Speculative Testing at Google with Transition Prediction

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

***Abstract*—Google's approach to testing includes both testing prior to code submission (for fast validation) and after code submission (for comprehensive validation). However, Google's ever growing testing demand has lead to increased continuous integration cycle latency and machine costs. When the post code submission continuous integration cycles get longer, it delays detecting breakages in the main repository which increases developer friction and lowers productivity. To mitigate this without increasing resource demand, Google is implementing Postsubmit Speculative Cycles in their Test Automation Platform (TAP). Speculative Cycles prioritize finding novel breakages faster. In this paper we present our new test scheduling architecture and the machine learning system (Transition Prediction) driving it. Both the ML system and the end-to-end test scheduling system are empirically evaluated on 3-months of our production data (120 billion test×cycle pairs, 7.7 million breaking targets, with ∼20 thousand unique breakages). Using Speculative Cycles we observed a median (p50) reduction of approximately 65% (from 107 to 37 minutes) in the time taken to detect novel breaking targets.**

I. Introduction

Continuous integration (CI) coordinates the development activity of large numbers of developers [1]. When two developers are working in the same portion of the code base, continuous integration ensures that conflicting changes combine into a conflict-free version before reaching the end user. In general, adopters of CI execute builds and tests to ensure that the final release artifact (server binary, mobile application, etc...) passes all relevant tests. These build and tests are executed frequently to reduce the amount of time conflicts are present in the code base. Continuous integration is both a system and a practice of automatically merging changes into a source of truth for an organization's source code and related artifacts.

For small software repositories and organizations, the implementation of continuous integration is well supported by off-the-shelf software such as Github Actions, CircleCI, Jenkins, and numerous other tools. However, as the repository and organization scales up challenges emerge.

As an organization adds projects and engineers, there are two distinct paths that emerge: the many-small-repository path and the single-mono-repository path. Both paths have distinct challenges and advantages. Neither path will enable organizations to use vanilla off-the-shelf solutions for their development environment. This paper is focused on a particular challenge faced in a mono-repository environment at Google, but similar problems arise in organizations with many small repositories that are strongly coupled together.

Google, an early adopter of mono-repositories and continuous integration [2], faces an enormous ever expanding code base that is centrally tested by the Test Automation Platform (TAP) [3]. As the code base and company grows, so does the *demand* for compute resources for continuous testing. Left unchecked, we have observed a double digit percentage organic demand growth rate year-over-year (compounding). Such a growth rate is untenable even for a large company like Google. To prevent this unchecked growth in resource demand, TAP has long had features that reduce demand by skipping some tests (before submission) and batching many versions together (after submission) to amortize the cost [4], [5], [3], [6], [7], [8], [9], [10], [11], [12], [13].

In this paper, we are primarily concerned with testing that occurs after the code has been submitted. We examine how to improve the developer experience by finding novel build/test breakages faster while continuing to control our resource footprint growth rate.

A. The Google Development Environment

Google's development environment uses a centralized repository [2] where code changes are managed with Blaze/Bazel[1] and automatically tested, primarily by TAP. Developers write code, build and test locally, then submit it for review via Critique [14], triggering Presubmit checks. After review and approval, changes undergo further checks before merging. Due to high submission rates, Presubmit testing is limited to avoid excessive resource consumption and delays.

B. TAP Postsubmit

When a developer's change gets submitted it may still have a bug that breaks an existing test or causes a compile breakage in another part of the repository. To find these bugs that have slipped through the pre-submission testing and validation process, TAP also has a "post-submission" mode (TAP Postsubmit).

In Postsubmit, TAP periodically (subject to compute resource availability in the Build system) schedules *all* tests that have been *affected* (based on build dependencies) since

[1]https://bazel.build/

the last run. We refer this execution cycle as *Comprehensive Testing Cycle*. Previously, this cycle has also been referred as Milestones [7]. There are two primary objectives for the Comprehensive Cycle:

1) Uncover all test failures at the version comprehensive testing was conducted at.
2) Provide project health signals for downstream services to trigger more expensive testing and/or start production rollout by triggering release automation

As of today's writing, it takes ∼1-2 hours for a test broken by a developer's change to start failing in TAP Postsubmit. Once the failure is detected automatic culprit finders [11] and rollback system [12] will spring into action to help keep the repository healthy and our developers productive. Automatically rolling back the change (after the first breakage was detected) might take as little as 30 minutes or as long as several hours depending on a number of factors. Thus, it could be 4 hours or more before our developer learns they have broken a test and either been notified or automatically had their change rolled back.

C. Cost of Breakages

The longer it takes to identify and fix a code breakage, the more challenging and expensive the remediation process becomes. This delay can lead to a loss of context for the original developer, potentially requiring teammates to resolve the issue. Additionally, a bad change can affect other developers, especially if it occurs in widely used parts of the code base. When interrupted by such a failure programmers need to distinguish between the change they are making and existing fault in the repository. Ultimately, prolonged breakages can disrupt release automation and even delay production releases.

D. Our Contributions and Findings

In this paper we are looking to increase developer productivity by reducing their friction with respect to broken code in the main code base. Specifically we are solving the following problem.

> ***Problem***: How can CI minimize the time between the submission of a faulty change and its identification within a continuous integration (CI) system to accelerate the detection and mitigation of bugs?

To solve the problem, we propose a new scheduling mode to our CI system, Speculative Testing Cycles, that opportunistically runs a small subset of tests to uncover novel failures and reduce the mean time to detection (MTTD) for new failures. Speculative Cycle prioritizes finding novel breakages by identifying tests that are very likely to fail. The new scheduling mode is driven by a *Transition Prediction* model that predicts when tests are likely to (newly) fail. It utilizes shallow machine-learning to run a smaller batch of tests predicted to be more likely to be broken at higher frequency in order to find breakages faster. We discuss both the design and constraints on the production Speculative Cycle system and the shallow machine-learning model (Transition Prediction) that powers this predictive test selection.

While the production system is highly tied to Google's unique development environment and organizational constraints (i.e.: large monorepos with centralized CI infrastructure), our ML model presents a very coarse-grain approach to prediction for test selection using test and code-under-test metadata that has equivalents in most development environments and is language/framework agnostic.

Our contributions in this paper are therefore:

1) Speculative Cycles: A system for frequent / cost-aware batch testing.
2) Transition Prediction (TRANSPRED): Predicting target status transitions (Pass to Fail) for arbitrary build & test targets. With a budget of 25% of the total targets, Transition Prediction achieves a 85% recall rate in detecting breakages.
3) A very large scale study (utilizing 3-months of production data: 120 billion test×cycle pairs, 7.7 million breaking targets, and ∼20,000 unique breakages) on the efficacy of Speculative Testing using Transition Prediction evaluating its performance against randomized testing.
4) A comparative assessment of dataset and training configurations for Transition Prediction, determining that the selection of features and the length of the training window substantially influence the system's performance.

II. Background

A. Developer Services

1) Bazel: Google uses Bazel as its build system monorepo wide. Bazel allows specifying build/test "targets" that depend on other targets. Each target is comprised of a set of "actions" to be executed. Therefore, the execution of a target corresponds to a Directed Acyclic Graph (DAG) of actions. Given Bazel makes dependency management declarative and all dependencies are built at the same version of the codebase, builds are ostensibly held to be hermetic – builds at the same version of the codebase should produce the same output for each action. This opens up the possibility for massive compute savings through caching of intermediate actions across multiple builds.

2) Forge: Build/Test actions at Google run under a centralized compute cluster called Forge which attempts to cache these intermediate actions to avoid recomputation at the same version of the codebase. For Postsubmit testing, we make use of this caching by batching builds/tests at single versions every hour or so.

B. Comprehensive Testing Cycles

In the standard case, TAP Postsubmit performs each Comprehensive Testing Cycle once the previous cycle's active consumption of resources drops below a threshold of their allocated build system resources. This cycle consists of picking a recent change(snapshot of the repository), at which we batch and run a "comprehensive" set of tests. Comprehensive cycles run all tests that are "affected" since the previous cycle.

"Affected" is our term for dependence as implied by the build system's package graph, allowing us to perform a degree of static test selection at a coarse granularity.

C. Ancillary Systems For Build Gardening

The development life cycle explained above relies on Postsubmit testing to make up for the lack of comprehensive testing prior to submission. This introduces the concept of *Postsubmit breakages* which are caught sometime after submission. The process of detecting the breakage, identifying the cause of the breakage (culprit finding), and fixing the breakage (sometimes via a rollback) is called "Build Gardening" at Google.

1) Culprit Finding: We have previously outlined Google's culprit finding approach in [11], [12]. To formulate the problem, we are given a test/build "target" and must determine at which change C over a range of sequential changes $[A, B]$ did the target start breaking. The change *C* is referred as a *Culprit Change* or simply *Culprit*. Given the presence of build/test non-determinism ("flakes"), we want to find the change at which the test was "truly" failing but "truly" passing at the previous change. Our previous work explores how we do this in a time and run-efficient manner using a historical flake rate aware Bayesian model to sample runs.

2) Culprit Verifier: Unfortunately, persistent flakiness and non-determinism can still slip by the culprit finders. The pathological case is that of build system non-determinism. Google's build system Bazel assumes that builds are hermetic and should produce the same output at the same version. This assumption means that unlike tests, failed actions are cached by the build system so immediate reruns will continue to assert what could potentially be a flaky build failure. The culprit verifier does extensive reruns on a subset of culprit conclusions produced by the culprit finders. We sample from two different sampling frames: (1) a simple random sample of all conclusions produced and (2) the first conclusion produced for each uniquely blamed culprit. For details on the design of the verification system, see the Flake Aware Culprit Finding paper [11] and the SafeRevert paper [12].

Although the verifier is fallible, the dataset it produces is still the most accurate representation of true breakages at Google. We know it fails because we have seen specific examples of its failures. But, from a measurement perspective, bounding its accuracy is challenging because its failure rate is low enough that it exceeds our ability to accurately measure. We have recently conducted manual verification while launching a new version of our Auto Rollback system on all culprits rolled back during the "dogfood" phase (an opt-in beta). Out of 250+ hand verified culprits: 5 were incorrect culprits from the culprit finder, 2 incorrect culprits were incorrectly verified correct by the verifier. This gives us an estimated culprit finding accuracy of 98% (matching verifier produced accuracy dashboard) and a verifier accuracy of 99.2%. Given the low sample size the confidence interval on that measurement is $99.2 \pm 3.3\%$ which is too wide. To get sufficient power to accurately estimate the verifier accuracy, we would need to hand verify at least 5000 culprits.

Thus, while we are confident in the overall accuracy measurements provided by the verifier, we acknowledge that "ground truth" is difficult to obtain. We utilize the dataset produced by the verifier to label the culprits to train our prediction model (detailed below). This approach vastly reduces the impact of flakiness on our model training data.

3) Autorollback: At Google, there is a developer guideline to prefer rollbacks to fix forwards. Given this context, it is appropriate to automatically roll back a change if someone is confident that the change broke the build. Unfortunately, given the accuracy issue of the culprit finder described above, we can't immediately rollback upon a culprit finding completion for one target. In the SafeRevert paper [12], we discussed how we attempt to determine the amount of evidence needed to proceed with a rollback. In practice, we currently restrict rollbacks to changes that break at least 10 targets.

D. The Evolution of TAP

Prior to development of TAP, Google had a more traditional continuous integration system. Primarily, the decision to run continuous integration was left up to individual teams and the system that existed was federated: teams brought their own capacity (machines, possibly under their desks) and had to configure and run the CI system(s). TAP, upon launch in 2009, was a massive improvement from setup and continuous maintenance perspective alone. Unlike the previous systems, TAP ran all builds centrally and only required a small amount of configuration: it simply required users to specify which paths they would like to test and where the (failure) notifications should go.

The central builds and low configuration overhead made TAP immediately successful. Within a few years it had completely displaced the prior system. But, success came at a price: it was plagued by build capacity limitations and scalability challenges stemming from design decisions of running tests on each change in the initial implementation. By 2012 (following a full launch in 2010) Google was running low on machine capacity to run TAP. The testing model had naively assumed that the build dependence based test selection would be enough to control demand; It was not.

TAP both pre- and post-submit were thus rewritten (multiple times). TAP Presubmit now has several different modes, adapts to resource availability, uses machine learning driven test selection, and has "advisors" which can ignore or rerun tests on failure [13]. Unfortunately for you (the reader) discussing these innovations is out of scope for this manuscript.

Today, TAP Postsubmit no longer runs on every change. It waits for there to be capacity in the Build System [9], [10] and then enqueues all tests that have been affected since their last definitive status. At Google's scale and growth rate, we have had to continuously innovate and rewrite core parts of TAP just to maintain this model. However, we have long known that just using build-graph dependency based test selection in Postsubmit was untenable in the long term [7], [8]. It is only now that we have both the high-accuracy culprit finding infrastructure [11], [12] and the organizational will to

fundamentally change (again) core assumptions for how TAP works.

E. The Current Dilemma

The cost of TAP Postsubmit's *comprehensive* testing continues to rise due to ever-increasing compute demand, increasing fleet machine diversity, and evolving development practices. This means that with no systemic changes in our approach towards Postsubmit testing:

1) Teams start needing to wait longer and longer for signal on their projects' health (or "greenness") from TAP blocking their releases which has significant monetary impact
2) Developers are now alerted hours after they submit of a breaking change, creating developer toil

III. Speculative Testing Cycles

In order to protect release quality and improve developer productivity, TAP is introducing *Speculative Testing Cycles* - a process by which we non-deterministically select a smaller subset of targets that we deem more likely to fail, and run this smaller set more frequently than traditional comprehensive cycles. For traditional comprehensive test cycles, we attempt to batch as many builds/tests together at the same change as possible to get the caching benefits from Forge. We do the same here for Speculative Cycles, picking a single change approximately every 20 minutes, Forge build resources allowing. As with Comprehensive Cycles, we first perform static build system level dependency analysis to find all targets which could have a status change.

A. Determining Breakage Likelihood

We now have a set of statically-filtered targets $\{T\}$ and a sequence of commits $\{C\}$ consisting of all commits since the last Speculative Cycle. Our problem is then to determine for a given target $t \in T$: what is the likelihood it was broken by a change $c \in C$ affecting it? For compute tractability, given the frequency of change submissions and number of targets in our codebase, we simplify by choosing to aggregate across C and ask: "Was target t broken by any change in C?" We identify this problem formulation and approaches to answer it as *Transition Prediction*, and discuss it in detail in the next section.

B. Making Scheduling Decisions

Our Transition Prediction model gives us "scores" from 0 to 1 indicating the propensity of this target to be newly broken. Given that we rely on machine learning models for these predictions, we must be careful to note that these scores do not meaningfully represent probabilities. They are the product of an algorithm attempting to minimize a loss function relative to its training dataset. While they can be colloquially understood to imply likelihood and allow relative comparison, moving the threshold at which scores map onto decisions to (not) schedule will produce non-linear changes to the actual observed probability of finding a breakage ("recall") or scheduling false positives ("false positive rate").

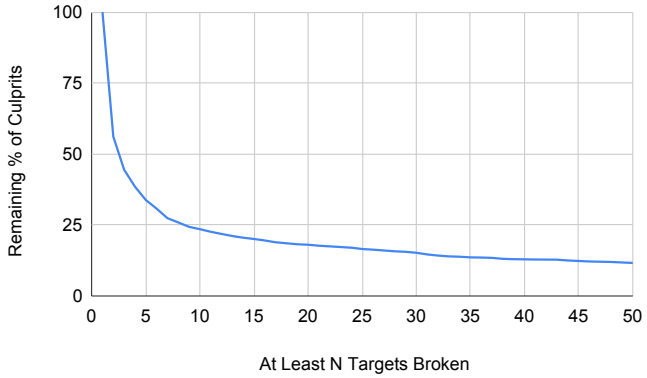

Fig. 1: The proportion of culprit changes remaining as we filter for minimum number of targets broken. Close to 50% of culprits break only a single target. The curve quickly flattens out giving us a reliable and tractable quarter of culprits having broken at least 10 targets.

Given this fact and our desire to have stable execution cost/latency under our scheduling scheme, we choose to primarily rank targets and schedule the top-k highest ranking targets under some top K parameter. For the purpose of this paper, this is our main parameter for tuning the Speculative Cycle system while the remaining knobs exist in feature selection and training configuration for the Transition Prediction ML model.

C. Goal of Finding Test Breakages

As we've discussed, Postsubmit testing feeds into both Release pipelines and the breakage triage/fixing workflow (what we call "Build Gardening"). Release implications of Speculative Cycle(s) depend on many several other projects at play which are outside the scope of this paper. The direct outcome is that newly detected build/test failures will now propagate to Build Gardening consumers faster, namely:

1) Developers directly through emails sent upon detection of a failure
2) Automated Culprit Finders that attempt to identify the specific culprit change
3) Auto-Rollback reverts changes once it acquires enough evidence from culprit finders

A developer or the automated culprit finders may only need a single target breakage (We will denote this criterion as $AtLeast(1)$) to consider investigating and to identify a breaking change that needs to be reverted. But given the non-determinism issues discussed above, auto-rollback requires a stronger signal – at least N distinct targets were culprit-found to the same change ($AtLeast(N)$). Both these goals are important to the health of the codebase, and while the former is a simple and more intuitive metric, there is a strong argument to be made for the latter – in that the number of distinct broken targets is a good proxy for the cost of that bad commit. The larger the number of broken targets, the larger the likelihood that it falls under a configured Presubmit testing directory, impacting developer productivity like we discussed above. Targets could belong to multiple release-level project

groups, and more targets being broken is a higher cost to overall release health with more automation and production pushes being delayed or blocked. Figure 1 presents the total proportion of culprits remaining as we increase N which follows a power law distribution.

D. Evaluating Performance

Culprit detection is then split between Comprehensive and Speculative Cycles. Under a specific scheduling scheme, Speculative Cycles successfully detect some proportion of culprits before Comprehensive Cycles. This proportion is the recall of the system. This recall in combination with the actual run frequency and execution runtime of the two cycles determines the actual reduction in culprit detection latency, where latency is defined relative to the submission time of the culprit change. Depending on the consumer, latency improvements will have non-linear marginal payoffs. For example, reduction from half a day to an hour may have measurable impacts on development friction caused by failing tests, but reduction from a half hour to 10 minutes may be negligible. Therefore, both metrics can be compared to the cost of a specific scheme in order to justify value / viability of the system.

IV. Transition Prediction

Transition Prediction (or *TRANSPRED* for short) attempts to predict transition likelihood every 20 minutes for millions of targets with a submission rate of over tens of thousands of changes per day. In order to make this tractable and most importantly cost-beneficial relative to simply running more tests, our predictions must be cheap and low latency. This informs using simple tree-based model techniques like Gradient Boosted Decision Trees and Random Forests. Tree-based models are used across industry for being low cost for inference, avoid over-fitting, and achieve high performance with simple coarse grained features like change and target metadata. We have found consistently throughout our work that Gradient Boosted Tree models perform best for Google's coarse grained test selection problems.

A. Features

Figure 2 shows an overview of the structure of the feature vector. In a production system, only features that can be acquired quickly and reliably at scale can be used. The end-to-end latency of deciding which tests to run (including fetching the data, assembling the feature vectors, and running the prediction) needs to be well under 20 minutes such that the current Speculative Cycle finishes before the next one begins. Thus, we are using simple, cheap to compute, coarse grained metadata features from targets and commits. The predictions are done per target so the features for one target are included in each feature vector. However, the predictions are not done for a single commit but across a range of commits. To increase the robustness of the commit based features, we only include commits that *affect* the target. A commit affects a target if the target is reachable in the build graph from the files modified in the change.

TABLE I: Example features used in our model. The production model uses 'BASE' features, while 'AUG' features were added for this paper. We have a total of 38 BASE features and 11 AUG features.

	Type	**Example Features**
BASE	Commit Metadata	# reviewers, # approvers
		# of bugs: associated and fixed
		Lines of code
	Presubmit History	# passing Presubmits over 28 days
		# failing Presubmit results over 28 days
		# failing Presubmits in the cycle window
	Postsubmit History	# passing runs over 28 days
		# flaky runs over 28 days
	Commit Features	Minumum build graph distance [7]
AUG	Commit Features	# of robot authors
	Target Metadata	Target Language (eg: Java, C++, sh)
		Target Type: build vs test
	Additional History	# true breakages over 28 days

Target level features include static and dynamic characteristics. The static items are unchanging (or very rarely changing) and include target's programming language, whether it is a test or build target, and its Bazel rule type, etc. The dynamic items are based on historical execution data, such as the failure count in Postsubmit and Presubmit over several different time windows. Commit level features include: change metadata like lines of code modified, number of reviewers, number of linked bugs, description, etc. Finally, some features correspond to the relationship between the target and the changes in this specific cycle – specifically its build graph distance to the files modified and whether the target was run at Presubmit time.

Table I contains details on the features used in the production model ('BASE') and those added for this paper ('AUG'), for augmented).

B. Labeling

Our data set has a row/vector for each target in each Speculative Cycle. In order to train the model, we need to label each vector with whether or not that target should be run in the given Speculative Cycle. Optimally the targets should only be run when a new failure is introduced into the repository. We use the verified culprits data set described previously [11], [12] as our source of truth on which commits introduced breakages. Unfortunately, the automated culprit finders de-prioritize comprehensiveness in favor of finding results quickly for active breakages blocking developers. This can leave some target level breakages with no culprit-identifying labels for up to two weeks after the failures was detected.

In addition, some flakiness in tests and builds may be caused by something outside of the code and configuration checked into the repository. For example, a buggy compiler action may non-deterministically miscompile a file causing the linker to fail when it happens. Because the compile action "succeeds" (produces an output and no error code), the output of the compiler (the object file) will be cached. Subsequent builds will re-use the cached output and fail when attempting to link. However, if the cached item expires or the build happens in a different data center with a different cache the linker action

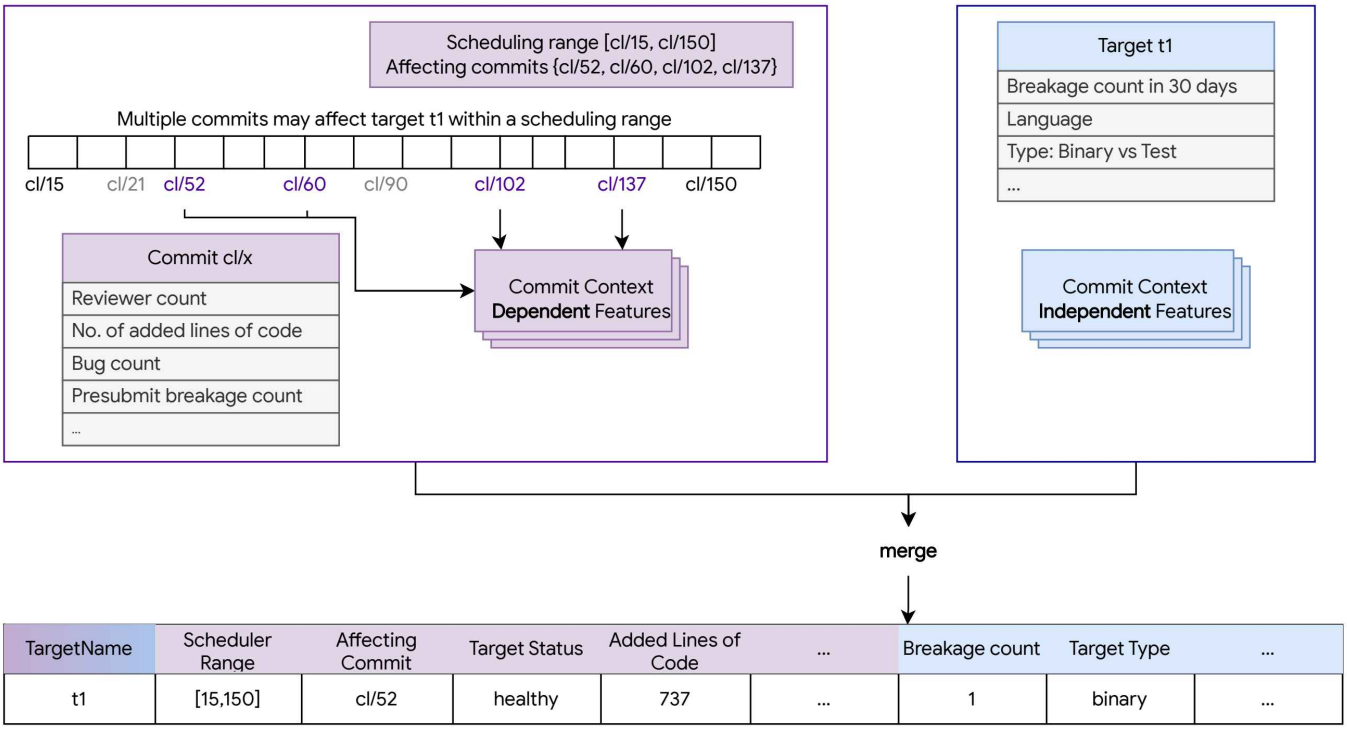

Fig. 2: Static target features like type of target and language are combined with commit metadata such as number of lines of code modified, number of reviewers etc to form the final feature set

may succeed (when the compiler behaved correctly). This type of flakiness is difficult to diagnose real-time with only build re-executions due to the multiple layers of caching. However, build executions separated in time have a better chance of identifying the flaky behavior.

In order to protect against unavailability of data, we perform labeling a full week after the inference examples are acquired. As we've described, this still does not completely shield us from both incorrect positives and lack of coverage but is the best ground truth we have.

C. Super Extreme Class Imbalance

Postsubmit breakages are exceedingly rare. The only target-vector rows in our dataset that need to be scheduled are targets that are newly broken. This only occurs 0.0001% of the time. In model training, we refer to the class with fewer examples at the *minority class* while the class with more examples is the *majority class*. In our problem, the minority class is these newly broken targets just described.

When there are too few examples in the minority class, the effectiveness of the many training algorithms is reduced. The "loss" (e.g. the function model training is optimizing) becomes biased towards the majority class. A classic pathological case is a model that trivially predicts the majority class – that is, predicts the majority class every time and never predicts the minority class. These models will actually minimize the loss function even though they never make useful predictions. Our approach must guard against these biases if we want to usefully predict which tests to run during a Speculative Cycle.

For our system, we downsample the negative class (targets that are not newly failing). This reduces the imbalance between the classes. It also implicitly values the True Positives (new breakages) higher than the True Negatives. Every false negative (missed breakage) leads to developer toil as the detection of the breakage is delayed. Given the super extreme imbalance between the classes, we expect most targets run during a Speculative Cycle are to be False Negatives (and not indicate a new breakage).

D. Training Configuration

For our model, we use the Yggdrasil Decision Forest library (YDF) [15]. YDF touts superior decision tree sampling and splitting algorithms over prior algorithms (such as XGBoost [**?**]) and provides better integration with Google's machine learning infrastructure.

Conventional training schemes randomly shard their dataset between train, validation, and test splits or perhaps perform cross-validation with a set of parallel splits. For our problem, randomized splits are harmful and lead to us overestimating our performance as we leak time-dependent attributes of the repository across sets. Instead, we create time-ordered splits where the train set consists of the first A days, the validation set the next B days, and the test set the final C days.

V. Empirical Evaluation

A. Evaluation

We empirically evaluated Speculative Cycles against a baseline random scheduler on culprit-detection recall and against an optimal algorithm for evaluating culprit detection latency improvements. We evaluated different dataset pruning/selection and training configurations to discover important hyper-

parameters for the Transition Prediction model on a standard set of features. We examined three research questions:

B. Research Questions

RQ1: What is the performance of Speculative Cycles with Transition Prediction in terms of percentage of novel breakages detected (recall) in comparison to a baseline randomized testing approach?

RQ2: Speculative cycles compete with full testing cycles for machine resources. Do speculative cycles improve productivity outcomes for Google developers?

RQ3: What aspects of the model design most impact system performance? Considered aspects include features selected, super-extreme class imbalance corrections, and training environment.

C. Measuring Performance (Recall)

Speculative Cycles as a system attempts to detect bug introducing changes (culprits) through running and surfacing at least N newly failing targets. Specifically, the goal is to capture the majority of (newly failing) target results that detect novel bugs introduced into the repository with the faster Speculative Cycles instead of the slower Comprehensive Cycles. This naturally lends itself to measurement in the form of recall – the percentage of these events we capture.

We additionally observe that our dataset is imperfect. Not every commit labeled as a "culprit" is indeed a bug introducing change. As discussed above, our datasets contains some inaccuracies particularly when a particular culprit commit only a few targets "blaming" it.[2] The distribution in the number of targets blaming a culprit follows a power law distribution. The majority of culprits are blamed by fewer than 10 targets. However, the lower the number of blamed targets the more likely it is that the culprit finding was inaccurate and that the commit does not in fact introduce a bug. In our labels, the signal is highest for culprits with many targets blaming them and lowest when there is only a single target. To protect our system from the noise these "small" culprits are filtered out during training.

We use $AtLeast(n)$ to denote the subset of culprits that have at least n targets blaming each culprit. In our evaluation, we provide recall over both (a) $AtLeast(1)$ - the set of all culprits and (b) $AtLeast(10)$ - the set of culprits causing at least 10 target breakages where we successfully detect 10 breakages. $AtLeast(10)$ also ensures that we meet the (traditional) minimum required threshold (i.e. 10 failing targets) to trigger an automatic rollback. Speculative Cycles sees its highest value in terms of mean time-to-fix when automatic rollbacks are correctly triggered (especially for commits that have wider impact radius). Therefore, by assessing Recall using $AtLeast(10)$ we also assess whether or not the Speculative Cycle can find enough evidence to trigger rollbacks.

[2]When a target breaks we perform culprit finding. The conclusion of the culprit finding is the culprit commit. We colloquially say the target is blaming the culprit.

D. Measuring Productivity Outcomes (Latency Improvement)

Although recall is a concrete metric, our true goal is to improve productivity by reducing *friction* caused by breakages that slipped into Postsubmit. Speculative Cycles focuses on early detection of breakages (and their culprit detection and fix) to reduce the friction caused by bad commits in Postsubmit. Its impact is measured by the reduction in "breakage detection latency". Early detection and fix also improves the mean time to fix latency for bad commits. We will evaluate the "breakage detection latency" decrease as a percentage relative to the current latency using empirically observed timings for running both Speculative and Comprehensive over the days present in our test dataset.

E. High Priority Configuration of Training Scheme

In general, it is considered good practice in machine learning to prioritize dataset quality over attempting to fine tune model level hyper-parameters. We have similarly observed little value in specific training parameters and use YDF's defaults for Gradient Boosted Trees for this work to show the general case performance expected. The more interesting questions that pertain to our dataset configuration are:

1) Training Split / Window Size: As we use time based splitting to avoid the issue of "time-traveling" across the datasets, our training split configuration determines both the training "window size" and the "delay" between the training and test datasets.

We hypothesized that the training data may have some recency bias. A shorter/ more recent look-back horizon on the training data may contain more useful signal for predicting the likelihood of a target transitioning. A longer window could dilute the usefulness of more recent breakages, while introducing noise from old breakages that have since been fixed and are less likely to reoccur. In order to test this hypothesis, we ran the model five times using the same feature set and configuration, only changing the number of weeks of data used during training. Note that our production pipeline uses a 3:1:1 week split for training:test:validation.

2) Downsampling and Upweighting: Our extreme class imbalance of $40,000 : 1$ necessitates downsampling our negative class to a ratio that is compute-efficient, but also avoids trivially predicting 0. Upweighting the positive class can be useful to appropriately value positive examples with respect to the loss function of the training algorithm. The ratios of down-sampling listed above reduced our class imbalance ratio to $400 : 1$, $200 : 1$, and $40 : 1$ respectively. For each case, we then upweighted the positive samples of target breaking commits by the new class imbalance ratio to see what effect, if any, it had on improving model performance.

F. Dataset

Our evaluation dataset consists of roughly 3 months of Transition Prediction data split across Train, Validation, and Test sets. In total, the dataset consisted of 120 Billion target-cycle pairs, 7.7 million of which were breaking targets. This corresponds to $\sim$20 thousand unique culprits. As mentioned

above, we ensure that our splits are entirely sequential and disjoint to avoid cross contamination. The exact split and training configuration is varied while answering RQ3. We then fix our most performant model for evaluating RQ1 against the BASELINE random prediction and for RQ2 against latency in the CI environment with only Comprehensive Cycles.

VI. Results

A. Summary of Results

> ***Note***: For RQ1 and RQ2, the model was trained using the 'AUG' feature set (Table I), a 3-week training window, a 0.01 downsampling rate for the majority class, and no upweighting for the minority class. This configuration matches our production model, except for the feature set. RQ3 explores performance across configurations.

RQ1*: What is the performance of Speculative Cycles with Transition Prediction in terms of percentage of novel breakages detected (recall) in comparison to a baseline randomized testing approach?*

> ***Summary***: TRANSPRED demonstrates superior recall compared to the BASELINE model across all budgets (see Figure 3). At a budget of 25% of the total targets, TRANSPRED achieves a recall of 85%, greater than the BASELINE model's recall of 56%. This highlights the model's effectiveness in identifying breaking changes with limited resources.

In the context of Transition Prediction, recall measures the percentage of breaking changes identified using a given budget. The budget parameterizes the recall metric as shown in Figure 3. To simplify the complex accounting of machine and test costs, we use the percentage of targets scheduled by TRANSPRED (versus a Comprehensive Cycle). Our budget for this assessment is 25% of the targets that would have been used by a Comprehensive Cycle. Recall is then defined as percentage of culprits identified out of the total set of culprits.

Figure 3 provides a visual representation of how recall changes as the percentage of scheduled targets varies, comparing the performance of TRANSPRED against the BASELINE model. For each model the figure shows the recall of both $AtLeast(1)$ and $AtLeast(10)$. $AtLeast(1)$ corresponds to finding any target for a breakage or informally, that breakages are at least "identified". The recall for $AtLeast(10)$ concentrates on finding breakages with at least 10 targets broken – including scheduling at least 10 of those broken targets to ensure auto rollback has enough evidence to trigger.

For TRANSPRED, the $AtLeast(10)$ recall is higher than the $AtLeast(1)$ recall. This indicates the model is better at catching bad commits that break 10 or more targets. It's intuitive that larger breakages would be better handled by the model as they are much better represented in the dataset, but an interesting result that we have is better coverage of them, i.e.,

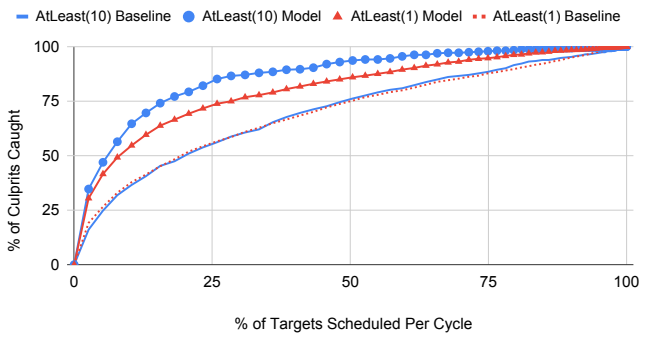

Fig. 3: System culprit recall when scheduling K% of affected targets using Transition Prediction versus using the Baseline (a random subset of targets). Performance is evaluated against *AtLeast1* and *AtLeast(10)* detection criteria. We use *AtLeast(10)* as it matches auto-rollback criteria. Performance actually improves for the model as we go to *AtLeast(10)* indicating that the model is much better at detecting members of larger breakages.

consistently finding more individual targets broken by such commits. Culprits that only break a few tests are arguably less consequential than the few culprits that break a large number of tests. This strong performance on commits that break at least 10 targets indicates that when TRANSPRED does find a break, it has a good chance to be rolled back automatically (see Section II-C3).

Both configurations for TRANSPRED outperform the BASELINE random model, indicating that the model is learning something useful from metadata and that the metadata alone (without considering the content of the changes) can be predictive of breakages. We look forward to future experiments utilizing the content of commits that compare both the efficacy and the model costs.

RQ2*: Speculative Cycles compete with full testing cycles for machine resources, what configurations of Speculative Cycles improve productivity outcomes for Google developers?*

> ***Summary***: Speculative Cycles reduce the median (p50) breakage detection latency by 65% (70 minutes) over the existing Comprehensive Cycles.

Figure 4 visualizes a histogram of the performance of the model in terms of *breakage detection latency*. We consider this metric to be the "outcome metric" for this work as it is a proxy for improving overall developer productivity. It is known from internal work that developer productivity degrades when they need to debug build and test breakages they did not cause during interactive development. Productivity also degrades when they are interrupted to troubleshoot releases stalled due to breakages. By reducing the total time from culprit commit submission to detection, the model reduces the amount of friction developers experience from breakages.

As with RQ1, we have fixed TRANSPRED to represent Speculative Cycles using the same Transition Prediction model and scheduling 25% of targets. In the visualization, the model (in black) is compared against two alternatives. The first labeled "Only Comprehensive" shows the distribution of time

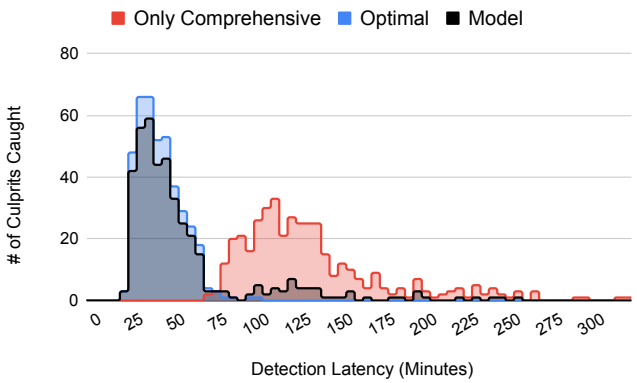

Fig. 4: Histogram of culprit detection latency comparing detection using only Comprehensive Cycles (Only Comprehensive) to Speculative Cycles using Model vs traditional comprehensive test cycles

(in minutes) it takes the current TAP Postsubmit scheduling algorithm (Comprehensive Cycles) to detect breakages. The second alternative, "Optimal" imagines a perfect model that always correctly predicts which target need to be scheduled. Our model, TRANSPRED, falls between these two extremes of no-improvement to full-improvement. Observe that in comparison to "Only Comprehensive" the variance of breakage detection latency for TRANSPRED is substantially reduced. While, the new model isn't perfect (as shown in the figure), it is a welcome result that the Speculative Cycles reliably finds culprit commits in 37 minutes in the median case (p50), a 65% latency reduction over traditional Comprehensive Cycles that (p50) take 107 minutes.

RQ3*:What aspects of the model design most impact performance? Considered aspects include features selected, super-extreme class imbalance corrections, and training environment.*

> ***Summary***: The feature set used and the training window length had the greatest impact on model performance as measured by ROC AUC. Downsampling is critical for training speed but further downsampling hurts performance while upweighting has negligible impact.

To better understand how model design impacts performance, we ran multiple experiments across the different axes of decision making, including 1) feature selection, 2) training environment. and 3) different ways to correct for the super-extreme class imbalance

1) Feature Selection: Table III shows an improvement in performance when the model is trained on the 'AUGMENTED' feature set vs 'BASE' set, regardless of number of weeks considered for the training window. Critically, we see a lot of signal coming from the added target history features over longer time intervals.

2) Training Environment: Table III indicates our hypothesis was correct – performance starts to degrade as we increase or decrease the training window length from the optimal 3 weeks.

TABLE II: Model performance at different levels of downsampling on the majority class of non-breakages, with and without upweighting the minority positive class of breaking targets within a commit range.

Majority Class Downsample Ratio	Minority Class Upweight Factor	AUC
0.01	**1**	**0.824**
0.005	1	0.814
0.001	1	0.815
0.01	400	0.818
0.005	100	0.822
0.001	40	0.826

TABLE III: Comparing the effects of training window lookback with the base set of features used in our case study of the production pipeline vs an augmented feature set that includes extra commit and target metadata that are not currently used.

Training Data Lookback Window	BASE Features AUC	AUG Features AUC
1 week	0.790	0.791
2 weeks	0.779	0.808
3 weeks	**0.815**	**0.824**
4 weeks	0.808	0.817
5 weeks	0.778	0.782

3) Correcting for Super-Extreme Class Imbalance: Production systems exhibit extremely low Postsubmit breakage rates (40,000:1 in our case). This class imbalance can hinder model training by biasing it towards the majority class. However, due to the asymmetric cost of false positives (unnecessary execution) versus false negatives (late breakage detection), prioritizing catching newly breaking targets at the cost of incorrectly scheduling some healthy targets is acceptable. This lets us experiment with some imbalance-correction techniques without being strictly bound to the true breakage rate, as shown in Table II.

VII. Related Work

This paper explored the problem of reducing the time to discovery of novel test failures in continuous integration. This is a variation of the well studied problem of Test Case Prioritization and Selection [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31] adapted to the CI environment [32], [33], [34], [35], [36], [37], [38], [31], [39], [40], [41], [42], [43], [44]. Our approach is predicated on having highly accurate historical information on what causes breakages in the Google environment. We achieve this through culprit finding and verification [11], [12]. Culprit finding [45], [46], [47], [48], [49], [50], [51], [52], [53] aims to identify the true "bug introducing commits" (BICs) [54], [55], [56], [57], [58] by rerunning tests over the search space of possible versions.

The extensive amount of work on test selection and prioritization makes it difficult to fully summarize outside of a survey [20], [22], [26], [29], [30]. Here are some highlights. Early work in test selection focused on selecting tests while maintaining test suite "adequacy" according to some (usually coverage based) criterion. For instance Bates and Horwitz [16]

define a coverage criterion based on coverage of elements in the Program Dependence Graph (PDG) [59], [60] and Rothermel and Harrold developed a regression selection technique based on the PDG [17]. By 2000 papers such as Elbaum *et al.* [18]'s were looking at prioritizing test cases based statement coverage to speed up test suite execution and evaluating performance based on the now standard *average percentage of faults detected* (APFD) metric. By 2003, the community was starting to look for other sources of data to prioritize and filter test cases. For instance, Leon and Podgurski [19] compared the bug finding efficacy (APFD) of coverage techniques versus techniques that prioritized diversity of basic block execution profiles (that included execution counts). In 2010 Engstrom *et al.* [20] surveyed the state of the test selection and documented 28 techniques.

In 2017 the TAP team at Google began publishing on the problem [7], [8] and separately a partner team had put into production an ML based test filtering system for TAP Presubmit. In contrast to this prior work by TAP, our current work is better grounded because of the verified culprits dataset we created in the last several years (see Section II-C2). For Memon *et al.* [7] TAP did not yet have a robust culprit finding system to precisely identify the bug introducing changes. Instead, that paper attempted to infer what commits might be the culprit. In Leong *et al.* [8] TAP did have a culprit finder but it was not as robust against flakiness and non-determinism as our current system. Our current culprit finder uses the Flake Aware algorithm (FACF) that adaptively deflakes the tests while performing culprit finding. Additionally, we further *verify* all culprit commits by doing additional executions in patterns designed to catch flaky behavior [11], [12]. The FACF paper [11] performed experiments that show the FACF algorithm is statistically significantly more accurate than the "deflaked variant" of the standard "bisect" algorithm used in the Leong paper. Specifically the experiments indicate FACF is $>10\%$ more accurate at identifying flaky breakages than Bisect with 8 additional deflaking runs. By properly deflaking our training and evaluation labels, our model is much less likely to prioritize flaky tests and our evaluation more accurately reflects our desired productivity outcomes.

Meanwhile in 2019, Machalica *et al.* [28] reported on a similar effort at Facebook in 2019 to develop a test selection system for their version of Presubmit testing. Conceptually, the Presubmit selection system at Facebook and the Presubmit selection system at Google have similarities. Both use shallow machine learning models with sets of features not dissimilar to the features presented in this work. Both systems (like this work) take a systematic approach to reducing the effect of flaky failures on developer productivity. The primary contribution of this paper versus this past work is the application and evaluation of selection techniques to the Postsubmit environment. In Postsubmit, the features must be necessarily adjusted as the prediction of whether or not a test will change behavior is not against a single commit but against a sequence of commits. Evaluation also changes: the concern is not just about recall but also latency of breakage detection and time-to-fix.

Recently the wider field (outside of Google and Facebook) has produced compelling work [31], [39], [40], [41], [42], [43], [44]. We will note a few items here. Jin and Servant [31] presented HybridCISave which uses a shallow machine learning based approach to do both fine-grained (individual tests) and coarse grained (CI workflow builds) selection. By combining both selection types both cost savings and safety are improved. The authors evaluate their results using TravisTorrent [61]. This work is a good representative of the general approach of using shallow machine learning for selection outside of the mega-corp environment. Wang *et al.* [43] address the problem of feature selection by using a transformer based language model, GitSense, that automates statistic feature extraction. The authors evaluate their results using the dataset from Chen *et al.* [32]. Zeng *et al.* [44] use mutation testing to better assess the safety of a commercial CI system (YourBase) that skips tests based on inferred dependencies. This use of mutation testing highlights one weakness of traditional safety evaluation, they only use historical faults and may miss novel bugs. By injecting mutation faults a more holistic view of the performance of the model can be obtained. Determining how to apply mutation testing to shallow machine learning approaches that use metadata features (such as ours) is an open problem.

VIII. Conclusion

In this paper, we presented a new scheduling system for Google's main test automation platform (TAP). The new scheduling system, Speculative Cycles, prioritizes finding novel test breakages faster. Speculative Cycles are powered by a new machine learning model, Transition Prediction, that predicts when tests are likely to fail. While the results are not perfect, they indicate that the new system reduces the breakage detection latency by 65% (70 minutes) on average. This reduction in breakage detection time leads to a reduction in developer friction and increased productivity. In terms of model design, the most impactful choice made was total amount of prior history used to train the model. As the training set size went from 1 week to 3 weeks model performance improved. But as the training set size further increased the model performance actually began to degrade. Only high level metadata features were used in this paper. In the future we look forward to experimenting with more complex models.

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

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

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

[6] 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

[7] 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

[8] 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/

[9] 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

[10] K. Wang, D. Rall, G. Tener, V. Gullapalli, X. Huang, and A. Gad, "Smart Build Targets Batching Service at Google," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Madrid, ES: IEEE, May 2021, pp. 160–169. [Online]. Available: https://ieeexplore.ieee.org/document/9401973/

[11] 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.

[12] T. A. D. Henderson, A. Kondareddy, S. Azad, and E. Nickell, "SafeRevert: When Can Breaking Changes be Automatically Reverted?" in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Toronto, ON, Canada: IEEE, May 2024, pp. 395–406. [Online]. Available: https://ieeexplore.ieee.org/document/10638594/

[13] M. Hoang and A. Berding, "Presubmit Rescue: Automatically Ignoring FlakyTest Executions," in *Proceedings of the 1st International Workshop on Flaky Tests*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–2. [Online]. Available: https://dl.acm.org/doi/10.1145/3643656.3643896

[14] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. Gothenburg Sweden: ACM, May 2018, pp. 181–190. [Online]. Available: https://dl.acm.org/doi/10.1145/3183519.3183525

[15] M. Guillame-Bert, S. Bruch, R. Stotz, and J. Pfeifer, "Yggdrasil Decision Forests: A Fast and Extensible Decision Forests Library," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Long Beach CA USA: ACM, Aug. 2023, pp. 4068–4077. [Online]. Available: https://dl.acm.org/doi/10.1145/3580305.3599933

[16] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '93*. Charleston, South Carolina, United States: ACM Press, 1993, pp. 384–396. [Online]. Available: http://portal.acm.org/citation.cfm?doid=158511.158694

[17] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, Apr. 1997. [Online]. Available: http://portal.acm.org/citation.cfm?doid=248233.248262

[18] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. Issta '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 102–112. [Online]. Available: https://doi.org/10.1145/347324.348910

[19] 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/

[20] 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

[21] 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.

[22] 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.

[23] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, and D. Marinov, "Regression Test Selection for Distributed Software Histories," in *Computer Aided Verification*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, A. Biere, and R. Bloem, Eds. Cham: Springer International Publishing, 2014, vol. 8559, pp. 293–309. [Online]. Available: http://link.springer.com/10.1007/978-3-319-08867-9_19

[24] 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.

[25] 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.

[26] 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

[27] 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/

[28] 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/

[29] 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.

[30] 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

[31] X. Jin and F. Servant, "HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–39, Oct. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3576038

[32] B. Chen, L. Chen, C. Zhang, and X. Peng, "BUILDFAST: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 42–53.

[33] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," *Information and Software Technology*, vol. 128, p. 106392, Dec. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584920301579

[34] R. Abdalkareem, S. Mujahid, and E. Shihab, "A Machine Learning Approach to Improve the Detection of CI Skip Commits," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2740–2754, Dec. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8961089/

[35] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which Commits Can Be CI Skipped?" *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 448–463, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8633335/

[36] K. Al-Sabbagh, M. Staron, and R. Hebig, "Predicting build outcomes in continuous integration using textual analysis of source code commits," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. Singapore Singapore: ACM, Nov. 2022, pp. 42–51. [Online]. Available: https://dl.acm.org/doi/10.1145/3558489.3559070

[37] I. Saidani, A. Ouni, and M. W. Mkaouer, "Improving the prediction of continuous integration build failures using deep learning," *Automated Software Engineering*, vol. 29, no. 1, p. 21, May 2022. [Online]. Available: https://link.springer.com/10.1007/s10515-021-00319-5

[38] ——, "Detecting Continuous Integration Skip Commits Using Multi-Objective Evolutionary Search," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4873–4891, Dec. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9622158/

[39] X. Jin, Y. Feng, C. Wang, Y. Liu, Y. Hu, Y. Gao, K. Xia, and L. Guo, "PIPELINEASCODE: A CI/CD Workflow Management System through Configuration Files at ByteDance," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Rovaniemi, Finland: IEEE, Mar. 2024, pp. 1011–1022. [Online]. Available: https://ieeexplore.ieee.org/document/10589850/

[40] B. Liu, H. Zhang, W. Ma, G. Li, S. Li, and H. Shen, "The Why, When, What, and How About Predictive Continuous Integration: A Simulation-Based Investigation," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5223–5249, Dec. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10315109/

[41] Y. Hong, C. Tantithamthavorn, J. Pasuksmit, P. Thongtanunam, A. Friedman, X. Zhao, and A. Krasikov, "Practitioners' Challenges and Perceptions of CI Build Failure Predictions at Atlassian," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Porto de Galinhas Brazil: ACM, Jul. 2024, pp. 370–381. [Online]. Available: https://dl.acm.org/doi/10.1145/3663529.3663856

[42] G. Sun, S. Habchi, and S. McIntosh, "RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 996–1018, Jul. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3643771

[43] G. Wang, Z. Sun, Y. Chen, Y. Zhao, Q. Liang, and D. Hao, "Commit Artifact Preserving Build Prediction," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna Austria: ACM, Sep. 2024, pp. 1236–1248. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680356

[44] Z. Zeng, T. Xiao, M. Lamothe, H. Hata, and S. Mcintosh, "A Mutation-Guided Assessment of Acceleration Approaches for Continuous Integration: An Empirical Study of YourBase," in *Proceedings of the 21st International Conference on Mining Software Repositories*. Lisbon Portugal: ACM, Apr. 2024, pp. 556–568. [Online]. Available: https://dl.acm.org/doi/10.1145/3643991.3644914

[45] 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

[46] 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

[47] 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.

[48] 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

[49] 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

[50] 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/

[51] 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

[52] 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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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

[59] S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 146–157, 1988. [Online]. Available: http://portal.acm.org/citation.cfm?doid=73560.73573

[60] A. Podgurski and L. Clarke, "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*. New York, NY, USA: ACM, 1989, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/75308.75328

[61] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent," p. 414195226 Bytes, 2022. [Online]. Available: https://figshare.com/articles/dataset/TravisTorrent/19314170/1