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

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473931

ACM Reference Format:
Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. When Life Gives You Oranges: Detecting and Diagnosing Intermittent Job Failures at Mozilla. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. 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, we collect 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 train 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 interpret 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:

http://bit.ly/IntermittentJobs

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 OrangeFactor 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.

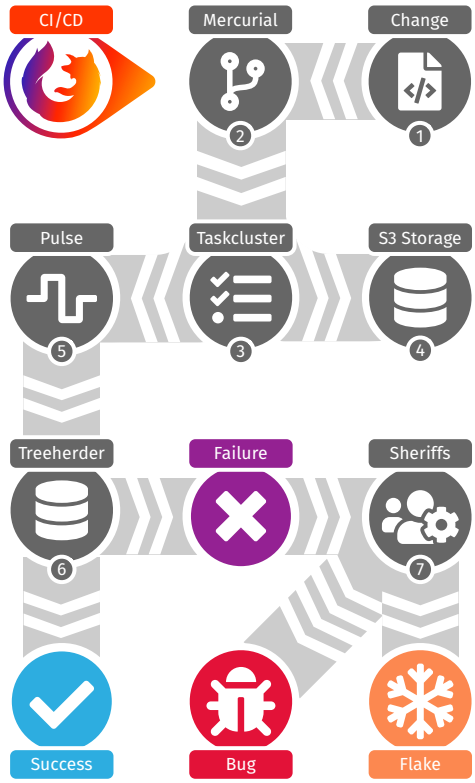

Figure 1: The Mozilla CI system. Incoming changes (1) are being pushed to the `mozilla-inbound` repository (2). The Taskcluster (3) service commissions machines and performs the build and test tasks. The results are persisted to AWS S3 storage in JSON format and emitted as telemetry via Pulse (5). Treeherder 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 intermittent 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 selection. 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* kinds 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 analysis 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.

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

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

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[1]), 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

[1] https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Building_with_Profile-Guided_Optimization

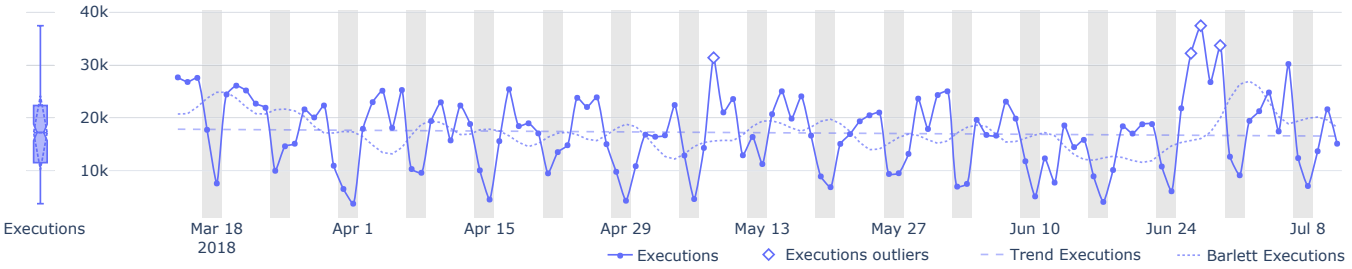

Figure 2: Mozilla's job runs per day. On average, the number of builds appears to be stable. On week-ends (highlighted in gray), the number drops to about 5,000. During the week, Mozilla runs up to ~40,000 test jobs a day.

Table 2: Overview of the data sets: Test suite runs in Mozilla's Treeherder database. Job results in *Success*/*Failure* refer to test suite execution results. *Retries* reflect automatic retries in certain error scenarios. Test suite runs with an *exception* represent an error in the build system.

Test Suite	# Success	# Failure	# Retry	# Exception	# Flaky	% Flaky	Total
mochitest-plain-chunked	261,765	6,218	6	0	4,255	68.43	267,989
mochitest-browser-chrome-chunked	230,116	9,505	2	0	7,121	74.92	239,623
mochitest-mochitest-devtools-chrome-chunked	172,806	4,967	4	0	3,392	68.29	177,777
reftest-reftest	171,501	2,455	5	0	1,429	58.21	173,961
reftest	153,457	1,141	1,073	103	445	39.00	155,774
mochitest	135,108	1,815	3,088	102	1,148	63.25	140,113
mochitest-mochitest-gl	126,853	688	73	0	335	48.69	127,614
xpcshell-xpcshell	119,596	2,158	1	0	760	35.22	121,755
mochitest-chrome	105,465	1,567	1,026	48	1,141	72.81	108,106
jsreftest	103,024	475	976	139	182	38.32	104,614
reftest-reftest-no-accel	87,953	902	9	1	610	67.63	88,865
marionette	66,747	870	452	74	600	68.97	68,143
mochitest-mochitest-media	59,990	3,418	3	0	2,991	87.51	63,411
xpcshell	48,982	754	111	0	472	62.60	49,847
reftest-crashtest	35,704	397	5	0	91	22.92	36,106
mochitest-a11y	23,766	417	0	0	320	76.74	24,183
reftest-reftest-fonts	19,170	229	4	0	128	55.90	19,403
mochitest-media	15,297	371	173	10	252	67.92	15,851
reftest-reftest-no-accel-fonts	10,710	131	6	0	106	80.92	10,847
reftest-reftest-gpu-fonts	2,970	118	5	0	93	78.81	3,093
Total	1,950,980	38,596	7,022	477	25,871	67.03	1,997,075

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 under-sampling 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]

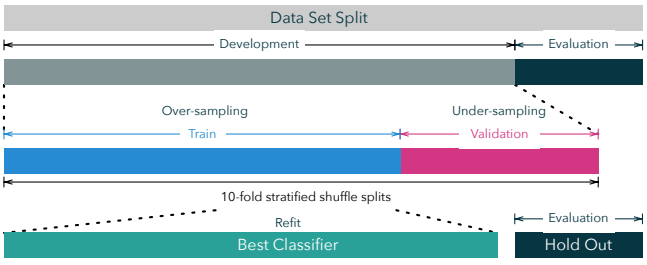

Figure 3: Training and evaluation of a classifier.

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 flaky failures 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

$$F^+ = \{ f \mid f \in F \wedge \text{shap}(f) > 0 \},$$

and the set of negative features as

$$F^- = \{ f \mid f \in F \wedge \text{shap}(f) < 0 \}.$$

We define ϕ^+, the set of impacting features for positive decisions, as

$$\phi^+ = \min_{X \in S^+} |X|,$$

with

$$S^+ = \left\{ X \mid X \subset F^+ \wedge \left(b + \left(\sum_{x \in X} \text{shap}(x) \right) + \left(\sum_{y \in F^-} \text{shap}(y) \right) \right) \geq t \right\},$$

and ϕ^-, the set of impacting features for negative decisions, as

$$\phi^- = \min_{X \in S^-} |X|,$$

with

$$S^- = \left\{ X \mid X \subset F^- \wedge \left(b + \left(\sum_{y \in F^+} \text{shap}(y) \right) + \left(\sum_{x \in X} \text{shap}(x) \right) \right) < t \right\},$$

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?

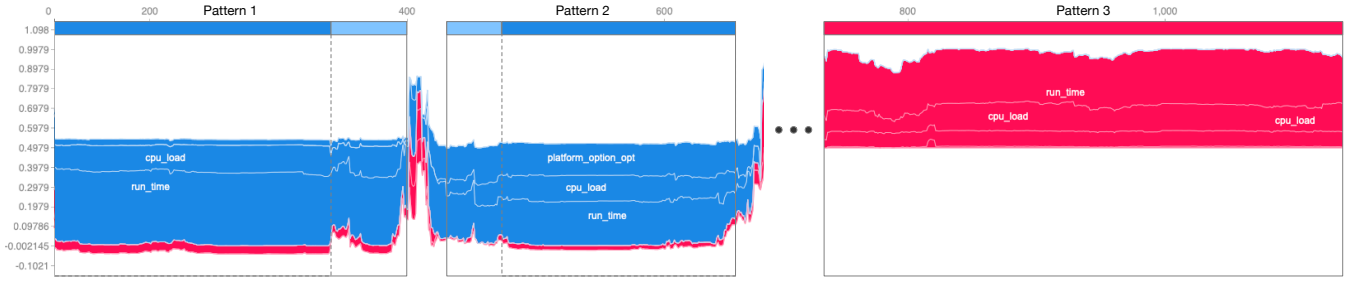

Figure 4: Feature impact visualization for test suite *reftest*. The partial visualization of the stacked force plot presents three different patterns. The patterns 1 and 2 push the model output towards an output of zero resulting in a strong confidence for regular failures while pattern 3 pushes the 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.

Test Suite	Precision	Recall	F-Score	Support
mochitest-plain-chunked	41.47%	80.52%	0.54	589 / 344
mochitest-browser-chrome-chunked	68.51%	85.13%	0.75	511 / 915
mochitest-mochitest-devtools-chrome-chunked	77.42%	81.44%	0.79	245 / 501
reftest-reftest	69.56%	66.66%	0.68	129 / 240
reftest	92.98%	91.90%	0.92	14 / 173
mochitest	72.12%	93.14%	0.81	113 / 175
mochitest-mochitest-gl	81.48%	81.48%	0.81	50 / 54
xpcshell-xpcshell	34.69%	61.81%	0.34	269 / 55
mochitest-chrome	78.01%	82.32%	0.80	62 / 181
jsreftest	86.30%	80.77%	0.83	15 / 78
reftest-reftest-no-accel	71.26%	84.93%	0.77	63 / 73
marionette	57.14%	87.50%	0.69	78 / 64
mochitest-mochitest-media	80.56%	93.48%	0.86	114 / 339
xpcshell	94.67%	81.61%	0.88	27 / 87
reftest-crashtest	50.00%	50.00%	0.50	56 / 4
mochitest-a11y	50.00%	85.18%	0.63	36 / 27
reftest-reftest-fonts	100.00%	85.00%	0.92	15 / 20
mochitest-media	55.56%	83.33%	0.67	52 / 6
reftest-reftest-no-accel-fonts	100.00%	85.00%	0.90	0 / 20
reftest-reftest-gpu-fonts	100.00%	94.44%	0.97	0 / 18

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 exist to facilitate test selection and lower the total number of tests that need to be executed. Therefore, 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.

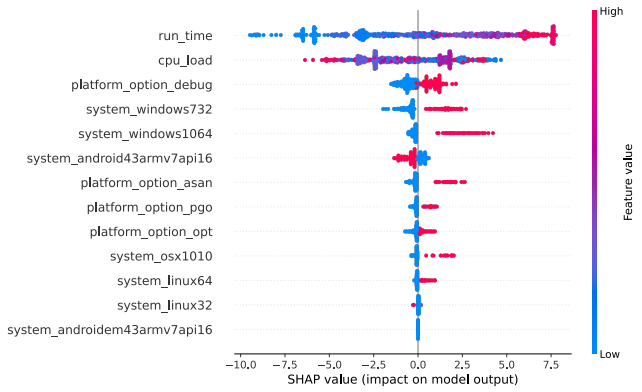

Figure 5: Summary plot for the *marionette* test suite. The real-valued features `run_time` and `cpu_load` dominate the decision of the classifier whereas the categorical values have barely any effect. Red represents a high feature value, blue a low feature value.

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

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.

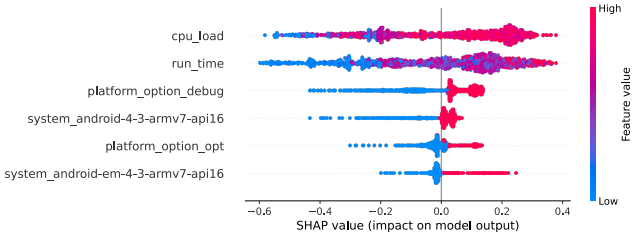

Figure 7: Summary plot for the *mochitest* test suite. A high `cpu_load` pushes the model output towards an intermittent failure verdict.

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.

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.

Table 4: Overview of the classification time for sheriffs.

Test Suite	Count	Mean	Std	Min	Q1	Median	Q3	Max
mochitest-plain-chunked	4,255	64.01 m	8.15 h	7 s	3.64 m	10.95 m	28.80 m	6.41 d
mochitest-browser-chrome-chunked	7,121	63.06 m	6.11 h	8 s	3.48 m	10.58 m	29.32 m	7.17 d
mochitest-mochitest-devtools-chrome-chunked	3,392	59.10 m	6.74 h	7 s	3.42 m	10.32 m	29.82 m	6.20 d
reftest-reftest	1,429	4.38 h	20.87 h	11 s	4.42 m	14.38 m	44.40 m	7.55 d
reftest	445	80.28 m	5.60 h	13 s	4.55 m	11.57 m	44.63 m	2.80 d
mochitest	1,148	30.32 m	95.60 m	6 s	3.01 m	10.25 m	26.04 m	41.16 h
mochitest-mochitest-gl	335	87.53 m	4.72 h	13 s	5.97 m	18.15 m	60.13 m	2.33 d
xpcshell-xpcshell	760	45.50 m	3.50 h	19 s	4.40 m	13.12 m	37.10 m	3.35 d
mochitest-chrome	1,141	41.11 m	6.09 h	16 s	3.03 m	8.72 m	25.10 m	6.29 d
jsreftest	182	3.24 h	18.43 h	16 s	3.86 m	11.65 m	35.92 m	6.70 d
reftest-reftest-no-accel	610	52.44 m	2.38 h	17 s	5.01 m	15.91 m	46.23 m	31.07 h
marionette	600	2.62 h	25.00 h	13 s	4.78 m	17.27 m	58.95 m	24.27 d
mochitest-mochitest-media	2,991	45.90 m	3.60 h	10 s	3.69 m	11.75 m	32.72 m	4.90 d
xpcshell	472	36.15 m	2.34 h	13 s	3.13 m	8.97 m	25.51 m	46.43 h
reftest-crashtest	91	59.88 m	2.07 h	25 s	3.84 m	15.27 m	49.40 m	11.42 h
mochitest-a11y	320	44.58 m	2.31 h	23 s	3.12 m	9.50 m	33.33 m	16.17 h
reftest-reftest-fonts	128	45.79 m	94.08 m	30 s	5.24 m	19.85 m	43.65 m	13.36 h
mochitest-media	252	33.24 m	88.59 m	12 s	2.66 m	8.17 m	28.18 m	16.53 h
reftest-reftest-no-accel-fonts	106	78.69 m	2.84 h	37 s	6.04 m	18.52 m	56.23 m	20.15 h
reftest-reftest-gpu-fonts	93	51.91 m	1.74 h	36 s	4.12 m	14.48 m	43.25 m	10.03 h
Total	25,871	71.11 m	8.34 h	6.0 s	3.38 m	11.12 m	31.42 m	24.27 d

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 rerunning 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.

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.

REFERENCES

[1] [n.d.]. Introducing Chaos Mode. https://robert.ocallahan.org/2014/03/introducing-chaos-mode.html. Accessed: 2019-06-18.
[2] [n.d.]. Mercurial source control management. https://www.mercurial-scm.org. Accessed: 2018-10-02.

[3] [n.d.]. TotT: Avoiding Flakey Tests. https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html. Accessed: 2018-10-02.

[4] 2015. *Efficient dependency detection for safe Java test acceleration.*

[5] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2019. Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners' Perceptions. *CoRR* abs/1906.00673 (2019). arXiv:1906.00673 http://arxiv.org/abs/1906.00673

[6] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *Proceedings of the 2021 International Conference on Software Engineering (ICSE)*. <ahref="https://jonbell.net/publications/flakeflagger">https://jonbell.net/publications/flakeflagger

[7] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. 2004. A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. *SIGKDD Explor. Newsl.* 6, 1 (June 2004), 20–29. https://doi.org/10.1145/1007730.1007735

[8] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the 2018 International Conference on Software Engineering (ICSE 2018)*. http://jonbell.net/publications/deflaker

[9] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-parameter Optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems* (Granada, Spain) *(NIPS'11)*. Curran Associates Inc., USA, 2546–2554. http://dl.acm.org/citation.cfm?id=2986459.2986743

[10] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2188385.2188395

[11] J. Bergstra, D. Yamins, and D. D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, I–115–I–123.

[12] B. W. Boehm and P. N. Papaccio. 1988. Understanding and controlling software costs. *IEEE Transactions on Software Engineering* 14, 10 (Oct 1988), 1462–1477. https://doi.org/10.1109/32.6191

[13] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (Oct. 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[14] Nitesh V. Chawla. 2005. Data Mining for Imbalanced Datasets: An Overview. *Data Mining and Knowledge Discovery Handbook* (2005), 853–867.

[15] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[16] Microsoft Corp. [n.d.]. LightGBM, Light Gradient Boosting Machine. https://github.com/Microsoft/LightGBM. Accessed: 2018-10-13.

[17] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. https://doi.org/10.1145/3338906.3338945

[18] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *SEIP* (seip ed.). IEEE - Institute of Electrical and Electronics Engineers. https://www.microsoft.com/en-us/research/publication/cloudbuild-microsofts-distributed-and-caching-build-service/

[19] M Ester, H P Kriegel, J Sander, X Xu Kdd, and 1996. [n.d.]. A density-based algorithm for discovering clusters in large spatial databases with noise. *aaai.org* ([n. d.]).

[20] Martin Fowler. 2011. Eradicating Non-Determinism in Tests. https://martinfowler.com/articles/nonDeterminism.html. Accessed: 2018-10-02.

[21] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification (ICST 2018)*. http://jonbell.net/publications/pradet

[22] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. 2017. CUT: Automatic Unit Testing in the Cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 364–367. https://doi.org/10.1145/3092703.3098222

[23] F. Guan, J. Shi, X. Ma, W. Cui, and J. Wu. 2017. A Method of False Alarm Recognition Based on k-Nearest Neighbor. In *2017 International Conference on Dependable Systems and Their Applications (DSA)*. 8–12. https://doi.org/10.1109/DSA.2017.11

[24] F. Harrell, K. Lee, and D. Mark. 1996. Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors. *Statistics in medicine* 15 4 (1996), 361–87.

[25] Frank E Harrell, Kerry L Lee, Robert M Califf, David B Pryor, and Robert A Rosati. 1984. Regression modelling strategies for improved prognostic prediction. *Statistics in Medicine* 3, 2 (April 1984), 143–152.

[26] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, 39–48.

[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[28] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[29] H. Jiang, X. Li, Z. Yang, and J. Xuan. 2017. What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 712–723. https://doi.org/10.1109/ICSE.2017.71

[30] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen 0034, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM - A Highly Efficient Gradient Boosting Decision Tree. *NIPS* (2017).

[31] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570

[32] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 312–322. https://doi.org/10.1109/ICST.2019.00038

[33] Jeff Listfield. [n.d.]. Google Testing Blog Where do our flaky tests come from? https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html. Accessed: 2018-05-16.

[34] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2019. Explainable AI for Trees: From Local Explanations to Global Understanding. *arXiv preprint arXiv:1905.04610* (2019).

[35] Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. 2018. Consistent Individualized Feature Attribution for Tree Ensembles. *CoRR* abs/1802.03888 (2018). arXiv:1802.03888 http://arxiv.org/abs/1802.03888

[36] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[37] Scott M Lundberg, Bala Nair, Monica S Vavilala, Mayumi Horibe, Michael J Eisses, Trevor Adams, David E Liston, Daniel King-Wai Low, Shu-Fang Newman, Jerry Kim, et al. 2018. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature Biomedical Engineering* 2, 10 (2018), 749.

[38] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[39] John Micco. [n.d.]. Google Testing Blog Flaky Tests at Google and How We Mitigate Them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed: 2018-05-16.

[40] John Micco. 2017. The State of Continuous Integration Testing at Google.

[41] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2020. Flake It 'Till You Make It: Using Automated Repair to Induce and Fix Latent Test Flakiness. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 11–12. https://doi.org/10.1145/3387940.3392177

[42] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2825–2830. http://dl.acm.org/citation.cfm?id=1953048.2078195

[43] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 492–502. https://doi.org/10.1145/3379597.3387482

[44] Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 857–862.

https://doi.org/10.1145/3236024.3275529

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[46] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

[47] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. http://jonbell.net/publications/flakymutants

[48] 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

[49] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 545–555. https://doi.org/10.1145/3338906.3338925

[50] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. arXiv:1206.2944 [stat.ML]

[51] Pavan Sudarshan. [n.d.]. No more flaky tests on the Go team. thoughtworks.com/insights/blog/no-more-flaky-tests-go-team. Accessed: 2018-10-02.

[52] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. 534–538. https://doi.org/10.1109/ICSME.2018.00062

[53] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 385–396. https://doi.org/10.1145/2610384.2610404