Testing the World's Apps

Jason Arbon

jarbon@gmail.com

Abstract

How do you test all of the world's apps? Over 10 million mobile apps and 5 billion web pages are updated and need testing. Today test automation efforts aim to test just one or a small subset of applications. Most of these apps and test cases are similar, yet each app team reinvents and re-implements the same test cases from scratch. Testing even a single app is difficult, slow and expensive, non-deterministic and often unreliable. Building test automation that executes reliably on all the world's apps requires rethinking all aspects of test automation. Three major problems needed to be solved to test at this scale:

- Reuse of test artifacts and logic
- Reliable test execution
- Contextualization of test results

This paper will help app teams and software testers understand the significant implications of testing at this scale, what testing at full scale looks like, and apply those lessons to your own test automation and infrastructure.

Biography

Jason Arbon is the CEO of test.ai, which has raised over $17 million in investments, and is delivering an AI-first approach to software testing. Jason previously held test and engineering leadership roles at Google (Search and Chrome) and Microsoft (Bing, Windows, SQL) where he tested large-scale neural network training systems. He was also director of engineering and product at Applause.com/uTest.com. Jason is a co-author of How Google Tests Software and App Quality: Secrets for Agile App Teams.

1. Introduction

Modern software test automation efforts target a single application, and teams often rewrite the same test case for each platform (iOS, Android, Web). This approach hasn't scaled well as most apps have no regression automation, and those that do have tests struggle to maintain a small amount of coverage. Engineering and organizational changes such as agile, continuous integration, DevOps, and better development tooling add additional pressures to test automation development. There is a different approach, and that is developing test automation at the scale of all apps.

Together, a few insights show that testing at this scale is now possible:

- AI / Machine learning enable test cases independent of the application's implementation.
- Apps share many common user interactions and flows.
- Reliable test execution infrastructure is now possible because of decreased compute costs.
- Test results are more insightful when compared to similar applications.

This paper details the implementation of a reference system that can test at the scale of all the world's applications, outlines the benefits of testing at this scale, and considers the impact this will have on the test automation profession and software quality. There are many implementations of such a scalable testing system. This paper describes the design, implementation, and lessons learned building test infrastructure that scales to test the world's apps at test.ai.

This paper also addresses UI-based functional regression testing. There are, of course, unit tests, integration tests, monitoring, etc. but at the end of the day the most important thing is that the software works for the human being controlling it via a user interface. This type of testing is also traditionally the most difficult and expensive to create, maintain, and scale, which is why it was chosen as a reference system. It is an exercise for the reader, or a later paper, to describe how these very same methods can be adapted to other aspects of testing.

2. Testing Apps Individually

The state of the art in automated testing focuses on one app at a time. Modern app test teams don't have the budget to thoroughly test their own app, let alone develop automation for other applications.

Testing apps one at a time is often a brutal experience, fraught with peril and full of anti-patterns. Test automation often breaks exactly at the moment when the team needs it--right after minor changes in the application. App teams find themselves in a cycle where they start manual testing that doesn't scale, build out automation that ends up in a boondoggle of maintenance, close down their automation efforts, return to manual testing, and inevitably automate again with new leadership, new tools, and the same dreams. Even the smartest teams at Google, Apple, Microsoft, and Facebook all have the same issues, but they can afford to keep trying to test their apps over and over again with little wins here and there. The tragedy is that the vast majority of apps today have no testing. Most app teams cannot afford a tester, or even know how to get started. The realization that so few apps have any testing at all motivated much of the work below.

There have been attempts at larger, cash-rich, companies that make 10's or 100's of applications to re-use some test automation code across their fleet of apps. But that reuse is often scarce, limited to shared components, in the form of code, and creates its own new set of problems to maintain single code bases for multiple apps in different stages of shipping. The most mature testing efforts in the world often turn to test frameworks to scale the problem and share them with other engineers. These test frameworks don't have the test logic in them--they only speed up the ability to generate more test code or add new capabilities for test authoring for verifications. These test frameworks make it better to generate test coverage for a single app, not for all apps.

Testing applications one at a time is the current state because of the nature of testing. Software testers are often risk-averse--introducing a new test framework or methodology distracts from testing itself and is often seen as a risk in itself. What tester would have the audacity to pitch the idea of testing apps other than their own? Testers are often working in silos within their own companies, let alone collaborating cross-company. Testers are also late adopters when it comes to changes in technology--very few testers are playing with machine learning these days. Strangely, test automation vendors who should be motivated to build reusable components are also motivated to build tests for individual apps because that is their business model--one app at a time, and billing for human labor increases the top line. Lastly, the most ambitious engineers are usually focused on product, not testing, so that energy and ambition have been lacking in the testing space. Testing has been viewed as a necessary evil and cost center--not a place to truly innovate. Ultimately, it is not surprising test automation has been trapped in a linear, app-by-app mode considering some of these uncomfortable observations.

3. Scaling with AI

The application of AI to the test automation challenges makes all this solvable. AI is a set of Machine Learning techniques, to teach a machine to replicate human behavior. Here, the human behaviors to mimic are recognizing elements in an application and knowing how to interact with them in sequences that represent test cases. Just like humans, machines can be taught to execute test cases.

Traditional software test automation requires a developer to discover a unique way to find an element in the application. Elements are defined via custom identifiers, CSS selectors, XPATH expressions, simple image recognition, or hard-coded into the testing logic. This takes time, has to be repeated for each application, and when the application changes slightly the code needs to be updated to reflect the new implementation. Additionally, the sequencing of test steps is often hard-coded in a procedural programming language, so if the flow of the application changes, the code also will break and must be updated. Test automation today is expensive, slow, and laborious. With tens of millions of apps, billions of web pages, there isn't enough money or engineers in the world to test all this software with the traditional approach.

AI-based test automation doesn't suffer from these issues. Most importantly, the AI-based approach enables reuse of test automation artifacts.

3.1. Element Identification via AI

AI-based element identification is as simple as training a system to recognize cats versus dogs in photos. Here the machine is shown images of a login button, search boxes, password fields, etc. until the machine, like a human, and can readily recognize a new element in an application it has never seen before. The machine looks at the screen, just like a human tester and can recognize all the interesting parts of the application. No need for magic IDs, CSS Selectors, or XPATHs.

The robustness of this form of element identification derives from the fact that the 'AI' processes many different attributes of the element, and it has been trained on many different variations of that element. Training a network to recognize say a search_box is done by passing hundreds of element attributes such as height, width, position, ID, average color, CSS, number of edges, text, etc. into a neural network and telling the network 'this is a search_box'. Much like a human unconsciously sees many of these attributes, not just one, to identify it as a search_box. Also like a human, the neural network has seen hundreds, even thousands of different search boxes and learns what range of values search boxes typically have, and what values are often correlated with each other. So, if the search box moves from the bottom of the screen to the top, or changes color, or gets wider, the neural net will still recognize it as a search_box if the new value seem to still be consistent with a 'search box'. Traditional element identification breaks if only a single feature of the search box changes (e.g. its ID, x/y, CSS, or HTML hierarchy). The AI approach to element identification is far more robust to changes in the element, meaning lower maintenance costs as the application is redesign or re-implemented.

Most notable is that this AI method of element identification means that for common elements, this training need only happen once. This means there is no need for the test developer to identify the CSS or XPATH for each element in an application--the AI already knows that it is a search button in a new

application it has never seen. So, speed of element identification drops to near zero, and the identification is far more robust to changes, reducing maintenance costs to near-zero.\

3.2. Step Sequencing via AI

AI-based test sequencing is a bit more complicated implementation of machine learning, but it still behaves much like a human executing a test case. When a human is executing a test case manually, they have several subgoals. An example test case for a search engine might be:

- Find the search text box
- Enter in the word “Gradient”
- Click the search button
- Verify that there is a search result item with the word “Gradient”’, in it.

It is worth noting that humans also intuitively check for things like usability, look and feel, and sanity checking behavior based on a larger context of human experience. AI can approximate much of this aspect of human testing as well, but is not the focus of this paper, which is scaling UI functional regression automation.

Each of these individual steps is a simple goal. The reference AI-based system leverages a mechanism called reinforcement learning, where the machine attempts to walk around in the application, and it is given a reward when it ‘accidentally’, but correctly, clicks the search button for example. After thousands of learning attempts, the machine builds up heuristics of what to do to accomplish each test case step. The machine can now take an application it has never seen, and execute the same test logic against that new application. This approach requires no cod and is re-usable across applications and platform. This approach also requires no prior knowledge of the application design or implementation, so it is robust to changes in application design. Just like a human tester, the AI is flexible and learns over time.

Not only does an AI-based approach mean faster test development and less maintenance, it means a test case need only be written once, and the test can be executed on another app where that test case should be relevant.

4. Scaling Test Case Definitions

With an AI-based approach to execute test cases, we need to design the tests for the automation to execute. For example, we need a way to tell the machine to ‘search for “Gradient”. Work is underway to teach AI-based systems how to create test cases themselves, but the human mind is still far better at test design. The humans need a quick way to compose these tests.

The AIT (Abstract Intent Test) schema is designed to be a human and AI/machine-readable test case format. The format deliberately doesn’t support or need, the complexity of magic IDs, CSS, XPath, etc. The format also only captures the ‘Intent’ of the test case and contains as little information as necessary to execute such a test case. AITs only deal with sequences of interactions defined at the ‘label’ level. Where a label is something like ‘search_button’, which is understandable to a human and readable by the AI-powered automation system. AITs are easy to construct in a text editor in a simple Gherkin-like language or an equivalent JSON format. The reference implementation also has a simple drag and drop User interface to compose test cases using pictures of components of the application. AITs can also be programmatically generated. Regardless, all the test case logic is embedded in the simple, human and machine readable AIT file.

Designing scalable test cases requires a different mindset than traditional test case definitions. Traditional test case design typically has a few ‘setup’ steps, to get to the part of the application where functional behavior or user-interface characteristics checked. Traditional test cases not only have to contain the steps needed to execute a test case, but they are also specific to the application’s current implementation.

4.1. Test Design for Reusability

Designing tests that can be reused across many apps take into consideration. An approach taken in the reference system is to group applications with similar functionality and implementation. For each cluster of applications, define the standard test cases as AIT tests, paying particular attention to ensure the underlying logic and flow of the test case is as general as possible, so those steps are logically similar across all the apps. With the 'search for "Gradient' test case example, the AIT test logic is general enough to be executed on all top search applications. The AIT doesn't contain any app-specific nuances of how to navigate to the search text input box--e.g., In applications, the search box is on the initial loading page, in other apps it only appears after other dialogs or screens. The key to writing scalable AIT test cases is to ensure they are app-agnostic and look for as much 'reuse' across as many applications as possible.

4.2. App-Specific Test Data

During test execution, the AITs are parameterized with app-relevant test data input. AITs can contain a specific string such as "LeBron James", but they can also contain test input or validations on the output of the form "<basketball_player>". The system maintains a mapping of these category names, to specific instances such as "Stephen Curry", or "Kevin Durant", or "LeBron James". The test execution system identifies the type of application at run-time, looks at parametrizable values in the AIT, and replaces those values with domain and application-specific values. The human editorial goes a long way toward ensuring test data is relevant at the individual app level.

This test input can impact later steps and the validation to be performed. So, later steps that also refer to <basketball_player> are all set to the same value. Meaning you can search for <basketball_player> then verify <basketball_player> appears somewhere on the search results page. Both input and output values will have the same value. Also test input may be labeled with a 'scenario' tag. This means that particular test data is relevant for a given 'scenario'. When parameterizing later test step values, only values that are also relevant in that given scenario are used. This is a quick, declarative way to ensure coherency in the test execution.

These tests are called the 'Standard Tests'. These are tests to be executed on all similar apps. Not only do we get a high return on the effort of designing test tests as they are executed on many apps, but this also has the benefit of relative quality assessments. More on that in the reporting section below.

4.3. Scaling Exploratory Testing

Having a list of reusable standard tests across large numbers of apps is great and reproduces rote, scripted testing. Another common form of testing is 'exploratory' testing. Essentially the human tester manually walks through the application and 'plays' with the application and observes basic behavior for correctness. The reference system not only executes the Standard Tests, but it also tests the application in an automated, and exploratory way. This is called "AIT Chaining'. The process is :

1. Identify the current app screen and elements
2. Search through a database of common, small AIT test definitions that match the label of the screen or element on the page.
3. Execute that general test case
4. Report Pass/Fail for that test
5. GOTO 1

This AIT-chaining allows for reuse of test cases for common subsections of applications. For example, many applications have similar sign-in flows where an invalid email address should block sign in flow. If you try to use the malformed email address of '@foo.com', the flow can be expected to stop. The AIT for a test case definition for this would be:

1. When on login_page,

2. Enter '@foo.com' into the email_address field',

3. Click 'login_button'

4. Verify still on 'login page' (we should be stuck here)

Many apps have similar sub-flows within them. AITs like this can be easily authored, and then when executing in an AI-Chaining mode, relevant test cases can be pulled from the repository and executed. The humans here still provide the test case logic, but the machine can intelligently and automatically select and execute them at scale on any application.

4.4. Scaling Authentication

How to test parts of applications that require a signed in user? Again, just a little human editorial work goes a long way. Functionally, the 'username' and 'password' values are passed into the application and executed like other AIT tests which use those values on things in the app that match the labels 'username' and 'password.' How do we the usernames and passwords for the words top apps? Two highly technical approaches to getting this login information in the reference implementation:

1. The team editorially goes down the top app lists--creating test accounts for all apps.

2. A quick email or ping on LinkedIn can often get a company to share a test account with us as we are adding to their testing coverage as a byproduct and are interested in seeing the results.

Interestingly, an engineer would assume that testing at this scale would get our infrastructure flagged as bots with so many login attempts. In practice, this hasn't been a problem. When apps flag the bot account, it is just an excuse to reach out to the team and share our findings--and get the test account blessed as a testing bot.

4.5. Scaling Test Design Summary

The key to defining human-intelligent test case definitions that can scale to the world's apps is re-use. Even if it takes a little bit of time to curate and manually define great test cases in a format such as AIT, the fact that they are quick to define, and there is no need to worry about the automated execution and reporting means testers can focus on what they do best--design great tests. With that efficiency, and the above test design approaches, a basic set of intelligent test cases for the top 10,000 applications is doable with the effort of just a few human testers.

At this point, the astute test or engineering mind will be thinking of all the corner cases AITs cannot support yet, or realize that you cannot write a test case that checks with a database to verify the stock price in the application is the correct value. This is all true! Scaling has its tradeoffs and benefits.

As an aside, it is worth noting that the feeling of writing a test case that can be run on hundreds of apps, even apps you've never seen, is a great feeling as a tester.

5. Scaling Infrastructure

Many teams might have a lab of between 5 and maybe hundreds of test devices. When you are testing at the scale of all the world's apps, all the problems normally associated with test labs grow as well.

Most test labs, whether machines laid out on a conference table, or in a modern, remote cloud-based test environment suffer from a few common challenges: flaky tests and limited capacity. Much of these problems come from engineering optimizing for the cost of computing, not person hours or reliability. When testing at large scale, these issues are too numerous to investigate. We turned the most common test lab design assumptions upside down in what is called 'SuperScaling' in the reference system

5.1. One-to-One

Most test labs are configured to have a single ‘controller’ machine drive as many test devices as possible. This costs money as controller machines can cost $1-$2 thousand dollars each. Labs can get up to 16 test devices or emulators per single controller machine. This configuration, however, leads to contention for network, CPU, and memory across all the devices. This contention leads to test steps or page loads timing out as the test logic or network is shared with the other devices. The scaling infrastructure instead uses only one controller per test device. This ensures there isn’t any contention for the controller, resulting in many, many fewer test step timeouts. This does cost about 15 times as much money to deploy, but that is still less than the cost of a single engineer to debug the infrastructure or deal with false positives across thousands of applications. The fundamental rule discovered to scale test infrastructure is that machines are cheaper than humans today. Modern test labs don’t realize how much human cost is associated with cheap infrastructure.

5.2. Networking

Most test labs share networking capacity across multiple devices, often tunneling through the controller’s single network connection. This means unpredictable networking which can cause false positives and also means that performance metrics won’t be consistent run over run. To SuperScale, we have a dedicated, guaranteed 10MB connection for each controller and each test device. Yes, this wastes a lot of network capacity, but at scale, this is far less expensive than having false positive test results and paying humans to investigate issues.

5.3. Local Procedure Calls

Many of the more popular test frameworks have the test code executing at a remote machine, and test case steps make a remote call over HTTP to the test lab where the test machines live. The internet is surprisingly unreliable, especially at scale. The time spent waiting for each remote call over HTTP, and the occasional lost request leads to test execution timeouts, invalid object (e.g., the object referenced in test step can change state while the command is traversing the web). With SuperScaling the solution is to write our communication layer to the application and put the test driving logic local, on the same computer as the simulator to device.

5.4. Pristine Machine State

Lastly, most lab machines are re-used across test cases, sometimes even test runs. This can lead to silly issues like running out of disk space, accumulated memory leaks, software state ‘cruft’ accumulation, all leading to unreliable test execution and the need for human intervention. The SuperScaling model again does the inverse: controller machines (e.g., MacMini’s) are re-imaged at the ROM level between each single test execution. The machine images are stored on nearby storage area networks (SANs) with dedicated network lines to not interfere with any application network traffic. Yes, this costs money as the machine cannot process work while they are re-imaging. But, the SuperScaling key to scaling and reliability is having these systems reliable without human intervention and minimize any complexity due to machine state.

Essentially, the SuperScaling test infrastructure design turns each device into its own isolated mini-lab. This lab design philosophy results in >.998 reliability in test execution, which enables an engineer (no DevOps) team to support the test execution of tens of thousands of applications.

6. Scaling Test Reporting

To scale test case reporting to millions of apps, and billions of web pages, the reference system has built a non-traditional reporting infrastructure. Most importantly, testing at this scale also enables a whole new way to view and judge application quality.

6.1. Delayed SQL

Traditional test reporting post results of test cases to a SQL server, and at run-time, the reporting UI runs SQL queries to report on the data Test case pass-fail results are also often just that, a little bit of text. Traditional reporting also only has one or perhaps ten users querying the data at any given time. The AI-driven testing bots, however, produce a GigaByte or more of results data per test run. Generally with scalable reporting design, things are kept as long as necessary in flat JSON files. These JSON results are then post-processed and saved in cloud storage. A separate process parses all these JSON files and pushes that data into ready-to-serve schemas in a SQL data store to enable real-time complex querying to support engineering, internal reporting. Keeping the data in JSON formats as long as possible, and preprocessing the data into standard report formats before pushing it into a datastore helps avoid any data store contention and dramatically reduces the compute time and latency of traditional dynamic test report generation.

6.2. Search

Traditional test reporting post results of test cases to a SQL server, and at run-time, the reporting UI runs SQL queries to report on the data Test case pass-fail results are also often just that, a little bit of text. Traditional reporting also only has one or perhaps ten users querying the data at any given time. The AI-driven testing bots, however, produce a GigaByte or more of results data per test run. Generally with scalable reporting design, things are kept as long as necessary in flat JSON files. These JSON results are then post-processed and saved in cloud storage. A separate process parses all these JSON files and pushes that data into ready-to-serve schemas in a SQL data store to enable real-time complex querying to support engineering, internal reporting. Keeping the data in JSON formats as long as possible, and preprocessing the data into standard report formats before pushing it into a datastore helps avoid any data store contention and dramatically reduces the compute time and latency of traditional dynamic test report generation.

To navigate this mass amount of data, where engineers aren’t even sure which apps are tested, or when, the reference system also has a search engine, indexing all test cases, apps, and test results.

6.3. Benchmarking and Competitive

The most exciting aspect of reporting on testing at this scale is the cross-application analytics. Most app teams can barely keep up with testing their own application--they don’t have time even to imagine testing their competitor's apps. When testing at this scale, with standard and common test cases, it is to pivot the reporting in a few ways:

- Competitive Performance: The performance of clicking a ‘search_button’ across the top web search apps can now be compared against each other. For example, we can easily tell that Walmart’s app is slower to log in, but has a faster search experience compared to Amazon’s app.

- Flow Comparisons: Since AITs describe a user intent, it is possible to see how other applications accomplish the same intent. One app may have extra steps due to better design or may have features that another app doesn’t have.

- Feature-Level Benchmarks: If an application’s performance for facebook OAuth login is slow, but they know it is also slow in another app, they can prioritize their time on other performance issues in the application since they likely don't have control over it. The performance of common actions in apps can now be compared to the ‘average’ in all apps.

6.4. Device, OS, App Store Testing

Individual app teams find it difficult to automate the testing of their app. But there are also test teams out there with almost impossibly large testing problems:

- App stores: App stores strive to ‘test’ every app that is submitted from developers to protect their users and brand from low-quality applications. App stores today, generally have a mix of human manual

testing and random monkey tests to look for basic crashes and other issues. Now App stores can test and verify the functionality of 'all' apps at scale.

- Operating Systems/ Browsers: The makers of OS's and browsers have the accountability to make sure applications work great on their platform. Today's approach to test automation means only a few apps can be tested to catch reliability, performance, and functional regressions when the operating system updates. With scaled testing, more intelligent quality reporting can be made to compare OS vs. OS, or Browser vs. Browser.

- Hardware Manufactures: Hardware also has an impact on application performance and sometimes functionality and reliability. Testing at scale means it is now feasible to verify how hardware changes impact applications. Finding issues, or just verifying that things still work, would be significant progress.

- Carriers: Cell phone carriers have to verify that now hardware, operating system versions, and applications won't adversely impact their network and services. Scaled testing can help verify the fully deployed system will behave from the app-level down.

The reporting on massive amounts of application test results can reveal the test results for lots of individual applications. This reporting also gives insights into:

- The relative quality between applications, can measure quality across all apps

- Regress new operating system versions

- Verify the correctness of fully deployed end-to-end carrier tech stacks

AI-driven testing at this scale enables new measures of quality not possible before. Performance, crash rates, even user flows can now be compared across applications and benchmarked. Most importantly, this approach to scaled testing means every team, even teams that cannot afford dedicated testing resources for automation, can benefit from the value of test automation.

7. Team Structure

Interestingly, when developing the reference system, the structure of the team was the most surprising aspect. The original thinking was was to bring the best test automation engineers on the planet together and just build it. It turned out that most test automation engineers were the very ones who had honed their skills on testing one or at most just a few apps, and they came with all their inherent biases. We also found that most of the problems were classical AI / Machine Learning problems that just happened to be in the service of a testing problem. Mid way we realized that what we had really build was a hybrid of a search engine and machine learning platform. The system is a search engine in the fact that it is crawling large numbers of apps, indexing all their content, scoring the results, and measuring quality in statistically and in aggregate. It is also a machine learning platform in that it has specialized tooling for labeling data, expressing rewards for the reinforcement learning, and a data pipeline that mass produces global and per-app classifiers. If we knew how hard it would be, we may not have started :)

The team construction that ended up being important:

- Machine Learning and Statistics PhDs focused on the ML

- Engineers with Experience in Search Crawling, Indexing, and Serving

- Test Automation Engineers experienced in testing at scale (e.g. Chrome, Search)

- Test Design Engineers well-versed in the breadth of testing problems, platforms, familiar with all the great test designs, and can recognize the anti-patterns of test design.

- Experienced Designer for Data Driven UX

- Senior Front End Engineers to build data-heavy, complex, performant UI

- Experienced Infrastructure engineers to design robust, distributed and scaling data pipelines, execution and reporting.

Interestingly, most of the team members aren't testers or automation engineers. Testing at this scale is really a combination of two product teams: Search and Machine Learning Infrastructure.

8. Summary

This paper has walked through the challenges of testing applications at scale, explored solutions, and discussed new benefits that this approach can bring to software quality. Testing at scale, with test case reuse, isn't just 'better, faster, cheaper', it impacts the role of testing, the overall quality of software, and given that apps power much of our daily lives, it might just make this a better world.

References

Special thanks to Jeff Nyman, Wayne Roseberry, Michelle Petersen, Joe Mikhail, Kevin Pyles, David Morgan, and Carlos Kidman for their insightful edits and suggestions.

Abstract Intent Test (AIT) Specification. https://goo.gl/3YMNcG (accessed Aug 29, 2018).