Revisiting Test-Case Prioritization on Long-Running Test Suites

Runxiang Cheng
University of Illinois Urbana-Champaign, USA
rcheng12@illinois.edu

Shuai Wang
University of Illinois Urbana-Champaign, USA
swang516@illinois.edu

Reyhaneh Jabbarvand
University of Illinois Urbana-Champaign, USA
reyhaneh@illinois.edu

Darko Marinov
University of Illinois Urbana-Champaign, USA
marinov@illinois.edu

Abstract

The prolonged continuous integration (CI) runs are affecting timely feedback to software developers. Test-case prioritization (TCP) aims to expose faults sooner by reordering tests such that the ones more likely to fail are run earlier. TCP is thus especially important for long-running test suites. While many studies have explored TCP, they are based on outdated CI builds from over 10 years ago with test suites that last several minutes, or builds from inaccessible, proprietary projects. In this paper, we present LRTS, the first dataset of long-running test suites, with 21,255 CI builds and 57,437 test-suite runs from 10 large-scale, open-source projects that use Jenkins CI. LRTS spans from 2020 to 2023, with an average test-suite run duration of 6.5 hours. On LRTS, we study the effectiveness of 59 leading TCP techniques, the impact of confounding test failures on TCP, and TCP for failing tests with no prior failures. We revisit prior key findings (9 confirmed, 2 refuted) and establish 3 new findings. Our results show that prioritizing faster tests that recently failed performs the best, outperforming the sophisticated techniques.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Software reliability**.

Keywords

Software testing, regression testing, reliability, test prioritization

ACM Reference Format:
Runxiang Cheng, Shuai Wang, Reyhaneh Jabbarvand, and Darko Marinov. 2024. Revisiting Test-Case Prioritization on Long-Running Test Suites. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680307

1 Introduction

Continuous integration (CI) has been widely adopted in software development to increase code quality and reduce release time [37, 72, 104]. CI runs test suites automatically, allowing developers to quickly identify mistakes in their commits. However, both codebase size and code commit frequency have grown rapidly over the years [3, 63, 79], which increases the test suite runtime in CI. The prolonged test suite execution can delay development cycles and prevent timely feedback to the developers.

Test-case prioritization (TCP) aims to expose potential faults in the regression change sooner by reordering tests in the test suite, such that the ones more likely to fail are run earlier [89]. The importance of TCP grows with the size and execution time of the test suite. If a test suite takes only seconds or minutes to run, then prioritizing tests will not save much time. In contrast, TCP can be especially important for long-running test suites.

To date, there is a wealth of TCP techniques, with several surveys conducted [18, 29, 36, 59, 61, 80, 89, 111]. Traditional TCP techniques use code coverage, and prioritize tests that cover more code elements [89, 90]. However, they have limited applicability, as code coverage is hard to collect [18, 71]. Recent empirical studies of TCP thus focus on techniques that rely on more accessible test features [19, 29, 110], such as history-based techniques that prioritize tests by their previous outcomes [48, 111], and time-based techniques that prioritize faster tests [13, 91]. Some other proposed TCP techniques use information retrieval (IR) [83, 92] or machine learning (ML) [9, 10, 50, 98] to leverage multiple information sources from CI to prioritize tests. For example, researchers developed Learning-to-Rank (LTR) TCP with supervised learning algorithms, and Ranking-to-Learn (RTL) TCP with reinforcement learning algorithms, both of which were shown effective.

Existing studies have investigated the effectiveness of TCP techniques in different contexts. Recent studies [9, 19, 67, 83, 110] evaluate TCP techniques on open-source projects. These studies consider historical test-suite runs and real test failures mostly from Java projects that use Travis CI [7]. For example, Peng et al. [83] studied IR-based TCP on 3,000 test-suite runs, Bertolino et al. [9] studied both LTR and RTL TCP, Elsner et al. [19] studied LTR TCP on 20 open-source [67] and 3 proprietary projects, and Yaraghi et al. [110] studied LTR TCP on test suites that last at least 5 minutes from 25 projects. They provide important findings on different techniques. However, most of their studied CI builds are outdated, e.g., from over ten years ago [8], and have test suites that are relatively short-running, e.g., on average several minutes (§3.3). As our analysis shows, longer-running test suites (e.g., on average several hours or more) from recent codebases have different characteristics than prior datasets (§3.5), which may result in different effectiveness and ranking outcomes of existing TCP techniques.

Other studies consider TCP on software from large companies [10, 18, 21, 29, 63, 101, 106]. While providing valuable experience, these studies focus on very few TCP techniques, e.g., history-based (§2.2). Further, their observations and techniques are based on large testing

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680307

infrastructures that are too costly or domain-specific for most open-source projects, such as prioritization for parallel test jobs at large-scale clusters or specific hardware [18, 106]. These studies also provide limited data for future investigations—their studied project(s) are inaccessible [10, 63, 101, 106], lack heterogeneity [21, 98], or have rather old artifacts [18].

One key challenge in studying TCP is the lack of up-to-date, high-quality datasets, especially in cases where TCP can help the most: long-running test suites. Moreover, many popular TCP techniques have only been studied separately across different datasets and settings—there has been no recent extensive evaluation of leading TCP techniques in a unified experiment setup. These challenges hinder researchers and practitioners from developing new research insights and identifying techniques applicable to their context.

In this paper, we introduce the first extensive, high-quality dataset of *long-running* test suites, called *LRTS*, curated from *recent* CI builds of popular *open-source* repositories (§3). On *LRTS*, we evaluate 59 TCP techniques from five leading technique categories: time-based, history-based, IR-based, learning-based (LTR and RTL) techniques, and cost-cognizant hybrid techniques (§2). We study the effectiveness of these techniques in three contexts: recent, long-running test suites (§5.1); impact of flaky tests and frequently-failing tests (§5.2); and prioritizing failing tests that have no prior failure (§5.3). Our study revisits key findings from recent TCP studies [9, 19, 21, 83, 110] and presents new findings.

Specifically, our paper makes the following contributions:

- **Dataset.** We collect *LRTS*, an extensive dataset focused on long-running test suites. It consists of 21,255 Jenkins CI builds with 57,437 test-suite runs from recent versions of 10 popular, large-scale open-source GitHub projects. Curated projects have different uses and are written in Java, Scala, Python, and C++. The builds span from 2020 to 2023, including 15,852 builds with 30,118 test-suite runs that have failed tests. The test-suite runs last for 6.5 hours on average (§3.3). We are releasing *LRTS*, with our code on: https://zenodo.org/records/12662090
- **Extensive Study.** We start with 26 *basic* TCP techniques—2 time-based, 6 history-based, 6 IR-based, 5 LTR, 6 RTL TCP techniques, and the *Random* baseline. We next apply two cost-cognizant hybrid TCP approaches to the basic techniques, to construct 33 hybrid techniques. In total, we evaluate 59 TCP techniques, on the widely-used metric Average Percentage of Faults Detected per Cost (APFDc) and APFD, under different failure-to-fault mappings [97]. We further assess how the effectiveness of these techniques is impacted by *confounding test failures* (failures of flaky tests and frequently failing tests). We also study their effectiveness in detecting the first failures of tests throughout the collected CI history.
- **Findings.** We revisit 11 key findings from recent TCP studies, confirming 9 and refuting 2 findings. We also present 3 new findings. Table 7 provides the summary of our findings. Among basic techniques, time-based techniques, e.g., running faster tests first, are the most effective and the least impacted by confounding test failures. Among all techniques, hybrid techniques that simply combine time-based and history-based heuristics perform the best, e.g., prioritizing faster tests that have failed recently, outperforming all sophisticated techniques. The overall ranking of techniques on *LRTS* is similar to that of prior work.

2 TCP Techniques

We first overview different TCP technique categories and describe the techniques we use in our study. We focus on evaluating only *previously proposed TCP techniques* and do not promote any new technique to mitigate potential bias in evaluating TCP techniques on our new dataset. TCP is the problem of finding a test execution order that detects more faults faster [89, 111]. Depending on the heuristics that guide the ordering, we can categorize basic TCP techniques in four main categories: *time-based*, *history-based*, *IR-based*, and *learning-based*. The fifth category is *hybrid* TCP techniques that systematically combine heuristics from other different categories.

2.1 Time-based TCP

A simple way of prioritizing tests is sorting them in ascending order by execution time, expecting that executing more tests within a given time can find more failures [91]. This TCP category, called Quickest-Time-First (QTF), has been recently shown to rival or outperform more sophisticated TCP techniques on short-running test suites [13, 19, 83]. We evaluate 2 time-based techniques: *QTF-Last* and *QTF-Avg*: the former uses the execution time of the previous test run as the prioritization heuristic, while the latter uses the average execution time from prior test runs.

2.2 History-based TCP

History-based techniques prioritize tests based on the tests' outcome information from prior executions—they assume a test that has failed or changed its outcome is more likely to detect faults in the new code version. History-based techniques can incorporate different outcome information, such as test failure, test transition, or the association between the test and changed code files.

2.2.1 Test Outcome. Two history-based TCP heuristics are most commonly used. Test failure history considers whether the test has previously failed. Test transition history considers whether the test outcome has changed (failing to passing, or vice versa). We evaluate 4 history-based techniques from this sub-category: (1) *MostFail* prioritizes tests that have a higher historical failure count [4, 63, 74, 76, 77, 83], (2) *LatestFail* prioritizes tests that failed more recently [4, 18, 42, 76], (3) *MostTrans* prioritize tests that have a higher historical transition count [19, 52], and (4) *LatestTrans* prioritizes tests that transitioned more recently [19, 52].

2.2.2 Test Outcome and Changed File Association. Test outcome history can be more informative when associated with the change under test. Researchers thus proposed to trace the outcome history and changed files, and to prioritize tests whose outcomes were more related to changed files based on previous test runs [3, 49, 75, 84, 96, 109]. We evaluate 2 history-based techniques from this sub-category: *TF-FailFreq* prioritizes tests with higher failure count with respect to the changed files, and *TF-TransFreq* prioritizes tests with higher transition count with respect to the changed files [19].

2.3 IR-based TCP

IR-based techniques rely on textual similarity to identify the tests that are more relevant to code changes [83, 92]. They extract code tokens from tests and code (or code change diff), and process them into a corpus of documents and a query with off-the-shelf IR models.

Table 1: LTR TCP feature sets.

F_1: test history features	F_2: (Test,File)-history features
Failure count	Max (test,file)-failure freq
Last failure	Max (test,file)-transition freq
Transition count	Max (test,file)-failure freq (relative)
Last transition	Max (test,file)-transition freq (relative)
Average duration	
F_3: (Test,File)-similarity features	F_4: change features
Min file path distance	Distinct authors
Max file path token similarity	Changeset cardinality
Min file name distance	Amount of commits

For a code change presented as a query, an IR-based technique prioritizes tests whose documents are more similar to the query. IR-based techniques can be configured to use different IR models, e.g., Term Frequency-Inverse Document Frequency (TF-IDF) [93] or BM25 [88], and the amount of context they consider for a code change [83]. For example, *NoContext* techniques only use tokens from the exact changed lines to construct the query, *WholeFile* techniques use all the tokens from the changed files, and *GitDiff* techniques use tokens from the git diff file (same as using 3 lines of context [23]). We evaluate 6 IR-based techniques from prior work [83] that use BM25 and TF-IDF IR models with the 3 different context lengths mentioned above.

2.4 Learning-based TCP

With the advent of machine learning (ML), a number of TCP techniques use ML algorithms to predict the ranking of tests. These TCP techniques can be broadly put into two sub-categories [9]: Learning-to-Rank (LTR) and Ranking-to-Learn (RTL).

2.4.1 Learning-to-Rank. LTR TCP techniques use supervised learning algorithms, in which an ML model is trained on historical CI builds to predict ranking of tests for future builds [9, 10, 19, 63, 66, 77, 80, 110]. LTR techniques train ML models with features from the test, code or code change, and execution history [19, 110], to predict the probability of test failure, which then determines the test order. The effectiveness of LTR techniques depends on the underlying ML model and the training process, even if trained on the same data. The choice of features can also substantially impact the model performance in LTR TCP.

Prior work evaluated how different ML algorithms impact the effectiveness of LTR techniques [9, 19, 110]. They also explored to what extent the training:testing data ratio, e.g., using the first (chronologically ordered) 50% or 75% of the test runs for training and the rest for testing, impacts the outcome of TCP. We revisit the most studied ML algorithm (gradient boosting trees) and training:testing data ratio (75%) [9, 19, 80]. We use the most effective features prior work identified that are also easily accessible in CI [19, 65, 110]. Table 1 lists them categorized into four feature sets, which follow the same definitions as in Elsner et al. [19]. In total, we evaluate 5 LTR techniques, 4 techniques using one set of features each, and 1 technique using all four feature sets.

2.4.2 Ranking-to-Learn. RTL TCP techniques use reinforcement learning (RL) algorithms [6, 9, 78, 98]. In contrast to LTR where a model is trained *offline*, RTL trains its model *online*—RTL TCP is deployed without learning on historical builds, and learns a test ranking policy for a project at runtime. It continuously (1) ranks tests based on test states of the current CI build, and (2) receives feedback from the ranking to improve its policy for the next build.

A test state encodes a test's metadata, e.g., previous outcome and duration. Given all test states of the current build, RTL TCP selects an action for each test (i.e., giving each test a priority score) with its current policy or by random exploration. After running the prioritized test suite, a reward is fed back to the model to improve the current policy—a higher reward encourages prioritizing a given test state. The effectiveness of RL TCP is sensitive to its parameters, e.g., RL model choice, data encoded in the test state, and definition of the reward function. As in prior work [9, 98], we evaluate neural network (*NN*) and Q-table (*Tabl*) as RL agents on three rewards functions: failure count (*FailCount*), test failure (*TestFail*), and time rank (*TimeRank*). In total, we study 6 RTL TCP techniques.

2.5 Hybrid TCP

After describing the basic TCP techniques, we now describe the hybrid TCP techniques, which combine the heuristics from previous categories for better effectiveness. For example, we can build a hybrid technique based on *MostFail* by prioritizing tests not only by higher failure count but also by shorter execution time. Hybrid approaches have improved the effectiveness of basic techniques in different TCP settings [14, 83], which motivated us to include them in our study. We adopt two hybrid TCP approaches from prior work [83]: cost-cognizant (*CC*) and cost-history-cognizant (*CCH*).

Cost-cognizant. Given a basic technique that ranks tests based on score s in the ascending order, a *CC* hybrid technique prioritizes tests in the ascending order of $s * t$, where t is the test execution time from the previous run. *CC* techniques promote prioritizing tests with a short execution time.

Cost-history-cognizant. Given a basic technique that ranks tests based on score s in the ascending order, a *CCH* hybrid technique prioritizes tests in the ascending order of $s * t/c$, where c is the test's failure count. *CCH* techniques promote prioritizing tests that failed more often per unit of time.

3 Dataset of Long-Running Test Suites

§3.1-3.2 describe our project selection criteria and the construction of *LRTS*. §3.3 provides more details on *LRTS*, with an analysis of the distributions of its CI builds and test failures. §3.4 describes how we account for confounding test failures (failures of flaky tests and frequently failing tests). §3.5 compares characteristics of *LRTS* with recent datasets of short-running test suites.

3.1 Project Selection

We sought projects that were open-source, because they often provide transparent data access to their *recent* CI builds, with test failures induced by real faults [67]. In selecting projects, we prioritize those actively maintained, with a substantial history of commits and builds. A large number of commits and a long build history increase confidence in generalizing the empirical findings and claims from the study. The most critical criterion for our work was the inclusion of projects with *long-running test suites*, because these projects can benefit the most from TCP.

We focus on selecting projects from the Apache Software Foundation (ASF) [5] because it offers a diversity of renowned open-source

Table 2: *LRTS* dataset summary. TSR denotes test-suite run, TC denotes test class, and TM denotes test method.

Project	Main PLs	SLOC	Period (days)	#CI build	#TSR	#Failed TSR	Statistics (Averages) on failed TSRs				
							#TC	#Failed TC	#TM	#Failed TM	Duration (hours)
ActiveMQ	Java	669K	827	207	207	109	676	3	6,081	34	4.36
Hadoop	Java	4M	1,094	1,299	1,299	543	829	6	7,289	24	5.57
HBase	Java	1M	504	278	553	215	1,061	2	6,369	3	9.28
Hive	Java, HiveQL	2M	618	2,056	2,056	1,419	1,273	9	40,921	83	26.12
Jackrabbit Oak	Java	694K	745	860	860	639	1,897	12	19,699	107	3.27
James	Java, Scala	793K	786	2,404	3,147	1,399	1,864	6	34,718	37	2.15
Kafka	Java, Scala	905K	984	11,843	39,006	24,047	1,232	4	19,399	12	7.59
Karaf	Java, Scala	186K	959	620	620	174	205	2	841	2	0.58
Log4j 2	Java	277K	436	270	528	162	641	3	3,918	4	0.25
TVM	Python, C++	818K	631	1,418	9,161	1,411	526	3	8,564	37	4.83
Total				**21,255**	**57,437**	**30,118**					

projects and has been studied by many researchers for over two decades [73]. While the source code of ASF projects is easy to find, collecting their build logs is challenging as they use different CI services and organize their CI build data differently. In particular, they rarely use free services, such as GitHub Actions or Travis CI, because their test-suite runs are rather long, beyond the usual limits offered in the free tier of these services [26]. Instead, they mostly use Jenkins CI, on public or private servers.

We consider only ASF projects that preserve CI history on publicly accessible Jenkins CI servers (e.g., [31, 45]), as these projects can have long-running test suites, and Jenkins CI provides uniformed API for downloading serialized build data [40]. We select from the longest-running projects, where the test-suite execution time for the majority of the project's most recent CI builds exceeds 30 minutes. Many of these projects delete build history regularly—our dataset thus includes some CI builds that are no longer available.

Table 3 lists the 10 projects in *LRTS*. All projects consist of several sub-projects (e.g., multi-module Maven projects in Java). They use a mix of programming languages (Java, Scala, Python, and C++) and build systems (8 Maven [68], 1 Gradle [28], and 1 CMake); all use Jenkins CI. To our knowledge, *LRTS* is the first open-source dataset for investigating the effectiveness of TCP techniques on *multiple* large-scale projects with long-running test suites and actual CI failures.

Table 3: Projects in our dataset.

Project	Primary Use	Stars
ActiveMQ	Message broker	2K
Hadoop	Big-data processing	14K
HBase	Big-data storage	5K
Hive	Data warehouse	5K
Jackrabbit Oak	Content repository	381
James	Mail server	848
Kafka	Stream processing	26K
Karaf	Modulith runtime	669
Log4j 2	Logging API	3K
TVM	Compiler stack	10K

3.2 Dataset Curation

We collect CI builds with *real test failures* for each project, and extract the corresponding test-suite runs and code change data. We use data collection procedure similar to prior work [67, 79, 83, 110] and describe our differences below.

3.2.1 CI Builds. We focus on CI builds triggered by PR commits, rather than branch pushes, because builds for PRs may fail more frequently than builds for a particular branch (e.g., `trunk`). Each PR can have multiple commits and multiple builds. We first collect build metadata from the CI server, then collect the metadata of corresponding PRs via GitHub API [25].

A Jenkins CI build can have multiple stages [39], similar to how a Travis CI build can have multiple jobs [8, 110]. In *LRTS*, we observe some builds having multiple stages, where each stage has a test-suite run on a different environment, and the test report of that build records all the runs. For example, a Kafka build can run the same code for four different environments (JDK 8, 11, 17, and 20) in four stages [46]. Following prior work [19, 83, 110] that treated each Travis CI ⟨build, job⟩ pair as a test-suite run, we treat the test-suite run of each ⟨build, stage⟩ pair as a data point for evaluating TCP. We also treat each stage in a project as having its own CI history, which consists of all builds that included that stage.

3.2.2 Test Suite Information. We obtain test report URLs from build metadata files, and extract test reports in `JSON` format via Jenkins CI API [40]. Our process differs from the extraction of test results from Travis CI [8] because the test report data from Jenkins CI provides much more uniform information, with no need to parse textual build logs. As a result, *LRTS* has more accurate information about test runs than datasets built from Travis CI [8, 67, 79, 83, 110]. Each test report contains the duration, outcome, and name of each test method and its test class in the test-suite run(s) of the build. It also contains stack traces for failed tests, and metadata of the run.

3.2.3 Code Change Information. The code change of a PR build is the diff between its PR commit head (denoted as `head`) and the branch commit head that `head` is being merged into (denoted as `base`) [8]. We extract `head` from the build metadata file, and `base` from the build log. For each pair of `head` and `base`, we extract the code change data via GitHub API [24], which includes the diff file URL, commit identifiers, authors, and the list of changed files. We use the diff file URL to download corresponding code change diff.

3.3 Dataset Overview

LRTS curates the data of 21,255 unique CI builds from 10 projects. These builds have 57,437 test-suite runs (TSRs), of which 30,118 (59%) TSRs had at least one failed test. A build can have more than one TSR (§3.2.1). Table 2 provides more details on *LRTS* [15]. The durations are based on Jenkins CI test reports, by summing up the durations of all executed tests in each TSR. If tests run in parallel to reduce the total elapsed time, TCP can prioritize and parallelize tests to find failures sooner [11, 115]. For a fair comparison, as in prior work [9, 19, 29, 83], we evaluate TCP techniques while considering that each TSR runs its tests sequentially.

In Table 4, we compare *LRTS* with other datasets in TCP studies since the RTPTorrent release in 2020 [29, 67]. We omit datasets

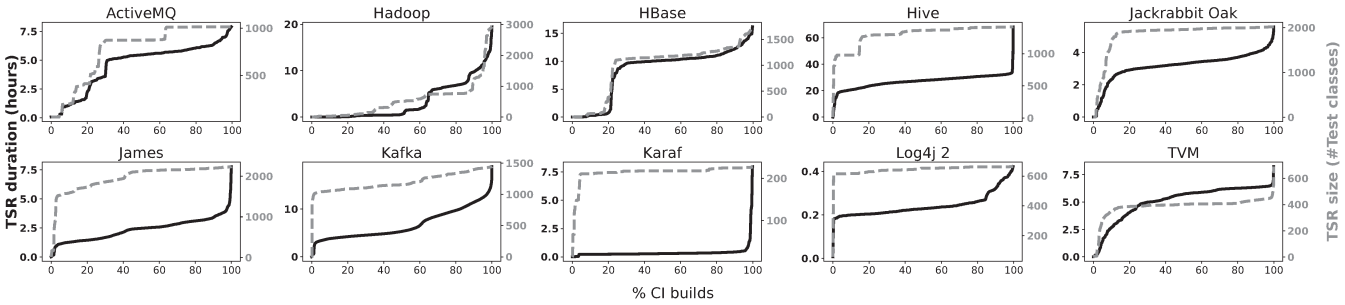

Figure 1: Distribution of CI builds by the duration (hours) and size (number of test classes) of all (not only failed) TSRs. The solid dark lines and left y-axes show CDFs by TSR duration. The dashed lighter lines and right y-axes show CDFs by TSR size.

with synthetic CI failures [2, 53, 115, 116], or whose long-running test suites come from inaccessible, proprietary projects [6, 19, 56, 65, 78, 95]. Many recent studies use the same set of builds from before 2016 in TravisTorrent or proprietary projects [7, 8, 18, 67, 98]. In comparison, the builds in *LRTS* span from 2020 to 2023, reflecting more current CI practices and are suitable for up-to-date TCP studies [80]. We continue to collect build data from these projects to preserve them before they get deleted: as of now, we have over 32K builds and over 108K TSRs [58].

Table 4 shows that the average TSR duration (measured in hours) in *LRTS* is (1) at least 18 times larger than the other datasets with multiple projects and (2) similar to the Google Chrome dataset that has only one project. Table 4 also shows that some prior datasets have more projects, because TravisTorrent collected data from the centralized Travis CI service that allowed mining 1000+ repositories uniformly, while we need to find specific Jenkins CI servers for each project. Those projects are also much smaller, with shorter-running test suites and fewer TSRs per project [7]. Overall, *LRTS* complements the existing datasets with a different set of projects that use Jenkins CI, with *recent* builds, and *long-running* test suites.

Table 4: Comparing TCP datasets.

TCP dataset	#Proj	#TSR	Duration
RTPTorrent [67]	20	100K	0.17
Peng et al. [83]	123	3K	0.09
RT-CI [9]	6	3K	<0.01
Pan and Pradel [79]	242	15K	0.35
TCP-CI [110]	25	21K	0.27
Chrome [21]	1	50K	7.96
***LRTS* (Ours)**	10	57K	6.50

3.3.1 Distribution of CI Builds. Figure 1 shows the distribution of builds in *LRTS* by their TSR duration and test-suite size. For a build with multiple TSRs, we use the average duration and test-suite size across its TSRs. In terms of duration, 7 out of 10 projects (all but Hadoop, Karaf, and Log4j 2) have over 80% of the builds with TSRs that last more than an hour, and Hadoop has over 50% of the builds with TSRs that last more than an hour. In terms of test-suite size, the number of test classes per TSR is several hundreds or higher in all the projects. For example, almost all TSRs in Kafka and TVM executed over 1000 and 400 test classes, respectively.

3.3.2 Distribution of Test Failures. In *LRTS*, failure frequencies of the *failed* test classes in a project follow a long-tail distribution. Namely, most of the failed test classes in a project only failed a handful of times across builds, while a few test classes failed order(s) of magnitude more often than the others. Table 5 shows distributions of failed test classes by their failure counts. In 7 projects, 75% of their failed test classes have failed at most 8 times across the collected CI history. In all the projects, the failure count of a failed test class is much smaller than that project's total number of failed TSRs in Table 2—the set of failed TSRs in each project is contributed by a diverse set of test classes rather than being dominated by just a few test classes. Each project also has a few test classes that failed much more often than others. These failures are likely flaky or intentionally ignored by developers [21, 83, 110]. §3.4 further describes these confounding test failures.

We also analyze the overlap of failed tests (excluding confounding test failures) across TSRs of the same multi-TSR build, by computing the Jaccard index of failed tests across failed TSRs for each build. It is rather low, on average 0.12. Our result shows that failed tests in TSRs are often different even in the same build, which further indicates that *LRTS* consists of diverse test failures.

Table 5: Distribution of failed test classes by their #failed TSR in each project.

Project	Min	Q1	Q2	Q3	Max
ActiveMQ	1	1	2	3	30
Hadoop	1	1	2	4	94
HBase	1	1	2	4	22
Hive	1	2	7	17	302
Jackrabbit Oak	1	3	3	7	124
James	1	2	5	12	67
Kafka	1	6	20	70	2,972
Karaf	1	1	3	5	41
Log4j 2	1	1	2	5	29
TVM	1	3	4	8	213

3.4 Confounding Test Failures

Some test failures distract developers and provide little to no value to be detected during regression testing, and thus should not be prioritized. Datasets of real CI builds may contain these failures, which can impact the apparent effectiveness of TCP techniques [19, 21, 51, 82, 83]. We call these failures *confounding test failures*—inspired by the term "confounding variable" in causal inference: a variable that relates the cause and outcome of interest (e.g., faults and test results) but is not of interest itself (e.g., flakiness-induced failures) [30]. We next describe two types of tests that cause confounding test failures—flaky tests and frequently failing tests—and how we account for them in our study.

3.4.1 Flaky Tests. Flaky tests are tests that can nondeterministically pass or fail for the same code under test [60]. Because they may impact the effectiveness of TCP techniques on detecting regression-induced failures that developers care about, some TCP studies on real CI failures explicitly account for flaky tests [21, 51, 82, 83].

Popular, large-scale systems often consider flaky tests in CI to some extent [21, 63]. In *LRTS*, 5 out of 10 projects (ActiveMQ,

Hadoop, HBase, Hive, and TVM) consider potential flaky test failures by re-running failed tests, using the rerun option in Maven [69] or Pytest [85]. HBase and Hive also maintain dashboards to proactively run jobs to track flaky tests and exclude them in CI runs [34, 38]. All 10 projects use JIRA [44] or GitHub issues actively to track the discovery and resolution of flaky tests. Some identified flaky tests are manually fixed or skipped during testing.

To properly account for flaky test failures in TCP studies, it is crucial to identify these failures. Some prior work [63, 83] reruns builds multiple times and finds tests with inconsistent outcomes. Due to resource constraints, we cannot rerun all 30,118 failed long-running TSRs multiple times [19, 21]. Unfortunately, Jenkins CI test reports do not include "flaky" tag even when Maven or Pytest has been used for reruns. We thus employ two alternatives to identify flaky tests in *LRTS*. Our key insight is to leverage issue trackers that all projects already actively use.

First, we manually inspect flaky-test-related JIRA and GitHub issues. We downloaded all issues returned from fuzzy search with the keyword "flaky" on each project's issue tracker [43]. We automatically filter out flaky test issues closed before the earliest build in *LRTS*; for the remaining issues, we inspect to determine if they indeed fix a flaky test and what the exact test name is. Across all projects, we inspected 746 issues and identified 344 flaky tests with their fix dates. For each identified flaky test, we label all its failures before the fixed date as flaky test failures and all its failures after the fixed date as actual regressions.

Second, for a build with multiple TSRs in different environments (§3.2.1), we treat failures that were not in all its TSRs as flaky. This approach is similar to rerunning [63] but each rerun is with a different environment—it bears the risk of misidentifying test failures as flaky due to actual environment-specific faults, but it ensures the remaining test failures are more likely to be non-flaky as they occurred in multiple environments [79]. Once we identify flaky test failures, we can remove all such failures from the TSRs.

3.4.2 Frequently Failing Tests. As shown in §3.3.2, for most of the projects in *LRTS*, some of the test classes failed frequently across failed CI builds. These tests often fail independent of the code changes [110], and some of the failures could be due to test flakiness [21]. In our case, 53% of the frequently failing tests are also identified as flaky tests. Frequently failing tests are often ignored by developers. Following prior work [110], we remove these tests by performing an outlier analysis with a three-sigma rule of thumb—we remove failures of test classes whose failure frequency is above the $mean + 3 * stdev$ of all builds for each ⟨project, stage⟩ pair.

3.5 Comparison with Short-Running Test Suites

Besides having more recent builds and codebases, one key characteristic of *LRTS* is in its long-running test suites (§3.3), which *may* lead to different effectiveness and ranking results of existing TCP techniques than the short-running test suites. Results may differ because identifying and prioritizing failing tests on longer-running test suites may be more difficult.

One difficulty comes from the fact that longer-running test suites on average have more tests but not more *failing* tests. By comparing *LRTS* and three recently used datasets (including an extended RTPTorrent) [19, 83, 110], we find that test suites in *LRTS* on average have 3–6 times more test classes but still a small number of failures, e.g., four in *LRTS* and 2–6 in others. The probability of a failure occurring in *LRTS* is thus 2–4 times *smaller*. Further, long-running test suites can have more *diverse* failures by simply having more tests. Failed tests in *LRTS* fail less frequently compared to the other datasets: the number of times a failed test fails over the number of failed TSRs in a project in *LRTS*, on average, is 6–13 times smaller.

The other difficulty stems from the increased runtime of tests in long-running test suites. Beyond having more tests, tests in *LRTS*, on average, run 10 to 20 times longer than tests in the other datasets. For example, the 3rd quartile of test class runtime in *LRTS* and extended RTPTorrent [19] are 10 and 0.4 seconds, respectively. Longer runtime often indicates that a test has more dependencies and interacts with more code elements, which can result in more complex behaviors that are harder to be captured by TCP techniques without code coverage or dependency information. Our results also show that minor imprecision in the TCP technique can cause a large penalty in the technique's failure-finding effectiveness (§5.1.2).

Overall, our analysis shows that projects with a longer TSR runtime often correlate with other properties such as (1) more tests, (2) longer-running individual tests, (3) more diverse set of failures, and (4) lower fail ratio (relative number of failures to the number of tests). Thus, TCP techniques that work well on short-running test suites may not work as well on long-running test suites. Therefore, it is not obvious *a priori* which TCP technique can effectively prioritize failing tests ahead of the passing tests in a much larger test suite, which motivates our study.

4 Experimental Setup

In this section, we describe our evaluation settings. We also discuss our data collection process and implementation for the studied TCP techniques and experimental procedure.

4.1 Evaluation Settings

4.1.1 Failure-to-Fault Mappings. Mapping test failures to the faulty code is crucial for evaluating TCP techniques—the goal of TCP is to find different faults, not just many failures due to the same fault. Some prior work injects artificial faults into the code to have the exact mapping from test failures to the injected faults. Recent studies [19, 67, 83], including ours, consider actual test failures from CI builds. In such cases, it is difficult to know the exact mapping without a deep investigation of each TSR. Prior work thus mostly uses two failure-to-fault mappings while evaluating TCP techniques: $FFMap_S$ that assumes that all test failures in a TSR map to the same fault; and $FFMap_U$ that assumes that each test failure in a TSR maps to a unique fault [19, 83]. We evaluate on both mappings, following prior work [97].

4.1.2 Test Granularity. To better revisit findings from prior studies in our new context, we use the same test granularity for prioritization as they use, at the level of test classes [9, 19, 83, 110] rather than test methods [16, 70] or test suites [18, 55].

4.1.3 Evaluation Metrics. Common metrics used to evaluate TCP techniques are Average Percentage of Faults Detected (APFD) [111] and Average Percentage of Faults Detected per Cost (APFDc) [17, 29, 64]. Both metrics are normalized to $[0, 1]$; a small difference

can indicate a large time for longer-running test suites. APFDc commonly uses the execution time of the evaluated TSR as the cost [13, 20]. Thus, it effectively measures how many faults are found per time, e.g., a 0.1 increase reduces the time to detect all faults by 10% of the test suite time on average (or 39 minutes in our studied projects based on Table 2). We evaluate on both metrics.

4.2 TCP Data Collection

We first order all the builds in *LRTS* chronologically to respect temporal dependencies in regression testing [19, 84, 102]. For time-based TCP, we collect relevant test execution time data from builds prior to the current build. We add 0.001 s to all execution times because Maven and Gradle report execution times as 0.000 s if less than 0.001 s [14]. For history-based TCP, we collect test outcome data from builds prior to the current build. For IR-based TCP, we checkout the `base` code version of the current build and apply the corresponding code change (diff between `base` and `head`). After applying the change, we collect code tokens from all test files and all changed files to construct *documents* and *query*. For LTR TCP, we collect all features from Table 1 following prior work [19, 110].

4.3 TCP Technique Implementation

4.3.1 General Logic. We wrote a generic pipeline to run and evaluate different TCP techniques. Given a TCP technique, the pipeline first processes the test data to compute the priority score of each test in the to-be-prioritized test suite. It then ranks the tests in the ascending order of the scores. For example, to evaluate the *MostFail* technique on a test suite T, the pipeline loads the historical failure counts of all tests in T, computes the priority score as the reciprocal of failure count, and sorts tests by their scores.

4.3.2 IR-based TCP. Prior work used an NLP-based or AST-based tokenizer to parse the content of the collected files into tokens. Both approaches yield similar performance [83]. We use the NLP-based tokenizer from Peng et al. [83] as it is language-agnostic. Tokens from a test file are treated as an individual *document*, and tokens from all the changed files are collectively treated as the *query*. The IR model takes test documents and a query as input, and outputs the similarity score between each test and the code change.

4.3.3 LTR TCP. We follow the same data processing, implementation, and training procedure as prior work [19, 110]. Given that we order *LRTS* chronologically, we use its first 75% (older builds) as training data to the ML algorithms, and evaluate the trained ML models on the remaining 25% (§2.4.1). Each data sample corresponds to a ⟨TSR, test⟩ pair, represented as a pair of a feature vector (consisting of features in Table 1) and test outcome. Given a TSR R, LTR TCP predicts the probability of failure for each test t in R based on $\langle R, t \rangle$'s feature vector, then prioritizes tests that have higher probabilities. As in prior work, we use gradient boosting regression model as the ML algorithm, and its lightGBM implementation from scikit-learn [19, 47, 94]; we use default hyper-parameter values provided by the scikit-learn package for training [13, 19, 110].

4.3.4 RTL TCP. We use the released implementations of RL agents and reward functions [35] for RTL TCP from Spieker et al. [98], as in prior RTL TCP studies [6, 9, 78]. We evaluate RTL techniques with the same hyper-parameter values as prior work [6, 9, 78], and new values that double the number of hidden layers and training iterations for neural network agent to account for the larger test suites. The effectiveness of different hyper-parameter configurations is similar [98]; we present the best one.

4.4 Experimental Procedure

We use 3 *LRTS* versions to study TCP: (1) *LRTS-All* keeps all test failures, (2) *LRTS-DeConf* omits identified confounding test failures, (3) *LRTS-FirstFail* only keeps the first failure of each non-flaky test over the collected builds of a stage. Table 6 lists the number of failed TSRs for evaluation per version. Each technique has its data collection and possible training done only on *LRTS-All*, then we directly evaluate its effectiveness on all versions.

Table 6: Dataset versions.

Version	#Failed TSR
LRTS-All	30,118
LRTS-DeConf	9,683
LRTS-FirstFail	2,076

To reduce randomness in the experiments, as prior work [19, 20, 98], we ran each non-LTR technique 10 times (with 10 random seeds) on each TSR of each project on each dataset version. For the LTR techniques, we trained the ML algorithm on the same training data of each project 10 times to obtain 10 ML models per project. We also evaluate a randomized TCP technique (denoted as *Random*) to serve as a baseline, which randomly shuffles all tests.

In total, we evaluated 59 TCP techniques: 26 basic techniques, of which 25 are described in §2.1-2.4, and the randomized baseline; and 33 hybrid techniques, of which 17 use *CC* hybrid approach and 16 use *CCH* hybrid approach. Applying hybrid approaches to a basic TCP with the same heuristic provides little value, e.g., applying *CC* to *QTF-Last*—we thus omit these combinations. We also omit applying hybrid approaches to RTL TCP because it solely learns from pre-defined states, actions, and rewards during runtime. Adding external heuristics would interfere with the learning process.

5 Evaluation

We aim to answer the following research questions:

- **RQ1:** How do different TCP techniques perform in detecting real test failures on long-running test suites from recent builds?
- **RQ2:** How do failures of flaky tests and frequently failing tests impact the effectiveness of different TCP techniques?
- **RQ3:** How do TCP techniques perform in detecting the first failure throughout CI history for each failed test?

Table 7 summarizes the revisited and new findings in our study. For each revisited finding throughout this section, we describe our expectation of its potential outcome, the actual outcome, the experiment results, and our analyses.

5.1 RQ1: Effectiveness of TCP Techniques

This RQ compares different TCP techniques on *LRTS-DeConf* that omits confounding test failures. In Figure 2, each box plot shows the distribution of APFDc or APFD values for each technique. For non-learning-based techniques, the values are from all failed TSRs; for learning-based techniques, the values are from failed TSRs of the latest 25% of the builds as the older 75% are used for training (and should not be used for evaluation [19, 63, 84, 102, 110]). Each box plot represents 100 (10*10) values, for 10 projects and 10 experiment

Table 7: List of findings in our study. A finding from prior work is marked "✓" if our study confirms the same on *LRTS*; it is marked "✗" if a finding differs in our study from that of prior work. New findings are marked "💡".

F1 Different failure-to-fault mappings lead to similar ranking of TCP techniques [83].	✓
F2 APFD can be misleading and give different ranking of TCP techniques than APFDc [13, 64].	✓
F3 Basic time-based and history-based techniques can rival or outperform sophisticated IR-based and learning-based techniques [19, 83].	✓
F4 All IR-based techniques perform worse than time-based and history-based techniques [83].	✗
F5 Different configurations have little impact on the effectiveness of IR-based techniques [83, 92].	✗
F6 LTR TCP is among the most effective TCP techniques when training with all available features [19].	✓
F7 In LTR TCP, training with all features (F_{all}) outperforms every individual feature set; execution time and outcome features (F_1) outperform associated history and similarity features (F_2 and F_3) which outperform change features (F_4) [19, 110].	✓
F8 RTL techniques generally perform better than random [98] but worse than LTR techniques [9].	✓
F9 Cost-cognizant hybrid TCP approaches can substantially improve the effectiveness of basic TCP techniques [83].	✓
F10 Among all techniques, hybrid perform the best [83], specifically techniques that combine time-based and history-based heuristics.	✓
F11 Techniques that rely on test outcome frequency, e.g., *MostFail* and *LTR* (F_1), are heavily impaired by confounding test failures [21].	✓
F12 Techniques that favor more recent test history, e.g., *LatestFail* and *RTL (NN-TestFail)*, are resilient to confounding test failures.	💡
F13 Time-based and change-aware techniques, e.g., IR-based, are the least affected by the presence of confounding test failures.	💡
F14 Time-based and change-aware techniques are effective in finding the first failures of tests, followed by *Random*, then history-based.	💡

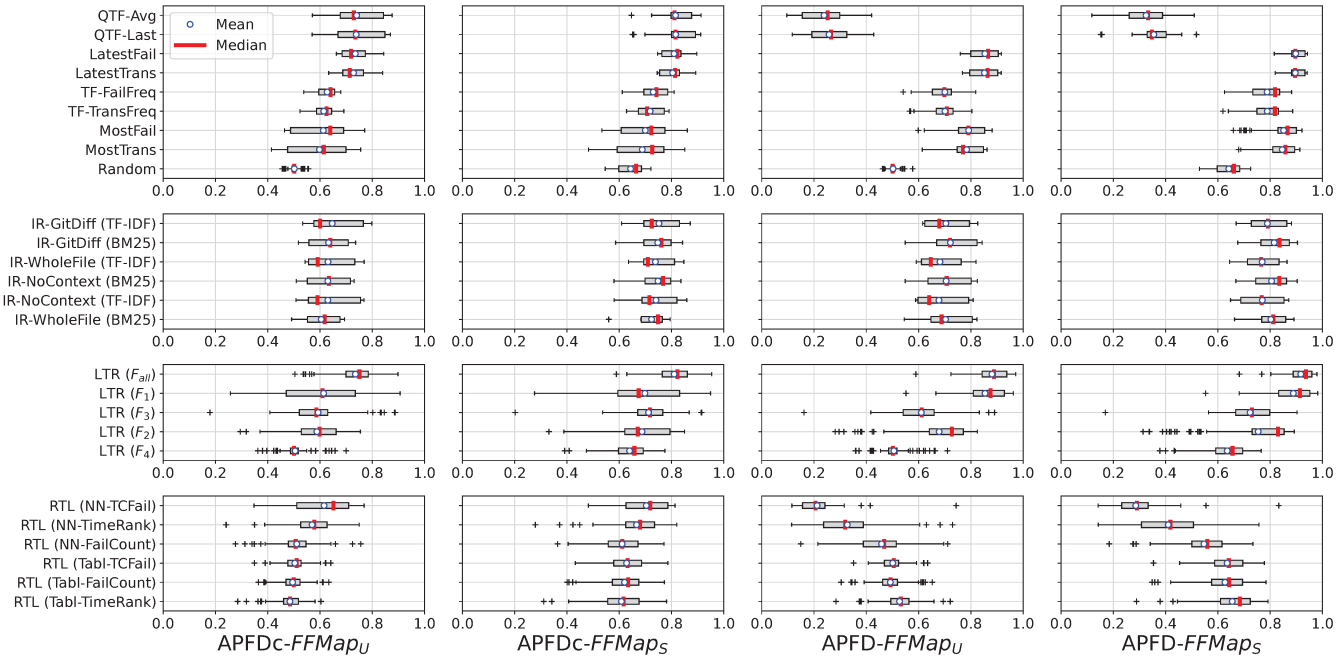

Figure 2: Evaluation results of TCP techniques on *LRTS-DeConf*. Four rows from top to bottom show the results of (1) time-based, history-based TCP, and *Random* baseline, (2) IR-based TCP, (3) LTR TCP, and (4) RTL TCP, respectively. Four columns from left to right show the value distributions of APFDc with $FFMap_U$ and $FFMap_S$, and APFD with $FFMap_U$ and $FFMap_S$, respectively. TCP techniques in each row are organized in the descending order by their mean APFDc-$FFMap_U$ values.

runs. We use average values across TSRs in each project to weigh all projects equally regardless of the number of TSRs [9, 14, 83].

5.1.1 Evaluation Settings. For failure-to-fault mappings, because over 70% of the TSRs in *LRTS* have multiple failures, the $FFMap_S$ values are 20% higher than $FFMap_U$ values for each of APFDc and APFD (Figure 2). For the same reason, we also expected the ranking of some techniques to differ between mappings, e.g., when test suites are larger (§3.5), it is more likely that a technique A puts failed tests at both ends of the TSR, while a technique B puts them in the middle, so A and B would be ranked differently across the two mappings. However, the ranking of all techniques is similar across mappings for each metric. By inspecting the prioritized positions of failed tests, we found that the similarity is because each TCP technique ran failed tests either early or late for all TSRs but rarely ran them in the middle (except *Random* where failed tests appeared anywhere with uniform probability). Our overall results confirm the prior finding (**F1** ✓), thus we only show results from one mapping ($FFMap_U$) in the following sections.

For metrics, prior work argued how APFD can be misleading because it does not consider the test execution time and could rank techniques greatly differently than APFDc [13, 14, 17, 64]. We expect the prior finding to not differ on test suites where tests run longer. Indeed, we confirm the prior finding (**F2** ✓). Figure 2

Table 8: APFDc-$FFMap_U$ results on *LRTS-DeConf*. Horizontal lines separate TCP technique categories.

TCP Technique	Basic			*CC*		*CCH*	
	Avg	G.Cat	G.All	Avg	Imp	Avg	Imp
QTF-Avg	.740	A	A	-	-	-	-
QTF-Last	.739	A	A	-	-	-	-
LatestFail	.735	A	A	.835	13%	.797	8%
LatestTrans	.728	A	A	.830	13%	.795	9%
TF-FailFreq	.627	B	BCD	.788	25%	.773	23%
TF-TransFreq	.614	B	BCD	.777	26%	.764	24%
MostFail	.613	B	BCDE	.773	26%	-	-
MostTrans	.598	B	CDE	.765	27%	.743	24%
Random	.502	C	F	-	-	-	-
IR-GitDiff (TF-IDF)	.647	A	B	.767	18%	.789	21%
IR-GitDiff (BM25)	.633	AB	BC	.743	17%	.771	21%
IR-WholeFile (TF-IDF)	.631	AB	BCD	.761	20%	.785	24%
IR-NoContext (BM25)	.630	AB	BCD	.741	17%	.770	22%
IR-NoContext (TF-IDF)	.630	AB	BCD	.758	20%	.784	24%
IR-WholeFile (BM25)	.605	B	BCDE	.739	22%	.767	26%
LTR (F_{all})	.736	A	A	.809	9%	.781	6%
LTR (F_1)	.614	B	BCD	.767	24%	.739	20%
LTR (F_3)	.593	B	CDE	.706	19%	.741	24%
LTR (F_2)	.588	B	DE	.727	23%	.735	24%
LTR (F_4)	.505	C	F	.717	41%	.747	47%
RTL (NN-TCFail)	.616	A	BCD	-	-	-	-
RTL (NN-TimeRank)	.570	B	E	-	-	-	-
RTL (NN-FailCount)	.511	C	F	-	-	-	-
RTL (Tabl-TCFail)	.504	C	F	-	-	-	-
RTL (Tabl-FailCount)	.495	C	F	-	-	-	-
RTL (Tabl-TimeRank)	.485	C	F	-	-	-	-

shows that the ranking of many techniques is similar across APFDc and APFD, but the ranking of time-based and RTL techniques are opposite. We thus focus on APFDc results in the following sections.

5.1.2 Analysis of Basic TCP Techniques. As prior studies [9, 13, 14, 19, 83, 110], we perform statistical tests on APFDc-$FFMap_U$ values to analyze the effectiveness difference across different techniques. We first perform a one-way ANOVA analysis and find that the APFDc values across techniques significantly differ (*p-value* < 0.001). We then perform Tukey HSD test as a post-hoc test [103], which assesses the difference and puts techniques into different groups if their APFDc values differ significantly [13, 59, 62, 83]. Groups are named by letters: "A" represents the best group, and the effectiveness degrades alphabetically. A technique with multiple letters performs in between these letter groups.

In Table 8, "Basic" columns show the results of basic techniques; "*CC*" and "*CCH*" columns show the results of hybrid techniques after applying *CC* and *CCH* hybrid approaches, respectively. "Avg" shows the mean APFDc values; "G.Cat" and "G.All" show the effectiveness group from Tukey HSD test within each basic TCP category and across all basic techniques, respectively; "Imp" columns show improvement from "Basic" values to hybrid values.

Time-based and history-based TCP. Prior studies have shown the effectiveness of sophisticated IR and ML TCP techniques in industrial settings [10, 12, 13, 100, 107], while more recent studies showed the simplest time-based and history-based techniques are equally effective on short-running test suites [19, 83]. Because longer-running test suites have different characteristics (§3.5), we expect that the simplest time-based and history-based techniques may perform worse than sophisticated techniques.

However, our evaluation confirms the prior finding from more recent studies that time-based and history-based techniques can match and often outperform the sophisticated IR and ML techniques (**F3** ✓). Table 8 shows that *QTF-Avg* and *QTF-Last* achieve the top-2 highest mean APFDc (0.740 and 0.739). Among history-based techniques, prioritizing recently failed or transitioned tests (*LatestFail* and *LatestTrans*) have the highest APFDc (0.735 and 0.728). They are also in the best effectiveness group with the time-based techniques and one LTR technique (group A).

To understand why QTF is the most cost-effective, we first analyzed the positions of failed tests in TSRs after *QTF-Avg* prioritization. We found that failed tests run much longer than the majority of the tests in their TSRs—75% of the failed tests in *LRTS-DeConf* are in 76% or later positions of their TSRs; on average, failed tests are in 83% positions of their TSRs. APFD values in Figure 2 are very low for QTF. We then study why QTF performs well even when it orders failed tests late. It turns out that long-running test suites commonly have a number of tests that run substantially longer than others, e.g., tens of minutes. These tests are often end-to-end and integration tests that largely contribute to a TSR's duration, but QTF runs them last. For example, `TestYarnNativeServices` from Hadoop runs for 15 minutes (i.e., 4.5% of Hadoop's average TSR duration) to start mini-clusters and test deploying services [32].

IR-based TCP. IR-based techniques have been shown to often outperform time-based and history-based techniques on short-running test suites [83]. We expected the prior finding to stand on *LRTS*, because test method bodies in long-running test suites are larger, and IR TCP is effective precisely because it captures textual relationships between documents [92].

Contrary to our expectation, however, our evaluation results refute the prior finding (**F4** ✗). In Table 8, the best IR-based technique, i.e., *IR-GitDiff (TF-IDF)*, achieves a mean APFDc of 0.647 in group B, worse than the 4 time-based and history-based techniques in group A that all have APFDc above 0.727. Our results also refute that IR model and query context length configuration substantially impact the effectiveness of IR-based TCP [83] (**F5** ✗). Figure 2 shows all 6 IR-based techniques have similar distributions; Table 8 shows that all 6 techniques differ by at most one effectiveness group, while 4 of them perform statistically the same (group AB).

To understand why IR-based TCP's effectiveness differs from prior work, we first explore the difference between *LRTS-DeConf* and the prior IR TCP dataset (denoted as IRDataset) [1, 83]. In *LRTS-DeConf* TSRs, the average duration and number of failures are 76 and 2 times larger, respectively, while the average code change size is 20% *smaller*. We then perform controlled experiments on each of these variables in *LRTS-DeConf* (selecting TSRs by the percentile ranges of each variable) for all basic IR-based techniques.

Table 9: IR experiment.

Variable	Variable Value Range			
	<Q1	Q1-2	Q2-3	>Q3
Duration	.644	.642	.628	.605
#Failure	.679	.672	.640	.569
Fail ratio	.693	.686	.607	.577
Chg size	.617	.612	.632	.648

Table 9 shows our experiment results; each cell is the APFDc-$FFMap_U$ values averaged across all IR-based techniques. From Table 9, we observed that IR-based techniques: (1) perform worse when TSRs have longer durations; (2) perform worse when TSRs have more failures ("#Failure") or more failures relative to test suite size ("Fail ratio"); and (3) perform better when code changes are

larger. These results suggest that IR-based techniques performing worse in our study is likely due to *LRTS* having longer-running test suites with more failures. Another possible reason is that smaller code change query in *LRTS* has less information, which leads to a lower retrieval accuracy [83].

In addition, we argue that textual similarity is not the perfect indicator of test failure probability, and the outcome of such imprecision can be amplified on longer-running test suites. By inspecting IR-prioritized TSRs in *LRTS-DeConf*, we saw that IR scores of different tests often differ marginally (e.g., less than 0.0001 in cosine similarity), while their durations have much bigger difference, especially on long-running test suites (e.g., standard deviation of test duration in *LRTS* is 15 times larger than that of IRDataset). Thus, even a minor IR score difference can substantially impact failure finding effectiveness—during inspection, we often saw that a failed test class is delayed for hundreds of seconds behind some passed tests because its code has some more or fewer tokens.

Learning-based TCP. We expected LTR TCP to be competitive on long-running test suites as LTR techniques can model a large amount of test and change data. Indeed, Table 8 shows *LTR (F_{all})* is in the best effectiveness group (group A), with the third highest mean APFDc (0.736) across all techniques (**F6** ✓). LTR is effective because its supervised learning algorithm learns which feature(s) can minimize test outcome prediction loss from historical builds at training time, and uses those features more often on unseen builds at inference time. We expected F_{all} to outperform individual feature sets, as using more features is often better in ML, but we have no expectation on the ranking of individual sets. Our results confirm prior finding (**F7** ✓): using all features (F_{all}) is the best in LTR; test time and history features (F_1) are better than similarity features (F_2, F_3) which are better than change features (F_4).

Compared to supervised learning (LTR), reinforcement learning (RTL) has been shown harder to optimize due to its large search space and random exploration, which leads to unstable TCP effectiveness [9, 98]. We thus expected, and confirm that, while some RTL techniques are certainly better than *Random* [98], they are usually outperformed by LTR techniques [9] (**F8** ✓). Table 8 shows that 2 RTL techniques are in better groups than *Random*, and 4 other RTL techniques are the same as *Random*. Most RTL techniques are in worse groups than most LTR techniques.

We also evaluated effectiveness degradation with time and found it only for LTR techniques in 4/10 projects, likely due to the common ML issue of distribution shift, where data of the latest builds become less similar to the older builds used for training. Because many LTR/RTL techniques perform no better than simpler techniques, while requiring elaborated effort to develop (feature engineering) and maintain (retraining) [19, 80, 110], we recommend time-based and history-based techniques over current learning-based ones.

5.1.3 Analysis of Hybrid TCP Techniques. We expected the evaluated cost-cognizant hybrid approaches to only marginally improve basic TCP techniques on longer-running test suites (based on §5.1.2). To our surprise, they lead to a much bigger improvement because of the high cost-effectiveness of test time and outcome heuristics as observed from basic time-based and history-based techniques (**F9** ✓). Table 8 shows *CC* and *CCH* approaches improve the mean APFDc of basic techniques by 9%-41% and 6%-47%, respectively. Table 8 further shows that fusing heuristics from the best basic TCP techniques *QTF-Last* and *LatestFail* gives *LatestFail+CC* that achieves the highest APFDc (0.835) among all techniques (**F10** ✓).

5.2 RQ2: Impact of Confounding Test Failures

Recent studies use real CI datasets that have confounding test failures [9, 19, 67, 110, 112], and detecting these failures earlier in TCP may provide no value to the developers [21, 51, 82, 83, 110]. However, there is very limited evaluation on the effectiveness of TCP techniques under the impact of confounding test failures [21, 82, 83]—prior work has only studied how flaky tests impacted one time-based, two history-based, and a few IR-based techniques on short-running test suites [83] or single-project dataset [21]. In this RQ, we aim to provide a broader investigation on a wider range of TCP techniques under the impact of confounding test failures. Compared to prior work, we evaluate 3 times more TCP techniques on a 10-project dataset (with the first evaluation of LTR and RTL TCP), and consider both flaky tests and frequently-failing tests.

Following prior work [21], we evaluate TCP techniques on two versions of the dataset—one version considers confounding test failures as *relevant failures* that need to be investigated (*LRTS-All*), while another version does not (*LRTS-DeConf*). We then compare the evaluation results between both versions.

We perform the same statistical analysis as in RQ1 (§5.1.2) and present our results in Table 10, which compares the mean APFDc values and effectiveness group of techniques between *LRTS-DeConf* and *LRTS-All* (it also presents results on *LRTS-FirstFail*, which we discuss in the next RQ). The top-5 techniques, with the highest APFDc values, on each dataset version are bolded.

We expected TCP techniques that rely on calculating test outcome frequency to be the most impaired by confounding test failures, because failure count can easily include confounding test failures. From *LRTS-All* to *LRTS-DeConf* in Table 10, we indeed observe significant drops in the ranking and APFDc values for techniques using test outcome frequency, e.g., *MostFail* and *LTR (F_1)*, which confirms the prior finding [21, 83] (**F11** ✓).

However, not all history-based techniques are heavily impacted by confounding test failures—in Table 10, *LatestFail*, *LatestTrans*, and *LTR (F_{all})* are in top-5 on both *LRTS-All* and *LRTS-DeConf*. Our results show that techniques that account for recent history (either by updating heuristic with recent builds or by weighing with other features) are resilient (**F12** 💡).

We also find that time-based and change-aware techniques are the least impacted by confounding test failures (**F13** 💡). From *LRTS-All* to *LRTS-DeConf*: QTF techniques rise to the best with large increases in APFDc; IR-based techniques also have higher APFDc. Overall, we recommend *LatestFail*, QTF, and *LTR (F_{all})* as they outperform others when properly accounting for confounding test failures, and *LTR (F_{all})* should be checked to not overly rely on outcome frequency features.

5.3 RQ3: Effectiveness on First Failures

Failing builds are relatively common in practice [21, 82]. For example, 52% of the TSRs (and 75% of the CI builds) in *LRTS* fail. Accordingly, uncommon failures, such as failures from tests that have been passing, may be more worthy of developer's attention

Table 10: Mean APFDc-$FFMap_U$ and effectiveness group of TCP techniques on all three versions of *LRTS*.

TCP Technique	*LRTS-DeConf*		*LRTS-All*		*LRTS-FirstFail*	
QTF-Avg	**.740**	**A**	.671	CD	**.796**	**A**
QTF-Last	**.739**	**A**	.677	CD	**.798**	**A**
LatestFail	**.735**	**A**	**.795**	**A**	.467	DE
LatestTrans	**.728**	**A**	**.788**	**A**	.464	DEF
TF-FailFreq	.627	BCD	.666	CD	.440	EF
TF-TransFreq	.614	BCD	.656	D	.422	F
MostFail	.613	BCDE	**.720**	**B**	.312	G
MostTrans	.598	CDE	.701	BC	.313	G
Random	.502	F	.502	I	.504	D
IR-GitDiff (TF-IDF)	.647	B	.589	EF	**.691**	**B**
IR-GitDiff (BM25)	.633	BC	.576	FG	.667	BC
IR-WholeFile (TF-IDF)	.631	BCD	.576	FG	**.679**	**B**
IR-NoContext (BM25)	.630	BCD	.579	FG	.666	BC
IR-NoContext (TF-IDF)	.630	BCD	.583	EFG	**.680**	**B**
IR-WholeFile (BM25)	.605	BCDE	.557	FG	.632	C
LTR (F_{all})	**.736**	**A**	**.764**	**A**	-	-
LTR (F_1)	.614	BCD	**.724**	**B**	-	-
LTR (F_3)	.593	CDE	.548	GH	-	-
LTR (F_2)	.588	DE	.618	E	-	-
LTR (F_4)	.505	F	.505	I	-	-
RTL (NN-TCFail)	.616	BCD	.549	GH	-	-
RTL (NN-TimeRank)	.570	E	.516	HI	-	-
RTL (NN-FailCount)	.511	F	.481	I	-	-
RTL (Tabl-TCFail)	.504	F	.508	I	-	-
RTL (Tabl-FailCount)	.495	F	.501	I	-	-
RTL (Tabl-TimeRank)	.485	F	.517	HI	-	-

as they are more likely due to recent change. Moreover, although history-based techniques have been shown effective (e.g., *Latest-Fail*), they often rely on *failure* history that is only informative for tests that had failed. But many tests often may not fail, e.g., 67% of the executed tests in *LRTS* had never failed. It is important to know how techniques prioritize failing tests that have no prior failures. This RQ thus studies the effectiveness of TCP techniques in detecting the first failure of each test in our CI history.

We evaluate on *LRTS-FirstFail* that only keeps the first failure of each non-flaky test. The first failures are with respect to the *entire* CI history, not the failures that transition a test suite from passing to failing [42, 79]. We omit learning-based techniques, because the latest 25% of the builds used for evaluating learning-based TCP have insufficient first failures to make generalizable observations.

The *LRTS-FirstFail* columns in Table 10 show that all history-based techniques perform as *Random*, because they prioritize tests based on failures, so tests without prior failures are prioritized randomly. In fact, history-based techniques are even worse than *Random* on a build if all previously failed tests pass but a new test fails. But they can be better than *Random* on a build if both a new test and some previously failed tests fail.

Time-based TCP remains the most effective in this RQ (**F14** 💡). Overall, our study has shown that *the simplest* QTF stands as the most cost-effective TCP technique across different RQs we studied. IR-based techniques also outperform *Random* when they prioritize tests similar to the change, which indicates that test failures in *LRTS-FirstFail* are more often related to current changes compared to *LRTS-All* or *LRTS-DeConf*. Our results motivate novel TCP techniques that lexicographically prioritize tests by history-based heuristics, and use time-based or IR-based to break ties.

6 Threats to Validity

External. The threats to external validity lie in the generalizability of our study. We use real build data from a heterogeneous set of projects. We evaluate on a large number of CI runs with statistical analyses as prior work [13, 19, 59, 62, 83, 110]. To reduce threat from (1) the evaluated TCP techniques, we use the same TCP data collection [19, 83, 98], settings, and implementations as prior work [9, 19, 83, 98, 110]; (2) randomness, we run all experiments 10 times [9, 19, 98]; (3) flaky tests [19, 63, 110], we perform both manual inspection and automated filtering (§3.4). Due to high cost of running test suites, we do not run the generated test orders [105].
Internal. The main threats to internal validity lie in the potential bugs of our techniques and experimental scripts. To address such threats, we regularly check the collected data and our experimental results with unit tests and manual examination.

7 Related Work

TCP Techniques. TCP has been extensively studied as summarized in several surveys [18, 29, 36, 59, 61, 80, 89, 111]. Besides the techniques in §2, prior work has also proposed techniques based on code coverage [89, 90], adaptive random testing [41], constraint solving [113], and genetic algorithms [54]. TCP has been applied to mutation testing [114], fault localization and repair [22, 27, 57, 86], testing configurable systems [14, 87, 99], and deep neural networks [81, 108]. We focus on studying techniques most widely used in recent work [9, 19, 21, 83, 98, 110].
TCP Datasets and Studies. TCP datasets are crucial for studying TCP techniques. Mattis et al. [67] listed TCP datasets prior to 2020 and released RTPTorrent that curated real CI builds from 20 projects via TravisTorrent. Prior to RTPTorrent, only 18 TCP datasets entirely consisted of real CI builds, and only two of them made their TSR data available [33, 98]. TCP studies in industrial settings exist but provide limited data for future work [10, 18, 63, 65, 101, 106]. Recent studies on open-source datasets extend RTPTorrent with proprietary projects [19, 65], and some collect their own TCP datasets from more Travis CI Java projects [6, 9, 21, 79, 83, 110].

8 Conclusion

We present *LRTS*, an extensive dataset focusing on recent, long-running test suites with 21,255 CI builds and 57,437 test-suite runs (average duration of 6.5 hours) of 10 large-scale, open-source projects that use Jenkins CI. On *LRTS*, we evaluate the effectiveness of 59 techniques from 5 leading TCP technique categories on longer-running test suites and on prioritizing tests with no prior failure. We also study the impact of confounding test failures on these techniques. Our study both revisits major findings (9 confirmed and 2 refuted) from prior work and establishes 3 new findings on the effectiveness and ranking of TCP techniques. We show that the best techniques combine the simplest time-based and history-based heuristics, e.g., prioritizing faster tests that have failed recently.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by NSF grants CCF-1763788, CCF-1956374, and CNS-2238045. We acknowledge support for research on testing from C3.ai, IBM, Microsoft, and Qualcomm.

References

[1] 2023. IR-Based TCP Dataset. https://sites.google.com/view/ir-based-tcp.
[2] Mohamed Abdelkarim and Reem ElAdawi. 2022. TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks. In *ICSTW*.
[3] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the Effectiveness of Test Suite through Mining Historical Data. In *MSR*.
[4] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2015. Striving for Failure: An Industrial Case Study about Test Failure Prediction. In *ICSE*.
[5] Apache 2023. Apache Software Foundation. https://www.apache.org/.
[6] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement Learning for Test Case Prioritization. *TSE* 48 (2021).
[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *MSR*.
[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *MSR*.
[9] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *ICSE*.
[10] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *ESEC/FSE*.
[11] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact. In *ASE*.
[12] Ryan Carlson, Hyunsook Do, and Anne Denton. 2011. A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study. In *ICSM*.
[13] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *ESEC/FSE*.
[14] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *ISSTA*.
[15] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077
[16] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. 2004. Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. In *ISSRE*.
[17] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *ICSE*.
[18] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *ESEC/FSE*.
[19] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *ISSTA*.
[20] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical Evaluation of Pareto Efficient Multi-Objective Regression Test Case Prioritisation. In *ISSTA*.
[21] Emad Fallahzadeh and Peter C Rigby. 2022. The Impact of Flaky Tests on Historical Test Prioritization on Chrome. In *ICSE-SEIP*.
[22] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*.
[23] Git 2023. Git Diff Documentation. https://git-scm.com/docs/git-diff#Documentation/git-diff.txt--Ultngt.
[24] GitHub 2022. GitHub API - compare two commits. https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28#compare-two-commits.
[25] GitHub 2022. GitHub API - pulls. https://docs.github.com/en/rest/pulls?apiVersion=2022-11-28.
[26] GitHub 2023. GitHub Workflow Limits. https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration.
[27] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan JC van Gemund. 2010. Prioritizing Tests for Software Fault Localization. In *QSIC*.
[28] Gradle 2023. Gradle. https://gradle.org/.
[29] Renan Greca, Breno Miranda, and Antonia Bertolino. 2023. State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review. *Comput. Surveys* (2023).
[30] Sander Greenland, Judea Pearl, and James M Robins. 1999. Confounding and Collapsibility in Causal Inference. *Statistical science* 14 (1999).
[31] Hadoop 2023. Hadoop CI Server. https://ci-hadoop.apache.org/job/hadoop-multibranch/.
[32] Hadoop 2023. TestYarnNativeServices. https://github.com/apache/hadoop/blob/trunk/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-applications/hadoop-yarn-services/hadoop-yarn-services-core/src/test/java/org/apache/hadoop/yarn/service/TestYarnNativeServices.java.
[33] Alireza Haghighatkhah, Mika Mantyla, Markku Oivo, and Pasi Kuvaja. 2018. Test Prioritization in Continuous Integration Environments. *Journal of Systems and Software* 146 (2018).
[34] HBase 2023. HBase Flaky Tests Dashboard. https://nightlies.apache.org/hbase/HBase-Find-Flaky-Tests/master/1208/output/dashboard.html.
[35] HelgeS 2017. RETECS. https://bitbucket.org/HelgeS/retecs/src/master.
[36] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-box and Black-box Test Prioritization. In *ICSE*.
[37] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *ASE*.
[38] Hive 2023. Hive Flaky Tests Dashboard. http://ci.hive.apache.org/job/hive-flaky-check.
[39] Jenkins 2023. Jenkins Pipeline Syntax. https://www.jenkins.io/doc/book/pipeline/syntax/#parallel.
[40] Jenkins 2023. Jenkins Remote Access API . https://www.jenkins.io/doc/book/using/remote-access-api/.
[41] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. 2009. Adaptive Random Test Case Prioritization. In *ASE*.
[42] Xianhao Jin and Francisco Servant. 2020. A Cost-efficient Approach to Building in Continuous Integration. In *ICSE*.
[43] JIRA 2023. JIRA Fuzzy Search. https://confluence.atlassian.com/jirasoftwareserver/advanced-searching-939938733.html.
[44] JIRA 2023. JIRA Issue. https://issues.apache.org/jira/issues.
[45] Kafka 2023. Kafka CI Server. https://ci-builds.apache.org/job/Kafka.
[46] Kafka 2023. Kafka Jenkinsfile. https://github.com/apache/kafka/blob/7d39d7400c919a519fb73d93e311eba9b13bbb97/Jenkinsfile#L101.
[47] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. *NeurIPS* 30 (2017).
[48] Jung-Min Kim and Adam Porter. 2002. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *ICSE*.
[49] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *RCoSE*.
[50] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-Level Test Case Prioritization Using Machine Learning. In *ICMLA*.
[51] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *ISSTA*.
[52] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-based Test Selection Algorithms at Google. In *ICSE-SEIP*.
[53] Feng Li, Jianyi Zhou, Yinzhu Li, Dan Hao, and Lu Zhang. 2021. AGA: An Accelerated Greedy Additional Algorithm for Test Case Prioritization. *TSE* 48 (2021).
[54] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *TSE* 33 (2007).
[55] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *ICSE*.
[56] Jackson A Prado Lima and Silvia Regina Vergilio. 2020. A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments. *TSE* 48 (2020).
[57] Yiling Lou, Samuel Benton, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. How Does Regression Test Selection Affect Program Repair? An Extensive Study on 2 Million Patches. *arXiv:2105.07311* (2021).
[58] LRTS 2024. *Dataset of Long-Running Test Suites.* https://doi.org/10.5281/zenodo.12662090
[59] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-World Software Evolution?. In *ICSE*.
[60] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *FSE*.
[61] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-Scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *FSE*.
[62] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. 2018. Assessing Test Case Prioritization on Real Faults and Mutants. In *ICSME*.
[63] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *ICSE-SEIP*.
[64] Alexey G. Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-Cognizant Test Case Prioritization.* Technical Report.
[65] Rezwana Mamata, Akramul Azim, Ramiro Liscano, Kevin Smith, Yee-Kang Chang, Gkerta Seferi, and Qasim Tauseef. 2023. Test Case Prioritization using Transfer Learning in Continuous Integration Environments. In *AST*.
[66] Dusica Marijan, Arnaud Gotlieb, and Abhijeet Sapkota. 2020. Neural Network Classification for Improving Continuous Regression Testing. In *AITest*.
[67] Toni Mattis, Patrick Rein, Falco Dursch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *MSR*.
[68] Maven 2023. Maven. http://maven.apache.org.
[69] Maven 2023. Maven Surefire rerunFailingTestsCount. https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html.
[70] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A Static Approach to Prioritizing JUnit Test Cases. *TSE* 38 (2012).

[71] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *ICSE-SEIP*.
[72] Ade Miller. 2008. A Hundred Days of Continuous Integration. In *Agile*.
[73] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *TOSEM* 11 (2002).
[74] Armin Najafi, Weiyi Shang, and Peter C Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *ICSE-SEIP*.
[75] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression Testing in the Presence of Non-code Changes. In *ICST*.
[76] Tanzeem Bin Noor and Hadi Hemmati. 2015. A Similarity-Based Approach for Test Case Prioritization Using Historical Failure Data. In *ISSRE*.
[77] Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying Test Case Failure Prediction for Test Case Prioritization. In *PROMISE*.
[78] Safa Omri and Carsten Sinz. 2022. Learning to Rank for Test Case Prioritization. In *SBST*.
[79] Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *ISSTA*.
[80] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. 2022. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. *ESE* 27 (2022).
[81] Zhonghao Pan, Shan Zhou, Jianmin Wang, Jinbo Wang, Jiao Jia, and Yang Feng. 2022. Test Case Prioritization for Deep Neural Networks. In *DSA*.
[82] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *TOSEM* 31 (2021).
[83] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *ISSTA*.
[84] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagappan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *ICSE*.
[85] pytest 2023. pytest-rerunfailures. https://pypi.org/project/pytest-rerunfailures.
[86] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair Through Fault-Recorded Testing Prioritization. In *ICSM*.
[87] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *ISSTA*.
[88] Stephen E. Robertson, Steve Walker, and Micheline Beaulieu. 2000. Experimentation As a Way of Life: Okapi at TREC. *Information processing & management* 36 (2000).
[89] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*.
[90] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases for Regression Testing. *TSE* 27 (2001).
[91] David Saff and Michael D Ernst. 2003. Reducing Wasted Development Time via Continuous Testing. In *ISSRE*.
[92] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *ICSE*.
[93] Gerard Salton and Christopher Buckley. 1988. Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing & Management* 24 (1988).
[94] scikit-learn 2023. sklearn.ensemble.HistGradientBoostingRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingRegressor.html.
[95] Aizaz Sharif, Dusica Marijan, and Marius Liaaen. 2021. DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing. In *ICSME*.
[96] Mark Sherriff, Mike Lake, and Laurie Williams. 2007. Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records. In *ISSRE*.
[97] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *ISSTA*.
[98] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *ISSTA*.
[99] Hema Srikanth, Myra B Cohen, and Xiao Qu. 2009. Reducing Field Failures in System Configurable Software: Cost-Based Prioritization. In *ISSRE*.
[100] Hema Srikanth, Charitha Hettiarachchi, and Hyunsook Do. 2016. Requirements Based Test Prioritization Using Risk Factors: An Industrial Study. *Information and Software Technology* (2016).
[101] Per Erik Strandberg, Wasif Afzal, Thomas J Ostrand, Elaine J Weyuker, and Daniel Sundmark. 2017. Automated System Level Regression Test Prioritization in a Nutshell. *Software* 34 (2017).
[102] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *ICSE*.
[103] John W Tukey. 1949. Comparing Individual Means in the Analysis of Variance. *Biometrics* (1949).
[104] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *FSE*.
[105] Hao Wang, Pu Yi, Jeremias Parladorio, Wing Lam, Darko Marinov, and Tao Xie. 2024. Hierarchy-Aware Regression Test Prioritization. In *ISSRE*.
[106] Shuai Wang, Shaukat Ali, Tao Yue, Øyvind Bakkeli, and Marius Liaaen. 2016. Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search. In *ICSE-Companion*.
[107] Shuai Wang, David Buchmann, Shaukat Ali, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. 2014. Multi-Objective Test Prioritization in Software Product Line Testing: An Industrial Case Study. In *SPLC*.
[108] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis. In *ICSE*.
[109] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multilevel Test-to-Code Traceability Links. In *ICSE*.
[110] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C Briand. 2022. Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. *TSE* 49 (2022).
[111] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *STVR* 22 (2012).
[112] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. In *FSE*.
[113] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-Aware Test-Case Prioritization using Integer Linear Programming. In *ISSTA*.
[114] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In *ISSTA*.
[115] Jianyi Zhou, Junjie Chen, and Dan Hao. 2021. Parallel Test Prioritization. *TOSEM* 31 (2021).
[116] Zhi Quan Zhou, Chen Liu, Tsong Yueh Chen, TH Tse, and Willy Susilo. 2020. Beating Random Test Case Prioritization. *Transactions on Reliability* 70 (2020).

Received 2024-04-12; accepted 2024-07-03