SafeRevert: When Can Breaking Changes be Automatically Reverted?

Tim A. D. Henderson*, Avi Kondareddy†, Sushmita Azad‡, and Eric Nickell§
Google LLC
1600 Amphitheatre Pkwy
Mountain View, California, USA 94043
*tadh@google.com †avikr@google.com ‡sushazad@google.com §esnickell@google.com

***Abstract*—When bugs or defects are introduced into a large scale software repository, they reduce productivity. Programmers working on related areas of the code will encounter test failures, compile breakages, or other anomalous behavior. On encountering these issues, they will need to troubleshoot and determine that their changes were not the cause of the error and that another change is at fault. They must then find that change and *revert it* to return the repository to a healthy state. In the past, our group has identified ways to identify the root cause (or culprit) change that introduced a test failure even when the test is flaky. This paper focuses on a related issue: at what point does the Continuous Integration system have enough evidence to support *automatically reverting a change*? We will motivate the problem, provide several methods to address it, and empirically evaluate our solution on a large set (25,137) of real-world breaking changes that occurred at Google. SafeRevert improved recall (number of changes recommend for reversion) by $2\times$ over the baseline method while meeting our safety criterion.**

I. Introduction

Large scale software development is enabled by automatically executing tests in a continuous integration environment. Continuous integration (CI) [1] is the industrial practice of using automated systems to automatically integrate changes into the source of truth for the software system or repository. This improves collaboration by helping software developers avoid breaking compilation, tests, or structure of the system that others are relying on.

Corporations and teams may engage in CI using adhoc tools across many independent software repositories. For instance Github (github.com) supports CI "Actions" which can be integrated into any repository. However, many organizations are recognizing the value of using a "mono-repository" (monorepo) development model where many or all teams in the organization use a single shared software repository [2]. At the largest organizations such as Google [2], [3], [4], Microsoft [5], and Facebook [6] large repositories are complimented by advanced centralized CI systems.

In these large, modern CI systems, the integration goes beyond just ensuring the code textually merges. Compilations are invoked, tests are executed, and additional static and dynamic verification steps are performed. The demand for machine resources can exceed capacity for build, test, and verification tasks desired by a large-scale CI system. To combat this problem, test case selection or prioritization is used [7], [8], [6] to select fewer tests to run.

Additionally, CI steps are often invoked at multiple points in the Software Development Life Cycle. In the past, it may have been assumed that the tests were executed by the CI system once, at the time a commit was created in the version control. Today, CI may execute tests multiple times during development: when a change is sent for code review, immediately prior to submission or integration into the main development branch, after one or more changes has been integrated into the development branch, and when a new release is created. *This paper is primary concerned with testing that occurs after the code has been integrated into the main development branch.*

A. Why is Testing Necessary After Code Integration?

When deploying CI for the first time, many organizations primarily focus on conducting testing at the time a change is merged into the main branch. For instance, they may test when a Pull Request (PR) on Github is going to be merged into the main branch. The PR is merged if the tests pass and no changes have been made to the main branch since the testing started. This strategy works well until the rate of PR submission exceeds the average amount of time it takes to run all the tests.

At this point, organizations may do a stop gap fix such as adding more machines to run tests in parallel or reducing the size of the test suite. However, at some point, these measures will prove ineffective and the rate that testing can be conducted will become a impediment to an organization's engineering velocity. To address this, one common solution is to introduce a "submission queue" which batches changes together for testing and merges them all if the tests pass [9]. If the tests fail, the offending change must be identified and the remaining changes in the batch must be retested (at least doubling the total testing time) [10].

As the submission rate continues to increase, the organization may add conservative test selection based on file-level dependence analysis [9]. But at this point, the organization will be reaching the limit of what can be done to completely prevent any breaking changes from being integrated into the main development branch beyond just buying more and more machines to further parallelize the testing.

And what about just buying more machines? Won't this effectively solve the problem? It would until the submission rate exceeds the time it takes to run the slowest test. At this

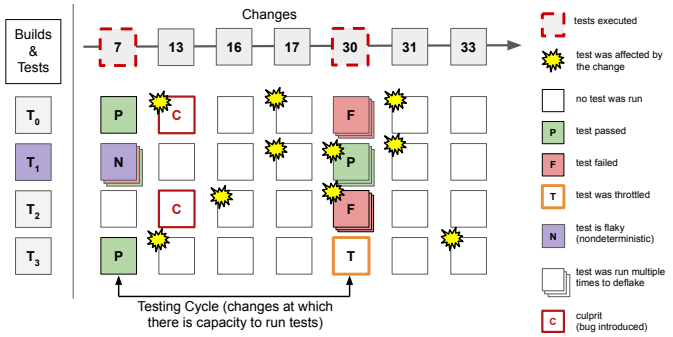

Fig. 1: Example for showing when tests are run by a continuous integration system after the code has been integrated into the main development branch. Tests are only run periodically when there is capacity at changes (versions) marked with red dashes. These are called "Testing Cycles" in this paper. Change 13 introduced a bug that broke tests T_0 and T_2. Change 13 is marked with a red "c" indicating "culprit." Not all tests are run during every testing cycle due to dynamic or static test selection [7]. Note, this is an improved version of Figure 1 from [4].

point, the number of machines purchased can only partially control impact to developer productivity. Even with a very large budget for testing, the number of changes per batch will continue to grow as increases in the number of machines will not reduce the batch size but only keep the testing time per batch relatively constant. The larger the batch size, the more likely that a developer will experience conflicts with another change when attempting to submit. These conflicts (either at the syntactical or semantic level) will make it difficult for developers to reliably submit their changes and require them to constantly monitor the submission process. This monitoring will hurt overall developer productivity.

At the highest submission rates seen in industry today, using submission queues that guarantee zero breaking changes becomes infeasible both from a machine cost perspective and from a developer productivity perspective. Therefore, many organizations relax the requirement that an integrated change will never break a test. Even while using submit queues to gate contributions for a single team, Google has relied on postsubmit testing for mission-critical software since the early 2000s as documented by Mike Bland [11].

B. Post Submission Testing

The purpose of testing after a change has been integrated (called *Post-Submission* Testing or *Postsubmit* Testing hereafter) is to identify defects that slipped through testing that occurred prior to integration. Typically, organizations are unable to run all tests at every integrated change. Instead, testing is conducted periodically. In this paper we will refer to this periodic testing as *Testing Cycles*.

Figure 1 is a simplified view of the postsubmit testing strategy used at Google. The test scheduler waits until our Build System [12] has capacity to start a test cycle. It then schedules tests and then waits until the system has capacity again. When scheduling tests, certain tests may be temporarily skipped or throttled to conserve resources. As shown in Figure 1, this leads to the system usually detecting new failures some time after the version that introduced them had been integrated. This leads to the problem of "Culprit Finding."

C. Culprit Finding

Culprit finding is conceptually simple: given a list of versions that may have introduced a fault, locate the offending change. One may solve this problem with a variety of techniques: Zeller's Delta Debugging [13], Git's bisection algorithm [14], or Google's Flake Aware Culprit Finding (FACF) algorithm [4]. Every technique used to identify the offending change will have some error rate in industrial practice.

But wait! How can a binary search or delta debugging have an "error rate"? The answer is that old nemesis of industrial practice: flaky or non-deterministic test behavior [15]. Flaky tests can be caused both by problems in the production code (ex: race conditions causing rare errors), in the test code (ex: use of "sleep") and by infrastructure problems (ex: unreliable machine with bad ram or network congestion). All of these problems behave in *version non-hermetic* ways, where failures may not be strictly linked to the version of code executed. Furthermore, even when the test is fully *version hermetic* the flakiness may not obey a Bernoulli distribution as subsequent executions may not be fully independent of the prior ones.

The above issues mean that even culprit finding algorithms such as FACF [4] that have been purpose-built to mitigate flakiness will have some error rate. While that error rate will be much less than a naive algorithm, it may still be high enough to cause problems when deployed in certain environments.

D. Automatically Reverting Changes

Once an organization has a reliable and high performance culprit finding system, it is natural to use it to automatically revert (undo, roll back) changes that introduce defects into the main development branch. By automatically undoing these changes, the system decreases the amount of time developers working with impacted tests will experience breakages that are unrelated to the changes they are working on. It will also increase the number of versions that are viable to be used to make software releases. Reducing development friction and increasing the number of release candidates improves developer productivity by reducing the amount of time developers spend troubleshooting tests that were broken by someone else.

Unfortunately, naively integrating FACF directly into a system that immediately reverts all changes it identifies will lead to unhappy developers. This is because even though Google's FACF has a measured *per-test accuracy* of 99.54%, when aggregated (grouped) by blamed change, the accuracy *per-change* is only 77.37% in the last 28 day window as of this writing.[1] This would translate into *incorrectly reverting* approximately 20-130 changes per day out of an approximately

[1]Both the *per-test* error rate and the *per-change* error rates quoted above are drawn from the same 28 day window. The Postsubmit result which triggers culprit finding may itself have been a flake, so there were spurious suspect ranges in the same period. These rates may differ slightly from the numbers reported in [4] as they reflect our most recent data as of 2023-11-12.

of 300-500 total changes per day reverted (see Figure 2). At Google, we find the cost of incorrectly reverting a change is extremely high in terms of developer toil and frustration. Therefore, we need to reduce the rate of bad reversions as much as possible. In this work, we aim to incorrectly revert fewer than one change per day on average.

E. Our Contributions and Findings

In this paper we propose a shallow machine-learning based method for using the output of a culprit finding algorithm to automatically revert (undo, rollback) a change that controls for potential errors from the culprit finding algorithm. Our method is generic and can be used with any culprit finding algorithm.

1) SafeRevert: a method for using the output of any culprit finding algorithm to automatically revert (undo) a change. SafeRevert controls for the error rate while increasing total number of reversions over baseline methods.
2) An ablation study on the models, designs and features used by the machine learning system used in SafeRevert to identify the most impactful features and best models.
3) A large scale study on SafeRevert's efficacy using ∼25,137 potential culprit changes identified by Google's production culprit finder over a ∼3 month window yields a recall of 55.7%, $\sim$ 2.1$\times$ higher than the baseline method.

II. Background

The Test Automation Platform (TAP) is a proprietary Continuous Integration (CI) system at Google. It runs tests that execute on a single machine without the use of the network or external resources.[2] All tests on TAP are required to run in under 15 minutes and exceeding that limit is considered a test failure. TAP executes tests both before a user submits their changes and after the user has submitted their change. This paper is only concerned with the testing that has occurred after a user has submitted a change. At Google, this is referred to as "Postsubmit Testing". The part of TAP that does Postsubmit Testing is called TAP Postsubmit.

TAP has appeared in the literature several times [7], [16], [8], [3], [17], [12], [4] and there has been gradual evolution its testing strategy over the years. However, in general TAP uses a combination of static and dynamic Test Selection, execution throttling, and just-in-time scheduling to control testing load. A simplified diagram of TAP Postsubmit testing is visualized in Figure 1. Tests are periodically run in Testing Cycles.[3] During a cycle, Projects that are eligible to run their test in the cycle are selected. Tests from those projects are included if some file was modified since that test's last execution can influence their behavior via inspection of a dependence graph at the Build Target and File level granularity [7], [16]. When breakages inevitably occur culprit finding is conducted using the FACF algorithm to locate the offending change [4].

FACF operates on a single test target at a time. For simplicity, we use the term "test" both for a target which executes test code or for a "build targets" which verifies that a binary can compile. Conceptually, FACF performs a "Noisy Binary Search" [18] (also called a Rényi-Ulam game [19]) which FACF models under the Bayesian Search Framework [20]. The input to FACF includes the suspect changes ("suspects") that may have broken the test and an estimate collected by an independent system of how likely it is to fail non-deterministically without the presence of a deterministic bug, it's "flakiness". Much like a normal binary search, it then divides the search space, executes tests, and updates a probability distribution based on the outcomes. Eventually, the system will determine that one of the suspects is above a set probability threshold (.9999) and is the source of the test failure, or that none of the suspects is at fault and the original failure was spurious, due to a flake (non-deterministic failure).

A. Culprit Finding Accuracy

Now some caveats: The math behind FACF [4] assumes that individual test executions of the same test at either the same version or different versions are independent statistical events. That is, a prior or concurrent execution of a test cannot influence a subsequent execution. Unfortunately, while this assumption is theoretically sound, in practice this property does not always hold. External factors outside a program's code can also influence a test's execution behavior. For example, the time of day, day of year, load on the machine, etc., can all influence the outcome of certain tests. Thus, while we have configured FACF to have an accuracy of 99.99%, in practice we do not observe this level of accuracy. As noted in the Introduction, our observed accuracy was 99.54% in the last 28 day window as of this writing. This level of accuracy is consistent with the empirical study conducted in the 2023 paper [4].

How was accuracy "observed"? What was the ground truth used? By what measurement technique? At Google, we continuously randomly sample a subset of culprit finding execution results as they are completed. The selected sample is then cross checked by performing extensive reruns that demand a higher level of accuracy and assume a worse flakiness rate for the test than used for culprit finding. Additionally, more reruns are scheduled for a later time to control for time-based effects. Finally, execution ordering effects are controlled for by ensuring that executions at versions that should fail can fail both before and after versions that should pass are executed. Full details on our methods for verification can be found in the FACF paper [4]. Even with extensive reruns there remains a small probability of error. While our method of verification is imperfect, it allows for continuously computing a statistical estimate on the accuracy of the culprit finder.

Measurements of FACF accuracy reported in the 2023 paper were performed per test breakage. That is, culprit finding is performed on a test t when it has an observed failure at some

[2]There are some legacy tests which are allowed to use the network but there is an on-going effort to clean up their usage.

[3]Previously, we referred to these cycles as "Milestones" in most of the previous literature but we are gradually changing our internal nomenclature as we evolve TAP Postsubmit's testing model.

version β when it was previously passing at a prior version α. The range $(\alpha, \beta]$ is referred to as the *search-range* and the key $\{t \times (\alpha, \beta]\}$ is the *search-key*. For a given search-key, the culprit finder first filters the search-range to changes on which t has a transitive build dependence, which we call the *suspect-set*. A culprit κ is identified somewhere between β and α (e.g. $\alpha \sqsubset \kappa \sqsubseteq \beta$ where $a \sqsubset b$ indicates version a is before version b). It is possible that there are multiple κ_t for different t w.r.t. the same search range, which we can call the *culprit-set* of the search range. What is measured is whether or not κ is correct (true or false) for a given t and search range $\alpha \sqsubset \beta$. Accuracy is the total correct measurements over the total number of measurements taken.

B. Applying Culprit Finding Results to Change Reversion

When breaking changes get merged into the main development branch they impede development productivity. At Google we term impediments to productivity as "friction." Breaking changes impact three types of friction tracked at Google: Release Friction, Presubmit Friction and Development Friction. Release Friction occurs when breaking changes that merge into the main development branch impede automated releases and require manual intervention by the primary team's on-duty engineer. Presubmit Friction occurs when pre-submission ("presubmit") testing in TAP fails due to a broken test in the main development branch. Development Friction occurs when a developer manually triggers the execution of a test broken on the main branch during active code development. All three types of friction can steal time from developers and reduce productivity.

To reduce friction, changes that break tests can be automatically reverted ("rolled back"). However, auto-revert ("auto-rollback") only improves developer productivity if the changes reverted actually broke tests. Let's consider for a moment what happens when a change is incorrectly reverted. The developer who authored the change now needs to go on a wholly different troubleshooting journey to determine why their change didn't stay submitted, and they are faced with the same damning-but-incorrect evidence used to revert their change. The developer is frustrated and must now debug tests that their team doesn't own and which their changes should not affect.

Some changes are particularly difficult to submit with a buggy auto-revert system: those that change core libraries. These changes have the potential to break a large number of tests and impact a large number of other developers. But, at the same time they are also more likely to be incorrectly blamed (as TAP tracks what tests are affected by each change and uses this as an input to culprit finding). Past auto-revert systems that have been too inaccurate have caused core library authors to opt their changes out of auto reversion as the productivity cost to the core library teams has been too great to bear.

C. Accuracy of Auto-Revert versus Culprit Finding

A change κ that breaks a test may have broken more than one. If so, then multiple independent executions of the culprit finding algorithm (one for each broken test) will blame the version κ as the culprit. As explored above, the algorithms have an error rate. A change κ that breaks n tests will have $m \leq n$ culprit finding conclusions that correctly identify κ as the culprit.

Our concern with auto-revert isn't the accuracy of culprit finding per search-key $\{t \times (\alpha, \beta]\}$ but rather per culprit change κ. Let K be the set of all changes in the repository blamed as culprits by the culprit finding system. Let $c \subseteq K$ be the subset of the culprits that are correct and $\overline{c} \subseteq K$ be the subset of culprits that are incorrect. Then if every culprit in K was automatically reverted, the accuracy of the auto-revert system would be $|c|/|K|$. In the introduction, we noted that the accuracy *per change* was 77.37% which equals $|c|/|K|$.

Why is the auto-revert accuracy (77.37%) lower than the culprit finding accuracy (99.54%)? Because there are many more tests than culprits and because correctly identified culprit changes may have broken more than one test. The chance of finding that culprit is higher because there are more chances to find it. Additionally, all of the culprit finding conclusions that identify one of those changes that break many tests will be correct. The set of incorrectly identified culprit $\overline{c}$ tends to contain mostly changes that are blamed by a fewer number of tests than those in set c.

D. Problem Statement

Identify a method for selecting as many as possible changes from the set of culprits K that can be safely reverted. A change is safe to revert if it is a true culprit. Safety can be defined in terms of probability as well: A change is safe to revert if the chance it is a true culprit is $> 99\%$. The number of changes selected should be maximized (while maintaining safety) to ensure the methods performs reversions rather than safely doing nothing at all.

III. Automatically Reverting Changes

We will describe several methods for selecting changes to revert while maintaining the intended safety property. The first method is a simple "baseline" method that the more complex methods will be evaluated against. The other methods use shallow machine learning systems to take advantage of metadata available at revert time.

Our baseline method is based on a single observation: a change κ_0 that is identified as the culprit by culprit finding on many tests is more likely to be a culprit than a change κ_1 that is only blamed by a few tests. Therefore, the simplest heuristic approach to selecting changes to revert is to threshold on a minimum number of tests identifying the change. BASELINE(N) will refer to this method with selecting changes with at least N blaming targets. Figure 2 shows the performance of this method at various minimum number of targets. The most conservative, BASELINE(50) can avoid most false positives while reverting only a $\sim$13% of true culprits. We choose BASELINE(10) for our final evaluation as this is the heuristic Google has historically used. Additionally, 10 is the lowest configuration of BASELINE that generally (although not always) meets the desired safety criteria.

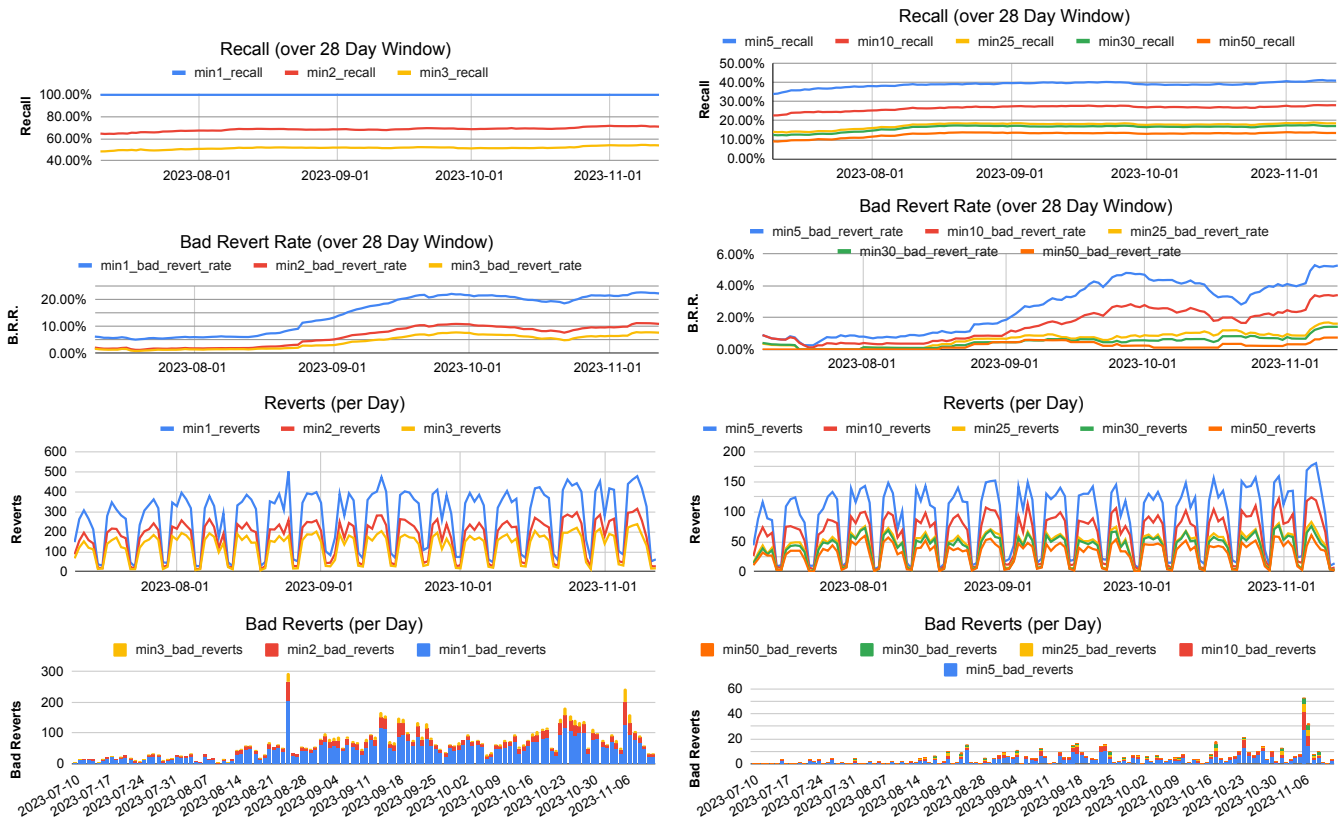

Fig. 2: BASELINE method performance. As the minimum required number of tests increases from 1 to 50 or more the bad revert rate decreases and the recall decreases. Decreasing the bad revert rate is good (this means the accuracy is improving) but decreasing the recall is bad (this means fewer changes are being reverted). Thus the BASELINE method has to be configured to be "just conservative enough" to be safe while still reverting a reasonable number of changes. At Google we currently utilize the BASELINE(10) configuration in production. **Note the change in Y-Axis between the left and right columns** in order to show details of the more conservative configurations.

A. Predicting Auto-Revert Correctness using Machine Learning

Instead of choosing just one suggestive feature – the number of blaming tests – to decide on a reversion, our proposed method uses multiple features and shallow machine learning models to improve performance relative to the baseline method. A selection of easily obtainable coarse-grained metadata on the changes, tests, and culprit finding process are used as features to the models. As these features are mainly simple numerical, categorical, and textual features about the code, we use simple model architectures to control for overfitting. Architectures examined are: Random Forests (RF) and AdaBoost (ADA) (both provided by scikit-learn[4] [21]), and XGBoost (XGB)[5] [22]. These tree-based model architectures are easy to use, have low computational costs, and are robust to the feature representation versus alternatives that require more preprocessing.

B. Features Used

A suspect change is one that has been identified by at least one culprit-finder in the process of culprit finding a test. We can consider this the test *blaming* the change. For any suspect, we have one or more blaming tests. Then, we have a few sources of information that may be valuable for predicting a true breakage: characteristics of the change itself and characteristics of the blaming tests or culprit finding results. Table I contains the grouping of the features against their logical feature category, arrived at using the Pearson correlation between numerical features and lift analysis for non-numeric features.

C. Feature Representation

1) Categorical Features: Given a set of categories, we can create a fixed length representation by encoding a choice of category as a one-hot vector. When we have a variable number of categories per instance, such as language per test, this trivially becomes a multi-hot representation by summing the vectors.

2) Numerical Features: Singular numerical fields are integrated without preprocessing or normalization, as we expect decision trees to be robust to value-scaling. Variable-length numerical fields are turned into variable-length categorical fields by generating bins representing quantiles of the training set distribution of that value. Then, bin-mapped values can be condensed into a fixed-length multi-hot encoding as above.

3) Token Set Features: Token Sets correspond to variable-length categorical features where the list of categories is

[4]https://scikit-learn.org/

[5]https://xgboost.readthedocs.io/en/stable/index.html

not known in advance. A good example of such a feature is compiler flags which should be dealt with adaptively by the model. We handle this issue by building a vocabulary per token-set feature on the training set parameterized by a minimum and maximum across-instance frequency. The change description, while potentially better dealt with through doc2vec or a more sophisticated NLP embedding approach, is treated like a token set and the resulting multi-hot encoding, having each field normalized by document frequency (TF-IDF).

D. Feature Grouping

1) Logical Sub-Grouping for Feature Lift Analysis: In order to avoid crowding out the evaluation and analysis with too many combinations of features, we've grouped them in Table I into logical groups of features based off of conceptual categories and to reflect cross-feature correlations for numerical features, computed via pearsons correlation coefficient. As our "Token Set" features represent a distinct class of features both in terms of encoding method and granularity (highly specific to individual changes), we separate them from the coarse-grain change features. These logical groups will allow us to clearly convey the individual signal being provided by each class of feature during our feature analysis.

2) Availability Sub-Grouping for Model Comparison Analysis: Separately, we define the feature sets F = {*BASE*, *AHEAD-OF-TIME*, *REAL-TIME*, *ALL*} to distinguish between features available at any potential inference time

1) *BASE* is just the number of blaming tests. This corresponds to the BASELINE heuristic method.
2) *AHEAD-OF-TIME* features are properties of the change under suspicion, and are therefore available immediately. Features under CHANGE_CONTENT, CHANGE_METADATA, and CHANGE_TOKENS from Table I belong here.

TABLE I: FEATURES FOR MACHINE LEARNING MODEL.

Category	Features
BASE	# of Blaming Tests
CHANGE_CONTENT	LOC (changed, added, deleted, total)
	# of Files Changed
	File Extensions
CHANGE_METADATA	Reviewer/Approver Count
	Issue/Bug Count
	Robot Author
CHANGE_TOKENS	Directory Tokens
	Description
CULPRIT_FINDER	|Suspect-Set| of Search-Key
	|Culprit-Set| of Search-Range
FLAKINESS	Historical Flaky Identification Count (Build/Test)
	Historical Execution Flake Rate
TEST_ATTRIBUTES	Type of Test (Test vs Build)
	Bazel Rule (Ex: "java_test")
	Machine Types (Ex: gpu)
	Test Language (Ex: java, c++, etc)
	Compiler Flags (ex: sanitizers)
	Average Machine Cost

3) *REAL-TIME* features are available only as a result of culprit finding processes at play continuously during the decision window, and can become available at different times based on different sources of generation, often after a change has already been suspected by a culprit finder – all other features from Table I fall in this set.
4) *ALL* features is the union of *AHEAD-OF-TIME* and *REAL-TIME*.

This distinction is important: the usefulness of automatically reverting changes is critically dependent on its latency from the point of discovering that a breakage exists. The longer we take to automatically revert the change, the more friction experienced by developers, and the more likely a developer would have had to manually intervene.

3) Feature Minimization: Given the demonstrated feature lift, we're also interested in minimizing the features needed to achieve similar performance levels without over-fitting on redundant features. We'll elaborate on the exact minimized feature sets in the evaluation section for feature analysis, where we use representative features from each logical subgroups that provide significant lift individually. Thus we have 3 further feature sets, MIN(f) for $f \in F$. Representative features are determined over the test set based on individual feature comparison, left out for brevity in the analysis, rather than the logical category evaluation presented here. We evaluate RF(f), ADA(f), XGB(f), RF(MIN(f)), ADA(MIN(f)), and XGB(MIN(f)) for each feature set $f \in F$.

E. Thresholding to use Model Prediction Score for Selection

Models produce an output probability score from 0 to 1. To discretize these scores into actual change selections. We dynamically pick a threshold using the TEST dataset to select the threshold that minimizes the bad revert rate while maximizing a positive, non-zero recall.

F. Hyperparams for ML Models

Using scikit-learn, we define RF as the RandomForestClassifier parameterized with a depth of 16 and ADA as the AdaBoostClassifier with default parameters, each of which provide discretized predictions with the above threshold selection procedure. XGBoost is configured with: objective='binary:logistic', eta=0.05, and max_delta_step = 1.

IV. EMPIRICAL EVALUATION

We empirically evaluated SafeRevert (the ML based method) against the BASELINE heuristic method. We evaluate: the overall performance of the different ML models considered, the importance features used, and compare the BASELINE method to a selected ML model.

A. Research Questions

RQ1: What is the safety and performance of studied methods in the context of the developer workflow?
RQ2: What is the marginal benefit provided by each feature?
RQ3: Did the chosen method significantly improve performance over the baseline method while maintaining required safety levels?

B. Measuring Safety

As mentioned in section I-D, an incorrectly reverted change causes unacceptably high developer toil. Safety is the likelihood that a method will produce a false positive result by incorrectly categorizing a change as being a culprit and reverting it. This is a referred to as a *Bad Revert* while correctly reverting a change that introduced a bug is a *Good Revert*. The total number of reverts is *Bad Reverts* + *Good Reverts*. A change that should have been reverted but wasn't is a *Missed Good Revert* and a change that was correctly not reverted is a *Avoided Bad Revert*.

In terms of classic terminology for evaluating binary classifiers:

1) *Good Revert* = True Positive (TP)
2) *Bad Revert* = False Positive (FP)
3) *Avoided Bad Revert* = True Negative (TN)
4) *Missed Good Revert* = False Negative (FN)

The safety properties we are most interested are (a) the total number of bad reverts, (b) total number of bad reverts per day, and (c) the bad revert rate. The bad revert rate (BRR) is the percentage of bad reverts out of the total number of reverts $= {}^{FP}/_{TP+FP}$. This is otherwise known as the False Discovery Rate (FDR). Note, that $FDR = 1 - Precision = 1 - {}^{TP}/_{TP+FP}$. Thus, safety can either be stated in terms of precision (ex. precision must be above 99%) or in terms of bad revert rate (ex. bad revert rate must be below 1%).

In this evaluation we will consider a method safe if its bad revert rate is below 1%, there are fewer than 14 bad reverts in the final validation set, and there are no more than 5 bad reverts per day.

Why these numbers? Our end goal is a production SafeRevert. Our small team supports a large number of services and users of TAP. We need to minimize toil for the 2 engineers per week who are "oncall" for them. The numbers above were selected to be manageable for us in terms of the overhead required for communicating with our users and investigating the root cause of bad reverts. While these numbers are subjective and dependent on our context, they are meaningful to our team. We expect other teams supporting central CI systems would make similar choices.

C. Measuring Performance

If a method is deemed safe (meets above criteria) then it is an eligible method to be used to pick changes to revert. To determine whether one safe reversion method is better than another we look at the how many *Good Reverts* a method is able to achieve out of the total possible good reverts. This corresponds to the metric known as recall $= {}^{TP}/_{TP+FN}$. The higher a safe method's recall the better it performs. As with safety, we prefer methods that are consistent and have low variability in their reversion recall per day. We then define performance as the number of breaking changes successfully reverted both in total and per-day.

D. Evaluation Dataset

Our evaluation dataset consists of roughly 3.5 months of data split between a training, test, and validation set and contains ∼25,137 unique changes identified by the production culprit finder as culprits. Each row has a boolean indicating whether followup verification confirmed the change was indeed a culprit. This verification continues to be done in the manner described in our 2023 paper [4].

For training and evaluation we produce a time-based split: TRAIN consists of the first 2.5 months, TEST the next ∼2 weeks, and VALIDATION the final ∼2 weeks. Time based splits avoid cross-contaminating our training with diffuse information about types of breakages that may be based on time-dependent attributes of the codebase. It is important that the training data is uncontaminated with any data from the time period where the evaluation occurs. If it is, the evaluation will not reflect the performance observed in production.

All comparative model and feature evaluation was performed against the TEST set in order to determine our optimal ML model configuration, OPTIMAL. We then evaluate OPTIMAL and BASELINE against the VALIDATION set.

V. Results

RQ1*: What is the safety and performance of studied methods in the context of the developer workflow?*

> ***Summary***: The XGB(ALL) was best overall, with a recall of 61.2%, and a bad revert rate of .3%.

> ***Note***: This experiment was conducted on the TEST data set as the VALIDATION set was reserved for RQ3.

Table III summarizes the critical metrics for safety (bad revert rate) and performance (recall) for the three model types considered: RandomForest [RF], AdaBoost [ADA], and XGBoost [XGB]. Figure 3 shows the Receiver Operator Curves (ROC) for all 3 model types and their associated area under the curve (AUC) values. The ROC curve better summarizes model performance over a wider range of target objectives than the table which is focused on the safest configuration (minimizing Bad Revert Rate). For instance if a different application had a higher tolerance for bad reversion/false positives the ROC curves show that you could achieve a 90% true positive rate with a 20% false positive rate.

Safety: The XGB model was the safest overall; XGB(REAL-TIME) with 4 bad reverts was marginally safer than XGB(ALL) with 5 bad reverts. In general, XGB and ADA both outperformed the RF in terms of safety, while RF was only safe for the REAL-TIME dataset. XGB and ADA were safe (under our criteria, see Section IV-B) using all the features or the REAL-TIME subset.

Performance: The highest performing configuration was XGB(REAL-TIME) and ADA(ALL) which had 1981 Good Reverts and a recall of 63.0%. XGB(ALL) was a very close second, with a recall of 61.2%. ADA(ALL) however had a higher bad revert rate than any of the XGB models.

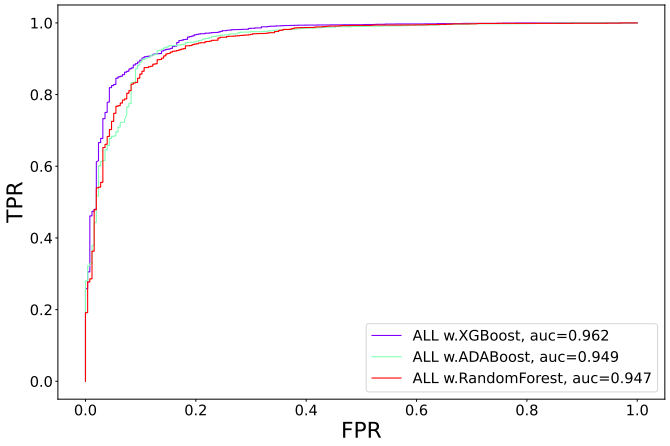

Fig. 3: Comparing the performance of the AdaBoost, XGBoost and RandomForest models against the entire feature set *F[ALL]*

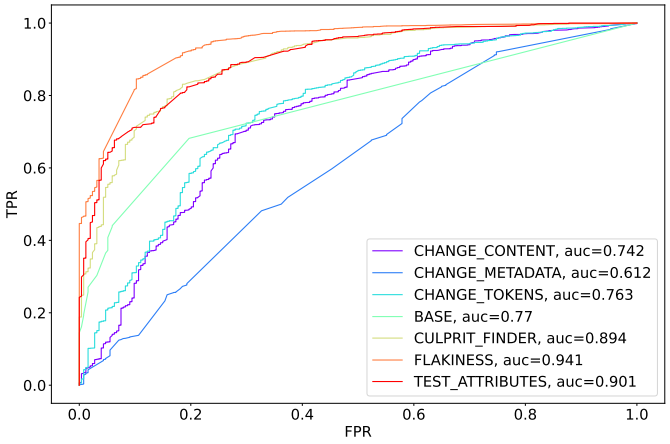

Fig. 4: Signal provided by each feature group alone on the XGB model. AUC is the area under the Reciever Operating Characteristic (ROC) curve.

We chose XGB(ALL) as our OPTIMAL model for this paper. While XGB's safety and performance is comparable between ALL and REAL-TIME features, observe in Table IV MIN(ALL) outperforming MIN(REAL-TIME) in terms of recall. This indicates if some features in the REAL-TIME set cannot be collected at decision time the model will be more robust with the inclusion of the AHEAD-OF-TIME features.

***RQ2**: What is the marginal benefit provided by each feature group*?

> ***Summary***: Historical flakiness data was the most valuable feature set, boosting model recall by 17%, while change metadata was the least important.

> ***Note***: This experiment was conducted on the TEST data set as the VALIDATION set was reserved for RQ3.

To determine marginal benefit we perform an ablation study to look at the marginal benefit provided by each feature (as grouped in Table I). The ablation study was performed using the XGBoost model which was chosen over AdaBoost due to its higher recall as seen in Table III. Three marginal benefit experiments were performed:

1) Measure the performance of the model trained against **only** the features in a single feature set. We visualize the results in Figure 4 as an ROC curve. Table II contains the critical safety and performance metrics of Bad Reverts (safety) and Recall (performance) as well as the AUC which summarizes overall model predictive performance – generally a model with a higher AUC will be more performant than one with a lower AUC.
2) Measure the **Positive(+)** ***Lift*** provided by adding a single feature group to our BASE feature (the number of blaming tests). For instance, we would add the features in CHANGE_CONTENT to create a XGB(CHANGE_CONTENT + BASE) model. The **+Lift** for Recall is computed as Recall[XGB(CHANGE_CONTENT + BASE)] - Recall[XGB(BASE)] and similar for AUC of the ROC curve. The results of this analysis are shown in Table II.
3) Measure the **Negative(–)** ***Lift*** provided by removing a single feature group from out ALL feature set. For instance we would remove the features in CHANGE_CONTENT to create a XGB(ALL - CHANGE_CONTENT) model. The **–Lift** for Recall is computed as Recall[XGB(ALL)] - Recall[XGB(ALL - CHANGE_CONTENT)] and similar for AUC of the ROC curve. The results of this analysis are shown in Table II.

As discussed in section III-D3, we attempted to minimize the feature subsets while providing comparable performance to the full feature sets. Table IV contains a breakdown of the performance for these minimal subsets.

***RQ3**:Did the chosen method improve performance over the baseline method while maintaining required safety levels?*

> ***Summary***: XGB(ALL) improved overall recall while meeting our safety requirements with a Bad Revert Rate under 1% and an average of 0.6 Bad Reverts per Day. BASELINE(10) had a recall of 26.3% while XGB(ALL) had a recall of 55.7% – an improvement of $\sim 2.1\times$.

> ***Note***: We used the reserved VALIDATION data set for this research question.

Table V shows per day metrics comparing XGB(ALL) against the heuristic BASELINE(10) method. As a reminder the BASELINE method thresholds the number of blaming tests to a fixed number. BASELINE(10)'s average bad reverts per day is 0.86 and its total Bad Revert Rate (BRR) is 1.5%. XGB(ALL) achieves the overall safety limit with a BRR of 0.5% and has an average bad reverts per day of 0.6 which meets our safety threshold.

XGB(ALL)'s bad revert rate outperforms BASELINE(10), with a recall 2.1 times higher than BASELINE(10). XGB(ALL) made 1,823 total good reverts in the validation

TABLE II: FEATURE ABLATION STUDY using XGBoost to compute the relative impact different types of features have on model performance. Feature importance is measured for each feature set **alone** - using the metrics: # bad reverts, # good reverts, recall and AUC of the ROC curve. The positive(+) and negative(-) lift is computed against Recall and AUC. Positive lift is the improvement of the metric when the feature set is added to the BASE model. Negative lift is the degradation of the metric when the feature set is removed from the model with ALL features.

Features	**Bad Reverts**	**Good Reverts**	**Recall (alone)**	**+Lift(Recall)**	**–Lift(Recall)**	**AUC (alone)**	**+Lift(AUC)**	**–Lift(AUC)**
BASE	*1*	*507*	*0.161*	*0.000*	*0.031*	*0.770*	*0.000*	*0.001*
CHANGE_CONTENT	10	236	0.075	0.018	0.011	0.742	0.078	0.001
CHANGE_METADATA	2	11	0.003	0.081	0.034	0.612	0.016	0.001
CHANGE_TOKENS	4	205	0.065	0.208	0.014	0.763	0.085	-0.001
CULPRIT_FINDER	8	1307	0.416	0.168	-0.056	0.894	0.126	0.002
FLAKINESS	0	533	0.170	0.114	0.218	0.941	0.172	0.039
TEST_ATTRIBUTES	1	891	0.283	0.235	-0.015	0.901	0.133	0.011
ALL	5	1925	0.612	0.000	0.000	0.962	0.000	0.000

TABLE III: COMPARISON OF MODEL TYPES ON THE TEST SET. The safety and performance of the 3 model types (RF, ADA, XGB) were compared against each other using 3 different feature sets: AHEAD-OF-TIME, REAL-TIME, and ALL. ADA performed best for the AHEAD-OF-TIME features while XGB performed best for the REAL-TIME and ALL feature sets. For the AHEAD-OF-TIME feature set, although ADA performed the best none of the studied methods met our safety criterion (BRR < 0.01) on this data set.

	Good Reverts	**Bad Reverts**	**Bad Revert Rate**	**Recall**
RF(AHEAD-OF-TIME)	927.000	23.000	0.024	0.295
ADA(AHEAD-OF-TIME)	*605.000*	*12.000*	*0.019*	*0.192*
XGB(AHEAD-OF-TIME)	442.000	9.000	0.020	0.141
RF(REAL-TIME)	2173.000	11.000	0.005	0.691
ADA(REAL-TIME)	1838.000	7.000	0.004	0.585
XGB(REAL-TIME)	**1981.000**	**4.000**	**0.002**	**0.630**
RF(ALL)	2554.000	21.000	0.008	0.812
ADA(ALL)	1982.000	9.000	0.005	0.630
XGB(ALL)	**1925.000**	**5.000**	**0.003**	**0.612**

TABLE IV: FEATURE MINIMIZATION STUDY: Fewest individual features needed for near optimal performance, per availability category. BRR stands for *Bad Revert Rate*.

Features	**Bad Reverts**	**Good Reverts**	**BRR**	**Recall**
ALL	5.0	1925.0	0.003	0.61
MIN(ALL)	12.0	1912.0	0.006	0.61
REAL-TIME	4.0	1981.0	0.002	0.63
MIN(REAL-TIME)	0.0	959.0	0.000	0.31
AHEAD-OF-TIME	9.0	442.0	0.020	0.14
MIN(AHEAD-OF-TIME)	5.0	273.0	0.018	0.09

set while BASELINE(10) performed 860 good reverts. Based on the data in Table V XGB(ALL) is both safer and more performant than BASELINE(10) on the validation data set.

A. Discussion

The proposed method, SafeRevert, is generic and can be used with any culprit finding algorithm. By grouping individual features from different sources of data into logical feature sets (Table I), performing a detailed feature ablation study (Fig 4), and running the model on a minimal set of features (Table IV), we hope to provide a template via which teams in other contexts build on when adopting the approach we outline. In particular, while some features used may be Google specific, our feature ablation study can be replicated on different features in other software development organizations.

TABLE V: PER-DAY PERFORMANCE AND SAFETY OF BASELINE(10) VS XGB(ALL) ON VALIDATION SET. BR stands for *Bad Reverts*. GR stands for *Good Reverts*. BRR stands for *Bad Revert Rate*. XGB(ALL) is safe: with average bad reverts per day of 0.6 (< 2) and bad recall rate of 0.005 (< 0.01). XGB(ALL) has recall rate that is $\sim 2.1\times$ higher than BASELINE(10). Overall XGB(ALL) performs an average of 121 good reverts per day while BASELINE(10) performs 57, a substantial improvement.

	XGB(ALL)				**BASELINE(10)**			
Date	**BR**	**GR**	**BRR**	**Recall**	**BR**	**GR**	**BRR**	**Recall**
2023-10-07	0	15	0.000	0.333	0	11	0.000	0.244
2023-10-08	0	11	0.000	0.440	0	6	0.000	0.240
2023-10-09	1	146	0.007	0.535	2	72	0.027	0.264
2023-10-10	0	183	0.000	0.575	1	83	0.012	0.261
2023-10-11	0	204	0.000	0.581	2	99	0.020	0.282
2023-10-12	0	160	0.000	0.552	0	80	0.000	0.276
2023-10-13	0	172	0.000	0.585	0	75	0.000	0.255
2023-10-14	0	19	0.000	0.352	1	7	0.125	0.130
2023-10-15	0	16	0.000	0.552	0	5	0.000	0.172
2023-10-16	0	153	0.000	0.591	0	82	0.000	0.317
2023-10-17	0	201	0.000	0.581	0	102	0.000	0.295
2023-10-18	3	194	0.015	0.553	4	80	0.048	0.228
2023-10-19	4	171	0.023	0.564	1	89	0.011	0.294
2023-10-20	0	155	0.000	0.546	0	62	0.000	0.218
2023-10-21	1	23	0.042	0.479	2	7	0.222	0.146
Total	**9**	**1823**	**0.005**	**0.557**	**13**	**860**	**0.015**	**0.263**
Average	**0.6**	**121.5**	**–**	**–**	**0.86**	**57.3**	**–**	**–**

While we don't expect a team implementing SafeRevert to achieve the exact Recall and Bad Revert Rates we report we do expect this method to out perform the BASELINE method once an appropriate set of features is identified.

B. Threats to Validity

Our dataset may misrepresent information available at inference time in the forthcoming service as it may include information not available to us at our decision time. This is due to using offline data in the dataset as the production system based on this paper is currently under construction. We adjust this threat by evaluating performance restricted to features available independent of any culprit finding event and restricting our real-time data to a time bound relative to change submission time.

The data presented in this final manuscript differs slightly than the reviewed manuscript. At the time of review approximated 15% of the dataset was lacking verification results (collected using the method described by Henderson [4]). This was disclosed in this section to the reviewers and we made two conservative assumptions: 1) any culprit change lacking verification results was considered a false positive and 2) we assumed the BASELINE method was correct for those changes (inflating the BASELINE methods performance versus the studied ML methods for SafeRevert). Since the peer review was completed, a bug in the verification system was identified and fixed. The bug caused a proportion of incorrect culprit changes to "get stuck" in a queue waiting for an additional test execution due to a typo in a comparison (using > instead of >=). Once the bug was fixed, the research team was able to rerun the study with the additional label data.

Post-rerun we observed: 1) the BASELINE method performed worse and 2) the studied methods were robust to the change in labeling data. In the reviewed manuscript, XGB(ALL) had the following results in RQ3: 13 Total Bad Reverts, 1817 Good Reverts, 0.7% Bad Revert Rate, and 55.8% Recall; BASELINE(5) was compared against and had 8 Total Bad Reverts, 1286 Good Reverts, 0.6% Bad Revert Rate, and 39.4% Recall. Compare against Table V and Figure 2. We switched to using BASELINE(10) for this final manuscript as it is the production method currently used at Google.

VI. Related Work

In this paper, we are presenting what we believe to be a novel problem to the wider software engineering community: how to safely choose changes to revert with incomplete but suggestive evidence. This problem relies on identifying these problematic changes. In our case we identify the problematic changes to revert via automated culprit finding [14], [23], [24], [25], [26], [27], [28], [29], [30], [4].

Culprit finding's development has often occurred outside of the academic literature. The first reference to the process of using binary search to identify a bug was in Yourdon's 1975 book on Program Design: Page 286, Figure 8.3 titled "A binary search for bugs" [31]. The process is explained in detail in the `BUG-HUNTING` file in the Linux source tree in 1996 by Larry McVoy [32], [33]. By 1997, Brian Ness had created a binary search based system for use at Cray with their "Unicos Source Manager" version control system [34], [33]. Previously in [4] we had credited Linus Torvalds for the invention based on lack of findable antecedent for his work on the `git bisect` command [14]. We regret the error and recognize the above individuals for their important contributions.

Other methods for identifying buggy code or breaking changes have been studied (some widely) in the literature. Fault Localization looks to identify the buggy code that is causing either operational failure or test failures [35], [36]. Methods for fault localization include: delta debugging [13], statistical coverage based fault localization [37], [38], [39], [40], [41], [42], information retrieval (which may include historical information) [43], [44], [45], and program slicing [46], [47], [48].

Bug Prediction attempts to predict if a change, method, class, or file is likely to contain a bug [49], [50], [51]. This idea has conceptual similarities to the work we are doing in this paper where we are using similar metrics to predict whether or not a change which has been implicated by culprit finding is in fact the change that introduced the bug. Our methods could be potentially improved by incorporating the additional features used in the bug prediction work such as the change complexity, code complexity, and object-oriented complexity metrics.

Test Case Selection [52], [53], [54], [55], [56], [57] and Test Case Prioritization [58], [59], [60], [57] are related problems to the change reversion problem we study. Instead of predicting whether or not a change caused a known test failure in Test Case Selection/Prioritization often the change is used to predict whether a given test will fail before it is run. There is a large body work in that uses dynamic information from past test executions (such as code coverage) to inform the selection process. We believe this hints that such information could be highly informative for the change reversion problem as well.

Finally, there are a family of methods for finding bug inducing commits for the purpose of supporting studies that data mine software repositories [61], [62], [63], [64], [65]. These methods typically used to conduct a historical analysis of a repository rather than as an online detection as in culprit finding.

VII. Conclusion

We presented: SafeRevert a method for improving systems that automatically revert changes that break tests. SafeRevert was developed as a way to improve the number bad changes automatically reverted while maintaining safety (rarely reverting good changes). To evaluate SafeRevert, we performed an empirical evaluation comparing the performance of SafeRevert against the baseline method that is currently utilized in production which utilizes a simple heuristic to determine if a change is safe to revert: the number of tests which "blame" the culprit change. When evaluating RQ3 in Section V, it was observed that the XGB(ALL) configuration of SafeRevert doubled the number changes reverted while reducing the number of bad reverts performed (see Table V).

While it is unlikely that a replication study in a different development environment would reproduce our exact results we do expect based on the robust difference observed between SafeRevert and BASELINE that SafeRevert (or similar ML based method) will be able to improve the number of changes eligible for automatic reversion. We hope that by introducing this problem to the larger software engineering community that new and innovative approaches to solving it will be developed.

References

[1] M. Fowler, "Continuous Integration," 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[2] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[3] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. Piscataway, NJ, USA: IEEE, May 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[4] T. A. D. Henderson, B. Dorward, E. Nickell, C. Johnston, and A. Kondareddy, "Flake Aware Culprit Finding," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2023.

[5] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms Using Association Rules," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 39–48. [Online]. Available: http://ieeexplore.ieee.org/document/7202948/

[6] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 91–100. [Online]. Available: https://ieeexplore.ieee.org/document/8804462/

[7] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," 2011. [Online]. Available: http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html

[8] J. Micco, "Continuous Integration at Google Scale," EclipseCon 2013, Mar. 2013. [Online]. Available: https://web.archive.org/web/20140705215747/https://www.eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf

[9] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A. R. Adl-Tabatabai, "Keeping master green at scale," *Proceedings of the 14th EuroSys Conference 2019*, 2019.

[10] A. Najafi, W. Shang, and P. C. Rigby, "Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 213–222. [Online]. Available: https://ieeexplore.ieee.org/document/8804426/

[11] M. Bland, "The Chris/Jay Continuous Build," Jun. 2012. [Online]. Available: https://mike-bland.com/2012/06/21/chris-jay-continuous-build.html

[12] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall, "Scalable build service system with smart scheduling service," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, Jul. 2020, pp. 452–462. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397371

[13] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 253–267, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/318774.318946

[14] C. Couder, "Fighting regressions with git bisect," *The Linux Kernel Archives*, vol. 4, no. 5, 2008. [Online]. Available: https://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html

[15] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–74, Jan. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3476105

[16] J. Micco, "Tools for Continuous Integration at Google Scale," Google NYC, Jun. 2012. [Online]. Available: https://youtu.be/KH2_sB1A6lA

[17] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing Transition-Based Test Selection Algorithms at Google," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, May 2019, pp. 101–110. [Online]. Available: https://ieeexplore.ieee.org/document/8804429/

[18] R. L. Rivest, A. R. Meyer, and D. J. Kleitman, "Coping with errors in binary search procedures (Preliminary Report)," in *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing - STOC '78*. San Diego, California, United States: ACM Press, 1978, pp. 227–232. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800133.804351

[19] A. Pelc, "Searching games with errors—fifty years of coping with liars," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 71–109, Jan. 2002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0304397501003036

[20] M. Ben-Or and A. Hassidim, "The Bayesian Learner is Optimal for Noisy Binary Search (and Pretty Good for Quantum as Well)," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. Philadelphia, PA, USA: IEEE, Oct. 2008, pp. 221–230. [Online]. Available: http://ieeexplore.ieee.org/document/4690956/

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] T. Chen and C. Guestrin, "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: ACM, Aug. 2016, pp. 785–794. [Online]. Available: https://dl.acm.org/doi/10.1145/2939672.2939785

[23] C. Ziftci and V. Ramavajjala, "Finding Culprits Automatically in Failing Builds - i.e. Who Broke the Build?" Apr. 2013. [Online]. Available: https://www.youtube.com/watch?v=SZLuBYlq3OM

[24] C. Ziftci and J. Reardon, "Who broke the build? Automatically identifying changes that induce test failures in continuous integration at google scale," *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*, pp. 113–122, 2017.

[25] R. Saha and M. Gligoric, "Selective Bisection Debugging," in *Fundamental Approaches to Software Engineering*, M. Huisman and J. Rubin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, vol. 10202, pp. 60–77. [Online]. Available: https://link.springer.com/10.1007/978-3-662-54494-5_4

[26] A. Najafi, P. C. Rigby, and W. Shang, "Bisecting commits and modeling commit risk during testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, Aug. 2019, pp. 279–289. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338944

[27] M. J. Beheshtian, A. H. Bavand, and P. C. Rigby, "Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2784–2801, Aug. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9392370/

[28] J. Keenan, "James E. Keenan - "Multisection: When Bisection Isn't Enough to Debug a Problem"," Jun. 2019. [Online]. Available: https://www.youtube.com/watch?v=05CwdTRt6AM

[29] G. An and S. Yoo, "Reducing the search space of bug inducing commits using failure coverage," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 1459–1462. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3473129

[30] F. S. Ocariza, "On the Effectiveness of Bisection in Performance Regression Localization," *Empirical Software Engineering*, vol. 27, no. 4, p. 95, Jul. 2022. [Online]. Available: https://link.springer.com/10.1007/s10664-022-10152-3

[31] E. Yourdon, *Techniques of Program Design*. New Jersey: Prentice-Hall, 1975.

[32] L. McVoy, "BUG-HUNTING," Mar. 1996. [Online]. Available: https://elixir.bootlin.com/linux/1.3.73/source/Documentation/BUG-HUNTING

[33] R. Cox, B. Ness, and L. McVoy, "Comp.lang.compilers "binary search debugging of compilers"," May 2023. [Online]. Available: https://groups.google.com/g/comp.compilers/c/vGh4s3HBQ-s/m/Chvpu7vTAgAJ

[34] B. Ness and V. Ngo, "Regression containment through source change isolation," in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*. Washington, DC, USA: IEEE Comput. Soc, 1997, pp. 616–621. [Online]. Available: http://ieeexplore.ieee.org/document/625082/

[35] P. Agarwal and A. P. Agrawal, "Fault-localization Techniques for Software Systems: A Literature Review," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–8, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2659118.2659125

[36] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7390282/

[37] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002.

[38] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://portal.acm.org/citation.cfm?id=1101949

[39] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, Feb. 2014. [Online]. Available: http://doi.wiley.com/10.1002/smr.1616

[40] T. A. D. Henderson and A. Podgurski, "Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Vasteras: IEEE, Apr. 2018, pp. 93–104. [Online]. Available: https://ieeexplore.ieee.org/document/8367039/

[41] T. A. Henderson, A. Podgurski, and Y. Kucuk, "Evaluating Automatic Fault Localization Using Markov Processes," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Cleveland, OH, USA: IEEE, Sep. 2019, pp. 115–126. [Online]. Available: https://ieeexplore.ieee.org/document/8930843/

[42] Y. Kucuk, T. A. D. Henderson, and A. Podgurski, "Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 649–660. [Online]. Available: https://ieeexplore.ieee.org/document/9402143/

[43] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," *Proceedings - International Conference on Software Engineering*, pp. 14–24, 2012.

[44] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug Localization Based on Code Change Histories and Bug Reports," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. New Delhi: IEEE, Dec. 2015, pp. 190–197. [Online]. Available: http://ieeexplore.ieee.org/document/7467300/

[45] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with BERT," in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 946–957. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510042

[46] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions of Software Engineering*, vol. 16, no. 9, pp. 965–979, Sep. 1990. [Online]. Available: http://dx.doi.org/10.1109/32.58784

[47] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *International Conference on Software Engineering*, vol. 9. Springer, 1992, p. 349. [Online]. Available: http://portal.acm.org/citation.cfm?id=24041&dl=

[48] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. Vancouver BC Canada: ACM, Oct. 2004, pp. 432–448. [Online]. Available: https://dl.acm.org/doi/10.1145/1028976.1029012

[49] C. Lewis, Z. Lin, C. Sadowski, and X. Zhu, "Does bug prediction support human developers? findings from a google case study," in *Proceedings of the …*, 2013, pp. 372–381. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486838

[50] K. Punitha and S. Chitra, "Software defect prediction using software metrics - A survey," in *2013 International Conference on Information Communication and Embedded Systems (ICICES)*. Chennai: IEEE, Feb. 2013, pp. 555–558. [Online]. Available: http://ieeexplore.ieee.org/document/6508369/

[51] H. Osman, M. Ghafari, O. Nierstrasz, and M. Lungu, "An Extensive Analysis of Efficient Bug Prediction Configurations," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. Toronto Canada: ACM, Nov. 2017, pp. 107–116. [Online]. Available: https://dl.acm.org/doi/10.1145/3127005.3127017

[52] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, vol. 2003-Janua. IEEE, 2003, pp. 442–453. [Online]. Available: http://ieeexplore.ieee.org/document/1251065/

[53] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, Jan. 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584909001219

[54] Z. Q. Zhou, "Using coverage information to guide test case selection in Adaptive Random Testing," *Proceedings - International Computer Software and Applications Conference*, pp. 208–213, 2010.

[55] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, pp. 1–10, 2015.

[56] S. Musa, A. B. M. Sultan, A. A. B. Abd-Ghani, and S. Baharom, "Regression Test Cases selection for Object-Oriented Programs based on Affected Statements," *International Journal of Software Engineering and Its Applications*, vol. 9, no. 10, pp. 91–108, Oct. 2015.

[57] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, p. 29, Mar. 2022. [Online]. Available: https://link.springer.com/10.1007/s10664-021-10066-6

[58] Y. Singh, A. Kaur, B. Suri, and S. Singhal, "Systematic literature review on regression test prioritization techniques," *Informatica (Slovenia)*, vol. 36, no. 4, pp. 379–408, 2012.

[59] H. de S. Campos Junior, M. A. P. Araújo, J. M. N. David, R. Braga, F. Campos, and V. Ströele, "Test Case Prioritization: A Systematic Review and Mapping of the Literature," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*. New York, NY, USA: ACM, 2017, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/3131151.3131170

[60] M. D. C. De Castro-Cabrera, A. García-Dominguez, and I. Medina-Bulo, "Trends in prioritization of test cases: 2017-2019," *Proceedings of the ACM Symposium on Applied Computing*, pp. 2005–2011, 2020.

[61] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, p. 1, Jul. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1082983.1083147

[62] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, Jul. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584917304275

[63] M. Borg, O. Svensson, K. Berg, and D. Hansson, "SZZ unleashed: An open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. Tallinn, Estonia: ACM Press, 2019, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3340482.3342742

[64] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn Estonia: ACM, Aug. 2019, pp. 326–337. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338962

[65] G. An, J. Hong, N. Kim, and S. Yoo, "Fonte: Finding Bug Inducing Commits from Failures," Feb. 2023. [Online]. Available: http://arxiv.org/abs/2212.06376