# *DANTE*: Data-Driven Test Case Selection and Prioritization for Long-Running Test Suites

Simone Reale*, Elisabetta Di Nitto*, Luciano Baresi*, Massimiliano Di Penta†, Giovanni Quattrocchi‡
*Politecnico di Milano, Italy †Università del Sannio, Italy ‡Università degli Studi di Milano, Italy

***Abstract*—As software systems evolve, large test suites increasingly delay developer feedback in Continuous Integration and Delivery (CI/CD). Test Case Selection and Prioritization (TCSP) mitigates this issue, and recent evidence shows that simple heuristics, such as prioritizing recently failed or fast-running tests, often outperform sophisticated machine learning (ML) approaches, which incur high training costs and suffer from distribution shift. However, these simple heuristics remain rigid and fail to leverage the rich historical data generated continuously by CI/CD pipelines. This paper introduces DANTE, a data-driven, yet training-free framework that evolves simple TCSP heuristics into a data-driven selection and prioritization strategy. DANTE processes historical signals through a probabilistic stickiness metric that automatically adapts to project-specific failure and fixing patterns, eliminating the need for manual tuning. This history-aware logic is complemented by a lightweight, commit-aware component that captures the relevance between code changes and test artifacts. We evaluated DANTE on the Long-Running Test Suite (LRTS) dataset, focusing on the Java projects, which constitute the vast majority in LRTS (9 out of 10 in total) and comprise more than 21,000 CI builds with multi-hour test suites. The results show that DANTE consistently outperforms state-of-the-art cost-history cognizant heuristics and ML baselines in both test selection and prioritization, while remaining robust to flaky tests.**

***Index Terms*—test case selection, test case prioritization, software testing, continuous integration, CI/CD**

## I. Introduction

Any software system deemed useful continuously evolves [1], and modern systems even do it daily [2]. As the code grows, so do its accompanying test suites. Large projects now execute thousands of test cases spread across unit, integration, and end-to-end levels [3], using Continuous Integration and Delivery (CI/CD) pipelines [4], [5]. However, running the entire test suite for each build could take hours, stalling developer feedback, and increasing compute costs [6]. Indeed, long builds have been found to be a bad practice in CI/CD and must therefore be avoided/limited [4], [7]. Therefore, a more selective testing strategy is essential to keep up with rapid evolution while retaining confidence in code quality [8].

A growing body of research efforts tackles this problem with *Test Case Selection and Prioritization* (TCSP) techniques [8], [9], [10]. Such approaches, for each new build, choose a subset of test cases likely to uncover faults introduced by the latest changes and execute them in an order that surfaces failures as early as possible. When applied in large industrial settings, TCSP can reduce testing cost without sacrificing defect detection [6]. Previous work on TCSP spans a spectrum of strategies. Early studies rely on heuristics, e.g., prioritizing tests with short execution times or that failed in the most recent builds [11], [6]. Other methods select tests relevant to the files or functions touched by the commit [12], [10]. Recent work frames TCSP as a learning task using supervised models to estimate fault likelihoods or reinforcement-learning agents to minimize testing cost under time budgets [13], [14].

Despite such advances, TCSP remains a challenging task with important gaps. Many advanced [15], [16], [17] TCSP techniques rely on fine-grained coverage data, deep static analysis, or heavy machine learning pipelines. These techniques scale poorly when the suite contains tens of thousands of tests, and each test run may last hours [18]. Feature extraction, model retraining, and data collection quickly become bottlenecks when the tight turnaround expected in modern CI/CD is required. Moreover, much of the available empirical evidence comes from small or synthetic benchmarks whose execution times are measured in minutes [19]. Another challenge occurring when executing tests is in their *flakiness*: some tests pass on one run and fail on the next, even when the code is unchanged. Flakiness arises from sources such as concurrency, external resources, and non-deterministic assertions [20]. If TCSP techniques over-prioritize flaky tests, they might produce useless information for developers, who have to waste time chasing non-reproducible faults. Cheng et al. [18] took a step toward realism by assembling the *Long Running Test Suite* (LRTS) dataset and re-evaluating a broad range of TCSP methods on more than 21,000 CI builds. Their study shows that when test suites take hours to run and failures are both rare and flaky, simple heuristics-such as prioritizing recently failed and fast-running tests-can detect as many faults as relevance- or learning-based methods, but with significantly less overhead. These findings expose a gap in the literature: solutions that effectively exploit richer information than past failures or execution times, yet remain fast enough for highly frequent builds and industrial-scale software projects [21].

In this work, we propose *DANTE* (**Da**ta-drive**n** Priori**t**ization and S**e**lection), a novel TCSP approach that explicitly targets this gap. *DANTE* is designed as a data-driven, training-free approach that preserves the scalability of heuristic-based TCSP while exploiting the rich historical information accumulated in modern CI pipelines. Rather than replacing existing heuristics with heavyweight learning models, *DANTE* systematically evolves them into a data-driven decision mechanism grounded in observed execution behavior.

The core of *DANTE* is a probabilistic interpretation of test execution history. Instead of treating past failures as binary

or frequency-based signals, *DANTE* models the *persistence* of test failures over time through a notion of *stickiness*, which captures how strongly a test remains associated with failures across consecutive builds. This signal is computed using statistically grounded estimators based on Wilson confidence intervals, allowing *DANTE* to automatically adapt to project-specific bug-fixing cycles and failure patterns without manual thresholds or offline training phases.

To complement historical information, *DANTE* incorporates change awareness by relating code changes to test artifacts through textual similarity. This commit-aware component allows *DANTE* to prioritize tests that are both historically informative and contextually relevant to the current change, while avoiding expensive static analysis or coverage instrumentation [19]. Importantly, all signals used by *DANTE* are readily available in standard CI/CD environments. In contrast to learning-based approaches, *DANTE* continuously adapts its selection and prioritization decisions as new builds become available, without requiring any retraining step.

We evaluated *DANTE* through an extensive empirical assessment on the LRTS dataset, focusing on nine large-scale open-source Java projects with test suites whose execution time spans several hours (we did not consider a single project in LRTS to maintain uniformity across the analyzed programming language). Our evaluation assesses both test case selection and prioritization effectiveness using standard metrics and compares *DANTE* against a comprehensive set of state-of-the-art baselines, including cost-history-aware heuristics and learning-based TCSP techniques, along with their cost-cognizant variants. Following recent best practices, we also explicitly account for confounding factors such as flaky and frequently failing tests. The results show that *DANTE* consistently improves fault detection effectiveness while maintaining robustness and scalability in realistic settings.

The remainder of this paper is organized as follows. Section II provides a background on the relevant literature on TCSP. Section III presents *DANTE*. Then, Section IV describes the empirical evaluation design, Section V presents its results, and Section VI discusses the threats to its validity. Section VII concludes the paper.

## II. Related Work

TCSP aims to reduce regression testing costs. Selection focuses on executing only relevant tests (e.g., those that test the code affected by recent changes), while prioritization sorts tests by likelihood of failure so that faults are discovered as early as possible. Such techniques are often used in CI/CD pipelines to reduce unnecessary executions and ensure that the most valuable tests are run first [8], [22], [23].

Research in this area has explored a spectrum of strategies, ranging from lightweight heuristics to sophisticated learning algorithms. *History-based* techniques—e.g., Elsner et al. [24], Najafi et al. [25], and Paterson et al. [26]—rank test cases using past results, typically giving higher priority to tests that have failed in recent builds or have been skipped for many cycles. In contrast, *Information-Retrieval (IR)* methods treat the modified code as a query, and each test or its related artifacts as a document, then order tests according to lexical or semantic similarity. Representations for IR methods range from bag-of-words models—e.g., Bafna et al. [27] and Roberson et al. [28]—to neural code embeddings that capture deeper semantics, see for instance, Saha et al. [29], Magalhães et al. [30], and Yang et al. [31].

*Time-based* methods prioritize test execution time. Some treat it as a tight budget: given a fixed window, they schedule tests to expose faults as soon as possible. For example, Walcott et al. [32] and Hou et al. [33] frame this as an optimization problem that maximizes fault detection under a time cap. Others fold cost directly into the ranking itself: for example, Peng et al. [34] scales the IR similarity by each test's past execution time, so that quick and failure-prone cases are run first. Chen et al. [35] proposes a framework that learns from a project's coverage and time profile the prioritization strategy that produces the best cost-benefit trade-off and automatically selects it for each build.

Several TCSP techniques that utilize *machine learning* (ML) have been proposed [36]. Specifically, supervised models estimate failure probabilities based on features such as code complexity and historical outcomes, and then rank test cases accordingly. Jahan et al. [37] show how neural network-based approaches outperform traditional ML classifiers in multiple projects. Learning-to-rank approaches extend this idea by directly optimizing the test execution order rather than predicting binary outcomes [10]. Unsupervised methods (e.g., clustering) aim to increase diversity by identifying test cases that cover different functionality [38]. Other efforts have explored *learning-to-rank* and *reinforcement learning* formulations of TCSP. Learning-to-rank approaches directly optimize test case execution order by learning relative preferences rather than absolute failure probabilities [10]. Reinforcement learning techniques, instead, model TCSP as a sequential decision-making problem, where an agent iteratively selects tests to maximize long-term rewards under time or cost constraints [14].

Selection and prioritization should also consider test flakiness: the test outcome is not deterministic and may fail regardless of the applied code change. Li et al. [39] presents a method to systematically produce specific test sequences, prioritizing the orders most likely to reveal these unstable behaviors. Similarly, Rahman et al. [40] focuses on diagnosing these flaky tests once they are found.

Overall, existing TCSP approaches either rely on lightweight but rigid heuristic rules or adopt learning-based techniques that introduce training overhead and are sensitive to distribution shifts [18], leaving a gap for methods that can adapt to evolving CI/CD environments while remaining lightweight and robust to noisy signals.

## III. *DANTE*

*DANTE* is a data-driven framework for TCSP, whose peculiar characteristics make it very suitable for CI/CD environments and the presence of long-running test suites.

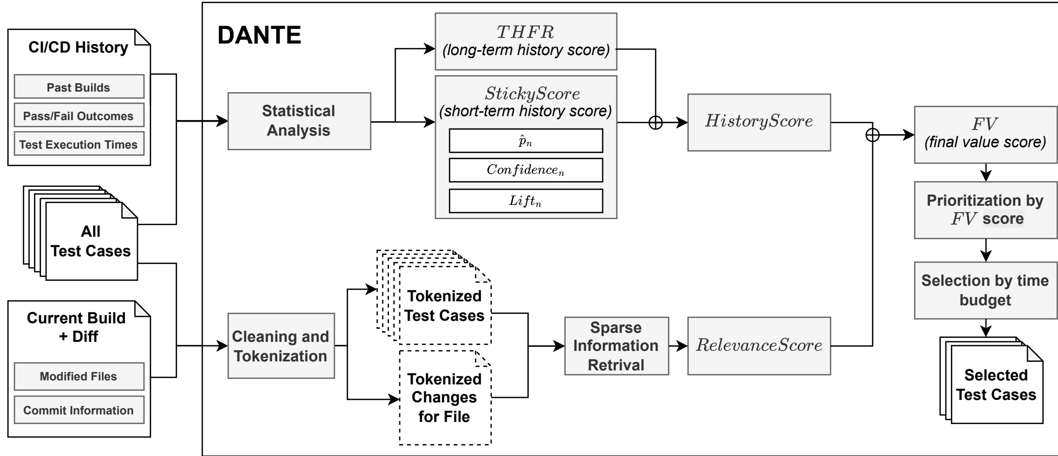


Fig. 1. *DANTE* overview.

Figure 1 provides an overview of our approach. Starting from the CI/CD history and the current build diff, *DANTE* computes two complementary signals: a *history score*, derived from failures observed in previous builds, and a *relevance score*, capturing the relationship between the latest code changes and test artifacts. These signals are combined into a *Final Value* (FV) score, which is used to rank tests; finally, *DANTE* greedily selects the top-ranked tests until the cumulative execution time reaches the given budget.

The *history score* (detailed in Section III-A) leverages execution history to estimate how informative each test is with respect to failure discovery. However, raw failure counts or simple recency rules are often insufficient in practice, as projects differ substantially in how quickly failures are resolved and in how frequently non-deterministic flaky failures occur. For this reason, *DANTE* models historical failure risk through a notion of failure *stickiness*, which captures the persistence of failures in consecutive builds and automatically adapts to project-specific fixing behaviors. The *relevance score* (detailed in Section III-B)incorporates lightweight change awareness by measuring the textual similarity between the code modified in the current build and test artifacts, so that tests that are historically informative but unrelated to the current change are de-prioritized.

**Guiding principles.** The definition of *DANTE* has been guided by a preliminary empirical analysis of the LRTS dataset.[1] In practice, we quantified (i) the reliability and decay of historical failure signals over time, (ii) cross-project variability in failure handling and persistence, and (iii) the temporal stability of failure-prone code areas across build windows, using build-ordered CI logs and package-level failure traces. The analysis highlighted key characteristics of real-world CI/CD environments that directly motivate the architectural choices we made.

First, *there exists no single signal capable of precisely predicting test failures*. Historical failure information, commit-text textual relevance, and execution patterns are all weak and noisy predictors: historical signals suffer from temporal decay and flakiness, while change-based relevance can both under- and over-estimate the actual impact of a commit. Second, *different projects exhibit substantial heterogeneity in how failures are handled and resolved*. The lifetime of failing tests, the speed of bug fixing, and the persistence of failure patterns vary widely across repositories. Third, *failure-prone areas of a codebase are highly volatile over time*: components that fail in one period are often different from those failing in subsequent builds. This non-stationarity undermines approaches based on global historical aggregates and supports the need for commit-aware signals that continuously realign test prioritization with the current evolution of the system.

As a consequence, lightweight heuristics struggle to cope with uncertainty, heterogeneity, and temporal volatility because they rely on fixed rules and static aggregations, while learning-based approaches attempt to address these aspects through trained models but remain sensitive to build-to-build distribution drift, thus requiring continuous retraining, and often degrade when faced with noisy and rapidly evolving CI/CD signals [18]. Instead, *DANTE* addresses these challenges by combining multiple weak signals in a lightweight, training-free framework that adapts automatically to project-specific dynamics while remaining robust to noise and distribution drift in CI/CD environments.

**Terminology.** Before explaining the details of our approach, we introduce some key terms to avoid ambiguities. A *project* is a software development project associated with a repository managed by a version control system (e.g., git). A project's codebase includes modules, classes, and *test classes*, each of which includes one or more test cases, and has a *test class execution time*. A *change* is any addition, deletion, or modification to a code file compared to its previous version. A *build* is an executable project release and is defined as the set of code files needed to run the software, along with the corresponding test cases. The *current build* (CB) is the last

[1]Due to space constraints, the full analysis is omitted; data are available in the replication package (folder `preliminary_analysis`).

one generated. Given a current build, a corresponding *look-back window* is defined as a set of those previous builds that immediately precede it. The *size* of a specific look-back window is defined in terms of the number of builds it contains (look-back windows are a modeling stratagem we adopt to study the past of a current build).

A *failing test class* reveals a fault in the build. Since a test class typically includes multiple test cases, similarly to Cheng et al. [18], we define a test class as failing if at least one of its test cases fails. A *flaky test class* passes or fails non-deterministically when applied to the same (version of) code. *Stickiness* is the tendency of a test class to fail again in future builds if it has failed in recent builds.

### A. Learning from historical data

The effectiveness of a test class varies throughout the development process. Its tendency to fail, and therefore to discover failures, can be a strong indicator of its importance, but this can vary over time. For instance, if a test class had failed multiple times in the past but not recently, we can argue that it is less important for the current build than a test class that has failed frequently recently.

*DANTE* analyzes a corpus of previous builds to characterize the history of test class failures, thus deriving predictive signals about their potential importance. For each test class $TC_i$ we compute:

$$\text{HistoryScore}_i = \underbrace{\text{THFR}_i}_{\text{long-term}} + \underbrace{\text{StickyScore}_i}_{\text{short-term}} \quad (1)$$

where $\text{THFR}_i = \dfrac{\#\text{fails of } TC_i}{\#\text{executions of } TC_i}$ captures the *total historical failure rate* over the entire build history, and $\text{StickyScore}_i$ is formalized in the following and measures how likely $TC_i$ is to fail again after a recent failure.

Let $W_n$ be the look-back window containing the $n$ builds immediately preceding the current build CB. We denote the event "$TC_i$ fails in build $b$." as $\text{fail}(TC_i, b)$. The $\text{StickyScore}_i$ depends on $\hat{p}_n(TC_i)$, i.e., the estimated conditional probability that a failure for $TC_i$ occurs in CB given at least one failure in $W_n$. More formally:

$$\hat{p}_n(TC_i) = \hat{P}\big(\text{fail}(TC_i, \text{CB}) \mid \exists\, b \in W_n : \text{fail}(TC_i, b)\big)$$

Intuitively, a high $\hat{p}_n$ means that the test class is *sticky*: given that it has failed in any of the last $n$ builds, it is more likely to fail in CB compared to a randomly selected test in CB.

To compute $\text{StickyScore}_i$, we compute $\hat{p}_n(TC_i)$ for a set $\mathcal{W}$ of look-back windows of various sizes (i.e., different values of $n$). Moreover, we estimate the confidence associated with this probability, which quantifies the statistical reliability of $\hat{p}_n(TC_i)$ as:

$$\text{Confidence}_n(TC_i) = 1 - (U_n(TC_i) - L_n(TC_i))$$

where $U_n$ and $L_n$ are the upper and lower bounds of the Wilson confidence interval at 95% of $\hat{p}_n$. A value near $1$ indicates a narrow confidence interval and, consequently, a high reliability.

Then, for each test class $TC_i$, we consider all the look-back windows $\mathcal{W}_{\text{sticky}} \subseteq \mathcal{W}$ where the precondition of $\hat{p}_n$ holds. Then, the stickiness score ($\text{StickyScore}_i$) is calculated as the mean of the failure probabilities, pondered by their confidence:

$$\text{StickyScore}_i = \frac{\sum_{W_n \in \mathcal{W}_{\text{sticky}}} \hat{p}_n(TC_i) \cdot \text{Confidence}_n(TC_i)}{\sum_{W_n \in W_{\text{sticky}}} \text{Confidence}_n(TC_i)} \quad (2)$$

This model considers all significant look-back window signals, giving more weight to the most statistically reliable ones. It does not discard information; rather, it weighs it based on its credibility.

An important element in the StickyScore calculation is the definition of the set $\mathcal{W}$ that requires a preliminary project-specific analysis step. We vary $n$ and assess the stability and aggregate predictive power of $\hat{p}_n$ across the entire test suite. Such analysis considers $\text{Confidence}_n(TC_i)$ (as defined above) and *Lift* metric defined as:

$$\text{Lift}_n(TC_i) = \frac{\hat{p}_n(TC_i)}{P(\text{Fail})_{\text{baseline}}}$$

where $P(\text{Fail})_{\text{baseline}}$ represents the overall, unconditional probability that any given test class is a failing one, across the entire historical dataset. For example, if a test class has a $Lift_n$ of 5.0, it has a failure probability in the look-back window of size $n$ that is five times higher than a random test class. As such, $Lift$ indicates the predictive power of the computed $\hat{p}_n(TC_i)$.

Intuitively, enlarging the look-back window increases *Confidence* but lowers *Lift* because it averages over more observations (reducing variance) while mixing in older, less predictive failures that dilute the signal. We therefore choose $\mathcal{W}$ by qualitatively inspecting these trends and selecting window sizes that balance reliability with predictive power. Based on our analysis of the nine projects presented in Section IV, we determined that, most of the time, a set of small to medium windows, such as $\mathcal{W}\{1, \dots, 10\}$, provides the best trade-off. This data-driven selection process ensures that $\text{StickyScore}_i$ is built on robust and reliable signals.

### B. Relevance through Information Retrieval

While historical data provide a strong, context-free signal of test importance, effective regression testing must also account for the specific code changes introduced in the current build. To this end, *DANTE* estimates the relevance of each test case to the current *Git diff* by framing the problem as an Information Retrieval task, where code changes act as the query and test cases as the documents to be ranked.

**Cleaning and tokenization.** The process begins with two primary inputs: i) the current build (CB), which contains the entire test suite, and the *Git Diff*, which represents the code changes between the current and previous builds.

The raw *Git Diff* is first split into distinct chunks, each corresponding to the changes within a single file. In this phase a filtering process is applied to discard modifications to non-executable or configuration files (such as `.xml`, `.json`, and

`.md`), ensuring that the analysis focuses only on meaningful source code changes.

The source code of the test classes and the different chunks of the *Git Diff* are normalized by removing comments, annotations, and other non-semantic structures, including common keywords (e.g., `if` and `while`). The code is then tokenized with a *split-and-expand* strategy, which deconstructs the identifiers (e.g., 'MyClassName' → 'my', 'class', 'name') to create a richer vocabulary. To achieve this, we developed simple tokenizers tailored to *Java* and *Scala* (the two languages used in the projects evaluated in Section IV) and subsequently implemented a generic fallback option for all other languages. This step produces two intermediate artifacts: the *Tokenized Test Classes (TCs)* and the *Tokenized Changes for File (C4F)*.

**Sparse Information Retrieval.** We rely on pre-existing IR techniques to compute a relevance score between every *Tokenized TC* and every *Tokenized C4F*. We experimented with both dense and sparse IR techniques, including CodeBERT [41], TF-IDF [27], and Okapi BM25 [28]. In our setting, dense retrieval based on CodeBERT embeddings proved ineffective, as the resulting similarity scores lacked sufficient discriminative power for ranking test cases. This behavior is likely due to the fact that pre-trained code models capture general syntactic and semantic similarities across code artifacts, rather than the fine-grained execution-level relationships required for test-to-change relevance.

In contrast, sparse retrieval methods based on bag-of-words representations produced more informative relevance distributions, with a clear separation between highly relevant and irrelevant tests. TF-IDF and BM25 achieved comparable performance; following prior evidence on long-running CI/CD pipelines [18], and for simplicity and efficiency, we adopt TF-IDF with cosine similarity in our experiments.

The IR process produces a *relevance vector* for each test case $TC_i$, $\vec{R_i} = [Rel_{i,1}, Rel_{i,2}, \ldots, Rel_{i,F}]$, where $F$ is the number of modified files and $Rel_{i,j} \in [0, 1]$, for $j \in [1, F]$.

From $\vec{R_i}$, we derive the aggregated $\text{RelevanceScore}_i$ by adopting the *maximum relevance* as our primary signal:

$$\text{RelevanceScore}_i = \max_{f \in \text{ChangedFiles}} (Rel_{i,f}) \tag{3}$$

where *ChangedFiles* is the set of all modified files. This choice is motivated by the intuition that a test case is more valuable if it is highly specialized and maximally relevant to at least one specific code change. Therefore, $RelevanceScore_i$ represents the maximum relevance of a test for the current modifications.

### C. TCSP Heuristic

Once the historical and relevance scores have been computed, *DANTE* combines them into a unified prioritization signal. To ensure the two signals are comparable, *DANTE* first applies Min–Max normalization to both scores within the current build, obtaining $\text{HistoryScore}_i^{norm}$ and $\text{RelevanceScore}_i^{norm}$. The unified value score for each test case $TC_i$ is defined as:

$$V_i = \text{HistoryScore}_i^{norm} + \text{RelevanceScore}_i^{norm} \tag{4}$$

Still, a high value score alone is insufficient for practical TCSP, as it does not account for the execution cost. A valuable but extremely long-running test might consume the entire budget, preventing the execution of many other potentially useful tests. Following the empirical findings of Cheng et al. [18] (Finding 10), which show that cost-history cognizant formulations are particularly effective for long-running CI/CD test suites, we incorporate execution cost and failure count into the final prioritization metric. Specifically, we compute the *Final Value* score as:

$$FV_i = \frac{V_i \cdot f_i}{t_i} \tag{5}$$

where $f_i$ denotes the number of observed failures of test case $TC_i$ within the considered look-back window, and $t_i$ is its execution time.

Test cases are ordered (prioritized) in decreasing $FV_i$, favoring tests that combine high estimated value, frequent failures, and low execution cost. Tests are then selected sequentially from the top of this list and added to the final suite until the cumulative execution time reaches the predefined time budget, $T_{\text{max}}$. This greedy strategy yields a practical and efficient approximation of high-value test subsets under a time constraint.

### D. Complexity Analysis

From a computational perspective, *DANTE* is designed to remain lightweight and compatible with the execution cadence of modern CI/CD pipelines. The computation of the historical score scales linearly with the number $N$ of test cases, as it relies on aggregated execution outcomes extracted from CI/CD logs. Thus, the complexity is $\mathcal{O}(N)$. The relevance score is computed through a lightweight IR process whose cost is proportional to the size of the code changes in the current build, yielding a complexity of $\mathcal{O}(\Delta)$, where $\Delta$ denotes the number of modified files. This is achieved through caching, which avoids reprocessing unchanged test artifacts and source files across consecutive builds. Test prioritization requires sorting test cases by their $FV_i$ score, incurring a complexity of $\mathcal{O}(N \log N)$ for a test suite of size $N$, while budget-aware selection is performed greedily in linear time. Overall, the end-to-end complexity of *DANTE* is $\mathcal{O}(\Delta + N \log N)$ and remains comparable to that of simpler heuristics, making *DANTE* suitable for large-scale, long-running CI/CD environments.

## IV. Evaluation Design

This section provides the definition and design of *DANTE*'s evaluation, which aims to answer the following research questions:

- **RQ1**: How does *DANTE* compare to state-of-the-art approaches in terms of test case *prioritization* effectiveness?
- **RQ2**: How does *DANTE* compare to state-of-the-art approaches in terms of test case *selection* under time budget constraints?
- **RQ3**: How does test flakiness affect the performance of *DANTE* and competing techniques?

TABLE I
ANALYZED PROJECTS FROM THE LRTS DATASET [18].

| **Project** | **SLOC** | **#Build** | **#TC** | **#TC$_{FLK}$** | $\mu F$ | $\mu T$ |
|---|---|---|---|---|---|---|
| *activemq* | 669 K | 207 | 676 | 3 | 3 | 4.36 |
| *hadoop* | 4 M | 1 299 | 829 | 46 | 6 | 5.57 |
| *hbase* | 1 M | 278 | 1 061 | 51 | 2 | 9.28 |
| *hive* | 2 M | 2 056 | 1 273 | 91 | 9 | 26.12 |
| *jackrabbit oak* | 694 K | 860 | 1 897 | 9 | 12 | 3.27 |
| *james* | 793 K | 2 404 | 1 864 | 0 | 6 | 2.15 |
| *kafka* | 905 K | 11 843 | 1 232 | 85 | 4 | 7.59 |
| *karaf* | 186 K | 620 | 205 | 1 | 2 | 0.58 |
| *log4j 2* | 277 K | 270 | 641 | 5 | 3 | 0.25 |

RQ1 evaluates the effectiveness of *DANTE* in test case prioritization, analyzing whether it can rank test cases so as to detect failures earlier than state-of-the-art baselines, according to widely used prioritization metrics. RQ2 investigates the effectiveness of *DANTE* in test case selection under time budget constraints, assessing whether the selected subsets achieve better fault detection than competing approaches when only a portion of the test suite can be executed. RQ3 examines the robustness of *DANTE* to test flakiness, analyzing how non-deterministic test failures affect its ability to detect genuine faults compared to other techniques.

All experiments were executed on a machine equipped with a 13th Gen Intel(R) Core(TM) i7-1355U processor and 16GB DDR4 RAM.

### A. Study Context: Datasets and Baselines

**Datasets.** To evaluate *DANTE*, we leverage the *LRTS* dataset recently published by Cheng et al. [18]. LRTS aggregates 21,255 Jenkins CI/CD builds and 57,437 test suite runs drawn from ten large Apache projects developed between 2020 and 2023; each test suite run takes roughly 6.5 hours on average. Importantly, LRTS is, to the best of our knowledge, the most comprehensive and realistic public benchmark currently available for TCSP in long-running CI/CD settings, as it captures both the scale (thousands of builds) and the operational constraints (multi-hour suites) faced by real pipelines. For methodological consistency and to avoid confounding factors due to language- and tooling-specific test infrastructures, we restrict the analysis to the nine projects (out of ten) whose primary language is Java. Some Java projects (in particular *kafka*) have some Scala components, the *DANTE*'s tokenizer is also able to work on such a language, which is, therefore, also covered. The subset of Java projects covers 95% of the builds in LRTS, spans a diverse set of mature, industrially relevant systems, and provides enough cross-project variability to assess robustness beyond a single repository.

The selected projects span messaging systems (*activemq*), bigdata platforms (*hadoop*, *hbase*, *hive*), infrastructure libraries (*jackrabbit oak*, *kafka*, *karaf*, *log4j 2*) and a mail server (*james*). The code bases range from 186K to 4M SLOC, and their CI/CD histories include a few hundred to almost 12000 builds. Table I summarizes the key characteristics of the projects considered. For each project, column $SLOC$ shows the total amount of source code lines, column $\#Build$ the total amount of builds available in the dataset, column $\#TC$ the total amount of test classes within the project, column $\#TC_{FLK}$ the amount of *flaky* test classes within the project, column $\mu F$ the average amount of failed test classes per build, and $\mu T$ the average total time in hours to execute the full test suite. The figures reported in Table I show that even the smallest project (*karaf*) includes more than 200 test classes per build, while large systems such as *kafka* contain more than a thousand. The failure density is low (typically two to nine failing classes per build), and the widespread variation in execution times, from 15 minutes to 26 hours, motivates the need for precise selection and prioritization.

**Baseline approaches.** We compared *DANTE* against the best TCSP techniques identified in the literature according to the empirical findings of Cheng et al. [18]. In their large-scale study on the LRTS dataset, Cheng et al. show that augmenting classical heuristics with execution cost and historical information significantly improves performance in long-running CI/CD pipelines. In particular, they distinguish between *cost-cognizant* (CC) variants, which incorporate execution time into the prioritization logic, and *cost-history-cognizant* (CCH) variants, which jointly account for execution cost and historical failure information. Following Cheng et al.'s findings, for each baseline, we adopt the most comprehensive variant reported in their study. The considered baselines are:

- *LatestFail*$_{CCH}$ [42]: a history-based heuristic that prioritizes test cases according to their last failure time, extended with a cost-history-cognizant formulation.
- *MostFail*$_{CC}$ [6]: a history-based heuristic that ranks tests according to their cumulative number of past failures, using a cost-cognizant variant.
- *QTF-Last* [35], [24]: a cost-cognizant heuristic that prioritizes tests based on their most recent execution time.
- *QTF-Avg* [35], [24]: a cost-cognizant heuristic that ranks tests according to their average execution time.
- *LTR*$_{CCH}$ [43]: the best-performing learning-based approach identified by Cheng et al., which leverages gradient boosting and all available features using a cost-history-cognizant formulation.
- *Random*: a baseline that selects test cases uniformly at random, providing a lower bound for comparison.

In our experiments, we rely on the implementations and parameter settings provided by the original authors or by the replication package of Cheng et al. [18]. For *LTR*$_{CCH}$, we leverage the build history included in the benchmark in [18] to perform model training. However, we observed that in the associated replication package the training and test splits are not always strictly ordered chronologically, which may introduce look-ahead bias when learning from historical data. To mitigate this issue, we enforce a strict chronological ordering of builds during training. Moreover, to evaluate the method under ideal conditions, we retrain the model for each considered current build while preserving the original 75/25 train–test split. In practice, to keep execution costs and time under control, retraining would occur less frequently;

therefore, our results should be interpreted as an upper bound on the method's effectiveness.

### B. Study Methodology

**Evaluation metrics.** We evaluate the approaches with metrics aligned with our research questions.

To address **RQ1** (test case prioritization effectiveness), we use $APFD_c$ [44] (higher is better), a cost-aware extension of $APFD$ [36] that accounts for test execution time:

$$APFD_c = 1 - \frac{\sum_{i=1}^{m} C_i}{m \cdot C_{\text{tot}}} + \frac{1}{2\,C_{\text{tot}}}$$

where $m$ is the number of test failures in the build, $C_{\text{tot}}$ is the total cost (execution time) to run the whole test suite, and $C_i = \sum_{j=1}^{k_i} c_j$ is the cumulative cost up to the first test exposing failure $i$ (with $c_j$ the cost of test case $TC_j$ and $k_i$ its position in the ordered suite).

To address **RQ2** (test case selection effectiveness), we use $\%DTC$ (higher is better), i.e., test recall [6], which evaluates selection quality independently of execution order. The metric is defined as follows:

$$\%DTC = 100 \cdot \frac{DTC}{TF}$$

with $DTC$ the number of detected failures within the executed test case subset and $TF$ the total number of failures in the full test suite for the build.

We also report $SAPFD_c$ [45], which accounts for incomplete failure detection by scaling $APFD_c$ by the proportion of detected failures:

$$SAPFD_c = \%DTC \cdot APFD_c$$

To complement fault-detection metrics in RQ2, we report a qualitative metric called *Structural Diversity Score* (SDS). SDS is a static proxy for the architectural breadth of the selected test case subset, avoiding the overhead of collecting dynamic code coverage. Given a selected subset $S = \{t_1, \ldots, t_k\}$ and a function $\text{Pkg}(t)$ that extracts the test package (namespace prefix), we define:

$$\text{SDS}(S) = \frac{|\{\text{Pkg}(t) \mid t \in S\}|}{|S|}$$

SDS ranges in $(0, 1]$: values close to 1 indicate that selected tests span many distinct packages, whereas lower values indicate concentration within fewer packages. We use packages (treating packages and subpackages as distinct) as a lightweight structural unit, under the assumption that, in large systems, namespace boundaries typically reflect developers' modular decomposition and thus provide a reasonable proxy for covering different functional components. SDS can be interpreted as follows: high values suggest a more *exploratory* selection that spreads the budget across modules, while low values indicate stronger *exploitation* (concentration) on a limited set of modules. SDS is reported as a heatmap, showing the SDS achieved by the different approaches on the studied projects.

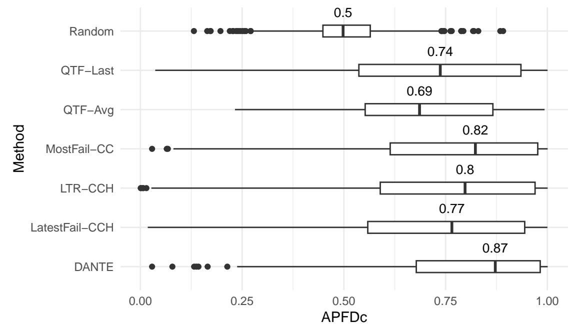


Fig. 2. $APFD_c$ distribution boxplots by technique.

TABLE II
RQ1: Wilcoxon pairwise signed-rank test comparison for $APFD_c$ aggregated over all projects ($p_{adj} < 0.001$ in all cases).

| Approach 1 | Approach 2 | Eff | Magn |
|---|---|---|---|
| *DANTE* | LatestFail$_{CCH}$ | 0.65 | L |
| *DANTE* | LTR$_{CCH}$ | 0.30 | M |
| *DANTE* | MostFail$_{CC}$ | 0.30 | M |
| *DANTE* | QTF-Avg | 0.49 | M |
| *DANTE* | QTF-Last | 0.40 | M |
| *DANTE* | Random | 0.79 | L |

Finally, to address **RQ3** (robustness to test flakiness), we use $\%DTC_{FLK}$ (higher is better):

$$\%DTC_{FLK} = 100 \cdot \frac{DTC - DTC_{FLK}}{TF}$$

where $DTC_{FLK}$ is the number of detected flaky test cases [6]. A higher value indicates that fewer detected failures are due to flaky behavior, yielding more reliable feedback about the current build.

**Statistical Procedures.** We perform statistical analyses in *R* [46] with significance level $\alpha = 0.05$. Since the same builds are evaluated by multiple techniques, our data consist of paired observations; therefore, we adopt non-parametric tests.

For each comparison scenario, we first apply Friedman's test [47] (`friedman_test` in *rstatix* [48]) as an omnibus test to assess whether at least one technique differs from the others. When the test is significant, we run a post hoc analysis using pairwise Wilcoxon signed-rank tests [49] (`pairwise_wilcox_test` in *rstatix* [48]), with Holm correction [50] to control for multiple comparisons. We also report Wilcoxon effect sizes (S: small; M: medium; L: large) to quantify the magnitude of the observed differences.

We apply this procedure to both the aggregated dataset and each individual project. Depending on the analysis, observations are paired by build only, or by build and budget level when a time budget constraint is involved.

## V. Evaluation Results

In the following, we present and discuss the results obtained while addressing the research questions posed.

TABLE III
WILCOXON PAIRWISE SIGNED-RANK TEST COMPARISON OF APPROACHES BY PROJECT: EFFECT SIZES FOR SIGNIFICANT DIFFERENCES (N/S WHEN NONSIGNIFICANT)

| Approach1 | Approach2 | activemq | hadoop | hbase | hive | jackrabbit-oak | james | kafka | karaf | log4j |
|---|---|---|---|---|---|---|---|---|---|---|
| **$\mathbf{APFD_c}$ effect size across projects** | | | | | | | | | | |
| *DANTE* | LatestFail$_{CCH}$ | 0.49 (M) | 0.49 (M) | 0.47 (M) | 0.86 (L) | 0.82 (L) | 0.78 (L) | 0.81 (L) | n/s | 0.58 (L) |
| *DANTE* | *LTR*$_{CCH}$ | n/s | n/s | n/s | n/s | 0.42 (M) | n/s | 0.65 (L) | n/s | n/s |
| *DANTE* | *MostFail*$_{CC}$ | n/s | n/s | 0.47 (M) | n/s | n/s | 0.42 (M) | n/s | 0.44 (M) | 0.57 (L) |
| *DANTE* | *QTF-Avg* | 0.42 (M) | n/s | 0.40 (M) | 0.63 (L) | 0.73 (L) | n/s | 0.87 (L) | 0.43 (M) | n/s |
| *DANTE* | *QTF-Last* | n/s | n/s | 0.62 (L) | n/s | 0.47 (M) | n/s | 0.87 (L) | 0.40 (M) | 0.58 (L) |
| *DANTE* | *Random* | 0.63 (L) | 0.74 (L) | 0.74 (L) | 0.87 (L) | 0.86 (L) | 0.76 (L) | 0.87 (L) | 0.70 (L) | 0.83 (L) |
| **%DTC effect size across projects by budget** | | | | | | | | | | |
| *DANTE* | LatestFail$_{CCH}$ | 0.26 (S) | 0.32 (M) | 0.34 (M) | 0.80 (L) | 0.39 (M) | 0.51 (L) | 0.44 (M) | n/s | 0.27 (S) |
| *DANTE* | LTR$_{CCH}$ | 0.27 (S) | n/s | 0.27 (S) | n/s | 0.46 (M) | n/s | 0.45 (M) | 0.14 (S) | 0.33 (M) |
| *DANTE* | MostFail$_{CC}$ | 0.30 (M) | n/s | 0.53 (L) | n/s | n/s | 0.47 (M) | 0.25 (S) | 0.26 (S) | 0.33 (M) |
| *DANTE* | QTF-Avg | 0.51 (L) | 0.56 (L) | 0.34 (M) | 0.65 (L) | 0.70 (L) | 0.41 (M) | 0.85 (L) | 0.34 (M) | 0.59 (L) |
| *DANTE* | QTF-Last | 0.44 (M) | 0.47 (M) | 0.47 (M) | 0.41 (M) | 0.58 (L) | n/s | 0.86 (L) | 0.44 (M) | 0.72 (L) |
| *DANTE* | Random | 0.72 (L) | 0.77 (L) | 0.71 (L) | 0.86 (L) | 0.86 (L) | 0.82 (L) | 0.85 (L) | 0.59 (L) | 0.86 (L) |
| **$\mathbf{SAPFD_c}$ effect size by budget** | | | | | | | | | | |
| *DANTE* | LatestFail$_{CCH}$ | 0.74 (L) | 0.62 (L) | 0.47 (M) | 0.81 (L) | 0.84 (L) | 0.71 (L) | 0.69 (L) | 0.42 (M) | 0.83 (L) |
| *DANTE* | LTR$_{CCH}$ | 0.40 (M) | n/s | 0.26 (S) | n/s | 0.39 (M) | n/s | 0.49 (M) | 0.36 (M) | 0.13 (S) |
| *DANTE* | MostFail$_{CC}$ | 0.60 (L) | 0.36 (M) | 0.51 (L) | 0.25 (S) | 0.38 (M) | 0.69 (L) | 0.59 (L) | 0.54 (L) | 0.73 (L) |
| *DANTE* | QTF-Avg | 0.54 (L) | 0.61 (L) | 0.31 (M) | 0.70 (L) | 0.72 (L) | 0.48 (M) | 0.86 (L) | 0.62 (L) | 0.83 (L) |
| *DANTE* | QTF-Last | 0.79 (L) | 0.72 (L) | 0.78 (L) | 0.74 (L) | 0.86 (L) | 0.82 (L) | 0.86 (L) | 0.63 (L) | 0.84 (L) |
| *DANTE* | Random | 0.78 (L) | 0.79 (L) | 0.73 (L) | 0.86 (L) | 0.86 (L) | 0.83 (L) | 0.86 (L) | 0.63 (L) | 0.86 (L) |

### *A. RQ1: Prioritization Effectiveness*

To answer the first research question, we evaluated the approaches' ability to order the *entire* test suite to discover failures as early as possible using the APFDc metric. Fig. 2 shows the box-plots for *APFDc*. *DANTE* stands above the other approaches, with a median $APFD_c = 0.87$, followed by *MostFail*$_{CC}$ and *LTR*$_{CCH}$ (0.82 and 0.8, respectively). The remaining heuristics form a second tier: *LatestFail*$_{CCH}$ reaches a median around 0.77, while time-based baselines (*QTF-Last* and *QTF-Avg*) are lower (0.74 and 0.69, respectively), and *Random* provides the expected lowest reference point (median 0.50). Beyond medians, *DANTE* also shows a visibly right-shifted distribution, indicating that its improvements are not driven by a small number of favorable builds, but occur across a broad portion of the build population. The statistical analysis in Table II confirms such differences. At an aggregate level, *DANTE* emerges as the most effective strategy. It significantly outperforms all traditional heuristics, including the strongest competitor, *MostFail*$_{CC}$ (*Eff* $= 0.30$, medium effect size). This is a relevant comparison because *MostFail*$_{CC}$ already captures a strong signal in long-running suites (tests that historically fail more often), but lacks the additional structure exploited by *DANTE*. A critical comparison is with the ML baseline, *LTR*$_{CCH}$. *DANTE* obtains a statistically significant improvement, again with a *medium effect size* (*Eff* $= 0.30$). Such empirical results lead to a noticeable finding: *DANTE*, while training-free, not only matches but surpasses the performance of a learning-to-rank regressor trained on historical data. The detailed project-level analysis, presented in the top part of Table III, offers further insights. Although the difference between *DANTE* and *LTR*$_{CCH}$ is not statistically significant in many projects, *DANTE* shows a decisive advantage in complex, large-scale environments. In particular, in *kafka*, *DANTE* outperforms *LTR*$_{CCH}$ with a *large effect size* (*Eff* $= 0.65$ and in *jackrabbit-oak* with a *medium effect size* (*Eff* $= 0.42$). This pattern is consistent with the intuition that learned models can be sensitive to build-to-build distribution shifts and to the volatility of failure signals, while *DANTE* relies on lightweight signals that remain available and stable across the entire project history.

> **RQ1 Summary:** *DANTE* outperforms state-of-the-art cost-cognizant heuristics in prioritization. Importantly, it matches or exceeds the performance of ML baselines (LTR), particularly in large-scale projects, without requiring training.

### *B. RQ2: Selection Effectiveness under Budgets*

We compare *DANTE* with the baseline approaches in terms of test *selection* quality under time budgets. We analyze three complementary aspects: (i) the ability to include failing tests in the selected subset ($\%DTC$), (ii) the quality of the ordering *within* the selected subset ($SAPFD_c$), and (iii) the structural diversity of the selected subset (SDS). We tested each approach under budgets corresponding to 2%, 5%, 10%, 15%, 20%, and 30% of the test suite's total execution time.

**Fault detection under constraints ($\%$DTC).** At an aggregate level (Table IV, top block), *DANTE* significantly outperforms all baselines ($p_{adj} < 0.001$ in all pairwise comparisons). The effect sizes indicate that the closest competitors for recall are the history- and learning-based techniques (*LatestFail*$_{CCH}$: *Eff* $= 0.38$ (M), *MostFail*$_{CC}$: *Eff* $= 0.27$ (S), *LTR*$_{CCH}$: *Eff* $=$

TABLE IV
RQ2: WILCOXON PAIRWISE COMPARISON AGGREGATED OVER ALL PROJECTS ($p_{adj} < 0.001$ IN ALL CASES).

| Approach 1 | Approach 2 | Eff | Magn |
|---|---|---|---|
| **%DTC effect size (Recall)** | | | |
| *DANTE* | LatestFail$_{CCH}$ | 0.38 | M |
| *DANTE* | LTR$_{CCH}$ | 0.20 | S |
| *DANTE* | MostFail$_{CC}$ | 0.27 | S |
| *DANTE* | QTF-Avg | 0.56 | L |
| *DANTE* | QTF-Last | 0.49 | M |
| *DANTE* | Random | 0.80 | L |
| **SAPFD$_c$ effect size (Selection Order)** | | | |
| *DANTE* | LatestFail$_{CCH}$ | 0.69 | L |
| *DANTE* | LTR$_{CCH}$ | 0.22 | S |
| *DANTE* | MostFail$_{CC}$ | 0.51 | L |
| *DANTE* | QTF-Avg | 0.64 | L |
| *DANTE* | QTF-Last | 0.80 | L |
| *DANTE* | Random | 0.81 | L |

0.20 (S)), while the gap widens against time-based approaches (*QTF-Avg*: *Eff*= 0.56 (L), *QTF-Last*: *Eff*= 0.49 (M)) and *Random* (*Eff*= 0.80 (L)). This is consistent with long-running suites: in projects where the average suite runtime is high, a small number of wasted selections quickly dominates the available budget, making failure-oriented signals more valuable than purely cost-driven ones.

The project-level results (Table III, $\%DTC$ block) indicate that the advantage is generally consistent across heterogeneous repositories. In particular, the margin against time-based baselines is pronounced for repositories with a long CI history (e.g., *kafka*, 11,843 builds, and *james*, 2,404 builds), where *DANTE* yields large effects vs *QTF-Avg* (*kafka*: 0.85 (L), *james*: 0.41 (M)) and vs *QTF-Last* (*kafka*: 0.86 (L)). If compared with the strongest learning baseline, *DANTE* is significantly better in multiple projects (e.g., *jackrabbit oak*: 0.46 (M), *kafka*: 0.45 (M), *log4j 2*: 0.33 (M)) and statistically indistinguishable in other cases (n/s), indicating that training is not necessary to obtain strong selection quality under budgets.

**Ordering within the selected subset (SAPFD$_c$).** When considering the execution order within the selected budget (Table IV, bottom block), *DANTE* emerges even more clearly from the baselines. It achieves a large effect size against all non-ML techniques (*LatestFail$_{CCH}$*: *Eff*= 0.69 (L), *MostFail$_{CC}$*: *Eff*= 0.51 (L), *QTF-Avg*: *Eff*= 0.64 (L), *QTF-Last*: *Eff*= 0.80 (L), *Random*: *Eff*= 0.81 (L)), and still improves over *LTR$_{CCH}$* (*Eff*= 0.22 (S)). The project-level analysis (Table III, $SAPFD_c$ block) confirms that these gains are not limited to a single repository: for example, *DANTE* shows medium effects over *LTR$_{CCH}$* in projects with large test suites (*kafka*: 0.49 (M), 1,232 tests; *jackrabbit oak*: 0.39 (M), 1,897 tests), where ordering within a tight budget is particularly important.

**Structural diversity (SDS).** SDS (Fig. 3) complements the fault-detection results by characterizing how, across the nine studied projects, the different approaches allocate the budget across the package structure. *Random* provides a reference that reflects the intrinsic structure of each project, which differs substantially across projects (e.g., *karaf* has only 205 tests, while *jackrabbit oak* and *james* exceed 1,800). Time-based strategies consistently yield the lowest SDS, indicating concentration on a small set of fast-running packages. In contrast, *DANTE* avoids this structural concentration while remaining guided by change relevance and historical metrics. This behavior is particularly important in projects where failures are not rare (e.g., $\mu F = 12$ in *jackrabbit oak* and $\mu F = 9$ in *hive*), because concentrating the budget on a narrow region increases the chance of missing regressions outside frequent and known hotspots.

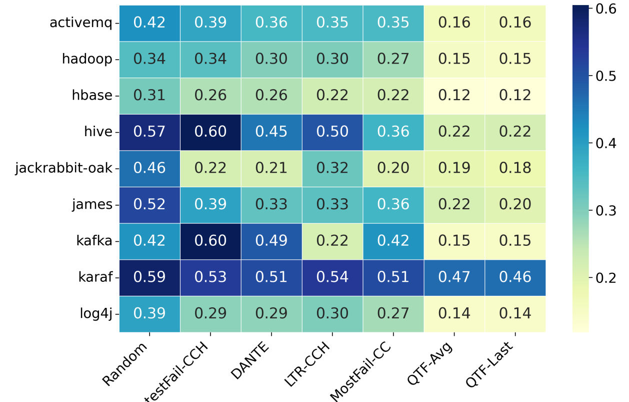

Fig. 3. Structural Diversity Score heatmap.

> **RQ2 Summary:** Under realistic time budgets, *DANTE* selects more failure-revealing tests and orders them earlier than the best baselines, including the learning-based technique, while avoiding the structural concentration typical of time-based strategies.

### C. RQ3: Effect of Flakiness

RQ3 investigates whether *DANTE* can prioritize *non-flaky* failure-finding tests. This is critical in long-running CI/CD pipelines, where a single flaky test can repeatedly consume a large fraction of the available budget while providing little actionable value to developers.

We quantify robustness to flakiness through $\%DTC_{FLK}$, which counts only *non-flaky* detected failures. Table V reports the statistical comparison aggregated over all projects. The results show that *DANTE* is consistently more robust than all baseline strategies ($p_{adj} < 0.001$ for all comparisons). In particular, *DANTE* achieves large effect sizes against the strongest history-driven baselines, *LatestFail$_{CCH}$* (*Eff*= 0.84, large) and *MostFail$_{CC}$* (*Eff*= 0.75, large), indicating that it is substantially better at avoiding failure signals that are not stable regressions.

This distinction matters because history-driven methods are naturally biased toward tests with frequent recent failures, regardless of whether those failures are persistent or intermittent.

TABLE V
RQ3: Wilcoxon signed rank test comparison for $\%DTC_{FLK}$ ($p_{adj} < 0.001$ in all cases).

| Approach 1 | Approach 2 | Eff | Magn |
|---|---|---|---|
| *DANTE* | LatestFail$_{CCH}$ | 0.84 | L |
| *DANTE* | LTR$_{CCH}$ | 0.75 | L |
| *DANTE* | MostFail$_{CC}$ | 0.75 | L |
| *DANTE* | QTF-Avg | 0.80 | L |
| *DANTE* | QTF-Last | 0.46 | M |
| *DANTE* | Random | 0.87 | L |

When flakiness is present, this bias can lead to spending the budget on tests that fail repeatedly but inconsistently, which does not help developers isolate regressions introduced by the latest change. *DANTE* mitigates this issue by grounding its historical signal in persistence: the probabilistic *stickiness* notion rewards failures that remain consistent across consecutive builds and attenuates failures that appear as isolated events. As a result, *DANTE* is less likely to select flaky tests and is more likely to prioritize tests whose failures are stable enough to be actionable.

The advantage of *DANTE* is also large against *QTF-Avg* (*Eff*$= 0.80$) and the *Random* baseline (*Eff*$= 0.87$). The gap narrows for *QTF-Last* (*Eff*$= 0.46$, medium). A plausible explanation is that time-based strategies, by selecting many short tests, can partially dilute the impact of individual long-running flaky tests under a fixed budget. However, this robustness is incidental: the approach does not explicitly distinguish flaky from non-flaky signals and therefore cannot systematically prefer stable failures when flaky tests are numerous.

In particular, *DANTE* also outperforms the learning-based baseline *LTR*$_{CCH}$ with a large effect (*Eff*$= 0.75$). This result suggests that in this long-running CI/CD setting, explicitly modeling persistence may be at least as important as learning complex decision boundaries from heterogeneous features.

These results clearly show that *DANTE* does not over-prioritize noisy tests: instead, it improves fault-finding effectiveness while increasing the proportion of detected *stable* failures, thus better isolating actionable regressions in the presence of flaky behavior.

**RQ3 Summary:** *DANTE* selects substantially more non-flaky failure-finding tests than all baselines, including the learning-based approach, showing strong robustness to flaky behavior and better isolation of actionable regressions.

## VI. Threats to Validity

In the following, we highlight some threats that can restrict the validity of the results obtained [51].

Threats to *internal validity* concern factors internal to the study that could influence our results. These may be related to the choice of particular budget levels considered. Although we experimented with different budget levels, it is possible that trends could change beyond the maximum budget (30%) considered. However, a budget too high would defeat the purpose of test selection. Other threats could depend on the specific settings used for *DANTE* and its competitors (Section IV). The settings used for *DANTE* may not be optimal, but we showed how *DANTE* generally outperforms the baselines anyway. As explained in Section IV-A, we mitigated errors arising from the implementation of baseline approaches by leveraging the replication packages from the original papers.

Threats to *external validity* concern the generalizability of our findings. Our evaluation is limited to nine large-scale projects written in Java and Scala from the *LRTS* dataset introduced by Cheng et al. [18]. Although these projects represent realistic long-running CI environments, more studies are needed to assess the applicability of our results to other programming languages and ecosystems. For comparison, we selected the strongest baselines identified by Cheng et al. [18], together with a *Random* baseline, to ensure a fair and well-established experimental setting.

Threats to *construct validity* concern whether our metrics capture the intended notions of effectiveness. The metrics used to evaluate *DANTE* and its competitors are widely adopted in TCSP research [18], [36], [45], [44], [6]. Regarding SDS, its interpretability depends on the assumption that test packages reflect the modular decomposition of the system; If tests are organized in ways that do not align with the application components, SDS can underestimate the actual diversity of the selected subset.

In this study, threats to *conclusion validity* primarily concern the statistical support for our findings. Throughout all research questions, we leveraged a suitable statistical procedure to pairwise compare multiple distributions.

## VII. Conclusions

This paper presented *DANTE*, a data-driven training-free framework for test case selection and prioritization tailored to long-running test suites in CI/CD pipelines. By evolving simple cost-cognizant heuristics with statistically grounded historical modeling and lightweight commit awareness, *DANTE* bridges the gap between rigid heuristics and heavyweight learning-based techniques. An extensive evaluation on large-scale CI/CD data shows that *DANTE* consistently improves the effectiveness of test selection and prioritization while remaining scalable and robust in realistic industrial settings.

## Data Availability and Replication Package

The replication package (implementation, analysis scripts, and results) is available at doi.org/10.5281/zenodo.18031888.

## References

[1] M. M. Lehman, “On understanding laws, evolution, and conservation in the large-program life cycle,” *J. Syst. Softw.*, vol. 1, pp. 213–221, 1980.
[2] B. Fitzgerald and K.-J. Stol, “Continuous software engineering and beyond: Trends and challenges,” in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, RCoSE 2014*. ACM, 2014, pp. 1–9.
[3] T. Stumm, “Testing in very large software projects,” Ph.D. dissertation, Ruprecht-Karls-Universität Heidelberg, 2021. [Online]. Available: https://archiv.ub.uni-heidelberg.de/volltextserver/31757/

[4] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston: Addison-Wesley, 2007.

[5] V. U. Ugwueze and J. N. Chukwunweike, "Continuous integration and deployment strategies for streamlined devops in software engineering and application delivery," *Int J Comput Appl Technol Res*, vol. 14, no. 1, pp. 1–24, 2024.

[6] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *Proceedings of the 41th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 263–272.

[7] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. C. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1095–1135, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09785-8

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[9] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, p. 102–112, Aug. 2000.

[10] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1–12.

[11] A. Haghighatkhah, "Test case prioritization using build history and test distances: An approach for improving automotive regression testing in continuous integration environments," Ph.D. dissertation, University of Oulu, 2020. [Online]. Available: http://jultika.oulu.fi/Record/isbn978-952-62-2492-9

[12] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 12–22.

[13] M. Bagherzadeh, N. Kahani, and L. C. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 3198–3219, 2022.

[14] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in *Proceedings of the 15th workshop on search-based software testing*, 2022, pp. 16–24.

[15] M. A. Khan, A. Azim, R. Liscano, K. Smith, Y.-K. Chang, G. Seferi, and Q. Tauseef, " ML-Based Test Case Prioritization: A Research and Production Perspective in CI Environments ," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2025, pp. 476–486. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST62969.2025.10989029

[16] T. Shu, Z. He, X. Yin, Z. Ding, and M. Zhou, "Model-based diversity-driven learn-to-rank test case prioritization," *Expert Systems with Applications*, vol. 255, p. 124768, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742401635X

[17] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2023.

[18] R. Cheng, S. Wang, R. Jabbarvand, and D. Marinov, "Revisiting test-case prioritization on long-running test suites," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 615–627.

[19] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–32, 2023.

[20] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of flaky tests in continuous integration: An industrial case study," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 329–340.

[21] H. Wang, R. Yu, D. Wang, Y. Du, Y. Zhao, J. Chen, and Z. Wang, "An empirical study of test case prioritization on the linux kernel," *Automated Software Engineering*, vol. 32, no. 2, p. 49, 2025. [Online]. Available: https://doi.org/10.1007/s10515-025-00522-8

[22] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, no. 3, p. 445–478, Sep. 2013.

[23] M. Rava and W. Wan-Kadir, "A review on prioritization techniques in regression testing," *International Journal of Software Engineering and Its Applications*, vol. 10, no. 1, pp. 221–232, 2016.

[24] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 491–504. [Online]. Available: https://doi.org/10.1145/3460319.3464834

[25] 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)*, 2019, pp. 213–222.

[26] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019, pp. 346–357.

[27] P. Bafna, D. Pramod, and A. Vaidya, "Document clustering: Tf-idf approach," in *2016 International conference on electrical, electronics, and optimization techniques (ICEEOT)*. IEEE, 2016, pp. 61–66.

[28] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[29] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 268–279.

[30] C. Magalhães, J. Andrade, L. Perrusi, A. Mota, F. A. Barros, and E. Maia, "Hsp: A hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study," *Journal of Systems and Software*, vol. 159, p. 110430, 2020.

[31] L. Yang, J. Chen, H. You, J. Han, J. Jiang, Z. Sun, X. Lin, F. Liang, and Y. Kang, "Can code representation boost ir-based test case prioritization?" in *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 240–251.

[32] K. R. Walcott, G. M. Kapfhammer, M. L. Soffa, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2006, pp. 1–11.

[33] S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009, pp. 213–224.

[34] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 324–336.

[35] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 656–667.

[36] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. C. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, pp. 29:1–29:34, 2022.

[37] H. Jahan, Z. Feng, S. M. Mahmud, and P. Dong, "Version-specific test case prioritization approach based on artificial neural network," *Journal of Intelligent & Fuzzy Systems*, vol. 36, no. 6, pp. 6181–6194, 2019.

[38] Z. Khalid and U. Qamar, "Weight and cluster based test case prioritization technique," in *Proceedings of the 2019 IEEE 10th International IEM Conference (IEMCON)*, 2019, pp. 1013–1022.

[39] C. Li, M. M. Khosravi, W. Lam, and A. Shi, "Systematically producing test orders to detect order-dependent flaky tests," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 627–638.

[40] S. Rahman, B. N. Chanumolu, S. Rafi, A. Shi, and W. Lam, " Ranking Relevant Tests for Order-Dependent Flaky Tests ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer

Society, May 2025, pp. 1999–2011. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00178
[41] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/
[42] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 13–25. [Online]. Available: https://doi.org/10.1145/3377811.3380437
[43] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2022.
[44] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, 2001, pp. 329–338.
[45] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *2007 IEEE International Conference on Software Maintenance*, 2007, pp. 255–264.
[46] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2023. [Online]. Available: https://www.R-project.org/
[47] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937.
[48] A. Kassambara, *rstatix: Pipe-Friendly Framework for Basic Statistical Tests*, 2023, r package version 0.7.2. [Online]. Available: https://CRAN.R-project.org/package=rstatix
[49] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[50] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.
[51] C. Wohlin, M. Höst, and K. Henningsson, "Empirical research methods in web and software engineering," in *Web Engineering*. Springer, 2006, pp. 409–430.