# REGRESSION TEST PRIORITIZATION FOR MODERN SOFTWARE

BY

RUNXIANG CHENG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

Doctoral Committee:

Professor Darko Marinov, Chair
Associate Professor Lingming Zhang
Assistant Professor Tianyin Xu
Associate Professor Wei Yang, The University of Texas at Dallas

**Abstract**

Continuous Integration (CI) is a common practice in software development for increasing code quality. CI runs regression test suite on each code change to help software developers find faults in the change. As codebase size and code change frequency have grown rapidly in modern software development, regression test suite runtime has also increased, thereby preventing timely debugging feedback to developers.

Regression Test Prioritization (RTP) aims to expose faults in code change sooner by reordering tests in the test suite, so that the ones likely to fail are run earlier. Despite that RTP has been extensively studied for nearly three decades, research results of RTP are rarely put into practice. To facilitate the adoption of RTP for modern software systems, RTP must be demonstrated effective in speeding up regression fault detection in relevant and practical testing settings. Moreover, robust and efficient RTP tooling must be available for practitioners to easily apply RTP research results in popular software ecosystems. This dissertation takes a step forward in addressing these challenges via three lines of work.

First, this dissertation examines key research findings from prior RTP studies on a newly-proposed dataset of long-running test suites with real test failures and up-to-date CI practices. Through extensive evaluation, this dissertation identifies the most effective and efficient RTP techniques under the impact of realistic CI issues such as flaky test failures.

Second, this dissertation applies traditional RTP techniques for configuration testing, an emerging but critical testing scenario; it further proposes novel and effective RTP techniques for configuration testing. This dissertation assesses various factors that influence the effectiveness of RTP in speeding up misconfiguration detection, and provides detailed analyses and guidelines in applying RTP to configuration tests.

Third, this dissertation presents PYTEST-RANKING, a readily-usable open-source RTP tool for Python and its most popular testing framework Pytest. This dissertation realistically evaluates the efficacy of PYTEST-RANKING, demonstrating that PYTEST-RANKING is well integrated with Python/Pytest ecosystems, can be conveniently deployed into CI, has low overhead, and can detect faults substantially faster than baselines.

## Acknowledgments

Many people have helped me towards the completion of my Ph.D. in ways I am aware or unaware of. In this acknowledgment, I want to sincerely thank you for your love and support, making this journey wonderful and fulfilling. I also want to apologize to people who helped me but I have failed to mention here explicitly, you all have my gratitude for your help.

First and foremost, I want to thank my advisor, Darko Marinov. Darko's dedication to the field, sharpness in research, care of students' well-being, and pursuit of truthfulness and rigorousness, all have heavily influenced me, and continue to inspire me. I have learned many things about doing research from Darko, he has also changed how I see and do many other things beyond research. I want to thank Darko for keeping his faith in me through the highest high and lowest low of this journey, and his support to all the students around him. One detailed thing I greatly appreciate is how fast Darko replies to emails and messages, which must have changed the course of my life in many moments—there are many other things about Darko that I am thankful for, which I will not exhaust in this acknowledgment.

I want to thank Tianyin Xu, Lingming Zhang, and Owolabi Legunsen for their advising especially in my junior years. I want to thank Tianyin and Darko for admitting me into UIUC, along with the other Ph.D. admission members who were involved in this decision which is pivotal to me. Tianyin guided me through my junior years, he taught me research ethics and professionalism in conducting research, making sure that I was on the right track, and doing the right thing from the get-go. Tianyin has also been working with and helping me throughout my Ph.D.. I deeply thank him, Lingming, and Owolabi for teaching me the skills and mindset in being a good computer science researcher, in the most honest and effective ways possible, allowing ample space for me to learn and improve with feedback on both the big picture and the details. Owolabi and Lingming both taught me ways of researching, thinking, and writing when I was doing my first and second Ph.D. projects with them. I am fortunate to have gained valuable knowledge from a glimpse of their research mindsets and techniques.

I want to thank Reyhaneh Jabbarvand and Wei Yang, who kindly helped me in completing this work, despite that I have only involved them at a later stage of my Ph.D. journey. Reyhaneh has given me the opportunity to write funding proposals highly related to this work. Wei has given this work useful feedback and guidance, as well as valuable suggestions for its future work and my career development through our discussions.

I am fortunate to have had several internship experiences in industry during my Ph.D..

I want to thank Satish Chandra and Michele Tufano at Google for giving me a valuable opportunity to work with them on an exciting project. I am also fortunate to be surrounded and supported by all the team members who are kind and experienced, including Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Sherry Shi, Aaron Sun, and Daye Nam. I am thankful for the lessons and work habits that I have learned from them on doing research in industrial context. At TikTok, I thank Ping Zhou and Wei Tang for hosting my internship, giving me plenty of freedom and resources to explore interesting research topics, and the other team members, Yupeng Tang, Fei Liu, and Tongping Liu. At Meta, I thank Mrinmoy Ghosh, Mehmet Yunt, and Selman Yilmaz for their detailed guidance throughout my internship, helping me navigate smoothly through the infrastructure. I also want to thank the rest of the team for continuing to work with and help me tremendously in the research project: Chris Cai, Xiaodong Wang, Rahul Mitra, Malay Bag, Menglu Yu, Taylor Robie, and Srinath Mandalapu. I also thank Ting Dai and Sai Zeng at IBM Research for giving me my first industry research internship experience when I was a junior Ph.D. student, during the special time of COVID. I want to wholeheartedly thank these people above for working on research projects and sharing their knowledge with me in cutting-edge industrial environments, giving me exposure to unique experiences that could be hard to obtain anywhere else.

The computer science Ph.D. program at UIUC provides a solid foundation that makes my Ph.D. journey possible. I want to thank Reza Farivar and Craig Zilles for giving me the opportunity to be a graduate teaching assistant for their courses, from which I have gained teaching and project mentoring experience. I want to thank the advisors from the graduate advising office from the computer science department, including but not limited to Jennifer Comstock, Viveka Kudaligama, and Kara MacGregor, for their timely support on issues I encountered throughout the years of my Ph.D.. I am thankful for the National Science Foundation (NSF) grants that supported this work (CCF-1763788, 1763906, 1816615, 1942430, 1956374, 2029049, and CNS-1740916, 1956007, 2238045), and support from Meta, Google, IBM, Futurewei, Microsoft, C3.ai, and Qualcomm. I thank the ACM International Symposium on Software Testing and Analysis (ISSTA) and International Conference on the Foundations of Software Engineering (FSE) for publishing chapters of this work; and the other prestigious conferences and journals for publishing my other research done in Ph.D..

The experience I have with my UIUC peers and collaborators is a large and essential part of my Ph.D.. Although the unexpected events from COVID and family circumstance have changed how I interacted with many of these people in the middle of my journey, the experience remains a source of inspiration and strength to me. I am honored and grateful that I could share moments with them, know about them, and learn from them. To start, I want to acknowledge people in my first Ph.D. project, Xudong Sun, Jack Chen, Ran Ang.

The first conference practice talk I attended at UIUC was from August Shi, whom I thank for setting up a great example that I learned from continuously since then. I want to thank Wing Lam for showing me the ropes of practicing software engineering and testing research, after the rest of Darko's research group had graduated and became faculties, before he also became a faculty. I also thank my peers who have subsequently joined Darko's group, whom I have meaningful research discussions and collaborations with, Shuai Wang, Sam Grayson, Kaiyao Ke, Yicheng Ouyang, Erkai Yu; as well as some folks from systems research for the same reason, Gangmuk Lim, Xinyu Lian, Siyuan Chai, Jinghao Jia. I want to deeply thank the Brett Daniel Software Engineering Seminar, and the people who participated in it. I am grateful that there is such an effective and transparent platform for me to receive feedback from so many people on my research, including paper drafts, talks, and research ideas, allowing me to learn quickly, while letting me give feedback to other people's work as well. I want to thank the people that I could still recall the most active in the seminar if they have not yet been mentioned elsewhere in this acknowledgment: Sasa Misailovic, Madhusudan Parthasarathy, Angello Astorga, Saikat Dutta, Jacob Laurel, Neil Zhao, Liia Butler, Yifan Zhao, Jonathan Osei-Owusu, Peilun Zhang, Zixin Huang, Keyur Joshi. Many of these people have also given me appreciated advice off the seminar. Likewise, I want to thank the people from the Systems Research Seminar if I have not yet mentioned them elsewhere: Indranil Gupta, Le Xu, Rui Yang, Haoran Qiu, Federico Cifuentes-Urtubey, Bingzhe Liu. I want to thank Han Zhao and Julia Hockenmaier for advising, Yifei He and Gargi Balasubramaniam for collaboration, on a fruitful experience in theoretical machine learning research. I thank Ruicheng Xian and Yuan Shen for the sporadic but meaningful discussions on research and Ph.D. lives. Last but not the least, I want to thank Wenyu Wang and Qingrong Chen for sharing their valuable experiences and advice on research, career, and life with me, when they were senior students and after they graduated, whom I learn a lot from.

I want to thank Vladimir Filkov, Bogdan Vasilescu, and Zhou Yu for hiring me as their research assistant during my undergraduate at UC Davis, giving me opportunities to learn on the way while working with them on important and interesting research projects. I thank them and their lab members who mentored me at the time, for showing me the possibility of computer science research.

Lastly, I want to thank my family, including but not limited to my parents and partner, and my friends kept in touch over the years. Thank you for being here. Thank you for listening to my problems even when they make the least sense, and for showing support and wisdom when I make tough decisions.

## Table of Contents

# Chapter 1: Introduction

Software systems are ubiquitous; they control the critical electronic infrastructures, devices, utilities, and applications that people use. Maintaining software reliability is therefore crucial, because unexpected program behaviors can emerge from software faults, leading to severe consequences with negative societal impact [1, 2, 3, 4].

Software developers commonly practice Continuous Integration (CI) to maintain the reliability of their software [5, 6, 7, 8, 9, 10, 11]. In CI, each change to the software from developers is pushed through a version control system. The CI service then automatically launches a CI build for the pushed change; the CI build compiles the changed code and runs regression testing. Regression testing runs tests on the changed code in the development environment, with the goal of helping the developers to detect and debug faults introduced by the change before the change could be deployed into production [12, 13, 14, 15, 16]. The regression test suite is a list of existing tests, sometimes including new tests written for the change. Test failures that emerge from regression testing expose potential fault(s); these failures are provided as debugging feedback to developers. Developers debug and fix the code leveraging the testing feedback, and then run more CI build(s) on the revised change—they repeat this process until resolving the regression test failures. Regression testing has been shown effective and practiced widely [9, 17, 18, 19, 20, 21].

As software systems offer more functionalities and are increasingly integrated into everyday life, the codebase size and change velocity have also grown rapidly in both proprietary and open-source software development [22, 23, 24]. Such growth directly increases the CI cost: regression test suites run longer and must be run more frequently. The prolonged test suite execution delays development cycles and prevents timely feedback to developers [18, 25, 26].

Researchers have proposed many techniques to improve the feedback efficiency of regression testing. These techniques can be broadly categorized into (1) Regression Test Selection, (2) Test Suite Reduction, and (3) Regression Test Prioritization [14, 27, 28]. Regression Test Selection samples and executes a subset of tests from the entire test suite. Its sampling algorithms are often based on the change [15, 29, 30, 31]. Test Suite Reduction removes redundant tests from the test suite with little to no impact on the overall fault detection ability of the entire test suite against the change [32, 33, 34]. Lastly, Regression Test Prioritization (RTP) aims to reorder the execution of tests in the test suite to expose failure-inducing faults sooner [17, 20, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44].

Compared to Regression Test Selection and Test Suite Reduction, RTP is guaranteed safe (i.e., including all tests that could expose fault in the change) and complete (i.e., including all tests that are transitively affected by the change), because it executes all tests in the test suite. Although RTP does not reduce the total regression testing cost, it reduces the developers' time to receive debugging feedback by speeding up regression fault detection. Test rankings produced by RTP techniques can also be leveraged in test selection and parallelization to further improve regression testing efficiency [23, 36, 41, 42, 45]. However, although RTP techniques have been studied for three decades with multiple surveys conducted, RTP research results have been rarely applied in CI practices [14, 17, 27, 28, 35, 46, 47, 48].

To enable a wider adoption of RTP in large software systems, existing or new RTP techniques should be demonstrated effective in relevant, practical testing scenarios that have up-to-date CI practices; RTP tools with robust performance and convenient usage should be developed for practitioners to apply RTP in popular or emerging software ecosystems.

This dissertation first studies a wide range of popular RTP techniques in realistic cases where RTP is especially important—long-running test suites [49]. We present the first RTP dataset of long-running test suites and revisit key research conclusions from prior RTP studies, to assess the effectiveness of existing RTP techniques in the latest CI context. This dissertation also applies RTP for configuration testing, and further proposes novel configuration-specific RTP techniques, with a comprehensive evaluation of these techniques [50]. Modern large-scale software systems are highly-configurable, where configuration change velocity has greatly increased and misconfiguration has become a dominant reliability concern. We demonstrate strong performance results of both the traditional and novel RTP techniques in an emerging yet critical testing scenario. Lastly, this dissertation presents a readily-usable, open-source RTP tool for Python and its most popular testing framework Pytest [51]. We realistically evaluate our tool by deploying it into over a dozen popular Python projects and rerunning their CI builds on the most popular CI service GitHub Actions. Evaluation results showcase the tool's effectiveness in speeding up fault detection, its low overhead, and its high usability.

## 1.1 THESIS STATEMENT

The thesis statement of this dissertation is the following:

*Regression Test Prioritization can improve fault detection cost-effectiveness in practical regression testing scenarios of modern software.*

This thesis statement has the following aspects: we can use RTP techniques to effectively speed up fault detection in the latest and most relevant regression testing context; we also can build convenient and effective tools to deploy RTP in popular software ecosystems.

This dissertation presents the following work to support this thesis statement: (1) revisiting key effectiveness results of RTP techniques on our dataset of recent, long-running test suites, (2) effectively applying traditional RTP techniques and proposing new techniques in the context of configuration testing, and (3) developing an RTP tool for Python and Pytest ecosystems with demonstrated efficacy and usability from realistic evaluation.

## 1.2 CONTRIBUTIONS

The work in this dissertation makes the following main contributions:

- **Regression Test Prioritization on Long-Running Test Suites.** Because RTP aims to expose faults sooner and reduce testing feedback time to developers, RTP is especially important for long-running test suites. While many studies have explored RTP, they are often based on outdated CI builds from over 10 years ago with test suites that last several minutes, or builds from inaccessible, proprietary projects.

  **Contributions:** This dissertation presents LRTS, the first RTP 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, this dissertation studies the effectiveness of 59 leading RTP techniques, the impact of failures from flaky tests and frequently-failing tests on RTP, and RTP for failing tests with no prior failures. It revisits prior key findings (9 confirmed, 2 refuted) and establishes 3 new findings. Results show that simply prioritizing faster tests that recently failed performs the best, outperforming the sophisticated techniques.

- **Regression Test Prioritization for Configuration Testing.** Configuration changes are among the dominant causes of failures of large-scale software system deployment. Given the velocity of configuration changes, typically at the scale of hundreds to thousands of times daily in modern cloud systems, checking configuration changes is critical to prevent failures due to misconfigurations. Prior work uses configuration testing, Ctest [52, 53], a technique that tests configuration changes together with the code that uses the changed configurations. Ctest can automatically generate a large number of ctests that can effectively detect misconfigurations, including those that are hard to detect by traditional techniques. However, running ctests can take a long time to detect misconfigurations, in situations where delay could be rather costly.

  **Contributions:** This dissertation proposes to apply RTP to reorder ctests to speed up the detection of misconfigurations, given that the traditional RTP has shown promises in speeding up the detection of regression code faults. This dissertation extensively evaluates a total of 84 traditional and novel configuration-specific RTP techniques. The experimental results on five widely used cloud projects demonstrate that RTP can substantially speed up misconfiguration detection. This dissertation also provides guidelines for applying RTP to configuration testing in practice.

- **Regression Test Prioritization Tool for Python.** While RTP has been researched for almost three decades, with many research techniques proposed, practical tools and evaluations are sporadic. Moreover, most prior work has only evaluated RTP techniques by simulating orders, which could miss important issues in test executions if RTP is to be deployed into CI, such as changed test durations and outcomes across executions, test-order dependency violations, and overhead of the deployed RTP techniques.

  **Contributions:** This dissertation presents PYTEST-RANKING, a robust tool for Python, and its most popular testing framework Pytest. We evaluate our tool on 4,308 builds for 14 open-source Python projects running on the GitHub Actions CI. Experiments show that PYTEST-RANKING integrates well with the Pytest ecosystem, has a low runtime overhead (0.03% of the test-suite run duration), and finds test failures 37%–75% faster than the Pytest default and *Random* orders on average across projects. Our test failure analysis on 1,121 failed TSRs also finds 146 flaky tests, in which 112 are order-dependent (OD). Our RTP effectiveness analysis also investigates how test duration variation and flaky test failure treatment impact the RTP technique effectiveness ranking on the executed TSRs. Our results demonstrate the practicality and effectiveness of PYTEST-RANKING, and provide implications of applying RTP in CI.

## 1.3 DISSERTATION ORGANIZATION

The rest of this dissertation is organized as follows:

**Chapter 2: Regression Test Prioritization on Long-Running Test Suites** This chapter presents (1) the RTP dataset of long-running test suites, and (2) a study of popular RTP techniques on the dataset. We describe the evaluated RTP techniques, and RTP evaluation settings, e.g., evaluation metrics. This chapter revisits key prior findings of RTP, and presents new findings, covering three aspects: RTP effectiveness on long-running test suites, impact of flaky tests and frequently-failing tests to RTP effectiveness, and RTP effectiveness on prioritizing tests with no prior failures [49].

This chapter is based on the paper "Revisiting Test-Case Prioritization on Long-Running Test Suites". Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024. Cheng, Runxiang; Wang, Shuai; Jabbarvand, Reyhaneh; Marinov, Darko. The dissertation author was the primary investigator and author of this work.

**Chapter 3: Regression Test Prioritization for Configuration Testing** This chapter presents applied traditional RTP techniques and our developed configuration-specific RTP techniques for prioritizing configuration tests. This chapter describes the concepts and implementations of these techniques, our proposed metrics for evaluating configuration test prioritization, and the study of the effectiveness of these techniques [50].

This chapter is based on the paper "Test-Case Prioritization for Configuration Testing". Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021. Cheng, Runxiang; Zhang, Lingming; Marinov, Darko; Xu, Tianyin. The dissertation author was the primary investigator and author of this work.

**Chapter 4: Regression Test Prioritization Tool for Python** This chapter presents the implementation, usage, and evaluation of PYTEST-RANKING, our proposed RTP tool for Python/Pytest ecosystems. In the evaluation, we run the RTP-reordered test suites, and analyze the test failures and test runtime variation from the collected test suite runs. We also report the effectiveness and overhead of PYTEST-RANKING [51].

This chapter is based on the paper "pytest-ranking: A Regression Test Prioritization Tool for Python". Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (Demonstrations), 2025. Cheng, Runxiang; Ke, Kaiyao; Marinov, Darko. The dissertation author was the primary investigator and author of this work.

**Chapter 5: Conclusions and Future Work** This chapter concludes the dissertation and provide discussions to the future work that can be done following the work from this dissertation.

**Appendix A: Other Work** The appendix lists the other work the dissertation author conducted during Ph.D. studies, including in parallel to the work presented in this dissertation [52, 54, 55, 56, 57, 58, 59].

## Chapter 2: Regression Test Prioritization on Long-Running Test Suites

This chapter presents the RTP dataset of long-running test suites, i.e., *LRTS*, and our study of RTP techniques on *LRTS*. Section 2.1 describes the limitations of prior RTP datasets and studies, and presents an overview of this work. Section 2.2 describes the leading categories of RTP techniques. Section 2.3 presents details of *LRTS*, purging steps of confounding test failures (i.e., failures of flaky tests and frequently-failing tests) in *LRTS*, and comparing *LRTS* to prior datasets of short-running test suites. Section 2.4 describes evaluation setup, including RTP effectiveness metrics, failure-to-fault mappings, and experiment procedure. Section 2.5 presents study results, including RTP effectiveness on long-running test suites (§2.5.1), impact of confounding test failures on RTP (§2.5.2), and RTP effectiveness on prioritizing tests with no prior failures (§2.5.3). Sections 2.6–2.7 describe the threats to validity and related work of this chapter. Section 2.8 provides a concluding summary.

### 2.1 OVERVIEW

The importance of RTP 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, RTP can be especially important for long-running test suites.

To date, there is a wealth of RTP techniques [14, 17, 27, 28, 35, 46, 47, 48]. Traditional RTP techniques use code coverage, and prioritize tests that cover more code elements [35, 60]. However, they have limited applicability, as code coverage is hard to collect [17, 20]. Recent empirical studies of RTP thus focus on techniques that rely on more accessible test features [28, 36, 37], such as history-based techniques that prioritize tests by their previous outcomes [14, 38], and time-based techniques that prioritize faster tests [12, 61]. Some other proposed RTP techniques use information retrieval (IR) [39, 40] or machine learning (ML) [41, 42, 43, 44] to leverage multiple information sources from CI to prioritize tests. For example, researchers developed Learning-to-Rank (LTR) RTP with supervised learning algorithms, and Ranking-to-Learn (RTL) RTP with reinforcement learning algorithms, both of which were shown effective.

Existing studies have investigated the effectiveness of RTP techniques in different contexts. Recent studies [36, 37, 40, 41, 62] evaluate RTP techniques on open-source projects. These studies consider historical test-suite runs and real test failures mostly from Java projects that use Travis CI [63]. For example, [40] studied IR-based RTP on 3,000 test-suite runs, Bertolino et al. [41] studied both LTR and RTL RTP, [36] studied LTR RTP on 20 open-source [62] and 3 proprietary projects, and Yaraghi et al. [37] studied LTR RTP 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 [64], and have test suites that are relatively short-running, e.g., on average several minutes (§2.3.3). 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 (§2.3.5), which may result in different effectiveness and ranking outcomes of existing RTP techniques.

Other studies consider RTP on software from large companies [17, 23, 28, 43, 65, 66, 67]. While providing valuable experience, these studies focus on very few RTP techniques, e.g., history-based (§2.2.2). Further, their observations and techniques are based on large testing 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 [17, 66]. These studies also provide limited data for future investigations—their studied project(s) are inaccessible [23, 43, 66, 67], lack heterogeneity [42, 65], or have rather old artifacts [17].

One key challenge in studying RTP is the lack of up-to-date, high-quality datasets, especially in cases where RTP can help the most: long-running test suites. Moreover, many popular RTP techniques have only been studied separately across different datasets and settings—there has been no recent extensive evaluation of leading RTP 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 work, 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 (§2.3). On *LRTS*, we evaluate 59 RTP techniques from five leading technique categories: time-based, history-based, IR-based, learning-based (LTR and RTL) techniques, and cost-cognizant hybrid techniques (§2.2). We study the effectiveness of these techniques in three contexts: recent, long-running test suites (§2.5.1); impact of flaky tests and frequently-failing tests (§2.5.2); and prioritizing failing tests that have no prior failure (§2.5.3). Our study revisits key findings from recent RTP studies [36, 37, 40, 41, 65] and presents new findings.

This work makes the following contributions:

- 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 (§2.3.3). We are releasing *LRTS*, with our code on:

`https://zenodo.org/records/12662090`

- We start with 26 *basic* RTP techniques—2 time-based, 6 history-based, 6 IR-based, 5 LTR, 6 RTL RTP techniques, and the *Random* baseline. We next apply two cost-cognizant hybrid RTP approaches to the basic techniques, to construct 33 hybrid techniques. In total, we evaluate 59 RTP techniques, on the widely-used metric Average Percentage of Faults Detected per Cost (APFDc) and APFD, under different failure-to-fault mappings [68]. 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.

- We revisit 11 key findings from recent RTP studies, confirming 9 and refuting 2 findings. We also present 3 new findings. Table 2.7 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.

## 2.2 RTP TECHNIQUES

We first overview different RTP technique categories and describe the techniques we use in our study. We focus on evaluating only *previously proposed RTP techniques* and do not promote any new technique to mitigate potential bias in evaluating RTP techniques on our new dataset. RTP is the problem of finding a test execution order that detects more faults faster [14, 35]. Depending on the heuristics that guide the ordering, we can categorize basic RTP techniques in four main categories: *time-based*, *history-based*, *IR-based*, and *learning-based*. The fifth category is *hybrid* RTP techniques that systematically combine heuristics from other different categories.

### 2.2.1 Time-based RTP

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 [12]. This RTP category, called Quickest-Time-First (QTF), has been recently shown to rival or outperform more sophisticated RTP techniques on short-running test suites [36, 40, 61]. 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.

### 2.2.2 History-based RTP

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.

**Test outcome** Two history-based RTP 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 [23, 40, 69, 70, 71, 72], (2) *LatestFail* prioritizes tests that failed more recently [17, 69, 71, 73], (3) *MostTrans* prioritize tests that have a higher historical transition count [36, 74], and (4) *LatestTrans* prioritizes tests that transitioned more recently [36, 74].

**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 [22, 75, 76, 77, 78, 79]. 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 [36].

### 2.2.3 IR-based RTP

IR-based techniques rely on textual similarity to identify the tests that are more relevant to code changes [39, 40]. 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. 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

Table 2.1: **LTR RTP 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 |

models, e.g., Term Frequency-Inverse Document Frequency (TF-IDF) [80] or BM25 [81], and the amount of context they consider for a code change [40]. 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 [82]). We evaluate 6 IR-based techniques from prior work [40] that use BM25 and TF-IDF IR models with the 3 different context lengths mentioned above.

### 2.2.4 Learning-based RTP

With the advent of machine learning (ML), a number of RTP techniques use ML algorithms to predict the ranking of tests. These RTP techniques can be broadly put into two subcategories [41]: Learning-to-Rank (LTR) and Ranking-to-Learn (RTL).

**Learning-to-Rank** LTR RTP techniques use supervised learning algorithms, in which an ML model is trained on historical CI builds to predict ranking of tests for future builds [23, 27, 36, 37, 41, 43, 72, 83]. LTR techniques train ML models with features from the test, code or code change, and execution history [36, 37], 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 RTP.

Prior work evaluated how different ML algorithms impact the effectiveness of LTR techniques [36, 37, 41]. 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 RTP. We revisit the most studied ML algorithm (gradient boosting trees) and training:testing data ratio (75%) [27, 36, 41]. We use the most effective features prior work identified that are also easily accessible in CI [36, 37, 84]. Table 2.1 lists them categorized into four feature sets, which follow the same definitions as in Elsner et al. [36]. In total, we evaluate 5 LTR techniques, 4 techniques using one set of features each, and 1 technique using all four feature sets.

**Ranking-to-Learn** RTL RTP techniques use reinforcement learning (RL) algorithms [41, 42, 85, 86]. In contrast to LTR where a model is trained *offline*, RTL trains its model *online*—RTL RTP 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. RTL RTP is initially deployed with no prior knowledge, and gradually learns a policy for prioritizing tests over time as it goes through more builds in the project.

A test state encodes a test's metadata, e.g., previous outcome and duration. Given all test states of the current build, RTL RTP 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 RTP 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 [41, 42], 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 RTP techniques.

#### 2.2.5 Hybrid RTP

After describing the basic RTP techniques, we now describe the hybrid RTP 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 RTP settings [40, 50], which motivated us to include them in our study. We adopt two hybrid RTP approaches from prior work [40]: cost-cognizant (*CC*) and cost-history-cognizant (*CCH*).

**Cost-ognizant** 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.

## 2.3 DATASET OF LONG-RUNNING TEST SUITES

§2.3.1-2.3.2 describe our project selection criteria and the construction of *LRTS*. §2.3.3 provides more details on *LRTS*, with an analysis of the distributions of its CI builds and test failures. §2.3.4 describes how we account for confounding test failures (failures of flaky tests and frequently failing tests). §2.3.5 compares characteristics of *LRTS* with recent datasets of short-running test suites.

### 2.3.1 Project Selection

We seek projects that are open-source, because they often provide transparent data access to their *recent* CI builds, with test failures induced by real faults [62]. 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 is the inclusion of projects with *long-running test suites*, because these projects can benefit the most from RTP.

We focus on selecting projects from the Apache Software Foundation (ASF) [87] because it offers a diversity of renowned open-source projects and has been studied by many researchers for over two decades [88]. 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 [89]. 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., [90, 91]), as these projects can have long-running test suites, and Jenkins CI provides uniformed API for downloading serialized build data [92]. We select from the longest-running projects, where the test-suite execution time for the majority of the project's

Table 2.2: **Projects in our dataset.**

| Project | Primary use | Stars | Main PLs | SLOC |
|---|---|---|---|---|
| ActiveMQ | Message broker | 2K | Java | 669K |
| Hadoop | Big-data processing | 14K | Java | 4M |
| HBase | Big-data storage | 5K | Java | 1M |
| Hive | Data warehouse | 5K | Java, HiveQL | 2M |
| Jackrabbit Oak | Content repository | 381 | Java | 694K |
| James | Mail server | 848 | Java, Scala | 793K |
| Kafka | Stream processing | 26K | Java, Scala | 905K |
| Karaf | Modulith runtime | 669 | Java, Scala | 186K |
| Log4j 2 | Logging API | 3K | Java | 277K |
| TVM | Compiler stack | 10K | Python, C++ | 818K |

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 2.2 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 [93], 1 Gradle [94], and 1 CMake); all use Jenkins CI. To our knowledge, *LRTS* is the first open-source dataset for investigating the effectiveness of RTP techniques on *multiple* large-scale projects with long-running test suites and actual CI failures.

### 2.3.2 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 [24, 37, 40, 62] and describe our differences below.

**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 the PRs via GitHub API [95].

A Jenkins CI build can have multiple stages [96], similar to how a Travis CI build can have multiple jobs [37, 64]. 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 [97]. Following prior work [36, 37, 40]

Table 2.3: ***LRTS*** **dataset summary. TSR denotes test-suite run.**

| Project | Period (days) | #CI build | #TSR | #Failed TSR |
|---|---|---|---|---|
| ActiveMQ | 827 | 207 | 207 | 109 |
| Hadoop | 1,094 | 1,299 | 1,299 | 543 |
| HBase | 504 | 278 | 553 | 215 |
| Hive | 618 | 2,056 | 2,056 | 1,419 |
| Jackrabbit Oak | 745 | 860 | 860 | 639 |
| James | 786 | 2,404 | 3,147 | 1,399 |
| Kafka | 984 | 11,843 | 39,006 | 24,047 |
| Karaf | 959 | 620 | 620 | 174 |
| Log4j 2 | 436 | 270 | 528 | 162 |
| TVM | 631 | 1,418 | 9,161 | 1,411 |
| **Total** | - | **21,255** | **57,437** | **30,118** |

Table 2.4: **Statistics (averages) on failed TSRs in *LRTS*. TC denotes test class, and TM denotes test method.**

| Project | #TC | #Failed TC | #TM | #Failed TM | Duration (hours) |
|---|---|---|---|---|---|
| ActiveMQ | 676 | 3 | 6,081 | 34 | 4.36 |
| Hadoop | 829 | 6 | 7,289 | 24 | 5.57 |
| HBase | 1,061 | 2 | 6,369 | 3 | 9.28 |
| Hive | 1,273 | 9 | 40,921 | 83 | 26.12 |
| Jackrabbit Oak | 1,897 | 12 | 19,699 | 107 | 3.27 |
| James | 1,864 | 6 | 34,718 | 37 | 2.15 |
| Kafka | 1,232 | 4 | 19,399 | 12 | 7.59 |
| Karaf | 205 | 2 | 841 | 2 | 0.58 |
| Log4j 2 | 641 | 3 | 3,918 | 4 | 0.25 |
| TVM | 526 | 3 | 8,564 | 37 | 4.83 |
| **Average** | **1020** | **5** | **14,780** | **34** | **6.4** |

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 RTP. We also treat each stage in a project as having its own CI history, which consists of all builds that included that stage.

**Test suite information** We obtain test report URLs from build metadata files, and extract test reports in `JSON` format via Jenkins CI API [92]. Our process differs from the extraction of test results from Travis CI [64] 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 [24, 37, 40, 62, 64]. 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.

**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`) [64]. 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 [98], 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.

### 2.3.3 Dataset Overview

*LRTS* curates the data of 21,255 unique CI builds from 10 projects. These builds have 57,437 test-suite runs (refer to as TSRs), of which 30,118 (59%) TSRs had at least one failed test. A build can have more than one TSR if it has more than one stage (§2.3.2). Table 2.3 provides more details on *LRTS* [99], including period length; total number of builds, TSRs, failed TSRs per project. Table 2.4 provides more details on the average number of test classes, test methods, and duration across *failed* TSRs. The durations are based on Jenkins CI test reports in each project's CI server, by summing up the durations of all executed tests in each TSR. If tests run in parallel to reduce the total elapsed time, RTP can prioritize and parallelize tests to find failures sooner [45, 100]. For a fair comparison, as in prior work [28, 36, 40, 41], we evaluate RTP techniques while considering that each TSR runs its tests sequentially.

In Table 2.5, we compare *LRTS* with other datasets in RTP studies since the RTPTorrent release in 2020 [28, 62]. We omit datasets with synthetic CI failures [45, 101, 102, 103], or whose long-running test suites come from inaccessible, proprietary projects [36, 84, 85, 86, 104, 105]. Many recent studies use the same set of builds from before 2016 in TravisTorrent or proprietary projects [17, 42, 62, 63, 64]. In comparison, the builds in *LRTS* span from 2020 to 2023, reflecting more current CI practices and are suitable for up-to-date RTP studies [27]. 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 [106].

Table 2.5 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 open-source Google Chrome dataset that has only one project. Table 2.5 also shows that some

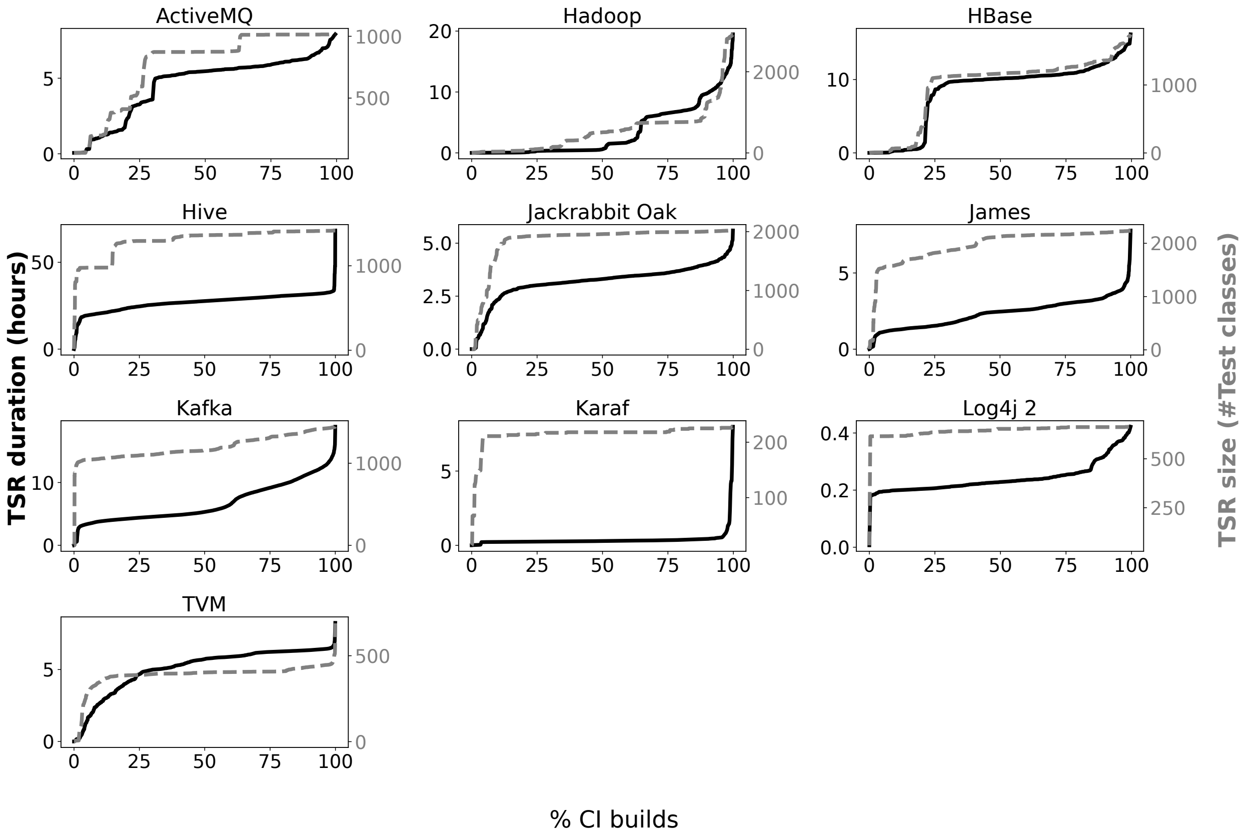


Figure 2.1: **Distribution of CI builds by the duration (hours) and size (number of test classes) of all (not only failed) TSRs. The solid dark lines and left y-axes show CDFs by TSR duration. The dashed lighter lines and right y-axes show CDFs by TSR size.**

Table 2.5: **Comparing RTP datasets.**

| RTP dataset | #Project | #TSR | Duration |
|---|---:|---:|---:|
| RTPTorrent [62] | 20 | 100K | 0.17 |
| Peng et al. [40] | 123 | 3K | 0.09 |
| RT-CI [41] | 6 | 3K | <0.01 |
| Pan et al. [24] | 242 | 15K | 0.35 |
| TCP-CI [37] | 25 | 21K | 0.27 |
| Chrome [65] | 1 | 50K | 7.96 |
| ***LRTS* (Ours)** | 10 | 57K | 6.50 |

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

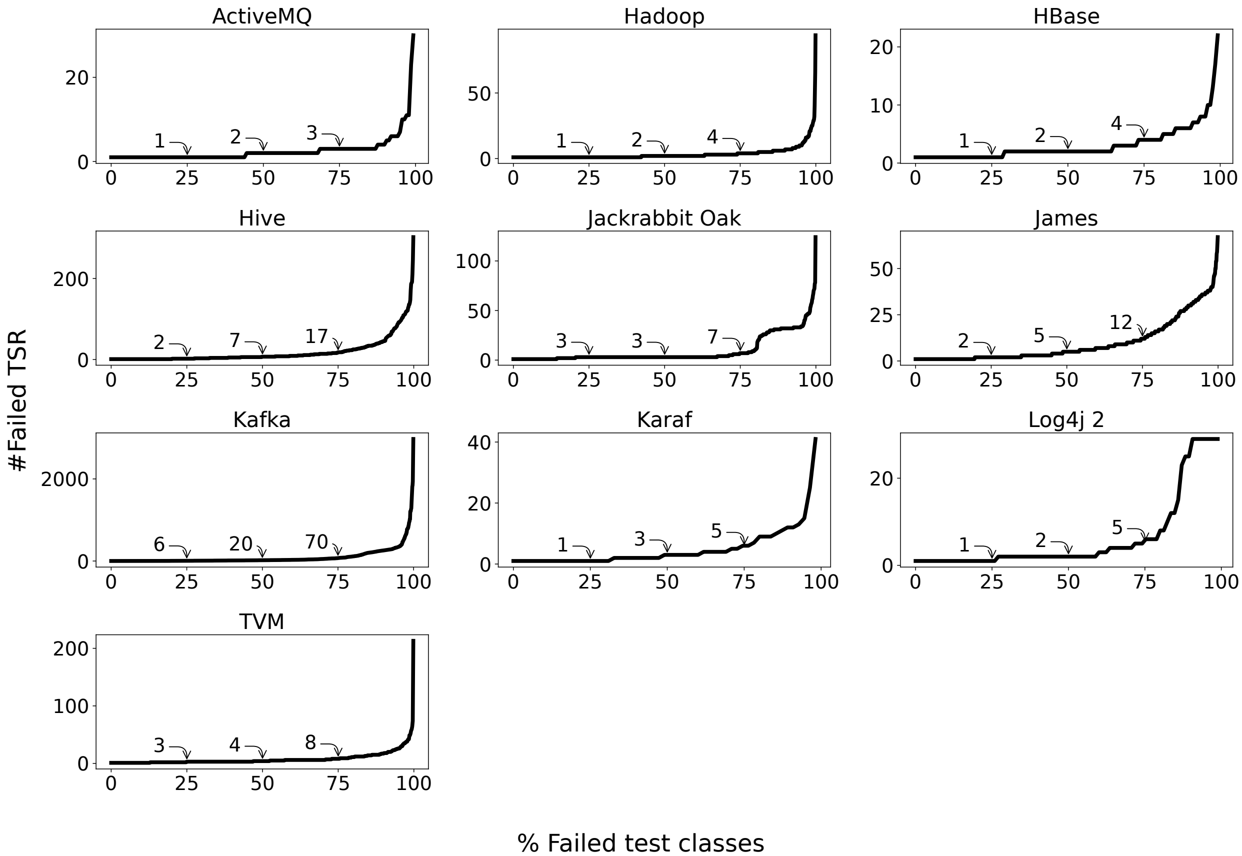


Figure 2.2: **Distribution of failed test classes by their #failed TSR in each project. The Q1, Q2, and Q3 values are annotated.**

shorter-running test suites and fewer TSRs per project [63]. Moreover, they had less diversity in programming languages and build systems (mostly in Java, single-module, use Maven). Overall, *LRTS* complements the existing datasets with a different set of projects that use Jenkins CI, with *recent* builds, and *long-running* test suites.

**Distribution of CI builds** Figure 2.1 shows the distribution of builds in *LRTS* by their TSR duration (measured in hours) and test-suite size (measured in the number of test classes). 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. These statistics show that *LRTS* consists of longer-running test suites.

**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. Figure 2.2 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 2.3—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 [37, 40, 65]. §2.3.4 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.

### 2.3.4 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 RTP techniques [36, 40, 65, 107, 108]. 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) [109]. 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.

**Flaky tests** Flaky tests are tests that can nondeterministically pass or fail for the same code under test [110]. Because they may impact the effectiveness of RTP techniques on detecting regression-induced failures that developers care about, some RTP studies on real CI failures explicitly account for flaky tests [40, 65, 107, 108].

Popular, large-scale systems often consider flaky tests in CI to some extent [23, 65]. 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 [111] or Pytest [112]. HBase and Hive also maintain dashboards to proactively run jobs to track flaky tests and exclude them in CI runs [113, 114]. All 10 projects use JIRA [115] 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 RTP studies, it is crucial to identify these failures. Some prior work [23, 40] 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 [36, 65]. 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 JIRA or GitHub issue tracker [116]. 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 (§2.3.2), we treat failures that were not in all its TSRs as flaky. This approach is similar to rerunning [23] 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 [24]. Once we identify flaky test failures, we can remove all such failures from the TSRs.

**Frequently failing tests** As shown in §2.3.3, 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 [37], and some of the failures could be due to test flakiness [65]. In our case, 53% of the frequently failing tests are also identified as flaky tests. Frequently failing tests are often ignored by developers and have no practical value to be prioritized. Following prior work [37], 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 $\langle$project, stage$\rangle$ pair.

### 2.3.5 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 (§2.3.3), which *may* lead to different effectiveness and ranking results of existing RTP 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) [36, 37, 40], 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 [36] 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 RTP techniques without code coverage or dependency information. Our results also show that minor imprecision in the RTP technique can cause a large penalty in the technique's failure-finding effectiveness (§2.5.1).

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, RTP 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 RTP technique can effectively prioritize failing tests ahead of the passing tests in a much larger test suite, which motivates our study.

## 2.4 EXPERIMENTAL SETUP

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

### 2.4.1 Evaluation Settings

**Failure-to-fault mappings** Mapping test failures to the faulty code is crucial for evaluating RTP techniques—the goal of RTP 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 [36, 40, 62], 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 RTP 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 [36, 40]. We evaluate on both mappings, following prior work [68].

**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 [36, 37, 40, 41] rather than test methods [117, 118] or test suites [17, 119].

**Evaluation metrics** Common metrics used to evaluate RTP techniques are Average Percentage of Faults Detected (APFD) [14] and Average Percentage of Faults Detected per Cost (APFDc) [28, 120, 121]. APFDc is a *cost-aware* variant of APFD that considers the cost of test executions [120, 121]. APFDc commonly uses the execution time of the evaluated TSR as the cost [61, 122]. Thus, it effectively measures how many faults are found per test execution time. Both metrics are normalized to [0, 1], 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 2.4). A small difference can indicate a large time for longer-running test suites.

The definitions of APFD and APFDc are as followed:

*Average Percentage of Faults Detected (APFD).* Let $n$ be the number of tests to be run, $m$ be the number of faults in the regression change, and $TF_i$ be the position (in the order) of the first failed test that detects the $i^{th}$ fault:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{2.1}$$

APFD computes the area under the curve between the percentage of detected faults in a regression change and the percentage of the test suite executed.

*Average Percentage of Faults Detected per Cost (APFDc).* Let $n$, $m$, and $TF_i$ be the same as for APFD, and $t_j$ be the execution time of the $j^{th}$ test in the prioritized order:

$$APFDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \times m} \tag{2.2}$$

APFDc computes the area under the curve between the percentage of detected faults in a regression change and the percentage of its test suite *cost* incurred.

### 2.4.2 RTP Data Collection

We first order all the builds in *LRTS* chronologically to respect temporal dependencies in regression testing [36, 78, 123]. For time-based RTP, 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 [50]. For history-based RTP, we collect test outcome data from builds prior to the current build. For IR-based RTP, 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 RTP, we collect all features from Table 2.1 following prior work [36, 37].

### 2.4.3 RTP Technique Implementation

**General logic** We wrote a generic pipeline to run and evaluate different RTP techniques. Given an RTP 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.

**IR-based RTP** 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 [40]. We use the NLP-based tokenizer from Peng et al. [40] 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.

**LTR RTP** We follow the same data processing, implementation, and training procedure as prior work [36, 37]. Given that we order *LRTS* chronologically, we use its first 75% (older

Table 2.6: **Dataset versions.**

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

builds) as training data to the ML algorithms, and evaluate the trained ML models on the remaining 25% (§2.2.4). Each data sample corresponds to a ⟨TSR, test⟩ pair, represented as a pair of a feature vector (consisting of features in Table 2.1) and test outcome. Given a TSR $R$, LTR RTP 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 [36, 124, 125]; we use default hyper-parameter values provided by the scikit-learn package for training [36, 37, 61].

**RTL RTP** We use the released implementations of RL agents and reward functions [126] for RTL RTP from Spieker et al. [42], as in prior RTL RTP studies [41, 85, 86]. We evaluate RTL techniques with the same hyper-parameter values as prior work [41, 85, 86], 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 [42]; we present the best one.

### 2.4.4 Experimental Procedure

We use 3 *LRTS* versions to study RTP: (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 2.6 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.

To reduce randomness in the experiments, as prior work [36, 42, 122], 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 RTP technique (denoted as *Random*) to serve as a baseline, which randomly shuffles all tests.

In total, we evaluated 59 RTP techniques: 26 basic techniques, of which 25 are described in §2.2.1-2.2.4, 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 RTP 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 RTP because it solely learns from pre-defined states, actions, and rewards during runtime. Adding external heuristics would interfere with the learning process.

## 2.5 EVALUATION

This work aims to answer the following research questions:

- **RQ1:** How do different RTP 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 RTP techniques?
- **RQ3:** How do RTP techniques perform in detecting the first failure throughout CI history for each failed test?

Table 2.7 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.

### 2.5.1 RQ1: Effectiveness of RTP Techniques

This RQ compares different RTP techniques on *LRTS-DeConf* that omits confounding test failures. In Figure 2.3, each box plot shows the distribution of APFDc values for each technique, while Figure 2.4 shows the distribution of APFD values. 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 [23, 36, 37, 78, 123]). Each box plot represents 100 (10*10) values, for 10 projects and 10 experiment runs. We use average values across TSRs in each project to weigh all projects equally regardless of the number of TSRs [40, 41, 50].

**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 (Figure 2.3) and APFD (Figure 2.4). For the same reason, we also expected

Table 2.7: **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 RTP techniques [40]. | ✓ |
| **F2** APFD can be misleading and give different ranking of RTP techniques than APFDc for evaluating the cost-effectiveness of RTP techniques [61, 121]. | ✓ |
| **F3** Basic time-based and history-based techniques can rival or outperform sophisticated IR-based and learning-based techniques [36, 40]. | ✓ |
| **F4** IR-based techniques perform worse than time-based and history-based techniques [40] (up to 18% or 11% worse in APFDc without or with hybrid, respectively). | ✗ |
| **F5** Different configurations, e.g., IR model, context length, have little impact on the effectiveness of IR-based techniques [39, 40]. | ✗ |
| **F6** LTR RTP technique is among the most effective RTP techniques when training with all available features [36]. | ✓ |
| **F7** In LTR RTP, 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 ($F_4$) [36, 37]. | ✓ |
| **F8** RTL techniques generally perform better than random [42] (up to 23% better in APFDc), but worse than LTR techniques [41] (up to 34% worse in APFDc). | ✓ |
| **F9** Cost-cognizant hybrid RTP approaches can substantially improve the effectiveness of basic RTP techniques [40] (improving APFDc values by 6%–41%). | ✓ |
| **F10** Among all techniques, hybrid perform the best [40], 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 [65] (APFDc values drop by up to 15%). | ✓ |
| **F12** Techniques that favor more recent test history, e.g., *LatestFail* and *RTL (NN-TestFail)*, are more resilient to confounding test failures (APFDc values drop by up to 7%). | 💡 |
| **F13** Time-based and change-aware techniques, e.g., IR-based, are the least affected by the presence of confounding test failures (APFDc values increase by up to 12%). | 💡 |
| **F14** Time-based and change-aware techniques are effective in finding the first failures of tests, followed by *Random*, then history-based. | 💡 |

the ranking of some techniques to differ between mappings, e.g., when test suites are larger (§2.3.5), 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

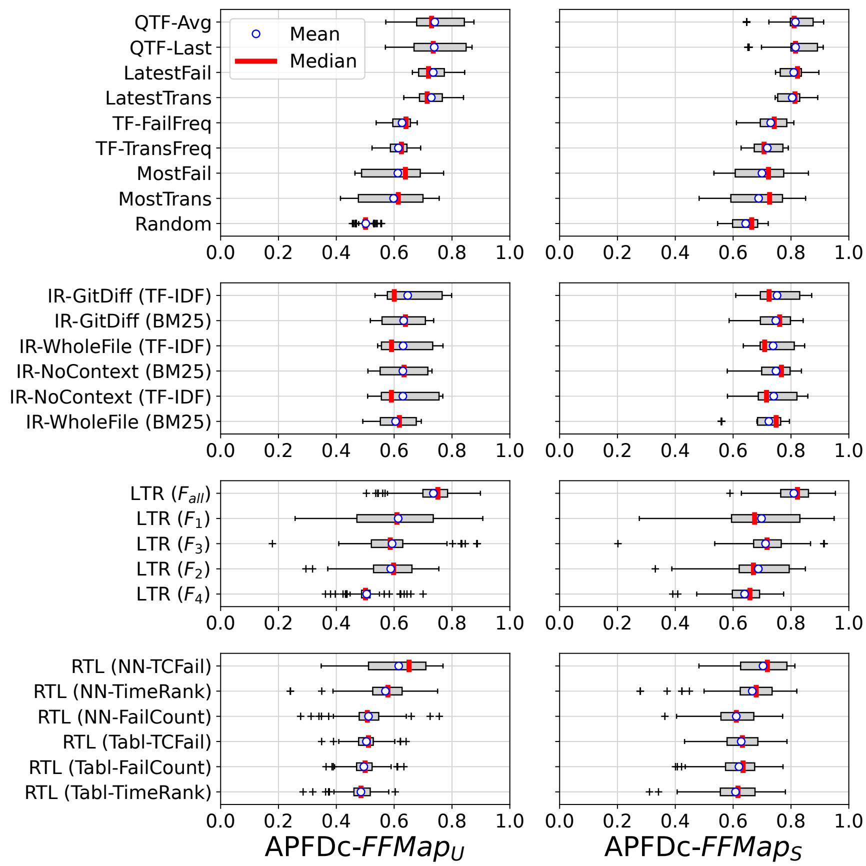


Figure 2.3: **APFDc results of RTP techniques on *LRTS-DeConf*. Four rows from top to bottom show the results of (1) time-based, history-based RTP, and *Random* baseline, (2) IR-based RTP, (3) LTR RTP, and (4) RTL RTP, respectively. Two columns from left to right show the value distributions of APFDc with $FFMap_U$ and $FFMaps_S$. RTP techniques in each row are organized in the descending order by their mean APFDc-$FFMap_U$ values.**

is because each RTP 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 [50, 61, 120, 121]. We expect the prior finding to not differ on test suites where tests run longer. Indeed, we confirm the prior finding (**F2 ✓**). Comparing Figure 2.3 and Figure 2.4, we can see that while the ranking of some techniques is similar across APFDc and APFD, the ranking of time-based and RTL techniques are opposite. We thus focus on APFDc results in the following sections.

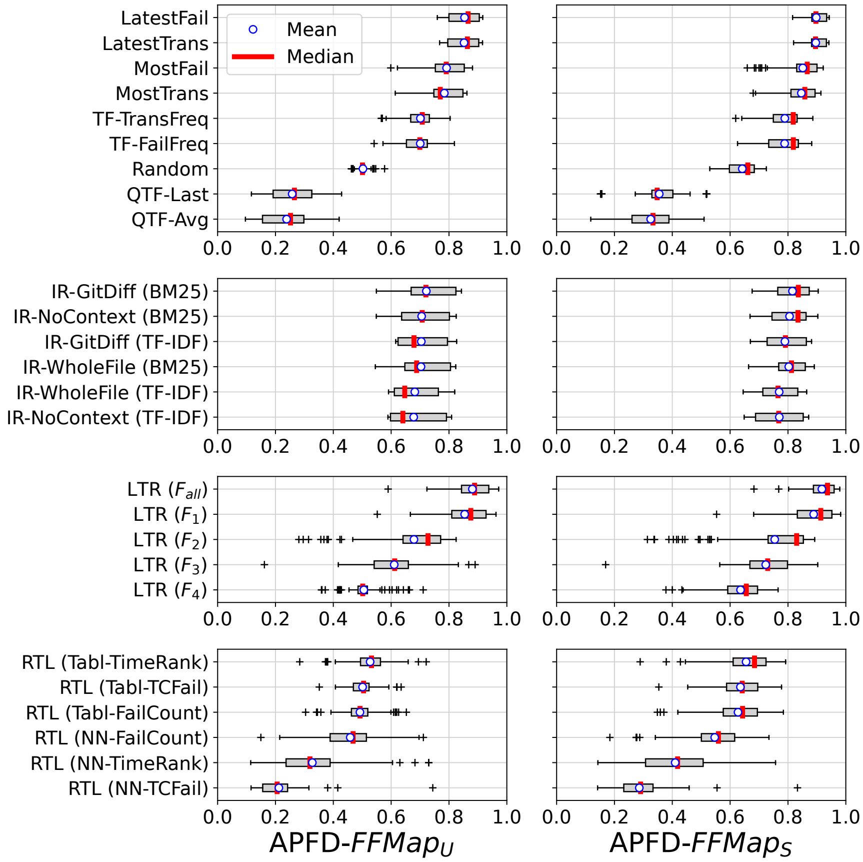


Figure 2.4: **APFD results of RTP techniques on *LRTS-DeConf*. Four rows from top to bottom show the results of (1) time-based, history-based RTP, and *Random* baseline, (2) IR-based RTP, (3) LTR RTP, and (4) RTL RTP, respectively. Two columns from left to right show the value distributions of APFD with $FFMap_U$ and $FFMap_S$, respectively. RTP techniques in each row are organized in the descending order by their mean APFD-$FFMap_U$ values.**

**Analysis of basic RTP techniques** As prior studies [36, 37, 40, 41, 50, 61], 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 [127], which assesses the difference and puts techniques into different groups if their APFDc values differ significantly [40, 47, 61, 128]. 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 2.8, "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 effec-

Table 2.8: **APFDc-$FFMap_U$ results on *LRTS-DeConf*. Horizontal lines separate RTP technique categories.**

| RTP 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 | - | - | - | - |

tiveness group from Tukey HSD test within each basic RTP category and across all basic techniques, respectively; "Imp" columns show the improvement of mean APFDc values from "Basic" column to the "*CC*" and "*CCH*" column(s).

**Time-based and history-based RTP** Prior studies have shown the effectiveness of sophisticated IR and ML RTP techniques in industrial settings [43, 61, 129, 130, 131], while more recent studies showed the simplest time-based and history-based techniques are equally effective on short-running test suites [36, 40]. Because longer-running test suites have different characteristics (§2.3.5), 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 2.8 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 2.3 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 [132].

**IR-based RTP** IR-based techniques have been shown to often outperform time-based and history-based techniques on short-running test suites [40]. We expected the prior finding to stand on *LRTS*, because test method bodies in long-running test suites are larger, and IR RTP is effective precisely because it captures textual relationships between documents [39].

Contrary to our expectation, however, our evaluation results refute the prior finding (**F4** ✗). In Table 2.8, 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. Even when combined with hybrid approaches, IR-based techniques have lower APFDc values than the best time-based and history-based techniques (i.e., *LatestFail*, *LatestTrans*).

Our results also refute that IR model and query context length configuration substantially impact the effectiveness of IR-based RTP [40] (**F5** ✗). Figure 2.3 shows all 6 IR-based techniques have similar distributions; Table 2.8 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 RTP's effectiveness differs from prior work, we first explore the difference between *LRTS-DeConf* and the prior IR RTP dataset (denoted as IRDataset) [40,

Table 2.9: **IR experiment results.**

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

133]. 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 2.9 shows our experiment results; each cell is the APFDc-$FFMap_U$ values averaged across all IR-based techniques. From Table 2.9, 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 [40].

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 RTP** We expected LTR RTP to be competitive on long-running test suites as LTR techniques can model a large amount of test and change data. Indeed, Table 2.8 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 RTP effectiveness [41, 42]. We thus expected, and confirm that, while some RTL techniques are certainly better than *Random* [42], they are usually outperformed by LTR techniques [41] (**F8 ✓**). Table 2.8 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) [27, 36, 37], we recommend time-based and history-based techniques over current learning-based ones.

**Analysis of hybrid RTP techniques** We expected the evaluated cost-cognizant hybrid approaches to only marginally improve basic RTP techniques on longer-running test suites (based on §2.5.1). 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 2.8 shows *CC* and *CCH* approaches improve the mean APFDc of basic techniques by 9%-41% and 6%-47%, respectively. Table 2.8 further shows that fusing heuristics from the best basic RTP techniques *QTF-Last* and *LatestFail* gives *LatestFail+CC* that achieves the highest APFDc (0.835) among all techniques (**F10 ✓**).

### 2.5.2 RQ2: Impact of Confounding Test Failures

Recent studies use real CI datasets that have confounding test failures [36, 37, 41, 62, 134], and detecting these failures earlier in RTP may provide no value to the developers [37, 40, 65, 107, 108]. However, there is very limited evaluation on the effectiveness of RTP techniques under the impact of confounding test failures [40, 65, 107]—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 [40] or single-project dataset [65]. In this RQ, we aim to provide a broader investigation on a wider range of RTP techniques under the impact of confounding test failures. Compared to prior work, we evaluate 3 times more RTP techniques on a 10-project dataset (with the first evaluation of LTR and RTL RTP), and consider both flaky tests and frequently-failing tests.

Following prior work [65], we evaluate RTP 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 (§2.5.1) and present our results in Table 2.10, 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 RTP 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 2.10, 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 [40, 65] (**F11 ✓**).

However, not all history-based techniques are heavily impacted by confounding test failures—in Table 2.10, *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.

### 2.5.3 RQ3: Effectiveness on First Failures

Failing builds are relatively common in practice [65, 107]. 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 as they are more likely due to recent change. Moreover, although history-based techniques have been

Table 2.10: **Mean APFDc-$FFMap_U$ and effectiveness group of RTP techniques on all three versions of *LRTS*.**

| RTP 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 | - | - |

shown effective (e.g., *LatestFail*), 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 RTP 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 [24, 73]. We omit learning-based techniques, because the latest 25% of the builds used for evaluating learning-based RTP have insufficient first failures to make generalizable observations.

The *LRTS-FirstFail* columns in Table 2.10 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 RTP remains the most effective in this RQ (**F14** 💡). Overall, our study has shown that *the simplest* QTF stands as the most cost-effective RTP 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 RTP techniques that lexicographically prioritize tests by history-based heuristics, and use time-based or IR-based to break ties.

## 2.6 THREATS TO VALIDITY

**External validity** 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 [36, 37, 40, 47, 61, 128]. To reduce threat from the evaluated RTP techniques, we use the same RTP data collection [36, 40, 42], settings, and implementations as prior work [36, 37, 40, 41, 42]. To reduce threat from randomness, we run all experiments 10 times [36, 41, 42] To reduce threat from flaky tests [23, 36, 37], we perform both manual inspection and automated filtering (§2.3.4). We then evaluate all techniques on our dataset with and without failures from the identified flaky tests. Due to high cost of running test suites, we do not run the generated test orders [135].

**Internal validity** 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.

**Construct validity** The threats to construct validity mainly lie in our evaluation metrics. To reduce such threats, we use the two most widely-used metrics for evaluating RTP techniques: APFDc and APFD. We also evaluate on metric Normalized Rank Percentile Average (NRPA), as in a few prior studies on RTL RTP [41, 85, 136], but NRPA cannot differentiate different RTP techniques (values are mostly the same to the third decimal place) in our experiment, and we thus omit detailed results.

## 2.7 RELATED WORK

**RTP techniques** RTP has been extensively studied as summarized in several surveys [14, 17, 27, 28, 35, 46, 47, 48]. Besides the techniques in §2.2, prior work has also proposed techniques based on code coverage [35, 60], adaptive random testing [137], constraint solving [138], and genetic algorithms [139]. RTP has been applied to mutation testing [140], fault localization and repair [141, 142, 143, 144], testing configurable systems [50, 145, 146], and deep neural networks [147, 148]. We focus on studying techniques most widely used in recent work [36, 37, 40, 41, 42, 65].

**RTP datasets and studies** RTP datasets are crucial for studying RTP techniques. Mattis et al. [62] listed RTP datasets prior to 2020 and released RTPTorrent that curated real CI builds from 20 projects via TravisTorrent. Prior to RTPTorrent, only 18 RTP datasets entirely consisted of real CI builds, and only two of them made their TSR data available [42, 149]. RTP studies in industrial settings exist but provide limited data for future work [17, 23, 43, 66, 67, 84]. Recent studies on open-source datasets extend RTPTorrent with proprietary projects [36, 84], and some collect their own RTP datasets from more Travis CI Java projects [24, 37, 40, 41, 65, 85].

## 2.8 SUMMARY

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 hours) of 10 large-scale, open-source projects that use Jenkins CI. On *LRTS*, we evaluate the effectiveness of 59 techniques from 5 leading RTP 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 RTP 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.

# Chapter 3: Regression Test Prioritization for Configuration Testing

This chapter presents our work on RTP techniques for configuration testing. Section 3.1 describes the challenges in configuration, and presents an overview of our work on applying RTP for configuration testing. Section 3.2 provides the background in configuration tests. Section 3.3 describe the concept of our applied as well as proposed RTP techniques for configuration testing: non-peer-based RTP techniques which include the traditional techniques (§3.3.1) and configuration-specific techniques, peer-based configuration-specific RTP techniques (§3.3.2), and hybrid RTP techniques (§3.3.3). Section 3.4 describes the experimental setup, including our proposed metrics for evaluating configuration test prioritization effectiveness. Section 3.5 presents our evaluation results of comparing the techniques described in §3.3. Section 3.6 and 3.7 describe the threats to validity and related work of this work, respectively. Section 3.8 further discusses the implications of our results in practice, and highlights future directions of this work. Section 3.9 provides a concluding summary.

## 3.1 OVERVIEW

Besides source-code changes, configuration changes are among the dominant causes of failures in large-scale software system deployments. In fact, configuration changes can be much more frequent than code changes. Many companies are deploying configuration changes to production systems hundreds to thousands of times a day [150, 151, 152, 153], hence *misconfigurations* become inevitable. For example, 16% of the service-level incidents at Facebook are induced by configuration changes [154], including major outages that turn down the entire service [155, 156], and misconfigurations were reported as the second largest cause of service disruptions in a main Google service [157]. The prevalence and severity of misconfigurations have been repeatedly reported by many failure studies [158, 159, 160, 161, 162, 163, 164, 165, 166].

Recently, Ctest has been proposed as a promising technique for configuration testing, i.e., testing a configuration before deployment [52, 167]. Ctest can effectively detect misconfigurations. The key idea of configuration testing is to connect configuration changes to software tests, so that configuration changes can be tested *in the context* of code affected by the changes. In this way, configuration testing can reason about the program behavior under the actual configuration values to be deployed and detect sophisticated misconfigurations that can hardly be detected by rule-based validation [154, 168, 169, 170, 171] or data-driven approaches [153, 172, 173, 174, 175, 176, 177, 178, 179]. Attractively, our prior work [52] shows that configuration test cases, or *ctests*, can be generated by parameterizing existing software tests abundant in mature software projects—up to 83.2% of existing tests can be transformed into ctests.

At a high level, a ctest is a software test parameterized by a set of configuration parameters. Running a ctest instantiates each parameter with a concrete value (e.g., the default value, the current value in production, or a new value to be deployed to production). Given a configuration change, all the ctests which are parameterized by at least one of the changed parameters are selected to run. Because one configuration parameter can parameterize many ctests, a configuration change can require running a large number of ctests. For example, some configuration changes from the HDFS project require running more than 2,000 ctests on average, which is over half of the total number of tests in that project [52]. Overall, in the Ctest dataset of five open-source projects (HCommon, HDFS, HBase, ZooKeeper, and Alluxio) [52], the number of ctests per configuration parameter is 1–3,069 (average 821), and a configuration change modifies 1–29 (average 6) parameters.

One main challenge in adopting configuration testing in continuous deployment is the time required to detect misconfigurations. This test-running time is on the critical path from the point where configuration changes are made to the point where they are deployed to production. For example, in the Ctest dataset, the time to run all ctests ranges from 20 minutes to 230 minutes (with an average of 97 *minutes*) per project. Given the velocity of configuration changes in modern deployment cycles [153, 154, 166], misconfigurations inevitably happen. With the large number of ctests to run before deployment, the time to detect the misconfiguration is crucial, because developers cannot start troubleshooting until the misconfiguration is detected. The time to detect the misconfiguration can greatly affect configuration deployment.

We are the first to address the cost of configuration testing using RTP. Traditionally, RTP aims to order regression tests to expose code bugs faster during software evolution. Inspired by traditional RTP, we aim to leverage RTP techniques to order ctests to substantially speed up misconfiguration detection for configuration changes.

We extensively evaluate 84 RTP techniques on the large Ctest dataset [52], with 7,974 ctests for five open-source projects and 66 real-world configuration change files collected from public Docker images that have some misconfigured parameter values. Our experiments with configuration changes do *not* involve code changes, matching realistic scenarios where only a new configuration is about to be deployed. We start with 16 basic RTP techniques: (1) randomized as the baseline, (2) traditional techniques based on code coverage, (3) quickest-time-first (QTF) technique, (4) recently proposed techniques based on information retrieval (IR), and (5) our novel configuration-specific RTP techniques.

We next enhance the basic RTP techniques using two sources of inspiration. First, using the idea of *cost-cognizant* RTP [120, 121], we enhance basic RTP techniques with the test execution time to design *hybrid* RTP techniques. Second, inspired by cross-checking configurations of multiple system instances used in troubleshooting systems such as the Microsoft PSS [152, 180, 181], we design a new family of *peer-based* RTP techniques that consider the test outcomes of ctests on related configuration changes. The insight is to prioritize earlier ctests that detected misconfigurations of a parameter in peer deployments, because these ctests are likely effective for the parameter change regardless of the value. Following Microsoft PSS, our peer-based RTP techniques are privacy preserving and do not use potentially sensitive value information of peer deployments but use only parameter names.

Our study leads to the following key findings:

- Among basic techniques, QTF yields competitive performance and often outperforms sophisticated techniques (e.g., based on code coverage or IR) and even some configuration-specific techniques (e.g., based on parameter-coverage and stack traces) by up to 22% (using an APFDc-like metric, §3.4.2).

- Hybrid RTP techniques that enhance basic techniques with the test execution time improve the performance of basic techniques by up to 27%. Our results confirm that hybrid, *cost-cognizant* RTP techniques are effective, even in the new domain of configuration testing.

- Peer-based RTP techniques can outperform other techniques and improve the performance of RTP even further by 15%. The results encourage sharing configuration test outcomes for the same project: "*make friends and don't test alone!*"

This work makes the following contributions:

- We reduce the time to find misconfigurations, one of the main challenges of adopting configuration testing in real-world continuous deployment process.

- We evaluate 84 traditional and ctest-specific RTP techniques for configuration testing, and we have released our code and data at `https://github.com/xlab-uiuc/ctest_prio_art`.

- We analyze the effectiveness of RTP for ctests and find highly promising results for reducing the time to find test failures and thus detecting misconfigurations early.

## 3.2 BACKGROUND ON CONFIGURATION TESTING

Configuration testing is a testing technique for detecting misconfigurations (manifesting as failing tests) to prevent them from being deployed to production systems. The basic idea is to connect software tests with the specific configuration to be deployed. In this way, configuration testing can test configuration changes in the context of code that is affected by the changed configuration. A configuration test case (*ctest*) is parameterized by a set of configuration parameters. Running a ctest instantiates each of its input parameters with an actual configuration value to be deployed to production. Like regular software tests, ctests exercise the program and check (via assertions) that program behavior satisfies certain properties (e.g., correctness, performance, security). Figure 3.1 illustrates an example ctest from prior work [52].

Ctest (configuration testing) differs from approaches that explore *multiple* configurations, e.g., configuration-aware testing, combinatorial testing, or misconfiguration-injection testing [145, 146, 182, 183, 184, 185, 186, 187], which sample representative configurations or misconfigurations through systematic or random exploration of the enormous space of value combinations. Systematic exploration can be prohibitively expensive due to combinatorial explosion [184], while random exploration can have a low probability of covering all the values that will be deployed [185]. Ctest has neither the cost of systematic exploration nor the low coverage of random exploration. Ctest focuses on testing only *one* specific configuration that is to be deployed to the production system.

A ctest $\hat{t}(\hat{P})$ is parameterized by a set of configuration parameters $\hat{P}$. Running a ctest instantiates each parameter $p \in \hat{P}$ with a concrete value as an argument. $\hat{P}$ is typically a small subset of all the configuration parameters (denoted as $\mathbb{P}$).

A system configuration is defined as the values of *all* the configuration parameters, denoted as $C = \bigcup_{i=1..|\mathbb{P}|}\{(p_i \mapsto v_i)\}$, i.e., it assigns a value $v_i$ to every parameter $p_i \in \mathbb{P}$. Running a ctest instantiates each parameter $p_i \in \hat{P}$ with its value in the system configuration $v_i$ such that $(p_i \mapsto v_i) \in C$.

A configuration change updates the values of a subset of the configuration parameters. A configuration change is in the form of a configuration file diff $D$. To test a given $D$, not all available ctests are run. A ctest $\hat{t}(\hat{P})$ is selected to test a given $D$ if at least one configuration parameter in $D$ is in the input parameter set $\hat{P}$. A configuration diff, $D$, passes if all *selected* ctests pass, and it fails if any selected ctest fails. Figure 3.2 gives an example of configuration testing for a given configuration file diff.

Overall, ctests check whether the configuration to be deployed has some misconfigurations, which will manifest as ctest failure(s). RTP for ctests pushes this further by trying to detect

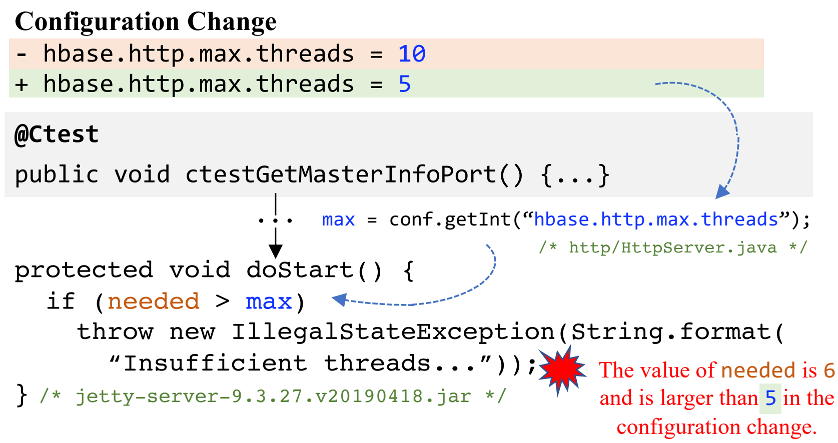


Figure 3.1: **A ctest which exercises `doStart` with the value to be changed and detects the misconfiguration.**

these misconfigurations, if any, as soon as possible by first running ctests that are more likely to fail for the new configuration.

## 3.3 APPLIED AND PROPOSED RTP TECHNIQUES

We next present all the RTP techniques we study for reducing the cost of detecting misconfigurations in configuration testing. §3.3.1 presents basic RTP techniques that do not require peer configuration changes, while §3.3.2 presents basic RTP techniques that analyze the correlation between peer configuration changes and test failures to achieve more precise test prioritization. Lastly, §3.3.3 further introduces hybrid RTP techniques that combine basic peer-based or non-peer-based techniques with test execution time. Table 3.1 summarizes the notation for all evaluated RTP techniques.

### 3.3.1 Non-peer-Based RTP

The non-peer-based RTP techniques include both traditional RTP techniques widely studied for regression testing (§3.3.1) and new RTP techniques we design for configuration testing (§3.3.1).

**Traditional RTP techniques** We study the following traditional RTP techniques:

*Code-Coverage-Based RTP.* RTP techniques based on code coverage have been extensively evaluated [14, 35, 47] and are still widely used for comparisons against newly proposed techniques [40, 188]. Code-coverage-based RTP techniques determine the test execution order

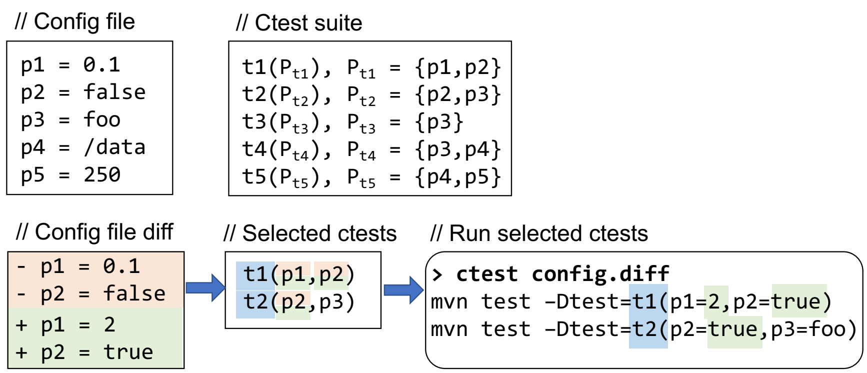


Figure 3.2: **An overview of configuration testing for a configuration file diff. Only `t1` and `t2` are selected to run because they may be affected by the configuration change. The ctest framework [189] is built on top of Maven.**

based on the code coverage of each test. For example, the *total* technique sorts tests in the descending order of the number of code elements (e.g., methods or statements) covered by each test, while the *additional* technique sorts tests in the descending order of the number of code elements covered by each test but uncovered by the already prioritized tests [60]. In the literature, code-coverage-based RTP has been widely studied at both the method and statement granularities [47]. Thus, we also evaluate *total* and *additional* code-coverage-based RTP at both method (denoted as $\text{Cov}^m_{tot}$ and $\text{Cov}^m_{add}$) and statement granularity (denoted as $\text{Cov}^s_{tot}$ and $\text{Cov}^s_{add}$).

*IR-Based RTP.* Techniques based on Information Retrieval (IR) have been recently proposed and shown effective in Regression Test Prioritization [39, 40]. IR-based techniques transform the RTP problem into an IR problem and address it with off-the-shelf retrieval models (e.g., TF-IDF [80] and BM25 [81]). A typical IR-based technique extracts code tokens from test files to form a corpus of *documents*, and represents code change information (e.g., tokens extracted from code change diff) as the *query*. In this way, a similarity value can be computed between the query and each test document. Tests that are more similar to code changes are prioritized earlier to detect problematic changes faster. We implement and evaluate the $\text{IR}_{high}$ and $\text{IR}_{low}$ techniques with BM25, as it has shown the best results [39, 40].

*QTF-Based RTP.* The Quickest-Time-First (QTF) technique simply orders all the tests in the ascending order of their execution time in prior testing runs [12]. Although simple, the QTF technique has been shown to be competitive compared with state-of-the-art RTP techniques for regression testing [61]. Therefore, we also evaluate QTF in the context of configuration testing.

Table 3.1: **Notation for all evaluated RTP techniques.**

| RTP category | Notation |
|---|---|
| ***Traditional (§3.3.1)*** | |
| Method-level code-coverage-based | $\text{Cov}^m$ |
| Statement-level code-coverage-based | $\text{Cov}^s$ |
| IR-based with high tokenization | $\text{IR}_{high}$ |
| IR-based with low tokenization | $\text{IR}_{low}$ |
| Quickest-Time-First | QTF |
| ***Configuration-specific (§3.3.1)*** | |
| Change-unaware parameter-coverage-based | PC |
| Change-aware parameter-coverage-based | $\text{PC}^D$ |
| Change-unaware stack-trace-based | ST |
| Change-aware stack-trace-based | $\text{ST}^D$ |
| ***Peer-based (§3.3.2)*** | |
| All configurations | $\text{Conf}^{all}$ |
| Configurations sharing parameter changes | $\text{Conf}^{DP}$ |
| Configurations sharing parameter coverage | $\text{Conf}^{PC}$ |
| Configurations sharing root causes | $\text{Conf}^{RC}$ |
| Shared parameter coverage with peers | $\text{Para}^{PC}$ |
| Shared root causes with peers | $\text{Para}^{RC}$ |
| ***Hybrid Models (§3.3.3)*** | |
| Cost-cognizant-divide hybrids | $*\text{+CC}_{div}$ |
| Cost-cognizant-break-tie hybrids | $*\text{+CC}_{tie}$ |
| ***Others*** | |
| Total techniques | $*_{tot}$ |
| Additional techniques | $*_{add}$ |
| Randomized order | *Random* |

**Configuration-specific RTP techniques** We further design the following RTP techniques specifically for configuration testing:

*Parameter-Coverage-Based RTP.* Inspired by traditional RTP techniques based on *code* coverage, we propose novel RTP techniques based on *parameter* coverage. Following the definition of ctest (§3.2), each ctest $\hat{t}(\hat{P})$ can test a non-empty set of input configuration parameters $\hat{P}$. We treat $\hat{P}$ as the parameters covered by $\hat{t}$. We propose *total* and *additional* RTP techniques based on such parameter coverage, denoted as $\text{PC}_{tot}$ and $\text{PC}_{add}$, respectively.

We also consider the parameter change information to design *change-aware*, parameter-coverage-based RTP. For each configuration change $D$, the set of changed parameters is denoted $P_D$. For each ctest $\hat{t}(\hat{P})$, we determine its priority based on the set of covered changed parameters, i.e., $\hat{P} \cap P_D$. Change-aware parameter coverage prioritizes ctests that are more relevant to the configuration change, thus can potentially detect misconfigurations earlier. We evaluate both *total* and *additional* techniques based on change-aware parameter coverage, denoted as $\mathrm{PC}^D_{tot}$ and $\mathrm{PC}^D_{add}$, respectively.

*Stack-Trace-Based RTP.* Different ctests may read and test the same parameter in different invocation contexts and thus may have different capabilities in detecting problematic parameter changes. For example, two ctests $\hat{t}_1(\hat{P}_1)$ and $\hat{t}_2(\hat{P}_2)$ may read $p \in \hat{P}_1 \cap \hat{P}_2$ in different source code locations, and the invocation contexts can be used to prioritize the two reads. Thus, we use the invocation contexts for each parameter read for more precise ctest prioritization.

A ctest $\hat{t}(\hat{P})$ instantiates each parameter $p \in \hat{P}$ by reading its value from configuration file(s) via API calls provided by the configuration management class(es) in the system. The ctest infrastructure [52, 189] intercepts the configuration APIs and logs the stack trace of each API invocation during generation of ctests from regular tests (and not necessarily during ctest execution). The set of methods within the invocation contexts for all parameter reads of each test can be extracted from the stack traces and leveraged for RTP. We implement both the *total* and *additional* techniques based on such information, denoted as $\mathrm{ST}_{tot}$ and $\mathrm{ST}_{add}$, respectively.

While $\mathrm{ST}_{tot}$ and $\mathrm{ST}_{add}$ consider all the methods from all stack traces where $\hat{t}$ reads all the parameters from $\hat{P}$, the change-aware variants for a configuration change $D$ consider all the methods from all stack traces where $\hat{t}$ reads only the parameters from $\hat{P} \cap P_D$. The *total* and *additional* techniques for this change-aware variants are denoted as $\mathrm{ST}^D_{tot}$ and $\mathrm{ST}^D_{add}$, respectively.

### 3.3.2 Peer-Based RTP

We now present a family of new ctest RTP techniques, termed *peer-based RTP*, that consider the test outcomes of ctests from related, *peer* configurations. Data from peer systems have been used in troubleshooting systems such as the Microsoft PSS [152, 180, 181], e.g., *PeerPressure* utilizes configuration data from peer machines to infer root causes of misbehavior [152]. Inspired by this idea, peer-based RTP prioritizes ctests that detected misconfigurations of a parameter in peer deployments, as these ctests are likely to be effective for the parameter change regardless of the value.

Deploying peer-based RTP can be done via a server/database that receives, anonymizes, and stores failed configurations and ctest outcomes from internal or community sources, to be used for future prioritization, e.g., *PeerPressure* utilized the GeneBank database to troubleshoot misconfigurations at Microsoft [152]. Specifically, our peer-based RTP are privacy preserving and do not use potentially sensitive values of peer deployments.

The general definition of peer-based RTP is simple. Let $D$ be a configuration change to be tested by a ctest suite $T$, and $S$ be a set of peer configuration changes ($D \notin S$) that have been tested. A peer-based RTP technique orders $T$ based on various statistics collected from $S$. Depending on the granularity of the peer analysis, we propose two categories of peer-based techniques, at the configuration granularity (§3.3.2) and the parameter granularity (§3.3.2). Given a ctest $\hat{t}$, $D$, and information from $S$, each technique computes a *set* of elements $X(\hat{t}, D, S)$ for the ctest; these sets can be ordered using a *total* ($X_{tot}$) or *additional* ($X_{add}$) approach, and we evaluate both on all categories of peer-based techniques.

We illustrate all our proposed techniques using the example shown in Figure 3.3. It contains a configuration change `D`, its ctest suite `T`, and a set of peer configuration changes `S`. Note that each change is with respect to some default configuration and lists parameters whose values changed. The root-cause information specifies the misconfigured parameter(s) that caused a ctest to fail on a configuration change (e.g., only `p3` caused `t1` to fail on `D1`). Empty cells indicate that the test passed. Thus, `T` has three types of orders for `D`: optimal (run passing `t3` last), sub-optimal (run `t3` second), and worst-case (run `t3` first).

**Techniques at the configuration granularity** We now discuss RTP techniques based on peer configuration changes at different granularity levels:

*All Configurations (Conf$^{all}$).* The Conf$^{all}$ set of each ctest $\hat{t}(\hat{P})$ is simply the set of all peer configuration changes where $\hat{t}$ failed:

$$\text{Conf}^{all}(\hat{t}, D, S) = \{D' \in S \mid \text{Fail}(\hat{t}, D')\} \tag{3.1}$$

where $\text{Fail}(\hat{t}, D')$ indicates that $\hat{t}$ failed on a peer configuration change $D'$. For the example in Figure 3.3, $\text{Conf}^{all}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}, \texttt{D4}\}$, $\text{Conf}^{all}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$, and $\text{Conf}^{all}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}, \texttt{D4}, \texttt{D5}\}$. Thus, $\text{Conf}^{all}_{tot}$ orders `T` as `t3-t1-t2`, and $\text{Conf}^{all}_{add}$ can order `T` as `t3-t2-t1` or `t3-t1-t2`. According to the root causes of `D`, both techniques only produce worst-case orders of `T`.

Conf$^{all}$ is *change-unaware* and can prioritize earlier a ctest that failed many peer configuration changes even if they share no changed parameter(s) with $D$, degrading $T$'s performance in detecting the misconfigurations in the parameters changed in $D$. Thus, all the following peer-based RTP techniques are *change-aware* and consider which parameters have changed

// Ctest suite

```
T = {t1, t2, t3}
t1(Pt1), Pt1 = {p2, p3, p4, p5}
t2(Pt2), Pt2 = {p1, p6}
t3(Pt3), Pt3 = {p2, p4}
```

// Peer config changes

```
S = {D1, D2, D3, D4, D5}
D1, PD1 = {p1, p2, p3, p4}
D2, PD2 = {p1, p2, p4}
D3, PD3 = {p1, p4}
D4, PD4 = {p4, p5, p6}
D5, PD5 = {p4}
```

// Current config change

```
D, PD = {p1, p2, p3}
```

// Root causes of ctest failures

| | t1 | t2 | t3 |
|---|---|---|---|
| D1 | {p3} | {p1} | {p4} |
| D2 | {p4} | {p1} | {p4} |
| D3 | {p4} | {p1} | {p4} |
| D4 | {p5} | | {p4} |
| D5 | | | {p4} |
| D | {p3} | {p1} | |

Figure 3.3: **An example to illustrate peer-based RTP.**

for better prioritization.

*Configurations Sharing Parameter Changes (Conf$^{DP}$).* The Conf$^{DP}$ set of each ctest $\hat{t}(\hat{P})$ restricts the set to peer configuration changes that have changed parameters in common[1] with $D$:

$$\text{Conf}^{DP}(\hat{t}, D, S) = \{D' \in S \mid P_{D'} \cap P_D \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \tag{3.2}$$

For our example, Conf$^{DP}$$(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$, because $\texttt{P}_{\texttt{D1}} \cap \texttt{P}_{\texttt{D}} = \{\texttt{p1}, \texttt{p2}, \texttt{p3}\}$, $\texttt{P}_{\texttt{D2}} \cap \texttt{P}_{\texttt{D}} = \{\texttt{p1}, \texttt{p2}\}$, and $\texttt{P}_{\texttt{D3}} \cap \texttt{P}_{\texttt{D}} = \{\texttt{p1}\}$. Similarly, Conf$^{DP}$$(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and Conf$^{DP}$$(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$. Both Conf$^{DP}_{tot}$ and Conf$^{DP}_{add}$ can produce all 6 permutations of `T` because all 3 ctests have the same priority.

While more precise than Conf$^{all}$, Conf$^{DP}$ could include $D'$ when changed parameters in common between $D'$ and $D$ are not even read by $\hat{t}$. In this way, a larger set for Conf$^{DP}$ may not indicate that $\hat{t}$ is more effective in detecting misconfigurations on the current changed parameters read by $\hat{t}$. Therefore, we next consider parameter coverage information for more precise RTP.

*Configurations Sharing Parameter Coverage (Conf$^{PC}$).* The Conf$^{PC}$ set of each ctest $\hat{t}(\hat{P})$ further restricts the set to peer configuration changes that have changed parameters in common with $D$ and also some parameter(s) in common read by $\hat{t}$:

$$\text{Conf}^{PC}(\hat{t}, D, S) = \{D' \in S \mid P_{D'} \cap P_D \cap \hat{P} \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \tag{3.3}$$

For our example, Conf$^{PC}$$(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}\}$ because $\texttt{P}_{\texttt{D1}} \cap \texttt{P}_{\texttt{D}} \cap \texttt{P}_{\texttt{t1}} = \{\texttt{p2}, \texttt{p3}\}$ and $\texttt{P}_{\texttt{D2}} \cap \texttt{P}_{\texttt{D}} \cap \texttt{P}_{\texttt{t1}} = \{\texttt{p2}\}$, while $\texttt{P}_{\texttt{D3}} \cap \texttt{P}_{\texttt{D}} \cap \texttt{P}_{\texttt{t1}} = \{\}$. Similarly, Conf$^{PC}$$(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and

[1] Note that it considers only parameter *names* and *not values*.

$\text{Conf}^{PC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}\}$. Both $\text{Conf}^{PC}_{tot}$ or $\text{Conf}^{PC}_{add}$ can order `T` as `t2-t1-t3` or `t2-t3-t1`. Either technique has 50% probability of producing an optimal or sub-optimal order of `T`, and produces no worst-case order.

$\text{Conf}^{PC}$ may still be imprecise when the exact parameter(s) that caused $\hat{t}$ to fail on $D'$ are not in $P_{D'} \cap P_D \cap \hat{P}$, which happens when the root-cause parameter(s) of $\hat{t}$ on $D'$ are not in $P_D$, thus not in $P_{D'} \cap P_D$. In such a scenario, even if the technique prioritizes ctests with larger $\text{Conf}^{PC}$, the misconfiguration detection efficiency on $D$ may not improve simply because the root-cause parameter(s) of the peer configuration changes are not in $P_D$. Therefore, we next consider the root-cause parameter information.

*Configurations Sharing Root Causes ($\text{Conf}^{RC}$).* The $\text{Conf}^{RC}$ set of each ctest $\hat{t}(\hat{P})$ further restricts the set to peer configuration changes whose root-cause misconfigured parameters are also changed in $D$:

$$\text{Conf}^{RC}(\hat{t}, D, S) = \{D' \in S \mid \text{RC}(\hat{t}, D') \cap P_D \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \tag{3.4}$$

$\text{RC}(\hat{t}, D')$ is the set of root-cause misconfigured parameter(s) that actually caused the failure of $\hat{t}$ in configuration change $D'$. Note that $\text{RC}(\hat{t}, D') \subseteq \hat{P}$ because a parameter must be read by $\hat{t}$ (i.e, in $\hat{P}$) to be a root cause of the failure of $\hat{t}$.

In Figure 3.3, $\text{Conf}^{RC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}\}$ because only $\text{RC}(\texttt{t1}, \texttt{D1}) \cap \texttt{P}_\texttt{D} = \{\texttt{p3}\}$ is non-empty. Similarly, $\text{Conf}^{RC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and $\text{Conf}^{RC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\}$. $\text{Conf}^{RC}_{tot}$ orders `T` as `t2-t1-t3`, and $\text{Conf}^{RC}_{add}$ can order `T` as `t2-t1-t3` or `t2-t3-t1`. The probability of producing an optimal order of `T` is 50–100%, and no worst-case order is produced.

While $\text{Conf}^{RC}$ is more precise than the earlier peer-based techniques, it requires to maintain the root-cause information for all failed peer configuration changes. Developers could record such information while debugging misconfigurations, but such information may not always be available (§3.4.3).

**Techniques at the parameter granularity** We now discuss our peer-based techniques based on individual parameters in peer configurations at different precision levels. $\text{Conf}^{all}$ and $\text{Conf}^{DP}$ techniques do not consider parameter coverage and thus have no parameter-granularity counterparts.

*Shared Parameter Coverage with Peers ($\text{Para}^{PC}$).* The $\text{Para}^{PC}$ set of each ctest $\hat{t}(\hat{P})$ is the set of parameters from peer configuration changes in $\text{Conf}^{PC}(\hat{t}, D, S)$ described in §3.3.2:

$$\text{Para}^{PC}(\hat{t}, D, S) = \bigcup\nolimits_{D' \in S,\ \text{Fail}(\hat{t}, D')} P_{D'} \cap P_D \cap \hat{P} \tag{3.5}$$

Collecting for each ctest the parameters instead of failed peer configuration changes explores another possibility where peer-based RTP could prioritize earlier ctests that failed on a relatively smaller number of peer configuration changes but a larger set of configuration parameters from the changes.

For our example, $\text{Para}^{PC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{p2}, \texttt{p3}\}$, $\text{Para}^{PC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{p1}\}$, and $\text{Para}^{PC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{p2}\}$. $\text{Para}^{PC}_{tot}$ orders `T` as `t1-t2-t3` or `t1-t3-t2`. $\text{Para}^{PC}_{add}$ orders `T` as `t1-t2-t3`. The probability of producing optimal orders of `T` is 50–100%, with no worst-case order produced, which is an overall improvement to the counterpart ($\text{Conf}^{PC}$) from §3.3.2.

*Shared Root Causes with Peers (Para$^{RC}$).* The $\text{Para}^{RC}$set of each test $\hat{t}(\hat{P})$ is the set of parameters from peer configuration changes in $\text{Conf}^{RC}$ $(\hat{t}, D, S)$ described in §3.3.2:

$$\text{Para}^{RC}(\hat{t}, D, S) = \bigcup\nolimits_{D' \in S,\ \text{Fail}(\hat{t}, D')} \text{RC}(\hat{t}, D') \cap P_D \tag{3.6}$$

For our example, $\text{Para}^{RC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{p3}\}$, $\text{Para}^{RC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{p1}\}$, and $\text{Para}^{RC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\}$. Both $\text{Para}^{RC}_{tot}$ and $\text{Para}^{RC}_{add}$ can order `T` as `t1-t2-t3` or `t2-t1-t3`. The probability of producing optimal orders of `T` is thus 100%, which improves over the counterpart ($\text{Conf}^{RC}$) from §3.3.2.

### 3.3.3 Hybrid RTP

Various RTP techniques have been reported to benefit by additionally considering test execution time [12, 40, 120, 121]. For example, the *cost-cognizant additional* code-coverage-based technique [121], which considers the additional code coverage per time unit for each test, can substantially improve the *additional* technique in terms of the time for detecting regression faults. Therefore, besides all the basic RTP techniques introduced in §3.3.1–3.3.2, we introduce hybrid techniques that combine the basic techniques with test execution time. Inspired by the prior work in cost-cognizant RTP [12, 40, 121], we define and implement two generic cost-cognizant hybrid RTP models. We apply both models to all aforementioned RTP techniques to construct hybrid RTP techniques, and evaluate their prioritization effectiveness for ctests.

**Cost-cognizant-divide** Following the traditional cost-cognizant RTP techniques, the Cost-cognizant-divide ($\text{CC}_{div}$) model constructs hybrid RTP techniques that prioritize tests in the descending order of the input tests' priority values per time unit, i.e., the original priority values divided by the test execution time. For example, a hybrid *additional* code-coverage RTP technique with $\text{CC}_{div}$ model ($\text{Cov}^{m}_{add}$+$\text{CC}_{div}$) prioritizes the test with the largest value of the number of uncovered methods divided by the test execution time. $\text{CC}_{div}$ is semantically equivalent to *CC* described in §2.2.5, where $\mathrm{CC}_{div}$ prioritizes tests with higher final scores while *CC* prioritizes tests with lower final scores.

**Cost-cognizant-break-tie** We further study how to use time information in the Cost-cognizant-break-tie ($\mathrm{CC}_{tie}$) model. It constructs hybrid RTP techniques that order tests that are "tied" by the basic RTP technique (i.e., multiple tests have the same priority score) in the ascending order of their test execution time (as QTF). For example, hybrid technique $\mathrm{Cov}^{m}_{tot}$+$\mathrm{CC}_{tie}$ orders the tied tests with QTF when multiple tests have the same amount of covered methods.

## 3.4 EXPERIMENTAL SETUP

### 3.4.1 Research Questions

This work aims to answer the following research questions:

- **RQ1:** How do basic non-peer-based RTP techniques perform in detecting real-world misconfigurations?
- **RQ2:** How do hybrid non-peer-based RTP techniques perform compared with the basic non-peer-based techniques?
- **RQ3:** How do peer-based RTP techniques perform compared to non-peer-based RTP techniques?

### 3.4.2 Metrics

Common metrics to evaluate traditional RTP techniques are Average Percentage of Faults Detected (APFD) and Average Percentage of Faults Detected per Cost (APFDc). §2.4.1 from Chapter 2 has provided more details on both metrics. APFDc is a *cost-aware* variant of APFD that considers the cost of test executions [120, 121]. In the context of configuration testing, however, test failures are caused by misconfigurations and not (code) regression faults. Thus, we adapted the definition of APFD and APFDc to derive two new metrics for evaluating RTP techniques for configuration testing: Average Percentage of Misconfigurations Detected (APMD) and Average Percentage of Misconfigurations Detected per Cost (APMDc). The *only* difference in the definitions is that APFD and APFDc consider code bugs, while our metrics consider misconfigurations. Higher APMD and APMDc values (i.e., closer to

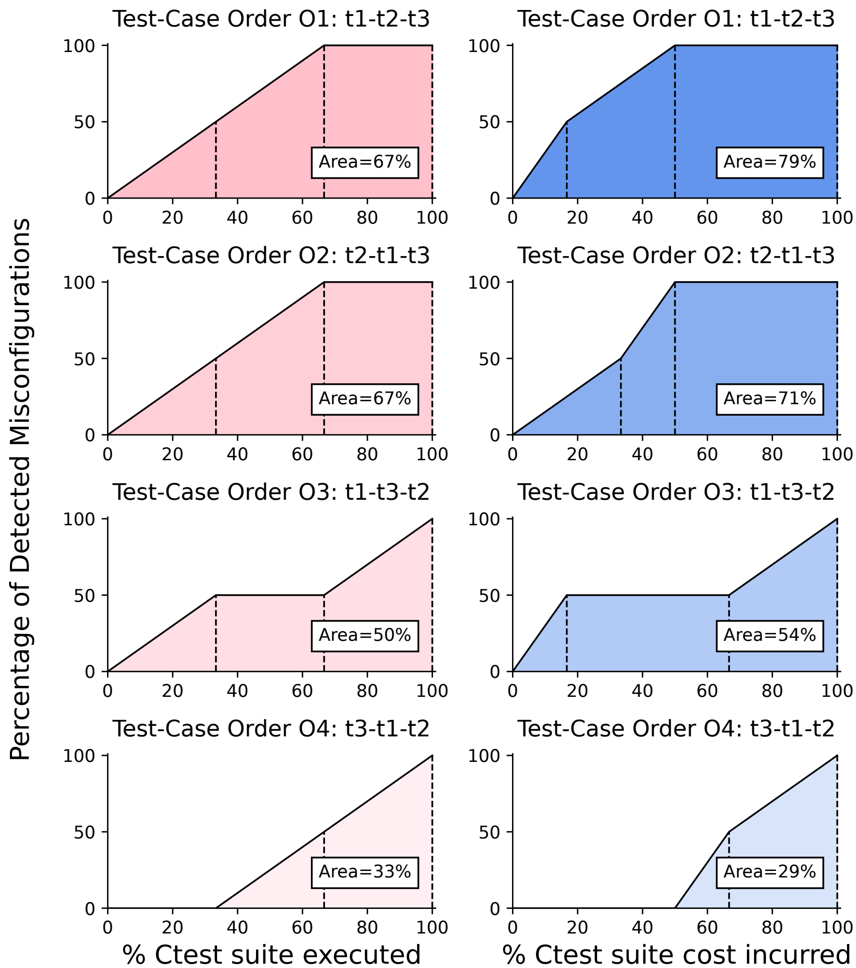


Figure 3.4: **An example to illustrate APMD (first column) and APMDc (second column) for four test-case orders.**

1.0) indicate all misconfigurations are detected earlier, while lower values (i.e., closer to 0.0) indicate all misconfigurations are detected later.

We illustrate APMD and APMDc for the following example scenario. Let `T` $=$ {`t1`, `t2`, `t3`} be a ctest suite for a configuration change `D`, $\mathtt{P_D} = \{\mathtt{p1}, \mathtt{p2}\}$; `t1` failed on `p1`, `t2` failed on `p2`, and `t3` passed; the execution costs of `t1`, `t2`, and `t3` are 1, 2, and 3 seconds, respectively. Figure 3.4 illustrates the APMD and APMDc values for four orders (i.e., `O1`, `O2`, `O3`, and `O4`) of `T`; from left to right, `O1` and `O2` are optimal (run passing `t3` last), `O3` is sub-optimal (runs `t3` second), `O4` is the worst-case (runs `t3` first).

**Average Percentage of Misconfigurations Detected (APMD)** APMD is our adaption of APFD [35] in the context of configuration testing. Let $n$ be the number of configuration tests to be run, $m$ be the number of misconfigured parameters in the configuration change, and $TF_i$ be the position (in the order) of the first failed configuration test that detects the $i^{th}$ misconfigured parameter:

$$APMD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{3.7}$$

APMD computes the area under the curve between the percentage of detected misconfigurations in a configuration change and the percentage of the test suite executed, as illustrated in Figure 3.4.A larger area always implies faster overall detection for *all* misconfigurations in the current configuration change. For example, `O1` detects 50% of the misconfigurations in `D` (i.e., `p1`) after executing 33.3% of `T` (i.e., `t1`), and `O1` detects 100% of the misconfigurations in `D` (i.e., `p1`, `p2`) after executing 66.7% of `T` (i.e., `t1`, `t2`). Thus, the APMD value of `O1` is 67% as $1 - \frac{1+2}{3 \cdot 2} + \frac{1}{2 \cdot 3} = 0.67$ using Formula 3.7. However, like APFD, APMD is *cost-unaware*. Although `O1` and `O2` have the same APMD value, `O1` is actually more cost-effective than `O2` because `O1` halves the cost to detect the first misconfiguration compared to `O2`.

**Average Percentage of Misconfigurations Detected per Cost (APMDc)** APMDc considers the cost, as in APFDc, which commonly uses test execution time [61, 122]. Let $n$, $m$, and $TF_i$ be the same as for APMD, and $t_j$ be the execution time[2] of the $j^{th}$ configuration test in the prioritized order:

$$APMDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i})}{\sum_{j=1}^{n} t_j \times m} \tag{3.8}$$

Similar to APMD, APMDc computes the area under the curve between the percentage of detected misconfigurations in a configuration change and the percentage of its test suite *cost* incurred, as illustrated in Figure 3.4. For example, the total cost of `T` is 6 seconds; `O1` detects 50% of the misconfigurations in `D` after incurring 17% of the total cost (i.e., 1 second from `t1`), and `O1` detects 100% of the misconfigurations in `D` after incurring 50% of the total cost (i.e., 1 second from `t1` and 2 seconds from `t2`). Thus, the APMDc value of `O1` is 79% as $\frac{(1+2+3-\frac{1}{2}\cdot 1)+(2+3-\frac{1}{2}\cdot 2)}{(1+2+3)\cdot 2} = 0.79$ using Formula 3.8. The APMDc value of `O2` is lower than that of `O1`, showing that APMDc can properly distinguish the more cost-effective order.

APMDc, like APFDc, more precisely captures the cost/time that developers would actually experience to detect *all* misconfigurations. Prior studies [61, 121] show that APFD can rank RTP techniques for regression faults differently than APFDc, and thus APFD is less reliable. We still evaluate both APMD and APMDc to check if the same holds for RTP techniques in

[2]Note that the time for APMDc is measured when running tests on the changed configuration, while the time used to prioritize tests (in QTF and hybrid techniques) is from running tests prior to the change.

Table 3.2: **Configuration change dataset.**

| **Project** | **Version** | **#Change** | **Avg #Param** | | **Avg #Ctest** |
|---|---|---|---|---|---|
| | | | **All** | **Misconf.** | |
| HCommon | 2.8.5 | 20 | 3.75 | 1.05 | 955.75 |
| HDFS | 2.8.5 | 16 | 5.19 | 1.31 | 1680.12 |
| HBase | 2.2.2 | 12 | 8.33 | 1.92 | 1254.25 |
| ZooKeeper | 3.5.6 | 14 | 6.57 | 1.71 | 74.36 |
| Alluxio | 2.1.0 | 4 | 13.75 | 1.25 | 949.00 |

the new application domain of configuration testing.

### 3.4.3 Dataset Collection

We build our evaluation dataset from the Ctest dataset [52], which contains 66 configuration changes with misconfigurations collected from real-world Docker images on Docker Hub [190, 191] for five widely-used projects: HCommon, HDFS, HBase, ZooKeeper, and Alluxio. The dataset also includes ctests for these projects. To compute APMD and APMDc, we ran ctests on all configuration changes and collected test outcomes and execution time.

We also identified the root-cause misconfigured parameter(s) for each test failure. Root-cause information is necessary to precisely compute APMD and APMDc for any RTP technique. (Prior research on regression testing has likewise had to map each test failure to the code fault(s) to compute APFD and APFDc [120, 121].) It is also necessary for constructing peer information for some peer-based RTP techniques (§3.3.2). Automated root-cause localization such as delta debugging [192] is *not* applicable because misconfigurations are not monotone due to configuration dependencies [171]. While several advanced misconfiguration-diagnosis techniques exist [193, 194, 195, 196, 197, 198], we manually localized the root causes to ensure the precision; most failure-inducing misconfigured parameters can be easily identified as root causes by inspecting failure logs. Besides the techniques that need root causes, all others are fully *automatic*. We excluded flaky tests from the dataset using best-effort reruns [199]. Table 3.2 shows the version, number of configuration changes, and average numbers of parameters, misconfigured parameters, and ctests per change of each project.

### 3.4.4 Implementation

We implemented all the studied RTP techniques in Python 3. Our infrastructure for test information collection and test prioritization is written in Java and Python.

**Test information collection** We next discuss how we collected the necessary test information required by the studied RTP techniques. We used OpenClover [200] to collect code coverage at statement and method granularity (§3.3.1). To collect ctest execution time (§3.3.1, §3.3.3), we ran each ctest 5 times *prior* to configuration changes on the same machine, and used the averages as the time for prioritization. Execution times reported as 0.000 by Maven are changed to 0.001 because Maven rounds off time to 3 decimal places. For IR data (§3.3.1), we implemented a parser in Java 8 with JavaParser 3.18.0 [201] to collect tokens from test class files for all evaluated projects. We also performed an automated step of ctest generation with the open-sourced Ctest prototype [189] to collect invocation contexts for stack-trace-based RTP techniques (§3.3.1). We directly collected parameter coverage (§3.3.1) from open-sourced ctests [189]. Inspired by cross validation [202], for each configuration change in the dataset, we treated the other configuration changes from the same project as its peer configuration changes (§3.3.2).

**Test prioritization** Because most of the studied RTP techniques are built based on the traditional *total* and *additional* techniques, we implemented generic *total* and *additional* RTP functions following the traditional definitions. We also implemented the QTF RTP technique according to the traditional definition.

For IR-based techniques (§3.3.1), the choice of retrieval model and the approach to construct data objects can substantially affect the performance [39, 40]. Our IR-based RTP techniques used the BM25 retrieval model [81], as well as $High_{token}$ and $Low_{token}$ for data-object construction, which have been demonstrated to achieve state-of-the-art performance by Peng *et al.* [40]. Specifically, our $\text{IR}_{high}$ RTP technique used the $High_{token}$ construction, where a document only contains identifiers from a test file. Similarly, our $\text{IR}_{low}$ RTP technique used the $Low_{token}$ construction, where a document contains identifiers, comments, and string literals from a test file. We collected documents at test-case level utilizing Saha *et al.*'s approach [39], treating each test method as a test case, as common in JUnit. We processed documents following standard tokenization steps [40]. Unlike code changes, which can contain a variety of elements, a configuration change only contains names and values of the changed parameters. To construct query for each configuration change, we only use tokenized names of the changed parameters, because actual configuration values are often too specific to be found in the test code.

### 3.4.5 Experimental Procedure

To compare all the studied RTP techniques, we also implemented a randomized RTP technique (denoted as *Random*) to serve as baseline, which shuffles ctests with a random seed. For all studied RTP techniques with no break-tie strategy specified, ties are also broken with random seeds. Thus, to amount for different results from randomization, we ran each RTP technique on every configuration change 100 times, each time with a different seed. Specifically, for each RTP technique, we did the following: (1) load the collected configuration change dataset, i.e., ctest outcome, execution time, root-cause analysis results under configuration changes, etc. (§3.4.3); (2) load the test information for the current technique (§3.4.4); (3) select a configuration change $D$ that has not been run under the current technique; (4) initialize a random seed; (5) apply current technique to order the ctest suite of $D$; (6) compute APMD and APMDc of the ctest suite order based on the collected ctest outcome, execution time, and root causes; (7) repeat steps (4)–(6) 100 times; (8) repeat steps (3)–(7) on all 66 configuration changes.

In total, we evaluated 84 RTP techniques for configuration testing: 16 basic non-peer-based techniques, of which 15 are described in §3.3.1 and 1 is randomized baseline; 12 basic peer-based techniques described in §3.3.2; 32 hybrid non-peer-based techniques, of which 16 each use $CC_{div}$ and $CC_{tie}$ models (§3.3.3); and 24 hybrid peer-based ones. In total, we performed 554,400 (84*66*100) unique RTP executions.

## 3.5 EVALUATION

### 3.5.1 RQ1: Basic Non-peer-Based RTP

This RQ compares non-peer-based traditional and configuration-specific RTP techniques on APMD and APMDc. In Figure 3.5, each violin plot and its embedded box plot show the distribution of APMD or APMDc values per project per run for each RTP technique. Each volin/box plot represents 500 (5*100) data points, for five projects and 100 random seeds. The white bar in each box plot shows the median, while the dot shows the (arithmetic) mean over all the data points for each RTP technique.

We further show the Tukey HSD test [127] results in Table 3.3. Tukey HSD is a post-hoc test based on the studentized range distribution; it compares all possible pairs of means to find out which specific groups' means (compared with each other) are significantly different. We performed this test on APMD and APMDc values to check for statistically significant differences among the studied RTP techniques [40]. In the table, Column "Average" shows

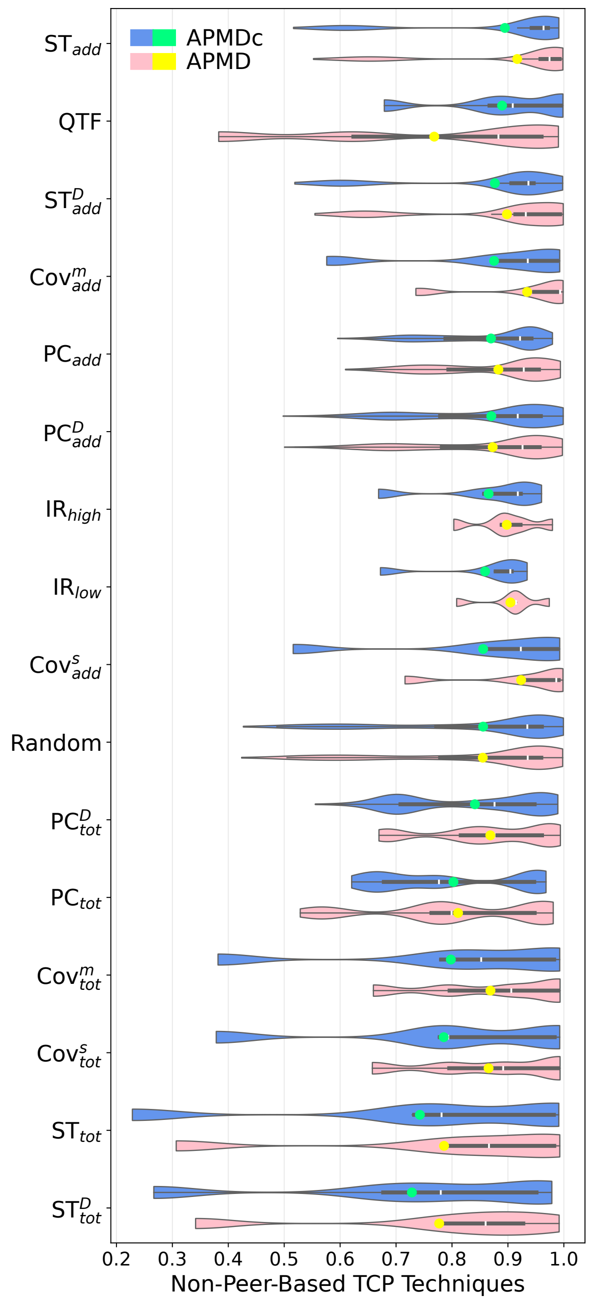


Figure 3.5: **Distribution of APMDc and APMD values for basic non-peer-based RTP techniques (sorted by average APMDc).**

Table 3.3: **Average APMDc (A.c) and APMD (A.) values and Tukey HSD groups of basic non-peer-based RTP techniques.**

| RTP | HCommon | | HDFS | | HBase | | ZooKeeper | | Alluxio | | Average | | Group | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **A.c** | **A.** | **A.c** | **A.** | **A.c** | **A.** | **A.c** | **A.** | **A.c** | **A.** | **A.c** | **A.** | **A.c** | **A.** |
| $\text{ST}_{add}$ | .990 | .998 | .608 | .652 | .971 | .974 | .963 | .964 | .941 | .995 | .895 | .917 | A | AB |
| QTF | .998 | .990 | .865 | .621 | .997 | .963 | .909 | .883 | .679 | .385 | .890 | .768 | AB | G |
| $\text{ST}^{D}_{add}$ | .998 | .999 | .604 | .646 | .908 | .928 | .934 | .923 | .939 | .996 | .877 | .898 | ABC | BCD |
| $\text{Cov}^{m}_{add}$ | .990 | .999 | .587 | .742 | .871 | .945 | .993 | .993 | .935 | .993 | .875 | .934 | ABC | A |
| $\text{PC}_{add}$ | .929 | .992 | .726 | .739 | .948 | .948 | .935 | .928 | .811 | .807 | .870 | .883 | ABC | CDE |
| $\text{PC}^{D}_{add}$ | .995 | .996 | .685 | .681 | .947 | .947 | .917 | .928 | .807 | .813 | .870 | .873 | ABC | CDE |
| $\text{IR}_{high}$ | .918 | .895 | .855 | .887 | .960 | .980 | .925 | .925 | .669 | .803 | .865 | .898 | ABC | BCD |
| $\text{IR}_{low}$ | .934 | .911 | .876 | .914 | .904 | .974 | .909 | .915 | .672 | .808 | .859 | .904 | ABC | ABC |
| $\text{Cov}^{s}_{add}$ | .991 | .998 | .520 | .718 | .853 | .922 | .993 | .993 | .923 | .986 | .856 | .924 | BC | AB |
| *Random* | .992 | .993 | .577 | .578 | .947 | .946 | .939 | .938 | .823 | .820 | .856 | .855 | BC | E |
| $\text{PC}^{D}_{tot}$ | .985 | .993 | .874 | .868 | .701 | .685 | .947 | .961 | .698 | .837 | .841 | .869 | C | DE |
| $\text{PC}_{tot}$ | .958 | .979 | .647 | .567 | .776 | .762 | .948 | .947 | .685 | .799 | .803 | .811 | D | F |
| $\text{Cov}^{m}_{tot}$ | .985 | .992 | .382 | .660 | .778 | .793 | .993 | .993 | .852 | .906 | .798 | .869 | D | DE |
| $\text{Cov}^{s}_{tot}$ | .986 | .992 | .378 | .658 | .776 | .793 | .993 | .993 | .793 | .892 | .785 | .865 | D | E |
| $\text{ST}_{tot}$ | .985 | .985 | .230 | .308 | .781 | .779 | .986 | .991 | .730 | .866 | .743 | .786 | E | FG |
| $\text{ST}^{D}_{tot}$ | .978 | .992 | .268 | .342 | .780 | .774 | .940 | .917 | .675 | .860 | .728 | .777 | E | G |

the mean APMDc ("A.c") and APMD ("A.") values per technique (same as the dots in Figure 3.5). Importantly, Column "Group" presents the results of the Tukey HSD test. Tukey HSD puts techniques into different groups if they have statistically significant differences. Groups are named by capital letters, where "A" denotes the best group, and the performance degrades in alphabetical order. A technique having multiple letters has performance between these letter groups. From the results, we make the following observations.

**Total vs. Additional** We can observe that *additional* techniques tend to outperform *total* ones on APMD and APMDc. For example, stack-trace-based RTP has the highest average APMDc value (0.895) among all studied techniques when using the *additional* strategy, but it has one of the lowest average APMDc values (0.743) when using the *total* strategy. Similar findings can be observed for code-coverage-based RTP on the APMD values, as well as other studied techniques. The Tukey HSD test results also confirm our observation, e.g., for APMDc, almost all *additional* techniques are in better Tukey HSD groups than *Random*, while all *total* techniques are in worse groups. The key reason is that the *additional* strategy considers the impact of already prioritized tests and tends to execute more diverse tests, which can expose misconfigurations earlier. This finding is consistent with prior studies on traditional regression testing, which showed that *additional* techniques generally perform better than *total* techniques in RTP [35, 39, 137, 203]. In summary, we are the first to find that the *additional* strategy is preferred over the *total* strategy even for configuration testing.

**Comparing coverage criteria** From Table 3.3, we can observe that traditional code coverage at method granularity is still effective in Regression Test Prioritization for configuration testing. For example, the *additional* code-coverage-based RTP techniques outperformed others in APMD, in which $\mathrm{Cov}^{m}_{add}$ has the best performance. The reason is that a ctest with higher code coverage is more likely to exercise its covered configuration parameters in more project components, and thus has a higher chance to detect potential misconfiguration(s). Moreover, configuration-specific coverage criteria can outperform traditional code coverage on APMDc. For example, the *additional* stack-trace-based RTP ($\mathrm{ST}_{add}$) is in a statistically better group than $\mathrm{Cov}^{m}_{add}$ in APMDc. The potential reason is that ctests with larger traditional code coverage also tend to run slower; in contrast, configuration-specific coverage can also effectively guide misconfiguration detection, but ctests with higher configuration-specific coverage do not necessarily run slower.

Among the configuration-specific coverage criteria, the best stack-trace-based RTP technique ($\mathrm{ST}_{add}$) usually performs better than the best parameter-coverage-based RTP techniques ($\mathrm{PC}_{add}$) on APMD and APMDc. The reason is that different ctests reading the same parameters may have greatly different invocation contexts and thus may have different capabilities in detecting misconfigurations. Another interesting finding is that both the best stack-trace-based and parameter-coverage-based techniques tend to outperform their change-aware counterparts. For example, $\mathrm{ST}_{add}$ achieves 0.895 (0.917) in APMDc (APMD), while $\mathrm{ST}^{D}_{add}$ has 0.877 (0.898). The reason is that majority of configuration changes are relatively small. Thus, the *additional* techniques cannot easily prioritize ctests with new change-aware configuration-specific coverage, and behave as random baseline when no ctests have new coverage.

**IR-based RTP** Although IR-based techniques have been recently claimed to be the state-of-the-art in Regression Test Prioritization and unsafe selection for traditional regression testing [39, 40], they never perform the best in configuration testing on APMD and APMDc. There are several potential reasons. First, configuration changes are usually small and less informative than code changes. Second, unlike code changes, configuration changes have no surrounding context [40]. Thus, each change query is built simply from tokenized names of changed parameters, which can often be too ambiguous. For example, the query built from changed parameters {`dataDir, dataLogDir`} is a bag of words `[data, dir, data, log, dir]`, which can be common in test files. Another interesting finding is that IR-based techniques never perform the worst in configuration testing. In fact, IR-based techniques are the most stable ones: in Figure 3.5, the plots for IR-based techniques are more concentrated near the median for both APMD and APMDc. The stability across runs for each project comes from test documents being large and diverse, so few ties are produced. Also, IR-based techniques prioritize ctests whose documents are more related to the names of changed parameters.

**QTF-based RTP** QTF has the second *highest* average APMDc, but the absolutely *lowest* average APMD across all projects. The reason is that a considerable portion of ctests are transformed from unit tests that have rather short execution time. Thus, QTF prioritizes these faster ctests first and can end up running many more ctests than other RTP techniques before detecting the misconfigurations, leading to low APMD values. However, when considering the test cost for APMDc, QTF is much more cost-effective, because the ctests prioritized earlier have short execution time. For example, on HDFS, many ctests prioritized earlier cost less than 0.1 second.

**APMD vs. APMDc** While the rankings of many RTP techniques are similar by both APMD and APMDc, the diametrically opposite ranking of QTF when using APMD and APMDc indicates that APMD is *not* appropriate and can be misleading for configuration testing. This finding is consistent with prior work on traditional regression testing: APFD has been shown to be misleading in comparing RTP techniques because it does not consider test execution time [61, 121]. Therefore, in the following sections, we only focus on the APMDc results. Moreover, the high effectiveness of QTF in APMDc also inspired us to combine the basic techniques with test execution time information for hybrid techniques (§3.3.3).

**Per-project results** Table 3.3 presents the detailed average results for each studied project. The main findings—such as *additional* is better than *total*, and QTF is competitive—from the overall distribution of APMD/APMDc across all projects are also similar for individual projects. In each project, QTF is also among the best, APMD can be misleading as it ranks QTF as one of the worst techniques.

### 3.5.2 RQ2: Hybrid Non-peer-Based RTP

This RQ evaluates the effectiveness of hybrid non-peer-based RTP techniques with two hybrid models discussed in §3.3.3. Figure 3.6 shows the distribution of APMDc values for each hybrid non-peer-based technique: the names of corresponding basic non-peer-based techniques are shown on the y-axis, while the green/orange violin plots show the distribution

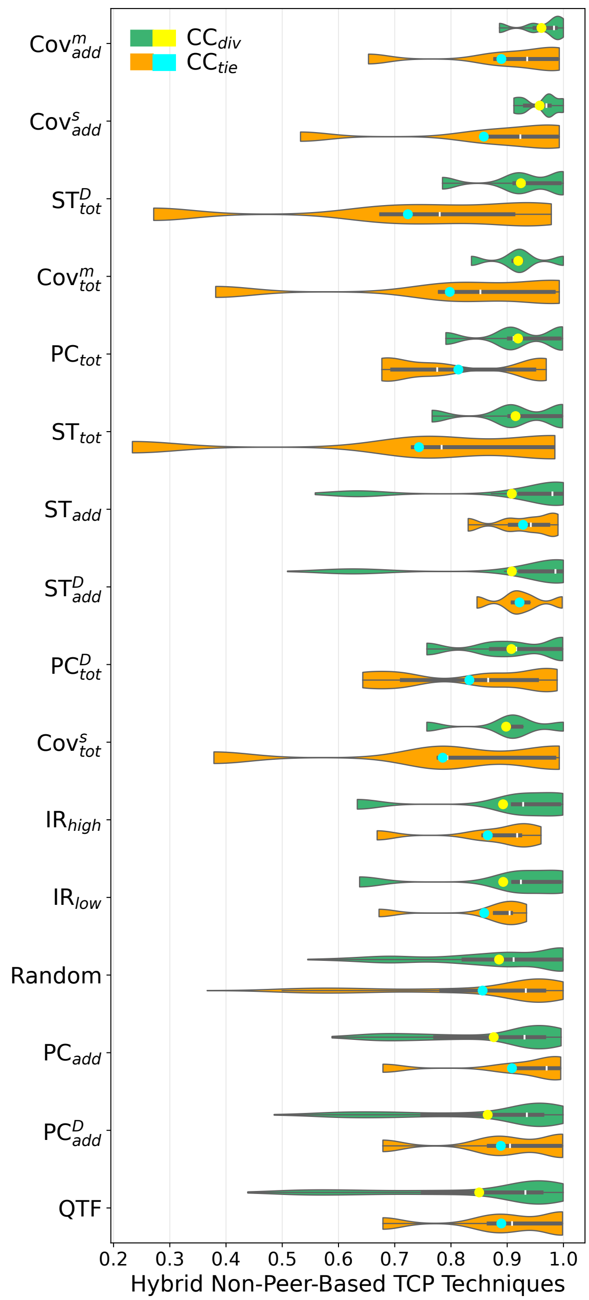


Figure 3.6: **Distribution of APMDc values for hybrid non-peer-based RTP techniques (sorted by average APMDc from CC$_{div}$).**

Table 3.4: **APMDc values and Tukey HSD groups of hybrid non-peer-based RTP.**

| RTP | HCommon | | HDFS | | HBase | | ZooKeeper | | Alluxio | | Average | | Group | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ | $CC_{div}$ | $CC_{tie}$ |
| $Cov^m_{add}$ | 1.00 | .990 | .886 | .653 | .989 | .876 | .946 | .993 | .983 | .936 | .961 | .890 | A | BC |
| $Cov^s_{add}$ | 1.00 | .991 | .912 | .532 | .978 | .853 | .929 | .993 | .970 | .923 | .958 | .858 | A | CD |
| $ST^D_{tot}$ | 1.00 | .978 | .931 | .271 | .996 | .780 | .910 | .913 | .785 | .674 | .924 | .723 | B | G |
| $Cov^m_{tot}$ | 1.00 | .985 | .928 | .382 | .925 | .778 | .909 | .993 | .836 | .852 | .920 | .798 | BC | F |
| $PC_{tot}$ | .998 | .969 | .901 | .693 | .995 | .776 | .911 | .950 | .791 | .677 | .919 | .813 | BC | EF |
| $ST_{tot}$ | .998 | .985 | .901 | .233 | .997 | .783 | .911 | .984 | .766 | .730 | .915 | .743 | BCD | G |
| $ST_{add}$ | 1.00 | .990 | .635 | .831 | 1.00 | .975 | .927 | .902 | .980 | .942 | .908 | .928 | BCDE | A |
| $ST^D_{add}$ | 1.00 | .998 | .626 | .846 | .999 | .918 | .931 | .907 | .986 | .940 | .908 | .922 | BCDE | AB |
| $PC^D_{tot}$ | .999 | .989 | .869 | .866 | .997 | .710 | .916 | .955 | .757 | .643 | .907 | .833 | BCDE | DE |
| $Cov^s_{tot}$ | 1.00 | .986 | .927 | .378 | .904 | .776 | .902 | .993 | .757 | .794 | .898 | .785 | CDEF | F |
| $IR_{high}$ | .999 | .918 | .928 | .855 | .995 | .960 | .908 | .925 | .633 | .669 | .893 | .865 | DEF | CD |
| $IR_{low}$ | .999 | .934 | .924 | .876 | .996 | .904 | .908 | .909 | .638 | .672 | .893 | .859 | DEF | CD |
| *Random* | .999 | .992 | .839 | .575 | .996 | .951 | .911 | .937 | .683 | .826 | .886 | .856 | EFG | CD |
| $PC_{add}$ | .992 | .995 | .690 | .970 | .953 | .995 | .930 | .904 | .815 | .679 | .876 | .909 | FG | AB |
| $PC^D_{add}$ | .996 | .999 | .628 | .865 | .948 | .996 | .942 | .905 | .812 | .679 | .865 | .889 | GH | BC |
| QTF | .992 | .998 | .573 | .865 | .945 | .997 | .940 | .909 | .800 | .679 | .850 | .890 | H | BC |

of APMDc values for Cost-cognizant-divide ($CC_{div}$)/Cost-cognizant-break-tie ($CC_{tie}$) hybrid non-peer-based RTP techniques. Table 3.4 shows the APMDc values and Tukey HSD groups for each RTP technique under the two hybrid models. Note that QTF+$CC_{div}$ serves as a baseline for $CC_{div}$ hybrid techniques—it is effectively *Random*—while *Random*+$CC_{tie}$ serves as a baseline for $CC_{tie}$ hybrid techniques—it is literally *Random*.

**Hybrid vs. basic non-peer-based RTP** Both hybrid models improved the average APMDc values across projects on most of the basic non-peer-based techniques. For example, excluding the baselines, the average APMDc values over all basic non-peer-based techniques is 0.838 (Table 3.3), while the same values for $CC_{tie}$ and $CC_{div}$ hybrid techniques are 0.848 and 0.905, respectively. Also, the best basic non-peer-based technique ($ST_{add}$) achieves an APMDc value of 0.895, while the best hybrid non-peer-based technique, $Cov^m_{add}$+$CC_{div}$, achieves an APMDc of 0.961. $Cov^m_{add}$+$CC_{div}$ performs much better than $Cov^m_{add}$: $Cov^m_{add}$ favors ctests with larger code coverage but they also tend to run slower, while $Cov^m_{add}$+$CC_{div}$ considers code coverage per time unit cost (§3.3.3), so $Cov^m_{add}$+$CC_{div}$ makes a better trade-off between the coverage and cost information. In summary, this finding indicates that the hybrid models can substantially boost the basic non-peer-based RTP techniques. This finding was previously reported for traditional regression testing [40] but not for configuration testing.

**Cost-cognizant-divide versus cost-cognizant-break-tie** Table 3.4 shows that $CC_{div}$ hybrid techniques overall perform better than $CC_{tie}$ hybrid techniques. The average APMDc values range from 0.850 to 0.961 for $CC_{div}$, while they range from 0.723 to 0.928 for $CC_{tie}$.

Interestingly, the *additional* RTP techniques with configuration-specific coverage tend to perform better with $\mathrm{CC}_{tie}$ than with $\mathrm{CC}_{div}$, opposite to our overall finding. The reason is that ctests usually do not read as many (changed) configuration parameters as they cover traditional methods or statements; when using the *additional* strategy on configuration-specific coverage, the basic priority scores of ctests quickly become 0 (already prioritized ctests cover all parameters, and yet-to-prioritize ctests cannot cover any more parameters), thus making $\mathrm{CC}_{div}$ effectively become *Random*. For example, on HDFS, $\mathrm{PC}^{D}_{add}$ cannot provide *additional* coverage after prioritizing 2–4 ctests. In contrast, the $\mathrm{CC}_{tie}$ hybrid model can break such ties by ordering the tied tests in the ascending order of their execution time (§3.3.3), thus outperforming $\mathrm{CC}_{div}$ in such cases.

**Total vs. Additional** With the $\mathrm{CC}_{tie}$ model, the *additional* hybrid techniques outperform all the *total* ones on average APMDc values. This finding is consistent with our finding for the basic non-peer-based techniques in §3.5.1. Interestingly, this no longer holds for the $\mathrm{CC}_{div}$ model. Although the very best $\mathrm{CC}_{div}$ hybrid techniques ($\mathrm{Cov}^{m}_{add}$+$\mathrm{CC}_{div}$ and $\mathrm{Cov}^{s}_{add}$+$\mathrm{CC}_{div}$) are *additional*, all other *additional* techniques under-perform their *total* counterparts with the $\mathrm{CC}_{div}$ hybrid model.

The reason is that the priority of ctests can easily become 0 when using the *additional* strategy, making $\mathrm{CC}_{div}$ behave as *Random* (§3.5.2), while the *total* strategy can still effectively prioritize different ctests. Thus, the $\mathrm{CC}_{div}$ hybrid model is more effective for *total* RTP techniques that seldom encounter 0 priority scores. Also, $\mathrm{CC}_{div}$ can be more effective for basic criteria that include more elements and are more diverse, such as traditional code coverage.

### 3.5.3 RQ3: Peer-Based RTP

This RQ evaluates the effectiveness of both basic (§3.3.2) and hybrid (§3.3.3) peer-based RTP techniques for configuration testing. Figure 3.7 shows the distribution of APMDc values for all the evaluated peer-based techniques. Table 3.5 further shows the average APMDc values and the Tukey HSD groups for these techniques.

**Peer-based vs. non-peer-based RTP** According to Table 3.5, 7 of the 12 basic peer-based techniques outperform the *best* non-peer-based technique (i.e., $\mathrm{Cov}^{m}_{add}$+$\mathrm{CC}_{div}$) by average APMDc. Moreover, as seen in Figure 3.7, all APMDc values for all peer-based techniques are well above 0.65, while multiple basic and hybrid non-peer-based techniques have APMDc values well below 0.65 even up to 0.2 (Figure 3.5 and Figure 3.6), indicating the effectiveness and stability of the basic peer-based techniques for configuration testing.

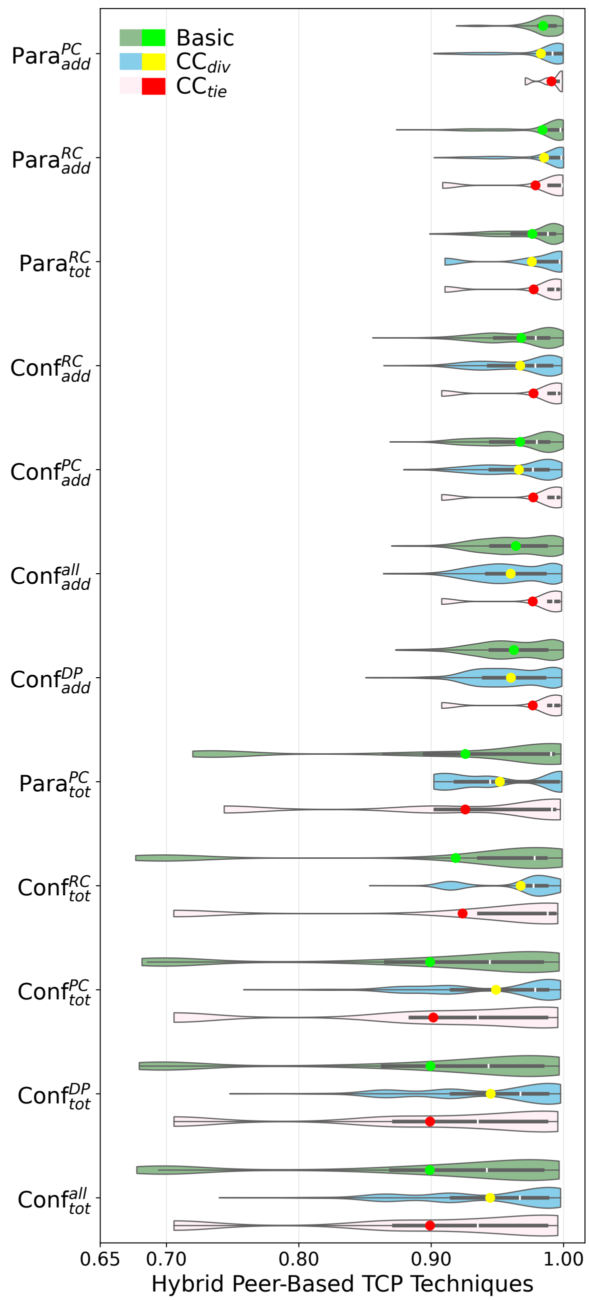


Figure 3.7: **Distribution of APMDc values for peer-based RTP techniques (sorted by average APMDc from Basic).**

Table 3.5: **Average APMDc values and Tukey HSD groups of peer-based RTP.**

| RTP | Average | | | Group | | |
|---|---|---|---|---|---|---|
| | Basic | $CC_{div}$ | $CC_{tie}$ | Basic | $CC_{div}$ | $CC_{tie}$ |
| $Para^{PC}_{add}$ | .985 | .983 | .991 | A | AB | A |
| $Para^{RC}_{add}$ | .984 | .985 | .979 | A | A | A |
| $Para^{RC}_{tot}$ | .976 | .976 | .977 | AB | B | A |
| $Conf^{RC}_{add}$ | .968 | .967 | .977 | B | C | A |
| $Conf^{PC}_{add}$ | .967 | .966 | .977 | B | C | A |
| $Conf^{all}_{add}$ | .964 | .960 | .977 | B | C | A |
| $Conf^{DP}_{add}$ | .962 | .960 | .977 | B | C | A |
| $Para^{PC}_{tot}$ | .926 | .952 | .926 | C | D | B |
| $Conf^{RC}_{tot}$ | .918 | .968 | .924 | C | C | B |
| $Conf^{PC}_{tot}$ | .899 | .949 | .902 | D | D | C |
| $Conf^{all}_{tot}$ | .899 | .945 | .899 | D | D | C |
| $Conf^{DP}_{tot}$ | .899 | .945 | .899 | D | D | C |

$Para^{PC}_{add}$ and $Para^{RC}_{add}$ are statistically significantly better than other basic peer-based techniques, as they are both within the best Tukey HSD group "A". These two techniques are not statistically different, although $Para^{PC}_{add}$ has a slightly higher average APMDc. This finding is surprising as $Para^{PC}_{add}$ requires *no* root-cause information, but still performs as well as $Para^{RC}_{add}$, which requires such information (§3.3.2). The reason is that on some projects (e.g., ZooKeeper), many ctests have similar $Para^{RC}$, so the *additional* strategy suffers the same problem as in §3.5.2. Meanwhile, $Para^{PC}$ values of these ctests are more diverse (and larger than their $Para^{RC}$ values).

Different from the results for the non-peer-based techniques, the hybrid models have only limited effectiveness for the peer-based techniques. The $CC_{div}$ model can only improve the effectiveness for the inferior peer-based techniques. For example, $Conf^{all}_{tot}$, the worst basic technique, is improved from 0.899 into 0.945, while the two best basic techniques ($Para^{PC}_{add}$ and $Para^{RC}_{add}$) have almost no change. The $CC_{tie}$ model can only slightly improve the effectiveness of the superior peer-based techniques. For example, $Para^{PC}_{add}$ changes from 0.985 to 0.991, while the inferior techniques (such as $Conf^{all}_{tot}$) do not change at all. The reason is that *total* techniques usually have fewer ties, making $CC_{div}$ more effective than $CC_{tie}$.

**Configuration vs. parameter granularity** Using both *additional* and *total* strategies, techniques at the parameter granularity have mostly outperformed techniques at the configuration granularity. For example, as seen from Table 3.5, with the *additional* strategy, the basic techniques at the parameter granularity ($Para^{PC}$, $Para^{RC}$) are both in group "A", while

Table 3.6: **Results for the best RTP techniques.**

| RTP | HCommon | HDFS | HBase | ZooKeeper | Alluxio | Average | Group |
|---|---|---|---|---|---|---|---|
| $\text{Para}^{PC}_{add}$+$\text{CC}_{tie}$ | .999 | .988 | .999 | .971 | .998 | .991 | A |
| $\text{Para}^{PC}_{add}$ | .995 | .982 | .990 | .975 | .981 | .985 | A |
| $\text{Cov}^{m}_{add}$+$\text{CC}_{div}$ | 1.00 | .886 | .989 | .946 | .983 | .961 | B |
| $\text{ST}_{add}$ | .990 | .608 | .971 | .963 | .941 | .895 | C |
| QTF | .998 | .865 | .997 | .909 | .679 | .890 | C |
| *Random* | .992 | .577 | .947 | .939 | .823 | .856 | D |

all basic techniques at the configuration granularity ($\text{Conf}^{all}$, $\text{Conf}^{DP}$, $\text{Conf}^{PC}$, $\text{Conf}^{RC}$) are in group "B". Similarly, with the *total* strategy, the basic $\text{Para}^{RC}$and $\text{Para}^{PC}$ are in groups "AB" and "C", respectively, while all basic techniques at configuration granularity are within groups "C" or "D". This result is expected as the parameter granularity captures parameter-level information from other failed peer configuration changes, while the configuration granularity is more coarse-grained (§3.3.2).

**Total vs. Additional** Similar to the results for the non-peer-based techniques, the *additional* strategy generally performs better than the *total* strategy for the basic and $\text{CC}_{tie}$ peer-based techniques. Except that Table 3.5 shows the basic $\text{Para}^{RC}_{tot}$ is a *total* technique at the parameter granularity that performed slightly better than basic *additional* techniques at the configuration granularity, because $\text{Para}^{RC}$ leverages more fine-grained information about peer misconfigured parameters to guide more effective prioritization.

### 3.5.4 Summary

We compare the best techniques from each of the basic/hybrid peer-based/non-peer-based categories, i.e., $\text{ST}_{add}$ (basic non-peer-based), $\text{Cov}^{m}_{add}$+$\text{CC}_{div}$ (hybrid non-peer-based), $\text{Para}^{PC}_{add}$ (basic peer-based), and $\text{Para}^{PC}_{add}$+$\text{CC}_{tie}$ (hybrid peer-based). We also include *Random* and QTF as the baselines. Note that the QTF technique is rather competitive as it outperforms almost all the basic non-peer-based RTP techniques (Table 3.3). Table 3.6 presents the main comparison results. We can observe that all four techniques significantly outperform the *Random* baseline, and three of them significantly outperform the QTF baseline. In summary: (1) $\text{Cov}^{m}_{add}$+$\text{CC}_{div}$ is the best non-peer-based technique and recommended when no peer configuration information is available, (2) $\text{Para}^{PC}_{add}$ and its $\text{CC}_{tie}$ counterpart are the best techniques (i.e., both in group "A") and recommended when peer information is available.

## 3.6 THREATS TO VALIDITY

**External validity** The threats to external validity mainly lie in projects and dataset used in this work. To reduce such threats, we directly use all the real-world projects and configuration changes from the Ctest dataset [189]. However, our evaluation is only based on ctests, which cannot represent all possible types of configuration tests. Future work should consider more diverse datasets and other types of configuration tests.

**Internal validity** The threats to internal validity mainly lie in the potential bugs in our techniques and experimental scripts. To reduce such threats, the authors regularly check the results and code to eliminate potential bugs. Furthermore, we released all our dataset and code to benefit the community.

**Construct validity** The threats to construct validity mainly lie in the metrics used in our study. To reduce such threats, we adapt two widely-used metrics for evaluating RTP techniques (APFD and its cost-aware variant APFDc) and propose new metrics (APMD and its cost-aware variant APMDc) for configuration testing.

## 3.7 RELATED WORK

We have already introduced the background on configuration testing (§3.2) and discussed the related test-case prioritization (RTP) techniques (§3.3), so this section briefly discusses the basics and applications of RTP. RTP techniques were initially proposed to reorder test executions for traditional software systems (e.g., common C and Java applications) to speed up detection of regression faults during software evolution. To date, a large number of code-coverage-based RTP techniques have been proposed for such purpose, including techniques based on traditional *total*/*additional* heuristics [47], adaptive random testing [137], genetic algorithms [139], and constraint solving [138]. More recently, researchers have also looked into RTP techniques that do not require code-coverage information, e.g., techniques based on information retrieval [40] or static program analysis [204]. Interestingly, although more and more RTP techniques have been proposed, the traditional *additional* technique and its cost-cognizant variant (e.g., hybrid with Cost-cognizant-divide) have still remained among the most effective RTP techniques [61].

Besides the traditional application scenarios, RTP has also been applied to various other scenarios, e.g., mutation testing [140], fault localization [141], and automated program repair [142, 143, 144]. Moreover, researchers have applied RTP techniques for testing configurable systems [145, 146]. However, they still target the traditional regression testing problem, i.e., detecting regression faults caused by code changes, while also considering prioritizing the potential configurations that may likely expose regression faults. In contrast, this work makes the first attempt to apply RTP for speeding up misconfiguration detection for configuration testing.

## 3.8 DISCUSSION

To better measure the overall detection time for all the misconfigured parameters within each configuration change, we introduced APMDc (together with APMD) as our main evaluation metric. However, APMDc may not be preferred for practitioners with more interest in how RTP affects the time to detect misconfigurations. Thus, we also relate changes to APMDc with changes to the total test time. APMDc captures time to detect *all* misconfigured parameters in a configuration change. If there is only one misconfigured parameter, then 0.1 increase in APMDc maps to exactly 10% reduction of total time. If there are more misconfigured parameters, 0.1 may map to less or more than 10% time reduction to detect either the first misconfigured parameter or all misconfigured parameters. For our studied projects, 0.1 increase in APMDc maps to from 7.86% (HCommon) to 21.93% (HBase) average time reduction to detect all misconfigured parameters. The reduction can be even larger to find the first misconfigured parameter, e.g., 0.1 increase in APMDc maps to 53.38% (Alluxio) average time reduction.

Our study also points to several directions for future work. Since historical data were reported to be useful in traditional Regression Test Prioritization [12, 17, 38, 40], we could leverage historical configuration change test results from earlier *code* versions to develop history-based RTP techniques for configuration testing. We also consider improving the current configuration-specific RTP techniques and evaluating them on larger datasets. For example, we can fuse deeper context information (e.g., how ctests *use* their parameters acquired from configuration taint analysis) into stack-trace-based RTP techniques, or improve peer-based RTP techniques by combining more data from peer configurations (e.g., test time, failure stack traces).

Furthermore, we plan to understand the impact of software evolution on the performance of our evaluated RTP techniques for configuration testing. Although prior work has shown that the traditional prioritization techniques remain robust over multiple system releases [48], this conclusion may not hold in the context of configuration testing. Configurations and configuration-related code are updated frequently [154, 205], so certain types of test information may be more sensitive to software evolution. For example, data from old peer configuration changes could be less accurate in guiding peer-based RTP techniques on recent system releases.

We only evaluate the performance of RTP techniques on configuration changes. However, sometimes software developers may change both configuration and code in the same commit. In such context, an RTP technique should consider both configuration and code information, and balance the effectiveness in speeding up both misconfiguration and code fault detection. Future work should study how the mixture of configuration and code testing can shift the performance of our evaluated RTP techniques, and understand how to develop competitive RTP techniques in such context [206].

## 3.9 SUMMARY

We have performed the first extensive study of RTP for configuration testing. We have implemented 84 traditional and novel ctest-specific RTP techniques. The experimental results on five popular cloud projects demonstrate that RTP can substantially speed up misconfiguration detection. We have also analyzed the impact of various controllable factors for applying RTP in configuration testing, including coverage criteria, hybrid models, *total*/*additional* strategies, peer-data granularities, and study metrics. In sum, our study reveals various practical guidelines for applying RTP in configuration testing, including: (1) among the basic RTP techniques, QTF is surprisingly competitive and often outperforms sophisticated techniques (based on code coverage or IR) and even some ctest-specific techniques (based on parameter coverage or stack traces), (2) hybrid RTP techniques (which enhance basic techniques with text execution cost information) can boost the performance of most basic techniques, and (3) peer-based RTP techniques (which leverage peer configuration data for better prioritization) can substantially outperform all other studied RTP techniques.

# Chapter 4: Regression Test Prioritization Tool for Python

This chapter presents an RTP tool for Python—PYTEST-RANKING. Section 4.1 describes the lack of readily-usable tool and realistic evaluation of RTP, and provides an overview of this work. Sections 4.2 and 4.3 describe the implementation and usage of PYTEST-RANKING. Section 4.4 describes the experiment setup of our evaluation on PYTEST-RANKING, including the project and build collection, and how we rerun CI builds with PYTEST-RANKING on GitHub Actions. Section 4.5 reports the evaluation results, including analyses of regression and flaky test failures (§4.5.1), variation of test-suite run durations, RTP effectiveness measured by APFDc and HAPFDc, and PYTEST-RANKING overhead (§4.5.2). Section 4.6 provides a concluding summary.

## 4.1 OVERVIEW

As prior chapters have also mentioned, many RTP techniques have been developed, e.g., using test costs [12, 61], past outcomes [14, 38], code coverage [35, 60], information retrieval [39, 43], and machine learning [41, 42]. These RTP techniques were shown effective on CI build datasets of both open-source and proprietary software [17, 27, 28, 36, 37, 46, 47, 48, 62].

Although many RTP techniques and datasets exist, there is no readily usable, open-source tool for running RTP in CI. Several research RTP tools have been developed, but mostly for Java where widely used testing frameworks change slowly, e.g., the test ordering extension for Surefire, the default test runner for the Maven build system, has been pending for 2.5 years [207]. Some prior studies have released scripts for RTP [36, 41, 42, 49, 65] but limited to reproducing evaluation results on research datasets; they provide no interface to integrate with common testing frameworks [135] or CI services. Lack of a readily usable tool makes it difficult for practitioners to adopt RTP research results in their CI practices and for researchers to evaluate their RTP innovations in widely used CI systems. Thusly motivated, we develop an RTP tool for Python, because it is one of the most popular programming languages, and Pytest, its most popular testing framework, is more open to accepting new contributions than Maven Surefire.

Moreover, most prior work [27, 28, 46, 48, 49] has only evaluated RTP techniques by *simulating* RTP orders on historical builds—they reorder tests from the original test-suite runs (TSRs) of the build but compute RTP effectiveness on the reordered test suite with test outcomes and durations *copied* from the original build, *without actually executing* the reordered test suite. Such simulations can miss issues in test execution, e.g., tests in the reordered test suite can have different outcomes or durations [135], reordering can violate existing test-order dependency [108], and test parallelism can impact RTP effectiveness [45]. These are important issues, and their influence on RTP effectiveness should be evaluated.

This work makes the following contributions:

- We develop PYTEST-RANKING, an RTP tool for Pytest, the most prominent testing framework for Python. We release its source on GitHub [208], binary on PyPI [209] for easy installation, linked from `https://zenodo.org/records/15149581`.

- We make PYTEST-RANKING easy to use as a Pytest plugin compatible with several other plugins for test selection, test parallelization, and test ordering. Moreover, PYTEST-RANKING is easily deployable, and we integrate it with 14 projects that use GitHub Actions, currently the most popular CI system.

- We evaluate PYTEST-RANKING by actually rerunning historical builds on GitHub Actions, rather than just simulating the test outcomes and durations. Our test failures analysis finds many flaky tests [110], which simulations would have missed. Out of 1,121 failed TSRs, we find 146 failed flaky tests: 112 flaky tests are order-dependent (OD), with others being concurrency or non-deterministic (ND).

- PYTEST-RANKING is effective, with a low overhead. Our analysis of RTP effectiveness shows that PYTEST-RANKING finds fault 37%–75% sooner than default Pytest order or randomly ordering tests. The runtime overhead of PYTEST-RANKING is 0.03% of the corresponding TSR duration on average across projects; its storage consumption is 5% of the corresponding TSR CI log on average across projects.

Experiments show that PYTEST-RANKING integrates well with the Pytest ecosystem, has a low runtime overhead (0.03% of the test-suite run duration), and finds test failures 37%–75% faster than the Pytest default and *Random* orders. In addition, test failure analysis on 1,121 failed TSRs also finds 146 flaky tests, in which 112 are order-dependent (OD). Our RTP effectiveness analysis also investigates on how test duration variation and flaky test failure treatment impact the RTP technique effectiveness ranking on the executed TSRs. Experiment results demonstrate the practicality and effectiveness of PYTEST-RANKING, and provide implications of adopting RTP in open-source software projects.

## 4.2 IMPLEMENTATION OF PYTEST-RANKING

We implement PYTEST-RANKING as a plugin to the Pytest testing framework. Figure 4.1 illustrates how the components of PYTEST-RANKING interact with the core Pytest. PYTEST-

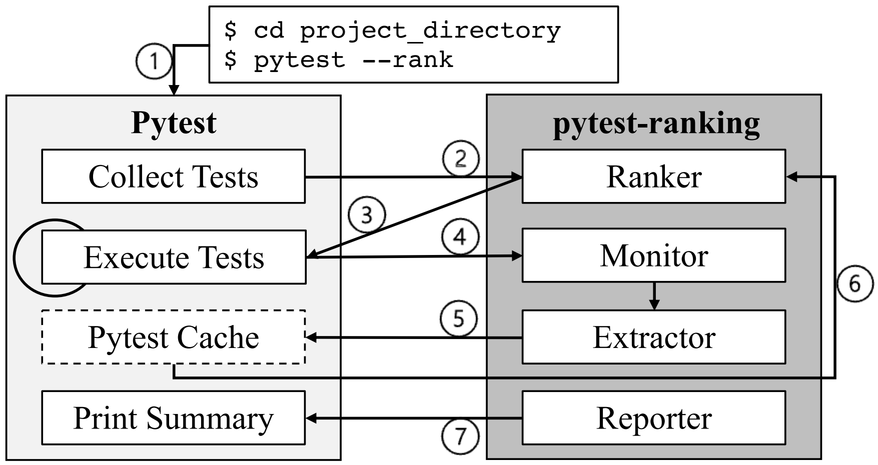


Figure 4.1: **pytest-ranking interaction with Pytest core.**

RANKING has four main components: Ranker, Monitor, Extractor, and Reporter. When PYTEST-RANKING is enabled (①), Pytest provides the selected test suite to the Ranker (②) and receives a prioritized test suite to run (③). Monitor collects the relevant test data (test duration and test outcome) as each test finishes (④). Extractor processes the data when the entire test suite finishes and saves the data of the TSR into the cache directory provided by Pytest (⑤); note that Ranker will use the cached data to prioritize tests in subsequent TSRs (⑥). At the end of the TSR, Pytest reports a summary, accompanied with a summary from the Reporter (⑦).

### 4.2.1 Tool Components

To develop PYTEST-RANKING, we follow the best practices from the Pytest documentation and popular plugins built by Pytest developers [210, 211]. Specifically, we define an entry point for PYTEST-RANKING [212], so when Pytest is run, it can automatically discover and load the installed PYTEST-RANKING [213]. Once PYTEST-RANKING is loaded, it initializes a runner object of the class `RTPRunner` that contains custom implementations of Pytest hooks, which are Pytest APIs that a plugin can re-implement to change Pytest's default behavior [214]. PYTEST-RANKING then registers the runner to Pytest plugin manager [215, 216], so that Pytest will execute `RTPRunner`'s implementations when running these hooks.

We next describe implementations of the main components of PYTEST-RANKING.

**Ranker** Ranker re-implements a Pytest hook [217] that takes a list of test items from Pytest [218] and reorders it. Ranker can apply various RTP techniques for reordering. In Pytest, a *test item*, which we shorten to just *test*, is either a test function/method or a parametrized unit test (PUT) with a concrete parameter value. Each test has a unique ID based on its “collection address” [219], e.g.:

```
dir/test_module.py::test_class::test_function[value]
```

By default, Pytest orders the list hierarchically before passing it to Ranker: tests in the same file follow the definition order, and files follow the directory tree traversal order. Ranker sorts the tests based on their RTP score, as computed by various RTP techniques. Ranker uses a stable sort, i.e., uses the initial order to break ties when multiple tests have the same RTP score.

From the myriad of prior RTP techniques [28], we implement three in Ranker—QTF, *RecentFail*, and *SimChgPath*—because recent research shows that simple RTP techniques often work better than more complex techniques, including machine-learning based techniques, especially when evaluated for realistic scenarios [36, 40, 49]. QTF prioritizes tests with a shorter runtime from their last runs; *RecentFail* prioritizes the recently-failed tests. *SimChgPath* prioritizes tests whose IDs are more textually similar to the paths of Python files changed since the last TSR; its change-awareness complements other techniques [39, 40]. PYTEST-RANKING tracks changed files between TSRs with file content checksums [220].

Ranker uses configurable weights to linearly combine values from all three RTP techniques into a test’s overall RTP score, and then prioritizes tests by their scores. Linear combination implements a hybrid RTP that is more effective than any individual RTP technique [40, 49] and easier to scale with new techniques [67]. To combine different RTP techniques, Ranker normalizes the values to the $[0, 1]$ range for each technique [36, 37], ensuring that a higher value means a higher priority. For tests with no prior RTP data, i.e., likely new tests, Ranker assigns the highest priority.

Ranker allows the user to prioritize tests at different granularity, from individual tests to test files, modules, or directories. Ranker is hierarchy-aware [108, 135], e.g., if a user runs module-level RTP, Ranker identifies a test’s parent module $m$ via its ID, computes the average score from tests in $m$ as the RTP score of $m$, and prioritize on modules; tests in each module follow the Pytest initial order.

**Monitor** Monitor re-implements a hook [221] to collect test execution data. After each test finishes, Pytest calls Monitor with a `TestReport` object [222] of this test as input. A `TestReport` has attributes such as execution duration and outcome. To minimize runtime overhead, Monitor only appends each `TestReport` to an in-memory list in `RTPRunner`; Extractor processes these reports when the TSR finishes.

**Extractor** Extractor saves the RTP data of the executed tests from `TestReports` to the Pytest cache [223]. The Pytest cache directory is in the target project's root [224]; Extractor creates a subdirectory to store mapping data for RTP techniques. For QTF, Extractor saves a key-value mapping from a test to its duration from the current TSR. For *RecentFail*, it saves a mapping from a test to how many times it has passed since its last failure, as computed from the previous cache value and the current outcome. *SimChgPath* data (file checksums) is handled by Ranker as change file set is only checked before a TSR. PYTEST-RANKING overwrites the previous cache with the new one, not saving data for historical TSRs to minimize cache usage.

**Reporter** Reporter adds a session to the Pytest terminal summary to report the configuration and overhead of PYTEST-RANKING [225].

## 4.3 USAGE OF PYTEST-RANKING

PYTEST-RANKING is a Python package and can be installed and managed via `pip`. PYTEST-RANKING is invoked by setting Pytest's command-line options or configuration files [226]. For example, `pytest --rank` runs PYTEST-RANKING with the default configuration. PYTEST-RANKING has these configurable options:

- `--rank-weight` sets RTP technique weights (§4.2). If all weights are 0, PYTEST-RANKING randomly shuffles tests, i.e., *Random*.
- `--rank-level` sets the granularity level on which RTP is run in the test suite hierarchy [135]. Its value can be `put` (each PUT is treated as its own group), `function`, `module`, and `dir`. Test units in the same group follow the default order, while groups are reordered by the average RTP score.
- `--rank-hist-len` sets the largest value that can be recorded for each test for *RecentFail* [65].
- `--rank-seed` sets the random seed for *Random*.

If a CI workflow runs different builds in a fixed location, e.g., a project directory on a specific machine, users can deploy PYTEST-RANKING into CI without any additional setup. If the CI workflow always starts a new virtual machine to run a build, PYTEST-RANKING requires setting up the workflow to pass the Pytest cache data across builds. In our experience with GitHub Actions, the setup adds only 14 lines of YAML to a CI workflow file [227].

PYTEST-RANKING works with test selection [210] and parallelization [228]. It also works with plugins for ordering tests [229, 230], by running ordered tests first in their declared order. Some Pytest options [231], or plugins that randomly order tests [232, 233], can interfere with PYTEST-RANKING as they use the same reordering hook [214].

## 4.4 EXPERIMENTAL SETUP

To evaluate PYTEST-RANKING, we collect and rerun builds from 14 projects that use Pytest and GitHub Actions.

### 4.4.1 Dataset Collection

**Project selection** We start from the 2,500 most downloaded projects in the year 2023 on PyPI [234, 235, 236] and select candidate projects through metadata filtering and manual inspection as follows.

First, we filter out projects that did not list a valid GitHub URL, a required Python version, or a required Pytest version in their PyPI metadata [236]—these steps yield 315 projects. Then, we filter out projects that did not run CI for their recent commits [237], and projects whose `git clone` takes over 1 minute (these are likely too large for our compute budget)—these steps yield 122 projects. To find popular projects with sufficient failures, we keep projects with over 1,000 GitHub stars, and at least one failed TSR on the default branch (§4.3) [233]—these steps yield 58 projects.

We manually inspect the top 40 most downloaded projects out of the 58 projects. We fork each project repository, find its latest CI build with passing TSR, rerun that build on the fork, and inspect the results. We include a project for evaluation if that CI build (1) ran tests on GitHub Actions (which provides uniform API access to test logs and artifacts [238]), (2) passed on `ubuntu-latest` and Python 3 [239], (3) has a TSR duration exceeding 45 seconds. Based on manual inspection, we select 12 projects to evaluate.

We also want to select projects that randomly order tests [232, 233], as they likely have no test-order dependency [108, 240] and could be open to RTP. However, from the 2,500 most-downloaded projects, we found only 1 eligible candidate after the selection steps as above. To find more candidates, we eventually searched the top 50,000 most-downloaded projects, found 91 projects that used related plugins, excluded 72 projects after filtering, and selected 2 projects after inspection.

In total, we select 14 (12 + 2) projects to evaluate.

**Build collection** An RTP tool is presumably deployed to CI at a certain time and run on all builds from that time. Thus, for each project, we collect all builds [241] from 2024-01-01 to 2024-12-01; we order the builds chronologically by their start time [36, 78, 123].

Some builds did not contain a TSR (e.g., builds from CI workflows for package release)—they cannot be used to evaluate RTP. We remove non-TSR builds by checking if the build is from a CI workflow file that runs tests: for each project, we manually find the CI workflow file that runs tests in the latest commit, collect the set of paths this file has historically taken via `git log`, and check if a build's workflow file path matches any of the collected paths. We only keep TSR builds that have a `completed` status, with a `success` or `failure` build conclusion [241]—we removed incomplete TSR builds (e.g., `cancelled`, `timed_out`, or `skipped` builds), because they are often ignored in development and would trigger reruns. We eventually collected 27,224 builds with TSRs, however, we did not evaluate them all to limit computation time (§4.4.2).

### 4.4.2 Experiment Procedure

Different from prior work, we do not only simulate RTP order on the historical builds (§4.1). Instead, we use PYTEST-RANKING to actually execute the RTP-ordered test suite of the historical build, on the code version that the build ran; then we study the test results.

To rerun historical builds on a project, we first fork the project and set up PYTEST-RANKING in its GitHub Actions CI test run file (§4.3). For each build, we get the repository content archive at the commit of the build [242], overwrite the fork with the archive content except for the modified CI file, and push the code to GitHub—GitHub Actions then automatically builds the pushed code with the modified CI file, and we finally download the produced TSR results. Rerunning historical builds can install later versions of the dependencies that are incompatible with the historical code version, and introduce unwanted failures. We eliminate such failure by adopting the package manager UV with its `--exclude-newer` option to each project's CI workflow, so that UV installs dependency versions released before the start date of a given historical build [243].

To mimic how these projects would have run PYTEST-RANKING, we (1) make minimal change to their original CI file; (2) use the same GitHub Actions CI service; and (3) run builds with one environment they used, e.g., `ubuntu-latest` and Python 3. We also mimic build overlap: if build $i+1$ started after build $i$ ended, we rerun $i+1$ after $i$ finishes rerunning (so $i+1$ uses the cache updated by $i$); otherwise, we run both builds in parallel as $i+1$ would have used the cache prior to $i$.

For each build, we run 6 orders: 4 RTP orders (QTF, *RecentFail*, *SimChgPath*, and

Table 4.1: **Evaluation dataset. TSR means test-suite run.**

| Project | #Commit | #TSR | #Failed TSR | Average TSR size #Test | Average TSR size Duration (s) |
|---|---|---|---|---|---|
| aeon | 49 | 294 | 121 | 32,295 | 2,202.2 |
| ansible-lint | 50 | 300 | 128 | 815 | 350.7 |
| apscheduler | 102 | 612 | 34 | 740 | 52.3 |
| dask | 51 | 306 | 65 | 12,613 | 802.1 |
| dvc | 49 | 294 | 86 | 2,693 | 306.1 |
| ipython | 42 | 252 | 171 | 1,422 | 91.2 |
| librosa | 32 | 192 | 30 | 13,752 | 450.9 |
| molecule | 50 | 300 | 54 | 458 | 182.3 |
| networkx | 50 | 300 | 84 | 6,073 | 125.7 |
| pytest-django | 51 | 306 | 56 | 219 | 59.1 |
| pytest-xdist | 34 | 204 | 105 | 208 | 105.4 |
| pytorch-lightning | 67 | 402 | 30 | 3,336 | 609.6 |
| trimesh | 49 | 294 | 78 | 599 | 325.4 |
| ultralytics | 42 | 252 | 79 | 73 | 524.7 |
| **Total or Average** | **718** | **4,308** | **1,121** | **5,378** | **442.0** |

*Hybrid* that combines the prior three with equal weights) and 2 baselines (Pytest *Default* and *Random*). Each order runs as a separate CI workflow to avoid interference of Pytest cache. We run orders at the function granularity (§4.3): test functions are prioritized, but parameter values of the same PUT function follow the default order as they often have order dependency [244], so reordering them can produce lots of non-regression failures.

Because we cannot rerun all 27,224 builds 6 times [89], we select all failed builds and the first non-overlapping successful build before each failed build. The successful builds are used to create the right data cache for running RTP on the failed builds. For each project, we rerun its selected builds until reaching the last build or the first build that does not overlap the $50^{th}$ build; we then rerun more builds until reaching at least 30 failed TSRs per project. We then collected the TSR results as JSON-formatted test reports [245].

We run each order once per build to (1) analyze test failures, and (2) compute effectiveness for all orders except *Random*. To compute effectiveness for *Random* [246] on failed builds, we rerun *Random* 10 additional times per build using rerun IDs as seeds.

Table 4.2: **Number of failed tests and test failures. Reg. means regression.**

| Project | Failed tests | | Test failures | |
|---|---|---|---|---|
| | Flaky | Reg. | Flaky | Reg. |
| aeon | 9 | 341 | 21 | 3,307 |
| ansible-lint | 11 | 10 | 310 | 140 |
| apscheduler | 3 | 322 | 22 | 1,948 |
| dask | 9 | 118 | 32 | 943 |
| dvc | 9 | 5 | 76 | 42 |
| ipython | 83 | 2 | 2,257 | 42 |
| librosa | 0 | 10 | 0 | 90 |
| molecule | 0 | 7 | 0 | 66 |
| networkx | 6 | 123 | 103 | 750 |
| pytest-django | 2 | 21 | 25 | 138 |
| pytest-xdist | 1 | 12 | 3 | 150 |
| pytorch-lightning | 0 | 21 | 0 | 126 |
| trimesh | 0 | 26 | 0 | 204 |
| ultralytics | 13 | 17 | 240 | 357 |
| **Total** | **146** | **1,035** | **3,089** | **8,303** |

## 4.5 EVALUATION

Table 4.1 shows the statistics of our evaluation dataset. Following §4.4.2, we successfully reran 718 commits/builds from 14 projects and obtained 4,308 TSRs (6 TSRs per commit). Some selected commits failed to rerun, because the original commit introduces errors that cause the TSR failed early, e.g., commits from a stale branch use deprecated dependencies and cause all tests to error. As a result, some projects do not have 50 commits reran. Of 4,308 TSRs, 1,121 had at least one test failure. §4.5.1 presents our analysis of these failures. §4.5.2 discusses the effectiveness and overhead of PYTEST-RANKING.

### 4.5.1 Analysis of Test Failures

**Analysis of flaky test failures** Flaky tests can nondeterministically pass or fail on the same code version [65, 107, 110, 236], therefore they do not always indicate regression bugs in CI. While research often uses "sanitized" datasets that carefully remove flaky tests, e.g., [247], runs on real CI do have flaky failures as also reported by some prior work [40, 49, 65]. Our inspection finds three types of flaky tests: OD (order-dependent) [248], Concurrency (non-desirable worker interactions), and ND (non-deterministic outcome, e.g., depending on random seeds [249]).

We semi-automatically examined all test failures to classify their nature. Tests that failed under some orders but not others for the same commit were potentially flaky. We reran each such test under the same order multiple times in an attempt to reproduce the failure. If the failure was deterministically reproduced, it was likely OD, and we performed binary search [250] on all tests earlier in the order to minimize the test sequence that revealed OD flakiness; if binary search failed, we fell back to linear search. If the test was not OD, we reran it 1,000 times to check if it was ND. All remaining flaky tests only failed with `pytest-xdist`, so we executed each such test concurrently with another test to identify the Concurrency.

For tests that consistently failed on all orders for the same commit, we inspected the logs to determine whether the failures are due to regression changes. Table 4.2 summarizes our results: we confirmed 146 flaky tests and 1,035 regression tests. 10 projects contain at least one flaky test, 6 projects contain at least one OD flaky test. Column "Failed Tests" shows the count of distinct tests that are flaky or regressions (i.e., failed at least once due to regression changes), and "Test Failures" shows the total number of test failures across all builds.

In Table 4.2, while flaky tests are much fewer than regression tests, the failed flaky tests on average had failed twice more often than the failed regression tests. The reason is that flaky tests tend to have a longer lifecycle, while regressions are often detected by deterministic test outcomes and addressed quickly [65]. Of all the 1,035 regression tests, 774 failed for only one commit. No regression test failed in more than 9 commits, while only 9 of the 146 flaky tests have ever been fixed by the time of the most recent commit we examined.

Of the 146 flaky tests, 112 are OD. Like prior studies on OD Python flaky tests [236, 251], we categorize them into Victims (i.e., fail if run after other "polluters" [250]) and Brittles (i.e., pass only if run after other "state-setters" [250]). Of the 112 OD tests, 107 (95%) are Victims, and only 5 (5%) are Brittles. The higher prevalence of Victims compared to Brittles matches prior findings in both Python and Java projects [236, 250]. Of all 60 polluters we confirmed for the Victim tests, 14 (23%) require a sequence of more than one test to be run before the victim fails. This percentage is greater than a prior study on Java (2.4%) [250].

Of the 34 non-OD flaky tests, 6 are Concurrency and fail under `pytest-xdist` [228], a plugin that enables `pytest` to spawn multiple worker processes to run tests. This parallel execution can lead to failures when, e.g., concurrently running tests attempt to use the same network port. The other 28 tests are ND, exhibiting flakiness due to the inherently non-deterministic behavior of certain APIs, e.g., random number generators [249].

We also inspect new test failures from the additional *Random* reruns (§4.4.2). We treat these reruns separately because we use them primarily to obtain more reliable results for the average effectiveness of RTP techniques. The *Random* reruns further expose 34 flaky tests from 6 projects. Of the 34 new flaky tests, and 27 are OD, 7 are ND. 3 of the 34 tests have

Table 4.3: **Number of TSRs with regression/flaky test failures.**

| Project | Reg. | Reg. only | Flaky | Flaky only | OD flaky | OD flaky only |
|---|---|---|---|---|---|---|
| aeon | 120 | 119 | 2 | 1 | 0 | 0 |
| ansible-lint | 24 | 16 | 112 | 104 | 112 | 104 |
| apscheduler | 30 | 30 | 4 | 4 | 0 | 0 |
| dask | 48 | 46 | 19 | 17 | 0 | 0 |
| dvc | 30 | 26 | 60 | 56 | 57 | 54 |
| ipython | 42 | 14 | 157 | 129 | 157 | 129 |
| librosa | 30 | 30 | 0 | 0 | 0 | 0 |
| molecule | 54 | 54 | 0 | 0 | 0 | 0 |
| networkx | 36 | 32 | 52 | 48 | 42 | 38 |
| pytest-django | 36 | 32 | 24 | 20 | 24 | 20 |
| pytest-xdist | 102 | 102 | 3 | 3 | 0 | 0 |
| pytorch-lightning | 30 | 30 | 0 | 0 | 0 | 0 |
| trimesh | 78 | 78 | 0 | 0 | 0 | 0 |
| ultralytics | 60 | 51 | 28 | 19 | 28 | 19 |
| **Total** | **720** | **660** | **461** | **401** | **420** | **364** |

been fixed in subsequent commits.

We find no OD flaky tests in projects that already run their tests in random order in CI, i.e., aeon and pytorch-lightning, in reruns of all orders including *Random*. Our analysis of test failures under RTP reordering suggests that flaky tests, especially OD flaky tests are not uncommon in CI, and should be properly identified, annotated, or fixed. Running tests in random order or RTP orders can effectively expose the OD flaky tests.

**Test failure occurrence** We first analyze the number of TSRs with regression or flaky test failures. Table 4.3 shows the number of TSRs with regression, only regression, flaky, only flaky, OD flaky, and only OD flaky test failures, respectively. Overall, majority of the failed TSRs have only regression test failures. Out of 1,121 failed TSRs, 720 (64%) have at least one regression test failure, and 660 (59%) have at least one regression test failure with no flaky test failure. On the other hand, the number of TSRs with flaky failures is nontrivial: 461 (41%) of the failed TSRs have at least one flaky test failure, within which 91% have OD flaky failures; 401 (36%) have only flaky failure.

Overall, 91% (660/ 720) TSRs that have regression failures have no flaky failures. For TSRs that have only flaky failures, we find that 58% of them are from two projects, ipython and ansible-lint; 47% them are from TSRs of *Random* order.

We now analyze the number of regression and flaky test failures in the TSRs. Table 4.4 shows the distribution of the number of regression test failures in TSRs that failed on

Table 4.4: **Distribution of regression test failure count in TSRs with regressions, represented in 25$^{th}$ (Q1), 50$^{th}$ (Q2), and 75$^{th}$ (Q3) percentile values.**

| **Project** | **Q1** | **Q2** | **Q3** |
|---|---|---|---|
| aeon | 4 | 7 | 25 |
| ansible-lint | 3 | 5 | 5 |
| apscheduler | 1 | 1 | 3 |
| dask | 3 | 16 | 37 |
| dvc | 1 | 1 | 2 |
| ipython | 1 | 1 | 1 |
| librosa | 2 | 2 | 2 |
| molecule | 1 | 1 | 1 |
| networkx | 2 | 5 | 8 |
| pytest-django | 2 | 3 | 4 |
| pytest-xdist | 1 | 1 | 1 |
| pytorch-lightning | 1 | 3 | 6 |
| trimesh | 1 | 1 | 2 |
| ultralytics | 1 | 1 | 15 |
| **Average** | **2** | **3** | **8** |

regressions. Comparing Table 4.4 to Table 4.1, we can see that the number of regression test failures in majority of failed TSRs is small, relative to the average number of tests executed per TSR. Meanwhile, Table 4.5 shows the distribution of the number of flaky test failures in TSRs that failed on regression(s), as well as in all failed TSRs. We can see that majority (75% as indicated by Q3) of the TSRs that failed on regression(s) have no flaky test failure, except in projects ipython and ansible-lint. When considering all failed TSRs, the number of flaky test failures slightly increases in some projects, however, majority of the failed TSRs in 7 and 3 projects have zero or one flaky test failures, respectively.

**Implication** Running RTP orders in CI can expose flaky test failures, especially those due to test-order dependency, which could distract developers from immediately looking into the regression test failures. However, the amount of OD flaky tests, can be manageable: 8 out of 14 the evaluated projects have no OD flaky test failures, and 75% of the failed TSRs in 10 projects have at most one flaky test failure. Since PYTEST-RANKING can respect declared test-order dependencies (§4.3), before adding PYTEST-RANKING in CI, developers should consider identifying dormant test-order dependencies via specific tools [251] or by running random order [240]. To mitigate the impact of OD flakt tests to RTP in CI, developers can further annotate [229, 230] or fix [250, 252] these order dependencies.

Table 4.5: **Distribution of flaky test failure count in failed TSRs. Columns "TSRs w/ Reg." show the distribution in failed TSRs that have regressions; columns "TSRs" show the distribution in all failed TSRs.**

| Project | TSRs w/ Reg. | | | TSRs | | |
|---|---|---|---|---|---|---|
| | Q1 | Q2 | Q3 | Q1 | Q2 | Q3 |
| aeon | 0 | 0 | 0 | 0 | 0 | 0 |
| ansible-lint | 0 | 0 | 2 | 1 | 2 | 3 |
| apscheduler | 0 | 0 | 0 | 0 | 0 | 0 |
| dask | 0 | 0 | 0 | 0 | 0 | 1 |
| dvc | 0 | 0 | 0 | 0 | 1 | 1 |
| ipython | 0 | 5 | 19 | 7 | 15 | 20 |
| librosa | 0 | 0 | 0 | 0 | 0 | 0 |
| molecule | 0 | 0 | 0 | 0 | 0 | 0 |
| networkx | 0 | 0 | 0 | 0 | 1 | 3 |
| pytest-django | 0 | 0 | 0 | 0 | 0 | 1 |
| pytest-xdist | 0 | 0 | 0 | 0 | 0 | 0 |
| pytorch-lightning | 0 | 0 | 0 | 0 | 0 | 0 |
| trimesh | 0 | 0 | 0 | 0 | 0 | 0 |
| ultralytics | 0 | 0 | 0 | 0 | 0 | 2 |
| **Average** | **0** | **0** | **2** | **1** | **1** | **2** |

### 4.5.2 RTP Effectiveness

**pytest-ranking effectiveness** Among the two common RTP evaluation metrics APFDc and APFD [14, 28, 120, 121], we use APFDc as it considers test runtime not just test count. APFDc is normalized to $[0, 1]$, so a 0.1 increase reduces the time to detect all faults by 10% of the TSR time. We compute APFDc with one-to-one and many-to-one failure-to-fault mappings [68]; they produce similar results, so we present only one-to-one.

Table 4.6 shows the mean APFDc across TSRs, considering only regressions from §4.5.1 as faults to be detected. Table 4.6 lists the name prefix of each technique: all RTP techniques (QTF, *RecentFail*, *SimChgPath*, *Hybrid*) outperform the baselines (*Default*, *Random*), being 37%–75% better than *Default*, and 29%–65% better than *Random*, on average across projects (e.g., the improvement for *Hybrid* over *Default* is $\frac{0.827-0.473}{0.473} \approx 75\%$) *Hybrid* that linearly combines the other RTP techniques performs the best, with a mean APFDc value of 0.827, 75% better than *Default* (0.473) and 65% better than *Random* (0.500). Our results show that running test suites reordered by PYTEST-RANKING can detect regressions earlier than running the default- or randomly-ordered test suites.

Table 4.6: **Average APFDc on regression test failures.**

| **Project** | ***Def.*** | ***Ran.*** | **QTF** | ***Rec.*** | ***Sim.*** | ***Hyb.*** |
|---|---|---|---|---|---|---|
| aeon | .477 | .498 | .725 | .723 | .475 | .696 |
| ansible-lint | .866 | .618 | .836 | .773 | .866 | .818 |
| apscheduler | .190 | .447 | .765 | .239 | .520 | .751 |
| dask | .437 | .525 | .795 | .481 | .409 | .668 |
| dvc | .536 | .528 | .697 | .704 | .532 | .754 |
| ipython | .582 | .437 | .989 | .868 | 1.00 | 1.00 |
| librosa | .255 | .477 | .767 | .627 | .455 | .825 |
| molecule | .675 | .480 | .710 | .760 | .760 | .880 |
| networkx | .614 | .494 | .971 | .814 | .932 | .990 |
| pytest-django | .566 | .517 | .604 | .590 | .729 | .683 |
| pytest-xdist | .433 | .491 | .550 | .882 | .467 | .917 |
| pytorch-lightning | .440 | .496 | .907 | .503 | .785 | .881 |
| trimesh | .291 | .485 | .857 | .572 | .538 | .914 |
| ultralytics | .268 | .500 | .750 | .528 | .824 | .801 |
| **Average** | **.473** | **.500** | **.780** | **.647** | **.664** | **.827** |

Table 4.7: **Average HAPFDc on regression test failures.**

| **Project** | ***Def.*** | ***Ran.*** | **QTF** | ***Rec.*** | ***Sim.*** | ***Hyb.*** |
|---|---|---|---|---|---|---|
| aeon | .581 | .520 | .741 | .786 | .577 | .717 |
| ansible-lint | .883 | .641 | .868 | .811 | .887 | .853 |
| apscheduler | .262 | .487 | .770 | .271 | .549 | .756 |
| dask | .494 | .555 | .804 | .526 | .469 | .686 |
| dvc | .585 | .588 | .777 | .738 | .577 | .829 |
| ipython | .608 | .469 | .990 | .880 | 1.00 | 1.00 |
| librosa | .278 | .498 | .780 | .647 | .473 | .828 |
| molecule | .700 | .526 | .729 | .786 | .779 | .881 |
| networkx | .654 | .535 | .972 | .832 | .938 | .990 |
| pytest-django | .585 | .549 | .629 | .612 | .748 | .708 |
| pytest-xdist | .452 | .511 | .568 | .884 | .486 | .920 |
| pytorch-lightning | .475 | .522 | .911 | .543 | .800 | .888 |
| trimesh | .310 | .496 | .860 | .586 | .550 | .916 |
| ultralytics | .297 | .540 | .751 | .542 | .830 | .806 |
| **Average** | **.512** | **.531** | **.796** | **.675** | **.690** | **.841** |

**Impact of test duration variation on RTP effectiveness** Both individual tests and test suite can have different durations in different orders [135]. We next study how test durations vary at both the individual test level and the TSR level under different orders for

Table 4.8: **Average APFDc/HAPFDc under different flaky test failure treatments.**

| Treatment | ***Def.*** | ***Ran.*** | **QTF** | ***Rec.*** | ***Sim.*** | ***Hyb.*** |
|---|---|---|---|---|---|---|
| *Average APFDc* | | | | | | |
| "As regression" | .473 | .486 | .782 | .646 | .630 | .824 |
| "Excluded" | .473 | .499 | .780 | .647 | .664 | .827 |
| "As passing" | .473 | .500 | .780 | .647 | .664 | .827 |
| *Average HAPFDc* | | | | | | |
| "As regression" | .512 | .519 | .797 | .673 | .658 | .839 |
| "Excluded" | .512 | .532 | .796 | .675 | .690 | .841 |
| "As passing" | .512 | .531 | .796 | .675 | .690 | .841 |

the same commit. We will also assess PYTEST-RANKING's effectiveness when accounting for duration variation, and how the variation can impact effectiveness results.

For each individual test, we compute the magnitude of Relative Percentage Difference (RPD) between the durations of its *Default* execution and non-*Default* executions of the same commit. We observe that individual test duration can change substantially across executions of different test suite orders for the same commit [135]—the RPD magnitude in 75% of the tests is at most 4%–38% (average 18%) across projects, and the average RPD magnitude across all tests in a commit is 6%–160% (average 45%) across projects. We also compute the RPD magnitude between the durations of each commit's *Default* TSR and its non-*Default* TSRs. The TSR duration difference is 1%–10% (average 4%) across projects. We can see that TSR duration has much less variation than individual test duration across executions of different orders, because duration changes from different tests are canceling each other as some tests run faster while other tests run slower from one order to another.

Given that test runtime changes under different orders, APFDc may not fairly compare different orders of the same test suite because a test suite will have different TSR durations [135]. Thus, we employ Hierarchy-aware APFDc (HAPFDc) to further assess PYTEST-RANKING's effectiveness. Given the set of TSRs from different orders of the same test suite, HAPFDc extends the shorter-running TSRs as if their durations were the same as that of the longest-running TSR. HAPFDc preserves the other properties of APFDc. Table 4.7 shows the mean HAPFDc values. We can still observe that all RTP techniques outperform the baselines, being 32%–64% better than *Default*, and 27%–58% better than *Random*, on average across projects. *Hybrid* still performs the best in HAPFDc as it does in APFDc. According to our evaluation results, HAPFDc ranks all evaluated RTP techniques exactly the same as APFDc.

**Impact of flaky test failure treatment on RTP effectiveness** Prior RTP studies that simulate RTP orders rely on using test outcomes and durations copied from the original historical builds to compute RTP effectiveness metrics like APFDc on the reordered test suites. Such setup assumes each test to always have the same outcome in different orders as in the original build for the same commit. When computing APFDc, these studies either treat all test failures from the original builds as regression test failures [28, 36, 37, 41], or exclude flaky test failures after manually inspecting the original builds [40, 65]. The main prior finding is that history-based RTP techniques, e.g., *RecentFail*, are ranked higher by APFDc when flaky test failures are treated as regression test failures than when they are not.

Despite different orders having the same regression test failures, however, different orders often have different flaky test failures when they are actually executed (§4.5.1). Now, we want to know whether applying different flaky test failure treatments from prior studies on our executed TSRs can influence our RTP effectiveness results and arrive at the same main finding as prior work.

Table 4.8 shows the average APFDc and HAPFDc values across projects for each evaluated order under different treatments. "As regression" means all flaky test failures are considered as regression test failures when computing APFDc [36]. "Excluded" means all failed flaky tests are omitted as if they were never executed (§2). "As passing" means all flaky test failures are considered as passing tests—their durations are added to the TSR duration, but their outcomes do not detect any fault (i.e., same as Table 4.6).

Table 4.8 shows that history-based RTP technique *RecentFail* have a higher APFDc value than change-aware RTP technique *SimChgPath* when flaky tests are treated as regression test failures (i.e., "As regression") than when they are omitted from the APFDc calculation (i.e., "Excluded"). Meanwhile, the relative ranking of other RTP techniques and the baselines is the same between treatment "As regression" and "Excluded". This observation is the same as prior finding obtained from simulated TSRs.

When comparing treatment "Excluded" and "As regression" in Table 4.8, both the APFDc values and the ranking of RTP techniques are the same. In other words, including or excluding flaky test durations for APFDc calculation do not significantly affect RTP effectiveness results, because durations of failed flaky tests are trivial compared to the entire TSR. The same observations above also hold for HAPFDc.

**Comparison between simulating and executing reorder test suites** On our collected TSRs, we analyze the difference in RTP effectiveness between simulating reordered test suites and executing reordered test suites. We use the test durations and outcomes from the *Default* TSR when we compute the APFDc values for the non-*Default* TSRs on the same

Table 4.9: **Average APFDc on regression test failures calculated by simulating non-*Default* TSRs with test durations and outcomes of the *Default* TSRs.**

| **Project** | ***Def.*** | ***Ran.*** | **QTF** | ***Rec.*** | ***Sim.*** | ***Hyb.*** |
|---|---|---|---|---|---|---|
| aeon | .477 | .525 | .774 | .742 | .475 | .721 |
| ansible-lint | .866 | .617 | .843 | .781 | .866 | .822 |
| apscheduler | .190 | .472 | .778 | .239 | .528 | .764 |
| dask | .437 | .533 | .789 | .478 | .408 | .655 |
| dvc | .536 | .493 | .658 | .693 | .534 | .735 |
| ipython | .582 | .454 | .991 | .868 | 1.00 | 1.00 |
| librosa | .255 | .541 | .820 | .637 | .470 | .865 |
| molecule | .675 | .501 | .709 | .759 | .763 | .881 |
| networkx | .614 | .492 | .975 | .812 | .932 | .990 |
| pytest-django | .566 | .519 | .599 | .589 | .735 | .683 |
| pytest-xdist | .433 | .491 | .551 | .882 | .466 | .917 |
| pytorch-lightning | .440 | .499 | .912 | .503 | .779 | .879 |
| trimesh | .291 | .485 | .859 | .572 | .539 | .914 |
| ultralytics | .268 | .521 | .754 | .530 | .829 | .803 |
| **Average** | **0.473** | **0.510** | **0.787** | **0.649** | **0.666** | **0.831** |

commit. Table 4.9 shows the average APFDc values calculated through simulation. Overall, we observe that average APFDc values do change in different projects and for different RTP techniques. The overall effectiveness ranking of all the evaluated RTP techniques is the same as executing reordered test suite data (shown in Table 4.6).

**pytest-ranking overhead** At last, we evaluate the overhead of PYTEST-RANKING on its runtime and Pytest cache usage, as presented in Table 4.10. We collect PYTEST-RANKING's runtime from Reporter's terminal summary (§4.2), which includes the time for computing RTP data and reordering. The time is 0.1 sec, or 0.03% of the TSR duration on average across projects. We collect the Pytest cache sizes after the RTP-ordered TSRs finish, and compare the uncompressed sizes between the TSR cache and the CI log. The cache size is 6 to 538 KB, or 1%–9% (average 5%) of the corresponding CI log sizes.

**Implication** To practitioners, our results motivate the adoption of PYTEST-RANKING in CI for effective and efficient default detection. Specifically, the increase in APFDc and HAPFDc are larger than the change in TSR duration from *Default* to RTP orders, indicating that Pytest test suites can benefit from RTP for faster fault detection. The small runtime overhead and cache size also show that PYTEST-RANKING can be deployed efficiently.

To researchers, our results show that order simulations produce a similar overall RTP

Table 4.10: **Average pytest-ranking time and cache overhead.**

| Project | Time (s) | % TSR time | Cache (KB) | % CI log size |
|---|---|---|---|---|
| aeon | 0.215 | 0.01% | 538 | 1.79% |
| ansible-lint | 0.015 | 0.00% | 71 | 9.07% |
| apscheduler | 0.016 | 0.03% | 15 | 2.61% |
| dask | 0.071 | 0.01% | 216 | 1.33% |
| dvc | 0.022 | 0.01% | 81 | 3.71% |
| ipython | 0.076 | 0.08% | 37 | 3.72% |
| librosa | 0.421 | 0.09% | 138 | 3.10% |
| molecule | 0.072 | 0.04% | 35 | 7.53% |
| networkx | 0.176 | 0.14% | 155 | 8.99% |
| pytest-django | 0.008 | 0.01% | 7 | 4.56% |
| pytest-xdist | 0.007 | 0.01% | 7 | 3.99% |
| pytorch-lightning | 0.197 | 0.03% | 107 | 4.73% |
| trimesh | 0.023 | 0.01% | 22 | 6.94% |
| ultralytics | 0.016 | 0.00% | 6 | 5.26% |
| **Average** | **0.095** | **0.03%** | **103** | **4.81%** |

effectiveness ranking as rerunning the reordered test suites. Specifically, copying test durations could be viable because the actual TSR duration varies much less than each individual test across different orders. Actual APFDc and HAPFDc values may differ between simulations and reruns, but the overall RTP technique ranking remains unchanged. On the other hand, copying test outcomes, especially flaky test failures, likely affects the ranking of history-based RTP techniques, but may not affect the other techniques. Overall, simulations can still be considered for RTP evaluation if reruns become costly. Future work is needed to establish if simulations and reruns lead to the same conclusions for more RTP techniques, projects, and CI settings.

## 4.6 SUMMARY

We presented our PYTEST-RANKING tool for RTP of Pytest test suites. The evaluation on 4,308 GitHub Actions builds of 14 Python projects shows that our tool integrates well with the Pytest ecosystem, has a low runtime overhead, and finds test failures faster than the baseline orders. Our evaluation also provides implication on flaky tests when running test suites in different orders. We hope that these promising results will enable more research and eventually lead to more adoption of RTP in practice.

# Chapter 5: Conclusions and Future Work

This chapter presents the conclusions and future work of this dissertation. Section 5.1 provides summaries and conclusions of the main work. Section 5.2 lists future work that could be built from this dissertation. Section 5.3 closes this dissertation with a remark.

## 5.1 CONCLUSIONS

RTP has been researched for nearly three decades, prior studies have shown promising results of RTP in speeding up regression fault detection with the hope of reducing debugging feedback time to developers. However, much less research outcomes of RTP have been put into practice. To facilitate adoption of RTP in modern software systems, this dissertation studies the effectiveness of prior RTP techniques, and proposes novel RTP techniques, in the latest and most practical testing scenarios. This dissertation also presents an effective and lightweight tool for developers to apply RTP conveniently in the Python ecosystem.

**Regression Test Prioritization on Long-Running Test Suites** First, this dissertation presents the RTP dataset of long-running test suites that reflects up-to-date CI practices—*LRTS*, and revisits key findings from prior RTP studies on *LRTS*. Prior RTP datasets mostly fall into at least one of these limitations: curated from severely outdated CI builds (e.g., from over 10 years ago), having short-running test suites that last for a few minutes on average, collected from inaccessible proprietary projects, curated from synthetic failed builds with seeded faults. It was uncertain whether RTP effectiveness results from these datasets remain true in long-running test suites with modern CI practices.

*LRTS* is large-scale, curated from popular open-source projects with rich CI history and real CI failures. More importantly, *LRTS* assesses RTP techniques on long-running test suites (test-suite run duration is 6.5 hours on average) where RTP can be the most beneficial. We present a detailed analysis of *LRTS*, including analyses of its CI builds and test failures, and its comparison with prior RTP datasets with short-running test suites.

On *LRTS*, we evaluated 59 RTP techniques from five leading technique categories. All of these techniques are lightweight and can be easily applied in practice as they only use readily-accessible CI features. We revisit the key prior findings of RTP on long-running test suites, and present new findings on the impact of realistic CI issues on RTP effectiveness, e.g., confounding test failures and tests having no prior failure history. RTP techniques are likely to face these issues when being deployed into CI pipeline. We find that simply prioritizing faster tests that have failed recently, outperforming all sophisticated techniques. Among standalone RTP heuristics, running faster tests first is the most effective and the least impacted by confounding test failures.

**Conclusion:** RTP is effective in speeding up fault detection on long-running test suites, even under the impact of realistic CI issues such as failures from flaky tests. The simplest techniques, such as running historically faster tests first, are (1) lightweight to implement and maintain, and (2) deliver stronger and more robust performance than the more sophisticated machine learning or information-retrieval-based techniques. This dissertation shows encouraging results in applying RTP on modern software projects with long-running test suites and up-to-date CI practices.

**Regression Test Prioritization for Configuration Testing** Second, this dissertation applies RTP in an emerging yet critical testing context—configuration testing. Running a configuration test suite against configuration change can be time-consuming, while configuration testing is run frequently given the high configuration change velocity in modern, highly-configurable systems.

We apply traditional RTP techniques that are based on code coverage, test duration, and information retrieval. We also develop novel, configuration-specific RTP techniques that are based on configuration parameter coverage, configuration API traces, and peer configuration change data. We further analyze the impact of various controllable factors for applying RTP in configuration testing, including coverage criteria, hybrid models, total/additional RTP strategies, peer-data granularities, and effectiveness metrics.

Our evaluation demonstrates that RTP can substantially speed up misconfiguration detection, with the following major results. Among the applied traditional RTP techniques, simply running faster configuration tests first can outperform the sophisticated techniques and even some configuration-specific techniques by up to 22%. Applying test execution time data into other standalone RTP techniques can further boost their effectiveness by up to 27%. Our novel RTP techniques that leverage peer configuration change data substantially outperform others.

**Conclusion:** RTP is effective in speeding up misconfiguration detection in configuration testing, a testing scenario in modern highly-configurable systems. In practice, developers could apply the simplest RTP technique of running faster tests first for configuration test prioritization, which delivers rival performance compared to techniques based on code coverage. To take a step further, developers can also apply standalone configuration-specific techniques or combine test duration time with configuration-specific techniques, which can be substantially more effective. This dissertation shows that it is also promising to apply

RTP in emerging contexts outside traditional regression testing.

**Regression Test Prioritization Tool for Python** Third, this dissertation presents the first readily-usable, open-source RTP tool for Python and Pytest—PYTEST-RANKING. PYTEST-RANKING is well integrated with the Python ecosystem and its most popular testing framework Pytest. It is managed as a Python package and can be deployed to local CI workflow off-the-shelf, or with a minimum change to CI configuration files if using a remote CI service such as GitHub Actions.

PYTEST-RANKING provides convenient and practical features for developers, allowing developers to find the suitable PYTEST-RANKING configuration for their projects. For example, in PYTEST-RANKING, developers can flexibly combine different lightweight RTP techniques that have been shown as some of the most effective ones in testing scenarios evaluated in this dissertation and some prior studies. Developers can also configure PYTEST-RANKING to run RTP at different test hierarchy levels, e.g., test input, test method, or test file level. PYTEST-RANKING is compatible with popular Pytest tools for test selection, test parallelization, and test-order dependency, as well as Python test management tool Tox.

Different from most of the prior RTP studies, we realistically evaluate PYTEST-RANKING by actually running CI builds with different RTP orders produced by PYTEST-RANKING, and comparing them to Pytest's default order. Our evaluation shows that RTP orders produced by PYTEST-RANKING enable a 37%–75% speed-up on regression fault detection, compared to Pytest's default order or randomly ordering tests. PYTEST-RANKING's runtime overhead is 0.03% of the corresponding test-suite run's runtime, and its consumed storage overhead is 5% of the corresponding test-suite run's CI log size. Overall, PYTEST-RANKING is effective and efficient. PYTEST-RANKING also exposes flaky tests in the evaluated projects, a majority of which are order-dependent flaky tests, and only a handful have been fixed by developers.

Our analysis on the impact of test duration variation shows that TSR duration varies much less than individual test duration under different orders in the same commit. As a result, although the APFDc values of an RTP technique are different from its HAPFDc counterparts, both APFDc and HAPFDc produce the same RTP technique effectiveness ranking.

We study the impact of flaky test failure treatment on the executed reordered test suites. We have the same observation as prior studies on simulated reordered test suites [40, 65]: the history-based RTP technique is ranked higher than the change-aware technique when flaky test failures are considered as fault-detecting failures in RTP effectiveness computation, e.g., APFDc and HAPFDc. The relative ranking of other evaluated techniques is unchanged.

At last, we simulate our reordered test suites with test outcomes and durations from the test-suite runs that were ran in default order. We compare the APFDc values from the simulations with the APFDc values from the actual reruns. The comparison shows that the ranking of RTP techniques obtained from simulations is the same as reruns.

**Conclusion:** PYTEST-RANKING can speed up regression fault detection in practice, and is accompanied by high usability, low runtime overhead, and minimum deployment effort. However, developers should actively resolve failures from order-dependent flaky tests before deploying RTP in practice. On the other hand, researchers can still simulate orders in RTP evaluation if executing the reordered test suites is too costly, because the overall RTP effectiveness ranking should be similar in both experiment setups.

## 5.2 FUTURE WORK

This section discusses some possible future work that this dissertation has opened.

### 5.2.1 Further Development of RTP Tool

**Hierarchy-aware RTP order with interleavings** When PYTEST-RANKING runs Hierarchy-Aware (HA) RTP at certain group level [135], test groups are reordered by their priority scores, while tests within each group are executed consecutively in default order so that within-group order dependencies are not broken. This implementation misses the opportunity of interleaving tests from different groups to further speed up fault detection.

We can implement and study RTP algorithms that interleave tests from different groups, and respect the relative within-group test orders. For example, consider a test suite with test groups $A, B$, where each group has two tests: $[A_1, A_2, B_1, B_2]$. Currently, HA RTP in PYTEST-RANKING can only run two orders at the group level: $[A_1, A_2, B_1, B_2]$, and $[B_1, B_2, A_1, A_2]$. However, it is possible that the following orders can find fault faster: $[A_1, B_1, A_2, B_2]$, and $[B_1, A_1, B_2, A_2]$—these orders interleave tests across groups, but still respect the relative order within each group. While implementing such an RTP algorithm, further studies are needed to understand the overhead from interleaving tests across groups in Python/Pytest [253].

**Running RTP with test parallelism** PYTEST-RANKING can be run together with test parallelism in Pytest [228]. Test Parallelism (TP) improves regression testing efficiency, while also adding extra complexity in CI. For example, tests can be first ordered by RTP, then workers are assigned tests from the RTP-ordered list in a certain fashion (e.g., round-robin). On the other hand, tests can also be first assigned to workers, after which RTP re-orders tests in each worker separately. TP has its own performance-affecting factors and constraints, e.g., load balancing strategies, and test group granularity at which TP is run. The trade-offs and optimization opportunities when combining RTP and TP are unclear.

Using PYTEST-RANKING, we can further (1) explore the opportunities and constraints when deploying both RTP and TP in CI, (2) assess cost-effectiveness of existing prioritized-then-parallelized RTP algorithms and metrics in practice [45], (3) study parallelized-then-prioritized RTP, and (4) develop robust and efficient parallel RTP.

**Handling non-regression test failures (e.g., flaky test failures)** RTP is supposed to report test failures to developers as soon as they occur. If the prioritized failing tests are irrelevant to regressions, developers can be distracted and annoyed by the early-reported failures. Flaky tests have been a major source of these irrelevant test failures.

To encourage more adoption of RTP, we should help alleviate the burden of flaky tests from developers. For example, we can integrate tools/techniques that identify flaky tests into PYTEST-RANKING, so that PYTEST-RANKING can isolate failures from these tests.

As a starting point, PYTEST-RANKING can regularly execute sanitization test-suite runs to record possible flaky tests using flaky test detection tools [248, 254, 255], and report their failures in an isolated summary section during subsequent RTP test-suite runs. PYTEST-RANKING may also de-prioritize the detected flaky tests during RTP ordering.

As more techniques for detecting and fixing flaky tests are being developed, future work on RTP should consider how to properly utilize them.

**Incorporating other RTP results** PYTEST-RANKING implements simplest and among the most effective techniques from recent RTP studies [36, 37, 49]. For future work, PYTEST-RANKING can be used as a testbed to experiment with other existing or newly-developed RTP algorithms and strategies. As a starting point, it could be helpful to (1) experiment with the use of learning algorithms to find optimal RTP technique combination in replacement of the linear-weighted combination baseline in PYTEST-RANKING [43], and (2) experiment with other possible, lightweight RTP techniques that use readily-usable CI features [36].

### 5.2.2 RTP for Configuration Testing

**Prioritization for configuration fuzzing drivers** Configuration tests can be selected as fuzzing drivers to find configuration faults in software systems [256]. A selected configuration test will be executed many times, each time with a different test input generated by the fuzzer, hoping that the test will fail on the test input and expose fault(s).

We can study whether RTP can prioritize configuration tests that are likely to expose faults, which will be first provided as candidates to the fuzzer to improve its fault detection efficiency. Such RTP algorithms can leverage heuristics extracted from the to-be-fuzzed software version or from prior configuration test-suite run results. For example, one possible heuristic can be the number of covered configuration parameters divided by the runtime of a test, i.e., the number of covered parameters per unit time.

**More RTP techniques and studies on configuration testing** Simple RTP algorithms that use historical test outcomes have shown promising results in some RTP studies, e.g., running frequently failing tests first. However, their actual fault detection cost-effectiveness is often plagued by misleading test failures from flaky tests [65]. However, the impact of test flakiness on configuration testing is unclear. We can (1) study the impact of test flakiness on configuration testing, and (2) assess the promise of RTP algorithms based on configuration change history, under configuration test flakiness.

Prior work has developed a configuration testing tool in Java [53]. Future work can develop an RTP tool for configuration testing.

### 5.2.3 Other Future Work on RTP

**Benchmarks, metrics, and user studies** With PYTEST-RANKING, we could build executable and reproducible RTP benchmarks on Python. The benchmark could provide a sandbox for convenient RTP evaluation. The benchmark could additionally incorporate test selection and parallelization, so that it can support regression testing studies on scenarios where multiple testing techniques are applied.

A number of metrics have been proposed for RTP evaluation, as surveyed [28]. Some metrics distinctly differ from the other, while some are variants with slight modifications. However, most of them have only been used on simulated test-suite runs to compute RTP effectiveness. Future work can study the differences and practical implications of these metrics on RTP-ordered test-suite runs obtained from the actual execution of PYTEST-RANKING.

Although RTP aims to reduce developement feedback time, such reduction may not be beneficial to all development settings. Developers like to be informed of the first test failure at the earliest convenience in some cases (e.g., perhaps where cost increases with a longer wait time), while waiting until the entire test-suite run finishes in some other cases. Future work can (1) identify testing scenarios that demand timely validation and debugging of software change, e.g., [257], where RTP could be more beneficial; (2) conduct study of open-source or industry developers to understand what part of their development could benefit from RTP.

**RTP and test selection** RTP has been applied to test selection [23, 36, 42], i.e., tests are selected if their RTP scores are above a certain cutoff. These test selection approaches are lightweight, but unsafe and incomplete. Some studies have thus provided insights into measuring the safety of RTP-based test selection. They provide limited analyses on safety-cutoff-tradeoff [36], or focus RTP features that were not the most effective in this work [23]. Future work can study both the safety and completeness of test selection on more RTP heuristics, to investigate their tradeoffs between cost-saving and safety/completeness. Moreover, because it is common that most tests may not fail even after typical test selection, future work can also study the safety and completeness of further applying RTP-based selection on top of the selected tests.

**Improving RTP evaluation with Large Language Models** One important component of RTP evaluation is failure-to-fault mapping, which links each test failure to the fault that causes the failure. However, failure-to-fault mapping is hard to obtain on real CI builds, because identifying the fault of each failure requires manual inspection. Thus, evaluation of an RTP technique on real CI datasets often makes simpler assumptions on the mapping, e.g., each test failure maps to the same fault.

Moreover, APFDc uses failure-to-fault mapping as input, and considers the cost of each fault. The cost should capture different aspects of the fault, such as the failed test duration and the severity of the fault. However, fault severity also requires significantly manual inspection effort to obtain. Without such information, RTP studies can only use test runtime as the cost, which implicitly assigns all faults with the same level of severity.

Recently, Large Language Models (LLMs) have been applied for fault detection and analysis [258, 259], demonstrating their capability in reasoning about software failures. Future work can thus consider applying LLMs for analyzing what fault caused which test failure, to approximate failure-to-fault mapping and fault severity efficiently.

## 5.3 CLOSING REMARK

Regression testing is an important step in maintaining software reliability in software development. This dissertation presents studies and a tool for applying Regression Test Prioritization to effectively speed up fault detection in regression testing for modern software systems. We hope our work can facilitate more future research and practices of Regression Test Prioritization, providing benefits to software development and testing in general.

# Appendix A: Other Work

This appendix chapter lists the other research work conducted by the dissertation author, in parallel to the work that was presented in the main chapters of the dissertation.

**Bug reproduction test** We develop an effective LLM-agentic approach to generate Bug Reproduction Tests (BRTs) from bug reports in an industrial development context. The generated BRTs can be leveraged to effectively improve downstream Automated Program Repair (APR) performance, as well as to select promising fixes generated by APR systems [57].

**Configuration testing and validation** We develop Ctest, a testing framework to effectively detect misconfigurations and configuration-related faults [52]. Ctest automatically identifies and transforms existing tests into configuration tests, i.e., ctests, to test the changed configuration values with the exercised code. However, Ctest is limited by its slow runtime and reliance on test abundance in the target project. We further develop Ciri, a fast configuration validation framework that leverages the reasoning capability of LLM to flag potentially erroneous values in configuration change [58]; Ciri complements Ctest.

**Efficient deep learning systems** To improve GPU memory efficiency of multi-terabyte-size Deep Learning Recommendation Models (DLRMs), we holistically characterize GPU memory consumption of DLRM distributed training workloads. Based on our findings, we develop an analytical and model-agonistic GPU memory provisioning system, proactively reducing Out-of-Memory (OOM) events and memory under-utilization due to mis-provisioning in DLRM production training [56]. High GPU memory consumption of key-value (KV) cache generated during LLM inference has also emerged as a challenge in LLM serving. We explore the opportunity of using Compute Express Link (CXL) as KV cache storage to improve prefill request serving throughput while adhering to given latency SLO [59].

**Efficient machine learning algorithms** We study the problem of modality selection—selecting the best size-$k$ subset of modalities from $v$ available ones ($k \ll v$) for multimodal learning under resource constraints (e.g., limited memory and GPUs). To tackle this problem, we identify key assumptions of multimodal data, based on which we derive modality selection and ranking algorithms that admit efficient approximate solutions [54, 55].

## References


[1] J. Gumbrecht, "More Than 200 People with Diabetes Injured After Software Issue Drained Insulin Pump Batteries," https://www.cnn.com/2024/05/08/health/tandem-insulin-pump-app-recall, 2024.

[2] S. Blanco, "Report: Tesla Autopilot Involved in 736 Crashes since 2019," https://www.caranddriver.com/news/a44185487/report-tesla-autopilot-crashes-since-2019, 2023.

[3] B. George, "Why Boeing's Problems with the 737 MAX Began More Than 25 Years Ago," https://www.library.hbs.edu/working-knowledge/why-boeings-problems-with-737-max-began-more-than-25-years-ago, 2024.

[4] L. H. Newman, M. Burgess, and A. Greenberg, "How One Bad CrowdStrike Update Crashed the World's Computers," https://www.wired.com/story/crowdstrike-outage-update-windows, 2024.

[5] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects," in *ASE*, 2016.

[6] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility," in *FSE*, 2017.

[7] A. Miller, "A Hundred Days of Continuous Integration," in *Agile*, 2008.

[8] M. Meyer, "Continuous Integration and Its Tools," *IEEE Software*, vol. 31, 2014.

[9] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous Deployment at Facebook and OANDA," in *ICSE Companion*, 2016.

[10] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, "The Top 10 Adages in Continuous Deployment," *IEEE Software*, vol. 34, 2017.

[11] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous Deployment of Mobile Software at Facebook," in *FSE*, 2016.

[12] D. Saff and M. D. Ernst, "Reducing Wasted Development Time via Continuous Testing," in *ISSRE*, 2003.

[13] D. Saff and M. D. Ernst, "An Experimental Evaluation of Continuous Testing During Development," *Software Engineering Notes*, vol. 29, 2004.

[14] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey," *STVR*, vol. 22, 2012.

[15] M. Gligoric, L. Eloussi, and D. Marinov, "Practical Regression Test Selection with Dynamic File Dependencies," in *ISSTA*, 2015.

[16] A. Shi, P. Zhao, and D. Marinov, "Understanding and Improving Regression Test Selection in Continuous Integration," in *ISSRE*, 2019.

[17] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," in *FSE*, 2014.

[18] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration," in *FSE*, 2017.

[19] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub," in *FSE*, 2015.

[20] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-Scale Continuous Testing," in *ICSE-SEIP*, 2017.

[21] Netflix, "Automated Testing on Devices," https://netflixtechblog.com/automated-testing-on-devices-fc5a39f47e24, 2016.

[22] J. Anderson, S. Salem, and H. Do, "Improving the Effectiveness of Test Suite through Mining Historical Data," in *MSR*, 2014.

[23] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," in *ICSE-SEIP*, 2019.

[24] C. Pan and M. Pradel, "Continuous Test Suite Failure Prediction," in *ISSTA*, 2021.

[25] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *ICSE*, 2015.

[26] I. Bouzenia and M. Pradel, "Resource Usage and Optimization Opportunities in Workflows of GitHub Actions," in *ICSE*, 2024.

[27] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review," *ESE*, vol. 27, 2022.

[28] R. Greca, B. Miranda, and A. Bertolino, "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review," *ACM Computing Surveys*, 2023.

[29] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *TSE*, vol. 22, 1996.

[30] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," *TOSEM*, vol. 10, 2001.

[31] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," *ACM Sigplan Notices*, vol. 36, 2001.

[32] H.-Y. Hsu and A. Orso, "MINTS: A General Framework and Tool for Supporting Test-Suite Minimization," in *ICSE*, 2009.

[33] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," in *ICSME*, 1998.

[34] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *TOSEM*, vol. 2, 1993.

[35] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: An Empirical Study," in *ICSM*, 1999.

[36] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration," in *ISSTA*, 2021.

[37] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts," *TSE*, vol. 49, 2022.

[38] J.-M. Kim and A. Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments," in *ICSE*, 2002.

[39] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes," in *ICSE*, 2015.

[40] Q. Peng, A. Shi, and L. Zhang, "Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization," in *ISSTA*, 2020.

[41] 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 *ICSE*, 2020.

[42] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration," in *ISSTA*, 2017.

[43] B. Busjaeger and T. Xie, "Learning for Test Prioritization: An Industrial Case Study," in *FSE*, 2016.

[44] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-Level Test Case Prioritization Using Machine Learning," in *ICMLA*, 2016.

[45] J. Zhou, J. Chen, and D. Hao, "Parallel Test Prioritization," *TOSEM*, vol. 31, 2021.

[46] Q. Luo, K. Moran, and D. Poshyvanyk, "A Large-Scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques," in *FSE*, 2016.

[47] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How Does Regression Test Prioritization Perform in Real-World Software Evolution?" in *ICSE*, 2016.

[48] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing White-box and Black-box Test Prioritization," in *ICSE*, 2016.

[49] R. Cheng, S. Wang, R. Jabbarvand, and D. Marinov, "Revisiting Test-Case Prioritization on Long-Running Test Suites," in *ISSTA*, 2024.

[50] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *ISSTA*, 2021.

[51] R. Cheng, K. Ke, and D. Marinov, "pytest-ranking: A Regression Test Prioritization Tool for Python," in *FSE Companion*, 2025.

[52] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *OSDI*, 2020.

[53] S. Wang, X. Lian, Q. Li, D. Marinov, and T. Xu, "Ctest4J: A Practical Configuration Testing Framework for Java," in *FSE Companion*, 2024.

[54] R. Cheng, G. Balasubramaniam, Y. He, Y.-H. H. Tsai, and H. Zhao, "Greedy Modality Selection via Approximate Submodular Maximization," in *UAI*, 2022.

[55] Y. He, R. Cheng, G. Balasubramaniam, Y.-H. H. Tsai, and H. Zhao, "Efficient Modality Selection in Multimodal Learning," *JMLR*, vol. 25, 2024.

[56] R. Cheng, C. Cai, S. Yilmaz, R. Mitra, M. Bag, M. Ghosh, and T. Xu, "Towards GPU Memory Efficiency for Distributed Training at Scale," in *SoCC*, 2023.

[57] R. Cheng, M. Tufano, J. Cito, J. Cambronero, P. Rondon, R. Wei, A. Sun, and S. Chandra, "Agentic Bug Reproduction for Effective Automated Program Repair at Google," *arXiv:2502.01821*, 2025.

[58] X. Lian, Y. Chen, R. Cheng, J. Huang, P. Thakkar, M. Zhang, and T. Xu, "Large Language Models as Configuration Validators," in *ICSE*, 2024.

[59] Y. Tang, R. Cheng, P. Zhou, T. Liu, F. Liu, W. Tang, K. Bae, J. Chen, W. Xiang, and R. Shi, "Exploring CXL-based KV Cache Storage for LLM Serving," *NeurIPS Machine Learning for Systems Workshop*, 2025.

[60] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases for Regression Testing," *TSE*, vol. 27, 2001.

[61] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing Test Prioritization via Test Distribution Analysis," in *FSE*, 2018.

[62] T. Mattis, P. Rein, F. Dursch, and R. Hirschfeld, "RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization," in *MSR*, 2020.

[63] M. Beller, G. Gousios, and A. Zaidman, "Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub," in *MSR*, 2017.

[64] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration," in *MSR*, 2017.

[65] E. Fallahzadeh and P. C. Rigby, "The Impact of Flaky Tests on Historical Test Prioritization on Chrome," in *ICSE-SEIP*, 2022.

[66] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search," in *ICSE-Companion*, 2016.

[67] P. E. Strandberg, W. Afzal, T. J. Ostrand, E. J. Weyuker, and D. Sundmark, "Automated System Level Regression Test Prioritization in a Nutshell," *Software*, vol. 34, 2017.

[68] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating Test-Suite Reduction in Real Software Evolution," in *ISSTA*, 2018.

[69] J. Anderson, S. Salem, and H. Do, "Striving for Failure: An Industrial Case Study about Test Failure Prediction," in *ICSE*, 2015.

[70] A. Najafi, W. Shang, and P. C. Rigby, "Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report," in *ICSE-SEIP*, 2019.

[71] T. B. Noor and H. Hemmati, "A Similarity-Based Approach for Test Case Prioritization Using Historical Failure Data," in *ISSRE*, 2015.

[72] T. B. Noor and H. Hemmati, "Studying Test Case Failure Prediction for Test Case Prioritization," in *PROMISE*, 2017.

[73] X. Jin and F. Servant, "A Cost-efficient Approach to Building in Continuous Integration," in *ICSE*, 2020.

[74] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco, "Assessing Transition-based Test Selection Algorithms at Google," in *ICSE-SEIP*, 2019.

[75] R. White, J. Krinke, and R. Tan, "Establishing Multilevel Test-to-Code Traceability Links," in *ICSE*, 2020.

[76] E. Knauss, M. Staron, W. Meding, O. Soder, A. Nilsson, and M. Castell, "Supporting Continuous Integration by Code-Churn Based Test Selection," in *RCoSE*, 2015.

[77] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression Testing in the Presence of Non-code Changes," in *ICST*, 2011.

[78] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagappan, “FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services,” in *ICSE*, 2019.

[79] M. Sherriff, M. Lake, and L. Williams, “Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records,” in *ISSRE*, 2007.

[80] G. Salton and C. Buckley, “Term-Weighting Approaches in Automatic Text Retrieval,” *Information Processing & Management*, vol. 24, 1988.

[81] S. E. Robertson, S. Walker, and M. Beaulieu, “Experimentation As a Way of Life: Okapi at TREC,” *Information processing & management*, vol. 36, 2000.

[82] “Git diff documentation,” https://git-scm.com/docs/git-diff#Documentation/git-diff.txt--Ultngt, 2023.

[83] D. Marijan, A. Gotlieb, and A. Sapkota, “Neural Network Classification for Improving Continuous Regression Testing,” in *AITest*, 2020.

[84] R. Mamata, A. Azim, R. Liscano, K. Smith, Y.-K. Chang, G. Seferi, and Q. Tauseef, “Test Case Prioritization using Transfer Learning in Continuous Integration Environments,” in *AST*, 2023.

[85] M. Bagherzadeh, N. Kahani, and L. Briand, “Reinforcement Learning for Test Case Prioritization,” *TSE*, vol. 48, 2021.

[86] S. Omri and C. Sinz, “Learning to Rank for Test Case Prioritization,” in *SBST*, 2022.

[87] “Apache Software Foundation,” https://www.apache.org/, 2023.

[88] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two Case Studies of Open Source Software Development: Apache and Mozilla,” *TOSEM*, vol. 11, 2002.

[89] “Workflow usage limits,” https://docs.github.com/en/actions/administering-github-actions/usage-limits-billing-and-administration, 2025.

[90] “Hadoop CI server,” https://ci-hadoop.apache.org/job/hadoop-multibranch/, 2023.

[91] “Kafka CI server,” https://ci-builds.apache.org/job/Kafka, 2023.

[92] “Jenkins remote access API ,” https://www.jenkins.io/doc/book/using/remote-access-api/, 2023.

[93] “Maven,” http://maven.apache.org, 2023.

[94] “Gradle,” https://gradle.org/, 2023.

[95] “GitHub API - pulls,” https://docs.github.com/en/rest/pulls?apiVersion=2022-11-28, 2022.

[96] "Jenkins pipeline syntax," https://www.jenkins.io/doc/book/pipeline/syntax/#parallel, 2023.

[97] "Kafka Jenkinsfile," https://github.com/apache/kafka/blob/7d39d7400c919a519fb73d93e311eba9b13bbb97/Jenkinsfile#L101, 2023.

[98] "GitHub API - compare two commits," https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28#compare-two-commits, 2022.

[99] A. Danial, "cloc: v1.92," https://doi.org/10.5281/zenodo.5760077, 2021.

[100] J. Candido, L. Melo, and M. d'Amorim, "Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact," in *ASE*, 2017.

[101] Z. Q. Zhou, C. Liu, T. Y. Chen, T. Tse, and W. Susilo, "Beating Random Test Case Prioritization," *Transactions on Reliability*, vol. 70, 2020.

[102] F. Li, J. Zhou, Y. Li, D. Hao, and L. Zhang, "AGA: An Accelerated Greedy Additional Algorithm for Test Case Prioritization," *TSE*, vol. 48, 2021.

[103] M. Abdelkarim and R. ElAdawi, "TCP-Net: Test Case Prioritization using End-to-End Deep Neural Networks," in *ICSTW*, 2022.

[104] J. A. P. Lima and S. R. Vergilio, "A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments," *TSE*, vol. 48, 2020.

[105] A. Sharif, D. Marijan, and M. Liaaen, "DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing," in *ICSME*, 2021.

[106] "Dataset of Long-Running Test Suites," https://zenodo.org/records/12662090, 2024.

[107] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A Survey of Flaky Tests," *TOSEM*, vol. 31, 2021.

[108] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-Test-Aware Regression Testing Techniques," in *ISSTA*, 2020.

[109] S. Greenland, J. Pearl, and J. M. Robins, "Confounding and Collapsibility in Causal Inference," *Statistical science*, vol. 14, 1999.

[110] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *FSE*, 2014.

[111] "Maven Surefire rerunFailingTestsCount," https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html, 2023.

[112] "pytest-rerunfailures," https://pypi.org/project/pytest-rerunfailures, 2023.

[113] "HBase flaky tests dashboard," https://ci-hbase.apache.org/job/HBase-Flaky-Tests/, 2023.

[114] "Hive flaky tests dashboard," http://ci.hive.apache.org/job/hive-flaky-check, 2023.

[115] "JIRA issue," https://issues.apache.org/jira/issues, 2023.

[116] "JIRA fuzzy search," https://confluence.atlassian.com/jirasoftwareserver/advanced-searching-939938733.html, 2023.

[117] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *ISSRE*, 2004.

[118] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *TSE*, vol. 38, 2012.

[119] J. Liang, S. Elbaum, and G. Rothermel, "Redefining Prioritization: Continuous Prioritization for Continuous Integration," in *ICSE*, 2018.

[120] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," in *ICSE*, 2001.

[121] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-Cognizant Test Case Prioritization," 2006.

[122] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical Evaluation of Pareto Efficient Multi-Objective Regression Test Case Prioritisation," in *ISSTA*, 2015.

[123] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online Defect Prediction for Imbalanced Data," in *ICSE*, 2015.

[124] "sklearn.ensemble.HistGradientBoostingRegressor," https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingRegressor.html, 2023.

[125] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," *NeurIPS*, vol. 30, 2017.

[126] "RETECS," https://bitbucket.org/HelgeS/retecs/src/master, 2017.

[127] J. W. Tukey, "Comparing Individual Means in the Analysis of Variance," *Biometrics*, 1949.

[128] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, "Assessing Test Case Prioritization on Real Faults and Mutants," in *ICSME*, 2018.

[129] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements Based Test Prioritization Using Risk Factors: An Industrial Study," *Information and Software Technology*, 2016.

[130] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-Objective Test Prioritization in Software Product Line Testing: An Industrial Case Study," in *SPLC*, 2014.

[131] R. Carlson, H. Do, and A. Denton, "A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study," in *ICSM*, 2011.

[132] "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, 2023.

[133] Q. Peng, A. Shi, and L. Zhang, "IR-Based TCP dataset," https://sites.google.com/view/ir-based-tcp, 2023.

[134] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "TERMINATOR: Better Automated UI Test Case Prioritization," in *FSE*, 2019.

[135] H. Wang, P. Yi, J. Parladorio, W. Lam, D. Marinov, and T. Xie, "Hierarchy-Aware Regression Test Prioritization," in *ISSRE*, 2024.

[136] X. Yang, K. Tang, and X. Yao, "A Learning-to-Rank Approach to Software Defect Prediction," *Transactions on Reliability*, vol. 64, 2014.

[137] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive Random Test Case Prioritization," in *ASE*, 2009.

[138] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-Aware Test-Case Prioritization using Integer Linear Programming," in *ISSTA*, 2009.

[139] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *TSE*, vol. 33, 2007.

[140] L. Zhang, D. Marinov, and S. Khurshid, "Faster Mutation Testing Inspired by Test Prioritization and Reduction," in *ISSTA*, 2013.

[141] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. van Gemund, "Prioritizing Tests for Software Fault Localization," in *QSIC*, 2010.

[142] Y. Qi, X. Mao, and Y. Lei, "Efficient Automated Program Repair Through Fault-Recorded Testing Prioritization," in *ICSM*, 2013.

[143] Y. Lou, S. Benton, D. Hao, L. Zhang, and L. Zhang, "How Does Regression Test Selection Affect Program Repair? An Extensive Study on 2 Million Patches," *arXiv:2105.07311*, 2021.

[144] A. Ghanbari, S. Benton, and L. Zhang, "Practical Program Repair via Bytecode Mutation," in *ISSTA*, 2019.

[145] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization," in *ISSTA*, 2008.

[146] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing Field Failures in System Configurable Software: Cost-Based Prioritization," in *ISSRE*, 2009.

[147] Z. Pan, S. Zhou, J. Wang, J. Wang, J. Jia, and Y. Feng, "Test Case Prioritization for Deep Neural Networks," in *DSA*, 2022.

[148] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing Test Inputs for Deep Neural Networks via Mutation Analysis," in *ICSE*, 2021.

[149] A. Haghighatkhah, M. Mantyla, M. Oivo, and P. Kuvaja, "Test Prioritization in Continuous Integration Environments," *Journal of Systems and Software*, vol. 146, 2018.

[150] A. Sherman, P. Lisiecki, A. Berkheimer, and J. Wein, "ACMS: Akamai Configuration Management System," in *NSDI*, 2005.

[151] B. Maurer, "Fail at Scale: Reliability in the Face of Rapid Change," *CACM*, vol. 58, 2015.

[152] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with PeerPressure," in *OSDI*, Dec. 2004.

[153] S. Mehta, R. Bhagwan, R. Kumar, B. Ashok, C. Bansal, C. Maddila, C. Bird, S. Asthana, and A. Kumar, "Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis," in *NSDI*, 2020.

[154] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," in *SOSP*, 2015.

[155] J. Shieber, "Facebook Blames a Server Configuration Change for Yesterday's Outage," https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage, 2019.

[156] R. Johnson, "More Details on Today's Outage," http://www.facebook.com/note.php?note_id=431441338919, Sep. 2010.

[157] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines.* Morgan and Claypool Publishers, 2018.

[158] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? Lessons from hundreds of service outages," in *SoCC*, 2016.

[159] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *USITS*, 2003.

[160] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and Dealing with Operator Mistakes in Internet Services," in *OSDI*, 2004.

[161] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An Empirical Study on Configuration Errors in Commercial and Open Source Systems," in *SOSP*, 2011.

[162] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys*, vol. 47, 2015.

[163] S. Kendrick, "What Takes Us Down?" *USENIX ;login:*, vol. 37, 2012.

[164] A. Rabkin and R. Katz, "How Hadoop Clusters Break," *IEEE Software*, vol. 30, 2013.

[165] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early Detection of Configuration Errors to Reduce Failure Damage," in *OSDI*, 2016.

[166] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *FSE*, 2015.

[167] T. Xu and O. Legunsen, "Configuration Testing: Testing Configuration Values as Code and with Code," *arXiv:1905.12195*, 2019.

[168] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud," in *Middleware*, 2017.

[169] P. Huang, W. J. Bolosky, A. Sigh, and Y. Zhou, "ConfValley: A Systematic Configuration Validation Framework for Cloud Services," in *EuroSys*, 2015.

[170] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection," in *VLDB*, 2015.

[171] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *FSE*, 2020.

[172] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems," in *KDD*, 2006.

[173] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing Configuration File Specifications with Association Rule Learning," in *OOPSLA*, 2017.

[174] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic Automated Language Learning for Configuration Files," in *CAV*, 2016.

[175] O. Tuncer, N. Bila, C. Isci, and A. K. Coskun, "ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud," IBM Research, Tech. Rep., Mar. 2018.

[176] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based Online Configuration Error Detection," in *USENIX ATC*, 2011.

[177] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection," in *ASPLOS*, 2014.

[178] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations," in *USENIX ATC*, 2020.

[179] C. Xiang, Y. Wu, B. Shen, M. Shen, H. Huang, T. Xu, Y. Zhou, C. Moore, X. Jin, and T. Sheng, "Towards Continuous Access Control Validation and Forensics," in *CCS*, 2019.

[180] Q. Huang, H. J. Wang, and N. Borisov, "Privacy-Preserving Friends Troubleshooting Network," in *NDSS*, 2005.

[181] H. J. Wang, Y.-C. Hu, C. Yuan, Z. Zhang, and Y.-M. Wang, "Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting," in *IPTPS*, 2004.

[182] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim, "SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems," in *FSE*, 2013.

[183] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines," *TSE*, vol. 40, 2014.

[184] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems," in *ASE*, 2018.

[185] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A Comparison of 10 Sampling Algorithms for Configurable Systems," in *ICSE*, 2016.

[186] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *SOSP*, 2013.

[187] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A Tool for Assessing Resilience to Human Configuration Errors," in *DSN*, 2008.

[188] 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 *ICST*, 2019.

[189] "openctest," https://github.com/xlab-uiuc/openctest, 2020.

[190] "Docker Hub," https://www.docker.com/products/docker-hub, 2020.

[191] T. Xu and D. Marinov, "Mining Container Image Repositories for Software Configurations and Beyond," in *ICSE*, 2018.

[192] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *FSE*, 1999.

[193] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?" in *ICSE*, 2014.

[194] M. Attariyan and J. Flinn, "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis," in *OSDI*, 2010.

[195] S. Zhang and M. D. Ernst, "Automated Diagnosis of Software Configuration Errors," in *ICSE*, 2013.

[196] A. Rabkin and R. Katz, "Precomputing Possible Configuration Error Diagnosis," in *ASE*, 2011.

[197] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration Debugging as Search: Finding the Needle in the Haystack," in *OSDI*, 2004.

[198] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software," in *OSDI*, 2012.

[199] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *ICSE*, 2018.

[200] M. Parfianowicz and G. Lewandowski, "OpenClover," https://openclover.org, 2017–2018.

[201] "JavaParser," https://javaparser.org/about.html, 2020.

[202] M. Stone, "Cross-Validatory Choice and Assessment of Statistical Predictions," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 36, 1974.

[203] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the Gap between the Total and Additional Test-Case Prioritization Strategies," in *ICSE*, 2013.

[204] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects," *TSE*, vol. 45, 2018.

[205] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An Evolutionary Study of Configuration Design and Implementation in Cloud Systems," in *ICSE*, 2021.

[206] S. Wang, X. Lian, D. Marinov, and T. Xu, "Test Selection for Unified Regression Testing," in *ICSE*, 2023.

[207] "[SUREFIRE-2041] Ordering test classes and methods according to -Dtest property," https://github.com/apache/maven-surefire/pull/560, 2025.

[208] "pytest-ranking," https://github.com/softwareTestingResearch/pytest-ranking, 2025.

[209] "pytest-ranking," https://pypi.org/project/pytest-ranking, 2025.

[210] "Pytest," https://docs.pytest.org/en/stable/, 2025.

[211] "pytest-dev," https://github.com/pytest-dev, 2025.

[212] "Entry points," https://packaging.python.org/en/latest/specifications/entry-points/, 2025.

[213] "Making your plugin installable by others," https://docs.pytest.org/en/stable/how-to/writing_plugins.html#making-your-plugin-installable-by-others, 2025.

[214] "Writing hooks," https://docs.pytest.org/en/stable/how-to/writing_hook_functions.html#writinghooks, 2025.

[215] "Pluggy: the pytest plugin system," https://pluggy.readthedocs.io/en/stable/index.html, 2025.

[216] "Pytest plugin manager," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytestpluginmanager, 2025.

[217] "pytest_collection_modifyitems," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_collection_modifyitems, 2025.

[218] "Item," https://docs.pytest.org/en/7.1.x/reference/reference.html#item, 2025.

[219] "Item node ID," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.nodes.Node.nodeid, 2025.

[220] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *ICSE*, 2015.

[221] "pytest_runtest_logreport," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_runtest_logreport, 2025.

[222] "TestReport," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.TestReport, 2025.

[223] "Config cache," https://docs.pytest.org/en/7.1.x/reference/reference.html#config-cache, 2025.

[224] "Cache dir," https://docs.pytest.org/en/7.1.x/reference/reference.html#confval-cache_dir, 2025.

[225] "pytest_terminal_summary," https://docs.pytest.org/en/7.1.x/reference/reference.html#pytest.hookspec.pytest_terminal_summary, 2025.

[226] "Configuration," https://docs.pytest.org/en/stable/reference/customize.html#command-line-options-and-configuration-file-settings, 2025.

[227] "Caching dependencies," https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/caching-dependencies-to-speed-up-workflows, 2025.

[228] "pytest-xdist," https://pypi.org/project/pytest-xdist/, 2025.

[229] "pytest-dependency," https://pypi.org/project/pytest-dependency, 2025.

[230] "pytest-order," https://pypi.org/project/pytest-order, 2025.

[231] "Rerun failed tests," https://docs.pytest.org/en/stable/how-to/cache.html#how-to-re-run-failed-tests-and-maintain-state-between-test-runs, 2025.

[232] "pytest-random-order," https://pypi.org/project/pytest-random-order, 2025.

[233] "pytest-randomly," https://pypi.org/project/pytest-randomly, 2025.

[234] "PyPI," https://pypi.org, 2025.

[235] "BigQuery," https://docs.pypi.org/api/bigquery, 2025.

[236] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An Empirical Study of Flaky Tests in Python," in *ICST*, 2021.

[237] "REST API endpoints for commit statuses," https://docs.github.com/en/rest/commits/statuses?apiVersion=2022-11-28#get-the-combined-status-for-a-specific-reference, 2025.

[238] "GitHub actions," https://docs.github.com/en/actions, 2025.

[239] "Using GitHub hosted runners," https://docs.github.com/en/actions/using-github-hosted-runners/using-github-hosted-runners/about-github-hosted-runners#using-a-github-hosted-runner, 2025.

[240] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically Revisiting the Test Independence Assumption," in *ISSTA*, 2014.

[241] "List workflow runs," https://docs.github.com/en/rest/actions/workflow-runs?apiVersion=2022-11-28#list-workflow-runs-for-a-repository, 2025.

[242] "Repository archive," https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28#download-a-repository-archive-zip, 2025.

[243] "UV: An extremely fast Python package and project manager," https://docs.astral.sh/uv/, 2025.

[244] "Parametrize," https://docs.pytest.org/en/stable/how-to/parametrize.html, 2025.

[245] "pytest-json-report," https://pypi.org/project/pytest-json-report, 2025.

[246] P. Yi, H. Wang, T. Xie, D. Marinov, and W. Lam, "A Theoretical Analysis of Random Regression Test Prioritization," in *TACAS*, 2022.

[247] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *ISSTA*, 2014.

[248] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests," in *ICST*, 2019.

[249] S. Dutta, A. Arunachalam, and S. Misailovic, "To Seed or Not to Seed? An Empirical Analysis of Usage of Seeds for Testing in Machine Learning Projects," in *ICST*, 2022.

[250] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests," in *FSE*, 2019.

[251] R. Wang, Y. Chen, and W. Lam, "iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests," in *ICSE Companion*, 2022.

[252] Y. Chen and R. Jabbarvand, "Neurosymbolic Repair of Test Flakiness," in *ISSTA*, 2024.

[253] "Xunit-Style Setup," https://docs.pytest.org/en/stable/how-to/xunit_setup.html#how-to-implement-xunit-style-set-up, 2025.

[254] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs for Detecting Order-Dependent Flaky Tests," in *TACAS*, 2021.

[255] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A Large-Scale Longitudinal Study of Flaky Tests," *OOPSLA*, vol. 4, 2020.

[256] J. Li, S. Li, K. Li, F. Luo, H. Yu, S. Li, and X. Li, "ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems," in *ICSE*, 2024.

[257] Y. Lou, J. Yang, S. Benton, D. Hao, L. Tan, Z. Chen, L. Zhang, and L. Zhang, "When Automated Program Repair Meets Regression Testing—An Extensive Study on Two Million Patches," *TOSEM*, vol. 33, 2024.

[258] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large Language Models for Test-Free Fault Localization," in *ICSE*, 2024.

[259] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic Root Cause Analysis via Large Language Models for Cloud Incidents," in *EuroSys*, 2024.