Effective CI Using Automated Quality Checker

Ruchir Garg

Red Hat

rgarg@redhat.com

1. Abstract

Continuous Integration is the need of the day for dynamic software, where time to market is the key. To achieve this, it's necessary that all incoming code Merge Requests (also known as Pull Requests) meet the quality criteria and doesn't break the master build. These checks cannot be only manual, and hence all repetitive steps need to be automated. This paper talks about automated checks that ensure that the incoming code meets the recommended coding guidelines, contains enough unit tests, references the feature it adds or the issue it fixes, maintains backward compatibility, contains documentation (if needed), etc. The comprehensive list of possibilities is dependent on the criteria defined by the team. Failure to meet such criteria results in rejected merge request. Once these checks pass, the reviewer has to only review the code and the artifacts of the automation manually. This cuts down on the review time drastically and prevents wastage. It also helps to keep the quality quotient high and the software always ready for continuous deployment.

2. Biography

Ruchir Garg is a Principal Engineer at Red Hat, where he is working on creating a new developer experience for enterprise developers. He has previously worked as a security consultant and as an automation architect, developing brand new test automation frameworks. Ruchir has spent more than 15 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.

3. Introduction

Dynamic software with frequent release cycles are dependent on Continuous Integration (CI) and Continuous Deployment (CD). In the cut-throat world of fierce competition, a delay in the release cycle due to an ineffective CI or a CD can negatively impact the businesses. In this paper, we'll focus on CI and discuss various ways of making it effective by leveraging automation and reducing the manual steps to the minimum. We'll also discuss how data generated by a CI system could be used as quality metrics.

4. Opportunity

Making CI effective, though most are already doing it, is a challenge. An effective CI system, to a large extent, depends upon automated quality checks. These checks should not only ensure complete regression (reactive), but should also ensure that the new code is not adding to the technical debt (proactive). Most automated quality checks focus excessively on the former, but ignoring the latter would eventually lead to a situation where regression will forever play catch-up with new code.

The proactive approach, too adds value but only when its automated, or else it may slow things down. For the scope of this presentation, let's refer to this automated CI plugin as the ***checker***.

5. Solution

5.1 Avoid technical debt

All new code must come with its own set of new tests, else it adds up to technical backlog and increases testing gaps. The *checker* must automate this check and take necessary action in case the minimum conditions are not met. let's consider the following few cases:

When: No tests written for the code being submitted.

What: The code merge request is Blocked and is also marked as such.

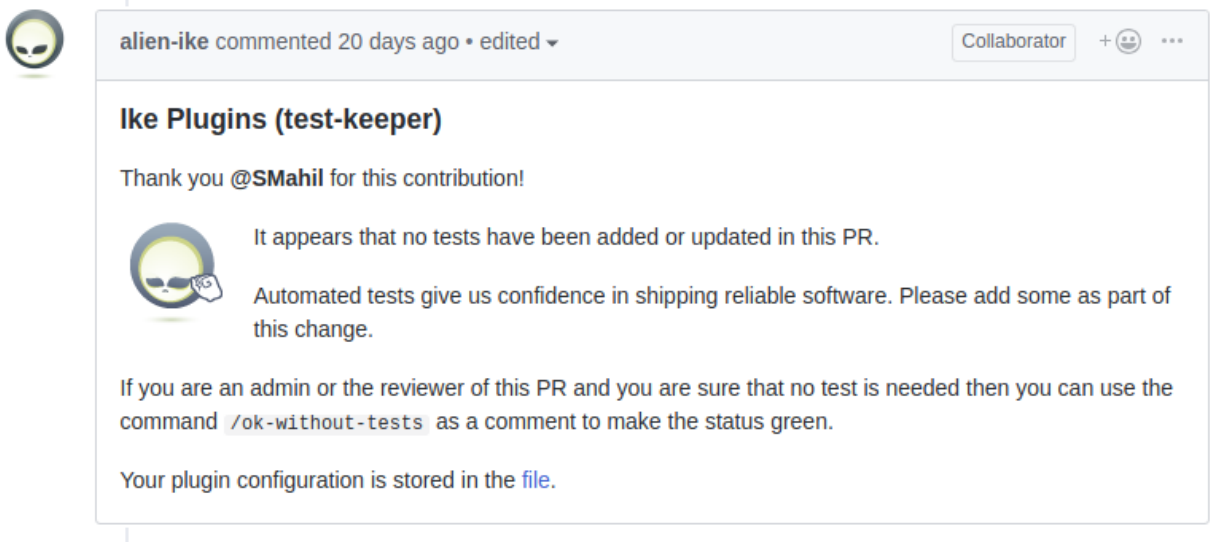

Figure-1: Checker blocking a merge request due to missing tests

How: This could be achieved by scanning the files submitted as part of the merge request and identifying the files added that represent tests. For example, assuming a project source directory has all tests nested under the "tests" directory, so additions to files under that directory could be assumed as new tests.

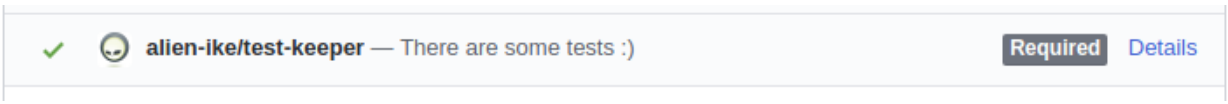

Figure-2: Checker marking the request when the author adds some tests

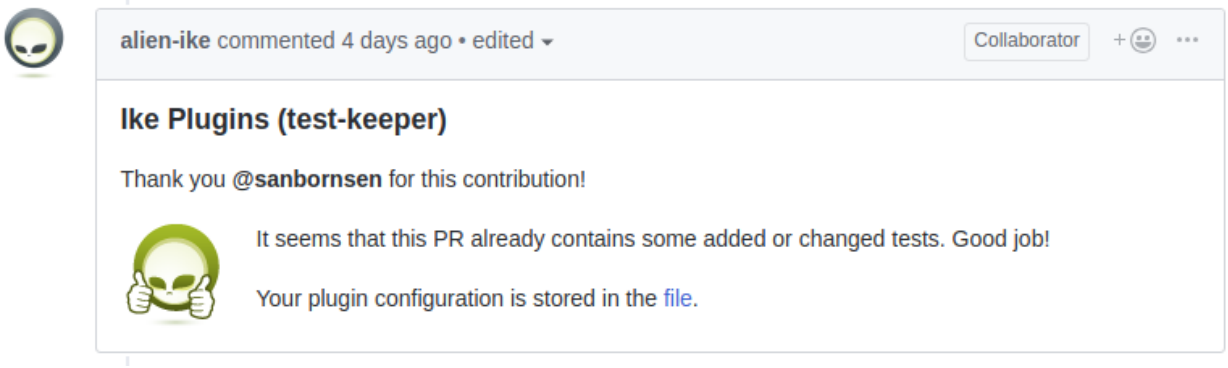

Figure-3: Checker lauding the author when the code is submitted along with the tests

Why: This process enforces a habit of writing tests along with production code. It ensures that people don't spend precious time reviewing incomplete and possibly dysfunctional code. It also makes regression easy to manage and hence increases code maintainability.

5.2 Don't break the working code

While it's important to ensure tests are written for new code, it's equally important to ensure that the old code continues to work as before. No amount of fancy new code can justify a breaking change at the customer's end. The CI system must automate this regression based on existing tests:

When: The new code submitted breaks an existing test.

What: The code merge request is Blocked and is also marked as such.

How: Configure the CI system to automatically run the existing tests (unit, integration, functional) against the build that is generated using the new code. Let the build fail on a test failure.

Why: This process ensures that the new code doesn't destabilizes the existing code and the current user experience is maintained. This check provides a safety net to engineering and instills confidence to roll out code more frequently without worrying about the outcome.

5.3 Maintain Code Coverage

Now that it's ensured that all new code is submitted with additional tests, we also need to ensure that the new tests added are sufficient too:

When: A fairly large amount of new code is submitted with a single new test.

What: The code merge request is Blocked and is also marked as such.

How: Configure the *checker* to automatically calculate the code coverage metrics. This could be done using tools like Codecov[1] or Gcov[2]. A simple goal could be that adding new code should not bring down the code coverage percentage any lower than before. The checker can compare the coverage numbers

1 https://codecov.io/
2 https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

and then take appropriate action.

Why: This ensures that the new code not only adds new tests, but also adds sufficient amount of tests. With this system in place, it would be difficult to game the system by adding minimum tests and getting away with it.

5.4 Ignore non-production code

Not all merge requests carry code that runs in production environment. Examples of such requests could be automated tests, non-production configuration changes, documentation, etc. Such requests could safely be ignored by the checker:

When: A new merge request is made that only contains test scripts.

What: The code merge request is Ignored by the *checker.*

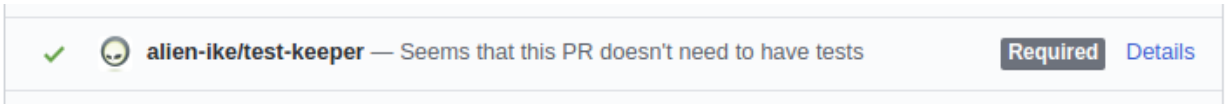

Figure-4: Checker skipping a merge request containing only non-production code

How: Configure the *checker* to specify directories that contain only non-production code. Commits to such directories would be assumed to contain non-production code and hence, would be ignored.

Why: This ensures that non-production code is spared the unnecessary scanning as the process to treat such code could be totally different than production code.

Note: It is quite possible that, even with all the check customization, there could be merge requests that carry production code, but doesn't require additional tests (e.g. config file changes). In such cases using a keyword, reviewers and administrators can bypass the otherwise mandatory quality checks.

5.5 Manage the process

Reviewer's accountability: A merge requests submitted requires a review from a peer, before it gets merged with the master branch. While we've been talking about the authors responsibilities, the reviewer, too has its own responsibilities. The reviewer needs to provide feedback within a stipulated time. A feedback provided towards the end of the sprint is undesirable as it would be then too late to make the changes and the author might be hurried into delivering the fixed code.

Work-in-progress = Ignore: Any merge requests that are marked for work-in-progress are to be skipped from the quality checks or peer review, as they are still not ready. The checker can take care of this by looking up strings like *WIP/work-in-progress* either in the title/description/labels.

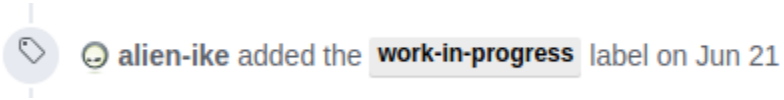

Figure-5: Checker adding a "work-in-progress" label to the merge request

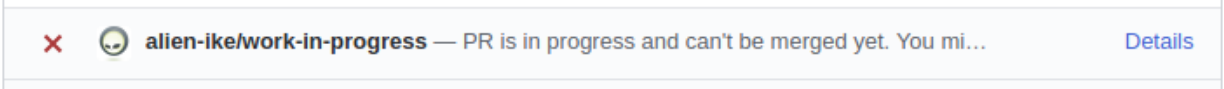

Figure-6: Checker blocking the merge request unless the "work-in-progress" label is removed

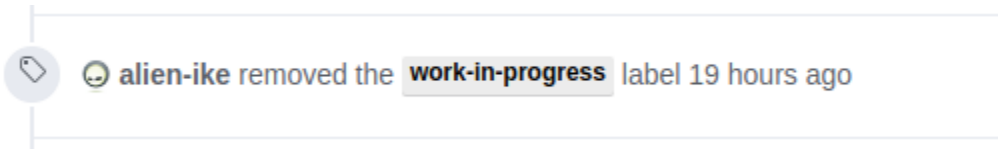

Figure-7: Checker removing the "work-in-progress" label from the merge request

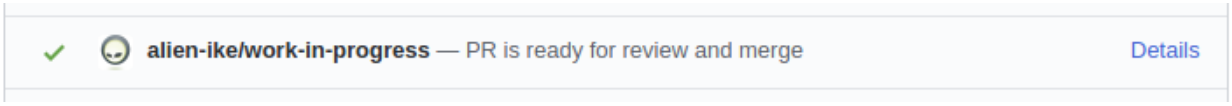

Figure-8: Checker unblocking the merge request

Reason of the merge request: When a merge request is submitted for review, it must either be to implement a story or to fix a defect. Random requests with no references are not encouraged as the reviewer would also need some context before reviewing the work. The *checker* ensures that this process is followed by ensuring that a reference hyperlink to the user story or the defect that is addressed is provided in the merge request.

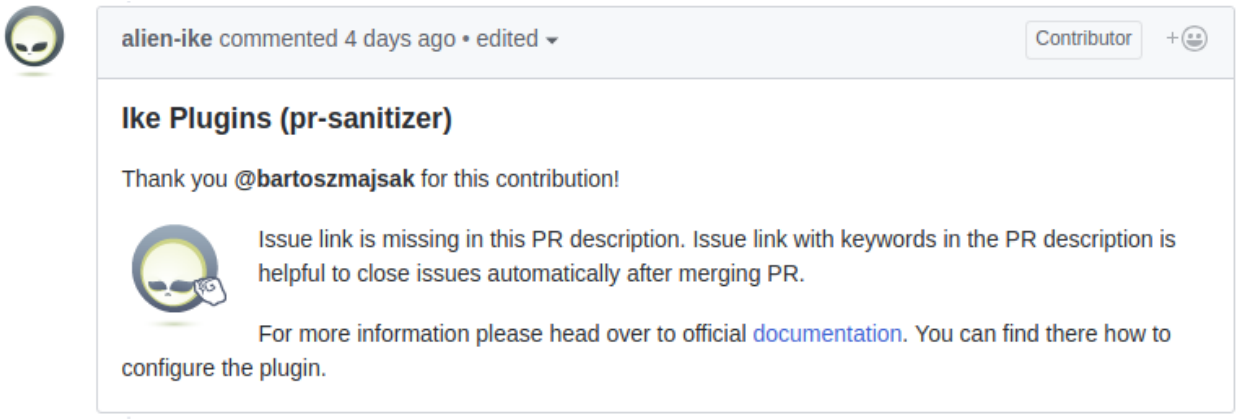

Figure-9: Checker marking the merge request as a link to the issue it fixes is missing

Optionally, the presence of minimum description text could also be enforced by counting the characters entered by the author in the description field.

5.6 Deep Insights

Having an automated quality *checker* has its advantages, beyond great quality software; quality metrics!

Each quality check failure could be captured as a metric and collection of such data can indicate deeper insights into the real issues that teams might be facing. Consider the following:

a) A large number of merge requests that are blocked due to missing or insufficient tests indicate a skill gap. It's possible that the team need additional training to start writing tests.

b) Sizable regression failures indicate careless coding practices and may require a stringent peer-review system to fix. This metric also indicates the number of potential breaking defects that were prevented from reaching the production system.

c) A peer feedback to the merge request could be captured using special flags representing defects. For example, feedback reported like this:

"/defect-high Missing null check"

This could be tracked as a defect against the code. Similarly, the author can mark a defect resolved like this:

"/defect-high-fixed Missing null check"

As a precondition to merge the code, all defect counts against the request must be zero:

i. All defects reported and fixed as reviews could be termed as early defects, which are the preferred type of defects.

ii. Early defects are the ones that failed to reach production. Similarly, an escaped defect is that defects that slipped your checks and processes to reach production.

iii. Using an automated CI process, keeping a count of such defects is possible.

d) Since the code and tests are tracked as part of single merge request, it's possible to create an automated map between the code churned and the tests that cover it. Let's call it code-test matrix. This could be used in future for effective and smart regressions.

e) If the code and the defect tracking is consolidated in a single system, then tracking back a defect to its original source of introduction could also be tracked. Since the code commit history is maintained at all times, a late defect while being fixed, could be traced back to its original source of injection. Not to play a blame game, but to learn about the reason the defect escaped in first place.

6. Results

An automated quality *checker* that not only performs regression, but also ensures process compliance results in a self-sustaining model of development that always keeps pace with the pace of development. The resultant product is of high quality with reduced probability of finding defects in production. The metrics that it can offer also acts as a feedback to allow continuous improvement of the continuous integration system.

7. Final words

The changes suggested in this paper require a system that may have to be developed grounds-up as it may not be available off the shelf. Folks using GitHub, you can look at the open source *Ike Prow Plugin*[3] developed by Red Hat. This may give you a head start, but remember the plugin is still in its early days and all functionality discussed in this paper may not be available. Here is the recommended workflow in the form of a flowchart:

3Arquillian – Ike-prow-plugin: http://arquillian.org/ike-prow-plugins/

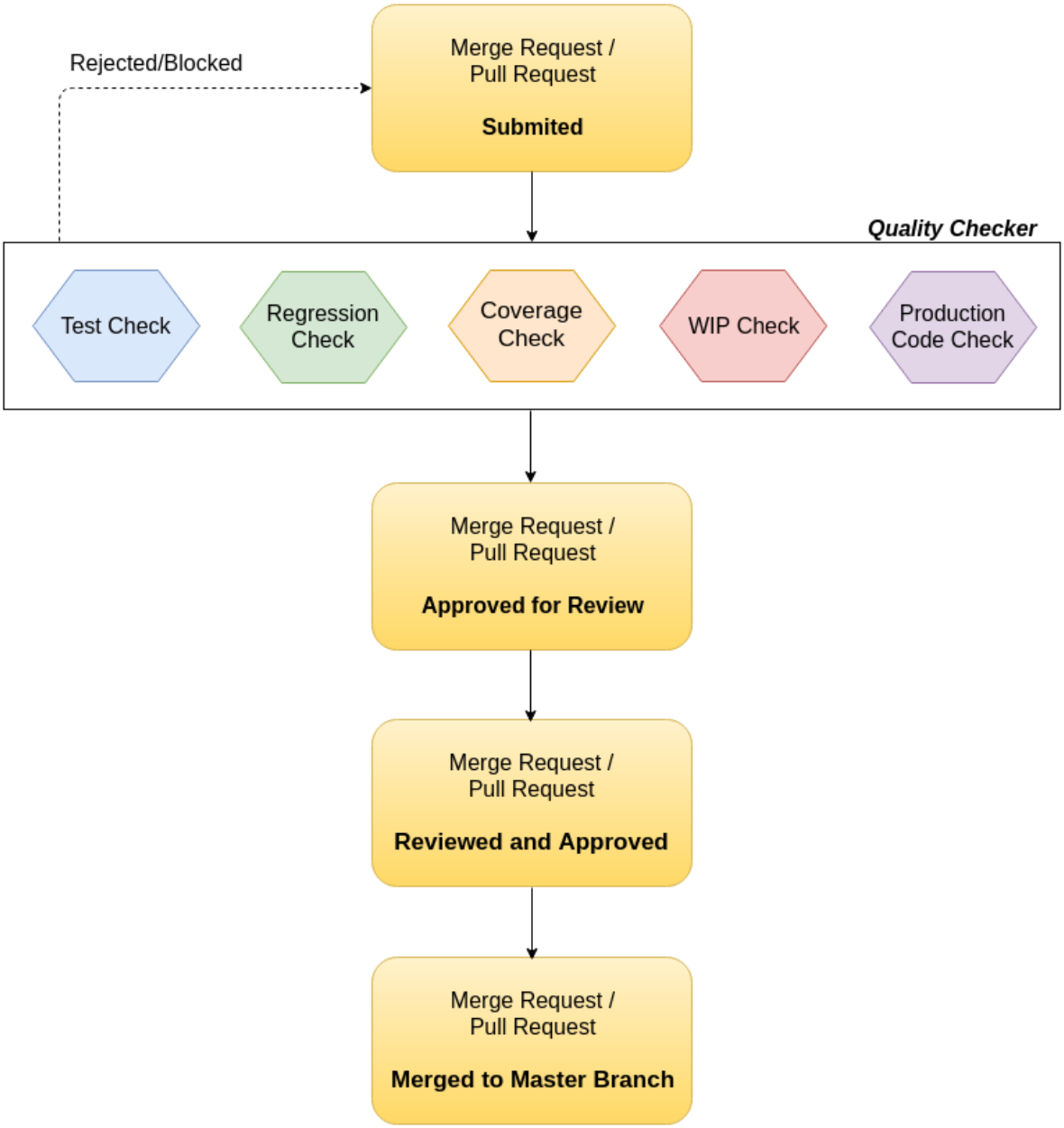

Figure-10: Recommended workflow

Such a system will extensively depend upon the API availability of the source repository system in use, and processes used in different organizations vary, so a one-size-fits-all approach will not work. Developing such a system will require investment and hence management commitment is needed to charter on this journey.

8. References

Arquillian – Ike-prow-plugin (The *checker*):

https://github.com/arquillian/ike-prow-plugins