5.3 Celebrate!**

Team lunches, deploy parties, and other celebrations can break up the monotony of daily routine. A surprise afternoon lunch on a cruise ship followed by sending everyone home early one afternoon can have an extremely positive effect on productivity in the weeks following the event. Or at least it did for the QA team in my organization.

The process of planning a celebration can be helpful for managers who also need a rest from the same sort of monotony that frustrates their reports.

Essentially, anything positive that gets teams out of the work environment for a few hours can be highly beneficial in the long term and can completely justify the loss of a few working hours for the group. Kiran Singh has a blog entry that explores other social engineering ideas that can be helpful in breaking up the normal routine. (*Singh, 2014*)

## 5.4 Free-for-alls

Another interesting strategy for revitalizing teams is to give them some reasonable amount of time (1 or 2 weeks) to work on anything they want. The ground rules are simple:

At the end of the designated “Free For all” period:

- There must be demonstrable output
- It should work as expected.
- It should meet a business need.

This kind of project allows a team to learn about and explore new ideas. The team may end up solving a gigantic business need or even fixing something small. But either way, the organization ends up with a new bit of useful software and the team gets to really think out of the box for some amount of time, which can be extremely refreshing.

## 5.5 Tech Debt or Process Improvement Cycles

A common recommendation for easing fatigue is that teams reserve roughly 10% of each sprint for Tech Debt or Process improvement. If a team and Product Owner are disciplined enough to include these kinds of stories in every sprint, it can be beneficial for both the team and the company. Tech debt stories can be less stressful to work on and can make life easier for everyone since they often result in cleaner, better documented code, more unit tests, new tools, or test automation. (*Cromarty, 2014*)

Unfortunately, in high pressure situations, tech debt can fall by the wayside. User Stories devoted to housekeeping, tool research, and code enhancement get postponed in favor of high priority features and fixes. A potentially effective way to combat this problem is to devote an entire sprint to a Process Improvement Cycle (PIC). One in every 6 or so sprints can be set aside to work on projects to improve the team. QA engineers can work on things like automation, test case auditing, and big picture test planning. Dev engineers can tackle writing additional unit tests, researching new tools, working on proof of concept projects, cleaning up and refactoring code, and documentation.

Setting aside an entire sprint allows the team to take a collective breather from high pressure development. Because the PIC sprint is a known quantity, Product Owners can plan accordingly to accommodate the time off of active feature development and bug fixing. Finally, because the PIC stories take up an entire sprint, it is more difficult to postpone them in favor of higher priority work because, for one sprint, they ARE the high priority work.

## 5.6 Directed Special Projects

This is much like the “Free For All” except that the team is given a problem to solve and the leeway to solve it in whatever way makes sense for them. The project should be time-boxed and the required output reasonably defined. But other than that, the team should be free to do whatever they feel is right to get a solution in place.

## 5.7 Vary the Work

Management will always try to keep a team working on the same thing. The principle is that the engineers know what they’re doing and they don’t have to learn anything new, so just keep them working on the stuff they’re good at. The problem is that the “stuff they’re good at” gets old and boring. Having the team work on something outside of their comfort zone can force them to stretch their minds and skillsets. It may slow

a team down temporarily, but in the long run, it can invigorate the engineers and get them thinking out of the box on the tasks they normally work on.

## **5.8 Mandatory Training Time**

Each team member can be allocated a certain amount of time and/or funds to take additional training during the year. Regardless of whether it is an in-house training, or a conference, or a formal class with an external company, each opportunity gets the employee off of a scrum team for a few days, gives them an opportunity for personal development, and lets them think about something other than the endless backlog. Additionally, the person returns to the team with a new skill or at least knowledge that can benefit the rest of the group. It is important each employee be encouraged or mandated to use this benefit. Otherwise, it's easy to let the opportunity pass in favor of "more important" work.

## **5.9 Careful Retrospectives**

Retrospectives are something that teams tend to do out of habit. They address the "What Worked" and "What Didn't Work" items and then they go back to their regularly scheduled routine. However, taking the time to do a full retrospective or to at least assess the temperature of the team members can be very beneficial. There are a many tools available to accomplish this, but simply asking, "How are you feeling?" as part of your retrospective can go a long way toward discovering who on the team most needs attention or at least a break from the mundane.

## **6 In Summary**

Agile Fatigue is a problem that tends to go un-noticed and unidentified in many organizations. Left unrecognized, it tends to eat away at development teams and, ultimately, it results in a perceived failure of the chosen Agile process (Scrum, XP, etc.) By choosing to fight the effects of fatigue immediately upon adopting an Agile development process, unfortunate and expensive problems like employee attrition and reduced productivity can be reduced or avoided entirely.

## **7 Thanks!**

I would like to thank everyone that has helped make this paper possible: My organization, NWEA, who allows me to work on PNSQC papers as a part of my job. The members of all of my scrum teams who, in the last 4 years, have allowed me to be their Scrum Master and have provided me with many amazing ideas and anecdotes from which I have been able to draw ideas to write about. The PNSQC editors, who read endless drafts and still cheerfully give feedback. And my husband, who always patiently listens while I work through my ideas.

## References

Khuon, Tony. 2014 "How to Defeat Burnout Syndrome Before it Kills Your Passions." Agilelifestyle.net <http://agilelifestyle.net/burnout-syndrome> (accessed May 2016).

Hazrati, Vikas. 2008 "Is Burnout Inevitable, while Facilitating Agile Projects?" InfoQ <http://www.infoq.com/news/2008/04/facilitating-agile-projects> (accessed May 2016).

Warden, Jesse. 2009 "Agile Chronicles #8: Demo, Burnout, and Feature Juggling" Jesse Warden Blog, entry posted January 17, 2009 <http://jessewarden.com/2009/01/agile-chronicles-8-demo-burnout-and-feature-juggling.html> (accessed May 2016).

Galen, Robert. 2014 "Can Agile Teams Get Burned out?" Project Management.com. <http://www.projectmanagement.com/articles/287337/Can-Agile-Teams-Get-Burned-Out-> (accessed May 2016).

Simon Cromarty. 2011 "Priority Fatigue" The Agile Pirate Blog, entry posted May 3, 2011. <http://theagilepirate.net/archives/tag/fatigue> (accessed May 2016).

Galen, Robert. 2013 "The Agile Project Manager - Can Agile Teams get "Burned Out"?" rgalen.com. <http://rgalen.com/agile-training-news/2013/9/16/the-agile-project-manager-can-agile-teams-get-burned-out> (accessed May 2016).

Appleton, Brad, Steve Berczuk, and Robert Cowham. 2009 "The Decline and Fall of Agile SCM—and the Rise of Lean SCM" CM Crossroads. <http://www.cmcrossroads.com/article/decline-and-fall-agile-scm-and-rise-lean-scm> (accessed May 2016).

Singh, Kiran. 2014 "Avoid Agile Fatigue" Kiran Sing. <http://kiran-singh.com/2014/06/19/avoid-agile-fatigue/> (accessed May 2016).

Guthrie, Robin. 2012 "Agile Fatigue: Sustaining the Change" Swift Ascent Blog <http://swiftascentblog.wordpress.com/2012/06/21/agile-fatigue-sustaining-the-change/> (accessed May 2016).

Wasson, Kim. 2014 "Agile Fatigue" Agile Project Management Blog <http://patmerg.blogspot.com/2014/09/agile-fatigue.html> (accessed May 2016).



# Building Stakeholder Confidence Through an Automated Testing Solution

Rebecca Long

rebeccal@stcu.org

## Abstract

How do you build stakeholder confidence in your software and also help minimize time spent on manual User Acceptance Testing (UAT) efforts? By getting stakeholders involved in the development of an automated testing suite, you build a foundation of understanding in both what is being tested and how the software is being tested. This in turn also allows the stakeholders to be more familiar with the implementation of the software. Involving stakeholders builds confidence in the software, the test coverage of features and gives them a clearer understanding of why less manual tests during UAT are required.

Building a system to meet this need requires good architecting to allow for stakeholder involvement, easy scalability, portability, and minimal maintenance. At Spokane Teachers Credit Union (STCU), we designed an automation test system called Browser Bot that uses SpecFlow to allow business owners to write the tests, Selenium for driving the browser, a custom library to dynamically pull test data based on test criteria, and TeamCity to run the tests on a Selenium Grid. The partnership between the software department and stakeholders to create this system has already begun to build the desired understanding and confidence in our systems from stakeholders. Using my team at STCU as an example, I will identify some of the common problems, solutions we used and the results we achieved. By following a similar approach, you can also get positive stakeholder participation while meeting the testing needs of your software team.

## Biography

*Rebecca Long is a quality assurance engineer at Spokane Teachers Credit Union in Spokane, Washington. She has passionately ensured the delivery of stable software with a delightful user experience through effective and efficient testing methodologies for the last decade. Her quality centric approach streamlines the software development process and ensures many problems are mitigated before they arise. Additionally, she also co-runs SpoQuality, a quality assurance user group based in Spokane.*

*Copyright Rebecca Long 2016*

# 1 Introduction

Companies are constantly pushing for software to be developed faster, tested in less time, and rushed out the door into production as quickly as possible. Management often believes that it's more valuable to get user feedback immediately with whatever can be released instead of solidifying an application before shipping it. The rise in popularity with the agile mentality seems to be lending to this misconception among business and project managers who may be jumping on this bandwagon before fully grasping the true intention behind the agile development lifecycle. As more managers fall in line with the falsehood of rushing software out the door, or as less managers push back on this idea, the harder it is for software teams to stand their ground to maintain good software practices. Yes, sometimes software does need to be rushed out the door. Yes, sometimes there is not enough time to fully regression test. Yes, there will always be bugs that escape to users. The key is balancing these inescapable realities with a best attempt at risk guided efforts toward avoiding pitfalls.

## 2 The Problems

At STCU, our software team ran into the following problems as we grew in both the size of our agile team and the amount of software we produced. Some of these problems were due to natural growing pains of a team and some of these problems were from the push to rush software out the door quickly.

### 2.1 Relying on Stakeholders as Bug Testers

It has been tried and proven by other companies – the best example being Netscape -- that having your end users as a main source for testing resources does not work out well (Spolsky 2000). Having your end users be your testers produces real world scenarios and real world problems, this is true. But the problem comes into play when your users get too early of access to your software and gain this perception that the software is only full of bugs. Lots of bugs. This damages your team and / or company's reputation with your users. If your users have alternative software to switch to, you could be at high risk of losing them.

One application built by my team is called Toolkit and is for employees to better serve our members with all their financial needs. Toolkit is a complicated Web application that connects multiple internal credit union (CU) systems and external vendor systems into a single point of contact with the goal of minimizing the number of applications employees need to use. Since our user base for this application is employees, our stakeholders and unofficial product owner for this application is another department within the CU. This department performs all UAT for Toolkit and manages communication to staff for new features and changes. As Toolkit began to take off in popularity and the vision of what it could do to help employees expanded, our backlog of features grew and along with it the pressure to push it out the door faster. This led to our internal quality assurance (QA) spending less time on regression testing and the software team relying heavily on UAT to find bugs or missed requirements. As the main source of testing started to shift to the UAT group, they began to find more and more problems that should have been caught prior to it being handed over to them. This led to stakeholders pushing back on tight deadlines in order to allow them enough time to do deeper testing which should have been completed earlier in the process. Toolkit has become such a core application for employees that if we ship a new release with a major (or even sometimes a minor) bug in it, it can cause a lot of turmoil for staff resulting in an increase of help tickets and phone calls from frontline staff to back-office support staff.

### 2.2 Release Tracking

Our team also suffered from some disorganization internally with tracking features and bug fixes. While we use JIRA (Atlassian 2016) for issue tracking, for a long time we did not have it setup to easily connect the user stories and bug reports to code and builds. Inside JIRA we mostly used good versioning but it wasn't consistent and we struggled to know when our manual versioning of issues lined up with the actual code releases.

Unfortunately, this led to the occasional big miss in our release notes that our stakeholders rightfully did not appreciate. Without reliable ways to track anything back to a point in the code base, it was essentially a guessing game at what was being included in a release. Stakeholders were regularly finding items during UAT that were not expected and they always noticed when promised items were missing. This poor ability to track items back to specific releases also made it hard for internal QA to know what to test and look for before shipping anything to UAT or production.

## 2.3 Environment Setup

Toolkit interfaces with multiple third party systems to centralize where employees need to go in order to service members. This includes systems like SharePoint to more complex systems such as our core credit union software as well as external vendors that manage credit cards, credit checks, check ordering, and electronic signatures. It is also connected to other applications we wrote that manage debit card ordering and rates for various account types. My software team only manages the Toolkit application itself, the associated two application database and the other connected internal applications. The majority of applications and vendors Toolkit interfaces with are managed by other teams or by the vendor themselves. There are many potential points of failure, making it all the more important to have solid testing practices and good test coverage before releasing changes into production.

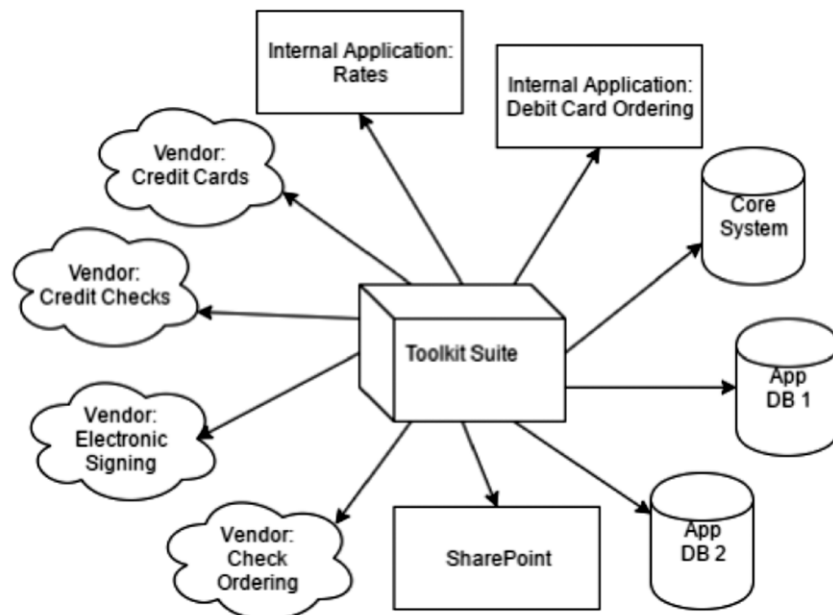


Figure 1: Toolkit Environment

## 2.4 Stakeholder Confidence

As might be expected, these problems led to our stakeholders, and also our users, losing confidence in both the application itself and the team's ability to produce reliable software. Stakeholders are not guaranteed to be technically savvy and thus can have a harder time understanding or sympathizing with problems with software releases. As stakeholders lose confidence in the software, the trust between them and the software team quickly erodes.

For us this meant that the stakeholders became more demanding as a means to compensate for what they felt was our shortcoming. They demanded more detailed documentation on everything included in a release no matter how small. They demanded ever increasing lengths of time for them to conduct their UAT sessions before they would sign-off on any release to production. Stakeholders recognized our testing was insufficient and would double-check every item in the release notes in addition to performing a full regression test of their own. This was both time consuming and a major duplication of effort given that

both departments were now essentially performing the same tasks on the same software before it was released.

Our department began to recognize the negative effect on our stakeholders when we left the majority of testing up to them and did begin shifting testing back to our software team. However, at this point the damage had already been done where stakeholders had seen too many bugs that should have been caught earlier and had seen too many post-release issues that weren't caught by anyone. They had already lost confidence in our software testing and release process.

Their response of demanding more UAT time for them to recheck all our work before a release slowed down our release cycle time. Slower release cycles is the opposite of shipping software faster and more frequently and caused extra tension between departments as we each pushed for what felt like opposing goals – greater testing versus shipping software faster.

## **3 Solutions**

While my team did a multitude of process improvements to address the problems outlined in this paper, one big push has been in creating an automation system that can help elevate some of the testing pressure jointly felt by our stakeholders and our internal QA. The automated system design had at its core the goal of involving stakeholders so that they could be invested in the testing and gain a deeper understanding of what is being tested. The hope was that this involvement would increase their confidence in our software, our processes, and ultimately our team. This would also help rebuild the trust between our departments.

### **3.1 Technical Considerations**

In building any automation system, you will have unique technical considerations to keep in mind. This is based on your specific infrastructure and resource availability. In the next section I describe the things we had to take into consideration when designing our framework.

#### **3.1.1 Complex Dynamic System**

The main part of our infrastructure includes the core vendor system which all our applications integrate into. This system has a nightly process and is not easily setup or mirrored to new environments. Because of the overhead with maintaining any test environment for our core system, our software team was not granted an isolated, dedicated test environment for our needs.

Due to the complexity of the financial world and our systems, the data needed to properly test is not easily replicated in test form so we often have to fall back to having to use copies of real data for many use cases. The test environment we are assigned to use is also used by other departments regularly which results in data moving around a lot without our knowledge. Test data you are using today is not guaranteed to still be in place and setup the same tomorrow.

This setup can create hang-ups in regards to any automated testing. We only have a couple of dedicated test accounts to use which is the case in most test environments. These accounts only cover a couple of common scenarios leaving huge gaps in test data needs. Some test data can easily be created on demand for more simple testing needs. Again, the more complex scenarios need to be using copies of real world data. Given we are a security minded institution, we will not commit into our code base any actual test data or member data that is classified as personally identifiable information (PII). This means that our existing automated tests are frequently breaking because the shared test accounts are used and require regular maintenance. Scaling out a new automation test system needed test data that would be available in a more reliable manner without risking member information.

### 3.1.2 Maintainability

Due to our own limited QA and development resources, it was not feasible to build a new automation system that would require a high level of maintenance. This meant we needed to be sure that test data was dynamic and not hard coded anywhere so that it could be guaranteed to work. The system needed to be smart enough to skip tests if test data was not able to be found or created on the spot for a given scenario. The tests picked for automating needed to take into account how frequently the application is updated and changed. The features to automate tests for were prioritized based on the likelihood of the feature being updated to minimize the immediate need for maintenance.

## 3.2 Automation Architecture

The following automation system, named Browser Bot, was designed as a solution to the major concerns described in this paper.

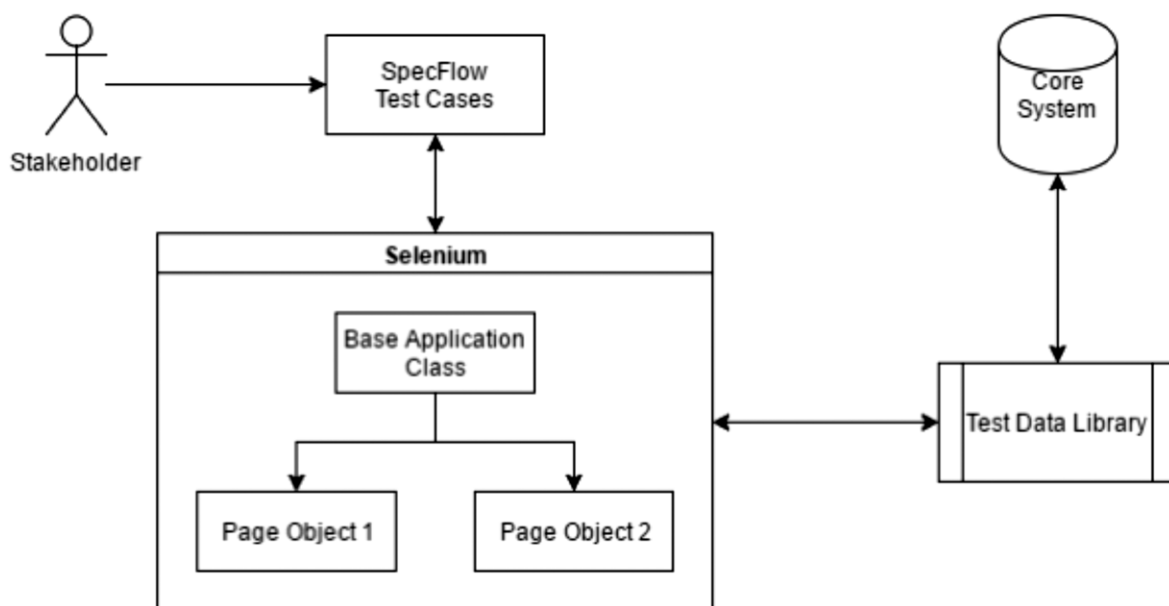


Figure 2: Browser Bot Architecture

### 3.2.1 SpecFlow

We picked SpecFlow to be the entry point into the automation framework. It is a tool that “aims at bridging the communication gap between domain experts and developers by binding business readable behavior specifications to the underlying implementation” (SpecFlow – Cucumber for .NET 2016). With the main goal in mind of building stakeholder confidence, SpecFlow provided an easy way to pull business people more into the software process. Stakeholders are now able to write the tests themselves using their knowledge of requirements, business processes, and real world scenarios. Besides generating an investment from stakeholders in the testing process and building confidence in the system, this partnership also leads to better test scenarios and greater test coverage of the application.

Tests are written using the Gherkin language in the Given-When-Then format (Cucumber 2016). This is a “structured format for expressing scenarios with example data, including pre- and post-conditions” (Gorman and Gottesdiener 2012). It simplifies the process of project stakeholders communicating requirements using business domain language. Involving them in this process generates an investment on their part into the software development cycle and ultimately results in greater confidence in the software being produced.

```

1  Feature: BuildVerification
2    --- Verify that the build is stable enough to run other regression/acceptance tests
3
4  Background:
5    --- Given Phoenix is in Day Mode
6
7  @BVT
8  Scenario: Load Toolkit
9    --- Given I want to use Toolkit
10   --- When I navigate to toolkit
11   --- Then Toolkit loads without error(s)
12
13 @BVT
14 Scenario: Load Member into Toolkit
15   --- Given I have Toolkit open
16   --- When I search for a Personal member
17   --- Then the member should be loaded into Toolkit

```

**Figure 3:** SpecFlow test case examples using Gherkin syntax.

### 3.2.2 Selenium

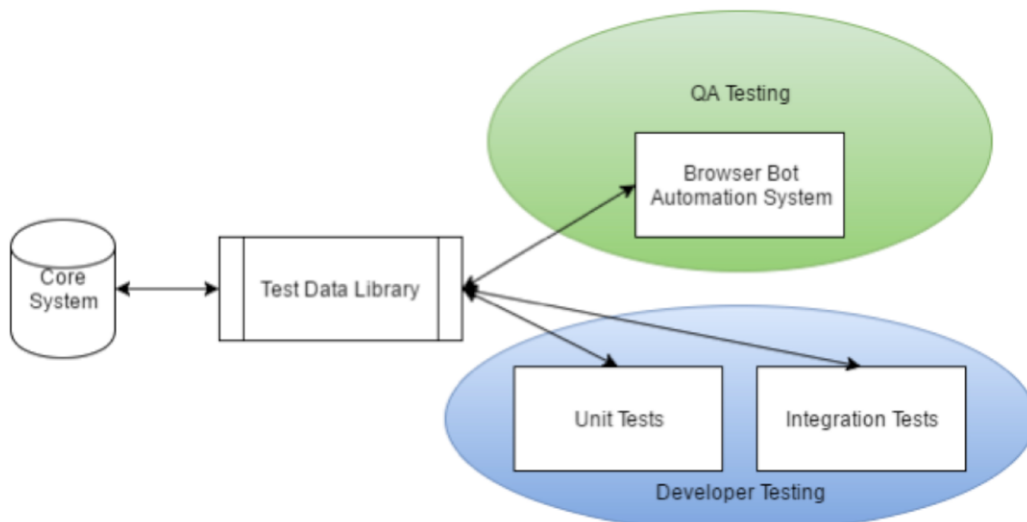
Since our team produces 99% Web based software, Selenium was a natural choice for an end-to-end software automation framework (Sundberg 2011). Selenium pairs with SpecFlow to run all the tests written by stakeholders in the browser (Heppinstall 2012). Selenium is not always the fastest or most efficient automation tool. It runs at the UI level vs at a lower level that can test specific units or functions in the code itself. This can make it harder to trace back bugs that are found due to the number of layers involved in digging into the application to find the code with the problem in it. However, for the purpose of essentially duplicating user and manual testing efforts with the bonus of having a nice visual that can be shown on demand to business people, Selenium is perfect. Business people can see the tests being run automatically. They can see the work that would take them hours or days to perform being done in a fraction of the time. This is a wonderful way to wow the stakeholders as well as demonstrate that the work they would normally do is already being done for them and done faster. Since they wrote the test cases themselves, they can have confidence that what they feel is important to test is being done before shipping the software.

In alignment with the goal of maintainability, the Page Object Model was used when designing out the Selenium part of this architecture. This model creates a page object for each Web page of the application. This page object is then responsible for owning and finding the needed elements contained within it. Using this design aids in creating “a clean separation between test code and page specific code such as locators (or their use if you’re using a UI Map) and layout” (SeleniumHQ 2016). It also provides simplicity when the user interface (UI) ultimately changes:

*“The benefit is that if the UI changes for the page, the tests themselves don’t need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place” (SeleniumHQ 2016).*

### 3.2.3 Test Data

In an attempt to solve multiple test data problems with a single solution, we created an internal NuGet package to dynamically pull data out of our core database system. Our core system does not officially support direct SQL connections and rather must be accessed through a proprietary XML based language which enforces business logic and data integrity. This adds a layer of complexity to querying the core database for test data. Placing the code to pull back dynamic data based on test criteria in a central location streamlined the efforts for acquiring the data. It also allowed it to easily be used by both Browser Bot and any unit or integration tests created by the rest of the software team.



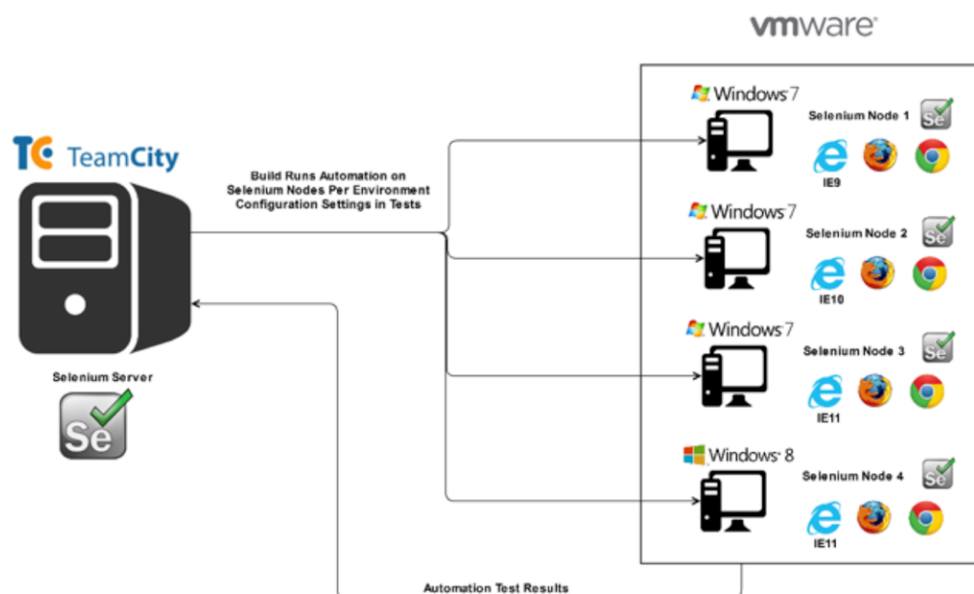
**Figure 4:** *Central library to pull test data from core system can be used by anyone on the team, not just QA.*

### 3.2.4 Build System Integration

Our team uses TeamCity, custom F# scripts and FAKE for running tests and publishing software to any environment (JetBrains 2016) (Hanselman 2014). We added the ability to run Browser Bot tests via these custom scripts. Due to UI based end-to-end tests taking longer to run, a special build in TeamCity was created to run Browser Bot tests on demand without interfering with any of our other builds or day-to-day tasks.

### 3.2.5 Selenium Grid

A Selenium Grid was setup on internal virtual machines to support running the SpecFlow tests (Colantonio 2014). The Selenium Grid server / hub was setup to live on the build server with TeamCity and stands ready for whenever the TeamCity Browser Bot build is run. Nodes for the grid were setup on virtual machines, each supporting a variety of browser options.



**Figure 5:** *Selenium Grid Infrastructure*

After completing the build, both a SpecFlow / Nunit report and a Pickles report is generated. The SpecFlow / Nunit report is more technical and helpful to the development team to troubleshoot errors and failed tests. Pickles “is a Living Documentation generator: it takes your Specification (written in Gherkin, with Markdown descriptions) and turns them into an always up-to-date documentation of the current state of your software - in a variety of formats” (Pickles 2016). This report is great for providing to stakeholders the following: what tests are being run, how those tests are written, what they mean, and what the test result was for them. It’s very end-user and business friendly.

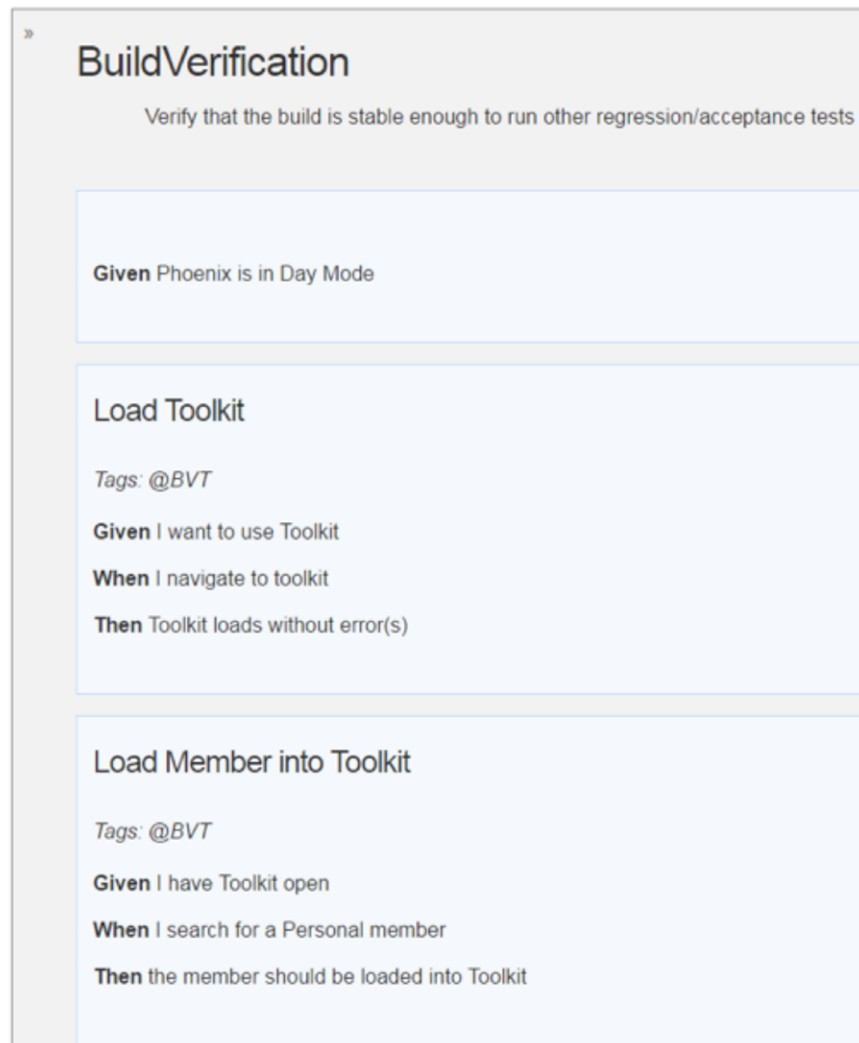


Figure 6: Pickles report generated after running SpecFlow tests through TeamCity

## 4 Results

While this system is still in its infancy stage on our team, we have already seen positive results from the partnership created out of it. During the initial development of the system, a representative from both the software team and the stakeholders were meeting weekly to learn the automation framework tools and educate each other. The software team was teaching stakeholders about the technical setup and infrastructure of the software. The stakeholders were teaching the software team more about the business rules and real world examples of how the Toolkit is used. The greatest educational piece was in learning how to better communicate with each other; building that mutual trust and understanding that fosters a stronger relationship between teams.

This trust was immediately seen during a post-release fire impacting live users. Previously a problem in the live environment would have caused panic from stakeholders and misunderstanding of the depth of the problem. Following the start of the work our departments were doing together for the automation system, the reaction was drastically different. Our stakeholders acted with calmness and even helped think through what could be the issue so that the problem could be resolved quickly. This experience helped reinforce the efforts being put forward with the automation system. While there are more efficient ways to automate software than going through the UI, the side effects of picking this path have far outweighed any cons of this solution.

## 11 Future Work

Going forward our plan is to continue to expand this system. This will include adding tests. Better reporting also needs to be implemented so that it includes integration with our logging system Seq, screen capture on error and even greater Pickles formatting (Seq 206). We will be investigating possible solutions for the Selenium Grid machines using vendors such as SauceLabs since the maintenance of the virtual test machines can often detract from the usefulness of the framework (Gochenour 2016). The goal is to get all our applications using this automated framework and building that same trust relationship with all our stakeholders.

## 12 Conclusion

Any team who desires to ship software quickly and frequently and has struggled with building confidence in their stakeholders or who have lost stakeholder confidence can benefit from creating a similar system. Building an excellent partnership with your stakeholders and including them in the development of an automated test suite can boost their confidence in your test coverage and the quality of your tests. Less time will be needed during UAT for manual testing efforts helping your team to deliver high quality software with a quick release cycle. Your stakeholders, users, management, and development team will be grateful for the increased trust and strengthened relationship that is created between teams. The benefits are huge and the reward is better quality software!

## References

Atlassian. 2016. "JIRA Software." Atlassian. <https://www.atlassian.com/software/jira> (accessed July 31, 2016).

Colantonio, Joe. 2014. "Selenium Grid – How to Easily Setup a Hub and Node." Joe Colantonio: Automation Awesomeness, entry posted October 7. <https://www.joecolantonio.com/2014/10/07/selenium-grid-how-to-setup-a-hub-and-node/> (accessed July 31, 2016).

Cucumber. 2016. "Gherkin." Cucumber GitHub Wiki, entry last edited June 10. <https://github.com/cucumber/cucumber/wiki/Gherkin> (accessed July 31, 2016).

Gochenour, Phil. 2016. "C# Test Setup Example." The SauceLabs Cookbook, entry last edited April 5. <https://wiki.saucelabs.com/display/DOCS/C%23+Test+Setup+Example> (accessed July 31, 2016).

Gorman, Mary and Ellen Gottesdiener. 2012. "Using 'Given-When-Then' to Discover and Validate Requirements." Success With Requirements, entry posted December 9. <http://www.ebgconsulting.com/blog/using-given-when-then-to-discover-and-validate-requirements-2/> (accessed July 1, 2016).

Hanselman, Scott. 2014. "Exploring FAKE, an F# Build System for all of .NET." Scott Hanselman Blog, entry posted March 14. <http://www.hanselman.com/blog/ExploringFAKEAnFBuildSystemForAllOfNET.aspx> (accessed July 31, 2016).

Heppinstall, James. 2012. "Behavioural testing in .NET with SpecFlow and Selenium (Part 1)." James Heppinstall: On Development, entry posted September 24. <https://jamesheppinstall.wordpress.com/2012/09/24/behavioural-testing-in-net-with-specflow-and-selenium-part-1/> (accessed July 31, 2016).

JetBrains. 2016. "TeamCity." JetBrains. <https://www.jetbrains.com/teamcity/> (accessed July 31, 2016).

Pickles. 2016. "What does Pickles do?" Pickles: Living Documentation. <http://www.pickleddoc.com/> (accessed July 31, 2016).

SeleniumHQ Browser Automation. 2016. "Page Object Design Pattern." Selenium Documentation, entry last edited April 15. [http://www.seleniumhq.org/docs/06\\_test\\_design\\_considerations.jsp#page-object-design-pattern](http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern) (accessed July 31, 2016).

Seq. 2016. "Machine data, for humans." Seq. <https://getseq.net/> (accessed July 31, 2016).

SpecFlow – Cucumber for .NET. 2016. "Getting Started." SpecFlow – Cucumber for .NET. <http://www.specflow.org/getting-started/> (accessed July 3, 2016).

Spolsky, Joel. 2000. "Top Five (Wrong) Reasons You Don't Have Testers." Joel on Software, entry posted April 30. <http://www.joelonsoftware.com/articles/fog0000000067.html> (accessed July 1, 2016).

Sundberg, Thomas. 2011. "Testing a web application with Selenium 2." Thomas Sundberg Wordpress Blog, entry posted October 18. <https://thomassundberg.wordpress.com/2011/10/18/testing-a-web-application-with-selenium-2/> (accessed July 31, 2016).

# Winning with Flaky Test Automation

Wayne Matthias Roseberry

wayner@microsoft.com

## Abstract

Flaky test automation drives everybody nuts. Run it once, it fails, run it again - nope! So you stop running those flaky tests, and everyone is happy until a nasty bug gets in there that those tests would have caught.

It is a trap. If you want to run faster and release more often, you cannot afford the wasted time from noisy flaky tests, but in order to go faster, you have to know about the bugs. This paper will discuss how the Office team came to terms with this problem, embraced the reality that real bugs live under the flaky crust and figured out how to use those tests, and their flaky behavior, to their best advantage.

## Biography

*Wayne Roseberry is a Principal Software Engineer at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, SharePoint and Microsoft Office. He currently works on the Office Engineering team, with a focus on test automation systems, strategies and architecture.*

*Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.*

*In his spare time, Wayne plays music, paints, and writes, illustrates and self-publishes children's literature.*

*Copyright Wayne Roseberry 2016*

# 1 Introduction

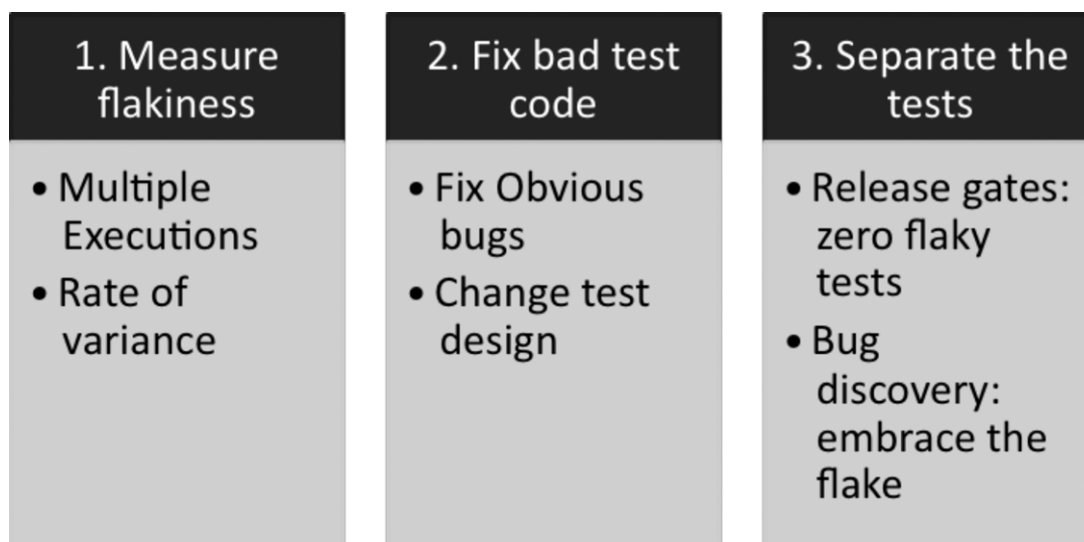
Flaky test automation presents a frustrating and special challenge to any team of software engineers. The tensions to mitigate risk with as much coverage as possible while maintaining small, safe feedback loops with efficient and fast releases pull in opposite directions. Tests that do not yield a consistent signal with failures that do not reproduce easily make that tension even more difficult to manage.

Unmanaged, the problem creates an enormous technical debt that destroys the value of the test automation. Engineers ignore results or bypass processes meant to protect the product from releasing bugs into the market. Teams and projects pay huge costs in lost time, investigation, or attempting to patch bugs that escape.

The problem of flaky automation becomes even more relevant as development cycles increase in speed and releases increase in frequency. Attention shifts from long test phases post code complete to short and fast release cycles and all the way to the developer desktop where developers execute tests during coding (the developer inner loop).

This document describes approaches by the Microsoft Office product group to manage flaky automation. Some of these approaches are centrally run by the engineering team, others are one-off methods used by specific teams or individual engineers within Office. While the examples here are mostly from Office specifically, the practices and approach that are emerging across Microsoft, and outside Microsoft in other companies, appears to be similar.

This document prescribes a three-part approach:



**Figure 1** *Approach for Addressing Flaky Automation*

## 2 Definition and Measure of Flaky Automation

Automation is flaky when it does not give a consistent signal. This may be because of flaws in the test code, but it is also frequently the system under test (SUT) that yields an inconsistent result. For sake of this document, inconsistency shall be measured in the following way:

1. Run a test (or suite of tests) multiple times under the same conditions (build, environment, etc.)
2. Count the number of distinct failures the test yields. A test may fail more than once, but do so differently. These would be separate distinct failures.
3. Count the number of times the test fails.
4. Calculate a reliability rate that rewards for consistent pass or failure, and penalizes for variation

```
Consistency = IF Failures > 0 THEN
  (1 / Distinct Failures) * (Failures / Executions)
ELSE 1
```

The formula above yields a measure of how often a test gives a variant result, whether or not it passes or fails. For simplicity, one can reduce the formula to a binary measure

```
IsReliable = (Consistency == 1)
```

It is then also useful to assess how often a given test is consistent across builds (*CountIf(IsReliable)/Count()*). This gives an image of behavior of a test or suite of tests over time. It is likely you will find that a subset of tests remain in constant stability/reliability flux, while others remains largely stable over time.

## 3 Test Characteristics and Flaky Automation

Test automation is problematic if it is being used in a way that it is not suited for. Flaky automation that is the result of buggy test code is always bad. Flaky automation that is the result of buggy product code is serving its primary purpose, which is to expose the bugs, but such automation is bad when the test is being used improperly.

We need to consider some aspects of test automation and how they relate to test consistency.

### 3.1 Precision Versus Recall

Test automation can be optimized to yield a precise pass/fail signal, to detect exactly one failure condition at a time. Test automation may also be optimized for recall, to find as many failures as possible. When test automation is optimized for precision, it compromises on recall. When test automation is optimized for recall, it compromises on precision.

The precision versus recall classification dichotomy cuts across many other types of tests. It is more about the test's optimizations than it is about exactly what kind of test is happening or exactly what process the test reflects.

#### 3.1.1 Precision Tests Are Highly Stable

Precision test validation is binary in nature. The result of the test is an absolute PASS or an absolute FAIL, requiring no further analysis of the results to establish the result.

Precision automation targets specific anticipated failure cases and fails exactly for those cases. The failure message will be something precise, such as *"Expected: 3, Actual: 2. Ensure that the newly added parent node relationship is counted as part of the set."* Precision tests will fail for no reason other than

what the test code asserted or unanticipated extremely critical conditions such as application crash. Unit tests are an example of high precision automation.

Precision automation is necessary for high speed, critical path processes, such as developer inner loop, code submission, and build release. The faster the release cycle, the more necessary the test precision. The process does not afford spurious failure beyond very specific targets.

Flaky tests are sometimes introduced as precision tests, but they fail to meet the goal by the way they introduce spurious, unanticipated failures into the result set. Time is lost trying to track down root cause, or trying to reproduce the failures.

### **3.1.2 Recall Tests Are Flaky by Nature**

Recall tests force us to be analytical. The final PASS or FAIL state of the test will often require analysis of logged test result data, and sometimes deep team triage.

Recall tests are designed to catch failures not anticipated. They exercise as much code as possible in as many different ways as possible. While failure discovery may come in the form of the same sorts of test assertions as precision tests, typically failure discovery is a result of culling as many possible sources as available; processes that do not disappear, deltas between pre and post execution state, exceptions, errors and assertions in product logs, and variances in execution time. It is often the case that in-code test assertions are turned off or ignored in favor of sustaining as much on-going random activity as possible, with the intent to discover failures from post execution data analysis.

One example of a recall optimized test is an end to end test, or end user test. Imagine a test called “insert image and resize”. The steps may be as follows:

1. Boot application
2. Repeat following for every document in test library, every image in test library
3. Load document from test library
4. Scroll to location image is desired
5. Insert/Image, select test file from test library, OK
6. Select image in document, format/resize, enter new size, OK
7. Confirm:
  - a. Image is expected size
  - b. New document length is as expected
  - c. Text still displays as expected
8. Close document

The test above is designed to happen along as many failures as possible on the way to inserting and resizing the image. Errors may come up relating to document or image handling for specific files. Errors may come up in terms of steps not exiting properly or unanticipated errors. Dialogs may take longer to dismiss than expected, causing errors in the automation navigation control. Virtually any unanticipated condition could trigger a failure, and that is the point, to report when something unexpected occurs. Were the above a precision optimized test the steps would be reduced to a single point of functionality – such as only “insert image” or “resize image”, skipping as many steps as possible.

Another example of a recall optimized test is one where the validation has been broadened. Imagine the same “insert image and resize” test as above, but this time, add the following validation:

- a. Compare screen shot at final state against prior, report diffs as failure
- b. Scan for processes still running after completion of entire sequence
- c. Scan for windows open that were not open prior to the test starting
- d. Execute in debug mode and capture any assert as a failure
- e. Scan product logs for the string “error” or “unexpected” or “exception”

Such a list of possible failure states is going to capture a lot of interesting bugs, but is also likely to capture a lot of false positives. These are the sorts of errors that when product ships with them people ask “Didn’t anybody test this thing?” but when bugs are reported in bulk the same people ask “Why are we wasting our time on these bugs?”

Recall optimized tests operate in an intrinsically flaky world. There is a lot of noise in the signal. There are a lot of real failures that do not manifest every time.

### **Why we sometimes use the wrong sort of tests**

Coverage pressure typically motivates engineers to use recall test suites in situations where high precision test suites are required. This is often a product of dysfunctional relationships between engineering functions or bad practices in release processes.

### **Regression control panic**

For example, if engineers in development and test roles do not feel a strong obligation to each other’s needs, then test engineers may “fatten up” the build validation with recall optimized tests, leading to developers ignoring the results, leading to testers adding even more tests to the suite. Long release processes also tend to motivate addition of recall optimized test automation into the regression and build validation suites, as long times between builds adds greater risk of regression and greater pressure on test engineers to achieve coverage, the result being that the releases take even longer as the automation suite takes longer and longer to execute and results longer and longer to analyze.

### **Kitchen sink**

Sometimes test engineers are instructed by developers to direct all automation at UI and end to end level to get all bugs from the top down. This is proposed as an efficiency technique, “get them all at once”. This doesn’t work as well as it sounds, as test fragility and flaky test results investigation costs overwhelm whatever gains were made by writing and executing fewer tests.

### **Protect against changing behavior**

Sometimes test automation is written at the UI and end to end level to protect against having to change tests when lower components change behavior. Developers frequently want the freedom to change lower level code behavior, being accountable for the top level behaviors only. This is a fine distinction between unit/component/system tests, but it is a poor excuse for having no mid-level layer automated tests at all. The tradeoffs and risks are similar to the “Kitchen sink” case stated above.

The conclusion in this paper on this point is that whether automation is optimized for recall or precision is an important distinction to be made, and that it is best practice to keep tests well separated according to their purpose.

## **3.2 Simple Versus Complex Test Conditions**

The complexity of the test condition has a huge impact on the consistency of a test.

### 3.2.1 Simple Test Conditions Yield Consistent Tests, Fewer Product Bugs

A simple test condition is one where the conditions of the test are both small in number and easy to control. Almost any variable that can change the behavior of the SUT is accessible to the test code. These sorts of tests are almost always unit tests, or very narrowly scoped component tests. Nearly every variable, such as environment, configuration, prior SUT state, integrated components or external systems are removed so long as they do not explicitly pertain to the exact test. These variables are introduced later as test conditions get more complex.

Under a simple test condition state, there is almost no reason why a test should ever yield different results from prior times it executes. Flaky tests for apparently simple problems at the unit test level are almost always the first sign of either bad test code, or that the product itself is written wrong (making simple problems non-simple). This is almost always the appropriate time to begin refactoring product code in order to simplify the test condition for sake of unit tests. This document does not go into detail on refactoring, but outstanding examples and methodologies can be found in the books “Refactoring”, “Working Effectively with Legacy Code” and “xUnit Design Patterns” referenced in the bibliography.

But such test code, once stabilized, tends to stay stable beyond the developer inner loop (within the inner loop, such tests yield high reward, rapidly catching regressions that the developer can often fix within seconds of discovery). They do not add much value catching large, complex bugs of the sort that have huge customer impact. They tend more to catch the kinds of bugs that would have immediately shown up during engineering – typically the first time a tester tried to use the code. The value of these tests is a safer, faster inner loop that forces developers to write better code (an almost inevitable side effect of refactoring code for testability).

### 3.2.2 Point to Point Integration: First Hint of Complexity, First Point of Flake

The first big unavoidable problem spots are the code boundaries between integration points, such as working with external objects, third party APIs or IO. Even when code is refactored, these integration points remain, just abstracted and componentized into isolated pieces of code, and there is almost no way to completely bring their behaviors under 100% control of the test. It is also the case that integration points are one of the most common sources of bugs. Here are some examples of integration factors that simultaneously cause bugs and drive flaky tests:

- Mistaken assumptions about exact behavior under inputs
- Unknown run-time states that may impact the behavior of integration points
- Mistaken assumptions about exception and failure condition contracts
- Inaccessible control points that affect behavior of integration point
- Asynchronous or multi-threaded operations that may alter behavior or drive race conditions
- Global/static state shared which may affect behavior

It is very common that all unit tests will pass splendidly, only to release code that fails because of something unexpected at an integration point. Lack of control of the test condition means the failure may not always reproduce when executed.

### 3.2.3 Complex Test Conditions Yield Inconsistent Tests, More Product Bugs

As test conditions get complex, inconsistency becomes inevitable. Particularly with end to end testing, aspects of the SUT and the fixture are beyond test control, and as such the test is unable to guarantee that the product, fixture or test will behave in the same way every time.

As we add other types of tests that fit complex test conditions, our list becomes expansive and broad. This is usually where one would introduce load tests, stress tests, configuration and environment tests, state based testing, fault mode and error handling tests. Each of these new test categories increases the number of variables under test, and decreases test fixture and test code control of the system state.

In this case, the complex test condition is not a disease to be avoided. The hardest, most difficult bugs in the system manifest in these complex test conditions only. The bugs are not the sort that are visible by code inspection, because they are almost always caused by a misunderstanding of the behavior of an external dependency. Perhaps error codes and exceptions are different than expected. Perhaps memory management and garbage collection behaviors of consumed classes are poorly understood. Such misunderstandings manifest in the code, but almost always only during execution in the end to end state, and usually under complex workloads.

## 4 What do We Know About Flaky Automation?

### 4.1 Stable end to end automation is feasible for most cases

There are lots of opportunity in fixing bad test code patterns.

There are a number of sloppy, bad test code patterns, which occur all too often and contribute to flaky and unreliable results. This document will not explore these patterns exhaustively (see Meszaros for an outstanding treatment of the topic), but a few common ones are:

- **Sleeps to wait out UI events:** change to either explicit check for UI objects, or inject an event-based construct in the code that tests can attach to
- **Timers to wait for asynchronous events:** similar to UI, launch something asynchronous and wait with a timer. It is better to poll status on an event, or create an event handler
- **Dirtying static and global state:** test fixtures commonly share state between tests such as test settings, application settings, database content, etc. Ideally the state can be crafted such that every test is completely independent from all others (for product state, this usually means refactoring), or the test code needs to be highly reliable with controlling state collision, initialization and cleanup
- **“Flying blind” assuming controls and state are ready without validation:** Often seen in combination with “Sleep and wait”, test code all too frequently begins sending commands to controls or resuming progress without confirming the test fixture and SUT are ready to proceed. Sometimes this happens because an application may not provide appropriate hooks (e.g. one time I inherited a piece of automation where in comments was the phrase “Hope to God that the icon is in the upper left of the window” right before the code to initiate a click).

The first attempt at stabilizing automation will likely close large gaps through cleaning up bad test code alone. I have seen teams make as much as 20 percentage point gains in test reliability with this sort of effort.

#### Do less in the test

As discussed in the end to end test example prior, doing more offers more opportunity for failure. If the point of a test is to test one behavior, then the test code should do as much as it can to only exercise that one behavior. This is easy at unit and component test level, but harder with integration and system tests, especially UI. There are several ways to approach this:

**Direct navigation:** Whether it is inside of an application or a web site, it is often possible to go directly to the point where the test executes rather than navigating through a long path of menus, pages, buttons, controls and dialogs. Web pages often allow direct access to some sequence via a URI. Applications may or may not have a means for automation to get directly to some state. In either case,

it is sometimes necessary to change product code to allow such direct connections to avoid spurious failures on the path to a specific test.

**API calls for initialization, UI for test:** One semi-direct path for a test is to call product APIs to initialize test state, and then use the UI and user level controls to perform the actual test.

### **Reduce surrounding content/state**

### **Shifting from out of proc to in-proc**

Most end to end and system tests operate out of process. The test code runs in a completely different process than the SUT. The test code may be sending UI events to the application via a message queue, or it may be sending requests across a network connection, or perhaps dropping packages or files off into work queues for processing. Whatever the means, out of process testing offers very little control of the test fixture for the test code is almost always very flaky.

Many times, teams will evaluate their tests and move many of them to in-process testing. This means that behaviors being tested must be exposed via an API that is directly callable by the test code and loads into the same process as the test code. This gives the test code more control over the fixture state. It also has the added benefit that call stacks at point of failure will contain whatever product code was active at that point in time alongside the test code.

The typical split is to take business logic related tests, the flow of the system from one state to another, and move them to in-process tests. They are separate from UI tests, which focus exclusively on whether or not the UI behaves correctly based on test conditions. This allows deeper, more in-depth coverage of complex business logic state without having to absorb the difficulty and flakiness of UI testing.

### **Shifting from end-to-end to component and unit tests**

Similar to moving tests from out of process to in process, much value is gained by moving tests from end to end and system testing level to component and unit test level. Unit tests are stable, fast and reliable and leveraged appropriately can return much more power than the same tests performed end to end.

The trick is to pick the correct tests. Traditionally, there has been an artificial and dysfunctional separation of “developers do verification tests” and “testers do all the other tests.” But a large portion of the testing domain includes tests which are far more efficient and practical at the unit testing level, but were written as end to end or system tests purely because the tester did not have permission to add unit tests to the project.

My rule of thumb is “fat test domains should run as unit tests.” A fat test domain is one where the number of tests expands rapidly from combinations of test variables against single aspects of the behavior under test. Some examples:

- Special character handling
- Complex data parsing
- Complex state permutations
- Incorrect data
- Error and exception state handling
- International and localized data
- International environment settings
- Etc.

All of the above in a typical test strategy are paired up against different system inputs or variables and result in an extremely large number of tests. Almost all of these tests could be crafted as well written data-driven test engines that execute against low level product APIs. The difference in time to execute is huge (sub-second versus sometimes multi-minute) and reliability enormous.

## **Testing in Production**

This document does not go into many details about testing in production, but moving tests from end to end automation into some sort of production system monitor and telemetry signal test is an often-used technique. Done correctly, and in a very intentionally well-planned manner, testing in production can yield faster bug discovery at lower cost. The service under test must meet certain criteria, though, before such practices are safe. See “Testing Services in Production” by Stobie for an excellent treatment of the subject.

It is also possible to do testing in product/end to end testing hybrids. Rather than deploy a simulated environment into a test laboratory, deploy the code under test into a slice of the production environment, but isolated from customer workloads. From there, the test automation targets this slice, generating a synthetic workload against a real deployment. This practice is becoming more and more common as service deployment and update gets more and more streamlined.

## **4.2 Small quantities of inconsistent tests have a huge impact**

It is common that 15-20% of tests remain flaky, even after efforts to stabilize them (Google reports 16% of tests are flaky, Micco). This creates a problem as pressure to release more often pushes teams to execute more tests more often. Therefore, the faster and more often you release, the more of an impact inconsistent tests have on the team. The more tests are executed by average engineer, the higher the average test consistency is needed to keep the team efficient. The more frequently any given test is executed, the more reliable it needs to be to avoid exposing somebody to a flaky result.

Consider a test with a 99.9% constancy rate executed 1000 times a day. On average, it is going to yield a false signal every day. Consider a test suite of 1000 tests, where the average test consistency rate is 99.9%. On average, that test suite is going to yield a false signal on every execution.

Example:

The average Office engineer executes ~300 automated tests prior to every code submission. This means that average test consistency rate needs to be better than 99.7% in order to avoid every single job yielding a failure.

## **4.3 More bugs fixed, but lower fix rate come from inconsistent tests**

Despite our frustrations with flaky tests, they yield real product bugs, and removal of tests that exhibit flaky results represent real risk against product quality. The following is gained from an analysis of test automation failure triaging inside Office. Results are likely to vary a great deal in other teams, companies or environments.

One analysis we did was to compare total product bugs found by test automation and fixed in product against only those that came from automated tests that were stable (60% of tests at time of analysis). The end result yielded a 27.5% reduction in product bugs fixed. In other words, those bugs would have escaped past the test automation phase and into the hands of end users:

<b>Total Bugs (test and product bugs)</b>	331
<b>Total Product Fixed Bugs</b>	120
<b>Total Bugs Remaining After Removing Flaky Scenarios</b>	220 (66.4%)
<b>Total Product Fix Bugs Remaining After Removing Flaky Scenarios</b>	99 (82.5%)
<b>Total Scenarios</b>	1278
<b>Remaining Scenarios (after removing)</b>	779 (60%)

The above analysis was performed before Office had started execution of reliability runs, so there was little known about the behavior of tests over large numbers of iterations against the same build. Afterward, many practices had changed in terms of bug triaging practices and automation failure turnaround. Later analysis showed even more interesting trends:

Reliability Rate	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	Total Bugs
Bugs Fixed	12	5	11	14	15	26	14	53	40	48	11	249
Percent of Fixed Bugs	5%	2%	4%	6%	6%	10%	6%	21%	16%	19%	4%	

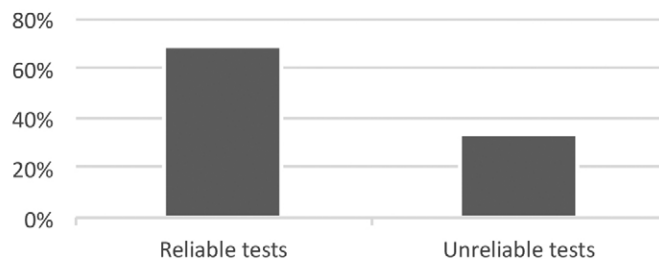
Tests that were 100% reliable accounted for only 4% of the bugs fixed overall (not distinguishing between product and all bugs). There was an obvious trend toward mostly fixing bugs where reliability was between and 60-99%, but still a non-trivial collection of fixed bugs remained.

This higher fix quantity comes at the price of much, much higher noise.

Percentage of bugs filed



Fix rate of bugs found by test automation



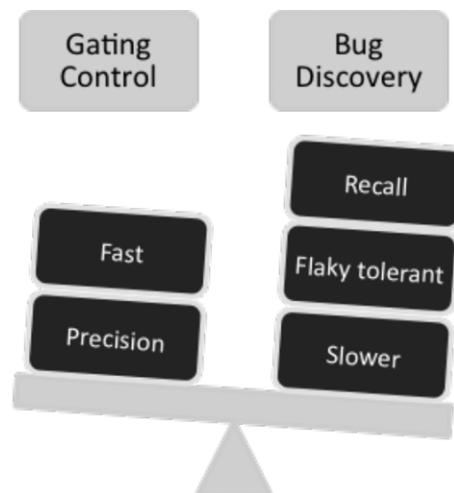
**Figure 2 Reliability of Tests and Bug Fixing**

As shown in the chart above, for the same set of tests, bugs found by non-reliable tests accounted for 98% of the bugs reported, but only 33% of those bugs were fixed. Meanwhile, reliable tests accounted for just 2% of the bugs reported, but 69% of those bugs were fixed. It is clear that unreliable tests were yielding far more bugs than reliable tests, but did so at a much lower rate of fixing.

## 5 What to Do with the Flaky Automation

Once test automation reliability is understood and measured, you need to do the right thing with it. The recommendation this document makes is to separate it into two suites. A gating control suite and a bug discovery suite.

### 5.1 Divide Execution into Gating Control and Bug Discovery



**Figure 3** *Gating Control Versus Bug Discovery*

Gating Control is any test suite that is used to control release of code to a next step in the engineering process. Such steps may be: engineer submits code to repository, branch integrates to parent branch, main build is released from a branch to engineering team, product code is released to end users. Failures on gating control slow down the release, creating expensive delays. Some gating control suites, such as submission to repository are executed at very high frequency, and others, such as product release to end-users are executed at a lower frequency. The primary purpose of gating control is criteria evaluation and regression detection. Gating control requires precision tests.

Bug Discovery is any test suite that is used to discover bugs in the product. Such a test suite does not have to gate release into any particular phase of the release process. Failures in bug discovery should generate work in the product backlog to analyze results and fix product bugs. The primary purpose of bug discovery is to discover as many bugs as possible. Bug discovery requires recall-optimized tests.

Example:

Office runs several types of automation test jobs:

#### **Developer Jobs (gating control)**

Automation run by developer prior to code submission to repository. Tests are a subset of the product BVT tests (determined by what code was changed on submission). Developers are expected to address failures prior to code submission. Hundreds of jobs per day.

#### **Team Branch Looping Jobs (gating control)**

Regular jobs against current code in team branch of repository. Tests are typically same as product BVT (Build Verification Tests). Failures may result in a back-out of any submissions since previous job. Dozens of jobs per day.

#### **Integration Jobs (gating control)**

Jobs run prior to push of product team branch into parent (main) branch. Tests are typically same as product BVT tests. Failures are addressed prior to integration. One job every 1-5 days.

### Main Build Validation Test (BVT) Jobs (gating control)

Jobs run daily against integrated main branch. Output is official drop of Office. Failures in suite set individual product team “zones” into read-only mode until the failures are addressed – i.e. integrations and submissions to that code are prevented. One job per day.

### Main Build Code Validation Test (CVT) (bug discovery)

Jobs run daily against integrated main branch. Failures are recorded as bugs, but do not gate any processes at all.

## 5.2 Create a Reliability Measurement Run

The tests used for both gating control and bug discovery need a reliability calculation. This means the tests must execute several times, the more times the better. How many times is determined by the amount of reliability precision you need to measure. In order to assert a 99% test reliability at least 100 executions of the test under the same conditions are required. Likewise, to assert 99.9% reliability, at least 1000 executions are required.

This many executions may not be practical or cost effective. It depends on the capabilities of the automation system and on the execution times of the tests. You may be able to accept a compromise. For example, if you can only manage 20 executions of a suite per build (which can assert, at best, an  $1/20=5\%$  failure precision) you could achieve up to 1% failure precision by aggregating the reliability of tests across 5 builds. The measurement is somewhat inaccurate, as tests where reliability got either better or worse inside the span of 5 builds will be obscured, but this may be a manageable inaccuracy. The measurement is not so much about extremely high precision every build as it is about generating a number capable of managing a suite for efficiency over larger periods of time.

### Example from Office: Reliability Run:

A background process picks up both the BVT and CVT test suites and executes them repeatedly against whatever most recent build completed the BVT run. During the week, where builds come out daily and there is less pressure on the automation system, 100-200 executions of the entire suite execute per build. Failures are automatically identified and checked against known failures, and new bugs automatically filed for any new failures. Existing bugs are updated with failure hit statistics. Reports show teams reliability statistics for every test in the suite.

Scenario ID	Scenario Name	Config	Scenario Path	Consistency Rate	Pass Rate	Executions
#Scenario: 1117						
211367	ATL_WordUnmanagedSimpleCustomTaskPane	client64-win10d-next	OfficeVSO\OC\Word\Core Engineering\Programmability	0.00%	19.64%	532
155518	Backwards Compatibility - Save Docx as Doc	client64-win10d-next	OfficeVSO\OC\Word\File IO\Format\OpenXML	0.00%	30.18%	532
137638	Roundtrip VB Project	client64-win10d-next	OfficeVSO\OC\Word\Core Engineering\Programmability	0.00%	31.96%	532
195503	InsertOCXviaRibbon	client64-c2r-win81	OfficeVSO\OC\Word\Client Services\Document Content\OCX	0.00%	42.77%	556
195503	InsertOCXviaRibbon	client64-c2r-win10d	OfficeVSO\OC\Word\Client Services\Document Content\OCX	0.00%	46.59%	557
183633	StylesPaneUI	client64-c2r-win10d	OfficeVSO\OC\Word\Client Services\Formatting\Styles	0.00%	67.12%	554
204326	Acceptance: Insert Envelopes	client64-c2r-win81	OfficeVSO\OC\Word\File IO\Mail Merge	12.50%	71.49%	554
379635	InProcAutomation : Client : Rtc : CVT	client64-c2r-win10d	OfficeVSO\OC\Word\Collaboration\Coauthoring\RTT	0.00%	75.00%	4
174440	OpenComplexFile_SaveAsDocm	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	12.50%	77.31%	554
177535	OpenComplexFile_SaveAsHtm	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	12.50%	77.35%	554
397308	Word OCS - Bluechicken E2E (IL+OL)	wac-dc	OfficeVSO\OC\Word\Collaboration\OCS Integration\Server-side	0.00%	79.47%	1683
380428	InProcAutomation : Client : EDP : CVT	client32-c2r-win10d	OfficeVSO\OC\Word\EDP	12.50%	88.98%	554
141434	OpenComplexFile_SaveAsDoc	client64-c2r-win10d	OfficeVSO\OC\Word\File IO\Open-Save	0.00%	89.20%	560
311152	BootOpenTypeText	client64-win10d-uni-next	OfficeVSO\OC\Word\File IO	14.29%	89.66%	536

Figure 4 Office test automation reliability report.

### 5.3 Move Automation into Gating Control Based on Reliability

Once the suites are separated into gating runs and bug discovery runs, it is time to manage the suites based on reliability numbers. Here are the principles to keep in mind

- Gates demand reliable “A fail is always a fail” signal – hence high reliability
- More frequently the gate is tested, the higher the reliability required
- More frequently the gate is tested, the fewer tests should be in the suite
- If there is no gate, then frequency of execution has no penalty

Pick a reliability threshold for each gating suite. Pick something pretty high, but not so high as to be impossible. If 50% of the suite is at 80% reliability and below, then you might need to compromise with 80% for a short period of time to give teams the chance to address reliability issues. Or maybe that 50% of tests does not need to gate releases. This is a balance you will need to gauge on your own.

Once you set your threshold, begin moving and removing tests. If you trust your threshold, consider doing so via an automated process. Anticipate a lot of back and forth conversation about whether a test should be just moved or removed completely. Anticipate a lot of anxiety over tests that cover critical functionality, but which do not meet the bar. Engineers will want exceptions to the threshold. You get to decide how rigid you want to be. Experience in Office has shown that a period of getting used to the idea works well before dropping a firm “no exceptions” hammer.

Ultimately, the threshold that works best is one that balances coverage goals and how much time you can afford delaying releases for spurious failure investigation. If at least one flaky failure manifests on every single job, dozens to hundreds of times a day, then perhaps moving to the bar to only happening one out of four jobs, or one out of ten jobs is a huge gain.

Coverage is similar. A brutal, numbers only movement of tests to later and later gates and into bug discovery risks a bug discovery later than you want. Maybe a customer sees a bug before your automation finds it. Or maybe an official build is released to engineers that breaks a feature from some other team. This is inevitable and the best way to approach it is to individually pull back tests where the cost of that escape is unacceptable. This means that whatever test it is needs to be stabilized before moving to the higher gate. Either that or accept the escape risk as a balance against faster releases.

Example:

**BVT Test Demotion:** Office runs an automatic daily process that examines the reliability results of all BVT tests for the last 7 builds. Any test that does not sustain an aggregate 95% reliability over 7 builds is automatically moved from the BVT suite into the CVT suite. Beyond automatic demotion, individual teams use the reliability report data to push their BVT test reliability as far as 99% and higher. Current (as of July 27, 2016) overall Office BVT test reliability rate is > 98%, often 99% or higher.

This same push has also substantially improved the developer experience. Previously, every automation job launched by a developer to evaluate code for check in had at least 1 failure, 100% of all jobs failed. Engineers would either execute twice (and hit a different failure) or scan prior results to see if their failure was known previously. The current rate of failed jobs is 60% - a full 40 percentage point improvement in passing job rate. It is well worth noting that the average number of tests for passing jobs is 85, whereas the average number of tests for failing jobs is 801, confirming positions elsewhere in this paper regarding the impact of test suite size on reliability demands.

This notion of curating automation suites based on test reliability is not isolated to Office or Microsoft. A similar practice as stated in the example happens at Google (Micco).

## 5.4 Move Flaky Automation into Bug Discovery

Any test that does not meet the reliability criteria for gating processes should be used for bug discovery. This permits the execution of tests where stabilization is either unrealistic, or would render the test too sanitized to effectively discover new bugs.

Working with flaky automation requires a different approach, both in terms of how the test is executed and in how the results are processed.

### Change in Execution Style

Almost all tests contain some sort of pass or fail assertion inside the test method. But at this point it is best to modify that approach. Look for more signals of potential failure. Some approaches might be looking for processes that are left open after test completion, windows or dialogs that are not dismissed, files left open, asserts thrown by the application. Application log files and event logs should be mined for exceptions and error content. System resource usage should be monitored for leaked memory, resources, over-active file system access. Latency and throughput checked for diminished performance.

Some forms of testing remove formal validation completely and just focus on applying load. This has been a long time common practice for performance, scale, load and volume tests. It is not too uncommon with “monkey” tests, applying stochastic load patterns to drive the application to error prone states. Rather than trying to increase the ability of the automation code to successfully maintain the complex state of the test fixture, reduce the amount the automation code cares about. Just keep the SUT moving.

### Change in Analysis

Recall optimized tests demand more analysis and time than relying on a report of which tests passed and which tests failed. The widened definition of potential failure and the lower validation of load generates a lot of false signals. These signals largely demand a significant investment in human time and effort, which is why the activity is used to feed the product backlog with work rather than gate the release processes.

The challenge, then, is to sort through the volume. The most common techniques are:

#### Failures that have never been seen before

The easiest way to address failures is to tackle everything that is new, but if the rate of intermittent failure is high enough, or the suite large enough, this becomes an overwhelming problem. Teams that start here, before cleaning up their product and test suite, usually abandon it for one of other methods mentioned here.

Teams also tend to ignore failures that appear in a run when that failure has been seen previously. The justification for this is that whatever change was submitted is not likely the cause of the failure, hence the engineer should proceed with code submission. This makes sense when focusing purely on whether or not a specific change has caused a regression, but extensive use of the practice creates a large pile of engineering debt and a general distrust of the test automation.

This was the first technique the Office team used to get a handle on its intermittency, and the impact on test system distrust was dramatic. Over time it came to a point where every automation job used to gate a check-in had some sort of failure in it, and engineers would either ignore them, or run the job twice, submitting code if the failure did not happen both times. The overall passing rate of the suite diminished over time until it was a largely ineffective way to evaluate a build.

Eventually, the Office team followed the suite splitting practices advised in this document and adopted a strict “100% pass” policy on code submission and build validation jobs. The result was a substantial improvement in the engineer code submission job passing.

To use this technique, the automation system must be able to identify failures as pre-existing or new. Automated tests may fail for many different reasons. The Office team uses a mechanism that utilizes a combination of string matching templates, call stack frame matching and comparing edit distance between failure messages to determine if a failure is new or already known (Robinson). Once a given failure is known and identified, the automation system can track when it was first seen, what builds it has been seen in and how often it occurs.

### **Failures that occur more often than others**

This is typically the first, and easiest priority decision with any failure. More frequent failures are easier to diagnose and easier to validate when fixed. But it is also clear that a failure which happens more often is going to cost more overall than a failure that is rare, all other aspects such as severity and scope being equal. Frequency therefore usually, although not always, works as a good rule of thumb for priority.

### **Failures that are likely to also occur inside gated processes**

It is advised to include gating tests inside the bug discovery suite. In addition to helping compare results between tests, it is also useful for ferreting out lower occurring intermittent failures in order to fix them in the gating tests. It is also useful to compare results of tests in one context or another. Test automation environments are not always completely clean, and factors like system load, resource availability and machine re-use may cause tests to change their intermittent failure. Regardless, tests that gate processes have high priority for fixing intermittent issues.

### **Failures that occur inside code paths that are known to be problematic for customers**

This requires more insight about how the product is used and where customers are having difficulty. If a feature is moderate to high priority, then fixing intermittent failures in that feature will have a larger return on investment than on features that are almost never used.

Don't fall into the trap at only looking the highest priority features, those that fit into marketing team demos. Usage patterns on any product or service tend to have a long, fat tail, which means that there is a very large quantity of behaviors that range from low to moderate usage, and any failure hitting that range is going to have a big cost impact on the product.

### **Failures that suddenly change their rate of occurrence between builds**

This technique acknowledges the system experiences intermittent failures, and uses sudden large increases in the rate of occurrence indicate a change in the system for the worse. We have seen teams inside of Office use this as a mechanism to draw team attention to failures they might have ignored otherwise. This also a failure investigation pattern that has been reported by companies such as Google (Micco).

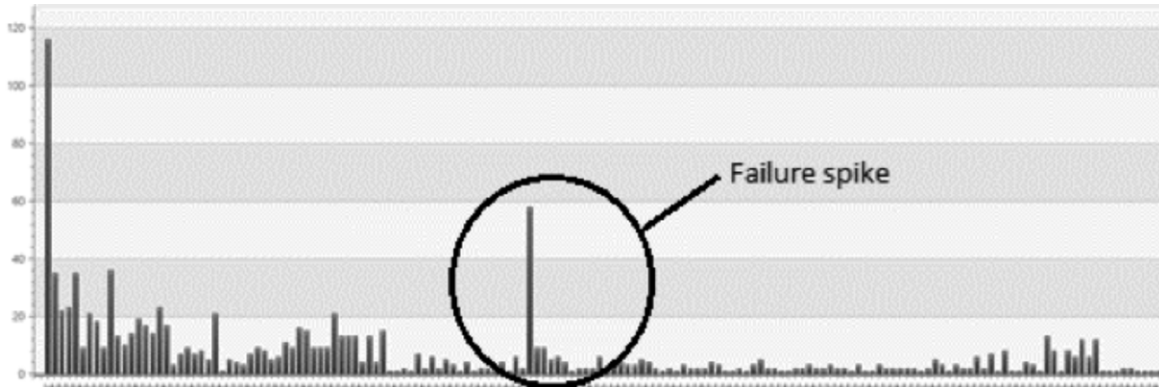
### **Failures that suddenly appear “new” against the same build**

I discovered an instance of this while collecting charts for this paper, and realized it was an important detail to examine. When a test suite is executed multiple times on the same build of the product the new failure discovery rate is expected to start high, and then drop off progressively with each iteration. There will be some degree of variance around the slope away from a projected trend line, but that variance should either be single spikes or not much larger than the average variance.

When there is either a series of spikes in “new issues” around the same sequence in time, or when the variance is much larger than the average new failure count discovered per iteration, then that is a clue that something systemic is affecting the test runs.

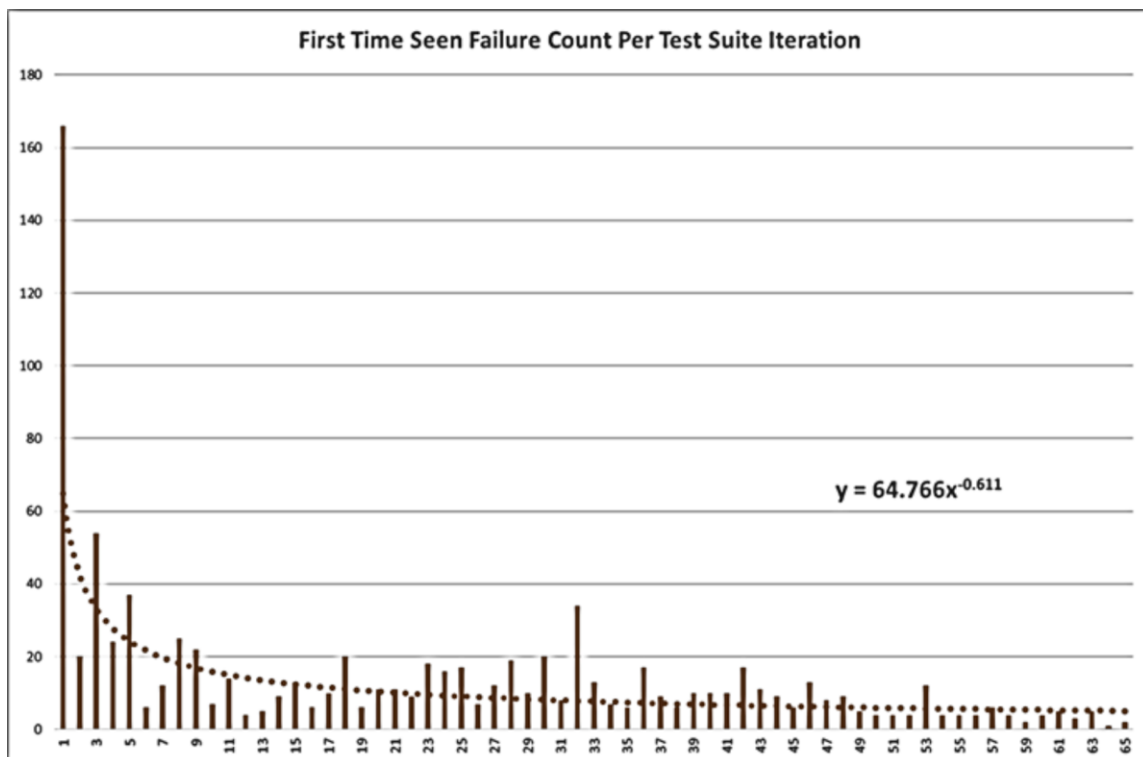
The chart below is taken from the Office test automation reliability run, all for the same build on July 24, 2016. The suite contains ~21k tests, mostly end to end. The horizontal axis shows progressive iterations of the suite; the vertical axis shows the number of new failures reported

per each iteration. The slope follows a diminishing number of discoveries over time, as expected, but there are several “new issue” spikes that vary a great deal from the trend line that suggest something was affecting the suite behavior overall. Such information offers clues as to the source of intermittency.



**Figure 5** Office test automation reliability suite, July 24 2016

**Projecting “When will these tests stop discovering bugs”:**



**Figure 6** Office test automation reliability suite, new failures from 7/19/2016

The fall off rate of new failures discovered per iteration tends to be a power curve, where the area under that curve is the total number of new failures it is possible to discover with that test suite on that build. While the value of new failures may be slowly approaching zero for a long time, it is possible to arbitrarily set a threshold (e.g. “less than 1 new failures discovered per run”) and project how many iterations are needed to stop discovering failures. In the above example, that formula would be:

$$\text{Fall off rate} = 64.766 x^{-0.611}$$

$$\text{Iterations at } < 1 \text{ failure per run} = \frac{1}{64.766} = x^{-0.611} = \left( \frac{1}{64.766} \right)^{\frac{1}{-0.611}} = \sim 921 \text{ iterations}$$

Likewise, total number of bugs discovered at that point is an integral of the equation bound by X= 1 and X= 921. That formula looks like:

$$\int_1^{921} 64.766 x^{-0.611} dx = \sim 2202 \text{ discovered failures}$$

From this, you can extrapolate how many iterations are needed to see any given percentage of the total bugs the suite will find. The progression is non-linear, so a substantial percentage of the total bugs is discovered with few iterations, with diminishing returns for each iteration. The values from the example are as follows:

1102 (50%) = 185 iterations

1652 (75%) = 467 iterations

1981 (90%) = 716 iterations

Doing the above gives a very powerful assessment of the discovery value of a suite. You can balance the value of extra iterations against the value of the bugs found with each iteration. You can also use the number of iterations required to hit your threshold as a quality statement about the intermittency. In this example, 921 iterations would require very large numbers of machines. We can find about 50% of the bugs the entire suite will find in 185 iterations, but even that is a substantial investment. A goal, then, would be to reduce that number so that all the failures the suite is found are discovered more quickly – in fewer iterations.

## 6 Conclusion

While flaky test automation presents many challenges and difficulties, there are practical ways to address those challenges. Sometimes the flake is a sign of bad test code, sometimes it is signal that there is a lot of bugs to be found with the test. Measure the automation for reliability and consistent results, and then use that information to either target fixes against the automation and product, or separate the automation into precise/reliable tests that protect phase gates in the engineering process or recall optimized less reliable tests that find bugs in large quantity. Different analytical approaches and practices can help any team get their test automation under control.

## References

Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Erich Gamma, 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional

Michael Feathers, 2004. *Working Effectively With Legacy Code*. Prentice Hall

Gerard Meszaros, 2007. *xUnit Design Patterns; Refactoring Test Code*. Addison-Wesley

David Scherfgen, Online Integral Calculator, <http://www.integral-calculator.com/> (accessed July 26, 2016).

Robinson, M. P. Test failure bucketing, U.S. Patent 8,782,609 July 15, 2014

John Micco, Flaky Tests at Google and How We Mitigate Them, <http://googletesting.blogspot.com/2016/05/flaky-tests-at-google-and-how-we.html>, blog entry May 27, 2016

Keith Stobie, Testing Services in Production, PNSQC paper 2011, [http://www.uploads.pnsqc.org/2011/papers/T-11\\_Stobie\\_paper.pdf](http://www.uploads.pnsqc.org/2011/papers/T-11_Stobie_paper.pdf) (accessed July 7, 2016)

# Automated Testing for Continuous Delivery Pipelines

Justin Wolf, Scott Yoon

justinwo@cisco.com, scoyoon@cisco.com

## Abstract

Today's continuous delivery (CD) pipeline is a collection of many technologies that ensures the highest quality product reaches production. As a new feature moves from concept to production, it goes through several phases of testing. However, for the CD pipeline to remain efficient, the bulk of this testing must be automated. Unit tests, load tests, integration tests, and user interface tests all require automation infrastructure. In our product, we use Gerrit, Jenkins, Gulp, Protractor, Jasmine, Docker, and Kubernetes among other supporting technologies to perform automated testing on every release, execute long running soak tests, and support rapid iterations on deep performance tuning activities with complex cluster configurations. This paper illustrates how to weave together these testing technologies and integrate them into the CD pipeline so that every release maintains high quality while continuously delivering value to our customers.

## Biography

*Justin Wolf has worked in software engineering since 1988 as developer, architect, product manager, and engineering manager. His employers have ranged from game companies, to government contractors, to computer networking giants. At his current employer, he manages a team of server and embedded device software engineers to provide home network management solutions to large Internet service providers. These SaaS products help service providers support their millions of customers to reduce cost and improve retention. He enjoys aviation and sailing in his spare time and lives in southwest Washington with his wife, daughter, and pets.*

*Scott Yoon has experience in international quality assurance, quality assurance, and quality engineering throughout his career at Adobe Systems, KIXEYE, and Cisco Systems. His works have covered many aspects of software quality using both manual testing and developing automated test solutions. He likes to travel, hike, and swim whenever time permits. He recently moved to Oregon to unfold the next chapter of his life.*

Copyright Justin Wolf, Scott Yoon 2016

# 1 Introduction

These days, just about every service and application across every platform is either already continuously delivering updates to users or seeking ways to shorten update cycle time. Until recent years, many products were hindered in this pursuit by keeping development and testing resources in separate silos and by a general lack of high fidelity automated testing. These two problems meant that overall test cycles were usually lengthy and frequently incomplete. A better way is to bring testing resources into the development team and integrate development and testing into the same continuous delivery flow.

Our team began working on a new product approximately 18 months ago, which allowed us to make a clean break from the previous development and test environment of our prior 8 year old product. However, even legacy environments can be retrofitted with the techniques used in our new pipeline and converted from heavy-lift upgrades to continuous delivery, regardless of deployment style.

We [throughout this paper, “we” refers to the authors and/or their product development team, --ed.] invest heavily in automated testing in the form of unit tests, user interface (UI) tests, and end-to-end (E2E) tests. Though creating and maintaining the automated tests and associated infrastructure is nontrivial, the return on investment is enormous because we can confidently release updates to our users many times a day without worrying that some seemingly unrelated piece of the product is now broken.

In this paper, we start by discussing our product development methodology and why automated testing is so critical to our continuous delivery (CD) pipeline. This is followed by detailed discussions of the technologies and techniques we use to automatically test each build, even as a part of the code review process. Finally, we’ll lay out our plans for continuing to improve our test infrastructure and coverage as our product becomes more established, complicated, and critical to our business.

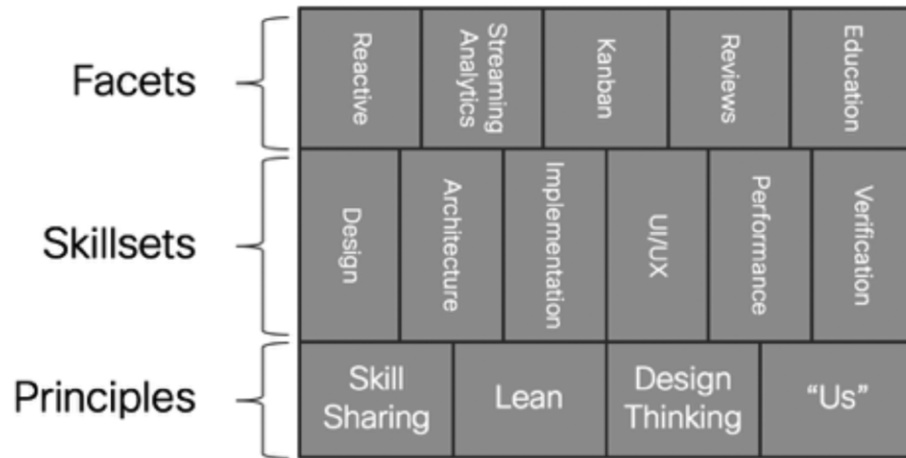
## 2 Lean Continuous Delivery

Before talking about how we test our product, it’s important to understand how our team builds products because this helps explain what kind of outcome we’re hoping to achieve and why we test our product the way we do. Our product development philosophy combines Lean and Agile Development, Design Thinking (Cross 1982, Wikipedia 2016), and other best practices for two main reasons. First, we operate as a commercial endeavor that must be profit optimized. While there are many indirect aspects that affect profit, one place we can directly influence it is avoiding unnecessary effort (e.g., avoid building things people won’t use and avoid releasing buggy code). Second, we want to build features that customers enjoy using. Delightful products generate more revenue and the sooner we achieve this level of refinement, the higher our overall profit will be.

We blend together many aspects of our philosophy when building products (Figure 1). We break these into *principles*, *skillsets*, and *facets*. *Principles* are:

- **Skill sharing:** We believe that skill sharing enables us to grow as a team. If each person on the team is the only person who knows about something, our team is weak. Weak teams have lower overall velocity and the ability to intuitively understand the best trade-offs as we implement new functionality is limited. Additionally, teaching a skill increases personal mastery.
- **Lean:** We believe that the best way to rapidly deliver value to our market is through relentlessly eliminating waste. By eliminating waste, we will be able to improve our velocity, our market accuracy (i.e., delivering the right thing at the right time), and customer satisfaction.
- **Design thinking:** We believe that the best solutions come from empathizing with the consumers of our product, whether they be users, requestors, benefactors, or anyone else our product affects, and that the best approach to this is design thinking. For us, design thinking leads directly to our internal value flow (see Figure 2). Design thinking provides us a view from the consumers’ perspective that directly leads to a higher quality product that people enjoy working with.

- “Us”: “Us” is a catch-all for the entire universe; everything influences our team, product, market, etc. *Everything* includes everything knowable and unknowable. Everything is everything. However, four things in particular have the greatest impact on our team: Our people and their experiences, the history of Agile development, our company culture, and our market.



**Figure 1 - Aspects of our philosophy**

*Skillsets* are the various functional disciplines of our team:

- **Design:** This describes the shape and behavior of our product. This skill set considers which features to implement, which to skip, how those features function (i.e., the features of a feature), and the user experience. While a product's user interface is its veneer, *design* is the overall experience it provides. Put another way, even the most beautiful UI won't fix an ugly design.
- **Architecture:** Architecture is to the insides what design is to the outsides of our product. It is the internal structure of the product. Architectural choices include which technologies and techniques (e.g., reactive) we adopt and how those are woven together and connected to create the framework within which we implement new functionality.
- **Implementation:** This is the actual writing of code, independent of other skill sets. In other words, it is the *how* of implementing a new feature, not the why or what, and writing code that adheres to best practices.
- **UI/UX:** This describes our direct human connection. Again, not all humans interacting with and affected by our product are experiencing it via the web UI. User interface and user experience (UX), like design, considers all users' concerns and all external effects of our product. UI/UX is closely aligned with design and architecture because poorly structured features will lead to worse experiences. UI/UX isn't just graphical user interface (GUI) functionality; it also includes APIs, file formats, and log messages.
- **Performance:** We believe that performance is the computational efficiency of our product. Performance is mostly concerned with raw numbers: how fast, how small, and how many. We want to build products that are enjoyable to use and, when it comes to computers, high performance is expected by humans.
- **Verification:** Verification ensures we deliver what we intended to build. Verification is asking, "Did we build it correctly?" (in contrast with *validation*, which answers "Are we building the right thing?"). Not only does this mean we build bug free (enough) products, but that our products do what we say they do; no more, no less.

Finally, *facets* are the best practices that we integrate into how we engineer our product:

- **Reactive (product resiliency):** We believe the tenets of the Reactive Manifesto (Bonér 2016) enable our design, architecture, and implementation to be an inherently robust product. By maintaining grace under load while simultaneously self-scaling to accommodate spikes, reactive designs are better at handling modern utilization patterns. While initially more complicated to build products in this manner, it ultimately results in less effort, more profit, and happier people (both on our team and in our market). Less effort comes in two forms: less time spent supporting a poorly built product and less time spent re-architecting the product to support higher scale.
- **Streaming analytics (team expertise):** Our team's unique value at our company is our specific focus on and expertise with streaming analytics. We use the best technologies and techniques to enable streaming analytics for our customers so that our team and our products are adept at providing real-time analytic results from massive amounts of data.
- **Kanban (status communication):** For internal status updates, we use kanban (Atlassian 2016) as our task management process because it is lightweight and is well suited for continuous delivery pipelines. Since we are pushing updates to production on frequent and irregular intervals, we need a way to manage tasks in a similar flow. To do this, we use Trello (Trello 2016) to record the status and description of pending, in process, and completed work and report that out at all times.
- **Education (technologies and techniques):** The best products come from highly-trained engineers that leverage the best technologies and techniques to solve our market's problems. We use internal training, vendor support subscriptions, and conferences to keep our skills fresh.
- **Review (ensuring quality):** We consider *quality* to include: Form, Function, and Necessity (i.e., does a feature increase profit?). To ensure quality, we review each change with respect to each skillset: Design, Architecture, Implementation, UI/UX, Performance, and Verification.

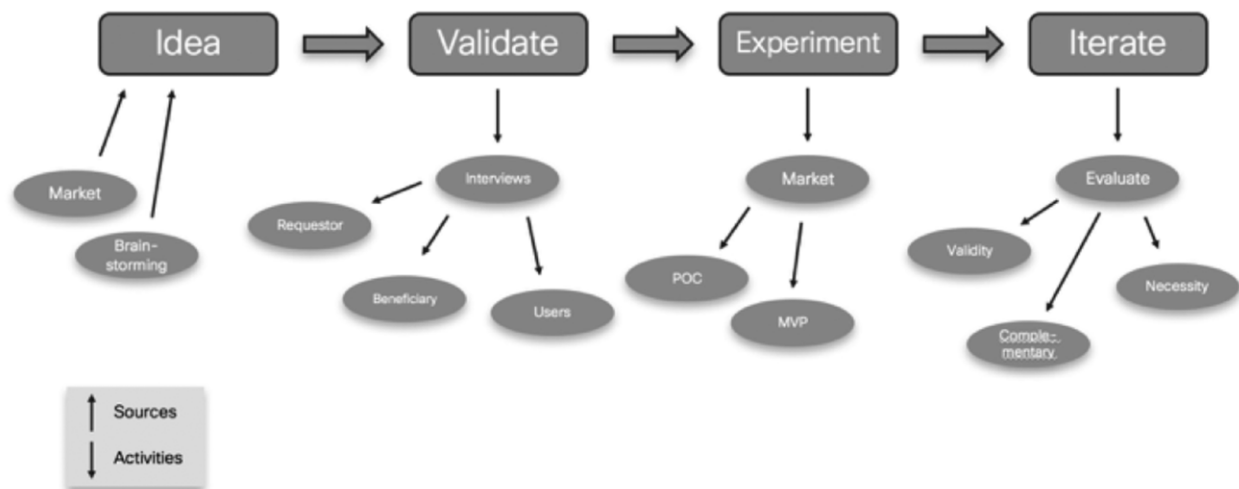


Figure 2 - Internal value flow

Our role is to take an idea and deliver it as value to the market as quickly and profitably as possible. Each step of the internal value flow (Figure 2) maximizes the likelihood that we are working on the right tasks, at the right time, to the right level of refinement, and nothing more.

- **Idea:** An *idea* is a described need. The need comes from the customer (market), but the idea might come from either the customer or internal brainstorming (i.e., ideation). In many cases, the underlying problem may not be understood or even known by the customer. In these cases, we must engage with the market to understand the need so that a precise solution can be built. We will manifest the idea both verbally and visually. Since verbal descriptions are often inadequate, visual engagement with either 2D or, preferably, 3D sketches is necessary. These descriptions are developed jointly with the customer to illustrate the problem needing to be solved, one or more proposed solutions, and the root “whys” of both the problem and its solution. The 2D/3D sketches will also be useful during the interviews in the next stage.

- **Interview:** Interviews are at the core of our process for validating ideas because the development team itself is usually unable to completely assess requirements simply by reading a written request or, worse, inventing it on their own and proceeding directly to implementation. Even purely internal technical tasks need to be scrutinized to ensure they are being built with precision (right scope, right time). The team interviews various personae in the pursuit of understanding the actual problem to be solved. At a minimum, we interview: *users* – the people who directly interact with the feature; *requestors* – the people who asked for the feature; and *beneficiaries* – the people who indirectly benefit from the feature existing.
- **Experiment:** We use the idea sketch and the interviews to identify the next step toward providing a solution. We strive to identify only the absolute minimal progress that must be made so that interviewees can provide feedback on direction. Ideally, this will be 1-2 days of implementation time. This is even less than minimum viable product (MVP). We're targeting a proof of concept (POC) of the next increment. With enough iteration, it will reach MVP. Further iterations, if necessary, will refine out the solution. If the level of effort is beyond 2-3 days, we simplify the criteria instead of spending more time on implementation. Remember, the goal is to spend as little time as possible moving in the wrong direction, which includes delivering a perfect solution too late. The *experiment* stage is where software engineers traditionally spend most of their time, so we want this time to be spent effectively.
- **Iterate:** Once the experiment is ready, we demonstrate it to the interviewees. We compare their feedback to the evaluation criteria to assess if we're moving in the right direction and to ensure that the solution is valid, necessary, and complementary. *Valid* means that the feature's design makes sense for the market. *Necessary* means that the market will pay enough for the feature to satisfy return on investment requirements. *Complementary* means that the feature fits nicely with the overall vision of the product. Lastly, we use the result of the demonstration to determine if another iteration is necessary (i.e., "Are we there yet?").

As you can see, bolting automated QA onto an existing product is insufficient. Though complex, all of these aspects (and others, of course) must be leveraged to efficiently deliver high quality products to the market. Weaving Lean and Agile Development as well as Design Thinking through our philosophy, while layering on relevant disciplines and best practices, allows us to minimize wasted effort. Together, this results in higher velocity, lower cycle times, and shorter test/deploy lead times and provides a foundation for automated QA to deliver maximum value.

### 3 Automated QA

Automating quality assurance (QA) steps for CI/CD pipelines requires correct technology selection as well as adapting QA techniques to the CI/CD environment. We use several testing layers ranging from UI-only testing using mocked data to full end-to-end testing with real or simulated data. Various technology combinations support each testing layer.

The bulk of our UI testing is done using mocked service responses because these tests are easier to write and maintain, execute much more quickly, and provide highly targeted results (i.e., if a test fails, it's easy to know where and why). However, we also use end-to-end testing to test all the service components together for both build verification (quick tests) and covering the greatest number of components (longer running tests).<sup>1</sup>

---

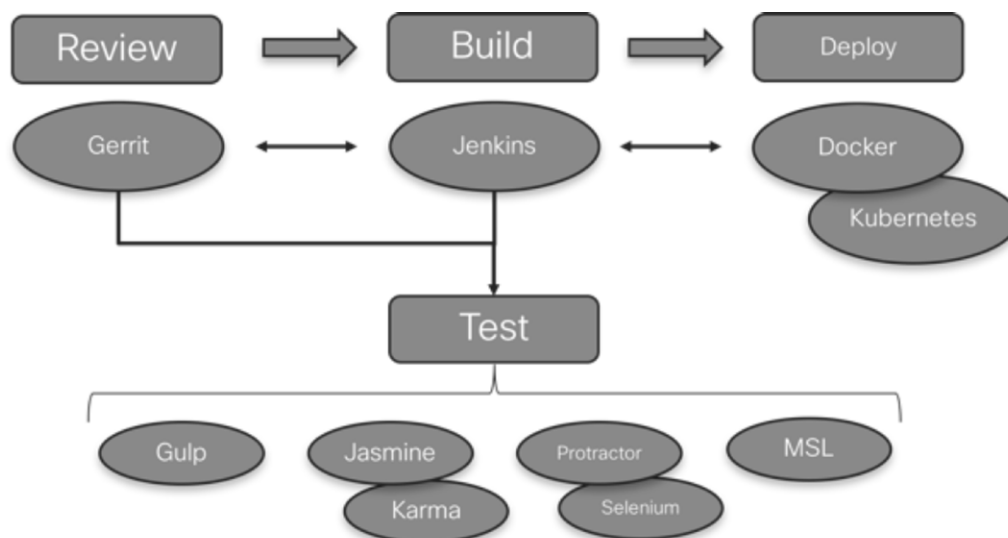
<sup>1</sup> End-to-end testing results in tests that cover the largest number of product components. However, they do not typically result in the highest amount of code coverage unless allowed to run for considerable duration. This lack of coverage is reduced by using fuzzing and long-running tests that exercise edge cases and other infrequent scenarios.

### 3.1 Testing Technologies

Our web UI is built on AngularJS (AngularJS 2016) and Bootstrap (Bootstrap 2016). The UI retrieves data from our servers using REST interfaces that respond with JSON formatted data. This is a very common setup for today's web applications, but guides technology selections because of their specific focus on these aspects of our product. For non-AngularJS/Bootstrap UI stacks, other technologies that accomplish the same function might be preferred (for example, Selenium itself is a very full-featured automated testing tool that works with a wide variety of data and UI technologies).

Our CD pipeline (Figure 3) starts with code review and ends with deployment. Within this pipeline, various processes invoke testing functionality to ensure build viability. There are three main portions of the pipeline where tests run.

- **Code Review:** When a developer submits code for review, Jenkins builds the product to ensure the code does not break the build. Since we use unit tests throughout our product, this build also ensures all unit tests pass.
- **Post-merge:** After the code review is approved, it is submitted through Gerrit and Jenkins builds the product for deployment. If any of the tests fail, the build fails, and the artifacts are not pushed to Docker Registry. So that builds do not take a very long time, these tests are restricted to UI tests using a mocked environment.
- **Pre-deployment:** Before containers are distributed to Docker Registry, a complete set of automated tests ensure that nothing in the product has broken. These pre-deployment tests do a full test deployment of every container including the data simulator and UI testing components using Gulp and Protractor.



**Figure 3 - Build and Test Pipeline (selected technologies)**

With our pipeline, we're able to ensure high quality throughout the process starting even before developers check in their code. This is because their code must pass all unit tests and build verification tests before Gerrit will allow a non-draft submission. However, it's a complicated system that comes with a wide range of technologies and integration complexities. The alternative is slower throughput and reduced CD velocity. In some cases, that's acceptable, but when there's a high-priority customer outage that needs to be fixed in production as soon as possible, high velocity is critical.

Gerrit, Jenkins, and Docker work together in concert to ensure product quality. Gerrit uses Jenkins to build the product, which includes automated testing even before new code can be checked into the main codebase. These tests require components to be built as Docker containers so that they can be executed

in virtual machines as a part of build verification. Since these containers are identical to the artifacts that will eventually be pushed to production, we're testing bit-for-bit identical versions of the code we want to release.

### **3.1.1 Jenkins, Gerrit, and SBT**

We use Jenkins as our build system orchestrator and Gerrit to manage our code reviews (Gerrit 2016). When a developer wants to submit a change for review, Gerrit triggers Jenkins to ensure the change doesn't break the build. The build process includes not just unit tests, but also some quick running automated tests to provide more coverage even before the code review request can be raised. Lastly, since we write all our code in Scala<sup>2</sup>, we use SBT (which stands for Scala Build Tool) (Lightbend 2016) to assemble runtime packages.

### **3.1.2 Docker and Kubernetes**

All of our components run in Docker containers. These include proprietary product components, database servers, and frontend web servers. We use Docker Registry to distribute containers from the build system to internal and external deployments. (Docker 2016)

To control our clustered deployments, we use Kubernetes (Kubernetes 2016) pods and services to manage our component containers. Though not strictly a testing technology, Kubernetes provides automated cluster management so that production-grade automated E2E testing can occur as part of the build process.

### **3.1.3 Data and REST Call Generator**

We created a highly-scalable traffic simulator to generate high load levels for stress testing deployments. It's also used in lower volume configurations to perform verification tests. The traffic simulator generates the proprietary data we receive via our southbound interface (the part of our system that collects data), inserts data directly into our database to simulate large amounts of historical data, and performs queries and UI calls via the northbound (web) interface. In the future, we will also add a web interface to this simulator so that automated end-to-end testing can generate and verify that specific data makes it all the way through the system as expected.

### **3.1.4 Gulp**

We use Gulp (Gulp 2016) to orchestrate our JavaScript build process, which includes executing our automated tests. For us, Gulp functions similarly to Make, Maven, or SBT. However, in addition to just packaging our JavaScript files (e.g., cleanup, removing debug code, minification, and obfuscation), it also executes our automated tests by starting the test tooling and then the tests themselves. We have Gulp start by launching JShint to detect errors and find other problems with our test scripts. Next, it launches the MSL server (see 3.1.7) and kicks off the automated tests.

A nice feature of Gulp is that during the dev/build loop, it watches for changes to source files and automatically updates the browser so that developers can evaluate changes instantly. These files are loaded without going through any minification steps. This makes it easy for developers to identify exactly where in their code the problem is occurring.

---

<sup>2</sup> Our team has used Scala for more than 7 years for multiple reasons: As practitioners of the Reactive Manifesto, we wanted the best access to Akka, Play, Spark, and other libraries that support Scala first; we take full use of its type system; we leverage various functional programming aspects such as object immutability; and we rely on its compile-time checks to ensure we aren't accidentally creating any unsafe runtime code.

### 3.1.5 Jasmine and Karma

Jasmine (Jasmine 2016) provides a framework for writing and running behavior driven development (BDD) and test driven development (TDD) tests for web UI. Supporting both parallel execution and running specific subsets of tests, it supports our commitment to BDD/TDD. We also find it extremely powerful, but easy to use.

Karma (Karma 2016) is a Jasmine test runner. With Karma, developers choose a JavaScript engine to test against (e.g. PhantomJS, Chrome, or Firefox) and Karma runs each Jasmine test and reports the results. Whereas Jasmine is the BDD/TDD test environment, Karma is the tool that actually runs those tests.

### 3.1.6 Protractor

Since we use AngularJS as our web application framework, we use Protractor (Protractor 2016) as the end-to-end (E2E) test framework. Though E2E tests can be difficult to maintain, the benefit of testing an entire E2E dataflow for our application across various test cases is critical. Protractor drives Chrome (Google 2016) running in a virtualized X environment (Ho 2015) to verify web UI functionality and E2E tests. We also tried to use PhantomJS, but it does not support some AngularJS code operations our product requires.

Even though we have optimized the tests and run them in parallel (presently we execute 16 tests at a time), a drawback to this approach is that every test runs via the Selenium server (Selenium 2016) and uses real service components throughout. Since the execution time on these tests can be an issue, we only run a subset of quick-running tests during the build (i.e., compiling/containerizing) process and then longer-running tests in a post-build timeframe (e.g., in a lab soak<sup>3</sup> test that runs for hours or days in parallel with deployments).

### 3.1.7 Mocking Service Layer (MSL)

The Mocking Service Layer (MSL; pronounced “missile”) (MSL 2016) gives us the ability to test the UI without needing a full service deployment behind it. MSL uses the Node.js server to provide a fake web server that provides mocked JSON responses to the UI under test.

MSL, and mocking in general, provides several advantages:

- We can test the UI layer without worrying if a failed test was due to something buried within our application. It gets rid of the databases, containers, virtual machines, networking, and everything else that is required to provide real JSON. However, this also means that if the UI test will cause a legitimate failure in the application, it won't be found until later in the test cycle.
- Because we don't need the service to be running, the UI for a new feature can be tested before the service is ready. We still do end-to-end testing and integration testing to ensure that the use cases supported by the UI are actually supported by the service, but completing these is decoupled from the early UI development.
- Edge cases that are difficult to trigger in the service are trivial to mock. This helps us make sure that rare events function as expected in the UI without needing to set up complex test cases within the service to encounter them.

---

<sup>3</sup> A soak test executes for many hours or days (and sometimes longer) to explore what happens with long uptime durations. Build and deployment oriented testing is essentially ephemeral where the components are only up for a few minutes. This doesn't allow development teams to know with any certainty what will happen when the application is ultimately deployed into production and then left to run there for a long time before components are ultimately restarted.

- Since the service isn't handling the REST calls, the responses coming from MSL are nearly instantaneous. This means that we can complete a testing iteration in much less time, have better coverage<sup>4</sup>, and improve iteration cycle times.

## 3.2 Testing Techniques

### 3.2.1 Unit Testing

Unit testing generally tests only a single piece of functionality in a single way. Multiple unit tests combine to fully test a single piece of functionality. We don't consider anything more than this to be unit testing, but more likely some form of integration testing. We use Gulp, Karma, and Jasmine for our UI unit tests and ScalaTest for our server code.

Unit tests are an important first filter for catching faulty code and, since they execute very quickly, can be run by a developer or a build system on every change. Also, by following the TDD philosophy, failing tests that exercise a bug can be put into place before the fix so that resolution is verified and tracked. Unit tests will always have a place in the test hierarchy, but end-to-end testing and other QA layers are necessary to fully test a product.

### 3.2.2 End-to-end Testing

End-to-end testing (E2E) is the best way to make sure the whole system will work once it's deployed. While unit testing and mocked-data testing ensure that portions of the product work in isolation, E2E is the way we check functionality all the way from data ingest to display.

Our team has developed a scalable data simulator that can provide data to our ingest endpoints as well as hit our REST interfaces to place a deployment under high amounts of load. In automated tests, we don't use real data sources because they are too slow and too difficult to control.

The primary advantage of E2E tests is that they actually exercise the entire product. However, E2E tests have a number of disadvantages:

- Tests are fragile and cumbersome to maintain since test data and application flow is vast and complex.
- Tests take a longer time to run because the entire system must be up and running. There is also the end-to-end lag time to get data through the system because data must actually be processed by real components of the solution.
- If a component (non-E2E) test could have found the fault within a single component that will ultimately result in an E2E test failure, then that component test would save all the E2E build/deploy/test time.
- Requires more investigation as bugs found from the test can come from any part of the system. Developers need to investigate the root cause of test failures as it may not actually be due to the feature being tested.
- Valid changes to datasets or lower-level components can cause test scripts to fail even if they shouldn't, requiring developers to update tests unrelated to their work.

Setting up automated E2E testing is the easy part. Maintaining the tests is an ongoing and perpetual task. However, the payoff is that every release meets quality standards and releases can deploy quite

---

<sup>4</sup> Mocked tests run much faster than full end-to-end tests. Since we want to minimize the time spent testing during the build process itself (i.e., the time actually building product packages), a mocked test environment means we can run more tests in the same amount of time. Therefore, mocking allows us to have greater code coverage at build time than we would otherwise be able to tolerate.

frequently. Additionally, by automating regression tests and tests that exercise edge cases, we can fix those faults before they are detected by customers.

### **3.3 Lessons Learned**

Automated testing at this level of complexity is clearly nontrivial. In our experience, most issues fall into two categories: stale tests and incompatible technologies. Overcoming these issues requires continuous investment and undying commitment to establishing and maintaining the tests and their underlying infrastructure.

#### **3.3.1 Stale Tests**

The well-known problem with maintaining unit tests of occasionally needing to fix a test is magnified many fold with automated UI and E2E tests. Sometimes, seemingly minor changes, or sometimes no explicit change at all result in massive test failures. Changing UI resource trees (which the tests traverse to test UI components), updating a technology that uses a different algorithm for naming or ordering objects, or even reinstalling some deep dependency can cause many tests become invalid and need repair.

Add to this the constant code changes by developers. The automated QA engineer often hears from developers, “Oh, I just changed this one little thing.” Those developers often don’t realize the impact of their changes or consider alternative ways to execute them that won’t cause mass test die-off. Even something as seemingly innocuous as updating a small upstream dependency can change the way things work just enough that tests must be updated.

In the case of E2E test execution, a non-breaking change in one part of the architecture may result in a breaking change in another part of the architecture. Perhaps changing the order of two elements in a list is inconsequential to the software using the list, but it can cause false negatives if the tests are not written with similar order-independent assumptions.

#### **3.3.2 Incompatible Technologies**

Early on, we knew we needed a headless browser to run the UI unit tests so that our build process could incorporate them without needing to start up ephemeral VMs with real (virtual) displays. However, we later found out that the technology we’d chosen (PhantomJS) did not properly execute some of our UI JavaScript (JS). This not only meant the tests failed, but that they could never succeed. As a result, we shifted to using a real Chrome browser running with xvfb (Ho 2015) so that Selenium could drive a solid JS engine.

Consider also that many of our technologies are built around using AngularJS, such as our selection of Protector. This means that any decision to change away from AngularJS also carries with it replacing that portion of our test infrastructure and rewriting all the tests that assume AngularJS is part of our architecture, even if the new component otherwise results in no philosophical change to our architecture (i.e., it behaves the same way, but is implemented slightly differently).

Finally, some components in our architecture are stubborn when it comes to adapting them to the Docker/Kubernetes environment. Part of the reason we wanted to use containers is because of the ease of creating and destroying deployments for the purposes of testing within our build process. However, some components don’t easily adapt to containerization. Similarly, when we add new components to our architecture, we must also consider their applicability to our environment and our automated QA infrastructure lest our entire product become untestable.

### 3.4 Example Test Script

This test script shows how Jasmine, Protractor, and MSL all work together to test product functionality. In this test, we're logging in and testing that some data exists on the page that loads immediately afterward. The test case steps are found as comments in the `describe` block.

```
// Requires the MSL client API
var msl = require('msl-client');

console.log("Executing Mock page....");
describe("Mock Signin", function () {
    «use strict»;
    var subscriberCount = element.all(
        by.css('div[class*="table-responsive"
tr[pagination-id="search-results"]')
    );

    describe("Mocking UI Test", function () {
        /* Step-1 Go to Signin page
        * Step-2 Enter user name
        * Step-3 Enter password
        * Step-4 Click submit button
        * Verify 10 subscribers exist in first page */

        var mockResponseObj = {};
        mockResponseObj.requestPath = "/api/authenticate";
        mockResponseObj.responseText = [{"username":"tester","password":"testme"}];
        mockResponseObj.contentType = "application/json";
        mockResponseObj.statusCode = 200;
        mockResponseObj.delayTime = 0;
        mockResponseObj.header = {"X-App-Token" : "r4nd0mT3x7"};

        it("should sign in", function () {
            // signin mock response
            msl.setMockRespond("localhost", 8000, mockResponseObj);

            // open a browser and go to sign-in page
            browser.get('http://localhost:8000');

            // sign in
            element(by.id('username')).sendKeys('user');
            element(by.id('password')).sendKeys('password');
            element(by.css('form[ng-submit=signInCtrl.submit()] button')).click();
        });

        // make sure it has what it should have immediately after signing in
        it("should have 10 subscribers in first page", function () {
            expect(subscriberCount.count()).toBe(10);
        });
    });
});
```

## 4 Conclusion and Next Steps

Our customers demand the time-to-fix responsiveness that CI/CD can provide, but we can only provide high quality updates through automated testing that's integrated into our pipeline. Building high quality and high value modern web applications capable of high reliability is a complex endeavor where just the automated testing infrastructure and individual tests is complicated. However, it's a necessary step in continuously delivering product improvements to our customers and, without it, we wouldn't be able to maintain the responsiveness they demand.

Now that we have the essential portions of our automated QA in place, we're adding to our system in two ways. First, there are always more tests to add for both new features and to improve code coverage of existing features. When we say "more tests," we don't mean just more of the same. The high-performance simulator that's a key part of our pipeline can only generate random data. This makes E2E testing that evaluates the accuracy of the data presented in the UI impossible. We know we have some data, but not necessarily the right data. Furthermore, our simulator has the ability to put a deployment under high load, but we don't have any coordinated UI or E2E tests against such a deployment to confirm the system responds as expected under that load.

Second, other types of testing that we can take advantage of within our existing pipeline are integration tests and soak tests. For integration tests, we could, for example, test the database-to-frontend interface in isolation. Presently, this interface is only exercised during E2E testing and a failure will only be caught if it causes a UI-layer failure. Relying on E2E testing also precludes any fuzzing or other intensive interface-targeted testing. With soak tests, we could put the system under high load and observe it over several hours, days, or weeks. While this obviously wouldn't be a part of the CI/CD testing, it still relies on automated testing so that the data feeds and frontend REST calls are realistic.

We're continuously learning how to improve our development methodology to ensure that we deliver the highest quality and value to our customers. Applying these learnings to our pipeline should be relatively efficient since the framework is flexible and highly capable.

## 5 References

AngularJS. "Angular JS - superheroic JavaScript MVW framework." <https://angularjs.org/> (accessed July 5, 2016).

Atlassian. "A brief introduction to kanban." <https://www.atlassian.com/agile/kanban> (accessed July 5, 2016).

Bonér, Jonas; Farley, Dave; Kuhn, Roland; Thompson, Martin. "Reactive manifesto." <http://www.reactivemanoifesto.org/> (accessed July 5, 2016).

Bootstrap. "Bootstrap - The world's most popular mobile-first and responsive front-end framework." <http://getbootstrap.com/> (accessed July 5, 2016).

Cross, Nigel. 1982. "Designerly ways of knowing." *Design Studies* 3.4: 221-27.

Docker, Inc. "Docker - Build, Ship, and Run Any App, Anywhere." <https://www.docker.com/> (accessed July 5, 2016).

Gerrit. "Gerrit code review." <https://www.gerritcodereview.com/> (accessed July 5, 2016).

Google, Inc. "Chrome." <https://www.google.com/chrome/> (accessed July 5, 2016).

Gulp. "Gulp" <https://www.npmjs.com/package/gulp> (accessed July 5, 2016).

Ho, Toby. January 9, 2015. "Headless browser testing with xvfb." <http://tobyho.com/2015/01/09/headless-browser-testing-xvfb/> (accessed July 5, 2016).

Jasmine. "Jasmine: Behavior driven JavaScript." <http://jasmine.github.io/> (accessed July 5, 2016).

Karma. "Karma - Spectacular test runner for JavaScript." <https://karma-runner.github.io/1.0/index.html> (accessed July 5, 2016).

Kubernetes. "Kubernetes - Production-grade container orchestration." <http://kubernetes.io/> (accessed July 5, 2016).

Lightbend, Inc. "sbt - The interactive build tool." <http://www.scala-sbt.org/> (accessed July 5, 2016).

MSL. "Mock service layer." <http://finraos.github.io/MSL/> (accessed July 5, 2016).

Protractor. "Protractor - end to end testing for AngularJS." <http://www.protractortest.org/> (accessed July 5, 2016).

Selenium. "Selenium - Web browser automation." <http://www.seleniumhq.org/> (accessed July 5, 2016).

Trello. "Trello tour." <https://trello.com/tour> (accessed July 5, 2016).

Wikipedia. "Design thinking." [https://en.wikipedia.org/wiki/Design\\_thinking](https://en.wikipedia.org/wiki/Design_thinking) (accessed July 5, 2016).



# Agile Testing Without Automation

John Duarte, Eric Thompson

john.duarte@puppet.com, erict@puppet.com

## Abstract

Most research on Agile Testing and QA have requirements on highly automated testing and an Agile or Scrum project-management structure.

How can we iterate towards a more Agile testing process, with all the benefits that entails, when some of the common requirements are missing or undesirable in the near-term?

Drive quality as a core principal through communication and collaboration with QA test “consultants” embedded with development teams: We discuss real-world results from Puppet on feature turn-around time and relevant defect rates

Derive transparency on upcoming features and requirements through quality user stories and acceptance criteria, BDD or otherwise

Risk analysis driven by all teams, particularly Product, QA and Development: We discuss the importance of risk-relative testing, and why it’s important in some cases to test less, rather than more.

Whatever is prioritized to test should be transparent to all groups and stakeholders: Everyone knows communication is important, but how can we agree upon the most effective details that need constant discussion with effortless transfer? We discuss methods and motivation on communication: Instances of success and opportunities for improvement at Puppet are presented.

## Biography

*John Duarte is a QA Engineer at Puppet in Portland, Oregon.*

*Eric Thompson has been an Electrical Engineer, Software Engineer, and Quality Engineer for 15 years, in organizations of all sizes, leading teams of engineers with varying degrees of expertise and backgrounds.*

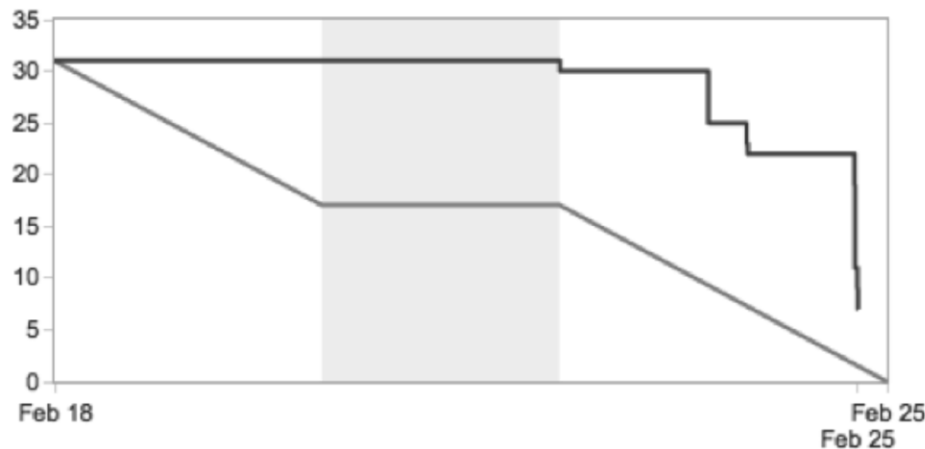
# 1 Introduction

In the fall of 2014 at Puppet, our QA organization was struggling to transition from waterfall to Agile testing practices. Engineering was practicing Scrum, but QA's involvement was sandwiched at either end of the development cycle. At the beginning of the cycle we were struggling to gather project requirements in order to write test plans. At the end of the cycle, we were frantically trying to validate functionality after it had been implemented. This type of practice is often referred to as *agilefall*.

In the winter of 2014, Puppet adopted an embedded model where each Scrum team had assigned QA resources. This model went a long way toward getting QA out from under the agilefall, but it posed new challenges.

The time to execute went way up for two primary reasons: QA resources were spread thinner by the embedded model and the Scrum teams assumed it was the duty of QA to test *All The Things*. Tickets needed QA approval to close and were being passed to QA late in the sprint without enough time for proper validation. This meant that a high number of committed story points rolled over, which created an undesirable trend in the burn-down chart.

## 1.1 Poor Burn-Down



# 2 Risk Driven QA

The purpose of Agile methodologies is to provide value to the customer as quickly as possible. The burn-down metrics were telling us that we were not delivering on that goal.

QA at Puppet needed to re-assess how we could re-align ourselves with the customer goal of delivering valuable features in a timely manner.

## 2.1 Analysis

In order to increase velocity without sacrificing quality, QA adopted a risk analysis driven approach to testing.

All tickets are evaluated for the risk that they pose to the product and the customer. We prefer to perform this analysis collaboratively with the developers during the sprint planning meetings. This process also forces the team to make sure that acceptance criteria are clearly defined, since the assessment cannot be performed without it. This analysis is transparently communicated to the stakeholders and their input is brought to bear in the final assessment.

Each ticket is evaluated based on the severity and probability of the risk that it poses. These assessments are expressed as simple *High*, *Medium*, or *Low* values. Reasons for these selections are added to provide the stakeholders context as to why a particular assessment was given.

The risk values for the severity and probability are then combined into an overall risk assessment.

### 2.1.1 Overall Risk Matrix

The following matrix provides a guideline for how the two risk vectors are combined at Puppet. It is only a guide and there are occasions where additional information will allow a QA professional to apply a diverging combined value. The matrix accomplishes several goals:

1. It provides a consistent heuristic for all of the QA personnel to apply when calculating the overall risk of an issue.
2. Establishes a clear protocol that allow QA and development professionals to quickly apply when discussing concerns about an issue.
3. Encourages documenting a combined value that deviates from the matrix. Although this is not common, there are certainly occasions where the a value will deviate from the guidelines. For example, a security vulnerability may pose a severity level that is so high that it commands the highest level of risk even if its probability is low.

	Low Probability	Medium Probability	High Probability
Low Severity	Low Risk	Low Risk	Medium Risk
Medium Severity	Low Risk	Medium Risk	High Risk
High Severity	Medium Risk	High Risk	High Risk

## 2.2 QA Engagement Protocol

The value of the overall risk assessment translates into a clear engagement strategy by QA:

### High

High risk tickets must have automated tests associated with them in order to validate them against the broadest compatibility matrix and to protect against future regression.

### Medium

Medium risk tickets will receive manual validation on a sub-set of platforms. Automated regression tests are not expected.

### Low


Low risk tickets receive no further QA attention.

In all cases, Developers are responsible for providing adequate unit tests.

## 2.3 Transparency

All of the risk assessment data is captured in each ticket in our issue ticketing system. We have placed it in a QA tab so that it is easy to find, but is out of the way when not needed. This transparency allows stakeholders to easily infer the level of QA commitment with regards to ticket validation. Any disagreement or lack of clarity can easily be discussed within the comments on the ticket.

### 2.3.1 Custom QA Fields in our ticketing system

 Factor / FACT-1056

Operatingsystem major release not resolved in OSX

Edit

Comment

Assign

More -

Open

Needs Information

Workflow -

Details

Type:Bug

Priority:Normal

Affects:FACT 3.0.0

Version/s:

Component/s:None

Labels:None

Status:CLOSED (View Work)

Resolution:Fixed

Fix Version/s:FACT 3.0.2

Field Tab

Scope Change

QA

QA Status:Reviewed

QA Risk:Medium

Assessment:

QA Risk:Medium

Severity:

QA Risk Severity:higher severity on osx where the OS \*has\* a version

Reason:

QA Risk:Medium

Probability:

QA Highest Test:Acceptance

Level:

## 3 Results

### 3.1 Less Automation

Since the QA engagement protocol for the risk matrix only mandates that automation be applied to testing of *High* risk tickets, we are writing fewer automated tests. This has several benefits:

Our people have a more robust understanding of what product risks are covered by unit tests and do not need higher level testing.

The automated tests we write focus on features and regression candidates that have the highest value to the customer.

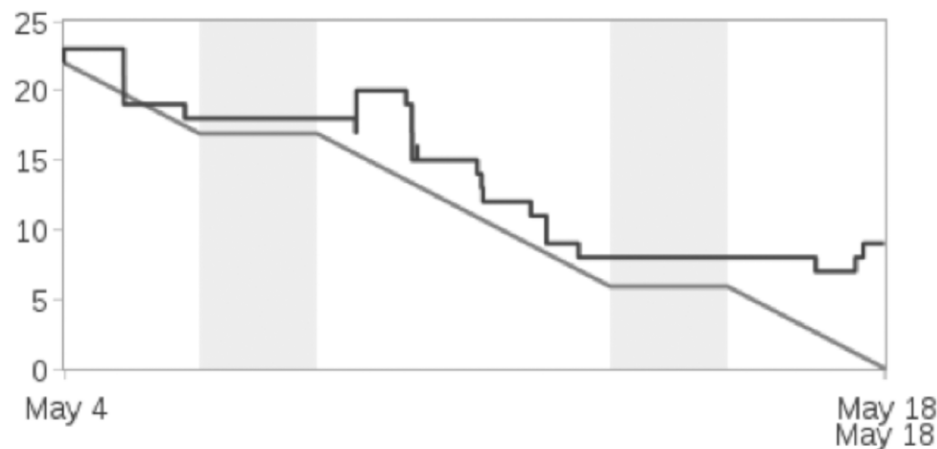
The cost of our automated test suites is not growing at the same rate as our product code bases.

## 3.2 Relationship

This work-flow has created a healthy relationship with stakeholders, especially developers. QA will openly discuss the validation required for an issue with developers. Clear acceptance criteria are defined. Based on these criteria, it may be determined that the issue is adequately covered by the unit tests written by the engineers. This means that even if a ticket has been risk assessed as *High*, the unit coverage may remove the need for QA to write more automation.

Teams have made this an explicit part of their process which has been very successful. Reviewing tests with developers and determining at which level each test should be automated definitely helped avoid duplication of effort, and ensured that we were getting sufficient coverage of the most important scenarios.

Developers have organized their work to deliver code to QA earlier in the sprint so that both parties can



continuously move tickets forward. In the event that QA cannot validate all the tickets by end of sprint then the developers will assist in ticket validation. This has resulted in better burn-down trends in sprints and more predictable velocities for the Scrum teams over time.

### 3.2.1 Improved Burn-Down rate

## 3.3 Focus on Higher Value

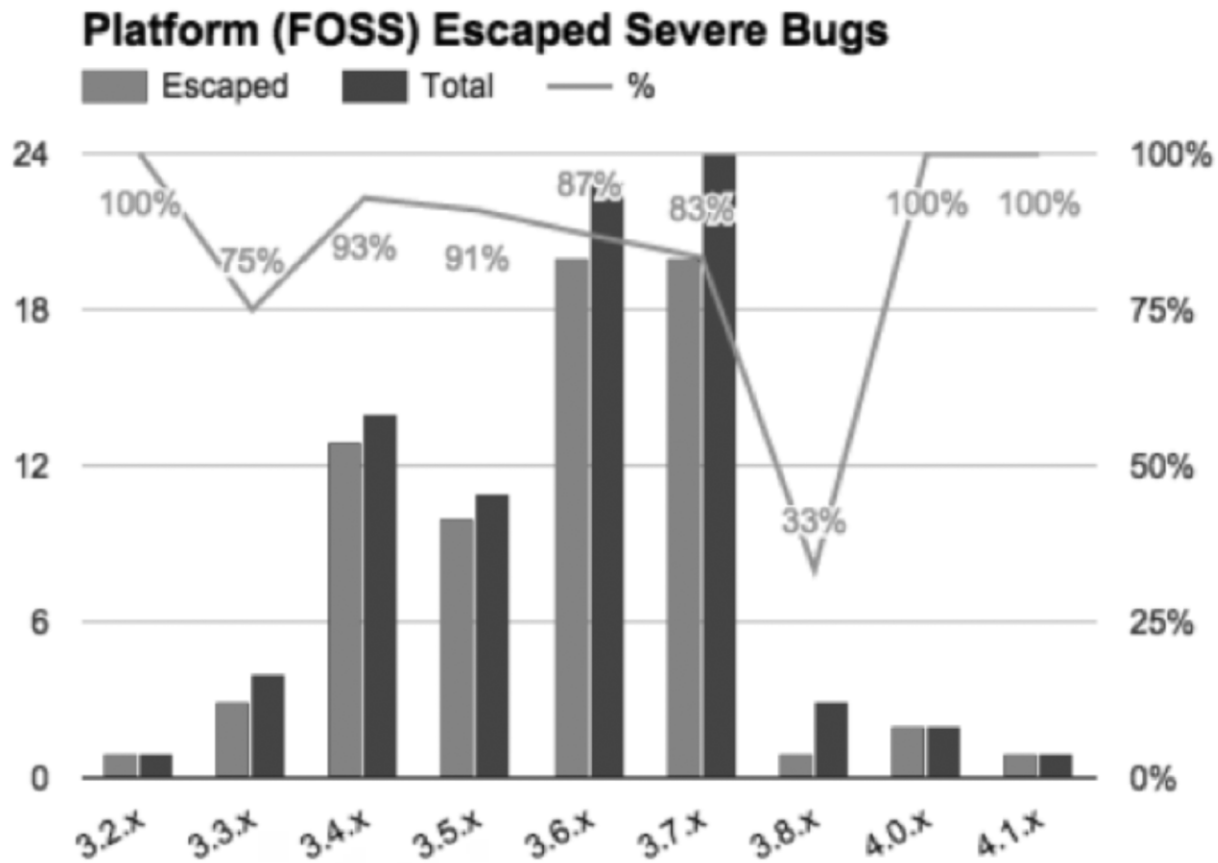
The risk driven method has allowed QA to focus on high value efforts.

QA collaborates on user stories for new features. QA works with product owners and developers in order to capture relevant user stories and define the expectations for error conditions.

Each ticket receives acceptance criteria to minimize ambiguity. Having all of this information centered in the ticketing system allows all of the stakeholders to contribute.

Using a risk driven approach has also resulted in a dramatic reduction in our defect rate when measured against severe and critical issues that have made their way into shipped product.

### 3.3.1 Defect rate



## 4 Conclusion

Applying a risk driven approach to testing will allow you to focus on the issues that have the greatest impact on your product and its users. Doing so in a way that is transparent to the organization, and fosters the input of all the stakeholders, throughout the process will result in a better product. It will also result in better relationships between QA and the other groups within the organization.

# Logic Programming to Generate Complex and Meaningful Test Data

Donald Maffly

dmaffly@huronconsultinggroup.com

## Abstract

Even in today's modern software engineering world, there still exist obstacles that confound testers' ability to generate test data—data that is complex and precise but also that can be generated in mass quantity quickly.

In data warehouse applications, which typically contain copious historical data, it is very cumbersome and time consuming to generate data through the 'front door' of these systems via import files and simulated user interaction. This approach can also have the shortcoming of not easily allowing application testers to land data on precise points called for in functional testing.

Furthermore, third party data generation tools, while excelling at generating copious data, generally don't provide application testers customization hooks that are powerful and expressive enough to generate meaningful data — particularly data where relationships are complex and where understanding of the data requires application knowledge (e.g. business logic) not found in a database schema.

To address this problem, we have taken an approach using constraint logic programming; it is loosely inspired by prior work done in the area of Reverse Query Processing (RQP). The general idea is that if a tester can write a SQL SELECT query for the test data s/he targets, then using RQP the tester has the tool to generate data sets that fit that query.

## Biography

*Donald Maffly has over 25 years of experience in software development, not only as a tester, but also as a developer and manager; over the years, he has worked on a diverse range of technologies and products, ranging from those serving the financial, healthcare, and R&D communities. Currently at Huron Consulting Group for the past 5 years, Donald has focused entirely on a Software Quality Assurance role in the Business Intelligence and Data Analytics areas.*

# 1 Introduction

If system architecture is the “bones” of a system, the software code is the “flesh” of a system, then data represents the “life blood” of a system. Data constantly flows through a system and it is constantly changing. It follows then that in the endeavor of testing the system, you need diverse means of getting vast and varied data into the system through all of its various inputs.

While there exist many ways to obtain test data, all of which can fill valid purposes, we found that there was something lacking in existing approaches that allowed us to practically, quickly get test data into the system to functionally test it.

We addressed this void by creating a tool that injects data directly into a database back end; it is loosely modeled after Reverse Query Processing (RQP) [BINNIG 07]. The idea is that if you can pinpoint the test data you need in the form a SQL query, then with an RQP tool you have the capability to generate data that satisfies that query.

We describe our system context and some of the challenges that we faced using existing approaches to obtaining test data. This sets the stage for deriving requirements for our tool, the design of the tool, and ultimately how it is used to generate test data.

## 2 Huron Consulting Group's Data Generation Problem

To provide some context into the data generation problem we are confronted with at Huron Consulting Group, we describe the essentials of the system under test. The Aeos® application is a collaborative work flow system that supports the Huron process improvement methodologies for healthcare providers with an Online Analytical Processing (OLAP) back end database. The primary goal of the Aeos system is to get *more* patient-based revenue into the books *sooner*. Aeos software has two primary input points: (1) hospital-specific account/patient/billing information in import files from the client Hospital Information System (HIS) and (2) the Aeos application that hospital representatives use to work items to completion.

Data reaches the front end of the system as concrete entities such as account numbers, patient names, balances, and dates. By the time data reaches the data warehouse back end, the dimension of tracking change is added. A further data refinement occurs next when data warehouse Extract-Transform-Load (ETL) processes transform this data into more abstract forms such as counts and balances relative to dates. Finally, the end user sees the highly distilled final product as metric calculations in the form of data tables, charts and graphs. Business analysts and managers use these to assist in day-to-day management as well as to help make longer range strategic business decisions.

A hypothetical example of such a metric that we work through later in this document is “First Day Touched Done Percent”. This metric is the percentage of work items that are in the *done* state and that were edited or modified by a user on the first day it was assigned. The simple sounding name of this metric belies the true complexity of it. The data supporting it is quite complex, requiring conditional joins across many tables combined with additional filtering criteria.

Examples of data constructs that support the computation of these metrics are:

- Two-way linked lists. Our back end data warehouse contains many fact tables that contain self-referential two-way linked lists. These can be traversed forward or backward to track change of state. To further complicate things, these tables contain two sets of two-way linked lists: one to track same day change and another to track all-time change.
- Multiple date fields exist on multiple tables that semantically relate, but in ways that are not derivable from examination of the database schema alone.

Complications like these present different challenges depending on the approach we might choose to obtain our test data.

### 3 Shortfalls of existing data generation approaches

In this section, we summarize the approaches that we have considered to obtain complex test data. Each approach has its strengths and weaknesses, but, in the end, we concluded that they all fell short of our complex test data need. From here, we were able to describe precisely what our need was in the form of a concise list of requirements.

#### 3.1 Where data harvested from production systems falls short

Data exists in bountiful quantity on production systems. Such data has the advantage that it is 'real' and it contains customized client specific properties that make the data truly unique. It is nearly impossible to mimic this production data in our test environments. For this reason, testing on client customized UAT systems is an indispensable part of our testing strategy. But it is not *central* to our testing strategy because working with this data comes with a few major obstructions.

- 1) Production data contains Personally Identifiable Information (PII). PII contains highly sensitive data such as social security numbers, account numbers, and other private information. In the health care arena, for instance, the Health Insurance Portability and Accountability Act (HIPAA) provides strict guidelines designed to protect this data (also referred to as Personal Health Information or PHI) and the privacy of individuals. Healthcare institutions can get fined for violating these rules. In other arenas, such as ecommerce, reputations of companies are at stake if PII is mishandled in any way. As a result, our clients (health care institutions) are generally reluctant to let their data migrate outside their network. So in cases where we do need production data to test with, we have no other recourse other than to test on client test systems over which we wield little control. We typically need to share these systems with other users (e.g., administrators, testers, business analysts), we are unable to install special software on them, and we are prohibited from performing 'destructive' type testing. This is far from an optimal environment in which to conduct testing.
- 2) Production data is *not* organized in a way that maps easily to our test cases; therefore it is nearly impossible to execute our entire test suite out on a client UAT system to validate it completely.
- 3) Production data is large and unwieldy. Consequently, it can lengthen testing cycles by an order of magnitude or more, and it is challenging to debug and diagnosis problems.
- 4) Production data doesn't necessarily land on all data points that are critical to functionally test the system in entirety.

#### 3.2 Where conventionally loaded test data falls short

At Huron Consulting Group, we test on internal systems that contain a single canonical configuration that is intended to represent the many configurations that our clients have on their production systems. This canonical configuration captures most of clients' specifics but surely not all. We use a custom built in-house data generation tool that does a reasonably good job of generating production-like data into our system for testing purposes. This tool generates data the same way it is done in production — via import files as well as by mimicking end-user activity via the application. While the data that results is very realistic, it comes with a couple of drawbacks:

- (1) The abstractions and language of our import files are very different than the abstractions that have been processed and land on the back end of our system in the data warehouse. This makes it challenging to script test cases using import file vocabulary that is targeting BI functionality expressed in an entirely different vocabulary. Thus, there is a cumbersome translation the tester needs to go through to script test cases with front end abstractions targeting back end functionality. If a test case requires user activity, it is 'hit or miss' using our in-house tool since it is not able to coordinate simulated end user activities on particular imported data. Ultimately we don't have the highest confidence that data is landing on precise points called out in our test cases.

- (2) We need a lot of generated historical data to adequately test BI functionality since it delivers trend and weekly average information to the end user. Because our in-house tool imports data via conventional channels, it necessarily depends on the system clock to import data. Loading data this way is tedious and can take several days or longer.

Using our in-house tool to create test data in the back end data warehouse is akin to spraying a machine gun in a general direction to hit a target across a vast expanse. Not only are the bullets likely to miss precise targets, but it also takes bullets quite a while to arrive. At first blush, it would appear to be easier to shoot the target from point blank range. But that too, is not as easy as it would seem.

### **3.3 Where direct SQL data insertion falls short**

When a front end data loading approach cannot meet our needs, an obvious fallback choice would be to code SQL INSERT statements to populate the data directly into the Aeos system's data warehouse back end, that is to say, shoot the target from point blank range. It turns out this SQL direct approach has proven quite useful in instances where it is sufficient to populate single tables (e.g. summary tables) in isolation of greater database context. However, this SQL direct approach falls short when creating complex networks of rows of data across many tables, if for no other reason than it is very complicated. After all, in live contexts, it requires a middle computing tier containing complex application logic to construct this data.

Furthermore, using a SQL direct approach, the test scripter loses greater relational context when he has to break his code into separate sections to perform SQL inserts across different tables. This has the effect of making the code more challenging to write, as well as to read and maintain. The approach also offers no obvious silver bullet to solving our problem of creating the two-way linked lists within single tables.

### **3.4 Where third-party data generation tools fall short**

We searched for commercial off-the-shelf (COTS) solutions that could assist in injecting data directly into our BI database. We were sorely disappointed.

There exist numerous test data generation tools on the market that are all capable of generating copious amounts of test data. These tools provide end users with all kinds of tricks for generating data — randomly populated fields, sequentially populated fields, drawing from finite user defined domains and even offering end users the ultimate power of using SQL and Python hooks to generate column values.

While the resulting data generally suffices for certain types of testing (e.g. performance testing, testing UI front ends, etc.), it falls short for functionally testing BI data warehouses containing highly complex and abstract data. Using such tools it is very difficult to get data to land in cracks and crevices in a data profile that are meaningful as far as ETLs and metrics are concerned. If data happens to land in these meaningful places, it does so accidentally, not by design, even with the use of these tools' customization hooks.

We were able to find one or two data generation tools that could scratch the surface of generating meaningful data, but they do so in a clumsy non-intuitive manner. The best tool(s) allow users to script custom data generation from the perspective of a single table column, but not from the greater picture that a SQL query would provide, for instance, where relations between tables and constraints are immediately obvious to the coder or anyone else needing to understand the code. Using these tools, custom generated code is broken up into bits and pieces on a table column basis, and the greater context of the intended semantic can be lost.

### 3.5 Requirements for generating complex test data

From this problem exploration, we were able to derive the following requirements needed in a data generation tool:

- 1) Generate 'smart' data that is business application aware. Like COTS, the database schema can be leveraged. Unlike COTS, better customization hooks are needed to capture higher level business application logic.
- 2) Generate data that targets precise points in the data warehouse. These precise data points may have complex inter-table dependencies.
- 3) Generate data using customization hooks that are not myopically confined to just column value generation, but that allow the scripter greater vision of broader relational context in a manner that might not be all that different from the SQL language.
- 4) Generate historic data relatively quickly (at least an order of magnitude faster than our in-house data generation tool).

## 4 From Requirements to Solution

Given the limitations with commercial off-the-shelf tools, we fell back on our in-house generation tool despite its long and tedious data generation process. Afterward, we'd validate that we had the data we needed before initiating the actual testing. It is worth noting that the intention of these data validation scripts was not to test BI functionality of our product per se, but rather to see if the data was present to perform a given test.

We came up with a few hundred such scripts, each mapping to a particular test scenario targeting ETLs, all written in SQL. Even though each of these SQL queries generally spanned a handful of large tables (some with over 100 columns) in our data warehouse, each query was concerned with only 10 to 15 total column values across several tables, mirroring the ETL computation for which it was intended to validate test data. It was as if the remaining columns were invisible as far as the ETL was concerned. So any solution to generate test data for them would not need to concern itself with populating *complete* data, but rather *minimally sufficient* data. When approached this way, the test data generation problem becomes less daunting.

It occurred to us sometime during the laborious process of writing these SQL validation scripts that it would be much easier if we could simply directly generate the test data that these SQL scripts were querying. If only there existed a tool that could perform 'reverse-sql', as it were — not a SQL SELECT, but the opposite of it. It turns out that there has been research done in this area [BINNIG 07]. In their paper, Reverse Query Processing (RQP) the authors cite as follows:

*"Reverse Query Processing (RQP) gets a query and a result as input and returns a possible database instance that could have produced that result for the query. [...] There are several applications for RQP; most notably, testing database applications [...]"*

This work never made it farther than the research phase, and no commercially available tool ever resulted from this work that we know about. We considered implementing RQP based on [BINNIG 07], but felt it would be difficult to translate into a practical tool. Our work presents a Constraint Logic Programming (CLP) based alternative to [BINNIG 07] which demonstrates RQP's usefulness in commercial application.

## 4.1 Prolog as a platform for generating data

Prolog [KOWALSKI 74], [COLMERAUER 93] promised to be a suitable platform from which we could solve our data generation problem. Like SQL itself, Prolog is a declarative language, and it is well suited for expressing relationships between objects. Furthermore, the *list* is a data structure that is central to the Prolog language, Prolog would likely assist in the generation of Aeos software's complex table data containing two-way linked lists. Finally, Prolog's logic programming and rules capability makes it very powerful at generating *many* solutions to a given problem. If we could only figure out how to express our problem in a data generation-like vocabulary, a Prolog solution to the problem could be the relational data we sought.

For the sake of illustration, we walk you through how Prolog can be used to solve a simple logic problem, and then we extend this problem solving model to the greater problem of data generation.

### 4.1.1 Solving a simple logic problem with Prolog

There is a logic problem in [SUMMERS 72] called "Card Number Six". Given a grid of 8 squares (Figure 1), you need to distribute two aces, two kings, two queens, and two jacks on the grid such that:

- Two cards with the same denomination (e.g. 2 aces) may not border each other
- Each ace borders a king
- Each king borders a queen
- Each queen borders a jack
- No queen borders an ace

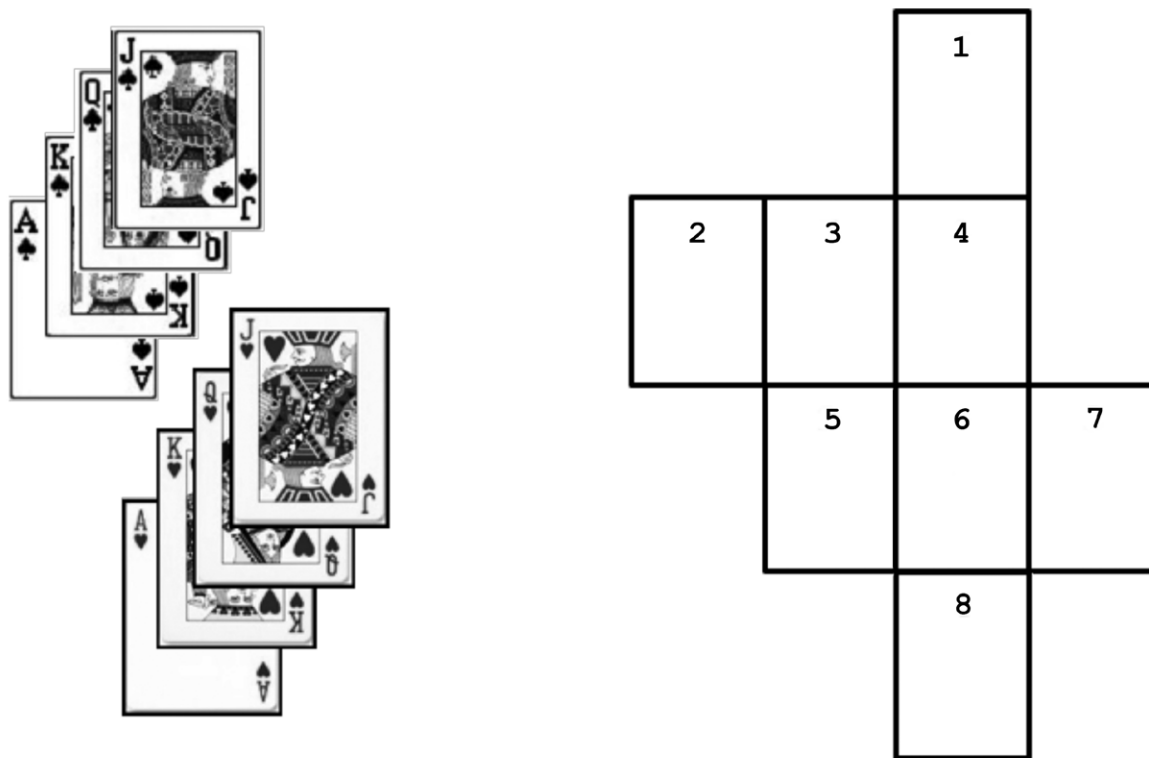


Figure 1

Think about how you might solve this in the most brute force way possible if you had limitless brain power at your disposal. Perhaps the simplest strategy would be to take all the combinations of the 8 cards across the grid and test each combination against our criteria listed above and see if it passes muster.

In Prolog, let's begin with our primary data structure, an ordered list of eight cards. The card in the first position of this list indicates that the card is on the first square on the grid, in the 2<sup>nd</sup> position, on the 2<sup>nd</sup> square, etc.:

```
CardList = [a1,a2,k1,k2,q1,q2,j1,j2]
```

Here, a1 denotes Ace One (i.e. Ace of Spades), a2 denotes Ace Two (i.e. Ace of Hearts), etc.

There is a built-in predicate in Prolog that is able to return all the permutations of this list:

```
permutation(CardList, PermutedList)
```

To run each permutation against our above listed criteria, we create some building blocks. First we code some Prolog facts that describe whether squares in the grid border each other or not:

```
borders(1,[4]).
borders(2,[3]).
...
borders(6,[4,5,7,8]).
```

...then two predicates `is_next_to` and `is_not_next_to` (whose body/definitions are not shown here) built on top of our **borders** facts from which we can express our filtering criteria:

```
is_next_to(a1,[k1,k2], PermutedList),
is_next_to(a2,[k1,k2], PermutedList),
is_next_to(k1,[q1,q2], PermutedList),
is_next_to(k2,[q1,q2], PermutedList),
is_next_to(q1,[j1,j2], PermutedList),
is_next_to(q2,[j1,j2], PermutedList),
is_not_next_to(q1,[a1,a2], PermutedList),
is_not_next_to(q2,[a1,a2], PermutedList),
is_not_next_to(a1,[a2], PermutedList),
is_not_next_to(a2,[a1], PermutedList),
is_not_next_to(k1,[k2], PermutedList),
is_not_next_to(k2,[k1], PermutedList),
is_not_next_to(q1,[q2], PermutedList),
is_not_next_to(q2,[q1], PermutedList),
is_not_next_to(j1,[j2], PermutedList),
is_not_next_to(j2,[j1], PermutedList).
```

Prolog returns not just the cards that could land on square number 6, but *all* the squares, as follows:

```
X = [k1, q1, j1, q2, a1, k2, j2, a2];
X = [k1, q1, j1, q2, a1, k2, a2, j2];
X = [k1, q1, j1, q2, a2, k2, j2, a1];
X = [k1, q1, j1, q2, a2, k2, a1, j2];
...
X = [k2, q2, j2, q1, a1, k1, j1, a2];
X = [k2, q2, j2, q1, a1, k1, a2, j1];
X = [k2, q2, j2, q1, a2, k1, j1, a1];
X = [k2, q2, j2, q1, a2, k1, a1, j1].
```

Not all solutions are listed here. For the sake of brevity, only 8 out of the 32 possible solutions are listed. Note that only a king can land in the 6th position, i.e., either the King of Hearts or the King of Spades.

#### 4.1.2 From 'Card Number Six' to data generation

It is not a huge leap to extend our approach to solving the "Card Number Six" problem to the arena of data generation with a few minor tweaks. Imagine instead that the grid in our card problem was a row of data in a table, each square being a column value. Then imagine that only one of four queens (heart, spade, club, and diamond) were allowed to land on square one, kings on square two, etc. The programmer could modify the criteria to ones that were appropriate for a particular test case; e.g. produce all the combinations where only cards with denomination of hearts are contained in the grid, or the row of data as it were.

Our data generation tool follows this same approach to generate complex data scenarios:

- 1) Create permutations of table row data
- 2) Throw out the meaningless permutations by evaluating each against test specific criteria

Recall in our "Card Number Six" approach, we didn't employ a particularly smart and efficient problem solving strategy, but it's worth noting that the complexity of the problem and the small size of the search space did not necessitate it. The initial set of permutations numbered 40,320 possibilities, large for the human mind to grapple with, but quite manageable for your average laptop computer to handle. In the case of more complex problems with larger search spaces, such as the data generation problem, search spaces can grow several orders of magnitude larger and can easily exhaust the memory and computing resources of a machine.

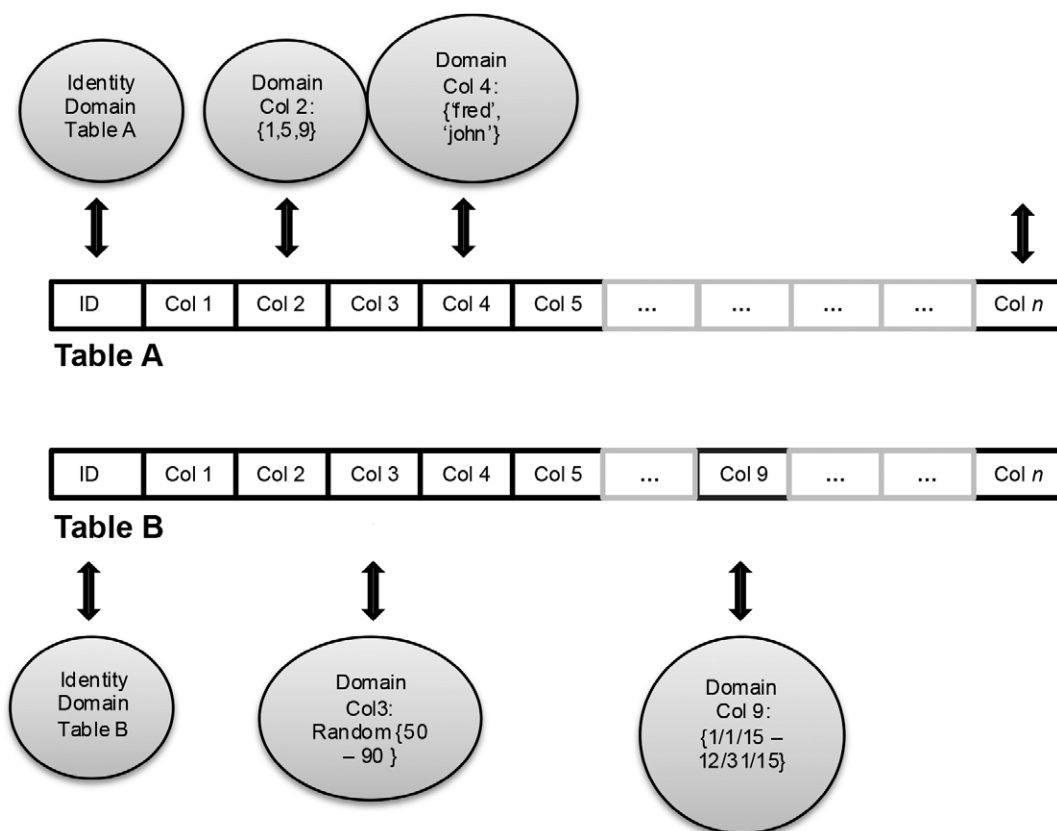


Figure 2

Our tool's approach to overcoming this limitation is to allow the test scripter to constrain the pool of possibilities from which values for a particular column can be drawn *prior* to generating data combinations. These are referred to as finite domains in the logic programming community; it is one of

several critical extensions to Prolog to support ‘constraint logic programming’ [JAFFAR 94]. While some dialects of Prolog provide programmers the ability to create finite domains for logic programming using integers [TRISKA 14], our tool has specialized finite domain machinery that is more suitable for the data generation problem.

Figure 2 depicts a simple example data scenario using finite domains across two different tables, A and B. We define finite domains for columns that are meaningful as far as a particular test scenario is concerned. Assume that the remaining columns are irrelevant to this particular test scenario. Columns are given either system or project defined single default values that are generally workable. Keeping the size of these domains in check has the effect of keeping the search space manageable. Generally, the number of data scenarios that can be computed as the Cartesian product of the size of all of the domains. In our example above, we end up with  $(1 \times 3 \times 2) \times (1 \times 1 \times 365) = 2190$  distinct data scenarios.

## 5 SDG: RQP powered by Prolog

We developed our test data generator using the problem solving strategy used in the “Card Number Six” example. We have initially dubbed it ‘Smart Data Generator’ (SDG) — patent pending. We chose SWI-Prolog [SWI-PROLOG 12] as the platform on which SDG runs. Attractive SWI-Prolog traits are its broad supporting library infrastructure, an active user community and a friendly licensing policy. However, SDG is theoretically capable of running on any Prolog platform since it is coded to standard Prolog APIs. An architectural overview of SDG is depicted in Figure 3.

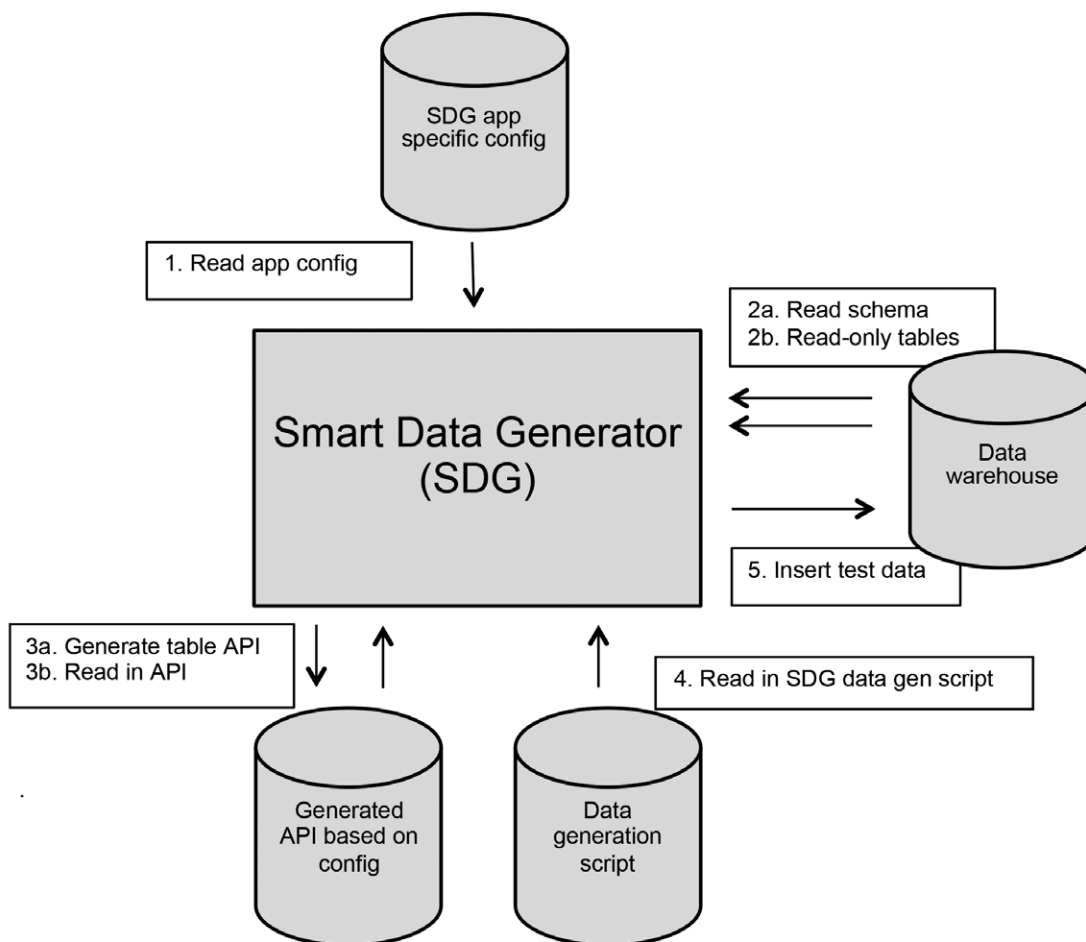


Figure 3

SDG reads in an application configuration file (#1, in Figure 3), which lists the tables (both read-only and write) needed to generate test data. This configuration file optionally can contain column specific information such as default values and column aliases. Using this configuration, SDG reads table and column schema information from the database (#2) that will be referenced in the creation of data. SDG then generates an API to these tables (#3), in addition to some "under the hood" machinery for handling *finite domains* on each table column. Generating database APIs from relational models is a well understood and solvable problem [ORM], [PERSISTENCE], [HIBERNATE]. SDG follows this same approach to generate APIs for each table as follows:

- Row object creators/constructors for each table
- Getters for each column on the table
- Setters for each column on the table

In step (#4), SDG processes test specific data generation scripts coded to the aforementioned generated API, Prolog's unification and back tracking inference engine does the lion's share of the work from there to generate data set solutions, and finally (#5) SDG inserts data directly into the database.

Referring back to our example of two tables in the prior section, we walk through a very concrete example, albeit fictional, of a scripted test case, and demonstrate how SDG would process it. For the sake of illustration, we'll simplify and remove the primary key from the example; given table T, with columns x and y.

```
assertDomain(T.x, [1,2]),
assertDomain(T.y, [a,b]).
```

Here we define the possible values that T.x can be to be 1 and 2; T.y to be a and b. Prolog will generate 2 x 2 (4) solutions for this goal, `create_T(T)` :

```
T=(1,a);
T=(1,b);
T=(2,a);
T=(2,b).
```

Adding a primary key identity to T will not increase the number of permutations; we add it in our next example adding a 2nd table T2 with foreign key to T, to which we will 'reverse' equijoin. Say T now has additional column oid (unique increment) and table T2 is containing columns oid,x,y,fkToT.

```
assertDomain(T2.X, [1,2]),
assertDomain(T2.Y, [a,b]).
```

The Prolog goal:

```
create_T(T), create_T2(T2), set_T2_fkToT(T2, T.oid).
```

will yield the following (2 x 2 x 2 x 2) 16 solutions:

```
T=(1,1,a), T2=(1,1,a,1);
T=(2,1,b), T2=(2,1,a,2);
T=(3,2,a), T2=(3,1,a,3);
T=(4,2,b), T2=(4,1,a,4);
T=(5,1,a), T2=(5,2,a,5);
T=(6,1,b), T2=(6,2,a,6);
T=(7,2,a), T2=(7,2,a,7);
T=(8,2,b), T2=(8,2,a,8);
T=(9,1,a), T2=(9,2,b,9);
T=(10,1,b), T2=(10,2,b,10);
T=(11,2,a), T2=(11,2,b,11);
T=(12,2,b), T2=(12,2,b,12);
T=(13,1,a), T2=(13,1,b,13);
```

```
T=(14,1,b), T2=(14,1,b,14);
T=(15,2,a), T2=(15,1,b,15);
T=(16,2,b), T2=(16,1,b,16);
```

Every solution is separated by semi-colons above and constitutes a separate data set. Let's finally demonstrate the addition of some constraints to filter out unwanted permutations. For example, we want to discard any data set where  $T1.x = T2.x$ . In bold *italics*, we add to our goal:

```
create_T(T), create_T2(T2), set_T2_fkToT(T2, T.oid), T.x \== T2.x.
```

This has the effect of culling out the number of solutions (data sets) by 8 :

```
T=(1,1,a), T2=(1,1,a,1);
T=(2,1,b), T2=(2,1,a,2);
T=(3,2,a), T2=(3,1,a,3);
T=(4,2,b), T2=(4,1,a,4);
T=(5,1,a), T2=(5,2,a,5);
T=(6,1,b), T2=(6,2,a,6);
T=(7,2,a), T2=(7,2,a,7);
T=(8,2,b), T2=(8,2,a,8);
T=(9,1,a), T2=(9,2,b,9);
T=(10,1,b), T2=(10,2,b,10);
T=(11,2,a), T2=(11,2,b,11);
T=(12,2,b), T2=(12,2,b,12);
T=(13,1,a), T2=(13,1,b,13);
T=(14,1,b), T2=(14,1,b,14);
T=(15,2,a), T2=(15,1,b,15);
T=(16,2,b), T2=(16,1,b,16);
```

## 6 SDG as a realization of Reverse Query Processing

For the intrepid reader, we provide an in depth example using SDG to generate data to test a data warehouse ETL in appendix A1. The biggest take away with this example is that the SDG script closely resembles the SQL query in structure, and that the semantic of the SDG is essentially the reverse (Reverse Query Processing if you will) of the SQL semantic. Figure 4 makes this similarity readily apparent by juxtaposing SQL side by side with the SDG / Prolog code:

Resemblance to SQL	
SQL	SDG / Prolog
<pre>SELECT   &lt;columns&gt;   ...   ... FROM   &lt;primary tbl&gt; JOIN   &lt;2ndary tbls&gt;   ...   ... WHERE   &lt; filters &gt;   ...</pre>	<pre>assertDomain(<u>dwi_WorkItemState</u>, ['Done']), assertDomain(<u>dwi_CreatedDate</u>, [NextWeekDay]) [...]</pre> <pre>create_dwi(Dwi), create_fwi(Fwi,Dwi), [...]</pre> <pre>Dwi.cCreatedDate = Fwi.cFactDate.</pre>

Figure 4

The `assertDomain` calls essentially define the column values that a SQL SELECT statement would return. The generated `create_XXX` calls are analogous to SQL JOINS. Finally, any trailing filters correspond to a SQL WHERE clause.

While the SDG does not generate *all* the data sets that could qualify for a SQL query, using finite domains, the SDG scripter has the tool he needs to limit the number of data sets returned — particularly down to those that are meaningful as far as a test case is concerned. Do realize that *all* the possible data sets could potentially be prohibitively large and no doubt contain massive redundancy as far as testing value is concerned. Using tried and true testing techniques such as combinatorial pairwise testing [KUHN 10], the SDG scripter can further pare down the number of data sets returned to gain the greatest testing value for a given minimal amount of data.

## 7 Closing Remarks

We have developed and utilized our SDG tool to test the OLAP back end of the Aeos system over the last year. We have found that it has proven highly effective at generating complex data used to functionally test complicated ETL processes used in the process of delivering business intelligence to end users. In fact, SDG has met all the other requirements that we called out originally. It requires users to script in a language that should be plainly intuitive to SQL programmers, and it also enables the easy derivation of data generation scripts from SQL where that is needed. We have also been able to inject data into the back end of our system a couple of orders magnitude faster than with our original in-house data generation tool.

Our approach can likely be generalized, from generating data to test an OLAP application to any scenario where relational data is sufficiently complex that it requires deep understanding of higher level business application logic.

While in truth, SDG is not a reverse SQL engine per se, there is no denying that a SDG script bears a very strong resemblance to SQL in structure and semantic. Our ETL testing example demonstrates that an SDG script is essentially a SQL SELECT in reverse, i.e. reverse query processing. So far, our use of SDG has been confined to mirroring relatively simple SQL statements. Further exploration will be required to see if SDG is up to the challenge of mirroring complex SQL statements containing, for example, OUTER JOIN, UNION, or COALESCE constructs.

An unanticipated result we found with SDG is that it does well to abstract away the often messy proposition of communicating with a relational database (e.g., order of inserts, complex SQL insert syntax, locks, optimization, etc.). Consequently, SDG scripts are more succinct, readable and maintainable than would be compared to writing SQL directly. Ultimately SDG allows the test scripter to focus on what is important, creating test data, without the distracting tedium of how to get data into the database. This is a generally recognized benefit that comes with any object relational technology [ORM].

Finally, in the interest of keeping the focus of this paper on Prolog as a test data generator, we have either glossed over highly detailed aspects of SDG or omitted mention of them completely. Additional details can be found in Appendix A2. Also, we made mention of the existence of two-way linked lists embedded in some of our tables, but did not discuss how SDG addresses the problem of generating this data construct; the reader should refer to Appendix A3 to gain deeper understanding.

## 8 Future Work

Since SDG is capable of generating data in CSV format, it could very easily be extended to supply data in scenarios where *any* data driven testing (DDT) methodology is being used, irrespective of the presence of a relational database. SDG could have even broader reach if it were able to generate into other data formats (e.g. XML) as well.

SDG is built leveraging the Prolog platform. However, other logic programming platforms could be considered. While we have not yet investigated Datalog [CERI 89], we suspect that it might provide a tighter integration with databases than do most Prolog platforms. Like Prolog, Datalog provides some of the same strengths that come with logic programming environments, but in addition to that, it also provides built in caching abilities so that data could be read in from the database without blowing up memory. Datalog might also provide improved database write functionality and performance. This could prove to be useful in scenarios where generating massive data was called for, as well as considering the state of data pre-existing in the database in the process of generating new data.

Although the SDG tool was originally conceived as a test data generator, because it provides an API to relational data, it could no doubt be extended to serve as middle ware in any multi-tiered application. SDG's logic programming capabilities could provide a whole new frontier that we don't typically see in multi-tiered applications. Furthermore, these features also make SDG a promising data analytics platform.

# Glossary

**Aeos® software** – Huron Consulting Group's OLAP/OLTP software solution to assist hospitals in bringing revenue into the books

**BI** – Business Intelligence

**Constraint logic programming (CLP)** – a refinement in logic programming to enable logic programs to operate on data types that might otherwise have virtually infinite domains and thus blow up search spaces

**data warehouse** – a back end data store architected in a 'star configuration' to support business intelligence and data analytics; contains fact records, dim records and summary table data

**dim records** – detailed data in a data warehouse that is generally static in nature and constitutes dimensional axes in data warehouse star architectures; contrast with fact records

**ETL** – acronym for 'Extract, Transform and Load'. In Aeos context, this means refining detail data into summary data via aggregate computation

**fact records** – detail data in a data warehouse that is more dynamic in nature, generally capturing deltas in data over time; contrast with dim records

**finite domain** – an important component in constraint logic programming

**measure** – an aggregate in a summary table computed via ETL from detail data (fact records and dim records) in a data warehouse

**metric** – a computation built on summary table measures, which has business meaning. It is intended to support business stakeholders in business intelligence and data analytics activities.

**OLAP** – Online Analytical Processing

**OLTP** – Online Transaction Processing

**SQL** – abbreviation for 'Structured Query Language', the de facto language of choice for the retrieval and manipulation of relational data

# Appendix

## A1. Using SDG to generate data to test an ETL

For the sake of protecting Huron Consulting Group's sensitive intellectual property, we do not give an example from testing Aeos software, but rather we concoct a fictional scenario — one that is simpler, and one with which the reader may be more familiar. Team Foundation Server (TFS), a Microsoft product, is a tool that you may have heard of or used before, particularly if you are involved in agile software development on the Windows platform. Like Aeos software, TFS is a collaborative workflow tool. *[ If this example bears any resemblance to any software analytics or business intelligence solutions (e.g. [CODEMINE 13] associated with Microsoft products, it is purely coincidental and not intentional.]*

Imagine now that there happens to be a business intelligence back end to TFS, and let's say data scientists have observed that when a work item has been touched by the assignee the very same day that it was first assigned to him/her, there was not only a higher likelihood the work item would reach a *done* state by the due date, but also the quality of the work would generally be higher. So business analysts figured it would be advantageous to create a metric and track TFS users' behavior over time, and finally report on it to managers who would then encourage his/her team to get an early start on their work. If the data analytics were right, the team would get higher percentage of work done on time.

Let's call this metric "First Day Touched Done Percent" to be reported on a monthly basis across TFS teams and work item types. This metric requires a numerator summary measure *WorkItemsFirstDayTouchDoneCount*, which is a count of work items Done in a particular month, and were touched by the assignee the first day of assignment (not necessarily within the month). The metric also requires a denominator summary measure *WorkItemsDoneCount*, which is the total count of work items Done in the last month *regardless* of when the work item was first touched by the assignee after assignment. Both of these summary measures are aggregated across teams and work item types to be computed using an ETL processing 'detail' data.

This metric will be displayed to managers in an intuitive graphical format that will display "First Day Touched Done Percent" side by side with Done Late Vs Done On Time pie charts, that might demonstrate

that as the percentage of First Day Touched work items increases, so does the work that is done on time. See graphical metric display in Figure 5.

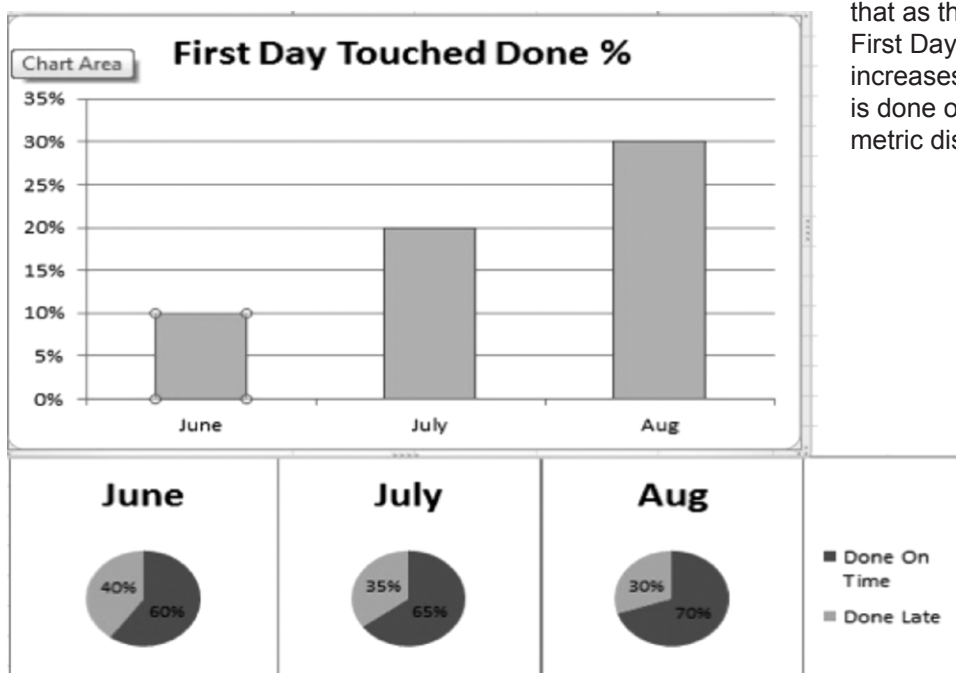
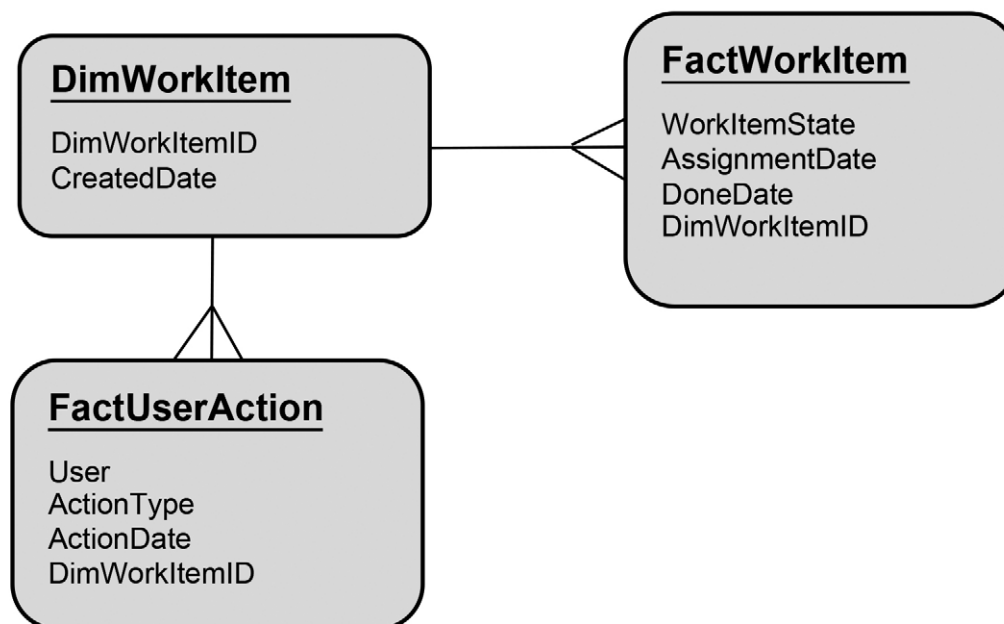


Figure 5

Detail data DimWorkItem table contains status information about the work item, such as work item type and created date. FactWorkItem table tracks changes on work items, containing information about what the state change was and when the state change occurred. The FactUserAction describes the actions that occurred on work items. It would contain the type of the action as well as when the action occurred and who performed the action. Refer to E-R Diagram in Figure 6.



**Figure 6**

We take a simple 'divide and conquer' approach to devising test data. Generate data that qualifies for the measures and data that does not qualify. We'll call these 'data cases' and we separate them into these 4 distinct categories:

1. Qualify for *WorkItemDoneCount* : We need work items that transitioned into Done state during the reporting week. Work items need be assigned to users on and after the CreatedDate and both on and before the DoneDate. Work items need to be assigned to users that span different teams. Work items should be of different types. Work items should have their Done Date coming before, on and after the DueDate.
2. Qualify for *WorkItemFirstDayTouchDoneCount*: same as for *WorkItemDoneCount* with the added constraint that the work item has a UserAction on the same date as the AssignmentDate.
3. Not qualify for *WorkItemDoneCount*: (1) Work Item is not Done or (2) DoneDate is not in the reporting week (i.e. not between first day of the week and the last day of the week), (3) work items to be assigned to different users on different teams; work items to span different work item types, work items whose Done Date comes before, on and after the DueDate. Not qualify for *WorkItemDoneCount*: (1) Work Item is not Done or (2) DoneDate is not in the reporting week (i.e. not between first day of the week and the last day of the week), (3) work items to be assigned to different users on different teams; work items to span different work item types, work items whose Done Date comes before, on and after the DueDate.
4. Not qualify for *WorkItemFirstDayTouchDoneCount*: same as for *WorkItemDoneCount* with the added constraint that the work item has a UserAction NOT on the same date as the AssignmentDate.

For the purpose of demonstration, let's focus just on data case 4. We'll first create a SQL SELECT query for this, and from there we will port it into SDG/Prolog to generate data that would qualify for this SQL query.

```
SELECT DWI.DimWorkItemID,
       DWI.Type,
       FWI.DoneDate,
       FWI.AssignmentDate,
       FUA.ActionDate AS FirstTouchDate
FROM DimWorkItem DWI
JOIN FactWorkItem FWI
  ON FWI.DimWorkItemID = DWI.DimWorkItemID
  AND FWI.DoneDate BETWEEN @Monday AND @Friday
  AND FWI.DueDate = @Tuesday
JOIN FactUserAction FUA
  ON FUA.DimWorkItemID = DWI.DimWorkItemID
  AND FUA.ActionDate > FWI.AssignmentDate
WHERE DWI.WorkItemState = 'Done'
AND DWI.CreatedDate < @Monday
AND DWI.Type IN ('SBI', 'PBI', 'BUG')
AND FactUserAction.User IN ('Viji', 'John', 'Hans')
```

In this data case, everything qualifies the data for the *WorkItemFirstDayTouchDoneCount* measure except for the line in bold italics.

The equivalent SDG/Prolog script to generate data sets for this is as follows:

```
assertDomain(dwi_WorkItemState, ['Done']),
assertDomain(dwi_CreatedDate, [LastWeekDay]),
assertDomain(dwi_Type, ['SBI', 'PBI', 'BUG']),
assertDomain(fwi_AssignmentDate, [Monday]),
assertDomain(fwi_DueDate, [Tuesday]),
assertDomainRange(fua_ActionDate, Tuesday, Friday),
assertDomain(fua_user, ['Viji', 'John', 'Hans']),
findall([Dwi,Fwi,Fua],
  ( create_dwi(Dwi),
    create_fwi(Fwi,Dwi),
    create_fua(Fua,Dwi)), DataSets),
sqlInsert(dataSets).
```

In this example, 'LastMonthDay', 'Monday', 'Tuesday', 'Friday' are variables that are set globally and adjusted per the reporting month that the tester is targeting. Note that with a simple change to the line in bold italics, we can change this script from one that generates non-qualifying data to one that generates qualifying data for our measure. By changing 'Tuesday' to 'Monday', data will be generated where the ActionDate indeed equals the AssignmentDate, (data case #2).

## A2. SDG in Finer Detail

- 1) **Expressibility in defining finite domains on a column.** Following the lead of third party data generation tools, SDG provides many options for creating domains on a given column, e.g. random N from a set of values, finite domains into read only tables either directly or symbolically, finite domains as ranges, finite domains via Prolog query, etc.
- 2) **Read-only tables (i.e., tables for which data will not be generated).** SDG recognizes the read-only table abstraction and it fits nicely with our finite domain machinery. Scripters are free to set domains for column values that are actually foreign keys into read-only tables (typically containing

dimensional data). These read-only tables can be referenced in script code symbolically (e.g. via code or English description) as long as there exists a designated column in that read-only table that can act as a symbolic identifier (e.g. 'John Smith', 'Legal Department', etc.).

- 3) **Triggers and data values that are dependent on other data values.** We have not given much attention to generic reusable trigger-like mechanisms. These would surely be useful, but we have been able to get by with setting dependent column values on a case by case basis as they arise. Furthermore, there is nothing to prevent an SDG scripter from writing customized Prolog predicates to address inter or intra table data dependencies in an application specific manner.
- 4) **Representation of primary and foreign keys.** In tables for which SDG generates data, it initially maintains all primary and foreign keys as object references just up until the time that they are inserted into the database. At this point, these object references are converted into object ids (OID) as they are represented in the database. We have taken this approach primarily because a lot of these data sets filtered out depending on the data generation script, and it is wasteful and problematic to give them OID keys prematurely.
- 5) **Tables with composite primary keys.** The Aeos data warehouse application has an overwhelming majority of its tables whose primary key is defined as a single unique OID, so we were not immediately compelled to grapple with the problem of composite primary key data generation. In the few cases where we do have tables with composite primary keys, we were careful in our data generation scripts NOT to generate more than one row with the same composite key. Certainly, there is possibility for improved support in SDG in this area.

### A3. Table Data Two-way Linked Lists

This snippet of Prolog code creates a two way linked list of FactWorkItem rows; see ER diagram in appendix A1. Added columns are FactWorkItemID primary key, and NextID, PrevID as our list pointers. Prolog's built-in list data structure and the use of recursion help make this code very succinct.

```
create_fwi_chain_(1, _, ChainIn, ChainOut) :-
    ChainOut = ChainIn,!.
create_fwi_chain_(Len, LastFact, ChainIn, ChainOut) :-
    NewLen is Len - 1,
    create_fwi(F1),append(ChainIn, [F1], ChainTemp),
    get_fwi_FactWorkItemID(F1, Pk_F1),get_fwi_FactWorkItemID(LastFact,Pk_LastFact),
    set_fwi_NextID(LastFact, Pk_F1),
    set_fwi_PrevID(F1, Pk_LastFact),
    set_fwi_NextID(F1, null),
    create_fwi_chain_(NewLen, F1, ChainTemp, ChainOut).
create_fwi_chain_main(Len, ChainOut) :-
    create_fwi(F1),
    set_fwi_PrevID(F1, null),set_fwi_NextID(F1, null),
    create_fwi_chain_(Len, F1, [F1], ChainOut),!.
```

## References

- [BINNIG 07] C Binnig, D. Kossmann, E. Lo. *Reverse Query Processing*. Data Engineering, 2007. ICDE 2007. IEEE 23<sup>rd</sup> International Conference on Data Engineering.
- [CERI 89] S Ceri, G Gottlob, L Tanca; *What You Always Wanted to Know About Datalog*; IEEE Transactions on Knowledge and Data Engineering, Vol 1. No 1, March 1989
- [CODEMINE 13], J. Czerwonka, N. Nagappan, W Schulte, B. Murphy, *CODEMINE: Building a Software Development Data Analytics Platform at Microsoft*, IEEE Software, Issue No. 4 – July Aug (2013 vol 30), pp. 64-71
- [COLMERAUER 93] A. Colmerauer, P. Roussel. *The Birth of Prolog*, ACM SIGPLAN Notices, 1993.
- [HIBERNATE] Hibernate. [https://en.wikipedia.org/wiki/Hibernate\\_\(Java\)](https://en.wikipedia.org/wiki/Hibernate_(Java))
- [JAFFAR 94] J Jaffar, M Maher; *Constraint Language Programming: A Survey*, Journal of Logic Programming 1994: 19,20: pp 503-581.
- [KUHN 10] D Kohn, R. Kacker, Y Lei, *Practical Combinatorial Testing*, NIST Special Publication 800-142, October 2010
- [KOWALSKI 74] R Kowalski. *Predicate Logic as Programming Language*, Proceedings IFIP Congress, Stockholm, North Holland Publishing Co. 1974.
- [ORM] *Object Relational Mapping*, [https://en.wikipedia.org/wiki/Object-relational\\_mapping](https://en.wikipedia.org/wiki/Object-relational_mapping)
- [PERSISTENCE] Persistence Software. [https://en.wikipedia.org/wiki/Persistence\\_Software](https://en.wikipedia.org/wiki/Persistence_Software)
- [SUMMERS 72] George J Summers; *Test Your Logic: 50 Puzzles in Deductive Reasoning*, Dover Publications Inc., New York, 1972.
- [SWI-PROLOG 12] Wielemaker, Jan and Schrijvers, Tom and Triska, Markus and Lager, Torbj, *SWI-Prolog*, Theory and Practice of Logic Programming, vol 12, no. 1-2 p67-98, 2012
- [TRISKA 14] Markus Triska, *Correctness Considerations in CLP(FD) Systems*, Vienna University of Technology, 2014

## Acknowledgements

Many thanks to Huron Consulting Group for creating an environment that fosters a culture of innovation, without which this work would not be possible. Thanks to Huron Consulting Group colleagues Eswara Jadda , Viji Ramadoss, Yogesh Lulla, Joe Woolston and Rob Weight for paying a critical eye to this work and profoundly shaping its direction. Special thanks to referees Patt Thomasson and Dave Patterson for their thorough review of this document to transform it into a polished and finished product; to Anne Ogborn of the SWI-Prolog community for validating SDG's use of Prolog; to Chris Keene and Derek Henninger for vetting this work from the perspective of object relational mapping (ORM).



# Testing Cloud Hosted Solution in Production

**Arvind Srinivasa Babu**

**Sajeed Mayabba Kunhi**

**Joshy A. Jose**

[arvind.srinivasa.babu@intel.com](mailto:arvind.srinivasa.babu@intel.com),  
[sajeed.mayabba.kunhi@intel.com](mailto:sajeed.mayabba.kunhi@intel.com),  
[joshy.a.jose@intel.com](mailto:joshy.a.jose@intel.com))

## Abstract

The evolution of software from a simple application running in an isolated system to a multi-node solution in a network and now to a geographically distributed cloud solution with hundreds or thousands of nodes, has brought about new testing challenges. Testing strategies which work well in one model are not always applicable to others.

While many of the tests for a cloud-hosted solution can be done by replicating a subset of the actual production environment, some tests need a production-like environment to be effective. This could be challenging if the hosted solution is complex or large. Though one might successfully replicate the production environment for testing, it is very difficult or too expensive to replicate end user behaviour, as the sample set of users in cloud could be very large.

One option to overcome the given problem is to perform the testing in a live production environment. This paper attempts to bring forth the challenges and possible suggestions for testing cloud hosted solutions in a live production environment.

## Biography

*Sajeed Kunhi is a senior engineering manager with Intel with nearly twenty years of software engineering experience. He is highly passionate about process automation, product quality and DevOps principals. His expertise in software development, technology domain and managerial experience has helped in building optimized software development lifecycle.*

*Joshy Jose is a Software Architect at Intel, with more than sixteen years of experience designing and implementing software. He has been instrumental in improving development and testing processes incrementally over the years. His areas of interest span security management, device drivers, system programming and mobile application development.*

*Arvind Srinivasa Babu is a Technical Lead at Intel, with more than six years of experience designing and implementing software. He has been performing the role of a scrum master and strives for continuous improvement in testing and development. His areas of interest span over network, user interface interaction, system programming and mobile application development.*

# 1 Introduction

When we look at validating the quality of software, the term “testing” covers a very broad domain. There are various stages of testing and strategies which are formulated and employed before a solution goes to a world wide release. A product is made up of many modules and features, which can serve as an isolated standalone solution or scale onto multiple machines that are geographically distributed.

When a solution is cloud based, there are several phases of testing and deployment strategies which ensure a quality release, but very often the quality comes into question when a production environment causes an issue in the solution that has slipped through various testing phases. Issues can be attributed to infrastructure differences, load on server and performance at various loads. The goal of every phase in testing is to qualify the solution from the smallest testable unit within the software through validation of end-to-end solution. Each phase has an associated cost attached to it and as the test strategy moves through these different phases, a complexity arises where factors in production cannot be replicated in a test environment. These factors are unpredictable, and cannot be assessed during design of the software. Testing in a production environment would allow such data to be collected and be refactored into design.

In this paper, we will refer to existing strategies and phases of testing as “*Traditional testing*.” It includes unit testing, integration testing, system testing, automation, mocking etc. But not all the tests that are covered in a test strategy, would prepare a product/solution from being ready for deployment in a cloud production environment. Most cloud solutions have a commitment to have an uptime of 99.99% or higher. If a solution fails to deploy through traditional testing and deployment strategies, we would be forced to break an agile cycle to either fix the issues or rollback the production environment to its last known state. This adds to the cost in maintaining a production environment with latest version of the solution. There is a significant difference between quality inferred from traditional test environments and production environments. We will explore such differences and how to overcome these problems in upcoming sections

First we will give a brief overview on the different teams and their roles in delivering cloud solutions for deployment in a traditional product lifecycle and the challenges involved in it. In section 3 we will discuss the differences between testing in traditional test environments and production environments and why it is essential to test in production. We will also suggest essential best practices and how they influence a solution to achieve a successful production testing strategy. In section 4 we will detail how production testing can be achieved. In section 5 we will discuss how telemetry can be leveraged to quantify and collect metrics and use them to build better solutions. In section 6 we will discuss risks involved with production testing and how to mitigate them.

## 2 Traditional Product Life Cycle for Cloud Deployment

### 2.1 Traditional validation of cloud solutions

Traditional testing as mentioned earlier is a collective term used for various testing methodologies, phases and strategies that are put in place for a deployment of a cloud solution. These methodologies include unit testing, feature testing, integration testing, system testing, acceptance tests, user acceptance test, feature verification tests, smoke tests and automation. These validation methodologies have their own test environment which is an approximation of known production environment parameters. Quality inferred from the results of these methodologies are an approximation of how well the solution might work in the actual production environment. These phases of validation or methodologies are targeted to infer quality with a particular scope and are often disconnected with each other. Once the solution is put through this traditional cycle of testing, it is qualified as a high quality release build which is taken up for deployment.

A cloud-hosted solution could be developed by a single engineering team or multiple engineering teams

depending on the complexity of the solution that is offered. Each team that takes ownership of one or more components within the solution may employ different validation strategies that works well within the team's delivery process. When such complex solutions are integrated during this traditional validation phase, a lot of time is invested to ensure that the integration between different components do not affect the overall quality of the solution.

We will briefly provide insight on how one such complex solution is integrated from different teams, how the solution validation is carried out before being deployed into production and the challenges faced with this approach.

## 2.2 Roles involved in traditional cloud validation

In a typical cloud deployment of a complex solution, there are three major roles or teams tasked with taking the solution from inception to deployment.

- Engineering
- System Validation
- IT Operations

A component within a complex solution or the solution itself is designed, developed and tested by an **Engineering** team. This team will be responsible for maintaining the components codebase, bug fixes and validation of the feature. They will reach out to other components' engineering teams and develop interdependent relationships. Each component within the cloud solution will have its own measure of quality validation process as defined by the Engineering team. If the quality confidence is accepted by stakeholders, the component is released to a System Validation team. The validation of the component by the Engineering team will be performed in a test environment that suits the component and that environment may not scale up to a production environment.

A **System Validation** team is responsible for bringing different releases of components from Engineering teams together to form the solution. This team validates whether components are compatible with each other and are conforming to cloud performance standards. Any issues within this validation phase will require back and forth communication with the respective component's Engineering team. Once the System Validation team approves the various components, the solution is rolled up for deployment. The test environment employed here will be a mock-up or scaled down version of the actual production environment.

A typical **IT Operations** team manages the cloud infrastructure and deployments. This team will pick up the deployment packages validated by the System Validation team and starts the deployment. A typical deployment process will include ensuring the database connections are closed, redirecting users to a downtime notification page or zones within the cloud infrastructure where deployment is scheduled for a later time, all backups are performed, etc. and then proceed to deploy the new version. Depending on the maturity of the solution and deployment methods that the solution will support, downtime will be incurred. These are scheduled downtimes, and the timespan of such downtimes will vary depending on how complex the deployment is and how many issues arise during deployment.

## 2.3 Challenges

When the **IT Operations** team starts deployment, they ensure all sufficient precautions like backup, downtime notifications etc. are performed. If the operations team faces issues with the deployment, the typical response would be to either rollback the production environment to its last backed-up state or a special team (possibly including members of the Engineering, System Validation and IT Operations team) is formed that fixes issues as they surface and ensures the deployment goes through, possibly incurring additional downtime due to additional validation cycles on the fixes provided.

Engineering and System Validation teams have their own delivery process, which can ensure a deliverable is on a daily, weekly, bi-weekly or monthly basis. Each deliverable can bring in new features or fixes addressed for issues discovered from previous deployment, but if there is no mature deployment pipeline, the identified issues that are existing within production cannot be addressed in a timely manner. If the different components employ different processes, delivery might differ on the complexity of each component and System Validation will incur a lot of time and cost in validation of different components at different times to deliver a stable deployable release.

These challenges may not be restricted to this traditional validation methodology but may span different solutions. There could be other strategies employed before a cloud hosted solution is deployed into production and it depends on the stability and complexity of the solution itself.

## **3 Production Testing**

### **3.1 What is Production Testing?**

Production testing is a validation phase that can be adopted for a cloud based solution, where certain types of tests are executed and collection of quality related information happens in the actual production environment. These tests will be executed in a planned and systematic manner as part of deployment and post-deployment which will help minimize failure rates and optimal utilization of cloud resources. Production testing is not intended to replace traditional validation strategies but provide an additional drive towards a higher quality and reliable release.

### **3.2 Why add Production Testing?**

Solutions that plan to be deployed on cloud or are already being deployed to cloud, need to have a mechanism that allows the different teams to rapidly respond to issues that arise from within the production environment. Issues can arise from within a solution's component or within the production environment which is very difficult to simulate in traditional validation methodologies. Quality inferred from production environments intend to carry more weight than quality information generated from a traditional test environment. A test environment is traditionally modelled based on an approximation of known production environment parameters and are generally not optimal for quality inference with scalability tests or load tests. The quality of the solution inferred is directly proportional to how close traditional test environments are modelled to production environments.

#### **3.2.1 Differences between traditional testing and production environments**

There are many aspects that vary between traditional test environments and production environments, a few are discussed in the following sub-sections.

##### **3.2.1.1 Infrastructure**

Infrastructure can refer to memory distribution, hardware differences, network topology, etc. The cost involved in replicating a production environment is directly dependent on the complexity of the infrastructure of the production environment. This cost factor affects how traditional environments are modelled and cost incurring infrastructure items like hardware, memory or network topology is scaled down or overlooked.

Network topology and its role in environments can be a crucial differentiator between production and local test environments. Test environments within a traditional lifecycle are often setup within local networks that usually have a very high throughput. A production environment can employ a complex network topology and can have different network throughput between different components of the hosted solution.

A latency ping between a database server and web application in a cloud solution can be very high in a production environment compared with a local test environment.

There are multiple infrastructure differences that can be cited between a traditional test environment and a production environment, but the quality inferred from these environments have significant differences in terms on performance due to the varying hardware, faster networks etc.

### 3.2.1.2 Quality data

Cloud-based solutions are designed and validated for deployment in production environments based on data regarding quality collected across various testing phases. A traditional testing methodology gives only partial information on how well the solution is performing and whether the solution is working well within the parameters of a local test environment. For example, unit testing only provides quality data as to how much code functional coverage and conditional coverage is achieved, it does not give a measure of how well different modules of the solutions are integrated with each other. As the solution goes through various phases of testing, all the data regarding quality is a measure of how good the solution works in an approximate production-like environment.

Engineering teams do not actively collect quality information from production environments and even if little quality data is collected, it is overlooked due to the round trip involved from Engineering to production. The data collected here would be invaluable to the product engineering team. The quality data obtain from live production environment will better position the solution to tackle real issues proactively.

### 3.2.1.3 Load simulation

A part of traditional testing is to load the solution with number of requests to evaluate the performance under load. These requests originate from a single network and do not give an accurate reflection of the actual performance. A production environment will actually provide the performance data accurately based on how users have loaded it.

## 3.3 Setting up for Production Testing

A solution that is validated in a traditional testing methodology has various gaps when it moves from one phase to another. The following are some best practices that influence a cloud-hosted solution to be “*Production Test Ready*.”

### 3.3.1 Continuous Integration

This is an **Engineering** centric strategy for how developers integrate code into a mainline branch continuously. This is partially implemented in solutions that use branching mechanisms or where code is not checked in for a few days until it is logically complete. Such practices lead to a problems when a large block of code developed across days or weeks gets integrated into the mainline suddenly thereby introducing lower code coverage, higher validation load, development process overheads and unfinished stories in agile processes. Following Continuous Integration within the team’s delivery process can ensure that tasks are not planned for more than few hours and that the code developed gets checked into the mainline several times during the day. Though this is not directly related to testing in a production environment, it is an essential process that will help in a Continuous Delivery/Deployment pipeline. Advantages of following Continuous Integration is that at any given integration, only a small manageable, testable amount of change will merge into the main codebase that can be validated with minimal effort.

### 3.3.2 Continuous Delivery

Continuous Delivery follows the same idea of discrete, well defined work items as Continuous Integration. Traditional testing methodologies will have some level of automation that are either triggered manually

or on a semi-automatic basis depending on when validation is required. Production testing would require a completed automated system by which small changes can be validated. Continuous Deployment is a continuous process, new code might require new automation tests written, and invalid tests can be removed. Planning for such activities within a solution's process cycle is critical for a successful Continuous Delivery pipeline.

### 3.3.3 Continuous Deployment

Continuous Deployment and Continuous Delivery are significantly different depending on the complexity of the solution. The team can either stop at Continuous Delivery or also employ a Continuous Deployment. The only difference between the two is that Continuous Delivery aims at producing high quality release that can be deployed manually anytime and is a required minimum when a solution wants to be ready for production testing. Continuous Deployment performs the deployment automatically once the solution/product has reached the end of Continuous Delivery process. This is a significant breakthrough for a solution and is extremely challenging to achieve, but when implemented successfully it will have a significant effect in maintaining minimal or zero downtime. We will explore different deployment methods when we discuss on setting up the production testing pipeline.

### 3.3.4 Features

Typically solutions offer various technologies as features or feature sets but a common practice within the Engineering process is that these features or feature-sets are targeted to address big functionality changes to completion that span multiple iterations. Such features carry a high risk of introducing new issues in production environments since a big change has skipped the **Continuous Integration** pipeline. When a solution wants to go to cloud, it must ensure that any and all features must be scoped into minute functionality that can be delivered in few hours. Existing feature enhancements should be broken down into something that can be delivered in a short span.

Designing features that address a small part of functionality can go a long way in minimizing issues within the production environment, allowing teams to respond to issues in a systematic manner. This also allows the team to track changes effectively.

### 3.3.5 Feature Flags

A feature flag or toggle is a valuable product design mechanism through which an entire feature can be disabled or enabled. These feature flags can be implemented at various levels, as part of UI features, deployment process and as part of the solution to target specific users. Feature flags can be used as part of production tests to measure a feature's performance in production and decisions to disable the feature can be taken either at end of deployment or when the solution is actively serving users.

Delays and disruptions in deployment strategies have a huge impact in production environments uptime. Feature flags reduce the risk of increased downtime by disabling features that perform poorly in the entire deployment process. Features can be released based on these toggles, but these toggles need to be removed once a feature is enabled for all users to avoid overhead of managing feature toggles that accumulate over time.

### 3.3.5 Incorporating Telemetry across the pipeline

#### 3.3.6.1 Why Telemetry is essential for production testing?

Telemetry refers to the automated process of collecting information within a sourced environment. The information collected can be anything that the module has access to, and it is essential that telemetry needs to be built across the production testing pipeline. Data collected and collated can provide better

data regarding the quality of the solution from Continuous Integration all the way until the solution is deployed and active. Analysis of this data and incorporating the feedback within the solution will help the solution be proactively ready to tackle production issues.

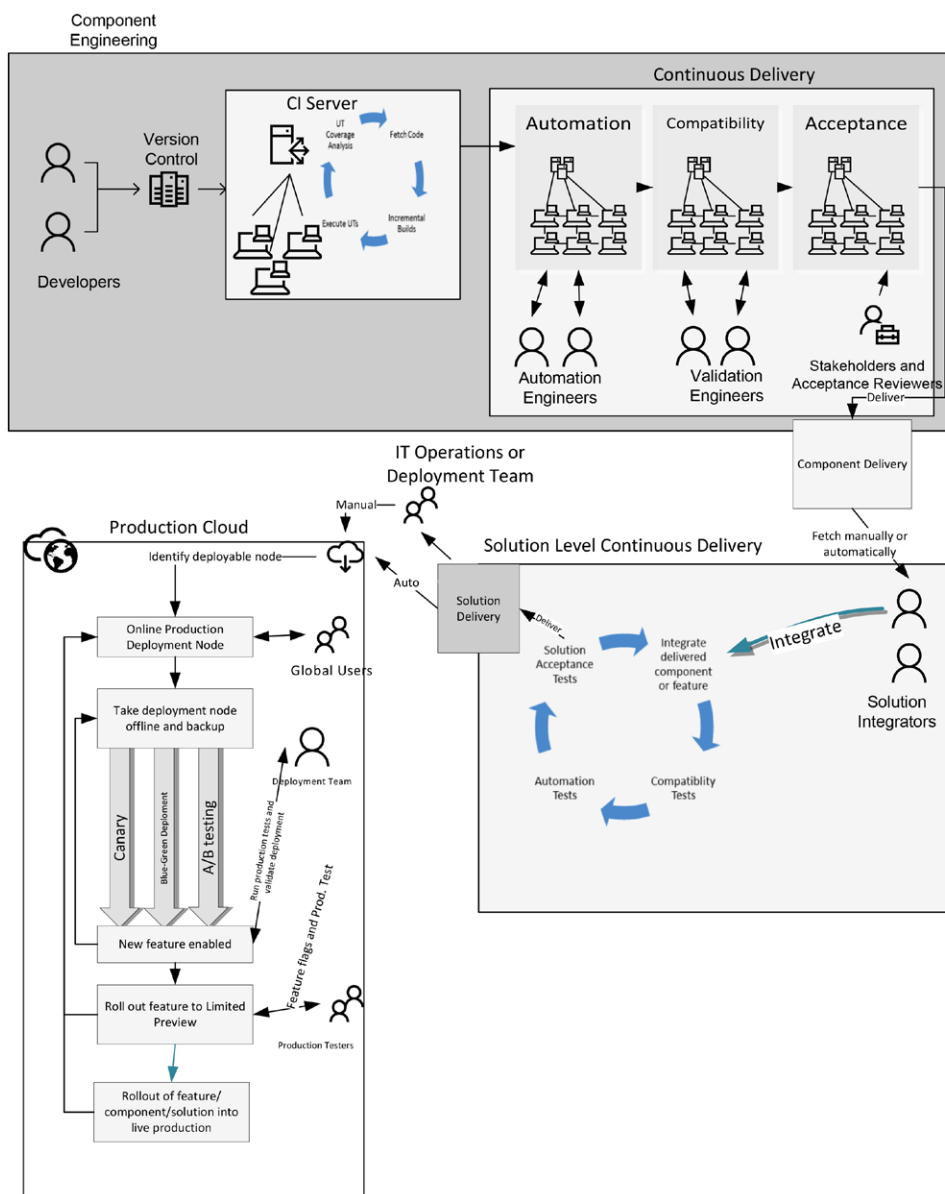
### 3.3.6.2 Privacy and Legality

There are laws and regulations in many countries that determine and guard what personally identifiable information is collected about their citizens. It requires clarity on the type of data collected and how the data is stored and utilized. Solutions should carefully consider collection of information which must conform to these rules and regulations. We will explore more on how data can be anonymized to respect privacy in upcoming sections.

## 4 Production Testing Pipeline

### 4.1 Overview

We explored the advantages of implementing different steps within a solution in the previous section. These concepts are better achievable by implementing a pipeline comprising of a well-defined process from Engineering all the way until the solution is deployed by IT Operations. Figure 1 depicts a suggested flow of how a cloud-hosted solution can be production tested.



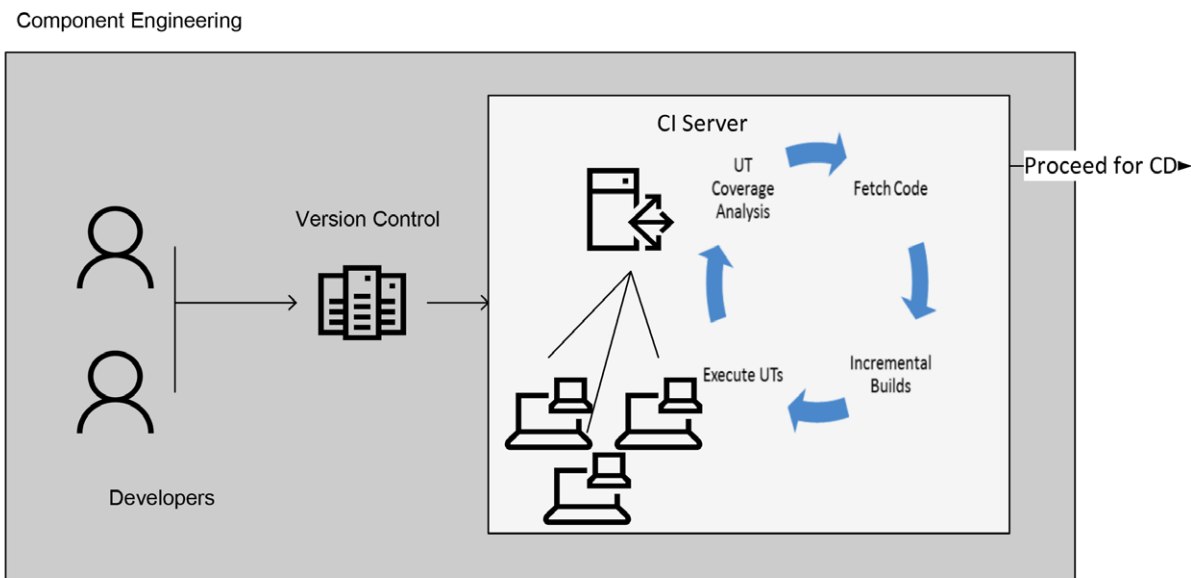
**Figure 1** Production Testing Pipeline, from component engineering to deployment and production testing

## 4.2 Continuous Integration

Continuous Integration (CI), as mentioned earlier, is an Engineering-centric strategy to maintain a shared codebase among different developers. Features should be broken into testable units that can be completed in a few hours and committed in the mainline codebase continuously. Automated Unit Tests should be written and executed in this phase. Code reviews can be controlled before the code is committed with an approval process as mandatory requirement for code-commit. CI servers can be configured to fail this phase if sufficient coverage is not met or number of tests falls below the acceptance criteria. If sufficient coverage is met in the Continuous Integration phase, it ensures that the code which was integrated during several times during the course of the day has not broken an existing test and new code has sufficient coverage. There is no set metric for code coverage in this phase and is entirely dependent on the design architecture and complexity of the code that was written and what can be achieved through unit tests.

The Continuous Integration pipeline need not be restricted to just individual production deployable modules or components but also to automation suites that support qualifying solutions and their components within the Continuous Delivery pipeline. Figure 2 represents a typical Continuous Integration pipeline within a components Engineering team.

Telemetry can be incorporated in this phase to collect coverage reports for the Engineering team to spend cycles in delivering incremental code with acceptable code coverage.



**Figure 2** Continuous Integration within a component engineering before moving to component's Continuous Delivery

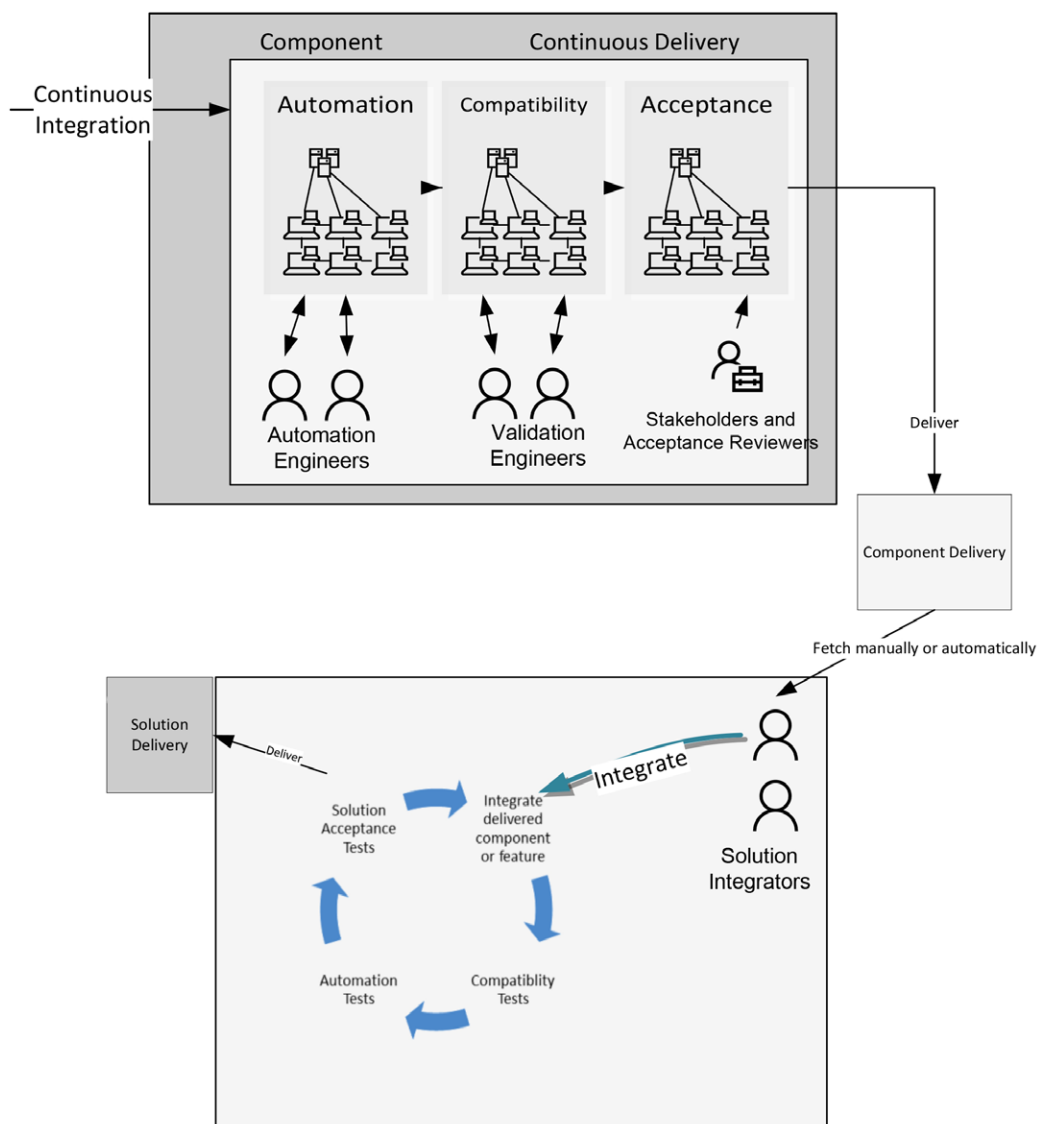
## 4.3 Continuous Delivery

Continuous Delivery is an extension to Continuous Integration, which performs automated test runs to meet acceptance criteria for stories. System level acceptance tests are performed to achieve overall compatibility between interdependent components. This process can be complicated and has a direct correlation with the complexity of the solution which is trying to achieve a continuous delivery pipeline. Large complex solutions brought to cloud may not have sufficient automation coverage and largely rely on black box testing. Such solutions should actively work on increasing automation coverage and devise a process that will ensure that CI delivered solution is tested by end of a sprint is of high quality and ready to ship.

One of the essential factors for Continuous Delivery is how solutions and their components are designed as features. Features, as mentioned earlier, allow different teams to have visibility to the changes made, and provide feedback on failure as soon as possible. This also allows the feature developed to be validated and readily available to be deployed into production. Application release automation and build automation are some of the type of methodologies that allow and execute various parts of Continuous Deployment pipeline.

Continuous Delivery can be implemented both at a component level (for complex solutions) and at a solution level. For a solution level Continuous Delivery pipeline, the System Validation Team can integrate the different components delivered as part of the component level Continuous Delivery pipeline. A System level automation can be run to ensure compatibility between different components and proceed with a system level automation to validate end-to-end functionality and acceptance tests for the entire solution. The different Continuous Delivery pipelines will provide a high quality deployable package that can be deployed manually or can be deployed automatically using Continuous Deployment strategies.

Figure 3 represents a Continuous Delivery pipeline at both a component level and solution level.

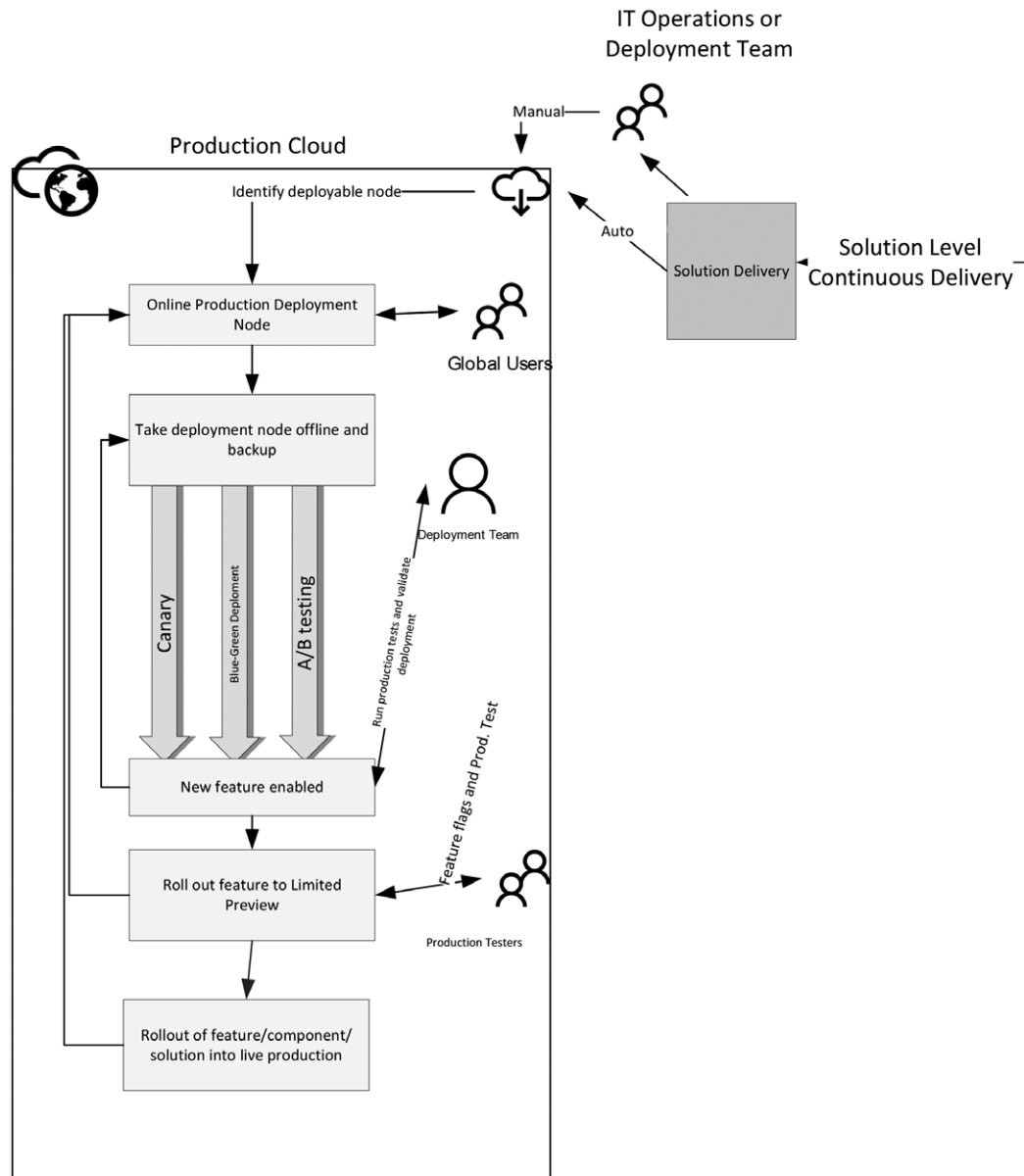


**Figure 3** Continuous Delivery at Component Level and at Solution Level

## 4.4 Continuous Deployment

Deployment is started once a solution is delivered as part of the Continuous Delivery. This process is currently manual for most cloud solutions and is done in a scheduled maintenance window. Continuous Deployment is a process that is elusive for most solutions, attributing to huge deliveries in traditional methodologies. When proper production testing pipeline is implemented, deployment can be triggered automatically where features or small incremental changes have gone through various automated tests and can be safely deployed into production.

Figure 4 is a representation of Continuous Deployment within the Production Testing Pipeline



**Figure 4** Continuous Deployment takes the solution and uses any of the below mentioned deployment strategies and uses feature flags to control features for production testers.

Next we will discuss strategies that can be employed to deploy the solution into production environment.

#### **4.4.1 Canary Deployment**

Canary deployment is done when a new version of the solution is rolled out. It starts with rolling out features on to a subset of the production infrastructure where no users are routed. A few production tests can run here to test whether the environment meets the prerequisite for the solution. Next step would be to slowly route a small batch from the total set of users into the infrastructure holding the new version. If an issue is encountered with a new feature, feature flags can be used to disable the feature completely without impacting the rollout. If the rollout was successful, the users can be redirected to the updated portion of the infrastructure by a load balancer and other zones within the infrastructure can be updated until the solution is fully deployed in the entire production environment.

#### **4.4.2 Blue-Green Deployment**

This kind of deployment is done where multiple copies of the solution are maintained inside the production environment. Select pods within such environment are marked as “Blue” and others as “Green”. Blue environments are set to handle live traffic, and the Green environment would be brought offline to perform the deployment. Rapid quick tests can be executed in this window to ensure that the environment will start up properly. Once confidence on the new version is developed, users can be switched or routed to the new version (Green pods) which will go live and the same process will now be deployed onto the Blue environment that will be taken offline. This kind of deployment can be performed where the solution needs to be deployed to indicate version changes. Performing tests in such a stage will help plan future deployments safely with minimal downtime.

#### **4.4.3 A/B Deployment and testing**

A/B testing is followed to determine a better visual user experience by providing the same information in different formats within the web page. In production testing, A/B testing can be implemented easily by incorporating both visual designs as features with feature flags based on continuous assessment from production environment across different targeted users. This testing initiative will help design better user experience on the user interface since the quality data is derived from the actual production environment from real users.

### **4.5 Testing in Production**

So far, we discussed on different deployment mechanisms and running production tests within them. These tests are specifically designed to be run on production environment and are a collective effort of Engineering, System Validation and IT Operations team. Teams representing various components should collaborate and plan features. These features have feature flags incorporated to toggle them during production validation among certain targeted users. Based on their performance the feature flags are ensured to be always on and removed and rolled out to the public.

During deployment, features use deployment or build related feature flags to control deployment. Once features are enabled, they are targeted to certain types of users using permission flags. These features will remain in production and rolled out to all users based on their acceptance in production environment. Validation engineers become a subset of targeted users who arrive at rolling out to different release phases. Features can be enabled or disabled based on how the quality information derived from running these types of production tests within the pipeline.

### **4.6 Release Strategy**

One important component in a production testing pipeline is having a proper release strategy. The release strategy involves formulating a release timeline from inception to deployment of every small code block

that gets committed within the Continuous Integration pipeline. Withholding solution delivery over a period of time accumulates risk of deployment failures and increased downtime. If a code block is delivered as part of production pipeline, it needs to be deployed with production tests. Controlling the failure of features can be done with planned releases targeting specific users initially before being rolled out to all users.

## 4.7 Phased Releases

Phased releases are part of release strategy that should be implemented within the production environment so that new features are rolled out to smaller audiences initially and eventually to larger audiences. Examples of phased releases to consider are **Internal Release**, **Limited Preview Release**, and **Global Rollout**. There can be more intermediate release phases according to the requirements of production testing. The advantage of doing phased releases is that in production, features can always move into next phase independent of when they were delivered. For example, Feature A and Feature B have entered a limited preview phase, after few cycles, Feature B's performance and quality confidence is far superior compared to Feature A, then Feature B can move onto a global rollout even though Feature A might have been delivered first.

### 4.7.1 Internal Release or Targeted Release

This is a limited release targeted to users within the organization or early adopters. Typically, new features that have passed the various tests in production environment are put to test here to see their performance and functionality. If features have passed through this release, the next step is to roll it out as a limited preview. Feature flags can be used here to allow validation and automation engineers to hook up automation in live production.

### 4.7.2 Limited Preview Release

A limited preview release phase rolls out the features into a larger audience for example, to external beta adopters. Feedback from the users can be engaged to improve upon the feature. If there are no issues and confidence on quality of the feature is high, it can be rolled out to a much larger audience or the public. Features that have passed internal release can stay in limited preview while new features are rolled out to the public. The whole idea is to measure feedback and improve a solution.

### 4.7.3 Global Release

A feature that has passed through internal release, and limited preview can now be considered to be released to the public. This is the final release phase where everyone in the system will be able to use the feature. A feature that has achieved this milestone has gone through various rigorous testing phases in traditional test environments as well in production environment with smaller audience. A feature that has been promoted to this milestone indicates that it is of a very high quality.

## 5 Quantifying Telemetry Data

We will briefly look on how telemetry can be used to quantify metrics and utilize the data for delivering better quality releases for Production Testing.

### 5.1 Design as a feature with feature flag

One of the features that can be built within the solution is a telemetry feature. Telemetry can raise flags across the industry with concerns of what information is being collected, but the data collected here would help a long way in helping Engineering teams plan better features. Design and integration of this module

is of great importance, core modules that do the low level work can integrate with telemetry feature, publish information of their activity which allows telemetry feature to translate into metric sets.

## **5.2 Quality and Performance metrics**

### **5.2.1 Quantifying a metric**

A metric is a standard measure of degree of a property that the solution possesses, e.g. quality metrics, software metrics, infrastructure metrics, etc. Metrics collected from production environments have huge value as they represent the actual data on how well the solution is performing in real time. The metrics derived from traditional testing are different from the metrics derived from production testing as the data collected from production can be viewed as sensitive by customers/clients and would require multiple levels of privacy being built in.

When a cloud-hosted solution is deployed live, key metrics to collect include amount of warnings and errors generated during deployment, downtime, uptime, requests served per time units, database access times, ping time and subnet distance between different components, round trip time between various network communication due to a request, file I/O's, resource consumption.

Metrics are currently collected only when there is an issue or they are mostly ignored. Having this information would go a long way in providing better performing solution in future releases. Integrating metric analysis within a product lifecycle will go a long way in anticipating production issues and provide better features proactively.

### **5.2.2 Categorizing Metrics**

Grouping metrics is a critical component of telemetry. Metrics need to be grouped according to the anonymized information collected. In each telemetry session, the datasets need to include number of I/O's performed, database access, database query performance, number of requests served and average time between responses. Feature specific metrics can be built into during the development of the feature, which in turn can generate lot of metrics to better anticipate their performance.

The data should be grouped into sub-categories relevant to the solution. For example, network related information like ping time, subnet distance, and round trip time can be collated under network metric. This network metric can be individually collected for various component to component interactions (WebApp to DB and/or WebApp to file system etc.). The design of the data set would vary from solution to solution due to varying components in the design of the cloud hosted solutions.

Reporting all this information should not hamper the performance of the solution, which is why telemetry should be designed as a plug-and-play feature that can be turned on or off at any time.

### **5.2.3 Anonymizing metrics**

Anonymizing metrics is a key strategy to be able to legally collect the data from within cloud solutions. All cloud performance metrics must include an identifier (custom-generated) that would uniquely identify different components for which metrics are calculated.

As pointed out in an earlier example, a ping time between a web solution and its production database can be a simple metric collected by anonymizing the components such as the WebApp and database which should be represented as unique identifiers. This would obfuscate the data where no client information would be included, including IP, DNS names, etc. A similar example can be made for subnet distance where only hop count can be collected between components and information on routers/IP/Subnets on the way need not be collected.

### 5.3 Utilizing metrics for continuous feedback

A feature's quality is determined on how well it works under different conditions under various scenarios. Through every phased release, a feature will be subjected to different loads, usages and tests. Metrics derived from this production testing phase will provide constant feedback to the engineering team to invest better to anticipate user expectations.

### 5.4 Designing data for anonymization

With production testing, collection of data is critical for better quality. Design of anonymized datasets would allow customers' information to remain private. These datasets could measure performance between different components which is part of the solution and would grant the merit of legally accessing the information.

Analysis of the anonymized dataset would require a thorough inspection of what are the network issues, performance issues on script executions, throughput of data, request/response speed ratio etc. The datasets which are categorized identify data valuable to engineering teams regarding the production infrastructure, load and performance of the solution without risking the privacy of the customers. Categorized grouping of data allows anonymized data to generate patterns where the solution is failing and addressing issues within the production pipeline.

## 6 Risks of Production Testing

### 6.1 Bugs, feature sizing and brand

Bugs are a fact of software life — with production or test environments. Early detection helps improve quality release over release. A bug that arises from production reflects poorly and can show a gap in the testing strategy. It could be environmental or a product defect. Not every code path can be tested when there is release pressure. Features should be designed to be very incremental. A wrongly sized feature can be an oversight and has potential to break or perform badly. When a feature is marketed beforehand and fails to deploy, it can lead to bad publicity for the solution. Cloud hosted applications require rapid response and code gets checked in throughout the day. A bad release strategy, overzealous marketing, and continuous deployment of breaking features detracts customers and clients which can have lasting impacts on the company brand.

### 6.2 Deployment Failures and Rollbacks

Another risk that most cloud solutions run into are deployment failures and rollbacks. With feature flags, new features should be disabled, if there are incremental improvements are made to features that are already rolled out to global production, the feature should not be disabled. Proper deployment rollback needs to be performed, frequent deployment failure and rollbacks can significantly increase the downtime. Another important thing to note is following bad deployment practices like failing to ensure the database has no active transactions or failure to backup all necessary information can affect rollbacks or risk losing critical business information.

Another risk with deployment failures can arise from incorrectly managing feature flags that can potentially allow a feature to be released without proper validation. As multiple features and their corresponding feature flags are developed, these flags accumulate within configuration files over multiple releases. This adds the overhead to micro-manage each flag beyond their intended use at the time of production testing and global rollout. Once features are enabled in production and enter into the global release phase, it's associated feature flags that control the deployment should be removed from configuration files. Failing to

do this increases the time production testing team and deployment team spend on identifying the correct feature flags to enable or disable during each deployment cycle.

### **6.3 Net Promoter Score**

Net promoter score can be critical component to an organization to determine how well a solution is performing. A seamless deployment mechanism where downtime is not noticed by a user is an indication where the solution will likely have promoters. A cloud solution that has nothing new to offer over a period of time coupled with failed features and downtime has the potential which leads the detractors to simply give up on the cloud-based solutions. It is essential to balance testing strategies, features and production testing in line with traditional testing methodologies to focus on achieving a good net promoter score.

## **7 Conclusion**

In this paper we have discussed on how quality confidence is impacted across traditional and production test environments. We have suggested some best practices for making a solution to be production-test ready by designing a solution as small incremental features, incorporating feature flags to help disable poorly performing features during various strategies of continuous deployment. We have also discussed setting up a production testing pipeline alongside existing traditional testing methodologies and releasing features independently in a phased manner. We have discussed in length about the benefits of incorporating telemetry within a solution as a feature, anonymizing datasets and grouping them into different categories and design a better solution proactively by including telemetry analytics within a product's lifecycle. We also called out some risks in production testing if proper controls are not implemented. Production testing is an extension to existing traditional testing methodologies and provide accurate performance and behavioural results in actual production environments. Utilizing quality related information from production tests will help a web-based solution to deliver better quality features, with minimum or zero downtime.

## References

1. Canary deployments: <http://docs.octopusdeploy.com/display/OD/Canary+deployments>
2. A/B testing, Blue-Green, Canary Deployments: <http://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/>

# Test Estimation

Shyam Sunder

[shyamconf@gmail.com](mailto:shyamconf@gmail.com)

## Abstract

We have executed many projects Large Projects, Small Projects etc. Sometimes we miss our testing deadlines because there is no defined criterion that is used to build our execution test plan. The Testing timelines often get squeezed up or crunched because it is the last phase in Software Lifecycle. Sometimes we miss our schedules due to crunched testing timelines. In order to avoid such missing of our deadlines we have prepared the Test Estimation guidelines.

In this paper I present the various Test estimation techniques which will help us in proper execution of the Testing projects.

The estimation technique guidelines and templates created by me have really helped the Organizations significantly. Effort & Schedule Variations have significantly improved from +-60% to +-2%. Piloting these guidelines for couple of projects in your organization and comparing the estimated effort and actual effort will show significant improvements. As we get proficient with its implementation we will find that Estimated and Actual efforts are getting closer which will result in better execution of the testing projects.

I will explain in detail some of the Test Estimation templates which I have implemented in my organizations. I will also demonstrate how to choose Test Estimation templates which will best fit your organization's software development process.

## Biography

*Shyam Sunder is a PMP® Certified Manager with total IT Testing Experience of 18+ Years and currently working in Sidra Medical & Research Center (Qatar). Shyam has worked in various reputed organizations like United Health Group, HCL, and Dell etc. Shyam is strong in the area of Test Management, Software Testing areas and Client relationship management. He is well versed in testing areas and has been actively involved in IV&V along with his testing delivery projects. Shyam has a consistent track record of successful product introduction and implementation. And is Productive as both individual contributor and Project Manager. Shyam also possesses excellent communication and relationship-building skills.*

*Shyam has been a regular contributor in Testing Forums like Quality Assurance of India (QAI), Software Testing Conferences (STC) etc. with his papers, presentations and workshops.*

*Shyam has received excellent Client testimonials which can be seen in his linked-in profile.*

*Linked in Profile: <http://www.linkedin.com/in/sundershyam30>*

### **Shyam's Professional Certifications**

- PMI Certified PMP®.
- Certified Test Manager
- Certified Scrum Master
- 6-Sigma Green Belt Champion.
- KPMG Certified ISO 9001 Internal Auditor.
- Brainbench Certification in Project Management.
- AHM 250
- Healthcare Integration Architect
- Healthcare Consultant

Copyright Shyam Sunder 06-July-2016

# 1 Introduction

We have executed many projects Large Projects, Small Projects etc. Sometimes we miss our testing deadlines because there is no defined criterion that is used to build our execution test plan. To help avoid such missing our deadlines we have prepared these Test Estimation guidelines.

Why good estimates are important?

- Testing is often blamed for late delivery
- Testing time is "squeezed"
- It promotes early risk assessment

Test Estimation is a prediction based on probabilistic assignments and is a continuous process, which should be followed and used throughout the project life cycle. Effective software estimation helps track and control cost/effort overruns. Estimations cover following broad areas:

- Estimate size
- Estimate cost & effort
- Determine the schedule
- Assess risks

## 2 Issues while Estimating

There are 4 main issue categories which creep up while estimating. These are:

i. Process

Requirement Stability, Change Requests, Finalize types of testing ,Follow the test process,Non-availability of test cases and test data, Timely reviews of the test cases and other artifacts, Coordination with various teams/modules/interfaces, Scope changes.

ii. Environment & Tools

Separate Environment, Environment not similar to the deployment environment, Downtime of the environment available during testing, Availability of test management tools, Availability of test automation tools.

iii. Testing Resources

Management commitment towards completion and following the test life cycle, Common and realistic expectation toward the testing goal from all the stakeholder in project, Availability of key resources, Application knowledge among the test team, Connect / Attitude between the development and testing team, Correct resolution on the defect fixes, Clear Communication.

iv. Others factors

Complexity of the application under tests, incorrect assumptions during estimation, Ownership of testing, Independent \ Development Vendor\ Customer

Timeline assigned for testing, Development timeline and release of code for testing, Availability of correct test data during test execution, Geographical location of testing team involved, Not doing periodic re-estimation

### 3 Software Estimation Techniques

There are different Software Testing Estimation Techniques which can be used for estimating a task.

- 1) Delphi Technique
- 2) Work Breakdown Structure (WBS)
- 3) Three Point Estimation
- 4) Functional Point Method

But the above models are more suited for Development projects/areas. For Test activities, the below models implemented and recommended by me are more suitable and practical.

There are 4 different software test estimation techniques as mentioned below:

- a) Simple Medium Complex(SMC) Method
- b) Top Down Method
- c) Bottom Up Method
- d) Test Point Analysis(TPA)

Now in the following sections I would be explaining the above mentioned estimation techniques in detail:

#### 3.1 SMC Method

This model will consider the test functions / test conditions and their Complexities (Simple, Medium and Complex) as the basis for estimation and the effort involved for the following test activities can be estimated using this model. Following test activities could be covered

- Test Initiation
- Test Planning & Design
- Test Execution
- Test Closure activities

Test Initiation:

Effort estimate for the following activities can be done using SMC model under Initiation Phase:

- Knowledge Transfer
- Application Familiarity
- Requirements Analysis
- Functional Decomposition

Test Planning and Design:

Effort estimate for the following activities can be done using SMC model under Test Planning & Design Phase:

- Test Plan
- Preparation of Scenarios, Test Cases, Test Data.
- Test Case, Test Data Reviews
- Preparation of Execution Plan
- Test Ware Re-work & Reviews
- Prepare and Review of Zero-day checklist

Test Execution:

Effort estimate for the following activities can be done using SMC model under Test Execution Phase:

- Verify zero day check list
- Creation of test bed
- Test Execution
- Review of Incident logs
- Update the Incident report

Test Closure:

Effort estimate for the following activities can be done using SMC model under test closure Phase:

- Closure Metrics Preparation
- Closure meeting
- Archive project data (Project Closure Activities)
- Test / Project Management

### 3.2 Top Down Method

In this method, the Overall effort estimate for the project is determined first in FP or Line of code method. The estimation procedure is as follows:

- Get the total size in FP
- Define the lower level project test component.
- Based on experience and productivity data from previous projects, obtain the effort estimate
- Overall effort estimate = productivity \*size

### 3.3 Bottom Up Method

This is also known as “divide and conquer” technique. It is hierarchical decomposition of the test effort into stages, activities and tasks.

- Planning
  - Test environment & configuration
  - Test case creation
  - Test execution
- ◆ Again decompose the above activity in smaller packages which can be estimated in short period of time.
- ◆ Estimate the total effort by understanding the duration and effort of each activity.

### 3.4 Test Point Analysis (TPA)

Test Point Analysis can be used to objectively prepare an estimate for black box testing (excluding performance testing). Test Case Point Analysis methodology is based on Test Case Points. Test Case Point is a Verification Point used to verify that the value on AUT matches with the expected value. This O/p value can be I/p data for other verification points. Following factors will have influence on number of Test Case Points:

- **Complexity:** It relates to the number of conditions in a function. More conditions almost mean more test cases and therefore a greater volume of testing work.
- **Interfacing:** The degree of interfacing of a function is determined by number of data sets maintained by a function and the number of other functions, which make use of those data sets.
- **Uniformity:** The extent to which the structure of a function allows it to be tested using existing or slightly modified specifications, i.e. the extent to which the information system contain similarly structured functions.

Details about Test Case Point are as follows:

- **Low Complexity Test Case Point:** A Test Case Point having 1 to 3 Steps is considered as Low Test Case Point.
- **Medium Complexity Test Case Point:** A Test Case Point having 3 to 4 Steps is considered as Medium Test Case Point.
- **Critical Complexity Test Case Point:** A Test Case Point having 5 to 6 Steps is considered as Critical Test Case Point.

Test Scripts can be defined in following three Complexity Levels:

S. No.	Level	Definition
1	Critical	If a Test Script is having 6 to 8 Test Case Points Or Verification Points.
2	Medium	If a Test Script is having 4 to 5 Test Case Points Or Verification Points.
3	Low	If a Test Script is having 1 to 3 Test Case Points Or Verification Points.

Breakdown between Testing Phases:

	Testing Phase	% age
Test Scripting	Preparation (includes Functional Understanding)	10
	Specification (includes Test Conditions, Test Data identification, Test Script preparation)	40
Test Execution	Test Script Execution and Defect Management (includes Smoke, System, Integration, End to End and Regression Test)	45
	Completion (includes UAT)	5

Factors affecting Test Estimation

- **Productivity Figure:** It is based on knowledge and skill of test team members and is therefore specific to the individual organization. Productivity figure mentioned in these guidelines needs to be verified for couple of projects before implementation across the entire organization.
- **Environmental Factor:** Following environmental factors should be consider for test effort estimation:
  - **Test Tools:** It reflects the extent to which testing is automated, or the extent to which automation tools are used for testing.
  - **Development Testing:** It reflects the extent to which the development testing is down, a development test plan is available and test team is familiar with the actual test cases and test results
  - **Test Base:** It reflects the quality of system documentation upon which the test under consideration is to be based.
  - **Test Environment:** It reflects the extent to which the test infrastructure in which the testing is to take place has previously been tried out.
  - **Test ware:** It reflects the extent to which the tests can be conducted using existing test ware.
  - **Multiple Browsers:** Effort estimation for testing on multiple browsers is more than testing on one browser.

## 4 Conclusion

The estimation technique guidelines explained in the earlier section can be enhanced to cover the various environmental factors. Pilot these guidelines for couple of projects in your organization and compare the estimated effort and actual effort. As we get proficient with its implementation we will find that Estimated and Actual efforts are getting closer which will result in better execution of the testing projects.

The Four Test Estimation models explained in this paper have been widely recognized as Industry Standard and have been used in different Organizations in some form or the other based upon the Organization specific needs.

The estimation technique guidelines and templates created by me have really helped the Organizations where I worked significantly. Effort & Schedule Variations have significantly improved from +-60% to +-2%. Pilot these guidelines for couple of projects in your organization and compare the estimated effort and actual effort. As we get proficient with its implementation we will find that Estimated and Actual efforts are getting closer which will result in better execution of the testing projects.

## 5 Definitions, Abbreviation and Acronyms

Acronym	Description
QA	Quality Assurance
SEPG	Software Engineering Process Group
SMC	Simple Medium Complex Method
TPA	Test Point Analysis
FP	Function Point
UAT	User Acceptance Testing

## 6 References

Various Test Sites:

- <http://www.softwaretestingclass.com/software-estimation-techniques/>
- [www.test.com](http://www.test.com)

Feel free to get back to me at [shyamconf@gmail.com](mailto:shyamconf@gmail.com) in case of any clarifications.

# Unified Test Library and Fault-Tolerant Automatic Test Script Creation

Chandrashekhar M V; Manini Sharma

[chandrashekhar.m.v@intel.com](mailto:chandrashekhar.m.v@intel.com); [manini.sharma@intel.com](mailto:manini.sharma@intel.com);

## Abstract

The process of test content development, test automation and faultless execution of complex systems, involving unstable or under-development hardware and software, is a challenging task. The product design complexity, interoperability issues, component interdependencies, changing product requirements, crunched schedules for parallel product lines and market pressures lead to compromise on quality of test development, reuse, review, maintenance and automation scripting. How do we curb the churn in test content, bug escapes, coverage gaps, security vulnerabilities, automation reliability efforts, and test environment stability issues, and still save resources with maximum reuse across projects?

There are several commercial tools available for test management. These tools provide the option of traceability of requirements to test cases through relationship metrics. With product development methodology changing from traditional approach to agile, Behavior Driven Development (BDD) and Test Driven Development (TDD) are becoming vital for time-to-market (TTM) while maintaining high quality. While the new enhancements and feature additions are a continuous process in the agile world, regression testing is necessary to keep the product stable and reliable for every release. As regression testing is a costly process on continuous releases, the industry is moving towards more automated tests to minimize manual effort. Automating the test content in line with the manual test steps and having traceability at test-step level is a tedious process. There aren't many tools available for it. In complex multidisciplinary projects, the subsystem tests, Integration tests, and System Tests have commonality and can be re-used for use case testing while keeping focus on subsystem functionality testing. If the test system is carefully designed for re-use by decomposing the test steps with objective functions as meta data with the approach of BDD/TDD that is platform agnostic, the metadata can be used across subsystems, Integration tests, and System tests for functionality, regression, stress testing without much of effort both for manual and automated tests. Reliable automation can be achieved by coupling the BDD/TDD decomposed test functions with the Fault recovery mechanism during catastrophic failures of the test systems to auto recover during testing and continue the regression tests for lengthy (e.g. overnight) unattended tests.

The paper explains how the product development team built the Unified Tests and automation framework which makes the reliable test content for re-usability and automation with inbuilt Fault tolerance system for recovery of the system during catastrophic failure for test execution.

## Biography

*Chandrashekhar is an Engineering Manager at Intel India (Bangalore) with extensive experience in product development, with expertise in system architecture, operating systems and embedded software development related roles. He worked on implementing CMMi, ISO to the organizations with six sigma approach on the quality practices. He is currently managing teams which are responsible for Firmware (FW) System validation and automation at Intel products across all market segments (such as phones, tablets, laptops and desktops.)*

*Manini Sharma is a Platform Software Quality Engineer at Intel, based in Bangalore, India. She works on software for Windows devices based on Intel Architecture. She's worked in a variety of roles in the past decade including Product Marketing, Software Development and Application Engineering. She's been focused on Software Quality for the past five years. She has an M.Tech. in Information Technology from IIIT-Bangalore and an MBA from IIM Kozhikode.*

# 1 Introduction

Validation is a key element of a product's lifecycle. Script-based validation has evolved in both software and hardware industries with multiple tool chains and test management systems. There are several systems and methodologies for test management and automation. However, there is scope for standardization of the process for moving from manual testing to seamless automation. This can be done using test steps as objects and methods, based on the specific industry or technology. The general tendency is to maintain test cases for manual execution and automation separately. During the execution of a project with multiple subsystems (being developed at the same time), the probability of test content getting changed during the lifecycle of the product is very high. This results in divergence between manual and automated test systems, if not monitored and controlled effectively.

In this paper, we apply the test content development methodology with standardized process using the technique of metadata creation. A unified test and unified user-defined metadata has been developed which helps in tight coupling of requirement changes to the test content and subsequent automation. The paper is structured as follows: in section 2 we elaborate on the problem of generating test content and automation, in section 3, we detail available methodologies on test case and automation management and the chosen approach. Section 4 is dedicated to the case study of implementation and the result. In section 5, we discuss the key benefits.

## 2 Problem: Test Content Management and Reliable Automation

The process of test content development, test automation and faultless execution of complex systems, involving unstable or under-development hardware and software, is a challenging task. The product design complexity, interoperability issues, component interdependencies, changing product requirements, results in the continuous maintenance of test content and automation scripts. Delays in development and frequent requirement changes invariably result in crunched schedule during the validation phase. The crunched schedule for validation of parallel product lines and market pressures, lead to compromise on quality of test content, reusability, review effectiveness, maintenance and automation scripting. How do we curb the churn in test content, bug escapes, coverage gaps, security vulnerabilities, automation reliability efforts, and test environment stability issues, and still save resources with maximum reuse across projects?

## 3 Methodology of Test content Management & Automation

There are many methods the industry follows to define test content and automation. The usage varies from maintaining the test cases in Excel or SharePoint to test management systems like HPQC/JAMA/JIRA. With the BDD Approach, the tests are directly written for automation and maintained as source code, written using Cucumber-Gherkin or Google Autotests (Google Inc. n.d.). While these approaches are useful for the qualified automation engineer to write tests and execute them, many validation teams maintain manual procedures (Cognizant 2014) and automated procedures (Geometric White Paper 2011) separately for execution, in case automation fails. This paper deals with effective usage of available BDD methodologies, making the test methods readable as plain English for the users of manual execution and also for reading test results. It is blended with object oriented paradigm (Quali Systems n.d.) for exception handling during catastrophic failures, while enabling test class and methods, built in with hash mapping techniques for test decomposition and re-use. This is achieved by decomposing the test content creation and automation into 4 subsystems:

- 1) **Unified Tests and Meta methods creation** with BDD as focus. This is equivalent to a class template in code. Each functionality is decomposed and grouped into a test class and its parameterized test methods with defined test syntax.

- 2) **Instant Automation Development.** These are formed with hash mapping technique using regular expressions to map/pick the right template of test class/method and generates the test script. Test content is written for various purposes like testing functionality, unit level tests, scenario testing, stress and stability using the test methods with instance of test objects. This module gives context-based syntax checks for writing the test content easily.
- 3) **Fault Tolerance System (FTS) for environmental recovery** - This system is for environmental recovery during systemic failures ensuring uninterrupted test execution. FTS monitors for any systemic failures, launches its environment recovery services in the background to enable seamless execution without manual intervention during overnight, or large size test executions. This includes re-setting of the system to factory mode/equivalent, re-installation of OS, firmware and initializes the test bed for re-execution of the failed tests.
- 4) **Standardization of test bed for continuous Delivery.** This is an Integrated Framework deployment setup of test bed for continuous delivery of the incremental changes in the product to get tested automatically by triggering the test through Build servers which monitors the code change (Check-in) and generates/selects the required test scripts get executed instantly for regression and functional tests.

## 4 Case study: Firmware Subsystem Automaton

### 4.1 Background

At Intel, multiple programs are executed in parallel and in incremental fashion. With traditional test development and test management, it was tedious to accurately identify test re-use. Also, the traceability of re-use became difficult when the test cases were cloned and modified to suit various program needs. System test designs are complex in nature for the hardware, firmware and software system as single product deliverable. Test automation traceability and maintainability also gets affected, impacting the productivity. With the traditional approach of test development and automation, the process is subject to low degree of re-use in the scenario of parallel and incremental projects. There was a need to have a unified standard, providing holistic improvement on test coverage, automation and seamless execution process with debugging and analysis, while not compromising on quality. A solution was needed which allowed test content convergence, effortless, scalable automation, stable and reliable execution infrastructure, with increased velocity.

### 4.2 Standardization of Tests and Test Framework development

At Intel the Firmware team realized the need of standardization of test content which can be turned into instantly automated tests. The focus area was to develop a Framework which ties the following:

- 1) **Unified Tests and Meta methods creation** with BDD as focus
- 2) **Instant Automation Development**
- 3) **Fault Tolerance System** - A mechanism to recover the test bed from catastrophic failures to continue the test and
- 4) **Standardization of test bed** for continuous Delivery

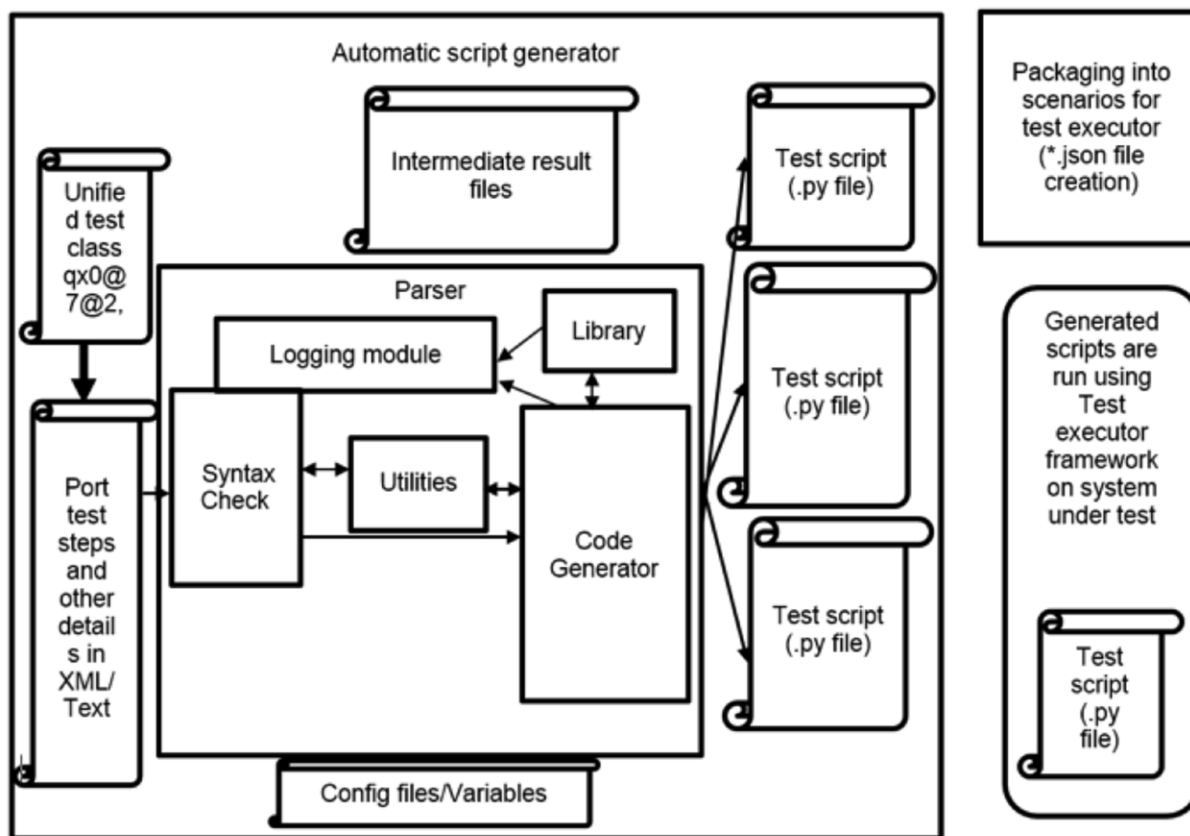
Test content development is done with BDD as focus and **Unified Tests and Meta-methods** are created by scrubbing through all the available test cases on each market segment and subsystem. The focus is on identifying the common test steps/procedures to create standard methods in each category of system features. These methods are defined in a framework or library which consists of defining the Method Name, Syntax for the method with parameters and Regular Expression for syntax checking. This makes it easy for test content writer and automation engineer to use the methods for auto complete of content/

scripts. Table 1 depicts the sample methods, syntax and the regular expression for syntax validation.

Test Method	Syntax	Regular Expression
<b>Boot to OS</b>	<b>Boot to &lt;environment&gt; {from &lt;source&gt;}</b>	( *) <b>Boot</b> ( + ) <b>to</b> ( + )(\bos\b \bedk shell\b \bsetup\b \bBIOS setup\b \bTBOOT\b \bubuntu\b \bmembx\b \bRAID\b \bFW RECOVERY\b)(( +) <b>from</b> ( + )(\bHDD\bSATA-SSD\b \bEMMC\b))?( *)
<b>Put system to S3 using pwr_btn for 30 seconds</b>	<b>Put &lt;System&gt; to &lt;Sx_state&gt; using &lt;Source&gt; {for &lt;time&gt; seconds}</b>	( *) <b>Put</b> ( + )(\bSystem\b \bClient-System\b \bHost-System\b)(( +) <b>to</b> ( + )(\bS[3-5]\b \bDeepS[3-5]\b \bCS\b \bDMoS\b \bCMoS\b)(( +) <b>using</b> ( + )(\bidle wait time\b \bKVM-USB\b \brtc\b \bpwr_btn\b \blid_action\b \bOS Power Policy\b)(( *)\$ (( +)for( +)\d+( +) <b>second</b> (s*)( *)

**Table 1** Sample methods, syntax and regular expression

**Instant Automation Development framework** is developed with a parser and code generation module which reads the test content and test steps, passes through syntax checker and hash mapping technique to pick the right template of test class/method and generates the test script on the fly for the expected behavior for the method and parameters used in the test scenario. Figure 1 depicts the Instant automation framework for unified test class which generates the test scripts automatically based on the test template and the input parameters/metadata. Automation code will be generated from the Instant Test framework by reading each step and using the library, test scripts will be ready for use as a script package.



**Figure 1:** Unified Test Framework for Auto Test script Generation

Figure 2 depicts the flow of dynamic code generation when the change happens in the test content. There are two scenarios (marked in Unified test class – Use case) of test content change and the related code change is highlighted in the sample code.

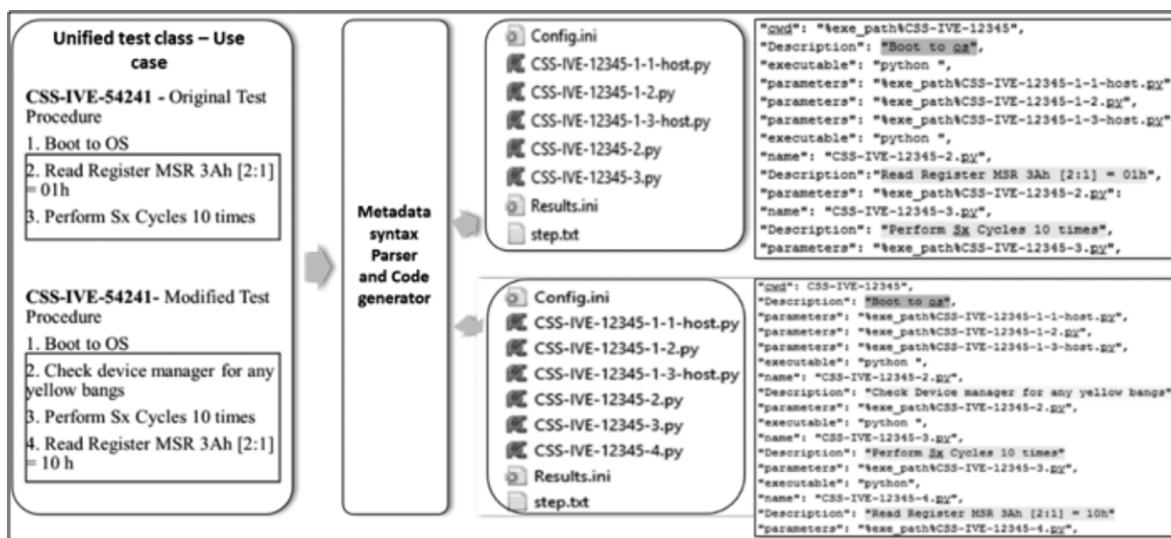


Figure 2: Code generation Flow

**Fault tolerance system (FTS) for environmental recovery framework developed** which interacts with test executor and identifies the systemic failure of the system (e.g. blue-screen-of-death (BSOD) error or system startup failures). During systemic failures, FTS launches its environment recovery services in the background to enable seamless execution without manual intervention during nightly, or large size test executions. FTS is a continuous monitoring service and has the capability of recovering the system by re-flashing the firmware and software through add-on devices like FW Programmer to the system under test (SUT). Along with the recovery of the system, FTS also collects the logs and information about the system failure, for easier debugging and creating the environment of faultiness to reproduce the issue as well as determine the root cause, helping ensure faster turnaround on product defect resolution. Figure 3 shows the FTS sequence.

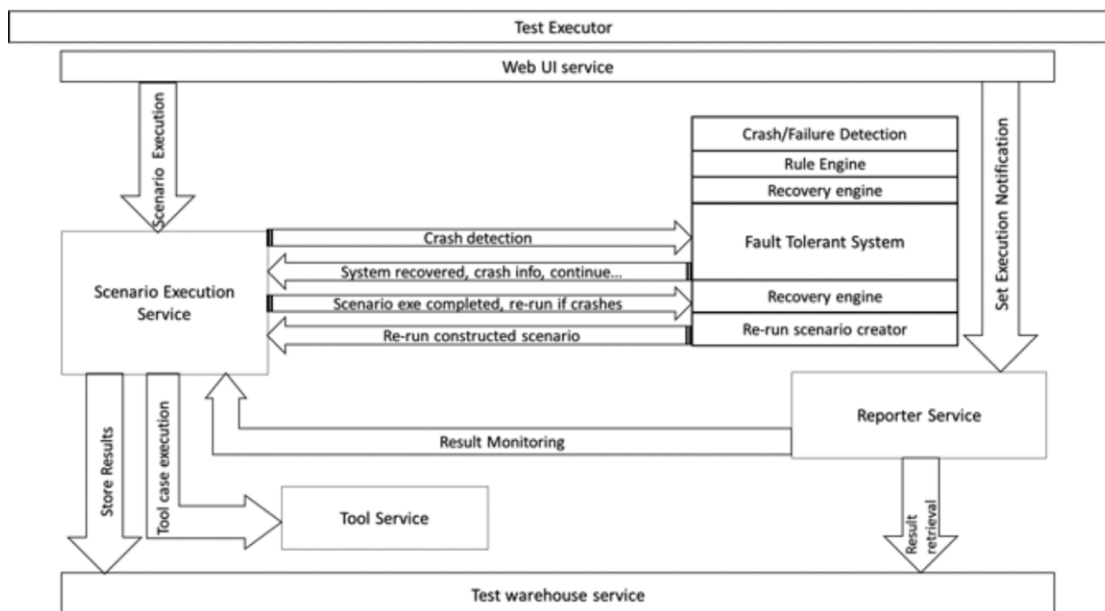


Figure 3: Fault Tolerance system Architecture

With the above framework developed, the last part is the **Standardization of test bed for continuous Delivery**. Below diagram shows the Integrated Framework deployment for continuous delivery of test content for execution. Test beds are paired with a Host monitoring system which acts as test agent, execution controller and Fault monitoring system. Test agents also connect with programmable HW device (relay, power switches) for any external electro mechanical action, like Power button control on/ off, switching between power devices and touch sensor tests. The system is also connected with the subsystem product source code through the build server for continuous integration.

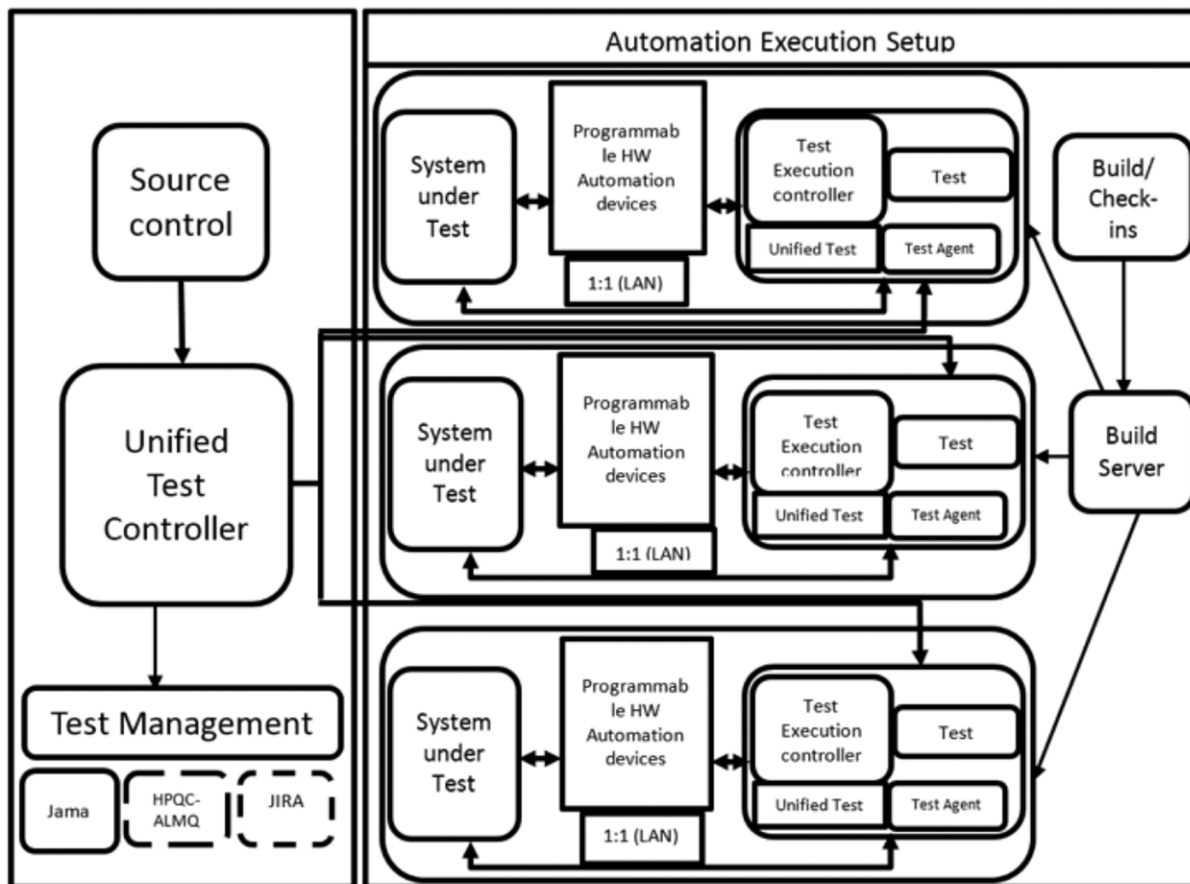


Figure 4: Unified Tests Automation Deployment Diagram

## 5 Key Takeaways and Benefits Derived

There were several key takeaways in the process of standardizing the specification for unified test framework:

1. Traditionally automation effort used to start after completion of test content creation. With this methodology, automation becomes instantaneous and can be used immediately.
2. Ramp up time for new resource reduced from 8 weeks to 2 weeks to understand the test methodologies. This happened because the focus was only on the smaller set of test objects rather than the entire suite of test cases.
3. Single library of test metadata helped cross-team synergy for subsystem and system validation with more than 60% reuse.
4. Fault tolerant system as a standalone mechanism for system recovery and telemetry recording was used by teams aiming at nullifying manual dependency during systematic failure such as Windows Blue screen error (BSOD).
5. Each of the unified meta-methods are associated with the dependent hardware and software tools as attributes. This helps in long term maintenance of the framework in case of any changes in the tools used.

## 6 Extending the application of Unified Tests

At Intel, this approach was first piloted with FW validation. Multiple improvements in the approach were implemented on the re-use and standardization of framework subsequently. The approach was then extended to System Integration and Validation teams to maximize re-use with minimal additions of test metadata. The benefit of re-use and instant automation reached up to 80% from the existing library. This methodology is planned to be extended further to multiple other subsystem teams to make an organization-wide test library, which can be used across projects and teams for instant automation readiness.

Application of this framework/methodology can be extended to any type of SW/HW/embedded program testing. Product development organizations can achieve significant productivity benefits across the organization. Having a common library of unified tests related to functionality, stress, regression, performance, integration and system tests can make the test organization to re-use and leverage across teams with common library. In the open source world, the same suite of test cases can be shared with the open source community to contribute similar source code to make standardization a solid test suite for the products.

## 7 Conclusion

The Intel product development team spent multiple weeks of effort to deal with high volume testing of client products across market segments. This inspired a search for better methodologies to optimize the test content with automation focus. A unified test content framework was piloted in a subsystem team and proliferated across other Intel teams to validate tests across. This was done while keeping re-use and standardization automation process in mind. This helped in saving of over a million USD in one subsystem team in a year.

Unified Test Class, User-defined Metadata and FTS make a powerful solution to have a tightly coupled automation focused test development and execution which can be used as a standardized software testing product.

## 8 Bibliography

- Cognizant. 2014. "Transforming Test Automation: An Unconventional Approach to Shift-Left by Removing Scripting from the Equation." <https://www.cognizant.com/>. Accessed July 2016. <https://www.cognizant.com/insightswhitepapers/transforming-test-automation-an-unconventional-approach-to-shift-left-by-removing-scripting-from-the-equation.pdf>.
- Geometric White Paper. 2011. "Scriptless Test Automation." <http://geometricglobal.com/>. February. Accessed July 2016. [http://geometricglobal.com/wp-content/uploads/2013/03/Geometric\\_Whitepaper\\_scriptless\\_test\\_automation.pdf](http://geometricglobal.com/wp-content/uploads/2013/03/Geometric_Whitepaper_scriptless_test_automation.pdf).
- Google Inc. *Autotest for Chromium OS developers*. Accessed August 2016. <https://www.chromium.org/chromium-os/testing/autotest-user-doc>.
- Quali Systems. "Object Oriented Test Automation." <http://www.webtorials.com>. Accessed July 2016. [http://www.webtorials.com/main/resource/papers/QualiSystems/paper1/Object-Oriented\\_Test\\_Automation.pdf](http://www.webtorials.com/main/resource/papers/QualiSystems/paper1/Object-Oriented_Test_Automation.pdf).

## 9 Acknowledgements

The authors would like to thank the reviewers for their valuable comments and suggestions to improve the quality of the paper. We also thank our organization, Intel Corporation, for the support and funding for this paper. We especially want to thank

- Keith Stobie – PNSQC Reviewer
- Patt Thomasson – PNSQC Reviewer
- Soumya Mukherjee – Intel Corporation
- Sneha Pingle – Intel Corporation
- Vivek Malhotra – Intel Corporation
- Christopher Hall – Intel Corporation
- Pramod Mali – Intel Corporation