When Life Gives You Oranges: Detecting and Diagnosing Intermittent Job Failures at Mozilla

Johannes Lampel  
CISPA Helmholtz Center for Information Security  
Graduate School of Computer Science, Saarland University  
Saarbrücken, Germany  
johannes.lampel@cispa.de

Sascha Just  
Microsoft Corporation  
Redmond, Washington, USA  
sascha.just@microsoft.com

Sven Apel  
Saarland University, Saarland Informatics Campus  
Saarbrücken, Germany  
apel@cs.uni-saarland.de

Andreas Zeller  
CISPA Helmholtz Center for Information Security  
Saarbrücken, Germany  
zeller@cispa.de

ABSTRACT
Continuous delivery of cloud systems requires constant running of jobs (build processes, tests, etc.). One issue that plagues this continuous integration (CI) process are intermittent failures—non-deterministic, false alarms that do not result from a bug in the software or job specification, but rather from issues in the underlying infrastructure. At Mozilla, such intermittent failures are called "oranges" as a reference to the color of the build status indicator. As such intermittent failures disrupt CI and lead to failures, they erode the developers’ trust in the entire process. We present a novel approach that automatically classifies failing jobs to determine whether job execution failures arise from an actual software bug or were caused by flakiness in the job (e.g., test) or the underlying infrastructure. For this purpose, we train classification models using job telemetry data to diagnose failure patterns involving features such as runtime, CPU load, operating system version, or specific platform with high precision. In an evaluation on a set of Mozilla CI jobs, our approach achieves precision scores of 73%, on average, across all data sets with some test suites achieving precision scores good enough for fully automated classification (i.e., precision scores of up to 100%), and recall scores of 82% on average (up to 94%).

CCS CONCEPTS
• Software and its engineering → Software testing and debugging

KEYWORDS
Software testing, continuous integration, flaky tests, intermittent failures, machine learning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE ’21, August 23–28, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8562-6/21/08...$15.00
https://doi.org/10.1145/3468264.3473931

1 INTRODUCTION
Continuous integration (CI) is a practice used to ensure software quality by continuously testing and deploying code changes [28, 46]. Typically, CI systems run a multitude of build scripts, static code checks, automated tests, and deployment scripts—called jobs in what follows. Testing and assessing the software continuously and fast is crucial to the integrity and timely delivery of the software under test [18, 28]. In practice, developers spend a lot of time and resources on writing and maintaining jobs, in particular, tests [27].

A common assumption is that jobs should be deterministic, meaning that a job, no matter when or how it is executed, always produces the same result as long as the code is not changed. In practice, however, this is not always the case [31, 52]. Some tests and, thus, jobs have non-deterministic behavior. This behavior can have several causes, such as resource availability and concurrency issues [38]. Tests showing such behavior are called flaky tests [5, 32, 47]. While flaky tests have a significant impact on CI, any CI job may fail because of non-determinism. Such failing jobs are referred to as intermittent failures or, in case of Mozilla, oranges. A simple cause for an intermittent failure may be a temporary network outage, resulting in a build and deployment job to occasionally fail or to create executables that later fail tests. Intermittent failures have been reported to be a frequent issue in practice [3, 20, 26, 51].

Intermittent failures weaken the developers’ trust in CI and its jobs drastically, as developers often cannot distinguish between real and intermittent failures. Also, they make developers’ lives harder as they do not know whether a failure is caused by a software change or some non-deterministic influence, thus compromising the entire testing and CI process. Furthermore, intermittent failures result in wasted resources: unnecessary waiting times, re-runs, or even manual investigation. Even if engineers recognize an intermittent failure, the failure still prevents them from integrating and getting their changes published.
At Mozilla, 53% of all job failures are classified as intermittent (Table 2); at Google, 84% of the transitions from passing to failing are related to flakiness [39]. These high numbers are the result of various, partially hard-to-reproduce issues causing intermittent job failures. Examples of such root causes include:

Network resources. Jobs relying on networking might fail intermittently due to the network being unavailable, being significantly slower than usual, or because network resources might not be available [38].

Job dependencies. Jobs frequently depend on earlier jobs to complete; test and deployment, for instance, require build first. Although tests are expected to be independent of other tests, in practice, this often is not the case [4, 20, 38, 49]. Some tests rely on other (previously executed) tests to change the environment in order to run properly. If the order of tests changes, these assumptions can be violated and result in a test failure [21, 22, 53].

Concurrency. Using multiple threads in a job can cause intermittent failures caused by data races or deadlocks [38].

The literature lists additional causes for flaky tests and intermittently failing jobs such as asynchronous wait, resource leaks, I/O, and floating point operations, but even this list is not exhaustive [17, 20, 38]. Even if properly identified, fixing the cause for non-determinism can be extremely complicated, time-consuming, and hard to achieve overall [38]. Therefore, it is difficult to efficiently mitigate the impact of intermittent failures by fixing the test itself.

To prevent real bugs from escaping into delivered software systems [44], where they are substantially more costly to fix [12, 48], it is crucial to be able to reliably distinguish between (1) actual failures caused by bugs and (2) intermittent failures. Common approaches to classify failures as intermittent include (1) running jobs multiple times, with intermittent jobs being those that pass and fail at least once under the same configuration [20, 39], (2) using coverage data to identify flaky tests and, thus, intermittent failures [8], and (3) having dedicated engineers (called sheriffs at Mozilla) classify job failures by hand in shifts around the clock. Clearly, repeated execution of jobs and manual classification is expensive; and state-of-the-art approaches such as Deflaker [8] only apply to tests, require code instrumentation, and do not generalize to arbitrary CI jobs.

To address these issues, we devise a novel approach that predicts intermittency using only data from existing job runs. We make use of job telemetry data collected in the CI process to train classification models for intermittent failures. These models allow us to detect and identify intermittent failures right from the start, without requiring repeated job runs, differential coverage, or expensive manual classification.

To evaluate our approach at Mozilla, we have collected almost four months of Mozilla job data amounting to over 2 million job execution runs with an average of 67% failing job executions manually marked as intermittent failures (Table 2). We have trained a number of classification models on these data with the goal of classifying job failures as intermittent or regular. Our approach also provides means to explain the causes of intermittent failures. Specifically, we can identify the features that have the largest effect on the classification to guide developers in finding the actual root cause of the failure. In cooperation with Mozilla sheriffs and engineers, we interpreted found patterns and assess the capabilities and feedback of our approach.

This paper makes the following contributions:

(1) We collect telemetry data of over 2 million job execution runs (Section 3), including a ground truth for intermittent failures, at Mozilla. This data set provides important insights into how and when jobs fail in an industrial setting. The data set is publicly available.

(2) We present a novel approach to detect and diagnose intermittent job failures based on telemetry data. Our approach creates recommendations (Section 4) on how to classify job failures and provides engineers with the most important features behind these decisions. We investigate whether there are common patterns that affect the predictive power of the classification model. In contrast to other approaches, our approach works on arbitrary jobs independent of test executions.

(3) We evaluate our approach on Mozilla job data (Section 5). Our approach achieves high precision and recall scores when classifying failures as intermittent and accurately classifies failures for arbitrary jobs independent of tests, code changes, or configuration changes. Based on the insights gained from the patterns that we have identified, we evaluate how our approach can point to underlying root causes.

To foster open science, the whole data set used for this study is publicly available, including a Jupyter notebook that allows to run and assess all classifications:


2 CONTINUOUS INTEGRATION AT MOZILLA

We chose Mozilla as industry partner, because all of their code is open source, and all of their data are publicly available and, thus, can be mined. Furthermore, the project size (i.e., developers and code) as well as the software development process reflect software development in industry.

2.1 The Mozilla CI Process

The CI process at Mozilla consists of multiple communicating services. We will explain the interaction (Figure 1) of those services with the help of the following scenario: Mozilla engineers develop using a Mercurial-based version control (Hg) system [2] at hg.mozilla.org. Hg contains most of Mozilla’s repositories. For some projects, Mozilla also uses GitHub. However, these projects are not connected to this integration process. Changes pushed to hg.mozilla.org are tested and deployed using Taskcluster, the Mozilla in-house CI system. Its main functionality is to schedule builds, execute tests, and deploy artifacts. Furthermore, it is responsible for emitting test telemetry. Test telemetry data are temporarily stored within Treeherder. Treeherder is a web service that lets developers monitor their test results. It is also used by sheriffs to classify test failures. Treeherder holds approximately four months of data in an internal SQL database. All failures that occur during Mozilla’s CI process are reported to Bugzilla, and a bug_id is
either created or an existing one linked to the build. If a failure occurs, sheriffs investigate these failures and classify them as either
oranges or regular failures. Currently, there is a team of 16 sheriffs
with, at least, one of them monitoring the tree at any given time. If
a failure is classified as an intermittent failure, it is reported to Or-
angeFactor where Mozilla stores records of intermittent failures.
All these services communicate using the Mozilla message queue
service Pulse. The interaction between those services is depicted in
Figure 1.

Figure 1: The Mozilla CI system. Incoming changes (1) are
being pushed to the mozilla-inbound repository (2). The
Taskcluster (3) service commissions machines and per-
forms the build and test tasks. The results are persisted
to AWS S3 storage in JSON format and emitted as teleme-
try via Pulse (5). Treeherder processes the telemetry data,
stores them in an internal SQL database and provides an
interactive Web view showing failed and successful
jobs. Mozilla sheriffs inspect failures and decide whether
those are real bugs or intermittent failures.

To better understand how intermittent failures manifest in the
Mozilla CI process, we interviewed developers and sheriffs. They
explained to us that there are a couple of common causes for inter-
mittent job failures, e.g., timeouts, or missing permissions. Timeouts
can be caused by jobs that require large sets of files to be processed
first, or jobs waiting for resources to become available. Both of
these issues will not result in a single test failing, but the entire test
suite.

2.2 Why the State of the Art does not Suffice
The setup at Mozilla and interviews with their engineers also
taught us three important lessons:
- Mozilla uses a distributed CI process. Such a distributed
process—while delivering a much faster build—is also more
prone to fail intermittently. Nodes becoming unavailable,
and networking issues are common problems that lead to
intermittent job failures. Because of this, it is not sufficient
to only look at jobs that fail because of flaky tests in an
industrial setting. There is also a need to identify jobs as
intermittent that failed for other reasons.
- Mozilla uses—like many other companies—test case selec-
tion. With test case selection, one tries to only run tests on
the part of the code that was recently changed. Because of
this, approaches to identify flaky tests based on differential
coverage (e.g., DeFlaker) will struggle to do so.
- While the majority of Mozilla jobs indeed run tests, there is
a large variety in what CI jobs do. Build jobs can fail, deploy
jobs can fail, static checkers can fail—and all intermittently
so. Hence, approaches that focus on test jobs alone will not
suffice.

These observations motivated us to design an approach that
would be able to predict and diagnose intermittent failures for all
types of jobs, and using a minimum of assumptions regarding their
applicability.

3 DATA EXTRACTION AND FEATURE
ENGINEERING
3.1 Data Extraction
For this approach, we relied on two main data-sources at Mozilla:
Treeherder and OrangeFactor. While the latter was only used
as a means to label job failures, the former provided us with the
needed telemetry data to do our analysis and to train classifiers.
When we started this study, Mozilla worked with us to acquire
Treeherder data, which covered all jobs on the mozilla-inbound
repository starting March 14th 2018 until July 11th 2018. In this
time frame, the number of jobs appears to be stable with Mozilla
running between 5,000 (on weekends) and 40,000 (during the week)
jobs a day (Figure 2). The obtained data contains the features shown
in Table 1.

Furthermore, we were provided with OrangeFactor data for the
same period of time containing the manual classification results of
the investigations by the sheriffs.

The raw Treeherder data were not suitable for a thorough analy-

sis given that it included test suites with too few runs presenting
each passing and failing results. The investigation of the test suites
showed that they were also relatively short-lived, which led us to
the assumption that they might only have been used to test new
configurations. Mozilla developers confirmed this. Therefore, we
removed all those short lived test suites. This left us with a total of
20 test suites, which corresponded to almost 2 million jobs in total.
The flakiness ratio of these test suites ranges between 22.92% and
87.51% for failing test suite runs. An overview of the data is given
in Table 2.
3.2 Feature Engineering

A major goal of our study is to inform the development of a tool that is capable of classifying intermittent job failures while providing developers with reasons as to why a job was classified as intermittently failing. For this purpose, we first analyze our data and only use features having an explainable relationship with intermittency instead of training a model with all available features. The first feature we investigated was run_time. run_time was constructed using the start_time and end_time columns of the data set. We found that there is a significant run-time difference between regularly failing jobs and intermittently failing jobs (T-test at a 95% confidence interval). For many of the test suites this hinted to high run_time being a good indicator for a job failing intermittently. When presenting this finding to Mozilla’s developers, they explained this correlation with the fact that the most frequently occurring intermittent failures are caused by timeouts.

A related feature we investigated was cpu_load. This feature also has a relationship with flakiness in a similar way as run_time. According to Mozilla employees, system appears to be another good indicator, since it is quite common that jobs executed on the Windows operating system fail intermittently because of broken paths. Furthermore, Mozilla engineers hinted us to platform_option as a good indicator because of multi-threading being handled differently depending on the platform_option.

Another feature we assumed to be worth investigating is the machine on which a certain job was executed. We assume that depending on the machine tests might be more or less likely to fail (e.g., because of resource starvation). Unfortunately, almost every machine_name in our data was just a hash of a virtual machine, making it impossible to recover a link to respective specifications. Furthermore, every hash occurs only once, which left us with no choice but to remove machine_name from our feature set.

There was no evidence showing that the person submitting a change to be tested was somehow correlated to jobs failing intermittently. Hence, we removed submitted_by from our feature set. Since job_type_name includes system as well as platform_option, we decided to not take this feature into consideration since it would have only introduced redundancy. We used suite as well as result to filter the jobs for further investigation. Therefore, we decided not to keep them in our feature set since they are already implicitly included.

This leaves us with run_time, cpu_load, platform_option, and system as features selected to train classifiers with.

4 METHODOLOGY

Our data set contains the test suite telemetry data explained in the previous sections, as well as the manual classification by the Mozilla sheriffs. We discard all test suites that did not present at least 40 intermittent and 40 regular, failing runs (complying to the One in Ten Rule [24, 25]). This ensures that we have at least 10 intermittent and regular failures for each feature to avoid overfitting.

4.1 Classification

To process the remaining test suites, we implement a classification pipeline using the Python Scikit-learn library [42]. For the purpose of this study we choose XGBoost [15], LightGBM [16, 30] and random forests [13] as our classification models to build a predictor for intermittent job failures. All three classification models are well established libraries that provide regularization against overfitting. We choose tree-based models as we have the means to quickly explain them. This is important as the goal is to give suggestions whether and why a failing job failed intermittently or not in as little time as possible. It is not our goal to train the best classification model possible.

We encode the categorical data (system, platform_option) using one-hot encoding to set it apart from the real-valued data. This transforms the system feature into up to 17 one-hot columns encoding the target operating system (e.g., system_linux64, system_windows10-64, and system_macosx64-qr), the feature platform_option encodes into 4 new columns representing different build types: asan (Address Sanitizer [45], a memory error detector for clang), pgo (Profile-Guided Optimization), debug (using debugging symbols), and opt (optimized/production).

We evaluate our approach on unseen data only. To do this, we split the data into 85% development and 15% evaluation (hold-out) parts. We sort the data set by timestamp to not predict past events.

Table 1: Overview of the feature of our data set.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>the ID of a job, assigned by Treeherder</td>
<td>Integer</td>
</tr>
<tr>
<td>start_time</td>
<td>the time the job was started</td>
<td>Timestamp</td>
</tr>
<tr>
<td>end_time</td>
<td>the time the job ended</td>
<td>Timestamp</td>
</tr>
<tr>
<td>result</td>
<td>the result of the job</td>
<td>Varchar</td>
</tr>
<tr>
<td>push_id</td>
<td>the ID of the push</td>
<td>Integer</td>
</tr>
<tr>
<td>machine_name</td>
<td>the name of the machine on which the job was run as a hash</td>
<td>Varchar</td>
</tr>
<tr>
<td>submit_time</td>
<td>the time the job was submitted to Taskcluster</td>
<td>Varchar</td>
</tr>
<tr>
<td>submitted_by</td>
<td>the email address of the developer who submitted the job</td>
<td>Varchar</td>
</tr>
<tr>
<td>job_type_name</td>
<td>includes the platform as well as the test suite</td>
<td>Varchar</td>
</tr>
<tr>
<td>system</td>
<td>the platform on which the job was executed e.g. macOS</td>
<td>Varchar</td>
</tr>
<tr>
<td>platform_option</td>
<td>the platform option for this job i.e. opt, debug, asan or pgo</td>
<td>Varchar</td>
</tr>
<tr>
<td>cpu_load</td>
<td>the average CPU load during the execution</td>
<td>Double</td>
</tr>
<tr>
<td>suite</td>
<td>the test suite that was executed</td>
<td>Varchar</td>
</tr>
</tbody>
</table>

---

1https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Building_with_Profile-Guided_Optimization
Figure 2: Mozilla’s job runs per day. On average, the number of builds appears to be stable. On week-ends (highlighted in gray), the number drops to about 5,000. During the week, Mozilla runs up to ∼40,000 test jobs a day.

Table 2: Overview of the data sets: Test suite runs in Mozilla’s Treeherder database.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Success</th>
<th>Failure</th>
<th>Retry</th>
<th>Exception</th>
<th>Flaky</th>
<th>% Flaky</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mochitest-plain-chunked</td>
<td>261,765</td>
<td>6,218</td>
<td>6</td>
<td>0</td>
<td>4,255</td>
<td>68.43</td>
<td>267,989</td>
</tr>
<tr>
<td>mochitest-browser-chrome-chunked</td>
<td>230,116</td>
<td>9,505</td>
<td>2</td>
<td>0</td>
<td>7,121</td>
<td>74.92</td>
<td>239,623</td>
</tr>
<tr>
<td>mochitest-mochitest-devtools-chrome-chunked</td>
<td>172,806</td>
<td>4,967</td>
<td>4</td>
<td>0</td>
<td>3,392</td>
<td>68.29</td>
<td>177,777</td>
</tr>
<tr>
<td>reftest-reftest</td>
<td>171,501</td>
<td>2,455</td>
<td>5</td>
<td>0</td>
<td>1,429</td>
<td>58.21</td>
<td>173,961</td>
</tr>
<tr>
<td>reftest</td>
<td>155,457</td>
<td>1,141</td>
<td>1,073</td>
<td>103</td>
<td>445</td>
<td>39.00</td>
<td>155,774</td>
</tr>
<tr>
<td>mochitest-mochitest-gl</td>
<td>126,853</td>
<td>988</td>
<td>73</td>
<td>0</td>
<td>335</td>
<td>48.69</td>
<td>127,614</td>
</tr>
<tr>
<td>xpshell-xpshell</td>
<td>119,596</td>
<td>2,158</td>
<td>1</td>
<td>0</td>
<td>760</td>
<td>35.22</td>
<td>121,755</td>
</tr>
<tr>
<td>mochitest-chrome</td>
<td>105,665</td>
<td>1,567</td>
<td>1,026</td>
<td>48</td>
<td>1,141</td>
<td>72.81</td>
<td>108,106</td>
</tr>
<tr>
<td>jsscript</td>
<td>103,024</td>
<td>488</td>
<td>976</td>
<td>139</td>
<td>182</td>
<td>38.32</td>
<td>104,614</td>
</tr>
<tr>
<td>reftest-reftest-no-accel</td>
<td>87,953</td>
<td>902</td>
<td>9</td>
<td>1</td>
<td>610</td>
<td>67.63</td>
<td>88,865</td>
</tr>
<tr>
<td>Marionette</td>
<td>68,747</td>
<td>870</td>
<td>452</td>
<td>74</td>
<td>600</td>
<td>68.97</td>
<td>68,143</td>
</tr>
<tr>
<td>mochitest-mochitest-media</td>
<td>59,990</td>
<td>3,418</td>
<td>3</td>
<td>0</td>
<td>2,991</td>
<td>87.51</td>
<td>64,411</td>
</tr>
<tr>
<td>xpshell</td>
<td>48,982</td>
<td>754</td>
<td>111</td>
<td>0</td>
<td>472</td>
<td>62.60</td>
<td>49,474</td>
</tr>
<tr>
<td>reftest-crashtest</td>
<td>35,704</td>
<td>397</td>
<td>5</td>
<td>0</td>
<td>91</td>
<td>22.92</td>
<td>36,106</td>
</tr>
<tr>
<td>mochitest-a11y</td>
<td>23,766</td>
<td>417</td>
<td>0</td>
<td>0</td>
<td>320</td>
<td>76.74</td>
<td>24,133</td>
</tr>
<tr>
<td>reftest-reftest-fons</td>
<td>19,170</td>
<td>229</td>
<td>4</td>
<td>0</td>
<td>128</td>
<td>55.90</td>
<td>19,403</td>
</tr>
<tr>
<td>mochitest-media</td>
<td>15,297</td>
<td>371</td>
<td>173</td>
<td>10</td>
<td>252</td>
<td>67.92</td>
<td>15,585</td>
</tr>
<tr>
<td>reftest-reftest-no-accel-fons</td>
<td>10,710</td>
<td>131</td>
<td>6</td>
<td>0</td>
<td>106</td>
<td>80.92</td>
<td>10,847</td>
</tr>
<tr>
<td>reftest-reftest-gpu-fons</td>
<td>2,970</td>
<td>118</td>
<td>5</td>
<td>0</td>
<td>93</td>
<td>78.81</td>
<td>3,093</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1,950,980</td>
<td>38,596</td>
<td>7,022</td>
<td>477</td>
<td>25,871</td>
<td>67.03</td>
<td>1,997,075</td>
</tr>
</tbody>
</table>

using future data. The evaluation part remains unseen during training and validation of the hyper-parameters, and models and is only used in the final evaluation. Figure 3 visualizes our approach handle the data for training and evaluation.

As each data set is highly imbalanced, we resample the data before training the models. For this purpose, we use over- and under-sampling techniques (specifically Smote+Tomek [7, 14]). We apply the sampling strategies after the split in the development parts to avoid data duplication across training and test sets. As undersampling potentially removes crucial data points from the majority class, we only undersample the validation part and oversample in the training part.

To improve the performance of our classification models, we optimize their hyper-parameters. We evaluate the performance of hyper-parameter combinations in a 10-fold Monte-Carlo cross-validation using stratified ShuffleSplits of the development set and perform a grid search [9] as well as Bayesian optimization [11, 50] to tune hyper-parameters. While grid search exhaustively combines hyper-parameters from a predefined grid of valid hyper-parameters [10], Bayesian optimization uses a probability model to find a new and potentially better set of hyper-parameters before
fitting and evaluating the model on these parameters and updating the model. During cross-validation, we optimize for precision (in contrast to the usual accuracy or ROC) as we are interested in a low false-positive rate in exchange for lower recall. Our goal is to provide engineers with a classifier that can be used to pass builds reliably without missing real test alarms. We accept that we potentially find less faulty cases and only classify build failures as intermittent with higher confidence. Once we optimized the hyper-parameters, we choose the best performing classifier and refit it using the optimized hyper-parameters on the development set. These classifiers will be used in the final pipeline.

4.2 Feature Impact

After successfully training tree ensembles for all 20 remaining test suites, we analyze the structure of the models to understand what affects their decisions. We compute and plot the feature impacts using SHAP (SHapley Additive exPlanations) [34, 36, 37]. The tree-explainer [35] computes how much each feature contributes to pushing the model output from the base value to the actual output. The base value is the average output over the whole training set. In this case, an output around or lower than 0 means regular failure, whereas 1 and above refers to an intermittent failure. Using force plots ordering samples by similarity, we identify certain areas or patterns giving insight on the contribution of the features towards the model output. To find these patterns, we cluster the SHAP values of the models using density-based spatial clustering of applications with noise (DBSCAN [19]), which is perfectly suitable for SHAP values. They present a non-flat geometry with uneven cluster sizes. DBSCAN groups together closely related points, i.e., points with lots of nearby neighbors, while at the same time finding low-density regions to flag outliers. Clustering SHAP values instead of feature values allows us to identify patterns where the same features have a similar impact on the decision even if the feature values are different. This knowledge will in turn help us when informing the developer about the feature that drove the decision.

As an example for a pattern, consider the force plot for the marionette test suite in Figure 4. Here, we find three dominant patterns—all featuring cpu_load and run_time as most impacting features, which even have sole decision power. These patterns only represent the model impact, but not the actual values of the features. To find the pattern itself, we perform inner clustering and variance analysis on the actual feature values within the clusters to find representative insights, such as: “high runtime and high cpu load are always symptoms for flakiness in test suite A”. We investigate representatives of these patterns and track these instances back to the original bug reports. With these bug reports, we investigate the underlying root cause of the failure and how it manifested itself. This way, we gain insights on whether we just found a coincidental correlation or can track the effect down to an observable cause with potential actionable insight.

4.3 Recommendations

As described in Section 1, we do not only want to provide developers with decisions, but also with reasons behind these decisions. For this purpose, we fit a SHAP-tree-explainer for each of our classifiers (Section 4.1). Using this explainer, we are now able to extract the SHAP-values, that is, feature importances for each decision our classifier makes. With these feature importances, we are now able to identify all features that actually contribute to the decision. These features are called impacting features.

Definition 4.1. Let \( F \) be the set of features for a single decision. Define the set of positive features as
\[
F^+ = \{ f \mid f \in F \land \text{shap}(f) > 0 \},
\]
and the set of negative features as
\[
F^- = \{ f \mid f \in F \land \text{shap}(f) < 0 \}.
\]
We define \( \phi^+ \), the set of impacting features for positive decisions, as
\[
\phi^+ = \min_{X \in S^+} |X|,
\]
with
\[
S^+ = \{ X \mid X \subset F^+ \land \left( b + \left( \sum_{x \in X} \text{shap}(x) \right) + \left( \sum_{y \in F^-} \text{shap}(y) \right) \right) \geq t \},
\]
and \( \phi^- \), the set of impacting features for negative decisions, as
\[
\phi^- = \min_{X \in S^-} |X|,
\]
with
\[
S^- = \{ X \mid X \subset F^- \land \left( b + \left( \sum_{y \in F^-} \text{shap}(y) \right) + \left( \sum_{x \in X} \text{shap}(x) \right) \right) < t \},
\]
where \( b \) is the base value of the classifier, \( \text{shap}(x) \) the SHAP-value of feature \( x \), and \( t \) the decision threshold.

For each decision we make, we now report the positive or negative impacting features alongside the decision and the original feature values.

5 EVALUATION

In this section, we evaluate our ability of classifying failing jobs as intermittent by computing precision-scores for all trained classification models, as well as our ability to find patterns, and our ability to infer reasons to our classification that help diagnosing the underlying root cause. For the evaluation, we only use the hold-out data as shown in Figure 3. Additionally to evaluating the performance of our classifiers, we conducted interviews with Mozilla sheriffs to validate our findings and gain more insight into the nature of intermittent failures. In this section, we answer the following research questions:

RQ1: Can one classify job failures as intermittent achieving high precision scores using job telemetry data only?
RQ2: Can one identify intermittent job failures that are not caused by a flaky test?
RQ3: Can one infer patterns for intermittent failures based on the features used for the classification to diagnose the underlying root cause?
RQ4: Can one automatically infer reasons hinting at underlying root causes for intermittent failures?
Figure 4: Feature impact visualization for test suite reftest. The partial visualization of the stacked force plot presents three different patterns. The patterns 1 and 2 push the model output towards an output of zero resulting in a strong confidence for regular failures while pattern 3 pushes the output close to 1.0, strongly suggesting an intermittent failure.

Table 3: Results of the classification experiments. Many test suites present high precision values while still maintaining high recall values.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>mochitest-plain-chunked</td>
<td>41.47%</td>
<td>80.52%</td>
<td>0.54</td>
<td>589 / 344</td>
</tr>
<tr>
<td>mochitest-browser-chrome-chunked</td>
<td>68.51%</td>
<td>85.13%</td>
<td>0.75</td>
<td>511 / 915</td>
</tr>
<tr>
<td>mochitest-mochitest-devtools-chunked</td>
<td>77.42%</td>
<td>84.44%</td>
<td>0.79</td>
<td>245 / 501</td>
</tr>
<tr>
<td>reftest-reftest</td>
<td>69.56%</td>
<td>66.66%</td>
<td>0.68</td>
<td>129 / 240</td>
</tr>
<tr>
<td>reftest</td>
<td>92.98%</td>
<td>91.90%</td>
<td>0.92</td>
<td>14 / 173</td>
</tr>
<tr>
<td>mochitest</td>
<td>72.12%</td>
<td>93.14%</td>
<td>0.81</td>
<td>113 / 175</td>
</tr>
<tr>
<td>mochitest-mochitest-gl</td>
<td>81.48%</td>
<td>81.48%</td>
<td>0.81</td>
<td>50 / 54</td>
</tr>
<tr>
<td>xpcshell-xpcshell</td>
<td>34.69%</td>
<td>61.81%</td>
<td>0.34</td>
<td>269 / 55</td>
</tr>
<tr>
<td>mochitest-chrome</td>
<td>78.01%</td>
<td>82.32%</td>
<td>0.80</td>
<td>62 / 181</td>
</tr>
<tr>
<td>jsreftest</td>
<td>86.30%</td>
<td>80.77%</td>
<td>0.83</td>
<td>15 / 78</td>
</tr>
<tr>
<td>reftest-reftest-no-accel</td>
<td>71.26%</td>
<td>84.93%</td>
<td>0.77</td>
<td>63 / 73</td>
</tr>
<tr>
<td>marionette</td>
<td>57.14%</td>
<td>87.50%</td>
<td>0.69</td>
<td>78 / 64</td>
</tr>
<tr>
<td>mochitest-mochitest-media</td>
<td>80.56%</td>
<td>93.48%</td>
<td>0.86</td>
<td>114 / 339</td>
</tr>
<tr>
<td>xpcshell</td>
<td>94.67%</td>
<td>81.61%</td>
<td>0.88</td>
<td>27 / 87</td>
</tr>
<tr>
<td>reftest-crashtest</td>
<td>50.00%</td>
<td>50.00%</td>
<td>0.50</td>
<td>56 / 4</td>
</tr>
<tr>
<td>mochitest-a11y</td>
<td>50.00%</td>
<td>85.18%</td>
<td>0.63</td>
<td>36 / 27</td>
</tr>
<tr>
<td>reftest-reftest-fonts</td>
<td>100.00%</td>
<td>85.00%</td>
<td>0.92</td>
<td>15 / 20</td>
</tr>
<tr>
<td>mochitest-media</td>
<td>55.56%</td>
<td>83.33%</td>
<td>0.67</td>
<td>52 / 6</td>
</tr>
<tr>
<td>reftest-reftest-no-accel-fonts</td>
<td>100.00%</td>
<td>85.00%</td>
<td>0.90</td>
<td>0 / 20</td>
</tr>
<tr>
<td>reftest-reftest-gpu-fonts</td>
<td>100.00%</td>
<td>94.44%</td>
<td>0.97</td>
<td>0 / 18</td>
</tr>
</tbody>
</table>

5.1 Classification

Table 3 contains the results of the classification. The table shows high precision and recall values for most test suites. Some test suites (e.g., reftest-crashtest, and xpcshell-xpcshell) performed exceptionally bad. This suggests that the features we used are not suitable to classify intermittent failures for these test suites. For reftest-crashtest, further investigation into its nature and repeated discussions with Mozilla engineers led to the following explanation: reftest-crashtest is a test suite that has a significantly less complicated setup than other test suites and is therefore less influenced by networking issues or missing permissions than other test suites. Since this test suite is no longer undergoing changes, it is safe to assume that the chosen features are simply not suitable to classify intermittent failures. For reftest, or reftest-reftest-fonts, we observe exceptionally high precision values. This shows that the chosen features perform especially well on these test suites. This is also supported by findings of intermittent failures often being caused by timeouts or resource starvation, as confirmed by Mozilla developers. For the other test suites, we still see that there is some predictive power by the features used in this study. Another fact that can be observed in our results in Table 3 is that amongst the worst performing test suites are chunked test suites. Chunked test suites refer to test suites for which the number of tests contained exceeds a certain threshold. These tests are chunked into smaller test suites. These chunks, however, are volatile. Whenever a new test is added to a chunked test suite or an old one is removed, chunks might change as tests can move between chunks. This means that, in contrast to all other test suites, the baseline can change resulting in a drastic effect on the cpu_load/run_time making these metrics incomparable.
Figure 5: Summary plot for the marionette test suite. The real-valued features run_time and cpu_load dominate the decision of the classifier whereas the categorical values have barely any effect. Red represents a high feature value, blue a low feature value.

Answer to RQ1: Automated methods for classifying job failures achieve precision scores of up to 100% (73% on average) as well as recall scores of up to 94% (82% on average). This shows that our selected features have predictive power. Furthermore, test suites for which the precision is exceptionally high, fully automated intermittent failure classification can be made available. For other test suites, one should use our feedback as guidance, when manually classifying job failures.

5.2 Model Inspection

Using SHAP, we compute the feature importances for our classifications. This is done to identify the features driving the decisions. We cluster the computed feature importances as described in Section 4.2. The resulting patterns show that, for most test suites, the two real-valued features run_time and cpu_load have by far the highest impact in the models. After performing inner clustering, we see that high run_time and low cpu_load are the strongest indicators, by far. The case for cpu_load can be inverted. Figure 7 shows a summary plot for the regression test suite mochitest, containing JavaScript tests. As we can see, flaky tests tend to have a high cpu_load and run_time. Given these patterns, we investigated Bugzilla bug-reports for the classified failures and asked Mozilla engineers whether these patterns would help them understand the underlying issue. Our investigations as well as the conversations with engineers yield the following explanations:

Long Run Time Mozilla engineers pointed out that many of the intermittent test failures observed by them are resulting from tests timing out. This is in line with our observations. Timeouts can be caused by a number of underlying root causes, including networking issues, overloaded VM’s, or resources being unavailable.

Low CPU Load In our interviews with Mozilla engineers, we were often pointed to tests failing early because required permissions were missing. This turned out to be one of the main issues Mozilla has with regards to intermittent failures.

High CPU Load When studying bug reports on intermittent failures, we were regularly confronted with test jobs failing intermittently when processing large numbers of files. Mozilla engineers pointed us to bug reports showing that the testing engine was often crashing under these circumstances. This means that there is not necessarily a specific test failing, but rather the testing framework itself.

The explanations we obtained from Mozilla, show that it is not always a flaky test itself causing jobs to fail intermittently. In many cases the test infrastructure itself is responsible for the failure at hand. With this finding, we are able to answer RQ2:

Answer to RQ2: Many intermittent failures are caused by missing permissions, timeouts, and the like. This means that there is not always a specific test responsible for the failure at hand. Our classifiers are able to detect intermittent job failures even and especially in cases where the root cause cannot be traced back to a flaky test itself. This is somethings other tools are not able to do by design.

In some cases, the categorical features barely have any influence on the classification and do not contribute to the model output as shown in the summary plot in Figure 5. The plot shows, that while there is a clear separation between classes for most categorical features, their impact is rather small. However, there are some test suites where a single categorical feature significantly influences the decision of the classifier (Figure 6). We assume that differences in test suites also lead to differences in the root causes of their intermittent failures. These root causes might not be resource related. At Mozilla, the test suites are very different. The mochitest test suite tests APIs accessible to Web pages, whereas reftest is testing the layout rendering by comparing images, and jsreftest tests the JavaScript compliance. crashtest loads scenarios that caused crashes in previous versions of the browser.

Figure 6: Partial summary plot for the mochitest-chrome test suite. The run_time shows no consistent contribution towards the models output, but the tests running on Windows10 64bit is consistently the strongest indicator.

Figure 7: Summary plot for the mochitest test suite. A high cpu_load pushes the model output towards an intermittent failure verdict.
**Answer to RQ3:** While we did not identify an overarching pattern that holds for all test suites, we are able to automatically infer patterns for intermittent failures using our trained models. These patterns capture an important aspect of the underlying issue and can thus help to diagnose the root cause of the failure at hand. Furthermore, as there is no overarching pattern for all test suites, being able to identify these patterns automatically is important.

In our interviews, Mozilla engineers confirmed that insights we obtained from our patterns will be helpful when trying to identify root causes of intermittent failures. It helps also in assessing the validity of the classification.

**Answer to RQ4:** We are able to infer patterns for our classifications. These patterns are in line with problem causes identified by Mozilla and capture an important aspect of the root cause of an intermittent failure. When we compute SHAP-values for a new failure classification, these values follow the same patterns as shown before. This, in turn, can help sheriffs identify the underlying root cause as confirmed by interviews at Mozilla.

### 6 DISCUSSION

Our evaluation shows that we are not just able to classify failures as intermittent, but also that we can gain insights into the origin of the intermittent failure. In this section, we want to discuss our findings of this paper in terms of their potential impact on practice, as well as the generalizability of our findings.

#### 6.1 Impact at Mozilla

As of today, classification of job failures at Mozilla relies heavily on the help of sheriffs, who investigate failures and classify them as intermittent or regular. Table 4 shows that this classification can take a significant amount of time. As seen in Table 3, we achieved high precision for some test suites for which we are able to classify test failures reliably within less than one second and thus significantly faster than sheriffs will ever be. This approach also allows engineers to optimize the performance of the models by choosing a different threshold for the classification model. While precision values for some test suites allow fully for automated job failure classification, others still need sheriffs to make the final decision. For these jobs, we can observe high recall values. High recall values are desirable in this context as they will notify sheriffs about most of the jobs that may have failed intermittently. For these classifications, sheriffs can now rely on the patterns we identified. We conclude that our approach would considerably speed up the classification and assist sheriffs by providing relevant information and insights to facilitate their work which would in turn lead to a faster recognition of real failures. Furthermore, engineers are hinted into a direction as to why a job failed intermittently and might thus be able to fix the underlying problem faster. We conclude that our approach has the potential to significantly improve the Mozilla CI process in terms of dealing with job failures as a whole.

#### 6.2 Generalizability

While for many test suites, run_time and cpu_load are good and reliable predictors, the reason for these two features predicting intermittent failures at Mozilla varies between test suites. Other features might actually be better predictors for intermittent failures as run_time or cpu_load depending on the test suite we are currently working with. This shows that there is no clear, overarching pattern for intermittently failing jobs as we have confirmed at Mozilla. Hence, the results for good predictors and the underlying root causes do not generalize. Still, despite working with a very diverse set of test suites, we are able to extract recurring patterns for failing test suites. These patterns can be used as a means to find the underlying root cause of the failure. Early results with data from other companies indicate that this approach will also work outside of Mozilla.

### 7 THREATS TO VALIDITY

Our study showed that we are able to classify intermittent test failures using solely telemetry data, and that it is possible to use this classification to get insights into the actual reason for the test to fail intermittently. While these results look very promising, let us point out limitations and threats to validity.

**External validity.** The whole research presented in this paper is solely based on Mozilla test data, the Mozilla CI process, and insights gained from talking to Mozilla employees. This could be a limiting factor when applying this approach to other CI processes. While we are confident that it is possible to classify job failures as intermittent in other setups as well, given enough telemetry data are collected, we have no insight to which extent we would be able to gain knowledge on the actual cause of an intermittent failure. Still, we are confident that our observations on the CI process and their requirements also hold for other high-profile industrial software development.

**Internal validity.** We assume that all manual classifications done by sheriffs are correct. We, however, lack the means to evaluate how good a manual classification by humans is. This would require a user study on the accuracy of the classification performed by multiple sheriffs or independent engineers and investigate their agreement, which was not feasible with our arrangement at Mozilla.

### 8 RELATED WORK

**Surveys of Job Flakiness.** Luo et al. conducted a study on the root causes of flaky tests [38]. They found that async wait, concurrency, and test order dependencies are among the most common root causes for flaky tests. Another finding was that most of the root causes were platform independent. Most flaky failures caused by concurrency, are due to concurrent memory access, while about half of test order dependency issues are caused by dependencies on external resources. Jiang et al. [29] use information retrieval techniques to find test alarm causes. In 2017, Guan et al. presented an approach to suppress false alarms, which was based on k-nearest neighbor [23]. Listfield searched for correlations between test size, RAM usage and flakiness in Google’s tests [33]. He found that 1.5% of the test runs at Google fail intermittently, and that test size as well as the tool, with which the tests are written strongly correlate with flakiness. The present work enriches our knowledge by adding an additional set of data points from industrial practice.
Intermittent failures are a significant obstacle in continuous integration (CI) processes. Using job telemetry data, we are able not only to classify job failures as intermittent, but also to derive indicators towards the underlying root cause. We showed that even simple approaches not requiring re-runs are [6, 43], both using machine learning techniques to classify tests as flaky. Other approaches like Bell et al. presented DeFlaker, a tool to automatically detect flaky tests [8]. Their approach is known to be flaky (also referred to as quarantining) [20]. At Google, the two most frequently used techniques are quarantining and re-running flaky tests [39]; 16% of tests are flaky tests and they spend up to 16% of their computing resources on re-running flaky tests [40]. While quarantining may be a workaround when applied complementary to current approaches, it also be able to identify patterns for intermittent failures at other companies as well. Furthermore, quarantining entire jobs, rather than single tests, is not an option, which we address in this work.

Automated Detection & Classification. Bell et al. presented DeFlaker, a tool to automatically detect flaky tests [8]. Their approach is based on differential coverage, without re-running the test. Other approaches not requiring re-runs are [6, 43], both using machine learning techniques to classify tests as flaky. Other approaches like the one presented in [41], or chaos mode [1] try to create an environment in which a newly introduced test is stress tested in order to reveal flakiness. All these approaches focus on identifying flaky tests while we focus on the classification of intermittent failures. Furthermore, intermittent failures can also be caused by reasons other than a flaky test. Our approach is also be able to deal with jobs that fail for reasons other than flaky tests.

9 CONCLUSION AND FUTURE WORK

Intermittent failures are a significant obstacle in continuous integration (CI) processes. Using job telemetry data, we are able not only to classify job failures as intermittent, but also to derive indicators towards the underlying root cause. We showed that even simple
correlations between certain features and flakiness are sufficient to classify failures with a high degree of precision and virtually no resources or time consumed at the time the failure occurs. While the classification models still need to be trained, this can be done independently of the running CI pipelines. Our evaluation indicates that our approach would significantly improve over state of the art when applied complementary to current approaches.

Intermittent failures and flaky tests are plaguing developers for years and we are aware that there is no single optimal solution for it. During the course of our work, we discovered follow-up questions and challenges that need further investigation:

As test suites continuously change, models will inevitably deprecate over time. Further research is required to understand the pace at which the models are deprecating and how frequently they need to be retrained to keep up with changes in the test suites or to the ecosystem. Using larger data sets, we might be able to understand how often models have to be re-trained. Furthermore, with more features, we might be able to predict intermittent failures for test suites that do not have a good predictor yet. In addition to building a full-fledged classification pipeline integrated into Mozilla's CI process, we also want to use telemetry data from other companies to verify that this approach will also be applicable in their CI process. Early experiments indicate that using this approach we will also be able to identify patterns for intermittent failures at other companies as well.

ACKNOWLEDGMENTS

We thank Christian Holler and Joel Maher at Mozilla for their continuous support during this study.

### Table 4: Overview of the classification time for sheriffs.

<table>
<thead>
<tr>
<th>Test Suite</th>
<th>Count</th>
<th>Mean</th>
<th>Std</th>
<th>Min</th>
<th>Q1</th>
<th>Median</th>
<th>Q3</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mochitest-plain-chunked</td>
<td>4,255</td>
<td>64.01 m</td>
<td>8.15 h</td>
<td>7 s</td>
<td>3.64 m</td>
<td>10.95 m</td>
<td>28.80 m</td>
<td>6.41 d</td>
</tr>
<tr>
<td>mochitest-browser-chrome-chunked</td>
<td>7,121</td>
<td>63.06 m</td>
<td>6.11 h</td>
<td>8 s</td>
<td>3.45 m</td>
<td>10.58 m</td>
<td>29.32 m</td>
<td>7.17 d</td>
</tr>
<tr>
<td>mochitest-mochitest-chrome-chunked</td>
<td>3,392</td>
<td>59.10 m</td>
<td>6.74 h</td>
<td>7 s</td>
<td>3.42 m</td>
<td>10.32 m</td>
<td>29.82 m</td>
<td>6.20 d</td>
</tr>
<tr>
<td>reftest-reftest</td>
<td>1,429</td>
<td>4.38 h</td>
<td>20.87 h</td>
<td>11 s</td>
<td>4.42 m</td>
<td>14.38 m</td>
<td>44.40 m</td>
<td>7.55 d</td>
</tr>
<tr>
<td>reftest</td>
<td>445</td>
<td>80.28 m</td>
<td>5.60 h</td>
<td>13 s</td>
<td>4.55 m</td>
<td>11.57 m</td>
<td>44.63 m</td>
<td>2.80 d</td>
</tr>
<tr>
<td>mochitest</td>
<td>1,148</td>
<td>30.32 m</td>
<td>95.60 m</td>
<td>6 s</td>
<td>3.01 m</td>
<td>10.25 m</td>
<td>26.04 m</td>
<td>41.16 h</td>
</tr>
<tr>
<td>mochitest-mochitest-gl</td>
<td>335</td>
<td>87.53 m</td>
<td>4.72 h</td>
<td>13 s</td>
<td>5.97 m</td>
<td>18.15 m</td>
<td>60.13 m</td>
<td>2.33 d</td>
</tr>
<tr>
<td>xpcshell-xpcshell</td>
<td>760</td>
<td>45.50 m</td>
<td>3.50 h</td>
<td>19 s</td>
<td>4.40 m</td>
<td>13.12 m</td>
<td>37.10 m</td>
<td>3.35 d</td>
</tr>
<tr>
<td>mochitest-chrome</td>
<td>1,141</td>
<td>41.11 m</td>
<td>6.09 h</td>
<td>16 s</td>
<td>3.05 m</td>
<td>8.72 m</td>
<td>25.10 m</td>
<td>6.29 d</td>
</tr>
<tr>
<td>jsreftest</td>
<td>182</td>
<td>3.24 h</td>
<td>18.43 h</td>
<td>17 s</td>
<td>5.01 m</td>
<td>15.91 m</td>
<td>46.23 m</td>
<td>31.07 h</td>
</tr>
<tr>
<td>reftest-reftest-no-accel</td>
<td>610</td>
<td>52.44 m</td>
<td>2.38 h</td>
<td>17 s</td>
<td>3.01 m</td>
<td>11.65 m</td>
<td>35.92 m</td>
<td>6.70 d</td>
</tr>
<tr>
<td>marionette</td>
<td>600</td>
<td>2.62 h</td>
<td>25.00 h</td>
<td>13 s</td>
<td>4.78 m</td>
<td>17.27 m</td>
<td>58.95 m</td>
<td>24.27 d</td>
</tr>
<tr>
<td>mochitest-mochitest-media</td>
<td>2,991</td>
<td>45.90 m</td>
<td>3.60 h</td>
<td>10 s</td>
<td>3.69 m</td>
<td>11.75 m</td>
<td>32.72 m</td>
<td>4.90 d</td>
</tr>
<tr>
<td>xpcshell</td>
<td>472</td>
<td>36.15 m</td>
<td>2.34 h</td>
<td>13 s</td>
<td>3.13 m</td>
<td>8.97 m</td>
<td>25.51 m</td>
<td>46.43 h</td>
</tr>
<tr>
<td>reftest-crashtest</td>
<td>91</td>
<td>59.88 m</td>
<td>2.07 h</td>
<td>25 s</td>
<td>3.84 m</td>
<td>15.27 m</td>
<td>49.40 m</td>
<td>11.42 h</td>
</tr>
<tr>
<td>mochitest-a11y</td>
<td>320</td>
<td>4.58 h</td>
<td>2.31 h</td>
<td>23 s</td>
<td>3.12 m</td>
<td>9.50 m</td>
<td>33.33 m</td>
<td>16.17 h</td>
</tr>
<tr>
<td>reftest-reftest-fonts</td>
<td>128</td>
<td>49.79 m</td>
<td>94.08 m</td>
<td>30 s</td>
<td>5.24 m</td>
<td>19.85 m</td>
<td>43.65 m</td>
<td>13.36 h</td>
</tr>
<tr>
<td>mochitest-media</td>
<td>252</td>
<td>83.89 m</td>
<td>2.66 m</td>
<td>12 s</td>
<td>4.60 m</td>
<td>28.18 m</td>
<td>65.43 m</td>
<td>16.53 h</td>
</tr>
<tr>
<td>reftest-reftest-no-accel-fonts</td>
<td>106</td>
<td>78.69 m</td>
<td>2.84 h</td>
<td>37 s</td>
<td>6.04 m</td>
<td>18.52 m</td>
<td>56.23 m</td>
<td>20.15 h</td>
</tr>
<tr>
<td>reftest-reftest-gpu-fonts</td>
<td>93</td>
<td>51.91 m</td>
<td>1.74 h</td>
<td>36 s</td>
<td>4.12 m</td>
<td>14.48 m</td>
<td>43.25 m</td>
<td>10.03 h</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>25,871</td>
<td>71.11 m</td>
<td>8.34 h</td>
<td>6.0 s</td>
<td>3.38 m</td>
<td>11.12 m</td>
<td>31.42 m</td>
<td>24.27 d</td>
</tr>
</tbody>
</table>

Quarantining. One approach to mitigating flaky tests would be disabling tests known to be flaky (also referred to as quarantining) [20]. At Google, the two most frequently used techniques are quarantining and re-running flaky tests [39]; 16% of tests are flaky tests and they spend up to 16% of their computing resources on re-running flaky tests [40]. While quarantining may be a workaround for intermittently failing tests, the problem is that tests covering critical behavior of the software system may be skipped and that this behavior might not be covered by other tests. Furthermore, quarantining entire jobs, rather than single tests, is not an option, which we address in this work.
REFERENCES


August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. 80–90. https://doi.org/10.1109/ICST.2016.40


