

Regression Test Selection at Scale

Laiba Mehnaz, Jingyuan Wang, Brandon Stewart, Daniel Jaewon Lee, Zhuolin Song,
Randall Parker, and Ilya Kavalero

Google LLC

Mountain View, California, USA 94043

Email: {laibamehnaz, wangjingyuan, brstewart, danjaewonlee,
charlison, randallparker, ilyak}@google.com

Abstract—Core libraries in massive codebases are dependency hot-spots with high code churn and an immense number of reverse dependencies. A small code change in these libraries can impact a vast number of clients, creating a massive blast radius of costly breakages if a bug or regression is submitted. To prevent expensive post-submit breakages, it is not feasible to test global dependencies during the pre-submit stage for each code change because of resource limitations and the desire for minimal developer friction.

In this paper, we propose a novel ranking-based approach that utilizes a hybrid call graph to test global dependencies during the pre-submit stage. Our method leverages the call graph to select a per-commit subset of these global dependencies and to generate inexpensive graph structural features that allow our machine learning model to learn the complex relationships between library code changes and client tests to predict failures. Evaluating our approach on 10 high-impact core libraries with an average test suite size of 220,000, we demonstrate that a fixed budget of 2,000 tests per commit, representing just 0.9% of the total execution cost, effectively catches regressions for 40% of failing commits.

Index Terms—Regression test selection, Continuous integration, Test case prioritization, Large-scale software systems, call graph analysis

I. INTRODUCTION

Continuous Integration (CI) is a software development practice where developers frequently merge code changes into a central repository, triggering automated builds and tests. Within each CI cycle, regression testing is performed to detect any breakages potentially caused by the changed code and ensure stability once the code is deployed. However, as the scale of software increases, the computational cost of testing grows prohibitively. Since executing all tests potentially impacted by the changes is often infeasible, subsets of the most likely test suites are selected. To address this, regression test selection (RTS) is used to identify a relevant subset of tests to execute, while test case prioritization (TCP) ranks these tests to facilitate earlier failure detection.

There is extensive research on regression test selection (RTS) and test case prioritization (TCP) which can be generally categorized into heuristic based and machine learning based approaches. Most heuristics-based studies [1]–[6], [25] have used coverage related features to select the set of tests to run, which in modern software systems can lead up to a prohibitive number of tests to run. One challenge to contend with is that the degree of code churn caused by continuous integration quickly renders data gathered by code instrumen-

tation imprecise or even obsolete [7]. Another popular choice of heuristic has been test execution and failure history. For instance, [7] utilizes execution and failure windows for test selection. Complementing this, [8] leverages test diversity for broader system coverage, while [9] evaluates additional heuristics including transition, author, and affected test counts, pre-submit failures, and shared monorepo directories.

Recent research has increasingly focused on applying machine learning (ML) to regression test selection [10]. Many approaches leverage test specific features such as execution and failure histories, as well as execution duration [12]–[19]. However, failure based metrics can be unreliable in systems prone to test flakiness. To address this, other machine learning based works [14], [16]–[19], [21], [22], [24] also incorporate code change related features and *cross-features* that link tests to the modified code. These include features like commit metadata, file change frequency, and developer experience, alongside structural metrics such as build dependency graph distance and text similarity between file paths. A comprehensive survey by [28] further categorizes machine learning based RTS and TCP solutions into supervised and reinforcement learning, emphasizing the inherent challenges of class imbalance and data drift present in these tasks. These issues arise because failing tests constitute a minute fraction of the total test suite, and historical training data often fail to reflect current software “hotspots” due to the evolving nature of the codebase.

Google manages its continuous integration through the Test Automation Platform (TAP) [29], which operates in two distinct stages: pre-submit and post-submit. Pre-submit testing occurs before code is merged and focuses on local or developer-specified dependencies, whereas post-submit testing evaluates all global dependencies. This latter stage is particularly challenging for core libraries, often referred to as “dependency hot-spots” [30] where a massive blast radius and high code churn result in a prohibitively large set of potentially impacted tests [31]. A rollback’s cost is the lost time of engineers that are blocked from submitting in a broken project. The more reverse dependencies a submitted breakage has, the more engineering time is lost. A rollback also becomes increasingly difficult to execute the longer a culprit change remains in the codebase: a feature that it introduced could be adopted by users who would then be broken by the rollback. Given the expense of post-submit testing, it is critical to

maximize failure detection during the TAP pre-submit phase.

While machine learning has been applied to post-submit test selection [19], we propose a machine learning driven approach for the pre-submit phase to recommend global dependencies. This is necessitated by the fact that prior pre-submit methods have been either too restrictive, allowing downstream breakages to bypass testing, or too broad, making it difficult to integrate changes due to high testing overhead.

Our methodology leverages call graphs [32] for both test selection and the generation of features that capture structural relationships between changes and tests. Building on the work of [26], [27], [33], we derive several graph based features to help the model distinguish failing tests from a prohibitively large candidate set. Our results demonstrate that within an extremely restrictive budget of 2,000 tests (representing the top 0.9% of potentially impacted tests), training a model on call graph derived features identifies potentially breaking tests for 40% of the failing commits.

Our contributions in this paper are:

- A machine learning approach for pre-submit test selection specifically designed for high-impact core libraries with very high code churn and an immense number of reverse dependencies
- The creation of call graph based features that capture the topological relationship between code changes and downstream tests. We show that these signals help the model learn to rank breakages effectively even with a strict budget.
- A comprehensive study utilizing 10 months of historical breakage data across 10 high-impact core libraries. We demonstrate that within this budget, our model identifies potential breakages for 40% of failing commits.

II. CALL GRAPH

To address the challenge of selecting a highly relevant yet small subset of tests from a vast number of global dependencies, especially during pre-submit, we leverage a deep understanding of code structure through a hybrid call graph analysis. This approach fuses high-precision static analysis with execution-based dynamic profiling and coverage data, balancing the broad reach of source-code dependencies with the runtime precision required to minimize false positives. This contrasts with traditional dependency analysis, which often over-approximates affected¹ tests, and some heuristic or ML features that may lack fine-grained resolution.

A. Static call graph

The static call graph is foundational, constructed from symbol and cross-reference data indexed by Kythe [23], an open-source ecosystem for semantic code analysis, covering most existing source code within the repository. For Python, a custom Kythe-compatible indexer utilizing an Abstract Syntax Tree [34] parser generates the necessary semantic information. The raw graph data from the indexer is processed

to transform these edges into a structured representation of symbols and various connection types, including direct calls, membership, inheritance, decorator usage, and aliases. An analogous pipeline is employed for C++ analysis, utilizing a Kythe-compatible indexer. The volume of raw Kythe data generated for C++ is significantly larger, approximately an order of magnitude greater than that for Python, reflecting the language’s complexity and compilation model. This symbol data for both languages crucially includes cross-references, such as C++ function calls from Python, forming a graph traversable from any code change to all potentially affected functions, enabling a finer test selection than reverse build dependencies. The pipelines generating and processing this graph data from Kythe run daily to ensure the call graph reflects recent code changes. Figure 1 provides the visualization of the static call graph.

This static call graph analysis is specialized into two distinct components tailored for the monorepo environment, where all project code resides in a single, unified repository and project boundaries are defined by directory structures rather than separate versioned repositories. The first component is an internal call graph, which focuses exclusively on the relationships and dependencies within the source code directories of the specific library under consideration. This graph is intentionally scoped to exclude call edges originating from external client code—projects maintained by other teams elsewhere in the same repository—as well as dependencies on lower-level fundamental libraries. Such boundary-setting is crucial for managing the size and computational complexity of graph traversal when performing localized analysis within the library’s own codebase.

To address the critical need of preventing breakages in external projects, the second component focuses on constructing an Entry-Point-to-Test (ETT) map. This map links unit tests across the entire codebase back to the specific API entry points of the target library they ultimately depend on. Built using the same underlying Kythe [23] cross-references, this process initiates from the library’s public API entry points. A multi-hop graph traversal algorithm explores outwards into the vast client codebase, iteratively identifying symbols and unit tests that directly or transitively call these entry points, up to a predefined search depth. The construction of this ETT map especially given the density of the call graph and the sheer volume of C++ Kythe data, necessitates a scalable data processing infrastructure and several key optimizations:

- **Efficient Data Handling:** Employing distributed file systems [35] and distributed SQL query engines [36] to manage and query cached subsets of the Kythe graph edges.
- **Orchestrated Execution:** Utilizing workflow management systems [37] to structure the complex multi-hop analysis.
- **Targeted Edge Selection:** Pre-filtering Kythe data to cache only the edge types most relevant to call dependencies (e.g., references and calls), minimizing the working set size for each hop.

¹Affected tests are defined as the subset of the entire test suite that has a direct or transitive dependency on the modified source files.

- **Identifier Compression:** Reducing data size during large-scale joins and graph operations by using techniques like fingerprinting node identifiers.
- **Advanced Traversal Logic:** Implementing graph traversal algorithms, often within SQL, that incorporate on-the-go cycle detection and pruning to handle recursive calls and improve efficiency.

The intermediate paths found during the traversal are not stored; the final output is a persisted bipartite graph. The resulting ETT maps about 10 million unit tests to about 500 thousand library source code symbols. To facilitate rapid lookups, essential for time-sensitive applications like pre-submit test selection, the ETT map is compressed through perfect hashing, and loaded into memory in an unsharded manner.

While the static call graph offers a significant improvement in granularity for unit test selection compared to reverse build dependencies, its precision is still limited. Static call graphs often create dependencies between unit tests and modified functions that never happen in practice. Consider, for example, a scenario where function F_1 conditionally calls function F_2 based on some boolean parameter. The static call graph will indicate a potential dependency path from F_1 to F_2 . However, if a particular unit test executing F_1 always supplies a boolean value that prevents the call to F_2 , this static edge is never taken. This issue is magnified when API requests are routed through orchestrator functions that statically connect to many handlers. This orchestrator function effectively connects any API caller to any API implementation, greatly reducing the precision of the static call graph.

B. Dynamic call graph

The dynamic call graphs were created for two reasons: (1) to address the extraneous connectivity of the static call graph and (2) to add connectivity between edges that cannot be easily discovered through static analysis. A typical example of (2) is

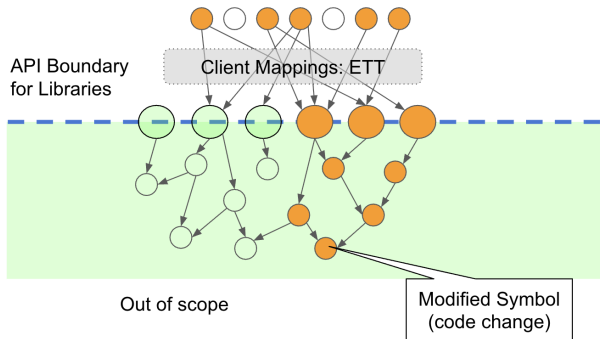


Fig. 1. Static call graph and ETT Mapping Architecture. Constructed from Kythe-indexed data, nodes represent symbols (functions, classes, tests) and directed arrows indicate semantic dependencies (calls, inheritance, etc.). The API Boundary (dashed blue line) separates internal library symbols (green shaded area) from external client code. The graph tracks the transitive impact from a Modified Symbol through the internal call graph to external unit tests via the Entry-Point-to-Test (ETT) mapping. White circles represent “Out of scope” nodes filtered to optimize traversal performance.

a callback passed into the library that calls other API functions. So the dynamic call graph represents the actual call paths taken by a unit test.

Since different unit tests can take different paths, there is technically one dynamic call graph per unit test. This is accurately represented by an extension of the ETT that directly connects each unit test to all the symbols that could affect it within the target API. While the bipartite graph from unit tests to symbols was feasible to store and use, it would be impractical to maintain one complete dynamic call graph per unit test. So intermediate edges of runtime path segments traversed by any unit test are instead added to a single internal dynamic call graph, which augments the internal static call graph. Figure 2 provides the visualization of the dynamic call graph.

Python provides a built-in API (`sys.setprofile`) to facilitate the profiling of every function call. All unit tests under consideration are profiled and the collected call stacks are disassembled into path segments added to the internal dynamic call graph. And the ETT is augmented with a connection from each unit test to all the symbols within the target API visited during the test’s execution.

C++ lacks a built-in API like `sys.setprofile`, and readily available tools rely on sampling-based profiling of running binaries. Sampling is insufficient as it will often miss ephemeral yet critical code path such as initialization routines. Compiler-based instrumentation techniques like Clang’s SanitizerCoverage (Sancov) [38] could potentially offer more complete function call tracing, but would require significant implementation effort to integrate with other tooling.

So for C++, we use code coverage collected from C++ unit test executions. Coverage data is available through standard build and test tooling (e.g., bazel [39] coverage), and indicates which lines of source code are executed by a given test. C++ test execution changes based on build configuration, so each unit test was considered multiple times depending on how many build configurations it had. The resulting data maps each unit test to the lines of source code that could be affected. But source lines are highly susceptible to churn, so the coverage mappings are immediately mapped to function symbols using each function’s current start and end line numbers, queried from the Kythe [23] graph. This enables further augmentation of the ETT similar to Python runtime profiling. However, coverage data does not provide edge data between symbols unlike the call stacks from runtime profiling. So coverage data cannot be used to augment the internal dynamic call graph.

Note that collecting dynamic call graph data requires actually running the unit tests, which is significantly more expensive than parsing library source code to construct the static call graph. For this reason, dynamic call graph data is updated only once per week, introducing staleness. The static call graph is also subject to staleness as it is updated once daily. Staleness could potentially limit the effectiveness of the call graph as critical missing connections cause breakages to be missed. But while source code lines change frequently, function call edges change less frequently. Empirically, for

our target APIs, staleness was not a significant issue. Of the breaking commits that we did not catch, fewer than 2% of these were due to stale connection data.

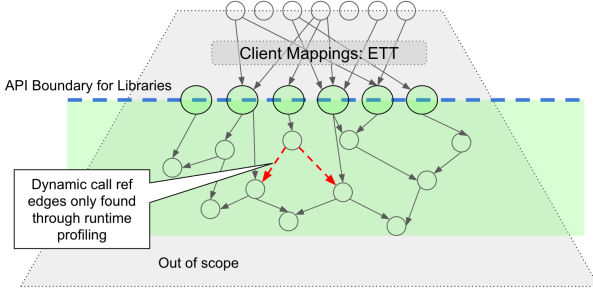


Fig. 2. Dynamic call graph Augmentation. This figure illustrates the enhancement of the static graph using runtime profiling to address extraneous connectivity and discover hidden edges. Red dashed arrows represent dynamic call reference edges, such as callbacks, found only through runtime profiling. While the API Boundary (blue dashed line) remains, the Client Mappings (ETT) are extended to link unit tests directly to any internal library symbol exercised during execution. This enables more precise test selection, allowing for a “dynamic ETT veto” where tests are excluded if profiling shows they never exercise a specific modified symbol.

C. File and Build Mappings

While call graph analysis (both static and dynamic) provides fine-grained insights into function-level dependencies, it has inherent limitations. Notably, many breaking changes do not involve modifications to function bodies. Changes to file-level constants, C++ macros, or, critically, build system rules (e.g., BUILD files) are often invisible to a call graph analysis focused on Kythe function symbols. Our analysis indicates that a substantial portion of regressions are build failures, which are not guaranteed to be detected by call graph-based methods, as the broken test might not call any modified function. To address these gaps and enhance test selection, particularly for build-related breakages and non-function code changes, we supplement the call graph-based approach with file-level and build-level dependency mappings.

First, to capture changes within source files that don’t alter function-level call flows, we utilize the previously generated function-level coverage data. These data, mapping tests to functions they execute, are aggregated to a file level by grouping by source file and unit test. This creates a file-to-test mapping, indicating which tests execute any code within a given file. This allows us to connect non-function code changes (like constants or macros within a source file) to tests that exercise code in that same file, providing a link where the function call graph would find none.

Second, to address breakages stemming from build system modifications and indirect compilation dependencies, we incorporate build dependency graph analysis. The build graph captures how targets depend on each other for compilation and linking. Changes to build rules, `#include` statements in C++, or even function signatures can break targets without a direct call relationship being present in the executed paths.

A common scenario involves a unit test calling function F_1 , which resides in the same build unit as function F_2 . If a change is made to the signature of function F_3 , upon which F_2 depends, the entire build unit may fail to compile, causing the test for F_1 to break, even if F_1 never calls F_2 or F_3 .

The test targets identified through file-level coverage and build dependencies significantly expand the pool of candidates beyond those found by call graph traversal alone. Given that these file and build mappings are more coarse-grained than call graphs and may include more false positives, we employ a tiered strategy for test selection. Tests identified by the call graph are prioritized. If the test execution budget (e.g., 2000 tests) is not exhausted, slots are filled with tests identified through file-level coverage, and then by build dependency mappings, prioritized by build hops. This approach ensures that we leverage the precision of the call graph while using the broader scope of file and build dependencies to catch breakages that would otherwise be missed, especially in cases where the call graph identifies few or no affected tests.

III. TEST SELECTION

A. Overview

The objective of our approach is to select a set of potentially failing tests $\mathcal{T}$ for a given commit $\mathcal{C}$ in the pre-submit stage. We formulate this as a learning-to-rank task, where $\mathcal{C}$ serves as the query and the candidate tests represent the items to be ranked and returned in a response to the query. The model’s goal is to rank the tests such that those with the highest probability of failure given the commit’s changes are positioned at the top of the ranked list.

Formally, we define a commit $\mathcal{C}$ as a set of source code symbols (methods or classes) that were modified in it. We only include symbols that were either updated or deleted in the set of modified symbols².

$$\mathcal{C} = \{s_1, s_2, \dots, s_m\}$$

We then use the global call graph G^3 as defined in Section II to identify a set of *affected* tests $\mathcal{T}_C = \{t_1, t_2, \dots, t_n\}$ that are reachably connected to the set of modified symbols of a given commit. For each commit-test pair $(\mathcal{C}, t_i)$, we construct a feature vector $\mathbf{x}_i$ which is a combination of call graph related features, test related features, change related features that characterize the nature of the commit, and build dependency graph derived features. Each of the four categories of the features are described in detail in section III-B.

B. Per Test Features

For each test we construct four types of features that are concatenated and input to the ranker. The majority of these are calculated at query time, but leverage data cached structures that are computed daily. The call graph features jointly depend on the commit and the test and are computed by aggregating

²New symbols are excluded as they are absent from pre-commit call graphs.

³These call graphs are extremely large as they can have millions of nodes and over tens of millions of edges.

symbol-test pair features. The test related features are constant for all commits. The change related features are constant for all tests but can act as conditioning information for the ranker. Lastly the build dependency features are also joined to commit-test features.

1) *Call graph Features*: To quantify the propagation of changes from the source code to the test suite, we derive a set of structural and reachability metrics from the global call graph G . For each test t_i in the test suite, we evaluate its relationship to the change $\mathcal{C}$ through the following features:

- **Shortest Path Distance**: The minimum number of edges separating any $s_j \in \mathcal{C}$ from the test entry point t_i . A smaller distance suggests a more direct and higher-probability impact.
- **Tree Size**: The total number of nodes in the subgraph reachable from s_j , which serves as a proxy for the broadness of potential side effects induced by the change.
- **Max Outdegree**: The maximum number of outgoing edges from any node along the call path between the test entry point t_i and the modified symbol s_j . This identifies whether the test interacts with the modification through high fan-out “hubs”.
- **Static vs. Runtime Paths**: We distinguish between theoretical paths existing in the static call graph and *active* paths observed during previous runtime executions. This allows the model to prioritize observed execution flows over dead code paths.
- **Path Enumeration**: The total count of distinct call sequences leading from the change symbol set to t_i , providing a measure of the coupling density between the modified code and the test.
- **Interface Sensitivity**: Number of distinct call paths in G between the test entry point t_i and a modified symbol s_j , specifically when s_j is identified as an interface symbol (public API signature or exported method). Changes to these symbols are statistically stronger indicators of potential breakage.
- **Affectedness Score**: A normalized metric representing the degree of intersection between the test t_i ’s historical call paths and all tests’ call paths that reach the current modified symbol set $\mathcal{C}$.
- **Dynamic Call Intensity**: This metric captures the frequency with which the modified symbol s_j is invoked during the execution of test t_i . By measuring the total number of calls, the model can distinguish between incidental reachability and core dependencies; a symbol that is frequently invoked by a test represents a higher risk of causing a regression if modified.
- **Relative Call Weight**: This feature represents the proportion of t_i ’s total execution effort (total callees) dedicated specifically to symbol s_j . This normalized metric identifies the relative importance of a modification to a test’s execution profile. A high proportionality score suggests that the test is critically coupled to the modified symbol, making it a highly sensitive indicator for that specific

change-set.

Since a commit $\mathcal{C}$ typically comprises multiple modified symbols, a raw metric is initially computed for every individual symbol-test pair (s_j, t_i) . To convert these into a single feature vector $\mathbf{x}_i$ for test t_i , we aggregate the values for every symbol s_j using descriptive statistics.

Specifically, for every base metric f , we compute a triplet of values representing the *minimum*, *median*, and *maximum* across the set of modified symbols:

$$\mathbf{x}_{i,f} = \left[\min_{s_j \in \mathcal{C}} f(s_j, t_i), \text{median}_{s_j \in \mathcal{C}} f(s_j, t_i), \max_{s_j \in \mathcal{C}} f(s_j, t_i) \right]$$

This aggregation strategy ensures that our model captures the full distribution of change propagation. While the *median* provides a measure of the central tendency that is robust against outliers, the *minimum* and *maximum* values are critical for capturing extreme cases. For instance, the *minimum* shortest distance identifies the modification most “proximal” to the test, while the *maximum* shortest distance identifies the most indirect or distant interaction, capturing potential regressions stemming from changes to deep-level utility functions or transitive dependencies. By concatenating these three statistics, we provide the learning algorithm with a representation of the commit’s structural influence on each test case.

2) *Test related features*: To account for the individual characteristics of the test suite, we include several test related features validated by previous literature [1], [2], [7]–[9], [11]–[21], [24]. These include failure and execution histories, flakiness indicators, and execution profiles characterized by their size and average runtime. These features help the model understand the relative stability and resource consumption of the test.

3) *Change related features*: These features describe the commit $\mathcal{C}$ itself independent of the test suite. We include metrics like the *number of modified files and symbols*, *number of reviewers assigned to the commit*, *number of comments exchanged between the author and the reviewers*, and *time elapsed between the creation and submission of the commit*. We include these features since prior studies [9], [14], [16]–[19], [21], [24] have shown that they are effective indicators of code complexity.

4) *Build dependency related features*: We also include features derived from the build dependency graph. Specifically, we measure the *dependency distance* as the number of build hops from our different client libraries. Prior research [14], [17]–[19], [40] has demonstrated that a test’s position in the build hierarchy is a strong indicator of its sensitivity to upstream changes.

IV. EMPIRICAL EVALUATION

A. Evaluating model performance

Test Selection is commonly evaluated using metrics such as APFD, NAPFD, and NRPA [41], [42]. While these provide valuable benchmarks for general ranking quality, we utilize a more straightforward metric that is better suited to the practical requirements of our use case. To maintain simplicity and direct

alignment with developer needs, we focus on the *Catch Rate* relative to a fixed *Selection Budget*.

- *Selection Budget (K)*: This represents the allowable computational cost, defined as the percentage of total affected tests scheduled for execution. In our setting, we target a budget of 2,000 tests which is approximately 0.9% of the test suite.

Since our service adds tests on top of existing testing infrastructure present during pre-submit, and runs multiple times for each commit, of which there are on average 444 each day, 2,000 tests is the maximum number of tests that we can add before incurring a non-trivial cost for developers.

- *Catch Rate ($\text{Recall}@K$)*: This reflects the percentage of breaking commits identified by the model by catching at least one failing test within the top K ranked items.

Formally, for a set of commits $\mathcal{C}$ and a set of failing tests T_{fail} , the Catch Rate at budget K is defined as:

$$\text{CatchRate}@K = \frac{\sum_{c \in \mathcal{C}} \mathbb{1}[(\text{Top}_K(c) \cap T_{\text{fail}}) \neq \emptyset]}{|\mathcal{C}|} \quad (1)$$

It is important to clarify that our system is designed to recommend potentially failing tests for a given commit, which are subsequently executed and validated by the underlying testing infrastructure.

B. Dataset

We constructed our dataset using 10 months of historical breakage data from high-impact core libraries. These libraries were selected based on their extensive downstream dependencies and high frequency of rollbacks. Ground truth labels were obtained from a culprit finder search based on the Flake Aware Culprit Finding algorithm [43] that conclusively identifies the specific commit responsible for each test failure, ensuring high label reliability. Accordingly, our dataset focuses exclusively on commits associated with confirmed breakages, excluding the hundreds of stable commits that typically occur for every single breaking change.

Following the call graph methodology described in Section II, we generated candidate test suites for 1,428 commits. These suites achieved an 86% recall in capturing actual failing tests. The remaining 14% of failures are primarily attributed to the inherent challenges of static analysis in C/C++ code, which constitutes the majority of our dataset. Table I shows the distribution of programming languages across the commits in our dataset, where *others* corresponds to commits that only modified build or configuration files. The resulting dataset is exceptionally large and sparse, comprising approximately 338 million commit-test pairs. On average, each commit requires ranking 220,000 candidate tests. Notably, failure events account for only 0.004% of the pairs, presenting an extreme class imbalance.

To prevent temporal data leakage, we employed a chronological split rather than random sampling, structuring our data to simulate a realistic production deployment where the model

TABLE I
DISTRIBUTION OF PROGRAMMING LANGUAGES PRESENT IN THE DATASET.

Programming Language	% of commits
C++ only	52%
Python only	34%
C++ and Python	8.4%
Others	5.6%

is trained on historical data and evaluated on future, unseen events. We partitioned the chronologically sorted commits into three disjoint sets: $\mathcal{D}_{\text{train}}$ (the first 70%), $\mathcal{D}_{\text{val}}$ (the next 15%), and $\mathcal{D}_{\text{test}}$ (the final 15%). All results reported in Section V are evaluated on the held-out $\mathcal{D}_{\text{test}}$ set to simulate real-world deployment performance.

C. Training Setup

a) *Feature Pre-processing*: To account for the varying scales of features across different commits, we perform commit-centric normalization. For each commit $\mathcal{C}$, we apply a three-stage transformation to the set of feature vectors corresponding to the affected tests: (i) a min-shift to ensure non-negativity ($x_{\text{shift}} = x - \min(x)$), (ii) a log-transformation ($\log_{10}(1 + x_{\text{shift}})$) to stabilize variance, and (iii) Z-score standardization. Most of our features follow a Poisson distribution and hyperparameter tuning confirmed that the log-transformation consistently outperformed other scaling methods such as Quantile, Anscombe, and Min-Max normalization by mitigating the impact of heavy-tailed distributions common in our features.

b) *Model Architecture and Training*: We explored several models, including decision trees, LightGBM [44], and various deep and shallow Dense Neural Networks (DNN). The DNNs were trained using the Adam [45] optimizer with weight decay, and learning rate scheduler to reduce the learning rate upon validation loss stagnation.

c) *Ranking Loss Function*: Since we are training with a ranking loss, training data is organized into batches of shape (*batch size, list size, feature size*). During training, we use a sampling strategy that populates each list with a mixture of failing and passing tests to ensure there is a meaningful loss for the model to learn from.

After extensive hyperparameter tuning, we concluded that a hybrid ranking and classification loss function works the best for our dataset:

$$\mathcal{L}_{\text{total}} = \alpha \mathcal{L}_{\text{pairwise}} + \beta \mathcal{L}_{\text{BCE}} \quad (2)$$

where $\mathcal{L}_{\text{pairwise}}$ optimizes the relative ordering of tests and $\mathcal{L}_{\text{BCE}}$ (Binary Cross-Entropy) provides a point-wise classification signal to sharpen the distinction between failing and passing targets. The weighting coefficients α and β were determined through hyperparameter tuning to be 0.1 and 0.9. RAX [46] library was used to implement the ranking loss functions.

d) *Hyperparameter Tuning*: We performed an exhaustive grid search over the following parameters: list size, batch size, initial learning rate, hidden layer dimensions, feature scaling,

ranking losses and the loss weights α and β . Model performance was evaluated using the *Catch Rate@K* for $K=2000$ metric defined in Section IV-A.

V. RESULTS

In this section, we discuss the results of the experiments we performed to find an optimal model to rank potentially failing tests and we focus on answering three research questions.

RQ1 *What is the performance of the model trained with call graph features in terms of percentage of breakages caught in comparison to baseline test selection approaches?*

Utilizing the previously defined Catch Rate metric and production budget of 2,000 tests (0.9% of affected tests), Figure 3 shows how the recall of breakages changes as the percentage of tests selected to run during pre-submit changes, comparing the performance against two baselines: one that involves random test selection, and another that ranks tests using the values of a feature *that measures the number of shared APIs between the commit and the test*. We use this feature as a heuristic to rank tests, as higher the number of shared APIs between a commit and a test, higher are the chances that the test breaks because of some change.

Our model outperforms consistently both the *random* baseline and the *heuristic* baseline, indicating that the model is learning useful information from the call graph features without using any content of the commit or the test. At our production budget of 0.9% of all affected tests, our model achieves a recall of 40%. It is also steeper compared to the other two baselines, highlighting further the ability of the call graph features in identifying potential breakages. By executing 10% of the test suite, the model captures approximately 78% of the total breakages. In contrast, to achieve this same 78% recall, the *heuristic* baseline requires approximately 45% of the test suite to be run, while random selection requires nearly 60%.

Furthermore, the model reaches near-total breakage detection much faster than the baselines. It achieves a 95% recall at approximately 50% of the test suite. Whereas the baselines achieve a maximum of 88% recall. These results suggest that the model trained on call graph features is highly effective at catching breakages, offering significant potential for reducing post-submit breakages and rollbacks. Most important unit tests are configured for TAP post-submit, so any failures missed by the model are still caught after submission with some delay.

RQ2 *Which model design decisions most significantly impact performance, specifically regarding feature selection and model selection?*

To evaluate the relative contribution of different feature categories, we conducted an ablation study by training our model on three distinct feature subsets. The results, illustrated in Figure 5, plot the *Catch Rate* against the percentage of tests selected (*Test Suite Size*). This analysis allows us to isolate

the impact of call graph features in comparison to features mentioned in sections III-B2, III-B3, III-B4 that we refer to as non-call graph features in this section.

The analysis of the ablation results reveals the following key insights:

While traditional meta features (test related features, change related features, and build dependency graph related features) have proven effective in standard TCP/RTS tasks [1], [2], [7], [11]–[20] they may encounter unique challenges when applied to the specific complexities of dependency hot-spots and large scale core libraries as present in our codebase.

Figure 5 shows a significant performance gap between the *Non-call graph features* and *Call-graph Features*. Utilizing only test related, change related, and build dependency graph related features, the model may face challenges in capturing sufficient signal to effectively prioritize failing tests. With call graph features the model achieves a steep increase in *Catch Rate* at low budgets, capturing nearly 30% of failures within the first 1,000 tests, and 40% of failures within the 2,000 tests.

Traditional meta-features (III-B2, III-B3, III-B4), can help characterize the general riskiness of a commit but fail to model the specific interaction between a code change and a downstream test. Call graph derived features help us model this interaction in a cost-effective way by isolating the specific propagation of a change through the system. Historical failure data which are often highly predictive of future failures by signaling to the model the healthiness or the reliability of a test, is often sparse for core libraries. A specific downstream test might not have failed for months, rendering history based features uninformative for that specific CI cycle. However, a structural change in a core library (an interface modification) creates a new failure vector regardless of the test’s history. Our call graph related features utilize the *current state* of the codebase to identify risky paths, making them more resilient to the cold-start problem inherent in newly added tests or long-stable modules.

In massive codebases such as the one used in our evaluation, meta-features (features mentioned in sections III-B2, III-B3, III-B4) alone can sometimes cause many tests to appear identical to the model. Call graph features allow the model to differentiate between these tests by quantifying their relationship to the change. This nuance helps our model to operate effectively within the highly restrictive 0.9% selection budget.

During the model selection phase, we compared multiple modeling techniques such as decision trees, LightGBM, and deep and shallow dense neural networks (DNN). We also experimented with models using an attention layer to allow the network to evaluate each test in relation to others in the same commit. Our experiments revealed that performance remained largely comparable across these models except the attention based models. Attention based models underperformed compared to simpler models. Since a breakage in one test does not influence the state of another test, the inter-test context provided by the attention layer likely introduced ranking noise rather than useful signal. We deployed a DNN

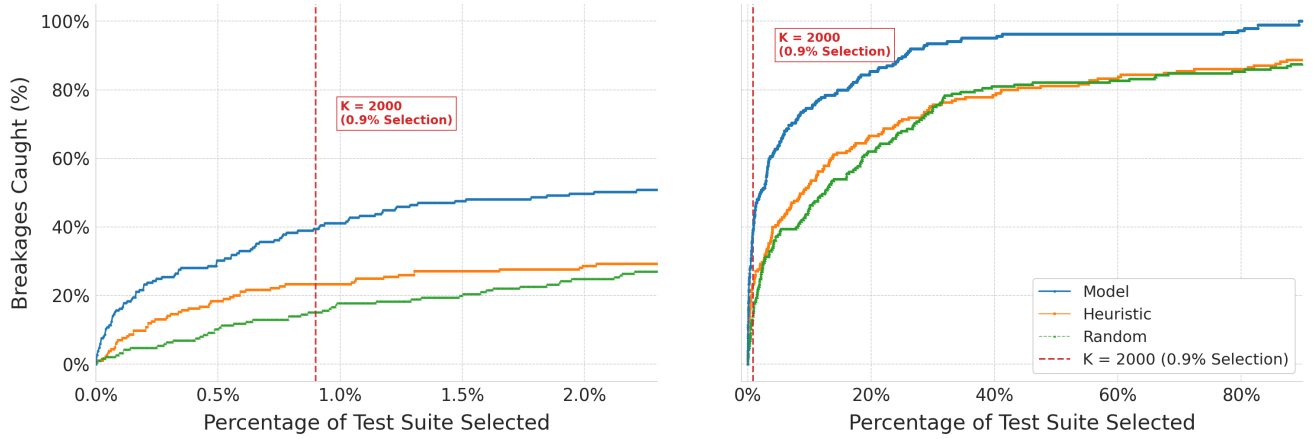


Fig. 3. Catch Rate as a function of the Selection Budget (K). Our model ($AUC=0.8057$) outperforms heuristic ($AUC=0.6646$) and random ($AUC=0.6511$) baselines, achieving 40% recall at production threshold of 2000 tests. The left plot magnifies the critical low budget region (0%–2.5%).

model with a single hidden layer of size 32. This model demonstrated the highest degree of consistency throughout months of evaluation, ensuring reliable performance in our dynamic production environment.

RQ3 *Our dataset consists of commits that cause very few breakages and also those that cause a large number of breakages. What is the performance of our model with respect to both sets? Is it significantly better on one set than the other?*

Figure 4 shows the distribution of the number of breakages caused by commits across our dataset. Our dataset is heavily right-skewed, i.e., the vast majority of commits cause few breakages. We split our dataset into two sets containing “high-impact”⁴ commits (those causing many breakages) and “low-impact”⁵ commits (those causing few breakages). Our findings indicate a significant performance difference between the two sets.

Using a threshold of 15 broken tests (B) to distinguish between the two sets, we observed the following:

- **High-Impact** ($B > 15$): Catch Rate@2000 = 83%
- **Low-Impact** ($B \leq 15$): Catch Rate@2000 = 31%

We use a threshold of 15 as most of our commits have fewer than 15 breakages. The model exhibits a 52-percentage-point lead in performance when predicting “high-impact” commits. This suggests that the model’s effectiveness scales positively with the magnitude of the regression.

From the perspective of a developer, these findings define our model as a robust architectural safety net. When writing high-risk commits such as those that refactor core libraries or modify global interfaces, developers can rely on the model with high confidence (83% catch rate) to flag potential regressions before they reach the integration stage.

However, the lower performance on “low-impact” commits (31%) highlights the inherent limitations of a purely structural



Fig. 4. Model performance (Catch Rate@ $K=2000$) with respect to the number of breakages caused by a breaking change. Each stacked bar represents the percentage of total regressions in the dataset, categorized as “Caught” (correctly recommended within the top 2000 tests) or “Missed.”

representation. Because the model operates solely on graph topology, it remains semantically blind to the specific logic of a change. These findings demonstrate the necessity of augmenting structural features with semantic information. In future work, we intend to explore cost-effective semantic representations that can supplement our call graph features, better positioning the model to identify breakages within low-impact commits.

VI. RELATED WORK

Extensive research in Regression Test Selection (RTS) and Test Case Prioritization (TCP) has been documented in several surveys [28], [47]–[49]. Earlier work primarily relied on coverage-related metrics, dependency-related metrics, and other heuristics, including test execution history, test failure history, and test diversity [1], [2], [7], [8], [11], [20], [25]–[27], [30].

More recently, researchers have applied machine learning to address these challenges. The features typically used in

⁴High impact here refers to the large number of tests broken by a change. It does not refer to the importance of a change.

⁵Low impact here refers to the very small number of tests broken by a change. It does not refer to the importance of a change.

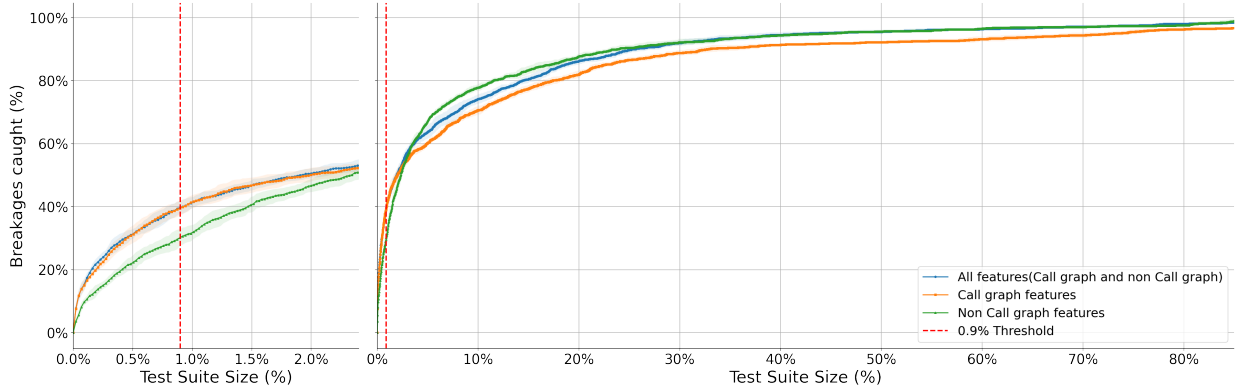


Fig. 5. Ablation results demonstrate that structural call graph features are essential for effective test selection, achieving a 40% catch rate at the 0.9% production budget. Metadata-only models significantly underperform at low budgets. The left plot magnifies the critical low budget region (0%–2.5%).

these ML models can be broadly categorized into change-related, test-related, and cross-features. Change-related features capture characteristics of the modified files (e.g., number and type of files, modification history, and churn) and the authors (e.g., number of distinct authors, developer experience, and reviewer metrics). Test-related features often measure a test’s “usability” [7], which can be quantified by historical failure rates over specific time windows (e.g., 7, 14, or 28 days), execution frequency, time since last execution, and test duration. Finally, cross-features capture the relationship between commits and tests, such as the distance between a test and the changed code in a build dependency graph, or the text similarity (e.g., shared tokens) between the test filepaths and the code change filepaths. Several studies [12]–[19], [21], [24], [50] have trained ML models for RTS and TCP using combinations of these features.

Figures 6 and 7 illustrate the distributions of two of our call graph generated features for failing tests: shortest path distance and max outdegree (detailed in Section III-B). As shown in Figure 6, the most frequently failing tests tend to have a shorter path distance to the changed code, which aligns with findings from previous work [9]. However, Figure 7 reveals that these failing tests also exhibit a very large max outdegree value. A high max outdegree indicates that a test is evaluating a modified symbol⁶ in the call graph that is invoked by a large number of other symbols that were acting as a “broadcaster”⁷ node.

Because commonly used features offer limited predictive power in the context of core libraries that are characterized by a massive blast radius and high code churn, we proposed utilizing advanced call graph features to provide more distinguishing signals. Section V provided an empirical evaluation of this approach, demonstrating its effectiveness in enhancing the model’s predictive performance.

⁶A modified symbol is any symbol that was either updated or deleted in the commit.

⁷In a call graph, a broadcaster node is a symbol with an exceptionally high outdegree that serves as a central distribution point for numerous downstream calls.

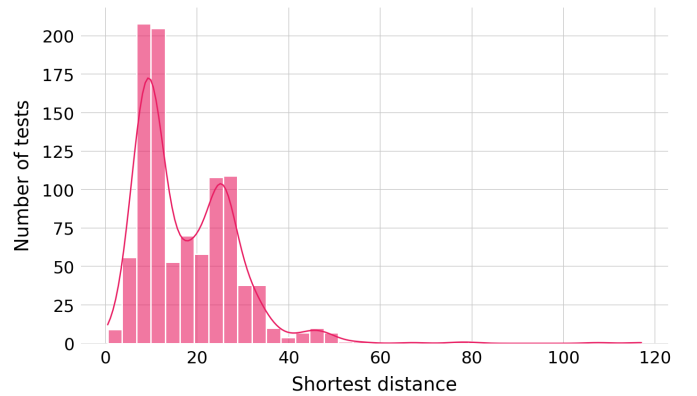


Fig. 6. Frequency distribution of the feature shortest distance for failing tests. Definition of the feature can be found in section III-B.

VII. CHALLENGES IN PRODUCTION

While our experimental results demonstrate a catch rate of 40% at K=2000 in a development setting, live production deployment introduces new challenges that lead to a drop in the catch rate. First, tests requiring hardware accelerators (GPUs or TPUs) are more expensive than CPU tests during pre-submit, so in practice no more than the top 5 such tests are run. Consequently, even when our model correctly identifies a potential breakage in such a test in the top K=2000, the underlying infrastructure may not execute it; this results in no catch being recorded for that commit leading to a lower than predicted catch rate. Second, we face a distributional drift between our training data that is composed of only post-submit breakages, whereas live inference is performed during the pre-submit phase on transient code snapshots some of which may be early drafts of a commit and thus have a different pattern of breakages. We are addressing these challenges in our ongoing and future research.

VIII. CONCLUSION

In large-scale software systems, Continuous Integration (CI) faces significant challenges due to the prohibitive cost of running exhaustive regression suites for every single code change.

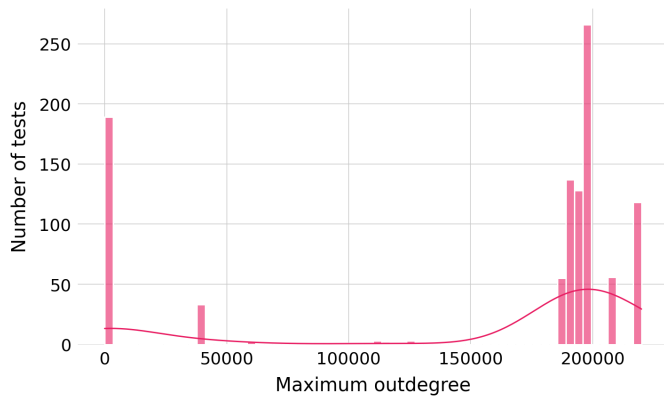


Fig. 7. Frequency distribution of the feature maximum outdegree for failing tests. Definition of the feature can be found in section III-B.

This is especially critical for core libraries in monolithic codebases with massive dependency hot-spots where a single change can affect millions of downstream targets. This paper presented a machine learning based approach to Regression Test Selection and Test Case Prioritization that leverages call graph analysis. We derived structural features such as path distance, tree size, and node outdegree to model the topological relationship between code changes and potential test failures.

We evaluated our approach using a neural network trained on ten months of historical data from a massive monorepo. Our results demonstrate that our approach outperforms traditional heuristic and metadata based baselines. At a highly restrictive pre-submit budget of 0.9% (approximately 2,000 tests), the model achieves a 40% catch rate for breaking commits. Our findings suggest that structural call graph features provide a critical signal for identifying regressions in complex environments where historical failure data is often sparse or imprecise.

ACKNOWLEDGMENT

We sincerely thank Chien-Ming Chen, Jingwen Sun, Rui Huang, Tianchen Liu, Brian Wider, Shashank Viswanadha, Ziyue Wang, Juan Martinez Castellanos, Zhiyan Shen, Junmin Hao, Tristen Allen, Marshall Bockrath, Fiona Lang, Timothy Buzzelli, Jamie Andrews, John Penix, Martin Demello, Shahms King, Sergei Lebedev, and Bogdan Stoica for their valuable support and contributions. We also thank Tim A. D. Henderson for his thoughtful discussions and thorough review. Additionally, we acknowledge the use of Gemini to assist in summarizing source code and documentation for section III, as well as in refining the manuscript's structure and clarity. We have reviewed and verified all content to ensure technical accuracy.

REFERENCES

- [1] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," in *Softw. Test. Verif. Reliab.*, vol. 12, no. 4, 2002, pp. 219–249.
- [2] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, 2002, pp. 97–106.
- [3] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," in *IEEE Transactions on Software Engineering*, vol. 27, no. 10, Oct. 2001, pp. 929–948. doi: 10.1109/32.962562.
- [4] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-Grained test minimization," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 210–221.
- [5] J. Black, E. Melachrinoudis and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," *Proceedings. 26th International Conference on Software Engineering*, Edinburgh, UK, 2004, pp. 106–115.
- [6] M. Harrold, R. Gupta, and M. Soffa, (1990). "A methodology for controlling the size of a test suite." *Proceedings. Conference on Software Maintenance 1990*, pp. 302–310.
- [7] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 235–245.
- [8] A. Haghighatkah, M. Mantyla, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *J. Syst. Softw.*, vol. 146, 2018, pp. 80–98.
- [9] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing transition-based test selection algorithms at Google," *Proc. 41st Int. Conf. Softw. Eng.: Softw. Eng. Pract. (ICSE-SEIP)*, 2019, pp. 101–110.
- [10] J. Son, G. An, J. Hong, and S. Yoo, "Evaluating machine learning-based test case prioritization in the real world: An experiment with SAP HANA," in *Proc. 18th IEEE Int. Conf. Softw. Testing, Verification and Validation (ICST)*, Naples, Italy, Apr. 2025.
- [11] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource-constrained environments," *Proc. 24th Int. Conf. Softw. Eng. (ICSE)*, 2002, pp. 119–129.
- [12] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, 2017, pp. 12–22.
- [13] A. Sharif, D. Marijan, and M. Liaen, "Deeporder: Deep learning for test case prioritization in continuous integration testing," *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2021, pp. 525–534.
- [14] M. Machalica, A. Samykin, M. Porth, and S. Chandra, "Predictive test selection," *Proc. 41st Int. Conf. Softw. Eng.: Softw. Eng. Pract. (ICSE-SEIP)*, 2019, pp. 91–100.
- [15] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 975–980.
- [16] M. A. Khan et al., "ML-based test case prioritization: A research and production perspective in CI environments," *Proc. IEEE Conf. Softw. Test. Verif. Valid. (ICST)*, 2025, pp. 476–486.
- [17] X. Jin and F. Servant, "HybridCISave: A combined build and test selection approach in continuous integration," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, Oct. 2023, pp. 1–39.
- [18] 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>
- [19] A. Kondareddy, S. Azad, A. Singh, and T. A. D. Henderson, "Speculative testing at Google with transition prediction," *Proc. IEEE Conf. Softw. Test. Verif. Valid. (ICST)*, 2025, pp. 510–521.
- [20] D. Marijan, A. Gotlieb and S. Sen, "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study," *IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, 2013, pp. 540–543, doi: 10.1109/ICSM.2013.91.
- [21] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 408–418.
- [22] Jeongki Son, Gabin An, Jingun Hong, and Shin Yoo. 2025. "Evaluating Machine Learning-Based Test Case Prioritization in the Real World: An Experiment with SAP HANA". In 18th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2025, April 2025. Naples, Italy.
- [23] Kythe, "Kythe: Open-source semantic indexer," 2024. [Online]. Available: <https://kythe.io/>
- [24] P. Plyusnin et al., "Targeted test selection approach in continuous integration," *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2025, pp. 1–12.

- [25] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies." *IEEE Trans. Software Eng.*, 28, 2002, pp. 159–182.
- [26] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2016, pp. 583–594, doi: 10.1145/2950290.2950361.
- [27] O. Legunsen, A. Shi and D. Marinov, "STARTS: STATic regression test selection," 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 2017, pp. 949–954, doi: 10.1109/ASE.2017.8115710.
- [28] Y. Zhao, D. Hao, and L. Zhang, "Revisiting machine learning based test case prioritization for continuous integration," *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2023, pp. 232–244.
- [29] J. Micco, "Tools for Continuous Integration at Google Scale," Google NYC, Jun. 2012. [Online]. Available: <https://youtu.be/KH2sB1A6IA>
- [30] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at Google using multi objective regression test optimisation," *Proc. 27th IEEE Int. Conf. Softw. Maint. (ICSM)*, 2011, pp. 203–212.
- [31] Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of Google. Retrieved from <https://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html>
- [32] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, May 1979, pp. 216–226, doi: 10.1109/TSE.1979.234183.
- [33] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2001, pp. 46–53, doi: 10.1145/379605.379661.
- [34] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, in "Compilers: Principles, Techniques, and Tools, 2nd ed". Pearson/Addison Wesley, 2007.
- [35] Apache Hadoop. "HDFS: Hadoop Distributed File System." (2024). [Online]. Available: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>
- [36] Presto Foundation. "Presto: Distributed SQL Query Engine." (2024). [Online]. Available: <https://prestodb.io/>
- [37] Apache Airflow, 2024. [Online]. Available: <https://airflow.apache.org/>
- [38] Clang. "SanitizerCoverage: Code coverage for C++." (2024). [Online]. Available: <https://clang.llvm.org/docs/SanitizerCoverage.html>
- [39] A. Mokhov, N. Mitchell, and S. P. Jones, "Build Systems à la Carte", in *Proc. ACM Program. Lang.*, vol. 2, no. ICFP. ACM, 2018.
- [40] A. M. Memon et al., "Taming Google-Scale Continuous Testing," in *Proc. 2017 IEEE/ACM 39th Int. Conf. Softw. Eng. Practice Track (ICSE-SEIP)*, 2017, pp. 233–242, doi: 10.1109/ICSE-SEIP.2017.16.
- [41] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *IEEE Int. Conference on Software Maintenance*, 1999, pp. 179–188.
- [42] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *IEEE Int. Conf. on Software Maintenance*, 2007, pp. 255–264
- [43] Henderson, T.A., Dorward, B., Nickell, E., Johnston, C., and Kondareddy, A. (2023). Flake Aware Culprit Finding. 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), 362-373.
- [44] G. Ke et al., "LightGBM: A highly efficient gradient boosting decision tree," in *Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 3146–3154.
- [45] Kingma, Diederik P. and Jimmy Ba. "Adam: A Method for Stochastic Optimization." *CoRR* abs/1412.6980 (2014): n. pag.
- [46] R. Jagerman, X. Wang, H. Zhuang, Z. Qin, M. Bendersky, and M. Najork, "Rax: Composable learning-to-rank using JAX," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, 2022, pp. 3051–3060, doi: 10.1145/3534678.3539130.
- [47] E. Engstrom, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Inf. Softw. Technol.*, vol. 52, no. 1, 2010, pp. 14–30.
- [48] Junior, H.D., Araújo, M.A., David, J.M., Braga, R.M., Campos, F., and Ströele, V. (2017). "Test case prioritization: a systematic review and mapping of the literature". *Proceedings of the XXXI Brazilian Symposium on Software Engineering*.
- [49] Y. Singh, A. Kaur, B. Suri, and S. Singhal, "Systematic literature review on regression test prioritization techniques," *Informatica*, vol. 36, no. 4, 2012, pp. 379–408.
- [50] Peter Spragins. 2018. "Efficacy Presubmit". Google Testing Blog. Retrieved from <https://testing.googleblog.com/2018/09/efficacy-presubmit.html>