SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Comparative Analysis of Different Approaches for Test Impact Analyses for Real World Test Suites

Malek Ben Slimane

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Comparative Analysis of Different Approaches for Test Impact Analyses for Real World Test Suites

Vergleichende Analyse unterschiedlicher Ansätze für Test-Impact-Analyse für industrielle Test Suites

Author:	Malek Ben Slimane
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	Dr. Elmar Jürgens
	Andreas Punz
Submission Date:	15.August 2023

I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Munich, 15.August 2023 Malek Ben Slimane

Acknowledgments

First of all, I am deeply grateful to Dr. Elmar Jürgens and Andreas Punz for their support and guidance throughout this thesis. Their expert advice has been extremely helpful in shaping the study, overcoming technical challenges and writing the thesis. Thank you once again for this interesting topic and for the opportunity to work with you!

Furthermore, I would like to thank Dennis Welter, Nelson Osacky, Marc Philipp and Pierre Attouche from Gradle Inc. for their time and assistance throughout this thesis. Thank you for the Gradle Enterprise license and for your guidance in installing, configuring and getting started with the tool!

Last but not least, a big thank you to my family for their invaluable support and encouragement.

Abstract

Automated tests play a crucial role in software development as they provide valuable feedback to developers regarding potential faults. However, as software projects grow in complexity and size, the execution time of test suites can become a significant challenge.

In order to address this issue, techniques such as test impact analysis and predictive test selection have been introduced. Both methods aim to accelerate the feedback cycle and optimize the regression testing task by selectively choosing tests based on recent code changes. This results in an improved testing process, ensuring quicker bug detection and more efficient software development. In this thesis, our goal is to conduct a case study on both techniques, in order to compare their efficiency. Test Impact Analysis (TIA) relies on testwise coverage to select and prioritize test cases, while Predictive Test Selection (PTS) employs machine learning algorithms.

The techniques were evaluated on two software systems, not only by using real error data, but also by creating new code mutations and inserting new bugs. The results show that by using TIA, we get a shorter time in which the first error is found, compared to using PTS. Furthermore, TIA stands out when it comes to potential time savings. On the other hand, PTS has a remarkable reliability. In fact, it worked for all types of bugs and could correctly select failing tests. Whereas in many cases, TIA was not able to detect bugs due to an inability to collect testwise coverage for some files. Regarding bugs related to code for which TIA knows its testwise coverage, it could achieve a recall of 100%.

Contents

1 Introduction

Successful software grows, and so does its test suite. At CQSE, e.g., increasingly more of the customers have test suites that take several hours or even days to execute completely. However, the longer the time between the introduction of a bug during development, and its discovery through a test failure, the harder it gets to pinpoint the change that introduced this bug. In other words: the longer the test execution time, the larger the pain or cost for all involved developers and testers.

If the execution of the entire test suite takes too long, a simple solution is to only execute a subset of the suite that requires less time. If this subset is chosen well, its execution can find the majority of the bugs (that the entire test suite can detect) very quickly. The remaining tests can then still be executed every night / week to detect the remaining bugs. This way, this approach can achieve rapid bug discovery times for most new bugs even for large, slow test suites.

How well this works obviously depends on how the subset of tests gets selected.

Many approaches for this have been studied in the research literature, and in recent years, some of them have been implemented in commercial tools. The approaches operate on different data, e.g. test coverage traces, code changes or historical test failures, and use different algorithms to select tests.

In an earlier work [DJG17] at TUM's Informatics Chair 4, one such approach was developed that operates on coverage per test case and uses this data as input to perform test case selection and test case prioritization. This work forms the basis of the test impact analysis functionality that, since 5 years, has been part of Teamscale and in use at some of CQSE's customers.

Since then, further commercial tools for test impact analyses or test selection in general have become available. Examples include: Launchable's and Gradle Enterprise's Predictive Test Selections and Microsoft's integration of a Test Impact Analysis to Azure Pipelines.

However, it is unclear how well these approaches perform on real world software, what their specific strengths and weaknesses are and under which conditions which approach is the best in practice.

This thesis aims at evaluating different approaches for test selection with a standardized set of criteria on real world software systems.

This comprises selecting and configuring commercially available tools and performing a case study on open source software systems with automated test suites in order to evaluate the commercial tools and do a comparative analysis of the implemented test selection techinques.

The thesis makes significant contributions to the field of software testing and test selection and prioritization in a way that it sheds light on the practical effectiveness of multiple test selection techniques and provides valuable guidance for selecting the most suitable approach and the most approptiate tool for optimizing test suites in real-world software projects.

The thesis first defines some frequently used terms and presents the papers related to this work. Then, in chapter 4, it introduces the tools that are relevant for the comparative analysis of the test selection techniques. Chapter 5 presents the case study. Finally, Chapter 6 and 7 serve as the concluding sections of the thesis, providing ideas for further work and deriving a final conclusion.

2 Terms and Definitions

Test suite The collection of all test cases designed to verify the functionality and quality of a software application or system.

Retest-All A testing approach where all test cases from a test suite are executed after a change or modification has been made to the software. It involves rerunning the entire suite of tests without using any form of test selection or prioritizing individual test cases.

Test Impact Analysis (TIA) A technique used in software testing to determine the subset of tests that need to be executed when changes are made to the software. By analyzing the relationships between the changed code in the previous runs and the tests that exercise that code, it identifies the tests that are most likely to be affected by the changes and excludes the unaffected tests, thereby reduces the time and effort required for testing [Rot19].

Impacted tests Subset of test cases that are expected to be influenced or affected by a particular recent change made to the software [Kas19].

Predictive Test Selection (PTS) A technique used in software testing to optimize the execution of test suites by selectively running a subset of tests that are likely to detect defects based on historical information. It usually accomplishes this by applying a machine learning model [Grab].

Avoidable tests Test class executions which were not selected by Predictive Test Selection [Grah].

Unavoidable tests Test class executions selected by Predictive Test Selection considered either relevant or likely to fail [Grah].

3 Related Work

This chapter presents an overview of related work to this thesis.
First, it provides a summary of the most relevant research papers in the areas of test effectiveness, test case selection and prioritization, and defect prediction. This allows us to gain insights into the evolutionary progression of test selection and prioritization techniques. Then, it showcases both past and ongoing concurrent research conducted within the research group.

3.1 Publications in the literature

The research on test selection and prioritization has a rich history, spanning back to earlier times and remaining highly relevant in the present day. As early as 1981, Fischer et al. highlighted the costly nature of the "Retest all" strategy, which relied primarily on individual expertise for test selection [FRC81]. While technology has progressed significantly since then, with automated testing becoming the norm, the ultimate goal remains unchanged: to identify potential errors resulting from program modifications through regression testing.

Since 1981, the majority of existing research literature has focused either on test case selection or test case prioritization individually, without integrating both aspects. However, to the best of our knowledge, the first approach that combines test case selection and prioritization was presented by Wong et al., in 1997 [Won+97]. This approach involves selecting tests based on modified code and subsequently prioritizing the remaining test cases. The prioritization takes into account the increasing cost per additional coverage of each test case.

Test prioritization approaches presented by Walcott et al. in 2006 [Wal+06], Singh et al. in 2010 [SKS10] and Suri and Singhal in 2011 [SS11] introduced the incorporation of a time constraint for test execution. These approaches implicitly perform test selection by excluding lower-priority test cases that exceed the available time frame. However, it is important to note that these selection decisions are not based on code changes since the previous test execution.

In their study, in 2016, Silva et al. [Sil+16] proposed a comprehensive hybrid approach that integrates test case selection and prioritization into a five-stage process, including mapping features to software classes to assess their relevance, calculating the criticality of classes based on coupling, complexity, and relevance, computing the criticality of individual tests by considering the classes they cover. Then, the approach identifies test cases that cover at least one changed class, selects the most critical tests that can be executed within the available time constraints and ordered them based on their criticality.

A more recent development in the field is the work of Spieker et al. in 2017 [Spi+17], who introduced a history-based approach for test case selection and prioritization integrated into continuous integration processes. By using reinforcement learning, the approach is able to learn and improve its efficiency based on past test execution experiences. Instead of coverage and change information, this approach utilizes test case meta-data, last execution time, and previous results for test case selection and prioritization.

In their research, at Facebook, in 2019, Machalica et al. [Mac+19] implemented a failure prediction machine learning model that is trained using various features extracted from continuous integration (CI) and version control system (VCS) metadata, as well as static build dependencies and project identifiers. The learned test selection strategy is based on a gradient boosted decision trees classifier, which utilizes historical test outcomes recorded for changes submitted over the past three months.

3.2 Previous and parallel work in the research group

Florian Dreier's master's thesis laid the foundation for test impact research within the research group. The thesis focused on implementing a tool for test case-specific coverage measurement and using the collected data to develop a combined test selection and prioritization approach. The aim was to leverage the coverage data to improve the efficiency and effectiveness of test case selection and prioritization [DJG17].

In a following master's thesis, Alexander Kaserbacher assesses the effectiveness of test impact analysis within a specific context. This is achieved through a comprehensive case study conducted on a complex information system. The chosen system is actively developed and maintained in an industrial environment, making it an ideal candidate for evaluating the practical application of test impact analysis. Additionally, the thesis emphasizes the significance of leveraging a rich development history to gather insights and evaluate the performance of test impact analysis using real error data [Kas19].

The goal of Jakob Rott's master's thesis is to investigate the effectiveness of test prioritization algorithms in practical scenarios. The thesis focuses on selecting and prioritizing a subset of tests based on real-world test failures and mutation-based analysis. The aim is to improve the efficiency of test execution by providing faster feedback to developers while considering factors such as code changes, test data variations, and infrastructure issues. The thesis also aims to enhance an existing prototype and adapt it to an updated version of the underlying analysis tool. By comparing results from mutations-based analysis with those from a real-world benchmark, the thesis seeks to contribute to the understanding and practical application of test prioritization techniques [Rot19].

Currently, there is a master's thesis underway that focuses on implementing predictive test selection for Teamscale, a software quality analysis platform developed at CQSE. By using advanced algorithms and machine learning models, this research aims to implement an efficient and effective approach to select tests using predictive techniques. The goal is to reduce the overall execution time of test suites while maintaining fault detection capabilities.

4 Tools

In this chapter, we focus on introducing and discussing the tools that are relevant to the evaluation of test selection techniques in this thesis. Specifically, we delve into the features and functionalities of two prominent tools: Teamscale and Gradle Enterprise. Each of these tools employs a distinct test selection technique, making them valuable resources for our analysis. To provide a comprehensive overview, we present a concise description of each tool, accompanied by an explanation of the specific technique employed by each in the process of test selection.

4.1 Teamscale

4.1.1 Tool Overview

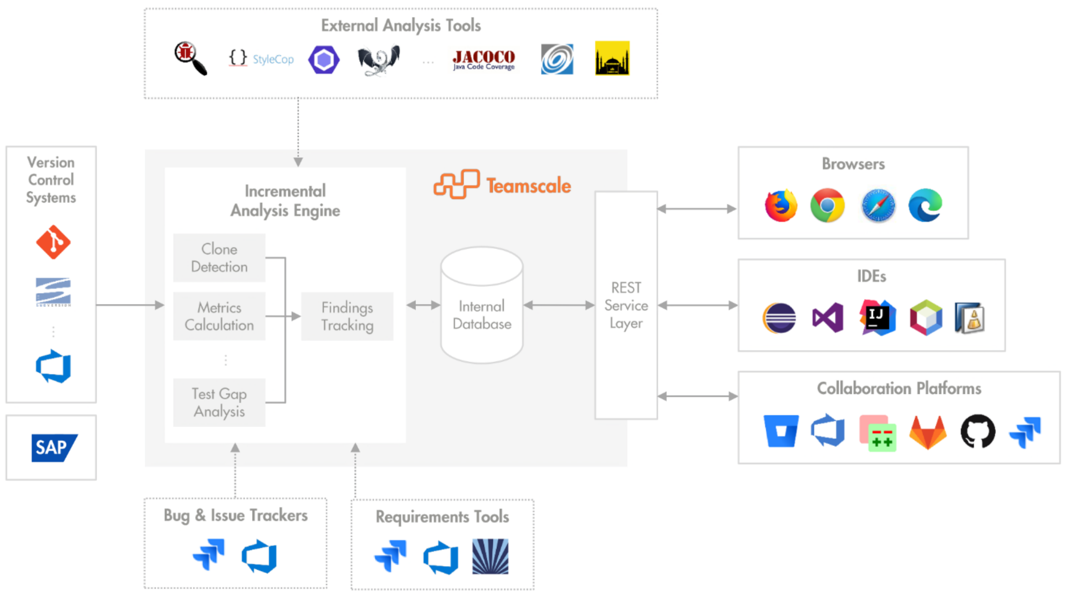


Figure 4.1: Teamscale's Architecture [CQSc]

Teamscale is a comprehensive platform designed to ensure high-quality software throughout the entire software development life cycle. It takes a revolutionary approach to static code analysis by combining essential data from development, aggregating metrics, and providing solid decision criteria. Teamscale goes beyond code quality and also addresses test quality, architecture, and usage scenarios. With its incremental analysis engine, Teamscale provides rapid feedback and identifies root causes on a commit-based level, enabling early detection of emerging problems and deteriorating trends. The platform offers a wide range of code quality analyses, integrates various data sources such as test coverage and usage data, and connects to bug and issue tracking systems. Teamscale's results are stored in a NoSQL store and made accessible through a REST Service API, with web and IDE clients available for seamless integration. As the only Software Intelligence Platform on the market, Teamscale empowers software projects to make data-driven decisions, ensuring maintainable code, efficient testing, and satisfying architecture [CQSc] [CQSb].

Figure 4.1 showcases the architecture of Teamscale.

4.1.2 Test Impact Analysis

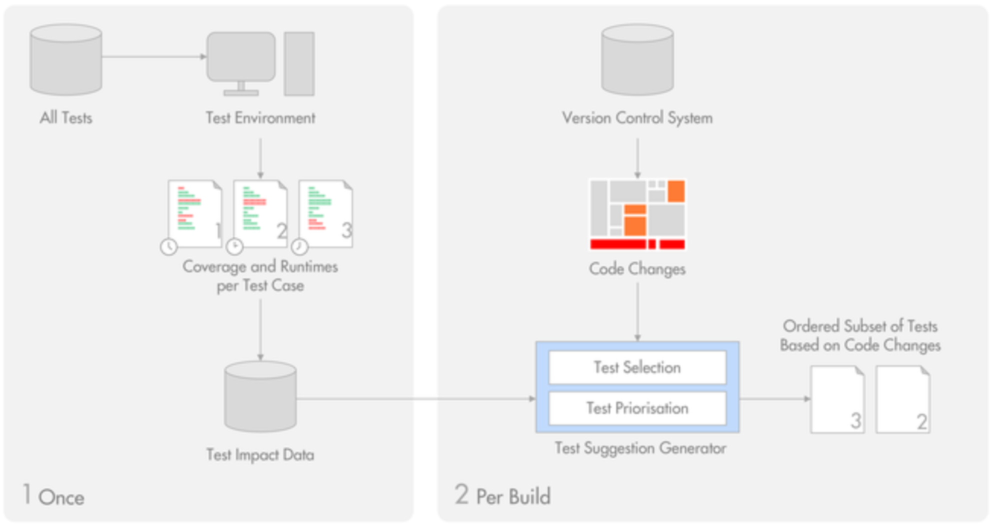


Figure 4.2: Overview of the execution of TIA [CQSa]

Besides various static code analyses, and most importantly for this thesis, Teamscale provides a feature called **Test Impact Analysis (TIA)**. As explained in Chapter 2, this is a mechanism that automates the *selection* and *prioritization* of tests based on the coverage achieved in previous test runs. As shown in Figure 4.2, in order to leverage this capability, testwise coverage data is recorded and uploaded to Teamscale. In subsequent test runs, the Test Runner interacts with Teamscale to determine the tests necessary for validating the changes made in the current commit. By analyzing the data collected during the initial run, Teamscale identifies the specific tests that execute the modified code. In total, it includes the following categories of test cases:

- Test cases that cover methods modified between the baseline and end commit.
- Test cases that have undergone changes themselves between the baseline and end commit.
- Test cases that have been added since the baseline commit

The selected tests are then assigned priorities. In fact, Teamscale employs a prioritization strategy that aims to achieve comprehensive coverage of all modified methods in the codebase as quickly as possible. By prioritizing tests in this manner, Teamscale ensures that the most critical areas affected by code changes receive immediate attention during the testing process. This is done by using a greedy heuristic to assign a score to each test case. As a result, the Test Runner can confidently execute tests in accordance with the prioritized test list, arranged by their score in descending order [Kas19].

4.2 Gradle Enterprise

4.2.1 Tool Overview

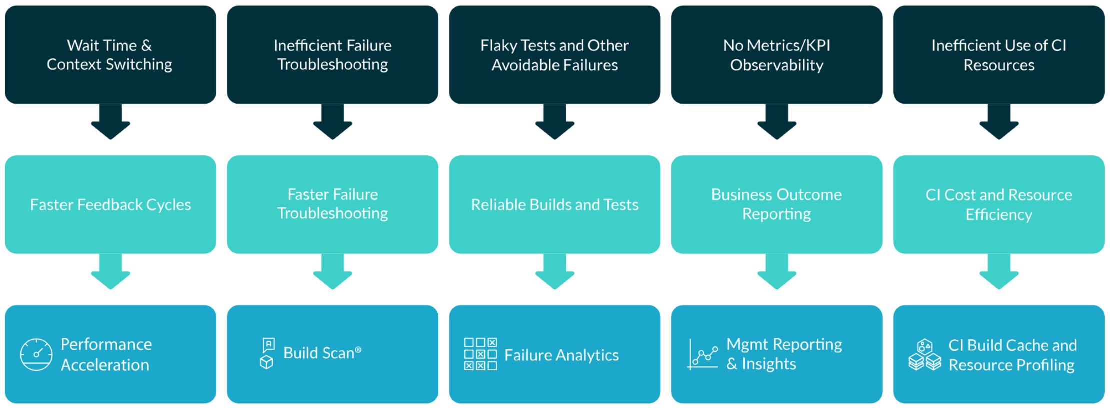

Wait Time & Context Switching
Inefficient Failure Troubleshooting
Flaky Tests and Other Avoidable Failures
No Metrics/KPI Observability
Inefficient Use of CI Resources
Faster Feedback Cycles
Faster Failure Troubleshooting
Reliable Builds and Tests
Business Outcome Reporting
CI Cost and Resource Efficiency
Performance Acceleration
Build Scan®
Failure Analytics
Mgmt Reporting & Insights
CI Build Cache and Resource Profiling

Figure 4.3: Gradle Enterprise solution framework [Grac]

Gradle Enterprise is a software tool known for its ability to enhance build and test performance in software development projects. It offers a range of features aiming at improving the developer experience and optimizing the development process.

One key feature of Gradle Enterprise is its utilization of acceleration technologies, which aim at reducing build and test times in order to achieve faster delivery of code and increase productivity. It also provides data analytics capabilities that enable efficient troubleshooting and performance analysis. Another strength of this tool is its support for various build environments, including Gradle, Maven, and Bazel. This flexibility allows seamless integration into existing build systems, making it a versatile tool suitable for different project setups. Using these different build systems, Gradle Enterprise connects to developer and continuous integration (CI) machines, which makes it an on-premises Software-as-a-Service (SaaS) product. This connectivity enables comprehensive visibility into key performance metrics and trends and empowers developers to make data-driven decisions and optimize their workflows [Grac].

Figure 4.3 demonstrates how Gradle Enterprise contributes to the improvement of the developer experience. Indeed, it illustrates a visual representation of three rows of boxes. In the first row, there are five boxes, each labeled with a distinct problem statement. These problems represent challenges or obstacles that need to be resolved. Moving to the second row, we find another set of boxes representing the goals to be achieved. Each goal corresponds to one of the problems stated in the first row, indicating the desired objective that needs to be accomplished. Finally, in the third and last row, we have a series of boxes that provide solutions to the problems outlined in the first row. These solutions represent the strategies or approaches that can be implemented to overcome the challenges and reach the specified goals.

4.2.2 Predictive Test Selection

Besides various capabilities, Gradle Enterprise offers a feature called **Predictive Test Selection (PTS)**, which holds significant relevance in the context of this thesis. As already explained in Chapter 2, it automates the selection and execution of the most relevant tests for a given code change. Predictive Test Selection operates by utilizing a predictive model developed through the analysis of code changes and test outcomes within Build Scan data. This model continuously learns and adapts to project structures and tests by drawing insights from millions of test executions across diverse projects. When a test suite is about to be executed, Predictive-Test-Selection-enabled builds consult Gradle Enterprise to predict the subset of tests that will offer valuable feedback on the specific changes being tested [Grag]. A more detailed overview is shown in Figure 4.4

The machine learning model employed in Predictive Test Selection considers multiple

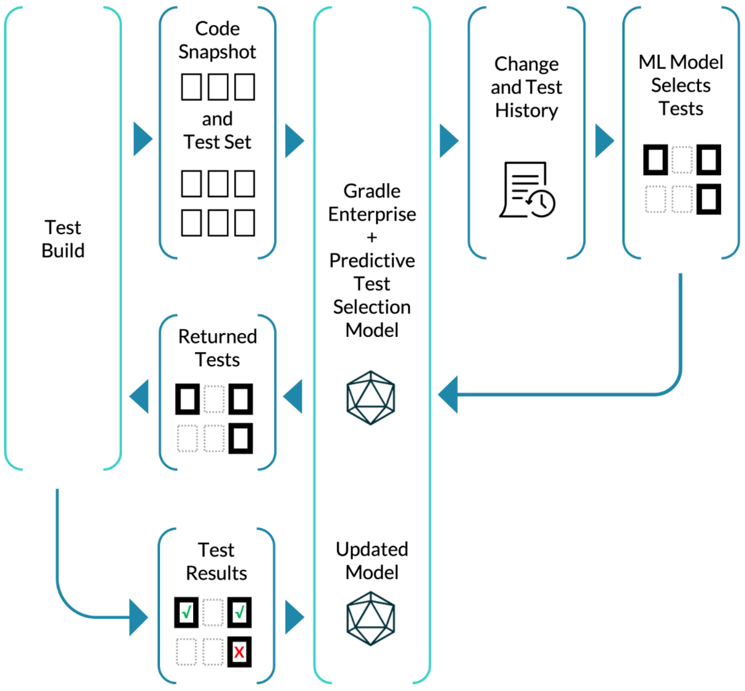


Figure 4.4: Overview of the execution of PTS [Grag]

variables when making decisions. Factors such as the impact of code changes on module tests, the sensitivity of tests to changes, recent modifications, and the flakiness of tests are taken into account. Tests that are new, recently changed, failed, or exhibited flakiness are always chosen. On the other hand, tests that have already passed against the same sources and dependencies are less likely to be selected, as they have demonstrated their stability [Grad].

The model has been meticulously trained using extensive data from Gradle's data partners. This training includes millions of test executions across a range of JVM-based projects, which have been configured on Gradle Enterprise public instances [Graf]. The training process for the model requires a duration of 14 days. This time constraint had a significant influence on our choice of the study objects for this thesis. Further details regarding the selection of study objects will be discussed in Section 5.2.

5 Case Study

In this chapter, we embark on an in-depth exploration and evaluation of the test selection techniques utilized by the previously mentioned tools: Teamscale and Gradle Enterprise. By undertaking a thorough case study on open source software systems with automated test suites, we aim to evaluate and compare the effectiveness and performance of these tools in real-world scenarios, specifically focusing on their test selection capabilities. Indeed, we aim to conduct a comparative analysis of the implemented approaches, namely Test Impact Analysis (TIA) for Teamscale and Predictive Test Selection (PTS) for Gradle Enterprise. Through this evaluation, we seek to shed light on the strengths, weaknesses, and practical applicability of these techniques in the context of real-world software systems.

5.1 Research Questions

The following research questions will be answered using both tools: Teamscale and Gradle Enterprise, contributing to the comparison of Test Impact Analysis vs. Predictive Test Selection. The results are based on previously committed code changes as well as newly generated mutations of the original code to simulate newly introduced bugs. In the following sections, each question will be discussed individually, and a general approach to addressing them will be described.

RQ1: How much faster is the first error detected using each of the test selection techniques compared to retest-all?
By answering this question, we aim to investigate the efficiency and effectiveness of test selection techniques, namely test impact analysis and predictive test selection, in terms of reducing the time required to detect the first error. By comparing the performance of each of these techniques against the traditional retest-all approach, we can assess their impact on accelerating the bug detection process. This is an important comparison metric. The reason behind this importance lies in the need for faster feedback and quicker bug identification, as delays in error detection can significantly impact the development cycle and hinder productivity.

RQ2: How does the number of correctly selected failing tests vary across different test selection techniques, namely Test Impact Analysis (TIA) and Predictive Test Selection (PTS)?
This question aims to examine the effectiveness of these techniques in accurately identifying failing tests that are relevant to a given code change. By comparing the number of correctly selected failing tests for each technique, we can evaluate their ability to pinpoint and prioritize the tests that are most likely to expose bugs. The focus on correctly selected failing tests addresses the need for developers to gain confidence in the safety of test selection techniques. This research question contributes to assessing the reliability and accuracy of TIA as well as PTS in improving the efficiency of bug detection processes.

RQ3: How does the time needed to run the selected tests using Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compare to retest-all approach?
This question aims to investigate and compare the efficiency of the test selection techniques in terms of reducing the overall test execution time. By comparing the time required to run the selected tests for each technique with the time needed for retesting all tests, we can evaluate the potential time savings achieved through test selection performed by TIA vs. PTS. This research question contributes along with RQ2 to understanding the trade-off between bug detection and test execution time, and provides insights into the effectiveness of Test Impact Analysis and Predictive Test Selection in optimizing the testing process.

RQ4: What is the number of failing tests that test impact analysis (TIA) and predictive test selection (PTS) would execute within only 1% of the time required for retesting all tests?
The research question revolves around the comparison of Test Impact Analysis (TIA) and Predictive Test Selection (PTS) techniques in terms of the number of failing tests they can catch within a significantly reduced time frame, specifically 1% of the time required for retesting all tests. As the previous questions, this question aims to explore and compare the effectiveness of TIA and PTS in identifying and executing the subset of tests that are most likely to fail, but this time, within a significantly reduced time frame. By doing so, we can differentiate the ability of these techniques to prioritize critical tests and expedite the bug detection process. It is also beneficial to get an insight on how much test execution time can be restricted and still detect a reasonable amount of errors.

5.2 Study Objects

In this section, we delve into the introduction of the projects utilized in the case study. The choice of the considered projects plays a crucial role in evaluating the effectiveness and performance of the test selection techniques. The selected open source projects for this thesis are **JabRef** and **Rewrite**.

The process of selecting the projects presented its own set of challenges. One of the key considerations was ensuring simultaneous compatibility with both Teamscale and Gradle Enterprise. Each tool had specific constraints and requirements that needed to be taken into account. In particular, Gradle Enterprise required a project with a large commit history and a significant number of failing tests in order to facilitate the training of the machine learning model. Additionally, the training process itself was time-consuming, taking approximately 14 days.

5.2.1 JabRef

Given the already described constraints, the decision was made to include JabRef as one of the projects. JabRef, a cross-platform citation and reference management tool, proved to be an excellent choice as it fulfilled the necessary conditions for training the model effectively. Its extensive commit history and the presence of a substantial number of failing tests provided a solid foundation for the evaluation. In fact, JabRef's codebase comprises 164,000 lines of code, including 117,100 lines of source code and 46,900 lines of test code. The project features a test suite with approximately 3,900 tests and a total of around 18,800 commits. Also, it is written in Java and built using the Gradle build system.

5.2.2 Rewrite

The second project chosen for the case study was the subproject Rewrite from the OpenRewrite project, which focuses on automated code transformations and refactorings for Java projects. An advantage of selecting this project was that it was already configured on a public instance of Gradle Enterprise. This pre-existing configuration saved valuable time that would have otherwise been spent on setting up and configuring the project, allowing the focus to be placed on the evaluation and analysis of the test selection functionality. This multi-modul-project employs the Gradle build system and the JUnit5 testing framework for its test suite. The codebase consists of 198,200 lines of code, with 125,800 lines of source code and 72,400 lines of test code. The project includes a test suite with around 3,200 tests and a commit history of roughly 5,400 commits.

5.2.3 Summary

Table 5.1 provides an overview of the important characteristics of both study objects discussed above.

Table 5.1: Summary of the most important attributes regarding the study objects.

Study object	JabRef	Rewrite
URL	https://www.jabref.org/	https://docs.openrewrite.org/
build	gradle	gradle
framework	JUnit5	JUnit5
Programming language	Java	Java
Total lines of code	164k	198.2k
Source lines of code	117.1k	125.8k
Test lines of code	46.9k	72.4k
Number of tests	3.9k	3.2k
Multi-Modul-Projekt	✗	✓
Number of modules	_	19
Number of commits	18.8k	5.4k

5.3 Study Design

In this section, I will provide a detailed description of the study design used in this thesis, covering the various steps undertaken from the very beginning of the bachelor's thesis work to the retrieval of the results.

First, I will go through the configuration process of both tools, Teamscale and Gradle Enterprise, as well as the setup of the open-source projects JabRef and Rewrite, within these tools. The reason for including the configuration details of the tools and highlighting them is that the evaluation of their setup is an integral part of the thesis and is something that we will discuss later on in Section 5.6.

After that, I will present the general approach used to address the research questions, offering insights into the strategies employed to analyze the data, retrieve results and draw meaningful conclusions from the tools.

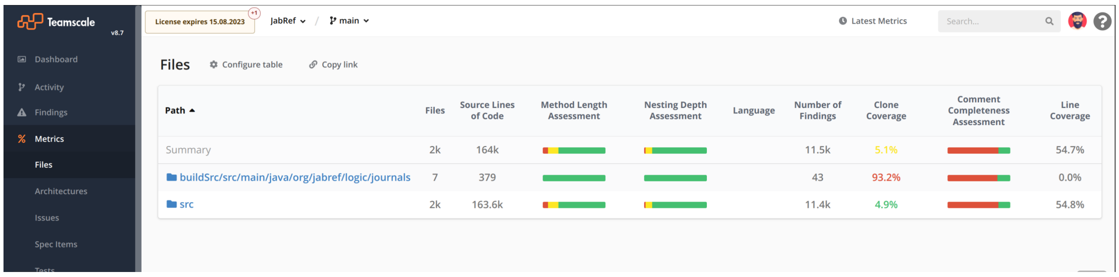
5.3.1 obtaining licenses

Obtaining licenses for a free trial of different software quality tools was a necessary step to conduct a comparative analysis. While acquiring a license for Teamscale was straightforward, as I am currently writing my thesis at CQSE, obtaining licenses for Gradle Enterprise, Launchable, or Sealights presented some challenges. Despite our efforts, we were only successful in securing a license for Gradle Enterprise.

5.3.2 setting up the study objects in Teamscale

Since both study objects are gradle projects and use JUnit5, configuring them in Teamscale and setting up Test Impact Analysis (TIA) is pretty similar.

The initial step involves importing the Gradle project into Teamscale. Once done, an access to view the source code is granted by navigating to the Metrics perspective (Figure 5.1).



Path ▲	Files	Source Lines of Code	Method Length Assessment	Nesting Depth Assessment	Language	Number of Findings	Clone Coverage	Comment Completeness Assessment	Line Coverage
Summary	2k	164k	🟥🟨🟩	🟥🟨🟩		11.5k	5.1%	🟥🟩	54.7%
buildSrc/src/main/java/org/jabref/logic/journals	7	379	🟩	🟩		43	93.2%	🟥🟩	0.0%
src	2k	163.6k	🟥🟨🟩	🟥🟨🟩		11.4k	4.9%	🟥🟩	54.8%

Figure 5.1: Teamscale’s Metrics perspective once a project is added

Next, the inclusion and setup of the teamscale-gradle-plugin is necessary to enable testwise coverage recording.

Before utilizing TIA, it’s essential to gather once testwise coverage for the complete test-suite, ensuring Teamscale’s awareness of the code that a single test case has covered. Based on this coverage information and the changes within a git commit, Teamscale can then calculate and prioritize impacted tests. To achieve this, a new Gradle task is created, functioning similarly to a regular test task, executing all tests by default. Consequently, a testwise coverage report is generated for all test cases. This information is then uploaded to Teamscale [CQSd].

Listing 5.1 shows a small part of an example of testwise coverage generated for the Rewrite project.

```
{
    "duration": 0.005,
    "paths": [
        {
                "files": [
                        {
                                "coveredLines": "60,85,88,229,237-238",
                                "fileName": "Recipe.java"
                        },
                {
                                "coveredLines": "32-34",
                                "fileName": "RecipeBasicsTest.java"
                        }
                ],
                "path": "org/openrewrite"
        }
    ],
    "result": "PASSED",
    "sourcePath": "org/openrewrite/RecipeBasicsTest",
    "uniformPath": "org/openrewrite/RecipeBasicsTest/recipeDoNextWithItself()"
},
```

Listing 5.1: testwise coverage example

5.3.3 setting up the study objects in Gradle Enterprise

As explained in Section 5.2, the Rewrite project has already been configured on a Gradle Enterprise public instance, with existing PTS results for previous commits. Therefore, our focus was only on setting up JabRef on our Gradle Enterprise instance, enabling PTS for this particular project.

To start using Gradle Enterprise features, including Predictive Test Selection, the Gradle Enterprise Gradle plugin must be applied to the build. This step is essential for generating build scans, irrespective of whether we plan to use PTS or not. Moreover, builds must be configured to capture task input files by using the buildScan extension added by the build scan plugin.

To enable Predictive Test Selection, there are various approaches available. One convenient method is to toggle test selection dynamically using project properties specified during the build invocation. By setting the "enablePTS" project property to true, the build script activates test selection for the test task. We can then easily run PTS by executing `./gradlew test -PenablePTS` with this configuration [Graa].

5.3.4 training the machine learning model

To activate Predictive Test Selection, the machine learning model requires training with historical commits to gather data about test execution results. To accomplish this, we replayed 1400 builds of the JabRef project, reaching 100 builds per day over a 14-day period. This data collection process provides the necessary information for PTS to start functioning effectively and making informed decisions about test selection during subsequent builds.

After completing the training phase, we proceeded to generate the necessary data to answer the research questions. To obtain the answers, we relied on historical commits and also introduced newly created mutations. Specifically, we replayed previously executed builds for both projects and generated new code mutations exclusively for JabRef. Further details on the methodology for both aspects are provided in the following subsections.

5.3.5 selecting historical commits

The process of selecting commits for JabRef was straightforward. The primary and only criterion was to choose commits that introduced bugs in the code. As a result, we identified 25 suitable commits for this project.

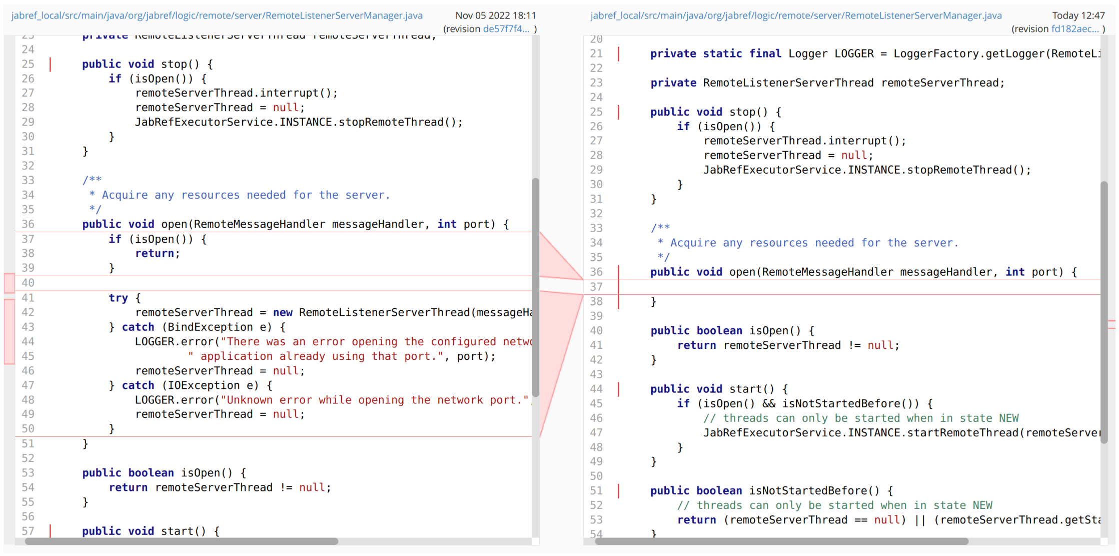
In contrast, selecting commits for Rewrite was a slightly more complex task. Besides requiring commits that introduced bugs, we had an additional criterion of ensuring that the necessary information to answer the research questions was available on Gradle Enterprise. As we used the public instance for this project, we faced limitations in executing PTS or retrieving more information beyond what was already available. Consequently, we carefully chose commits that met these specific requirements. However, our challenges didn't end there. We encountered difficulties in collecting testwise coverage for Teamscale for some commits. As indicated in Table 5.1, Rewrite is a multi-module project, and collecting coverage failed for certain submodules. This constraint led us to further narrow down our selection to commits that still contained bugs, even when ignoring the submodules where collecting coverage failed. In other words, we focused on commits that had bugs in submodules where we successfully collected testwise coverage for Teamscale. Finally, we ended up with 12 commits.

For both study objects, after closer examination of the bugs present in the commits, it becomes clear that the root cause of the failures is not the same across all failing tests. Some failing tests are related to direct code changes, while others are associated with configuration, architecture, or remote setup checks. Consequently, I divided the selected commits for each project into two groups based on the reasons for the failing tests and analyzed them separately.

5.3.6 performing mutation testing

Mutation testing was conducted only for Jabref, where five bugs were introduced into the code, with each bug being contained in a separate Git commit. The research questions were then answered by extracting information from both Teamscale and Gradle Enterprise. The locations where the bugs were introduced, i.e., the classes and methods, were selected randomly using Teamscale's random file picker feature. This allowed the selection of five random files, and within each file, I chose a method randomly and removed its body as part of the mutation process. In Figure 5.2, we can observe an example of a bug that I have purposely introduced as part of the mutation testing.

The decision to perform mutation testing only for JabRef and not for Rewrite was influenced by the configuration of the Rewrite project on a public instance. Unfortunately, we did not have the necessary authorization to use the public instance for running Predictive Test Selection. As a result, we were unable to generate the required data for mutation testing in the Rewrite project.



jabref_local/src/main/java/org/jabref/logic/remote/server/RemoteListenerServerManager.java Nov 05 2022 18:11 (revision de57f7f4...)

```
    private RemoteListenerServerThread remoteServerThread;

    public void stop() {
        if (isOpen()) {
            remoteServerThread.interrupt();
            remoteServerThread = null;
            JabRefExecutorService.INSTANCE.stopRemoteThread();
        }
    }

    /**
     * Acquire any resources needed for the server.
     */
    public void open(RemoteMessageHandler messageHandler, int port) {
        if (isOpen()) {
            return;
        }

        try {
            remoteServerThread = new RemoteListenerServerThread(messageHa
        } catch (BindException e) {
            LOGGER.error("There was an error opening the configured netwo
                    " application already using that port.", port);
            remoteServerThread = null;
        } catch (IOException e) {
            LOGGER.error("Unknown error while opening the network port.",
            remoteServerThread = null;
        }
    }

    public boolean isOpen() {
        return remoteServerThread != null;
    }

    public void start() {
```

jabref_local/src/main/java/org/jabref/logic/remote/server/RemoteListenerServerManager.java Today 12:47 (revision fd182aec...)

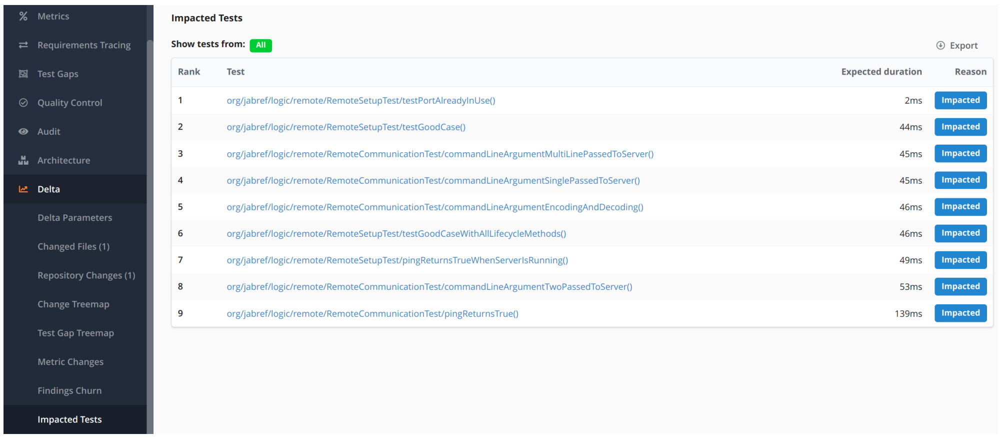
```
    private static final Logger LOGGER = LoggerFactory.getLogger(RemoteLi

    private RemoteListenerServerThread remoteServerThread;

    public void stop() {
        if (isOpen()) {
            remoteServerThread.interrupt();
            remoteServerThread = null;
            JabRefExecutorService.INSTANCE.stopRemoteThread();
        }
    }

    /**
     * Acquire any resources needed for the server.
     */
    public void open(RemoteMessageHandler messageHandler, int port) {

    }

    public boolean isOpen() {
        return remoteServerThread != null;
    }

    public void start() {
        if (isOpen() && isNotStartedBefore()) {
            // threads can only be started when in state NEW
            JabRefExecutorService.INSTANCE.startRemoteThread(remoteServe
        }
    }

    public boolean isNotStartedBefore() {
        // threads can only be started when in state NEW
        return (remoteServerThread == null) || (remoteServerThread.getSta
    }
```

Figure 5.2: Example of a mutation

5.3.7 retrieving data from the tools to answer the research questions

In order to address the research questions effectively, it is essential to gain a comprehensive understanding of the information provided by the user interface of each tool, enabling us to compute the results accordingly.

Starting with Teamscale, it provides the list of impacted tests for each commit along with their respective execution durations. However, to have this data, it is necessary to upload a testwise coverage report for a timestamp preceding the commit in question. A sample result is shown in Figure 5.3.



Impacted Tests

Show tests from: All

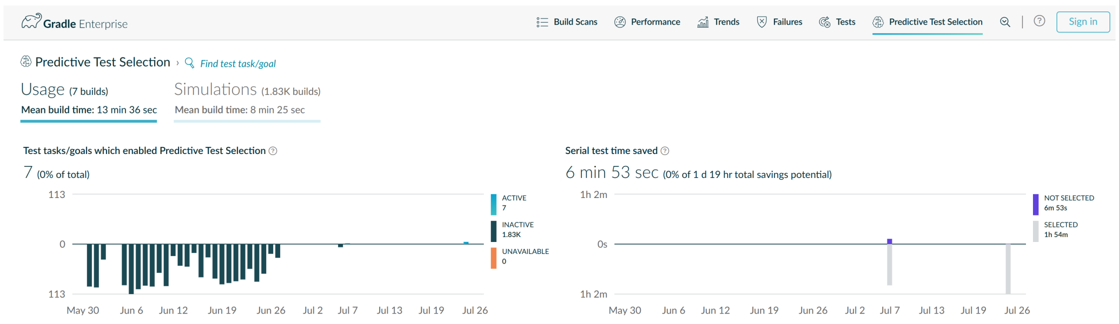
Rank	Test	Expected duration	Reason
1	org/jabref/logic/remote/RemoteSetupTest/testPortAlreadyInUse()	2ms	Impacted
2	org/jabref/logic/remote/RemoteSetupTest/testGoodCase()	44ms	Impacted
3	org/jabref/logic/remote/RemoteCommunicationTest/commandLineArgumentMultiLinePassedToServer()	45ms	Impacted
4	org/jabref/logic/remote/RemoteCommunicationTest/commandLineArgumentSinglePassedToServer()	45ms	Impacted
5	org/jabref/logic/remote/RemoteCommunicationTest/commandLineArgumentEncodingAndDecoding()	46ms	Impacted
6	org/jabref/logic/remote/RemoteSetupTest/testGoodCaseWithAllLifecycleMethods()	46ms	Impacted
7	org/jabref/logic/remote/RemoteSetupTest/pingReturnsTrueWhenServerIsRunning()	49ms	Impacted
8	org/jabref/logic/remote/RemoteCommunicationTest/commandLineArgumentTwoPassedToServer()	53ms	Impacted
9	org/jabref/logic/remote/RemoteCommunicationTest/pingReturnsTrue()	139ms	Impacted

Figure 5.3: Impacted tests perspective in Teamscale

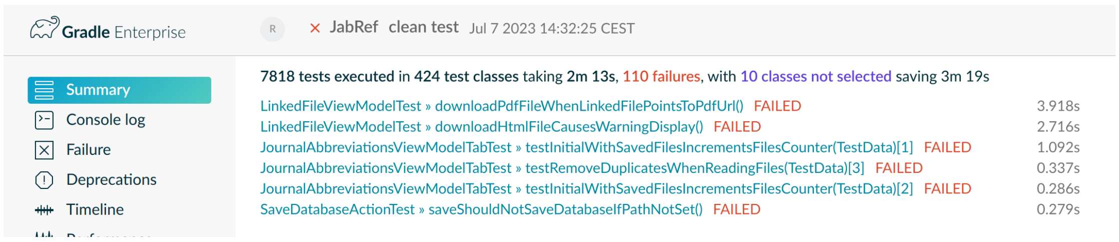
On the other hand, Gradle Enterprise offers two ways to inspect PTS results.

One way is to run PTS itself and then find its impact via the generated Build Scan or the Predictive Test Selection dashboard, once the test run completes. In fact, The Usage view of the Predictive Test Selection dashboard, shown in Figure 5.4a, provides a clear visualization of the usage of PTS and estimated savings trends, along with a summary of matching tasks/goals. Clicking on any test run allows users to access the "Summary" view (Figure 5.4b) of the associated Build Scan, where detailed information, such as the count and estimated time savings, is reported. The "Tests" view (Figure 5.4c) presents a structured and searchable list of tests, offering comprehensive details for each test, including estimated savings based on recent executions. Moreover, the logical prediction explanation for selected and not selected test classes is also available, enhancing the understanding of PTS decisions [Grae].

Alternatively, it is possible to use the Predictive Test Selection Simulator, whose dashboard is in Figure 5.4d. The simulator works by comparing actual test results from Build Scans to what would have happened if the build used Predictive Test Selection, visualizing the %test failures predicted, the time savings potential and the number of avoidable and unavoidable tests, explained in Chapter 2.


Gradle Enterprise
Build Scans
Performance
Trends
Failures
Tests
Predictive Test Selection
Sign in
Predictive Test Selection › Find test task/goal
Usage (7 builds)
Mean build time: 13 min 36 sec
Simulations (1.83K builds)
Mean build time: 8 min 25 sec
Test tasks/goals which enabled Predictive Test Selection
7 (0% of total)
113
0
113
May 30
Jun 6
Jun 12
Jun 19
Jun 26
Jul 2
Jul 7
Jul 13
Jul 19
Jul 26
ACTIVE 7
INACTIVE 1.83K
UNAVAILABLE 0
Serial test time saved
6 min 53 sec (0% of 1 d 19 hr total savings potential)
1h 2m
0s
1h 2m
NOT SELECTED 6m 53s
SELECTED 1h 54m

(a) "Usage" dashboard of Predictive Test Selection

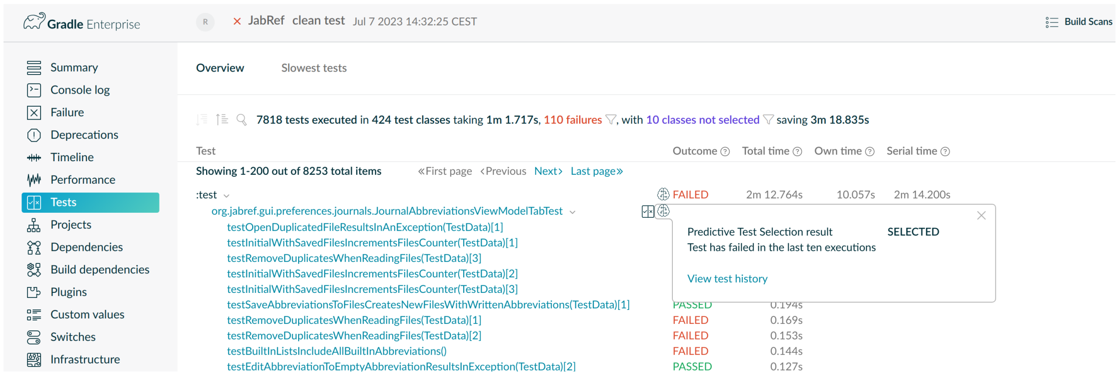


✕ JabRef clean test Jul 7 2023 14:32:25 CEST

7818 tests executed in 424 test classes taking 2m 13s, 110 failures, with 10 classes not selected saving 3m 19s

LinkedFileViewModelTest » downloadPdfFileWhenLinkedFilePointsToPdfUrl() FAILED 3.918s
LinkedFileViewModelTest » downloadHtmlFileCausesWarningDisplay() FAILED 2.716s
JournalAbbreviationsViewModelTabTest » testInitialWithSavedFilesIncrementsFilesCounter(TestData)[1] FAILED 1.092s
JournalAbbreviationsViewModelTabTest » testRemoveDuplicatesWhenReadingFiles(TestData)[3] FAILED 0.337s
JournalAbbreviationsViewModelTabTest » testInitialWithSavedFilesIncrementsFilesCounter(TestData)[2] FAILED 0.286s
SaveDatabaseActionTest » saveShouldNotSaveDatabaseIfPathNotSet() FAILED 0.279s

(b) "Summary" view



✕ JabRef clean test Jul 7 2023 14:32:25 CEST

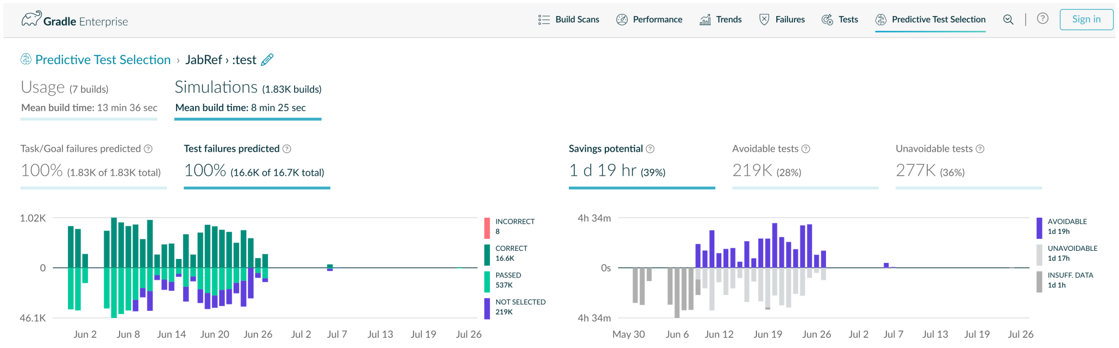
7818 tests executed in 424 test classes taking 1m 1.717s, 110 failures, with 10 classes not selected saving 3m 18.835s

Test	Outcome	Total time	Own time	Serial time
Showing 1-200 out of 8253 total items «First page ‹Previous Next› Last page»				
:test	FAILED	2m 12.764s	10.057s	2m 14.200s
org.jabref.gui.preferences.journals.JournalAbbreviationsViewModelTabTest				
testOpenDuplicatedFileResultsInAnException(TestData)[1]				
testInitialWithSavedFilesIncrementsFilesCounter(TestData)[1]				
testRemoveDuplicatesWhenReadingFiles(TestData)[3]				
testInitialWithSavedFilesIncrementsFilesCounter(TestData)[2]				
testInitialWithSavedFilesIncrementsFilesCounter(TestData)[3]				
testSaveAbbreviationsToFilesCreatesNewFilesWithWrittenAbbreviations(TestData)[1]	PASSED	0.194s		
testRemoveDuplicatesWhenReadingFiles(TestData)[1]	FAILED	0.169s		
testRemoveDuplicatesWhenReadingFiles(TestData)[2]	FAILED	0.153s		
testBuiltInListsIncludeAllBuiltInAbbreviations()	FAILED	0.144s		
testEditAbbreviationToEmptyAbbreviationResultsInException(TestData)[2]	PASSED	0.127s		

Predictive Test Selection result SELECTED
Test has failed in the last ten executions

View test history

(c) "Tests" view



(d) "Simulations" dashboard of Predictive Test Selection

Figure 5.4: Screenshots: Leveraging Gradle Enterprise to answer the Research Questions

5.4 Study Procedure

RQ1: How much faster is the first error detected using each of the test selection techniques compared to retest-all?
To answer this question, I calculated for each commit the time taken to detect the first failing test as a percentage of the retest-all run time. This computation was performed for three scenarios: retest-all, PTS in Gradle Enterprise and TIA in Teamscale.

RQ2: How does the number of correctly selected failing tests vary across different test selection techniques, namely Test Impact Analysis (TIA) and Predictive Test Selection (PTS)?
I collected the percentage of correctly selected failing tests from both test selection tools for each Git commit. To achieve this, I utilized the "%test failures predicted" metric offered by the PTS simulator and obtained the list of impacted tests provided by Teamscale. These data points allowed me to fill in the tables related to this question.

RQ3: How does the time needed to run the selected tests using Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compare to retest-all approach?
The procedure to answer this question is very similar to how I addressed RQ2. The only difference is that I used the "time savings potential" instead of "%test failures detected" in Gradle Enterprise.

RQ4: What is the number of failing tests that test impact analysis (TIA) and predictive test selection (PTS) would execute within only 1% of the time required for retesting all tests?
I calculated the percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all. Having the retest-all execution period and the selected tests by both tools ordered in the order of execution, along with the execution period of each test, I could determine which tests are executed in the first 1% of retest-all execution time.

5.5 Results & Interpretation

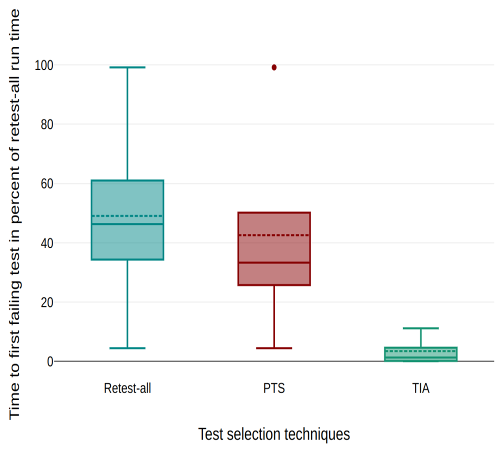
In this section, we present the findings of the case study for the two study objects: Rewrite and JabRef. We first discuss the results obtained from the evaluation of the tools using the Rewrite project, examining the impact of the test selection techniques on the efficiency of test execution. Next, we shift our focus to answering the research questions using our second study object, the JabRef project.

By presenting the results, we get a comprehensive understanding of the efficiency and accuracy of the test selection techniques: TIA and PTS. In fact, the outcomes of this comparative analysis allows us to identify the specific areas where each technique excels and the challenges they may encounter.

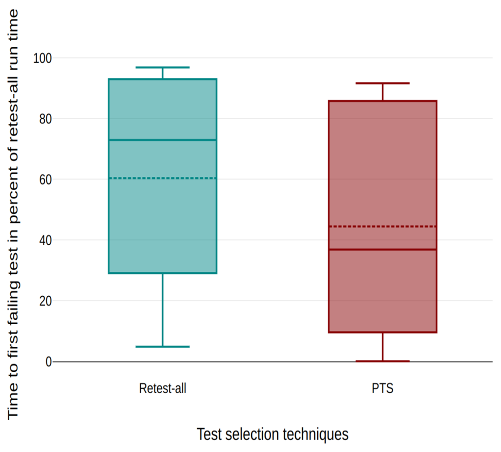
5.5.1 Rewrite

Results based on changes from the project commit history

RQ1: How much faster is the first error detected using each of the test selection techniques compared to retest-all?



(a) Code-related bugs (Table 5.2)



(b) Configuration-related bugs (Table 5.3)

Figure 5.5: Summary of the results of RQ1

For the code-related bug commits in Table 5.2, the average time to find the first bug, measured as a percentage of the retest-all run time, shows that PTS has a slightly better performance than retest-all with average percentages of 42.6% and 49.1%, respectively. However, the TIA technique stands out with an average of 3.4%, significantly lower than the other techniques. In other terms, concerning finding the first bug, PTS in Gradle Enterprise is 1.2 times faster than retest-all, whereas TIA in Teamscale is 14 times faster than retest-all. This indicates that TIA demonstrates superior efficiency in rapidly identifying code-related bugs.

Let us move to Table 5.3, which focuses on configuration-related bug commits. In this case, N/A is indicated for TIA as it is not possible to answer the question in Teamscale. The reason for this is that we are not able to collect testwise coverage when it comes to configuration files. As a matter of fact, Teamscale could not select any failing test related to configuration and hence, could not detect those bugs. So, we do not have any "time to find the first bug", that we could include in the table. Regarding PTS, the results show again that it exhibits a lower average percentage compared to retest-all, with PTS averaging 44.5% and Retest-all averaging 60.4%. This means, that by using Gradle Enterprise, the first bug can be discovered 1.4 times faster compared to executing the complete test suite.

The plots in Figure 5.5 serves as confirmation for the statements made earlier.

Table 5.2: Answers of **1.RQ** for **Rewrite** using commits containing **code**-related bugs: Time needed to find the first bug (time to first failing test in percent of retest-all run time).

Commit-ID	Retest-all	PTS	TIA
ee45c89553978ada3ca3259181b3c6e2ea4501ec	99.2%	99.2%	0.00067%
ceaa158152c7f2f35cad37da86da24454c5213cb	48.3%	33.8%	0.1%
03c458404073082275a752063887c188d213afbe	4.4%	4.4%	2.4%
90138b81efa3ff90ebea1ca4b358d1e8c5087d3a	44.3%	32.8%	11.1%

Table 5.3: Answers of **1.RQ** for **Rewrite** using commits containing only **configuration**-related bugs (no code-related bugs): Time needed to find the first bug (time to first failing test in percent of retest-all run time).

Commit-ID	Retest-all	PTS	TIA
d0ae6649cbaa6a9625ae35080642614901059b24	4.9%	4.9%	N/A
1ba999c9abfcb94019eb07fe7287b189e4fd14cf	96.8%	89.9%	N/A
c75b29fb808835a824593fc19bfb6321d4823c3f	92.6%	84.4%	N/A
f55524edd7757c06bb3197f95a1bd1209220703b	91.6%	91.6%	N/A
cce22dd9eab5e9c7e151ed38c4494eeba9a74205	54.2%	45.4%	N/A
610d056db0de291717e2667152ad7c31522001b8	15%	11.2%	N/A
aa566099d16cbcd4ecda6a6ca71cb305ddd82ee3	94%	0.1%	N/A
ca982c70e4a7c5113c80e466b8ebc49e47d132d4	33.8%	28.3%	N/A

RQ2: How does the number of correctly selected failing tests vary across different test selection techniques, namely Test Impact Analysis (TIA) and Predictive Test Selection (PTS)?

Starting with the commits containing code-related bugs, and looking at the data in Table 5.4, it is evident that both PTS and TIA consistently achieve 100% recall in selecting failing tests. This indicates that both techniques effectively identify and prioritize the execution of tests that are most likely to detect bugs introduced by code changes.

Upon analyzing the data in Table 5.5, it is clear that only PTS achieves a perfect recall rate of 100% in selecting failing tests for commits with configuration-related bugs. This implies the reliability of machine learning as a test selection technique and Gradle Enterprise as a tool. Regarding TIA, we can see that its column is filled with 0%, which confirms the limitation for Teamscale, described in the previous research question.

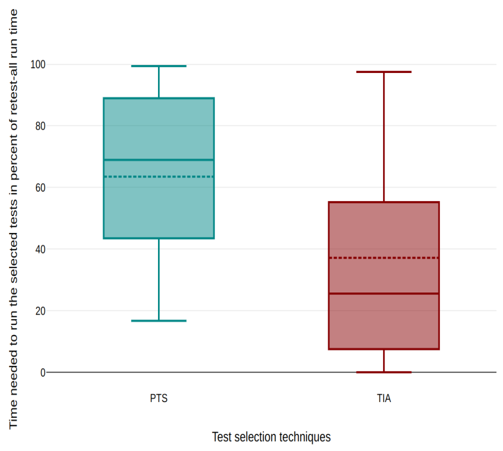
Table 5.4: Answers of **2.RQ** for **Rewrite** using commits containing **code**-related bugs: Percentage of correctly selected failing tests.

Commit-ID	#failing tests in retest-all	PTS	TIA
ee45c89553978ada3ca3259181b3c6e2ea4501ec	1	100%	100%
ceaa158152c7f2f35cad37da86da24454c5213cb	1	100%	100%
03c458404073082275a752063887c188d213afbe	90	100%	100%
90138b81efa3ff90ebea1ca4b358d1e8c5087d3a	1	100%	100%

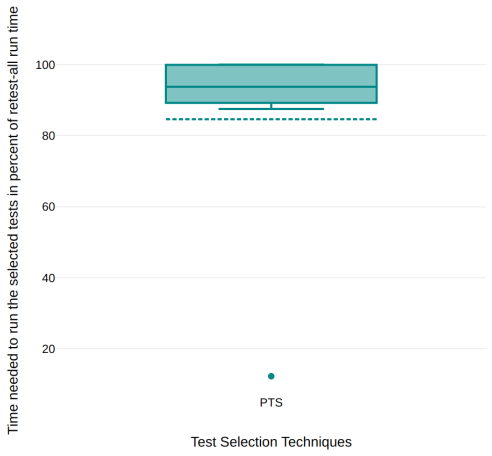
Table 5.5: Answers of **2.RQ** for **Rewrite** using commits containing only **configuration**-related bugs (no code-related bugs): Percentage of correctly selected failing tests.

Commit-ID	#failing tests in retest-all	PTS	TIA
d0ae6649cbaa6a9625ae35080642614901059b24	2440	100%	0%
1ba999c9abfcb94019eb07fe7287b189e4fd14cf	1	100%	0%
c75b29fb808835a824593fc19bfb6321d4823c3f	1	100%	0%
f55524edd7757c06bb3197f95a1bd1209220703b	1	100%	0%
cce22dd9eab5e9c7e151ed38c4494eeba9a74205	1	100%	0%
610d056db0de291717e2667152ad7c31522001b8	1	100%	0%
aa566099d16cbcd4ecda6a6ca71cb305ddd82ee3	47	100%	0%
ca982c70e4a7c5113c80e466b8ebc49e47d132d4	1	100%	0%

RQ3: How does the time needed to run the selected tests using Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compare to retest-all approach?
Measures reveal significant time savings achieved by Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compared to the traditional retest-all approach. Providing results for commits containing code-related bugs, Table 5.6 shows that TIA



(a) Code-related bugs (Table 5.6)



(b) Configuration-related bugs (Table 5.7)

Figure 5.6: Summary of the results of RQ3

achieves remarkable time reductions, with some commits needing only 0.004% of the retest-all run time and an average of 37.2%. PTS also demonstrates notable efficiency, with a minimum of 16.7% of the retest-all run time, but with an average of 63.5%. We can conclude that for this type of bugs, TIA in Teamscale outperforms PTS in Gradle Enterprise as it achieves a significantly higher average of time savings. The plot presented in Figure 5.6a supports this assertions.

Moving to Table 5.7, PTS once again proves efficiency regarding commits containing only configuration-related bugs. Similarly to the previous table, it achieves a minimum of 12.3%, meaning that only 12.3% of retest-all run time is needed to execute the selected tests by Gradle Enterprise. On the other hand, for two commits, PTS selected the complete test suite. Overall, an average of approximately 16% of time savings is achieved. The table's remarkably low values for TIA suggest outstanding efficiency, with time savings reaching 100% for three commits. At first glance, this might indicate that TIA performs exceptionally well. However, it is essential to consider that TIA struggles to select the desired test cases specifically for this type of commits. Consequently, achieving such high time savings in this scenario does not necessarily indicate beneficial performance or high efficiency. On the contrary, it serves as evidence of TIA's limited reliability and accuracy when dealing with this particular type of bugs.

Table 5.6: Answers of **3.RQ** for **Rewrite** using commits containing **code**-related bugs: Time needed to run the selected tests (in percent of retest-all run time).

Commit-ID	PTS	TIA
ee45c89553978ada3ca3259181b3c6e2ea4501ec	99.4%	0.004%
ceaa158152c7f2f35cad37da86da24454c5213cb	85.5%	10%
03c458404073082275a752063887c188d213afbe	16.7%	97.5%
90138b81efa3ff90ebea1ca4b358d1e8c5087d3a	52.4%	41.1%

Table 5.7: Answers of **3.RQ** for **Rewrite** using commits containing only **configuration**-related bugs (no code-related bugs): Time needed to run the selected tests (in percent of retest-all run time).

Commit-ID	PTS	TIA
d0ae6649cbaa6a9625ae35080642614901059b24	100%	0%
1ba999c9abfcb94019eb07fe7287b189e4fd14cf	89.8%	0%
c75b29fb808835a824593fc19bfb6321d4823c3f	87.5%	1.8%
f55524edd7757c06bb3197f95a1bd1209220703b	100%	2.8%
cce22dd9eab5e9c7e151ed38c4494eeba9a74205	91.3%	0.04%
610d056db0de291717e2667152ad7c31522001b8	96.2%	4.5%
aa566099d16cbcd4ecda6a6ca71cb305ddd82ee3	12.3%	0%
ca982c70e4a7c5113c80e466b8ebc49e47d132d4	99.9%	0.01%

RQ4: What is the number of failing tests that test impact analysis (TIA) and predictive test selection (PTS) would execute within only 1% of the time required for retesting all tests?
Focusing on commits containing code-related bugs, the results reveal that TIA and PTS exhibit different behaviors. For 50% of the analyzed commits in Table 5.8, TIA could execute all failing tests within that incredibly short time frame, namely 1% of retest-all run time. This promising performance of TIA, demonstrates the efficacy of the prioritization technique employed, which ensures that the most crucial tests are executed first. On the other hand, PTS shows no executions of failing tests when it comes to a limited lapse of time.

In the context of the second set of commits, as shown in Table 5.9, the analysis mainly focuses on the results obtained for PTS, as TIA was unable to detect any configuration-related bugs. Once again, we encounter a less than satisfactory efficiency of PTS when examining the number of bugs predicted within a limited time frame. Among the 8 commits analyzed, PTS managed to execute only a few failing test cases for a single commit, and the results were not significant. Specifically, it executed merely 14.9% of the failing tests for that particular commit, indicating a relatively low performance in this context.

Table 5.8: Answers of **4.RQ** for **Rewrite** using commits containing **code**-related bugs: Percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all.

Commit-ID	PTS	TIA
ee45c89553978ada3ca3259181b3c6e2ea4501ec	0%	100%
ceaa158152c7f2f35cad37da86da24454c5213cb	0%	100%
03c458404073082275a752063887c188d213afbe	0%	0%
90138b81efa3ff90ebea1ca4b358d1e8c5087d3a	0%	0%

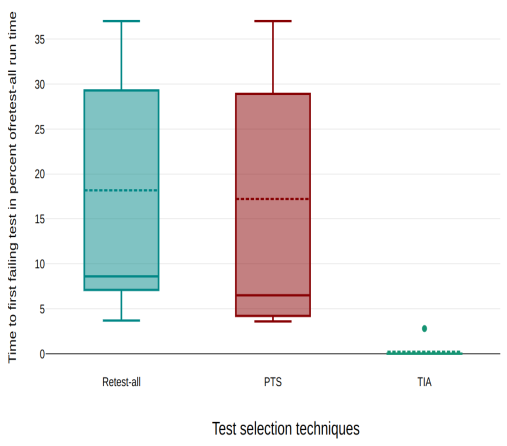
Table 5.9: Answers of **4.RQ** for **Rewrite** using commits containing only **configuration**-related bugs (no code-related bugs): Percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all.

Commit-ID	PTS	TIA
d0ae6649cbaa6a9625ae35080642614901059b24	0%	N/A
1ba999c9abfcb94019eb07fe7287b189e4fd14cf	0%	N/A
c75b29fb808835a824593fc19bfb6321d4823c3f	0%	N/A
f55524edd7757c06bb3197f95a1bd1209220703b	0%	N/A
cce22dd9eab5e9c7e151ed38c4494eeba9a74205	0%	N/A
610d056db0de291717e2667152ad7c31522001b8	0%	N/A
aa566099d16cbcd4ecda6a6ca71cb305ddd82ee3	14.9%	N/A
ca982c70e4a7c5113c80e466b8ebc49e47d132d4	0%	N/A

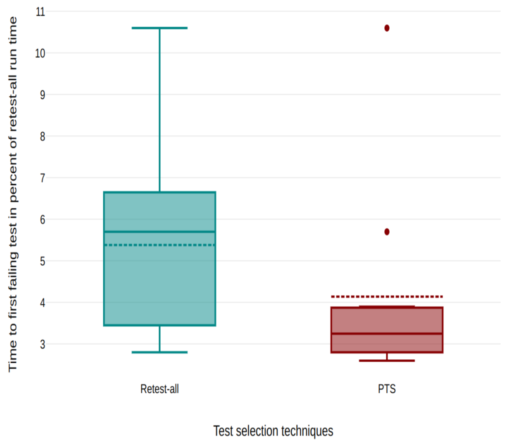
5.5.2 JabRef

Results based on changes from the project commit history

RQ1: How much faster is the first error detected using each of the test selection techniques compared to retest-all?
Table 5.10 presents the results for code-related bug commits, where we assess the average time to discover the first bug, represented as a percentage of the retest-all run time. On average, PTS shows a slightly better performance than retest-all, with percentages of 17.2% and 18.2%, respectively. However, similarly to the first study



(a) Code-related bugs (Table 5.10)



(b) Configuration-related bugs (Table 5.11)

Figure 5.7: Summary of the results of RQ1

object, TIA stands out impressively with an average of 0.22%, significantly lower than the other techniques. In essence, when it comes to detecting the first bug, PTS in Gradle Enterprise outperforms retest-all by 1 percent point, while TIA in Teamscale remarkably surpasses retest-all by 18 percent points. In other words, PTS in Gradle Enterprise is 1.1 times faster than retest-all, whereas TIA in Teamscale is 83 times faster than retest-all. These findings reinforce the superior efficiency of TIA in rapidly identifying code-related bugs.

Now turning our attention to Table 5.11, which centers on configuration-, set up- and architecture checks-related bug commits. In this instance, TIA is marked as "N/A" since it cannot provide answers in Teamscale. The reason behind this limitation lies in the inability to collect testwise coverage data for this type of files. Consequently, Teamscale was unable to select any failing tests related to those bugs, leading to the absence of "time to find the first bug" entries in the table. As for PTS, two cells in the table are also filled with "N/A". This is because PTS did not predict any bugs for those two specific commits, as further detailed in RQ2. The results for PTS reveal a lower average percentage compared to retest-all, with PTS averaging 3.5% and Retest-all averaging 5%. In practical terms, this indicates that by utilizing Gradle Enterprise, the first bug can be detected 1.4 times faster than by executing the complete test suite.

The plots in Figure 5.7 serve as compelling confirmation of the statements previously made.

Table 5.10: Answers of **1.RQ** for **JabRef** using commits containing **code**-related bugs: Time needed to find the first bug (time to first failing test in percent of retest-all run time).

Commit-ID	Retest-all	PTS	TIA
5448d36bd3ae1a370e900fa3ae19785878518a07	6.2%	6.2%	2.8%
fd851ace3ca9f4580643bc62f3aaafe82048a8ac	3.7%	3.7%	0.0002%
ed0915ab5e76bf0a81894db13d252a70c5c5b10d	32%	32%	0.02%
9f61a2a09375d261cd4202e1a4c9927b85d30097	29.3%	28%	0.0017%
ea58a783170a2fbf4aab6134dd54cc1df18656bd	37%	37%	0.00011%
5d0bb904d9466298abe3e688a2c35a23f9e72775	28.5%	28.5%	0.022%
7c344f42052b0be54d206803132278442e260b0e	35.8%	35.8%	0.002%
38ec96a8e63b3f79b332e37a570b119267523872	28.9%	28.9%	0.00034%
bdc9d454573d7bd4e0d4342b42d4f2b2b0b83db6	7.2%	6.5%	0.00011%
b829b83cb10588e9d62b6930df83f81d7405e2ae	8.6%	4.2%	0.0005%
6204ce1ecb9867989d5d9a0b36cdcf778b7441e9	8.3%	3.6%	0.0001%
c889560038f0e318c2a1c4ad500100351867c96b	7.1%	5.7%	0.035%
329f3d40c88650c86971fe6bed2482490afb38cf	3.7%	3.7%	0.0014%

Table 5.11: Answers of **1.RQ** for **JabRef** using commits containing only **configuration-, set up- or architecture checks-**related bugs (no code-related bugs): Time needed to find the first bug (time to first failing test in percent of retest-all run time).

Commit-ID	Retest-all	PTS	TIA
8a29d6e3c2b902b3ff9b9fbeee228c04ed308d24	6%	2.7%	N/A
8b3d26de318a5054690872d2eb8a6b7da51106b4	7%	2.6%	N/A
b575aae17f1646dabff036587c0ab035bef12fa3	6.4%	2.8%	N/A
99d2bf2739621b2677ba0a08280f0b059bd452ac	3%	3%	N/A
abccbdfbb0e9131cfa79a1173f56f3a4c2611528	2.8%	2.8%	N/A
037f747c2dd993c1e1703f5c3975f2ff5ddaabb0	3.5%	3.5%	N/A
522be490dbe5529cc82fbd335a0558a0fc72e75f	3.4%	N/A	N/A
736980f7ae19328e271d1dcf2ccf541f2cce9bcf	3.9%	3.9%	N/A
0b9d629e52f2ec3c42964af350cff582aa8f4330	97%	N/A%	N/A
4e2336a610b91ed4bce72f49ff3eacc2d2674a7b	6.9%	3.8%	N/A
9d9ad85f1f9131e3c00a24f2f26ca961f7eb2fe5	5.7%	5.7%	N/A
49f8b3825043a1360c273f9a2aa21f2d5e0675d3	10.6%	10.6%	N/A

RQ2: How does the number of correctly selected failing tests vary across different test selection techniques, namely Test Impact Analysis (TIA) and Predictive Test Selection (PTS)?
For code-related bug commits (Table 5.12), both TIA and PTS show an outstanding performance, achieving a 100% selection rate for failing tests in the retest-all run for most commits. However, for a few specific commits, TIA achieved a slightly lower percentage, reaching 25% for one commit, 50% for two commits and 66.7% for two others. Nevertheless, overall, in this category, both techniques demonstrate excellent efficiency in selecting the relevant failing tests in JabRef.

On the other hand, when examining configuration, setup, or architecture checks-related bug commits (Table 5.13), we find that TIA was unable to select any failing tests, resulting in a 0% selection rate for failing tests for all commits in the retest-all run, which confirms the limitation for Teamscale, described previously. PTS, while generally effective in code-related bugs, also showed limitations in this case, achieving a 0% selection rate for two commits in this category. Looking closer to those missed failing tests, it turns up that they are architecture and remote setup checks. For the remaining commits, PTS managed to select 100% of the failing tests. This confirms the reliability of the machine learning-based test selection technique in predicting all types of bugs.

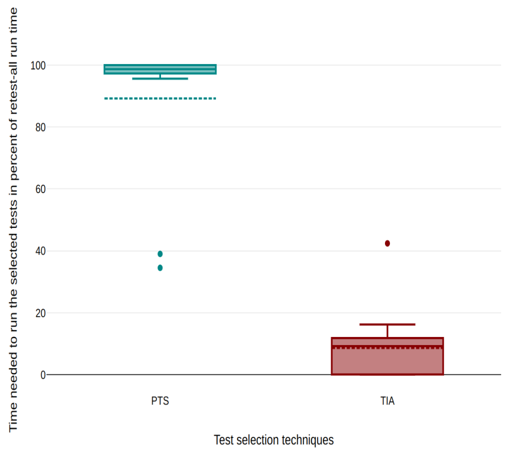
Table 5.12: Answers of **2.RQ** for **JabRef** using commits containing **code**-related bugs: Percentage of correctly selected failing tests.

Commit-ID	#failing tests in retest-all	PTS	TIA
5448d36bd3ae1a370e900fa3ae19785878518a07	6	100%	100%
fd851ace3ca9f4580643bc62f3aaafe82048a8ac	4	100%	50%
ed0915ab5e76bf0a81894db13d252a70c5c5b10d	5	100%	100%
9f61a2a09375d261cd4202e1a4c9927b85d30097	14	100%	100%
ea58a783170a2fbf4aab6134dd54cc1df18656bd	3	100%	100%
5d0bb904d9466298abe3e688a2c35a23f9e72775	2	100%	100%
7c344f42052b0be54d206803132278442e260b0e	4	100%	100%
38ec96a8e63b3f79b332e37a570b119267523872	1	100%	100%
bdc9d454573d7bd4e0d4342b42d4f2b2b0b83db6	51	100%	100%
b829b83cb10588e9d62b6930df83f81d7405e2ae	9	100%	66.7%
6204ce1ecb9867989d5d9a0b36cdcf778b7441e9	9	100%	66.7%
c889560038f0e318c2a1c4ad500100351867c96b	2	100%	50%
329f3d40c88650c86971fe6bed2482490afb38cf	4	100%	25%

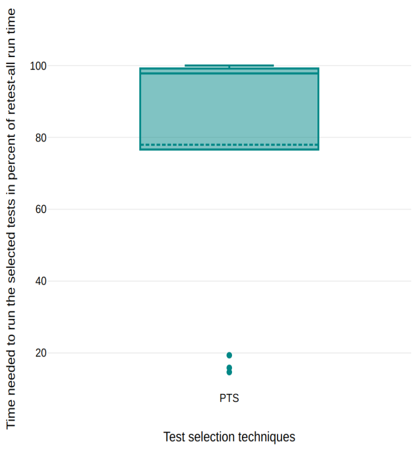
Table 5.13: Answers of **2.RQ** for **JabRef** using commits containing only **configuration-, set up- or architecture checks-**related bugs (no code-related bugs): Percentage of correctly selected failing tests.

Commit-ID	#failing tests in retest-all	PTS	TIA
8a29d6e3c2b902b3ff9b9fbeee228c04ed308d24	1	100%	0%
8b3d26de318a5054690872d2eb8a6b7da51106b4	1	100%	0%
b575aae17f1646dabff036587c0ab035bef12fa3	1	100%	0%
99d2bf2739621b2677ba0a08280f0b059bd452ac	1	100%	0%
abccbdfbb0e9131cfa79a1173f56f3a4c2611528	2	100%	0%
037f747c2dd993c1e1703f5c3975f2ff5ddaabb0	1	100%	0%
522be490dbe5529cc82fbd335a0558a0fc72e75f	1	0%	0%
736980f7ae19328e271d1dcf2ccf541f2cce9bcf	1	100%	0%
0b9d629e52f2ec3c42964af350cff582aa8f4330	1	0%	0%
4e2336a610b91ed4bce72f49ff3eacc2d2674a7b	1	100%	0%
9d9ad85f1f9131e3c00a24f2f26ca961f7eb2fe5	1	100%	0%
49f8b3825043a1360c273f9a2aa21f2d5e0675d3	6	100%	0%

RQ3: How does the time needed to run the selected tests using Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compare to retest-all approach?



(a) Code-related bugs (Table 5.14)



(b) Configuration-related bugs (Table 5.15)

Figure 5.8: Summary of the results of RQ3

The results from Table 5.14 highlight the time savings achieved by TIA and PTS.

Specifically, when examining commits containing code-related bugs, TIA stands out with remarkable time reductions, where some commits require only 0.0011% of the retest-all run time, and on average, it achieves an impressive 8.6% of the retest-all time. On the other hand, PTS also demonstrates efficiency, with a minimum of 39% of the retest-all run time for specific commits, but overall, it achieves a relatively high average of 89.2%. The comparison between TIA and PTS for this type of bugs in the JabRef project indicates that TIA, implemented in Teamscale, outperforms PTS in Gradle Enterprise, as it achieves a significantly higher average of time savings. The plot presented in Figure 5.8a further supports these findings.

In Table 5.15, PTS once more demonstrates effectiveness with regard to commits that exclusively contain configuration-related errors. It achieves a minimum of 14.6%, indicating that only 14.6% of the execution time for retest-all is required to complete the tests chosen by Gradle Enterprise. On the other hand, PTS chose the entire test suite for three commits. Generally speaking, time is saved by about 22% on average. Similar to the last research item, TIA appears to perform very well based on the table's remarkably low numbers. However, it is essential to take into account that TIA finds it difficult to choose the right test cases for this kind of commits. As a result, the answers to this question regarding TIA are again untrustworthy. Achieving such large time savings in this situation does not reflect favorable performance or high efficiency, but rather poor correctness.

RQ4: What is the number of failing tests that test impact analysis (TIA) and predictive test selection (PTS) would execute within only 1% of the time required for retesting all tests?

When analyzing commits containing code-related bugs in Table 5.16, it seems evident that TIA and PTS exhibit contrasting behaviors. Remarkably, TIA showcases exceptional performance for 54% of the examined commits, executing all failing tests within an incredibly short time frame of just 1% of the retest-all run time. For four other commits, TIA achieves an execution rate of over 50% of the failing tests. In general, during the limited time frame, TIA successfully detects the presence of at least one bug in 92% of the cases. These results underscore once again the efficacy of the prioritization technique utilized by TIA. In contrast, within the limited time frame, PTS does not execute any failing tests. So, it does not detect the presence of any bug.

Regarding the second set of commits, as presented in Table 5.17, the analysis primarily centers on the outcomes obtained for PTS, given that TIA was unable to detect any configuration-related bugs. Unfortunately, we find a notable lack of efficiency in PTS when assessing the number of bugs predicted within the limited time frame. Across the 12 analyzed commits, PTS failed to execute any failing test cases, indicating a less than satisfactory performance in this particular scenario.

Table 5.14: Answers of **3.RQ** for **JabRef** using commits containing **code**-related bugs: Time needed to run the selected tests (in percent of retest-all run time).

Commit-ID	PTS	TIA
5448d36bd3ae1a370e900fa3ae19785878518a07	100%	42.4%
fd851ace3ca9f4580643bc62f3aaafe82048a8ac	100%	0.0021%
ed0915ab5e76bf0a81894db13d252a70c5c5b10d	100%	16.2%
9f61a2a09375d261cd4202e1a4c9927b85d30097	98.7%	10.3%
ea58a783170a2fbf4aab6134dd54cc1df18656bd	100%	0.063%
5d0bb904d9466298abe3e688a2c35a23f9e72775	100%	11.8%
7c344f42052b0be54d206803132278442e260b0e	100%	0.082%
38ec96a8e63b3f79b332e37a570b119267523872	97.5%	9.8%
bdc9d454573d7bd4e0d4342b42d4f2b2b0b83db6	97.3%	12%
b829b83cb10588e9d62b6930df83f81d7405e2ae	95.6%	0.03%
6204ce1ecb9867989d5d9a0b36cdcf778b7441e9	34.5%	0.0011%
c889560038f0e318c2a1c4ad500100351867c96b	97.4%	9.2%
329f3d40c88650c86971fe6bed2482490afb38cf	39%	0.0013%

Table 5.15: Answers of **3.RQ** for **JabRef** using commits containing only **configuration-, set up- or architecture checks-**related bugs (no code-related bugs): Time needed to run the selected tests (in percent of retest-all run time).

Commit-ID	PTS	TIA
8a29d6e3c2b902b3ff9b9fbeee228c04ed308d24	96.6%	0%
8b3d26de318a5054690872d2eb8a6b7da51106b4	95.7%	0%
b575aae17f1646dabff036587c0ab035bef12fa3	15.8%	0%
99d2bf2739621b2677ba0a08280f0b059bd452ac	98.8%	11.9%
abccbdfbb0e9131cfa79a1173f56f3a4c2611528	100%	11.4%
037f747c2dd993c1e1703f5c3975f2ff5ddaabb0	98.9%	0%
522be490dbe5529cc82fbd335a0558a0fc72e75f	19.3%	10.3%
736980f7ae19328e271d1dcf2ccf541f2cce9bcf	98.9%	0%
0b9d629e52f2ec3c42964af350cff582aa8f4330	14.6%	0.4%
4e2336a610b91ed4bce72f49ff3eacc2d2674a7b	96.8%	0%
9d9ad85f1f9131e3c00a24f2f26ca961f7eb2fe5	100%	0%
49f8b3825043a1360c273f9a2aa21f2d5e0675d3	100%	0.6%

Table 5.16: Answers of **4.RQ** for **JabRef** using commits containing **code**-related bugs: Percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all.

Commit-ID	PTS	TIA
5448d36bd3ae1a370e900fa3ae19785878518a07	0%	0%
fd851ace3ca9f4580643bc62f3aaafe82048a8ac	0%	50%
ed0915ab5e76bf0a81894db13d252a70c5c5b10d	0%	100%
9f61a2a09375d261cd4202e1a4c9927b85d30097	0%	100%
ea58a783170a2fbf4aab6134dd54cc1df18656bd	0%	100%
5d0bb904d9466298abe3e688a2c35a23f9e72775	0%	100%
7c344f42052b0be54d206803132278442e260b0e	0%	100%
38ec96a8e63b3f79b332e37a570b119267523872	0%	100%
bdc9d454573d7bd4e0d4342b42d4f2b2b0b83db6	0%	100%
b829b83cb10588e9d62b6930df83f81d7405e2ae	0%	66.7%
6204ce1ecb9867989d5d9a0b36cdcf778b7441e9	0%	66.7%
c889560038f0e318c2a1c4ad500100351867c96b	0%	50%
329f3d40c88650c86971fe6bed2482490afb38cf	0%	25%

Table 5.17: Answers of **4.RQ** for **JabRef** using commits containing only **configuration-, set up- or architecture checks-**related bugs (no code-related bugs): Percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all.

Commit-ID	PTS	TIA
8a29d6e3c2b902b3ff9b9fbeee228c04ed308d24	0%	N/A
8b3d26de318a5054690872d2eb8a6b7da51106b4	0%	N/A
b575aae17f1646dabff036587c0ab035bef12fa3	0%	N/A
99d2bf2739621b2677ba0a08280f0b059bd452ac	0%	N/A
abccbdfbb0e9131cfa79a1173f56f3a4c2611528	0%	N/A
037f747c2dd993c1e1703f5c3975f2ff5ddaabb0	0%	N/A
522be490dbe5529cc82fbd335a0558a0fc72e75f	0%	N/A
736980f7ae19328e271d1dcf2ccf541f2cce9bcf	0%	N/A
0b9d629e52f2ec3c42964af350cff582aa8f4330	0%	N/A
4e2336a610b91ed4bce72f49ff3eacc2d2674a7b	0%	N/A
9d9ad85f1f9131e3c00a24f2f26ca961f7eb2fe5	0%	N/A
49f8b3825043a1360c273f9a2aa21f2d5e0675d3	0%	N/A

Results based on mutation testing

RQ1: How much faster is the first error detected using each of the test selection techniques compared to retest-all?
On average, the time required to find the first bug, as a percentage of the retest-all run time, shown in Table 5.18, indicates that PTS performs slightly better than retest-all, achieving average percentages of 33.6% and 34.9%, respectively. Conversely, the TIA technique exhibits exceptional efficiency with an average of 0.002%, significantly lower than the other methods. Explained differently, when it comes to finding the first bug, PTS in Gradle Enterprise outpaces retest-all by 1.3 percent points, while TIA in Teamscale surpasses retest-all by 34.9 percent points. This provides strong evidence that TIA excels in rapidly detecting bugs.

Table 5.18: Answers of **1.RQ** for **JabRef** using mutation testing: Time needed to find the first bug (time to first failing test in percent of retest-all run time).

	Retest-all	PTS	TIA
1.mutation	21.6%	21.6%	0.0051%
2.mutation	82.4%	76,1%	0.0013%
3.mutation	21.8%	21.8%	0.0002%
4.mutation	41.5%	41.5%	0.003%
5.mutation	7.32%	7.32%	0.0003%

RQ2: How does the number of correctly selected failing tests vary across different test selection techniques, namely Test Impact Analysis (TIA) and Predictive Test Selection (PTS)?
According to Table 5.19, both PTS and TIA consistently achieve 100% recall in selecting failing tests.

Table 5.19: Answers of **2.RQ** for **JabRef** using mutation testing: Percentage of correctly selected failing tests.

	#failing tests in retest-all	PTS	TIA
1.mutation	2	100%	100%
2.mutation	8	100%	100%
3.mutation	29	100%	100%
4.mutation	1	100%	100%
5.mutation	51	100%	100%

RQ3: How does the time needed to run the selected tests using Test Impact Analysis (TIA) and Predictive Test Selection (PTS) compare to retest-all approach?
After analyzing Table 5.20, it becomes clear that TIA stands out with remarkable time reductions, as some commits only require less than 0.015% of the retest-all run time, achieving an average of 0.03% of the retest-all time. In contrast, PTS selects the complete test suite in 80% of the cases.

Table 5.20: Answers of **3.RQ** for **JabRef** using mutation testing: Time needed to run the selected tests (in percent of retest-all run time).

	PTS	TIA
1.mutation	100%	0.04%
2.mutation	97.4%	0.013%
3.mutation	100%	0.019%
4.mutation	100%	0.006%
5.mutation	100%	0.068%

RQ4: What is the number of failing tests that test impact analysis (TIA) and predictive test selection (PTS) would execute within only 1% of the time required for retesting all tests?
Table 5.21 showcases that for all the five mutations introduced in the study, TIA achieves 100% efficiency in selecting all the failing tests within this limited time frame. On the other hand, PTS does not execute any failing test, indicating that it did not detect any bug within the specified time constraint.

Table 5.21: Answers of **4.RQ** for **JabRef** using mutation testing: Percentage of failing tests that TIA and PTS would execute in only 1% of the time needed for retest all.

	PTS	TIA
1.mutation	0%	100%
2.mutation	0%	100%
3.mutation	0%	100%
4.mutation	0%	100%
5.mutation	0%	100%

5.6 Discussion

Let us start with discussing the ease of setup and retrieval of necessary calculation input data for Teamscale and Gradle Enterprise. After setting up both tools for my thesis and working with them extensively, I found that the setup process for both tools was relatively straightforward, as there is sufficient documentation available for guidance. However, I did notice some differences in the ease of working with the tools themselves. When it comes to Teamscale, using Test Impact Analysis was more straightforward compared to Gradle Enterprise's Predictive Test Selection. TIA in Teamscale provided a smoother experience with its user interface and was easier to get started with. On the other hand, PTS in Gradle Enterprise required training the machine learning model for 14 days, making it not possible to retrieve necessary data during that time. Additionally, I found that Gradle Enterprise's user interface was slightly more complex, requiring more time to become familiar with its features and functionalities. But, it is worth noting that Gradle Enterprise surpassed Teamscale in terms of providing a more comprehensive range of information and data related to test selection. It not only offered two distinct ways of working with PTS - through "Usage" and "Simulations" - each with its own dedicated dashboard, but it also provided precomputed data such as time savings and the percentage of test failures predicted. In contrast, Teamscale offered only a list of impacted tests, which required manual computation to derive the necessary data.

Moving to assessing the efficiency of the tests selection techniques, we can summarize the strengths and weaknesses of TIA and PTS as follows:
Test Impact Analysis (TIA) proved to be highly efficient in detecting bugs in files where testwise coverage can be collected. It demonstrated accuracy in selecting the correct failing tests, leading to significant time savings. However, TIA's weakness lies in its inability to accurately select tests when the bug is located in a file where testwise coverage cannot be obtained e.g. bugs related to the project configuration.
On the other hand, Predictive Test Selection (PTS) exhibited a remarkable strength by being independent of testwise coverage, making it applicable to all types of failing tests and various bug scenarios. Nevertheless, PTS showed comparatively less significant time savings when compared to TIA.

Considering the strengths and weaknesses of each technique, we could say that the choice between TIA and PTS would largely depend on the specific project requirements.

5.7 Threats to Validity

The case study presents threats to validity.

One significant limitation is the inclusion of only two study objects that use Gradle as a build system, which might limit the generalizability of the findings to other software systems with different build systems, architectures, and technologies.

The selection process of commits for answering the research questions was not entirely random and might not fully represent the entire range of changes, as specific requirements outlined in Subsection 5.3.5 had to be considered. As a consequence, certain commits were excluded from the analysis, while others were included, leading to the fact that certain types of changes or commits with unique characteristics might have been over-represented or under-represented, potentially introducing bias in the results.

Furthermore, the proportion of configuration-related bugs in comparison to code-related bugs may not be fully representative. In this thesis, The number of analyzed commits with configuration-related failures is notably higher than those with code-related failures. This can be attributed to the fact that we just looked at the main/master branch of the study objects' repositories, where configuration changes tend to be more present than code changes. The reason for that, is that in case of an open source project, developers typically handle significant code changes on feature branches, leading to a lower number of commits with code-related bugs on main branch. Added to that, the active GitHub Dependabot in the study objects' repositories automatically performs configuration changes, such as updating dependencies with security risks, resulting in a considerable number of commits with configuration bugs.

Last but not least, it is important to note that Teamscale and Gradle Enterprise, the tools used in this comparative analysis, are subjects to ongoing development and evolution. The Changes and the updates made to them after the study period could impact the relevance of the findings.

Like any study, these limitations have the potential to introduce biases in the results or limit their generalizability.

6 Future Work

The study was conducted with only two study objects to gather data. Expanding this research to other software systems would be valuable to validate the findings. Considering that the study objects share similarities in their build system, exploring projects with different build systems could provide valuable insights.

Another possibility would be to further investigate tools which implement test selection techniques differently. Launchable is a good alternative. In my case, I could not include it in my thesis because of time constraints. A second alternative worth considering is Pareto testing, which is already implemented in Teamscale and merits consideration for future studies.

An other interesting direction for future research would be to develop a hybrid test selection approach that combines the strengths of both Test Impact Analysis and Predictive Test Selection. This approach could utilize code coverage for code changes and PTS for non-code files such as configuration files, potentially leading to a more effective and flexible test selection strategy.

As mentioned in Section 5.7, during the case study, I observed a higher number of commits with configuration changes compared to code changes for the specific study objects and analyzed commits. However, the generalizability of this finding remains uncertain. Therefore, a potential future research question is: What is the typical proportion of configuration changes compared to code changes in software projects overall? To answer this, a future study could incorporate an automated analysis of a large number of commits from different versions and branches, categorizing the changes into code or configuration changes using an automatic classifier. This would provide insights into the distribution of code and configuration changes in real-world software development in general.

7 Conclusion

The aim of this thesis was to evaluate different approaches for test selection. The process involved selecting commercially available tools, choosing open source software systems with automated test suites as study objects and finally conducting a case study to evaluate various aspects, including setup ease, data retrieval, supported scenarios, speedup compared to retest-all, and bug detection compared to retest-all. The choice was met to conduct the comparative analysis on the following techniques: Test Impact Analysis (TIA) implemented in Teamscale and Predictive Test Selection (PTS) implemented in Gradle Enterprise.

The case study on two study objects: JabRef and Rewrite, showed that every tool has its own characteristics, and that each technique has its own strengths and weaknesses. RQ1, RQ3 and RQ4 showed that TIA leads to a shorter time to failure and higher time savings. On the other hand, RQ2 demonstrates that PTS is able to predict any type of bug. In other words, it is able to correctly select failing tests, regardless of the root cause of the failure, including failing configuration tests and architecture and set up checks. However, Teamscale is limited to select test cases for which it is able to collect testwise coverage.

In the end, based on the presented results of the case study and the description of the ease of use of both tools, we can say that there is no good and bad choice. Both techniques could be employed for real world applications. The actual choice depends on the developers' needs and the specific project requirements.

List of Figures

List of Tables

Bibliography

[CQSa] CQSE. *Improving Test Execution Efficiency with Test Impact Analysis*. https://docs.teamscale.com/tutorial/improving-test-efficiency/. [Online; accessed 28-June-2023].

[CQSb] CQSE. *Software Intelligence Vision*. https://docs.teamscale.com/introduction/software-intelligence-vision/. [Online; accessed 26-June-2023].

[CQSc] CQSE. *Teamscale The Software Intelligence Platform*. https://docs.teamscale.com/#why-teamscale-is-different. [Online; accessed 26-June-2023].

[CQSd] CQSE. *TIA Setup*. https://docs.teamscale.com/tutorial/tia-java/#tia-setup. [Online; accessed 22-July-2023].

[DJG17] F. Dreier, E. Juergens, and A. Göb. "Obtaining Coverage per Test Case." Master's Thesis. Technische Universität München, 2017.

[FRC81] K. Fischer, F. Raji, and A. Chruscicki. "A methodology for retesting modified software." In: *Proceedings of the National Telecommunications Conference B-6-3*. 1981, pp. 1–6.

[Graa] Gradle. *Enabling Predictive Test Selection*. https://docs.gradle.com/enterprise/predictive-test-selection/#enabling_predictive_test_selection. [Online; accessed 24-July-2023].

[Grab] Gradle. *Gradle Enterprise Predictive Test Selection User Manual*. https://docs.gradle.com/enterprise/predictive-test-selection/. [Online; accessed 20-June-2023].

[Grac] Gradle. *Gradle Enterprise Solution Overview*. https://gradle.com/gradle-enterprise-solutions/. [Online; accessed 11-July-2023].

[Grad] Gradle. *How does Predictive Test Selection decide which tests are relevant*. https://docs.gradle.com/enterprise/predictive-test-selection/#how_does_predictive_test_selection_decide_which_tests_are_relevant. [Online; accessed 12-June-2023].

[Grae] Gradle. *Observing Predictive Test Selection in Gradle Enterprise*. `https://docs.gradle.com/enterprise/predictive-test-selection/#observing_predictive_test_selection_in_gradle_enterprise`. [Online; accessed 26-July-2023].

[Graf] Gradle. *OSS Projects Revved Up by Gradle Enterprise*. `https://gradle.com/enterprise-customers/oss-projects/`. [Online; accessed 12-June-2023].

[Grag] Gradle. *The predictive model*. `https://docs.gradle.com/enterprise/predictive-test-selection/#the_predictive_model`. [Online; accessed 12-June-2023].

[Grah] Gradle. *The Predictive Test Selection Simulator*. `https://docs.gradle.com/enterprise/predictive-test-selection/#the_predictive_test_selection_simulator`. [Online; accessed 26-July-2023].

[Kas19] A. Kaserbacher. “Empirical Study of the Prioritization of Automated Tests in Information Systems based on Recently Changed Code.” Master’s Thesis. Technische Universität München, 2019.

[Mac+19] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. “Predictive Test Selection.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 91–100. DOI: `10.1109/ICSE-SEIP.2019.00018`.

[Rot19] J. Rott. “Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis.” Master’s Thesis. Technische Universität München, 2019.

[Sil+16] D. Silva, R. Rabelo, M. Campanha, P. S. Neto, P. A. Oliveira, and R. Britto. “A hybrid approach for test case prioritization and selection.” In: *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2016, pp. 4508–4515.

[SKS10] Y. Singh, A. Kaur, and B. Suri. “Test case prioritization using ant colony optimization.” In: *ACM SIGSOFT Software Engineering Notes* 35.4 (2010), pp. 1–7.

[Spi+17] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. “Reinforcement learning for automatic test case prioritization and selection in continuous integration.” In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, pp. 12–22.

[SS11] B. Suri and S. Singhal. “Implementing ant colony optimization for test case selection and prioritization.” In: *International journal on computer science and engineering* 3.5 (2011), pp. 1924–1932.

[Wal+06] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. "Timeaware test suite prioritization." In: *Proceedings of the 2006 international symposium on Software testing and analysis*. 2006, pp. 1–12.

[Won+97] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. "A study of effective regression testing in practice." In: *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE. 1997, pp. 264–274.