# Adding Process to Increase Quality without Adding Tests

**Michael Millerick**

mmillerick@blend.com

## Abstract

Blend's primary product is a customizable web application. We use a large suite of pre-merge browser-based tests to ensure its continuing functionality and quality. The tests stand up a production-like configuration of our core platform and web application and allow calls to go out to other services in our sandbox environment. We refer to these tests as end-to-end tests. As the number of engineers increased, most had little visibility into the cause of flakey tests. The flakey tests would fail builds, yet engineers had little motivation or ability to remedy the cause of the failures. This tragedy of the commons created instability and prevented code changes from merging quickly. This instability in turn made it impossible to release as often as we desired.

We set out to improve stability without removing flakey tests from the test suite. We took a multifaceted approach to process and cultural improvement. We addressed the negative feedback cycle of instability with cultural and engineering changes:

1. We defined distributed ownership and analytics for tests to create visibility and accountability.
2. We provided engineers with tools that removed the need to watch the status of tests on code changes.
3. We eliminated escape hatches that allowed engineers to bypass tests.
4. We improved the scalability of our test infrastructure.

This approach improved the pass rate of tests and reduced the tribal knowledge and barriers to merging code. The improved ease of merging, combined with the infrastructure changes, improved the rate we released new features to production to almost daily. These changes cut the time engineers waited for tests to finish by half. They also reduced the amount of infrastructure per engineer required to run end-to-end tests. Over the course of the changes described in this paper, our engineering team roughly doubled in size while we were able to hold the size of our test infrastructure constant.

This paper describes the implementation details and impact of each of the four points in the approach described above. It will also examine how the changes made for each point in the approach reinforced the need for the other points.

## Biography

*Michael is an Engineering Manager at Blend responsible for Quality and Automation. His automation efforts focus on engineering velocity, productivity, and quality of life. He enjoys scaling distributed systems. He enjoys accounting, finance, investing, and previously worked in the StarMine group at Thomson Reuters.*

# 1. Introduction

Blend's primary product is a customizable web application. The large configuration space and desire to ship a product that will delight customers means that we always focus on quality. Early in the company's history, this took on the form of a large manual regression suite that QA would work through for every release candidate. This was a multi-day process and limited our ability to achieve our desired once per day release cadence.

In 2017 the engineering team began to decrease the size of the manual regression suite, primarily by transferring the majority of the manual test cases into our automated pre-merge browser-based end-to-end tests against the majority of our backend and frontend applications. This allowed us to achieve our goal of a daily release cadence of the platform, but had two consequences on the end-to-end test suite:

1. The run time for the suite, and therefore the amount of time it took to merge a Github pull request, increased.
2. Many of the newly added tests were flakey. They would fail in situations where there was no issue with the application. Running the tests did not give reliable results.

The unreliable test results eroded confidence in the suite and caused engineers to constantly battle against probability for their pull requests to pass all of the pre-merge tests. The increased run time of the suite also meant that this battle was often drawn out over the course of several hours. While we had achieved our goal of a daily release cadence, we had unintentionally decreased the velocity of the engineering team in the process.

To combat these two consequences, we built out analytics associated with our test results, built tools to help engineers merge their pull requests and regain confidence in the process, strengthened our process around merging code, and made significant improvements to our test infrastructure. Our overall goal was to increase engineers' confidence in the test suite and get to the point where flakey test results were more likely to be an indicator of bugs in the application than they were to be an indicator of a poorly written test.

# 2. Analytics and distributed ownership

We already had basic analytics on our end-to-end tests. Since early 2017, we published test result information into Datadog[1] at the end of every single test. We found the insights Datadog was able to give us inadequate because we could not view trends over a long period of time and we could not formulate complicated queries to give us insight into which tests were the most problematic. We decided to create a service, Test Center, which would store all of the data in a Postgres database and provide views into the data.

In order to decide which tests were flakiest, we first needed to decide on how to differentiate between "good" results and "bad" results. We ultimately decided that "good" results would be those where all of the tests passed within the maximum number of retries before the job finished—the ones that would result in a green status indicator on a pull request and allow the author to merge their pull request. All others would be classified as "bad". This allowed us to compare the test results for the known "good" jobs to the entire set of test results and gain information from the different trends in the two sets of data and filter out systemic, external, and infrastructure issues, from the signal in a systematic fashion.

We decided to operate under the assumption that the "bad" results that prevented engineers from merging code had one of two causes. The first are legitimate regressions in the product detected by the tests that would need to be fixed before the code was merged. The second are indicative of systemic, external, or infrastructure issues. Neither of these two situations provide signal on whether or not a given test is flakey since both are deterministic in nature. Our filtering to just test results that are indicative of flakey test behavior unfortunately also disregards results when tests are so flakey that they fail an entire

[1] A monitoring service for cloud-scale applications.

run and prevent engineers from merging code. While we filter out more results than we arguably need to, we found that the information we preserved provided us with a sufficiently strong signal. We could then use that signal to investigate whether the test was flakey because of issues within the test, or because of bugs in the application that would cause the test to fail periodically (e.g. race conditions).

| Test Run | Attempt | Result |
|---|---|---|
| 1 | 1 | Pass |
| 2 | 1 | *Fail* |
| 2 | 2 | *Fail* |
| 2 | 3 | *Fail* |
| 3 | 1 | *Fail* |
| 3 | 2 | Pass |

To contextualize our filtering logic in an example, assume we had the information shown on the right for a test in the database. This single example test was part of three separate test runs. In the first and third test run, the test passed prior to the end of the test runs. In the second test run, the test did not pass and would have resulted in a failing status indicator on the pull request. Therefore, our filtering logic excludes the second test run from the "good" job pass rate. With the results of the second test run excluded, the "good" job pass rate for this test is 66.6% (two passes out of three total attempts). The pass rate considering all test runs is 33.3% (two passes out of six total attempts).

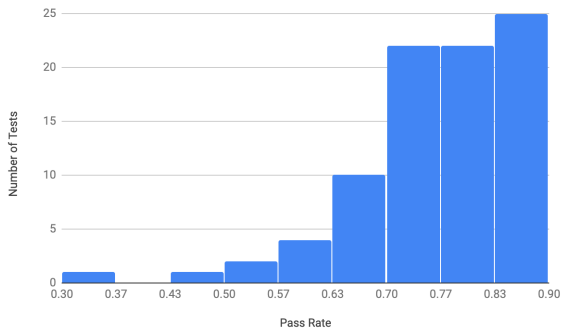



*Figure 1: The initial distribution of the "good" job pass rate for our tests. Shown are the 87 tests with "good" job pass rates below 90% to improve readability.*

With these simple analytics developed, we now had a strong picture of which tests were flakier than others. The "good" job pass rate on tests ranged from about 34% to 100%. 87/373 tests had pass rates lower than 90%, which lead to substantially more failed test jobs from false positives than we wanted.

We now had a concrete picture of which tests were in the worst shape, but with so many tests in need of fixing (either by fixing the application(s) being tested or by fixing the test, the former being more preferable to fix), we needed a way to know which engineering team was responsible for fixing each test. Using the information in the git history, and with guidance from engineering leadership, we added a comment to each of the test files to denote which engineering team was responsible for each test file. The clear assignment and distribution of ownership helped us build tooling around the distributed ownership.

We had data and ownership. We followed by adding additional process. We created the notion of a technical health team to apply upward pressure on the pass rate of tests. We set a threshold "good" job pass rate, below which we categorized a test as flakey, and set how many flakey tests each engineering team was allowed to have before they would need to contribute an engineer to the technical health team for a sprint. By carefully setting these thresholds and expectations around the technical health team, we have increased our threshold for a flakey test from a 70% to 85% "good" job pass rate without any engineers added to the technical health team. If engineers were to be contributed to the technical health team, they would spend the sprint focused on improving their team's flakey tests to reduce the likelihood that their team would need to contribute an engineer to the technical health team in a future sprint.

# 3. Tools

We built tools to allow the organization to push through the difficulties caused by frequently failing tests without eliminating tests from the suites. The overarching goal of the tools described below was to minimize the number of times engineers had to rerun tests on their pull requests and to reduce the likelihood that engineers would attempt to run tests on their pull requests while there was little to no chance that the tests would pass.

## 3.1. The merge bot

We built a bot that would help guide pull requests into a merged state. Pull request authors needed to get their necessary reviews, and then attach a label to the pull request in Github Enterprise. The bot would then perform the minimal amount of work necessary to merge the pull request into the appropriate branch.

The label also had the secondary benefit that it allowed us to better utilize our existing infrastructure. The label effectively formed a queue of pull requests that needed to be merged. Authors would attach the label, and then the bot would work tirelessly day and night to merge the pull requests. Before the label, we would frequently run into problems when engineers first came into the office and immediately after lunch, when engineers would attempt to run tests on their pull requests in bulk. It was a poorly kept secret that it was significantly easier to merge pull requests at midnight than it was to merge pull requests during working hours—a fact that we did not like but tolerated. The label also made it easier to recover from systemic issues in the test environment. If there were problems with tests, engineers could apply the label and have confidence that the bot would merge their pull request once the environment was stable again.

## 3.2. Slack notifications

We added Slack notifications that would be sent to the pull request author when tests failed on their pull request. It contained a link to Test Center that would state which tests failed on the pull request, what the "good" job pass rate was on that test, as well as which team owned the test(s) that failed after all retries were exhausted. This provided two benefits to us:

1. It drastically reduced the difficulty for pull request authors to understand whether or not they legitimately failed tests. If the one or two tests that failed their pull request were ones with very low "good" job pass rates, then it was likely they were just unlucky.

2. It put pressure on the owners of the worst performing tests to improve the pass rate of the tests. It quickly became public knowledge which tests lead to the most frequent failures.

## 3.3. Issue tracking

We use Jira to track issues and defects. We built a daily job that leveraged the Test Center data to create/close/reopen Jira issues based on the flakey tests. If a test became flakey for the first time, it would create a Jira issue in the Jira project for the team that owned the test and assign it to the lead for that team. If a test was no longer below the threshold for flakey tests, it would close the appropriate Jira issue. If a test became flakey again, it would reopen the existing Jira issue.

The automatic actions of the bot made it easier for teams to track their flakey tests. This enabled teams to be more proactive about correcting their flakey tests, since they were now visible in their backlogs rather than visible only in Test Center. The automatic actions also made it impossible for them to hide their flakey tests. If a Jira issue is closed out, and the test is still flakey a week later, the job will reopen the Jira issue and keep the team accountable for fixing their flakey tests.

## 3.4. Test status indicators

All engineers working on our core platform are in a Slack channel for developer support and notifications about the core platform. The channel topic contains indicators for the general health of the core platform in the sandbox environment. All of these indicators were manually maintained. We gave Test Center the ability to maintain the test related indicators using its own data. We use the following two heuristics to determine if it is possible for a test suite to pass:

1. Are there any tests that have failures, and zero successful results in the last hour? If so, those tests, or the portion of the application they are testing, are broken and it is not possible to pass the suite.

2. Are there any test results present in the database in the last hour? If not, it indicates an issue with the sandbox infrastructure preventing the tests from even beginning to run and it is impossible to pass tests.

The automated maintenance of the status indicators reduced our detection time for issues preventing tests from passing. Prior to this, test maintainers needed to be incredibly vigilant, or rely on engineers pointing out that tests were probably broken, to know that there were problems in the sandbox environment that needed to be resolved. These changes put a cap of one hour on our time to detection for broken tests.

# 4. Elimination of the escape hatch

When there were relatively few engineers, an "escape hatch" was created that would allow engineers to merge their code without passing any tests. This was tenable in a small engineering organization because the engineers had detailed knowledge of every area of the platform and therefore were able to make good decisions around when they could skip all of the tests without impacting any of the tests. Over time, engineers also began to use the escape hatch to bypass broken tests. This eventually turned into using the escape hatch to bypass flakey tests as the organization grew.

This was problematic. We frequently ran into this cycle:

1. There would be an issue with the infrastructure or an external service, which prevented one or more tests from passing.
2. Engineers would skip past the tests they failed in order to continue merging code during the outage.
3. The original cause of the outage would be fixed, but tests continued to fail because one or more of the pull requests merged during the outage broke one or more tests.

This would repeat in a cycle, sometimes for multiple days. Not only would engineers need to fix the original cause of the outage, but engineers would need to hunt down and fix the one or more bad pull requests that merged during the outage, an exercise that was time consuming and was only necessary because of the escape hatch. We decided that the escape hatch needed to be eliminated.

When we rolled out the merge bot discussed in the previous section, we also began tracking the number of days the organization went without skipping tests with the escape hatch. We tracked this information on a whiteboard near the Quality team and sent out a daily Slack message with a picture of the whiteboard. Initially, there was very little change in behavior. Gradually the organization became comfortable with the new workflow. As more and more engineers adopted the merge bot to merge pull requests, the organization was able to go more and more days without using the escape hatch. After about a month and a half of daily dashboard updates, we reached 10 consecutive days without using the escape hatch and disabled it entirely on 13 April 2018.

# 5. Scalable infrastructure

In parallel to these other initiatives, we also migrated from Protractor[2] to WebdriverIO[3] as the framework for running our end-to-end tests. With this migration, we also built out our own test runner to run the test containers on our Kubernetes cluster rather than directly on our Jenkins servers and to parallelize the tests in a more intelligent fashion than the default behavior of the test runner. Improvements in these two areas, combined with increases to the pass rates of tests, reduced the wall-clock time of tests from 30 minutes to 10 minutes.

[2] An end-to-end test framework for Angular and AngularJS applications.
[3] A next-gen WebDriver test framework for Node.js.

## 5.1. Test parallelization and sandboxes

In order to reduce the runtime of our tests, we run many tests in parallel. While we were using Protractor, we used the built in `shardTestFiles` option to parallelize the tests. We would run the tests once, see which tests failed, and rerun those tests. This led to a waterfall pattern where the retries of failed tests could only begin after all of the original tests had failed. This resulted in long periods of time where only a single test was running, and we were not effectively utilizing our infrastructure.

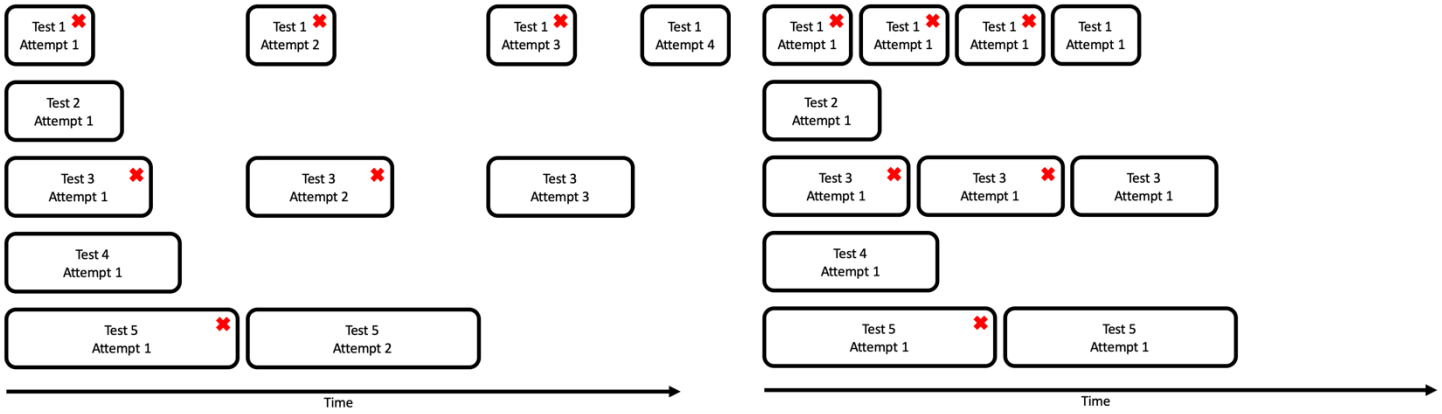



*Figure 2 (left) How tests were distributed with each of their retries with our original Protractor tests. (right) The test duration benefit from leveraging our platform's multitenancy feature.*

Our application is very configurable, which means that many of the configurations are modified over the course of our tests. Since the tests had grown from nothing under our Protractor framework, alongside a configuration space which had also grown from nothing, each individual test was not sandboxed very well. This led to flakey tests and frustrated engineers. With our switch in frameworks, we also began to leverage features from our platform to sandbox tests from each other. Our platform application supports serving multiple customers from a single instance of the platform application by completely segregating all information based on its knowledge of which customer API requests originated from. We call this feature multitenancy. We leveraged the multitenant capabilities of our platform to create N identical tenants in the platform application, which also dictated how parallelized our tests would be. We could rely on our multitenant capabilities to sandbox tests from other tests running in parallel since bugs in the implementation of our multitenant capabilities would mean that the entire test suite would fail deterministically.

We updated the setup function at the beginning of tests to reset feature flag and configuration states to a standard configuration. Our use of tenants on our platform meant that each test was isolated from each other, while our setup function guaranteed that each tenant began in the same state for each test. We updated our lint rules to require that the setup function was used properly at the beginning of every test. This forced all tests to begin with the same initial application state, which eliminated the potential for tests to fail because of state left behind by previous tests. This reduced the cognitive overhead when writing new tests since engineers no longer had to worry about the configuration changes made by other tests.

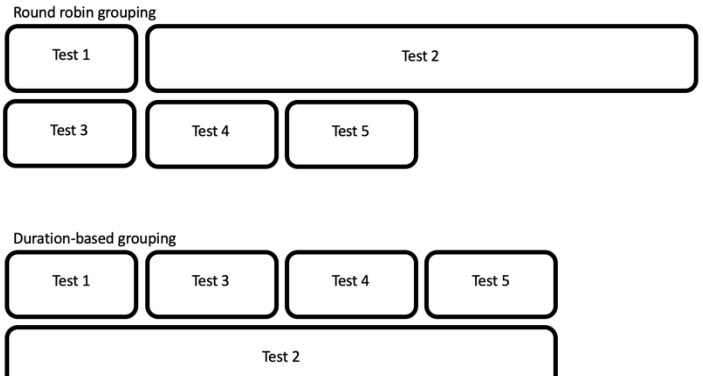



*Figure 3 (above) The impact of round robin grouping on test duration. (below) The improvement in overall run time from considering historical test duration while grouping tests.*

In addition to the success or failure of tests, Test Center also collects information on the run time of each test. If set to run two tests in parallel, our grouping algorithm creates two different test groups, with roughly equivalent total expected duration from looking at the historical run time information in Test Center. Assume a test suite has five total tests, four of

which are expected to complete in one minute based on historical data, and one expected to complete in four minutes. Our grouping algorithm would group the four tests expected to complete in one minute each into a group and would put the single test expected to complete in four minutes into its own group. Both groups would run in parallel, and the tests would finish in about four minutes. This guarantees that all tests will finish at almost exactly the same time, which minimizes the amount of time engineers need to wait for their tests to complete.

### 5.2. Jenkins v. Kubernetes

We found that Jenkins did a poor job scheduling work. While the number of jobs was fairly evenly distributed across all of our Jenkins servers, the load placed on each server was rarely evenly distributed. This led to some servers being put under extremely heavy load, and they would frequently remove themselves from the cluster as the Jenkins process crashed. Our tests were already run in containers, so we started launching those containers on our Kubernetes cluster instead of on the local Jenkins server. This allowed us to specify and tune exactly how much CPU and memory was needed for the tests to run and provided a stable environment in which the tests were guaranteed to have access to the resources that they needed to run to completion.

Putting all of our test traffic on the Kubernetes cluster in our sandbox environment also helped us to stress test the cluster. The tests were the first heavy load we placed on our Kubernetes clusters, and served to stress test the infrastructure and tooling built around Kubernetes. Our most important learnings were

1. Our original cluster overlay, Weave, was insufficient for our large amount of pod churn to be handled gracefully by Kubernetes. We switched to Calico and saw a noticeable increase in network and test stability (unfortunately the increase is difficult to quantify due to there being different metrics between the two overlays).
2. How to appropriately size the nodes in our cluster for our workload and tune resource requests and limits for our application. These learnings made it significantly easier to ultimately move our core platform from Amazon Elastic Container Service to our Kubernetes cluster in May 2019.

## 6. Conclusion

Through all of the changes described above, we have significantly improved the pass rate of our end-to-end tests. Refer to Chart 7.1. You can see several areas where the pass rate of tests degraded, and then rebounded shortly thereafter when we fixed the bug(s) identified in the application from examining which tests had degraded the most. These downward spikes are most apparent around April 2018 and April 2019. Also visible is the large increase in pass rate (and corresponding drop in overall test time) that came with a significant performance increase in the application, the opportunity for which was identified by examining the test data in Test Center.

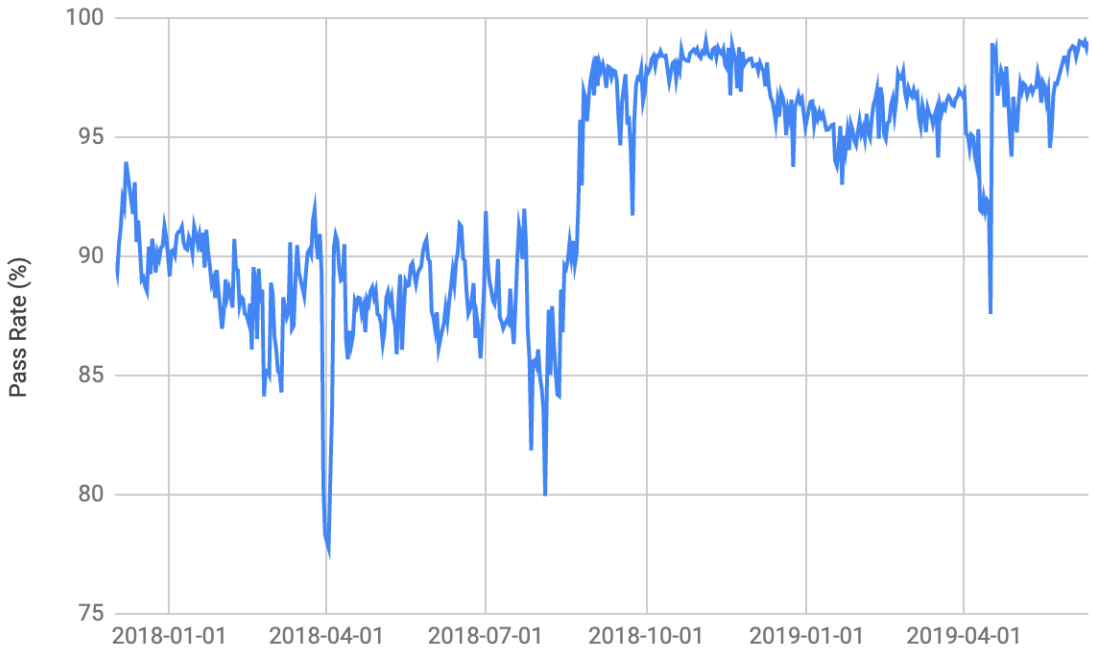
Improvements to test grouping and how we run the tests on our Kubernetes infrastructure have decreased the wall clock runtime of tests. Refer to Chart 7.2. When we began the improvements described above, the wall clock runtime of tests varied between 20 and 30 minutes. By improving the stability of tests, and with continuing improvements to how tests are grouped, the wall clock runtime of the tests has decreased to about 10 minutes and varies significantly less from run to run. This is notable because the CPU time used by the tests (Chart 7.3) has not decreased as much as the wall clock runtime has decreased.

Improving the pass rate of our tests has given us better signal on the health of the application from our test result analytics. When the pass rate of our tests begins to decline, we now have confidence that instability was introduced in the application and are able to quickly identify and remediate the instability. We believe this has had significant benefit on the stability, and therefore quality, of our platform and application. The majority of issues that have been corrected have been performance related, or pure race conditions in the application. The large volume of test runs has given us enough signal to identify these issues that are otherwise nearly impossible to find manually. Additionally, engineers trust the tests more

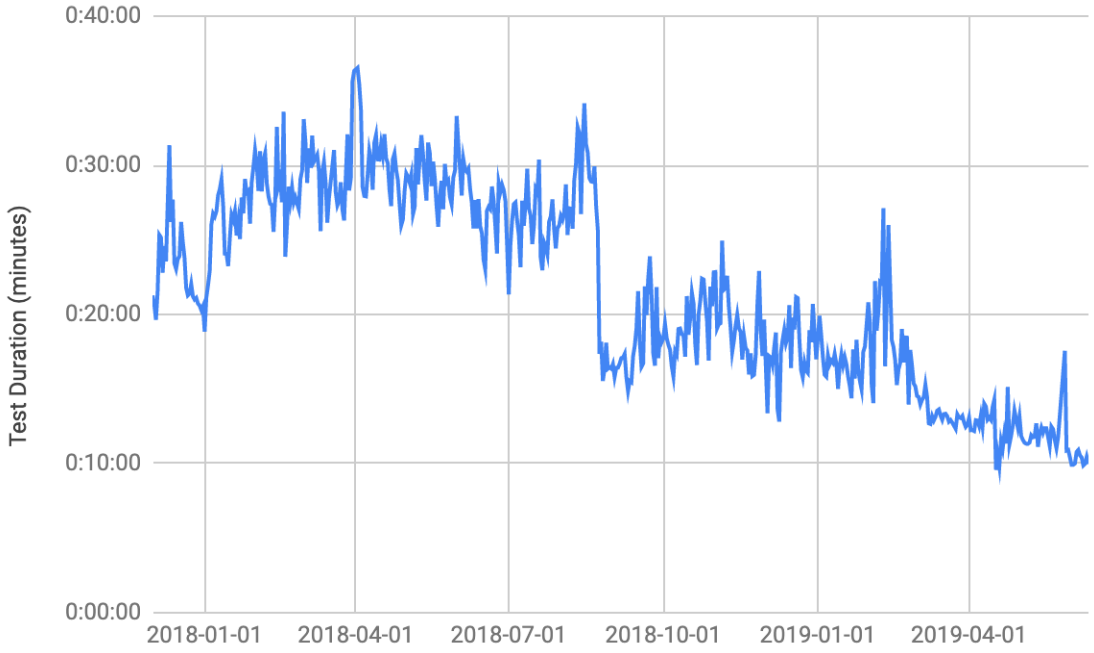
which has increased their willingness to add new tests, maintain the existing tests, and address likely issues uncovered by examining the test analytics in Test Center.

# 7. Charts

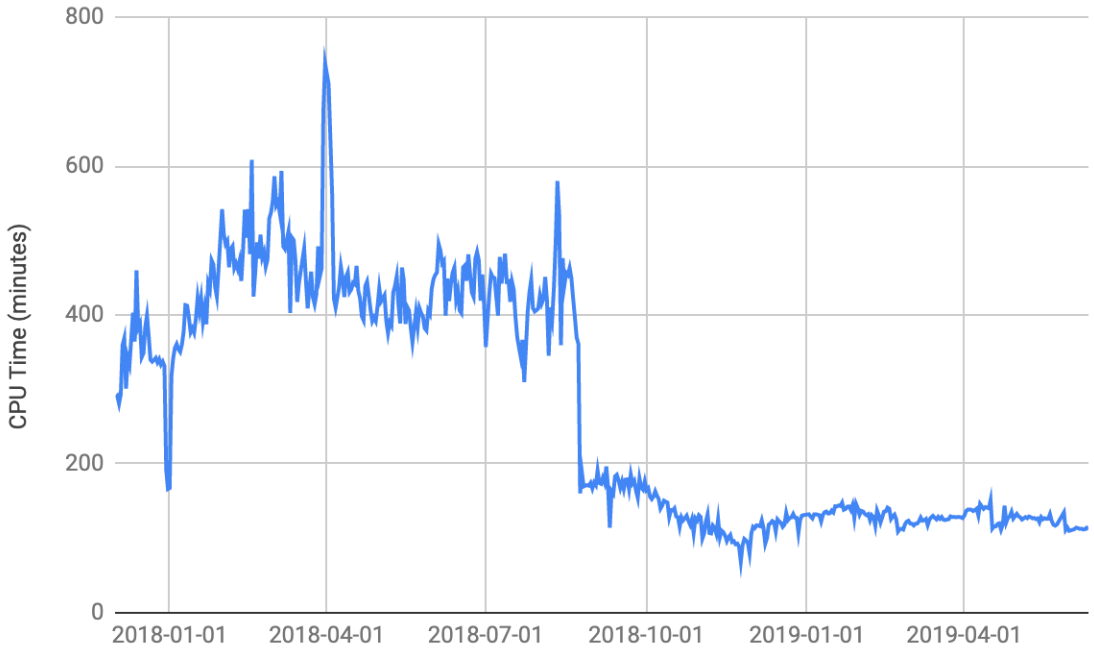
## 7.1. End-to-end test pass rate



## 7.2. Average wall clock duration per end-to-end test suite run




## 7.3. Average CPU Time spent per end-to-end test suite run



# References

Datadog, https://www.datadoghq.com/

Protractor, https://www.protractortest.org/#/

WebdriverIO, https://webdriver.io/