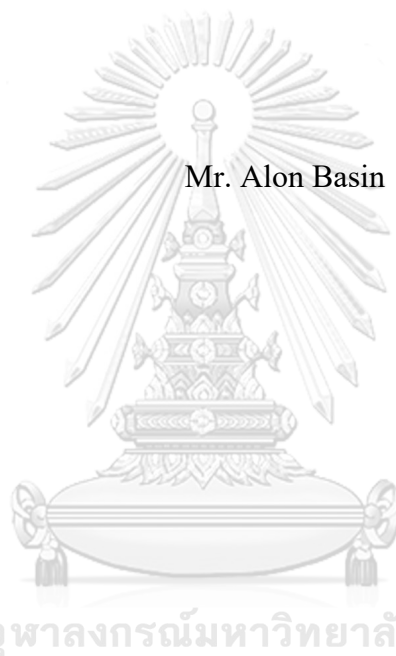


CLASS-LEVEL AND TOKEN-LEVEL APPROACHES FOR TEST IMPACT ANALYSIS



Mr. Alon Basin

A Thesis Proposal Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Computer Science and Information

Technology

Department of Mathematics and Computer Science

FACULTY OF SCIENCE

Chulalongkorn University

Academic Year 2022

Copyright of Chulalongkorn University

วิธีระดับคลาสและระดับโทเค็นสำหรับการวิเคราะห์ผลกระทบจากการทดสอบ



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ

ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2565

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title CLASS-LEVEL AND TOKEN-LEVEL APPROACHES FOR
TEST IMPACT ANALYSIS
By Mr. Alon Basin
Field of Study Computer Science and Information Technology
Thesis Advisor Assistant Professor ARTHORN LUANGSODSAI, Ph.D.
Thesis Co-advisor Associate Professor KRUNG SINAPIROMSARAN, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of
the Requirements for the Master of Science

Polkit Sangvanich

.....
(Professor POLKIT SANGVANICH, Ph.D.)

Dean of the Faculty of Science

THESIS COMMITTEE

ทศพร ปานิตานารักษ์

.....
(THAP PANITANARAK, Ph.D.)

Chairman

Arthur Luangsodsai

.....
(Assistant Professor ARTHORN LUANGSODSAI, Ph.D.)

Thesis Advisor

Krung Sinapiromsaran

.....
(Associate Professor KRUNG SINAPIROMSARAN, Ph.D.)

Thesis Co-Advisor

Peraphon Sophatsathit

.....
(Associate Professor PERAPHON SOPHATSATHIT, Ph.D.)

External Examiner

อลอน บาซิน: วิธีระดับคลาสและระดับโทเค็นสำหรับการวิเคราะห์ผลกระทบจากการทดสอบ

(CLASS-LEVEL AND TOKEN-LEVEL APPROACHES FOR TEST IMPACT

ANALYSIS) อ. ที่ปรึกษาหลัก : ผศ.ดร.อาทร เหลืองสดีใส, อ.ที่ปรึกษาร่วม : รศ.ดร.กรุง

ตินอภิรมย์สรานู

วัตถุประสงค์ของวิทยานิพนธ์คือการสำรวจประสิทธิภาพของการวิเคราะห์ผลกระทบของการทดสอบในบริบทของการทดสอบแบบต่อเนื่อง การทดสอบอัตโนมัติถูกเรียกใช้งานอย่างสม่ำเสมอเพื่อรับรองการคุณภาพของรหัสโปรแกรม ในขณะที่การทดสอบแบบต่อเนื่องสามารถเพิ่มประสิทธิภาพของรหัสและลดภาระการบำรุงรักษาได้ การทดสอบต่อเนื่องยังทำให้เกิดการเพิ่มขึ้นของต้นทุนในขั้นการทดสอบด้วย ในการศึกษาของเราเกี่ยวกับการวิเคราะห์ผลกระทบของการทดสอบ เราได้พัฒนาเทคนิคระดับคลาสที่ใช้ตัวแบ่งส่วนภาษาจาวา เพื่อสร้างกราฟความสัมพันธ์ขึ้นต่อกันระหว่างกรณีทดสอบและคลาสรหัสต้นฉบับโดยใช้ต้นไม้ไวยากรณ์นามธรรม เราประยุกต์เทคนิคนี้กับซอฟต์แวร์ระบบจาวา 7 ระบบ และประเมินผลการทดสอบในสองรูปแบบ การวิจัยพบว่ามีความสัมพันธ์ระหว่างประสิทธิภาพของการวิเคราะห์ผลกระทบของการทดสอบและระดับความสัมพันธ์ระหว่างกรณีทดสอบและคลาสรหัสต้นฉบับ การศึกษาของเรายืนยันว่างานวิจัยก่อนหน้านี้ที่แสดงว่าแม้จะมีการเปลี่ยนแปลงรหัสโปรแกรมเล็กน้อยในการยืนยันรหัสโปรแกรม ผลเฉลี่ยประมาณ 40% ของกรณีทดสอบได้รับผลกระทบสำหรับการทดสอบแบบแรก (ไฟล์ต้นฉบับที่เปลี่ยนแปลง) ในขณะที่ผลเฉลี่ยประมาณ 58% ของกรณีทดสอบได้รับผลกระทบสำหรับการทดสอบแบบที่สอง (โทเค็นคลาสที่เปลี่ยนแปลง) อย่างไรก็ตามเรายืนยันว่าการวิเคราะห์ผลกระทบของการทดสอบเป็นเครื่องมือที่มีประสิทธิภาพในการลดค่าใช้จ่ายอื่นสำหรับการทดสอบรหัสโปรแกรม นอกจากนี้เราได้เปรียบเทียบเทคนิคการวิเคราะห์ผลกระทบของการทดสอบของเรา (ทีไอเอ) กับเทคนิคระดับคลาสปัจจุบันในระบบที่ศึกษาทั้งหมดไป และพบว่าวิธีการของเราสามารถปรับปรุงดีขึ้นได้ประมาณ 13% เมื่อทำการเลือกโค้ดทดสอบที่ได้รับผลกระทบสำหรับการทดสอบแบบปรับเปลี่ยนไฟล์ต้นฉบับ ในขณะที่มีประสิทธิผลต่ำกว่าประมาณ 6.5% ในการทดสอบแบบโทเค็นคลาสที่เปลี่ยนแปลง


จุฬาลงกรณ์มหาวิทยาลัย

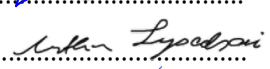
CHULALONGKORN UNIVERSITY


สาขาวิชา วิทยาการคอมพิวเตอร์และ

เทคโนโลยีสารสนเทศ

ปีการศึกษา 2565.

ลายมือชื่อนิสิต..... 

ลายมือชื่อ อ.ที่ปรึกษาหลัก 

ลายมือชื่อ อ.ที่ปรึกษาร่วม 

6278507823 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

KEYWORD: test impact analysis, javaparser, abstract syntax tree, test case selection, continuous testing.

Alon Basin : CLASS-LEVEL AND TOKEN-LEVEL APPROACHES FOR TEST IMPACT ANALYSIS. Advisor: Asst. Prof. ARTHORN LUANGSODSAI, Ph.D. Co-advisor: Assoc. Prof. KRUNG SINAPIROMSARAN, Ph.D.

The objective of the thesis is to examine the efficacy of test impact analysis in a setting of continuous testing, where automated test cases are routinely run to guarantee the integration of high-quality codes. While continuous testing can enhance code quality and minimize maintenance workload, it also leads to a notable rise in overhead for test execution. In our study on test impact analysis, we developed a novel static class level technique that utilizes JavaParser to create a dependency graph between test cases and source code classes based on abstract syntax trees. We applied this technique to seven Java systems and evaluated it against two configurations and discovered a correlation between the effectiveness of test impact analysis and the degree of interdependence between test cases and source code classes. Our study confirmed previous research indicating that even with minimal code changes in commits, roughly 40% of test cases were impacted and required execution on average in the first configuration (changed source files), while approximately 58% of test cases were impacted on average in the second configuration (changed class tokens). Nevertheless, despite these findings, we assert that test impact analysis is an effective tool for reducing test overhead. Moreover, we compared our proposed test impact analysis (TIA) technique with a state-of-the-art static class-level technique on commonly studied systems and observed that our approach achieved an improvement of approximately 13% in selecting impacted tests in the changed source files configuration, while it underperformed by about 6.5% in the changed class tokens configuration.

Field of Study: Computer Science and Information
Technology

Academic Year: 2022

Student's Signature

Advisor's Signature

Co-advisor's Signature

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Assistant Professor Dr. Arthorn Luangsodsai, and co-advisor, Associate Professor Dr. Krung Sinapiromsaran, for their invaluable guidance and support throughout my master's thesis. I am also grateful to the Department of Mathematics and Computer Science at Chulalongkorn University for providing me with all the necessary resources and facilities during my graduate studies.

I would like to extend my thanks to my dissertation committee members, Associate Prof. Dr. Peraphon Sophatsathit and Dr. Thap Panitanarak, for their valuable advice and contributions to the completion of my research.

Finally, I would like to express my heartfelt appreciation to my wife and family for their constant support and motivation throughout my work. Without you, I could not have achieved this.

Mr. Alon Basin



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

TABLE OF CONTENTS

ACKNOWLEDGEMENTS.....	vi
TABLE OF CONTENTS	vii
LIST OF TABLES.....	ix
LIST OF FIGURES	x
CHAPTER 1: INTRODUCTiON	1
1.1 Software Development Life Cycle (SDLC).....	2
1.2 Software Maintenance	4
1.3 Change Impact Analysis	6
1.4 Continuous Integration / Continuous Deployment.....	10
1.5 Continuous Testing.....	12
1.6 Test Impact Analysis.....	14
1.7 Dependency Graph	18
1.8 Objectives and Scope of Work	19
CHAPTER 2: BACKGROUND	21
2.1 Alternative Strategies to Shorten Test Automation	21
2.2 Tests Impact Analysis Techniques	25
2.2.1 Source Code Analysis	25
2.2.2 Predictive Test Selection	27
2.3 Continuous Integration (CI) and Continuous Testing (CT).....	28
2.4 Test Case Prioritization (TCP) and Test Case Selection (TCS).....	29
2.5 Test Case Dependencies and Test Case Quality.....	30
CHAPTER 3: EXPERIMENTAL SETTINGS	33
3.1 Studied Systems	33
3.2 Dependency Graph Detection	36
3.3 Impacted Test Cases.....	37
3.4 Compared Approach.....	42
CHAPTER 4: RESEARCH AND RESULTS.....	50

4.1	Approach	50
4.1.1	Test Impact Analysis - Source File Level	56
4.1.2	Test Impact Analysis - Class Tokens Level	60
4.2	Results	65
4.2.1	Test Impact Analysis - Source File Level	65
4.2.2	Test Impact Analysis - Class Tokens Level	70
CHAPTER 5: FUTURE WORK AND CONCLUSION		73
5.1	Future Work	73
5.2	Conclusion	75
REFERENCES		77
APPENDIX		84
VITA		85



LIST OF TABLES

Table 1 : Code Base Change Impact Analysis Techniques	7
Table 2 : Studied Systems Overview	35
Table 3 : Number of Commits, Mean Changed Files, And Median Percentage of Impacted Test Cases Using Test-to-Classes Dependency Graph (Changed Source Files Configuration)	57
Table 4 : Number of Commits, Mean Changed Class Tokens, And Median Percentage of Impacted Test Cases Using Simplified Test-to-Classes Dependency Graph (Changed Class Tokens Configuration).....	62
Table 5 : Comparison of Median Percentage of Impacted Tests Between Static Class-Level Techniques.....	70

LIST OF FIGURES

Figure 1 : Defects fix cost during software development life cycle	4
Figure 2 : Continuous Integration/Continuous Deployment pipeline illustration	11
Figure 3 : Example for Test Impact Analysis Architecture	17
Figure 4 : Example usage of the tagging approach.....	23
Figure 5 : High level of Facebook's predictive test selection model	28
Figure 6 : Comparison of the studied systems.....	35
Figure 7 : An example of a CompilationUnit in JavaParser	51
Figure 8 : The representation of the Callback interface in Kafka as a CompilationUnit .	52
Figure 9 : End-to-end diagram of source file level and class token level Test Impact Analysis approaches	60
Figure 10 : Distributions by percentage of impacted test cases using test to classes dependency graph (1st configuration - changed source files)	66
Figure 11 : The percentage of classes with the distribution of dependent classes.....	68
Figure 12 : The percentage of test cases with the distribution of dependent classes by Percentage.....	69
Figure 13 : Distributions by percentage of impacted test cases using test to classes dependency graph (2nd configuration - changed class tokens)	72

CHAPTER 1

INTRODUCTION

This chapter provides an overview of the research presented in this thesis and aims to introduce the primary topics and objectives of this work.

The first section of this chapter is dedicated to Software Development Life Cycle (SDLC) overview. SDLC refers to the process or methodology followed by software development teams to design, develop, test, and maintain software systems.

The second section of this chapter is dedicated to software maintenance, which is a critical aspect of software development. It includes various activities such as bug fixing, enhancing existing features, and adding new features to the software.

The third section of this chapter discusses change impact analysis, which is a technique used to identify the potential effects of changes made to the software. It involves analyzing the codebase and determining the areas that might be impacted by a particular change. Change impact analysis is essential for reducing the risk of introducing bugs or other issues into the software.

The fourth section focuses on continuous integration (CI)/continuous deployment (CD), which is a set of practices and tools used to streamline the software development process. It involves automating various tasks such as building, testing, and deploying software changes.

The fifth section introduces the concept of Continuous Testing (CT), which is a software development practice where automated tests are continuously executed throughout the CI pipeline to provide rapid and continuous feedback on the quality of the software. The main goal of continuous testing is to ensure that defects and issues are detected early, which reduces the cost of fixing them and minimizes the risk of delays in software delivery.

The sixth section introduces test impact analysis, which is a technique used to determine the impact of changes made to the software on the test suite. It involves identifying the subset of test cases that need to be executed to verify the changes made to the software.

The seventh section explains the concept of dependency graph in the context of software which typically represents the relationships between code components such as classes, functions, or modules in a codebase. Nodes in the graph signify individual elements, and edges between nodes indicate dependencies. A high-degree dependency graph has many interconnected nodes, while a low-degree dependency graph has fewer connections.

Finally, the last section covers the objectives and scope of this work.

1.1 Software Development Life Cycle (SDLC)

Software Development Life Cycle (SDLC) is a systematic process followed by software development teams to design, develop, test, deploy, and maintain software applications [58]. SDLC provides a structured approach to ensure that software projects are completed efficiently, within budget, and meet the specified requirements and typically consists of the following phases:

- **Analysis:** This initial phase involves gathering and analyzing the requirements of the software application. This includes identifying the needs of the end-users,

understanding business objectives, and defining functional and non-functional requirements.

- **Design:** In this phase, the software architecture and system design are created based on the gathered requirements. It includes defining the software components, modules, and their interactions, as well as specifying the database structure, user interfaces, and overall system architecture.
- **Development:** The implementation phase involves translating the system design into actual software codes. Programmers write codes based on the design specifications using programming languages, frameworks, and tools.
- **Testing:** Once the code is developed, the software undergoes rigorous testing to ensure its quality and reliability. Different testing techniques, such as unit testing, integration testing, system testing, and user acceptance testing, are employed to identify and fix bugs, errors, and usability issues.
- **Deployment:** After successful testing, the software is deployed to the production environment or made available to end-users. This phase involves installation, configuration, and setting up the software in the target environment.
- **Maintenance:** Once the software is deployed, it enters the maintenance phase. During this phase, updates, bug fixes, and enhancements are made based on user feedbacks and changing requirements. Ongoing support and maintenance activities are performed to ensure the software remains functional and up to date.

Software maintenance is the process of making modifications and improvements to software after its initial release, to keep it functional, up-to-date, and compatible with the latest hardware and software environments. It is a crucial aspect of software development that ensures the software continues to meet its intended purpose and delivers value to its users.

1.2 Software Maintenance

Software maintenance over time can be divided into several stages, which typically include the following [57]:

- **Corrective Maintenance:** This involves fixing defects or bugs that have been discovered in the software after its release. Corrective maintenance is usually necessary because the software was not thoroughly tested before release or because new bugs were introduced during the maintenance process.
- **Adaptive Maintenance:** This involves making changes to the software to adapt it to new or changing user requirements, hardware or software environments, or business rules. Adaptive maintenance is necessary because user needs or external factors may change over time, requiring updates to the software.
- **Perfective Maintenance:** This involves making enhancements to the software to improve its functionality, performance, or usability. Perfective maintenance is necessary to ensure that the software remains competitive and delivers the best user experience.
- **Preventive Maintenance:** This involves making changes to the software to prevent problems from occurring in the future. Preventive maintenance can include activities such as optimizing code, refactoring, or updating documentation.

During the various phases of software development, maintenance is commonly recognized as the most demanding, expensive, and labor-intensive task [1], [2]. To meet user demands, software products undergo inherent adjustments and alterations to accommodate evolving system prerequisites. In their work, Lehman and Belady [3] presented a set of laws that delineate the patterns of software evolution. Their first law asserts that a software system must continuously evolve, lest it become increasingly obsolete over time, while their second law posits that the complexity of a software system increases as it evolves, rendering it more challenging to comprehend, modify, and maintain.

According to estimates [48], more than 70% of the overall expenses of software development are incurred during the maintenance phase following the delivery of the software. This cost is strongly linked to the complexity of the software. Furthermore, research has shown that the cost of fixing defects during the maintenance stage far exceeds the combined cost of all other stages in SDLC. Figure 1 illustrates the cost linked with resolving defects at different stages of SDLC.

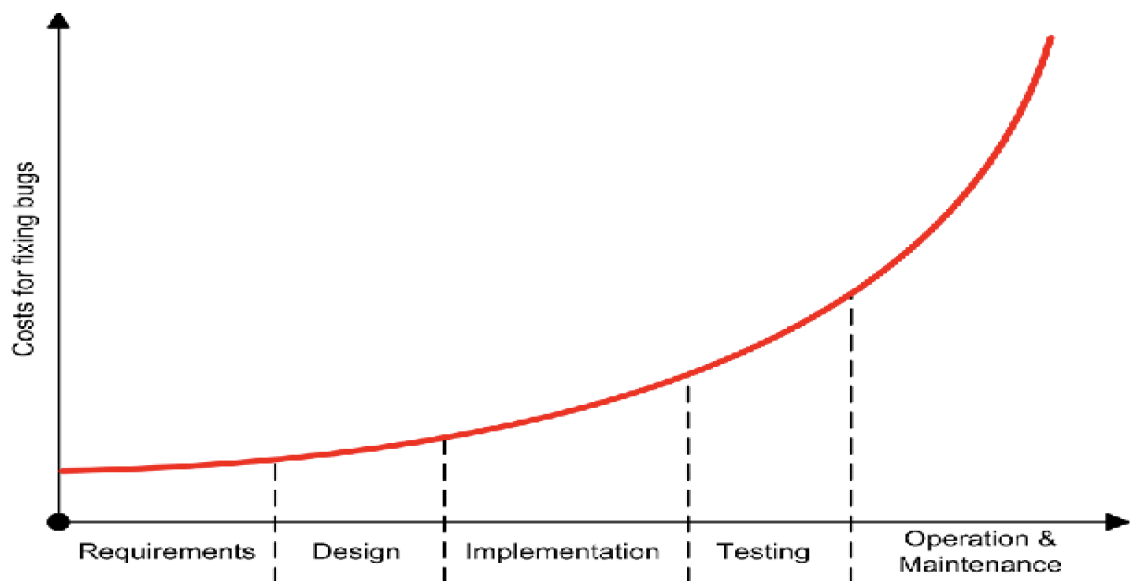


Figure 1. Defects fix cost during software development life cycle (SDLC) [48].

Moreover, the cost of software maintenance can be influenced by both technical and non-technical factors. Non-technical factors include the level of experience in the application area, staff stability, application time, external environment, and user requirements. Technical factors include software complexity, development of human capacity, documentation quality, configuration management technology, modern programming specifications, and database size. Factors such as complicated software structure, poor documentations, and larger database size can result in increased maintenance costs, while factors such as good programming style, effective testing methods, and detailed documentation can help reduce maintenance costs [49].

Software modifications can stem from various factors, such as emerging demands, rectifying defects, addressing change appeals, and more. As adjustments are implemented in the software, unforeseen repercussions are prone to emerge, potentially leading to discrepancies with other elements of the initial software. Below are several instances of different types of modifications that can be applied to software:

- **New requirements:** Adding new features or functionalities to the software [51]. For example, adding a payment gateway to an e-commerce website or adding support for new file formats to a media player.
- **Defect resolutions:** Fixing bugs or errors in the software [52]. For example, fixing a login issue or addressing a crash that occurs when certain actions are performed.
- **Change requests:** Modifying existing features or functionalities [51]. For example, changing the color scheme of a user interface or adjusting the sorting algorithm used for search results.
- **Security updates:** Implementing security fixes to protect against vulnerabilities or exploits [53]. For example, updating the software to address a known security flaw or implementing better encryption methods.
- **Performance improvements:** Optimizing the software for better performance [54]. For example, improving the speed of a search algorithm or optimizing the code to reduce memory usage.
- **Compatibility updates:** Ensuring that the software works with other programs or systems [55]. For example, updating the software to work with a new version of an operating system or adding support for a new database.
- **Regulatory compliance:** Making changes to comply with legal or regulatory requirements [56]. For example, updating the software to comply with GDPR regulations or adding accessibility features to comply with disability laws.

1.3 Change Impact Analysis

Change impact analysis (CIA) is a process that evaluates the potential effects of a proposed change to a software system. It involves identifying and analyzing the

relationships and dependencies between different components of the software system and assessing the potential impact of the change on these components. The goal of change impact analysis is to determine the potential risks, costs, and benefits associated with implementing a change, and to inform decision-making regarding whether to proceed with the change or not. By conducting change impact analysis, software developers can better understand the potential consequences of changes to the software system and make informed decisions about how to proceed with modifications while minimizing the risk of unintended negative impacts on the system.

To assess the potential influence of proposed alterations on various aspects of the software, software change impact analysis (CIA) employs diverse techniques such as dependency analysis, data flow analysis, test coverage analysis, etc [4]. These techniques play a crucial role in software development, maintenance, and regression testing [4]-[7]. CIA can be performed both prior to and after the implementation of a change. Before making modifications, CIA can assist in comprehending the program, forecasting the impact of the alteration, and estimating the associated expenses [4], [5]. Conversely, after implementing changes, CIA can be employed to trace the cascading effects, select appropriate test scenarios, and carry out change propagation [6]-[9]. Bixin Li et al. [45] conducted research on techniques for analyzing the impact of code changes and presented a variety of 23 techniques as listed in Table 1.

Table 1. Code Base Change Impact Analysis Techniques.

Technique	Description
T1	Use object-oriented coupling measurement to identify the impact set
T2	Use the coverage information of the field data collected from users to support dynamic CIA
T3	Provide a technique for dynamic CIA based on whole path profiling
T4	Apply data mining to version histories to extract the co-change coupling between the files for CIA

T5	Use the execute after relation between entities to support dynamic CIA
T6	Use the control call graph to perform static CIA
T7	Use dynamic programming on instrumented traces of different programmed binaries to compute the impact set
T8	Analyze influence mechanisms of scoping function signatures and global variable access to support CIA
T9	Use textual similarity to retrieve past change request in the software repositories for CIA
T10	Perform dependency analysis in object-oriented programs for CIA
T11	Use the measure of dynamic function coupling between two functions for CIA
T12	Create cluster of closely associated software program files in the software repository for CIA
T13	Apply two different data mining algorithms APRIORI and DAR in the software repository for CIA
T14	Analyze change records through singular value decomposition to produce cluster of co-change files for CIA
T15	Use call graph to compute the impact set
T16	Use conceptual coupling measurement for CIA
T17	Use a hierarchical model to interactively compute the impact set
T18	Blend conceptual and evolutionary couplings to support CIA
T19	Use source code comments and changelogs in software repository to support CIA
T20	Use multivariate time series analysis and association rules to perform CIA
T21	Analyze impact mechanisms of different change types for CIA
T22	Use relational topic-based coupling to capture topics in classes and relationships among them for CIA
T23	Use single and multi-labeled machine learning classification for CIA

Here are a few instances demonstrating the utilization of change impact analysis:

- **Planning software changes:** Change impact analysis can help software development teams plan and prepare for changes to a software system by identifying potential risks, dependencies, and impacts associated with the proposed changes.
- **Estimating project scope and cost:** By analyzing the impact of a change on different components of the software system, change impact analysis can help developers estimate the scope and cost of a project more accurately.
- **Prioritizing changes:** Change impact analysis can help prioritize proposed changes based on their potential impact and the resources required to implement them. This can help ensure that changes with the most significant impact are given the higher priority.
- **Identifying potential issues:** Change impact analysis can help identify potential issues that may arise because of a change, such as compatibility issues, conflicts with other software components, or performance problems.
- **Evaluating risks:** Change impact analysis can help evaluate the potential risks associated with a change and develop strategies to mitigate those risks.
- **Streamlining testing:** By identifying the components of the software system that are likely to be impacted by a change, change impact analysis can help streamline the testing process by allowing developers to focus testing efforts on the most critical areas.
- **Enhancing communication:** Change impact analysis can enhance communication among team members and stakeholders by providing a clear understanding of the potential impact of changes and the steps needed to address any issues that may arise.

1.4 Continuous Integration / Continuous Deployment

Continuous Integration/Continuous Deployment (CI/CD) is a software development practice that aims to automate and streamline the process of building, testing, and deploying code changes.

Continuous Integration is the practice of regularly merging code changes from multiple developers into a shared repository. This process is automated, and each merge triggers a build and automated testing process to ensure that the code changes integrate successfully with the existing codebase.

Continuous Deployment is the practice of automatically deploying the successfully built and tested code changes to production servers or environments. This process involves the automation of the deployment pipeline and can include additional testing, such as integration and acceptance testing, before deployment.

CI/CD helps reduce the risk of introducing errors into the production environment, speeds up the software development process, and enables organizations to deliver software updates more frequently and reliably.

Modern software development commonly employs continuous integration/continuous deployment (CI/CD) to facilitate software maintenance. The delivery of a new software version requires a series of tasks referred to as a "pipeline" in the continuous integration and continuous deployment (CI/CD) process. A traditional CI/CD pipeline typically involves the following processes and illustrated in Figure 2 [46]:

- Code changes: Developers make changes to the codebase and push them to the source code repository.
- Build: The CI/CD system automatically checks out the latest code changes, compiles the code, and builds the application.

- **Test:** Automated testing is performed to verify the functionality and quality of the application. This can include unit tests, integration tests, and acceptance tests.
- **Review and feedback:** If issues are found during the testing process, feedback is provided to the developer, who then makes the necessary changes and pushes them back to the repository.
- **Deployment:** Once the code changes pass all the tests and reviews, the CI/CD system automatically deploys the application to a staging or production environment.
- **Monitoring and feedback:** The application is monitored in the production environment to identify any issues or errors, which are then fed back into the CI/CD pipeline to trigger the necessary actions.
- **Notification:** The CI pipeline notifies the team of the build and test results and provides feedback on the quality of the code changes.

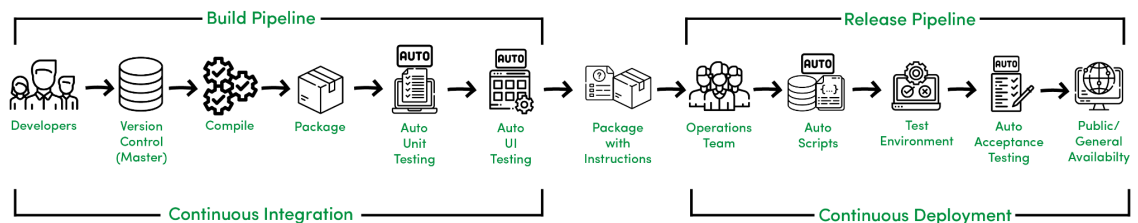


Figure 2. Continuous Integration/Continuous Deployment (CI/CD) pipeline illustration.

By automating these processes, a traditional CI/CD pipeline helps to ensure that code changes are thoroughly tested and validated before being released to production. This reduces the risk of introducing errors or breaking the application and enables faster and more reliable software releases.

In other words, the use of CI/CD pipelines is geared towards enhancing software delivery by automating various stages of the software development life cycle. Automation

of CI/CD across development, testing, production, and monitoring phases allows organizations to develop better-quality code more quickly. While it is feasible to manually execute each step of a CI/CD pipeline, the actual advantage of CI/CD pipelines lies in their automation.

1.5 Continuous Testing

Continuous Testing (CT) is a software testing approach that integrates automated testing throughout the software development lifecycle (SDLC), from planning and development to deployment and maintenance and it is a critical component of the Continuous Integration/Continuous Deployment (CI/CD) process.

In the CI/CD process, developers make changes to the codebase, which are automatically checked out, compiled, built, and tested in the CI pipeline. The CI pipeline integrates CT by continuously executing automated tests to ensure that code changes integrate well with the existing codebase, and the application functions as expected.

By doing so, CT provides real-time feedback on the business risks associated with a software release by continuously running automated tests and providing instant feedback to the development team. This helps identify defects and issues early in SDLC, which reduces the risk of introducing errors into the production environment and improves the quality of the software.

Typically, the Continuous Testing (CT) process involves the following steps:

- Test planning and design: The CT process begins with planning and designing tests that will validate the software requirements.
- Test automation: Once the tests are designed, they are automated and integrated into the CI/CD pipeline.
- Test execution: The automated tests are run continuously throughout SDLC, from development to production.

- Test reporting and feedback: Test results are reported in real-time to the development team, providing feedback on the quality and stability of the codebase.
- Test maintenance: Tests are updated and maintained to ensure that they remain relevant and effective.

By integrating CT into the CI/CD process, software development teams can build and release software quickly and with confidence, ensuring that software is delivered on time, on budget, and to the required quality standards.

Nevertheless, in most cases, a CI pipeline involves the integration of modified code into a codebase, which may happen several times within a day, with the objective of reducing the burden of software maintenance and providing users with the most up-to-date software version. This involves executing a series of test suites for each integration, thereby making the process time-consuming and resource intensive.

In fact, a challenge faced in modern software development is having too many test cases [44]. While having automated tests is an essential practice in software development, having too many tests can cause problems. One problem with having too many tests is that it can slow down the build and test process, which can impact developer productivity and the speed at which software updates can be released.

Additionally, when developers have too many tests to run, they may opt to skip running them locally on their developer workstation, which can lead to code that has not been thoroughly tested. This can result in errors or defects being introduced into the codebase, which can be costly to fix later in the software development lifecycle.

Another issue with having too many tests is that they can become difficult to manage and maintain. Over time, as the codebase grows and changes, tests may become outdated or redundant, and maintaining them can become challenging.

There are several techniques available to accelerate test execution, such as running tests in parallel across multiple machines and utilizing test doubles. Test doubles are techniques and tools used in software testing to replace real objects or dependencies with simulated ones to isolate the code being tested. This umbrella term encompasses fake objects, stubs, mocks, and spies, each serving a specific purpose in software testing. Using test doubles not only help accelerate test execution, but also enable testing in isolation, ensuring expected behavior, and facilitating testing in challenging scenarios.

The focus of this thesis is on Test Impact Analysis (TIA), a technique that has been proposed in prior research to reduce testing overhead by minimizing the number of test cases that need to be executed [10]-[16].

1.6 Test Impact Analysis

Test Impact Analysis (TIA), a branch of Change Impact Analysis (CIA), is a software testing methodology that aims to optimize the testing process by identifying and running only the relevant tests for the changes made in the codebase. By focusing on the potentially affected parts of the application, TIA helps reduce time and resources spent on testing, while maintaining a high level of test coverage and quality assurance. This contemporary approach accelerates the test automation stage of a build, consequently decreasing testing costs and overhead [44].

Key concepts in Test Impact Analysis are:

- **Code coverage:** This metric measures the extent to which the application's source code is tested. A higher code coverage percentage implies that more of the code has been executed during testing, increasing the likelihood of discovering defects.
- **Change impact analysis:** This process involves analyzing the changes made to a codebase, identifying the potentially affected parts of the application, and determining which tests are required to ensure that those changes do not introduce new defects.

- Regression testing: This type of testing is performed to ensure that existing functionality is not adversely affected by new changes to the code. TIA helps in optimizing regression testing by identifying the most relevant tests to execute based on the changes made.
- Dependency analysis: TIA relies on understanding the dependencies between code components, such as functions, classes, and modules, to determine the impact of changes on other parts of the application. This helps identify which tests need to be executed when changes are made to a particular component.
- Incremental testing: This approach to testing involves running only the tests that are relevant to the code changes, rather than executing the entire test suite. TIA supports incremental testing by narrowing down the set of tests to be executed, reducing the overall testing time.

Benefits of Test Impact Analysis are:

- Reduced testing time: By identifying and running only the relevant tests, TIA can significantly reduce the time it takes to test a software application.
- Resource optimization: TIA helps conserve resources, such as CPU and memory, by avoiding unnecessary test execution.
- Faster feedback: TIA enables faster feedback to developers by quickly identifying potential issues, allowing them to fix defects more rapidly.
- Improved test coverage: With a focused approach, TIA can help maintain high test coverage and quality, even with a reduced number of tests.
- Risk mitigation: By targeting the most relevant tests based on code changes, TIA helps mitigate the risk of introducing defects in the application.
- Continuous integration and delivery: TIA is particularly beneficial in continuous integration and delivery (CI/CD) environments, where rapid feedback and minimized testing time are crucial.

Challenges and limitations are:

- Accurate dependency analysis: Ensuring accurate dependency analysis can be challenging, especially in large and complex codebases.
- Overlooking tests: There is a risk that TIA may overlook some relevant tests, leading to reduced test coverage and potential defects.
- Initial setup and maintenance: Implementing TIA may require additional setup and ongoing maintenance efforts, such as updating dependency information and test mappings.

The fundamental idea is that not every test interacts with each production source file (or the classes/methods derived from that source file). By analyzing the relationship (i.e., dependency graph) between source codes and test cases, it becomes possible to effectively determine a subset of test cases that are connected to the modified code and, as a result, are considered "impacted" [17], [18].

In other words, TIA examines the call graph of the source code, which is a representation that illustrates the flow of function or method calls within a software program and shows the relationships and dependencies between different functions or methods and how they interact with each other during program execution, to identify the relevant tests to run after changes are made to the production code, which results in a reduction of testing overhead by executing only the necessary subset of tests. The approach employed to achieve this goal is known as test impact analysis (TIA) [19]. An example of TIA architecture is shown in Figure 3 [47].

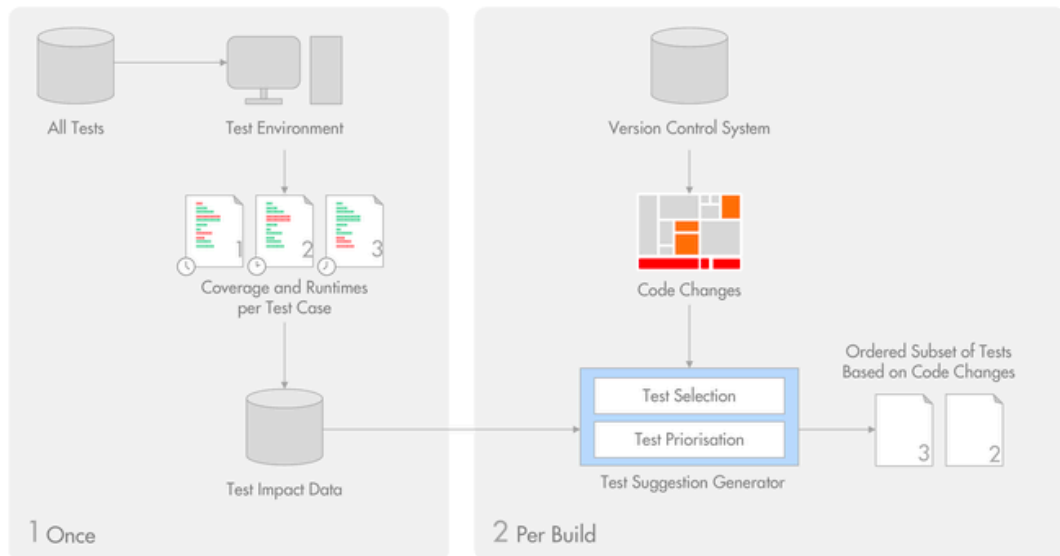


Figure 3. Example of Test Impact Analysis Architecture [47].

TIA can be integrated into the software development workflow, including Continuous Integration/Continuous Deployment (CI/CD) pipelines, to provide real-time feedback to developers on the impact of their code changes on the test suite. This allows developers to focus on fixing defects quickly and efficiently, reducing the time to market for software updates. Microsoft is one company that has integrated Test Impact Analysis (TIA) features directly into their development tools, such as Visual Studio. When these features enabled, they automatically execute impacted unit tests as code is edited [44].

Nevertheless, regular modifications to code and the increasing intricacy of a system can result in a deterioration in the continuous quality assurance of tests. These test scenarios may be interconnected with other origins or test codes, which we refer to as the level of interconnectedness (i.e., dependencies degree). This degree signifies the number of connections (i.e., dependencies) that a test scenario has with the source code.

An extensive amount of code interdependence can lead to an accumulation of testing burdens in CI, particularly when dealing with repetitive code merges and test runs.

Furthermore, heightened dependencies have the potential to undermine the efficacy of test impact analysis and exacerbate the challenges in test maintenance [16].

1.7 Dependency Graph

In the context of software development, a dependency graph typically shows the relationships between different components, classes, functions, or modules within an application or codebase [34]. A node in the graph represents the individual element (such as a class, a function, or a module), while the edge between the nodes represents the dependencies between these elements. A high degree dependency graph will have many interconnected nodes, while a low degree dependency graph will have fewer connections.

For example, let us consider two test cases (TC1 and TC2) and two classes (ClassA and ClassB) in a codebase.

- High degree dependency graph:

In a high dependency graph, both test cases have strong connections to both classes. This could be represented as:

TC1 → ClassA

TC1 → ClassB

TC2 → ClassA

TC2 → ClassB

In this scenario, both test cases depend on both ClassA and ClassB, meaning that any change to either class would require running both test cases to ensure continued correctness. This high dependency makes it difficult to isolate the effects of code changes and increases the number of tests to be executed when changes occur.

- Low degree dependency graph:

In a low dependency graph, each test case is connected to only one of these classes.

This could be represented as:

TC1 → ClassA

TC2 → ClassB

In this scenario, TC1 depends only on ClassA, and TC2 depends only on ClassB. This means that if a change is made to ClassA, only TC1 needs to be executed, while if a change is made to ClassB, only TC2 needs to be executed. This low dependency degree allows for a more efficient testing process, as it reduces the number of tests that need to be run when changes are made to the codebase.

In Test Impact Analysis, the goal is to identify and create low dependency degree graphs, which helps in reducing the number of tests to be executed and optimizes the testing process.

In addition, by analyzing the dependency graph, developers can gain insights into the structure of their codebases, identify areas of high coupling or tight dependencies, and make more informed decisions about refactoring or reorganizing their codes. Dependency graphs can also help in understanding the impact of changes to one part of the system on the rest of the system, as well as in managing and resolving issues related to circular dependencies, which can lead to problems in the build or runtime environments.

1.8 Objectives and Scope of Work

The goal is to improve test impact analysis at the code change level (i.e., commit) by implementing a new static class-level approach. This approach will involve using the abstract syntax tree from program source codes directly to optimize the selection of impacted test cases.

In our case study, we analyzed seven Java-based projects, which consisted of one private project and six open-source projects. We examined all Java files in each of the seven systems to detect the dependency graph among test cases and source codes. To

achieve this, we utilized a test impact analysis tool that employed JavaParser [40], a Java-based open-source library capable of representing a Java source code file as AST and compatible with the most recent Java release.

Our first step involved creating a dependency graph that included both test cases and source code files. This graph captured data on whether dependency existed between a given test case and the source code. To determine if a method was a test case, we checked if it employed testing framework APIs such as JUnit or TestNG (i.e., using the `@Test` annotation). Additionally, we excluded binaries and limited our analysis to dependencies that corresponded to source code files within the system. In other words, we only examined the source code of the subject system and disregarded exterior source codes such as Java API and external Jars.

The upcoming background chapter aims to present an overview of pertinent research studies in software testing, covering alternative strategies for test automation, techniques for impact analysis of tests, continuous integration and continuous testing, test case prioritization and selection, as well as considerations for test case dependencies and quality.

CHAPTER 2

BACKGROUND

Within this chapter, we explore various alternative methods for Test Impact Analysis (TIA) and delve into its different techniques. Moreover, the chapter presents an overview of related work and background in areas such as Continuous Integration (CI) and Continuous Testing (CT), Test Case Prioritization (TCP), Test Case Selection (TCS), Test Case Dependencies and Test Case Quality.

2.1 Alternative Strategies to Shorten Test Automation

To reduce the duration of test automation, various alternative strategies to Test Impact Analysis (TIA) are widely implemented. Here are some instances to illustrate this point:

- **Parallel Test Execution:** Running multiple test cases in parallel can significantly reduce the overall testing time. This can be achieved by setting up a distributed testing environment where tests can run on multiple machines simultaneously.
- **Smarter Test Design:** Creating more efficient and effective tests can reduce the total number of tests required. Using techniques such as risk-based testing or exploratory testing can help identify the most critical areas to focus on.
- **Test Environment Optimization:** A well-configured test environment can help speed up test automation. Preparing the test environment with appropriate settings, data, and dependencies can help avoid delays or issues that can slow down testing.
- **Tagging:** By utilizing tagging, test suites can be partitioned into functional and logical subgroups, and specific subgroups can be selected for execution at different stages of the CI/CD pipeline.

The tagging strategy can assist in categorizing tests based on features, user journeys, testing types, and other criteria. This strategy typically relies on test frameworks such as TestNG and JUnit, as well as build tools such as Maven and Gradle, to execute a specific subset of selected tests through configurable settings. A test framework is a software tool that provides a structured environment and set of functionalities to automate the process of testing software applications. It offers a framework for designing, organizing, and executing tests, making it easier for developers and testers to create and maintain tests effectively. And a build tool is a software tool or utility that automates the process of compiling, packaging, and deploying software applications. It helps streamline the build and release process by managing dependencies, executing predefined tasks, and organizing the overall build workflow.

Nevertheless, one of the primary obstacles in this strategy is the need for manual implementation and maintenance. The approach involves utilizing shared annotations to group subsets of tests based on specific criteria, such as feature, testing layer, etc. The idea is to execute only the relevant subset of tests by providing the required annotation to the build tool as a parameter.

An instance of how the tagging method can be utilized to categorize test cases into distinct logical and functional subsets is depicted in Figure 4 [44]. The figure presented in this example demonstrates the use of tagging to reduce test execution time. Test cases are organized into functional and logical suites, with each suite further divided into different layers such as unit tests, service tests, and functional UI tests for functional separation, and different features of the application such as shopping-cart, inventory, and express for logical separation. This organization enables developers to select and execute only a subset of test cases based on their specific changes, resulting in more efficient testing and quicker feedback in the development process.

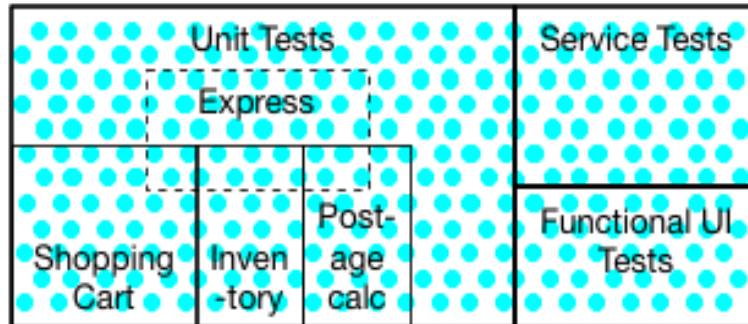


Figure 4. Example usage of the tagging approach [44].

We can elaborate on this concept further by examining the code snippet example presented below, which is based on Figure 4:

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

public class TestSuite {

    @Tag("functional")
    @Tag("shopping-cart")
    @Test
    public void testShoppingCartFunctionality() {
        // test shopping cart functionality
    }

    @Tag("functional")
    @Tag("inventory")
    @Test
    public void testInventoryFunctionality() {
        // test inventory functionality
    }

    @Tag("functional")
    @Tag("express")
    @Test
    public void testExpressFunctionality() {
```

```

    // test express functionality
}

@Tag("logical")
@Tag("unit")
@Test
public void testUnit() {
    // test unit level functionality
}

@Tag("logical")
@Tag("service")
@Test
public void testService() {
    // test service level functionality
}

@Tag("logical")
@Tag("ui")
@Test
public void testUI() {
    // test functional UI level functionality
}
}

```

In this example, we have defined different test cases with specific tags that indicate the logical and functional subsets they belong to. We can run specific subsets of tests by using the `@Tag` annotation to filter tests based on the tags assigned to them. For example, we can run only the functional tests related to the shopping-cart feature by executing the following Gradle [22] command:

```
./gradlew test --tests "TestSuite --tags functional --tags shopping-cart"
```

Which will execute only the `testShoppingCartFunctionality()` test case. Similarly, we can run only the logical tests related to the service layer by running:

```
./gradlew test --tests "TestSuite --tags logical --tags service"
```

This will execute only the `testService()` test case. In this example, Gradle is used as the build automation tool to run the tests. By executing Gradle commands, we can filter and run specific subsets of tests based on the tags assigned to them using the `@Tag` annotation. Gradle is a build automation tool for software development that supports multiple programming languages and platforms. It uses a Groovy-based domain-specific language to define build scripts, which can be customized and extended. Gradle also provides features like dependency management, caching, and parallel execution to improve build performance.

2.2 Tests Impact Analysis Techniques

Test Impact Analysis (TIA) techniques are methods used to determine the subset of tests that are relevant to the changes made in the codebase, thereby optimizing the testing process. In general, these methods can be classified into two primary categories [50]. The first is based on source code analysis, while the second is based on Artificial Intelligence/Machine Learning (AI/ML) models.

2.2.1 Source Code Analysis

In software testing, source code analysis is often utilized in impact analysis approaches to construct dependency graphs between codes and tests.

Some typical TIA techniques based on source code analysis include:

- **Dependency analysis:** This method analyzes the dependencies between different modules, classes, or methods within the source code. By understanding these dependencies, it is possible to identify the tests that are impacted by a particular code modification.

- Call graph analysis: This technique examines the call graph of the source code to determine the relationships between functions or methods. It helps identify the relevant tests to run after making changes to the production code.
- Static analysis: This method involves analyzing the source code without executing it. Static analysis can help to identify potential issues, such as syntax errors or security vulnerabilities, which may require additional testing.
- Dynamic analysis: In contrast to static analysis, dynamic analysis involves executing the code to observe its behavior during runtime. This method can help to identify issues, such as memory leaks or performance bottlenecks, that may not be detectable through static analysis alone.
- Risk-based analysis: This technique prioritizes tests based on the perceived risk associated with each code change. High-risk changes, such as those affecting critical functionality, may require more thorough testing, whereas low-risk changes may require fewer tests to be executed.
- Code coverage analysis: This technique involves identifying the parts of the code that are executed by each test case. By analyzing code coverage, it is possible to determine which tests are affected by a specific code change.

Paul Hammant, in his article "The Rise of Test Impact Analysis" [44], discusses how code coverage or instrumentation can be used during test runs to establish a map between tests and production code. The concept is to execute a single test, observe which code is exercised, and then add these references to a map. This process is repeated for all tests in the suite. Once the map is constructed, a small program runs every time a developer pushes code that examines modified files and determines which tests need to be executed. This list is then passed to the test runner for execution. Additionally, the program periodically performs map updates to maintain an accurate dependency graph as code changes.

However, the code coverage-based approach presents a challenge as the data can rapidly become very large. Furthermore, it is difficult to demonstrate reliably that a change in one area will not impact the code execution path in another area. Just because a test previously did not exercise certain parts of the code does not necessarily mean that it will remain the same in subsequent runs.

2.2.2 Predictive Test Selection

Predictive Test Selection is an approach in software testing that utilizes machine learning algorithms to predict which tests are most likely to fail based on past testing data. The technique involves analyzing historical test data, identifying patterns and trends, and then using this information to predict which tests are most likely to fail in future testing cycles.

This approach can help reduce the time and effort required for testing by identifying the most critical test cases to run, rather than running the entire suite of tests. Predictive test selection can also help improve the overall quality of testing by identifying tests that are more likely to catch defects and issues, which can help ensure that critical bugs are caught and fixed in a timely manner.

Traditionally, most TIA techniques have relied on source code analysis to construct dependency graphs between code and tests. However, some companies use newer approaches such as the predictive test selection. Google [25] and Facebook [37] have developed systems that utilize both machine learning and code analysis. The high-level architecture of Facebook's predictive test selection model is illustrated in Figure 5 [37].

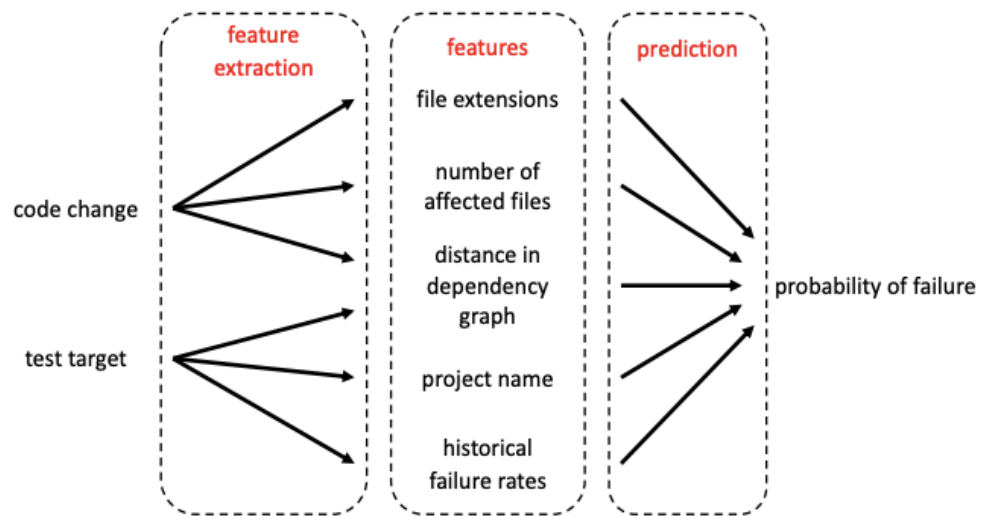


Figure 5. High level of Facebook's predictive test selection model [37].

2.3 Continuous Integration (CI) and Continuous Testing (CT)

The process of (continuous integration) CI entails the frequent merging of code modifications conducted by software developers into a centralized source code management system. Automation platforms like GitLab [20] are widely used to automatically incorporate, compile, and test the code, thereby streamlining the CI workflow. These platforms can trigger the CI process based on a personalized timetable that includes the most recent code changes.

Continuous testing (CT) plays a vital role in the CI workflow as it encompasses the automated running of test scenarios to assess the caliber of modified code. According to recent research [21], the adoption of CI enhances efficiency in software development and aids in identifying a higher quantity of defects. Developers often rely on diverse build utilities like Gradle [22] to handle test execution. With each build execution, Gradle

automatically runs all test scenarios specified within the build scope and generates relevant reports.

The objective of CT is to enable the prompt identification of regression failures and reduce development inefficiencies [23]. Another study emphasized the benefits of employing CT to detect system errors resulting from erroneous data. Their research showcased the effectiveness of continuous data testing in addressing significant data debugging challenges [24]. Google conducted a comprehensive examination outlining their strategies for expanding CT, which encompassed the management of 2 billion lines of code and the daily execution of 150 million tests [25].

In a recent study, researchers examined the effectiveness of test impact analysis (TIA) within a continuous testing environment and evaluated the impact of code interdependence on TIA performance. The study revealed that numerous test scenarios are designed to assess the overall functionality of the system being tested (i.e., integration tests), which inherently involve a higher degree of dependencies that can potentially impede the efficacy of TIA [16].

2.4 Test Case Prioritization (TCP) and Test Case Selection (TCS)

A multitude of research studies have been conducted on test case prioritization (TCP) and test case selection (TCS) techniques.

Test Case Prioritization (TCP) is a technique used to order test cases based on specific criteria, such as the likelihood of detecting faults, the criticality of the functionality, or the potential impact of a code change. By prioritizing test cases, developers can focus on executing the most relevant tests first, which helps to identify and fix defects more quickly. Several studies, including [26]-[33], have investigated this approach.

On the other hand, Test Case Selection (TCS) is the process of identifying a subset of test cases from the entire test suite that are relevant to a particular code change. The goal of TCS is to reduce the number of tests that need to be executed while still maintaining high levels of test coverage and quality assurance. Techniques such as code coverage analysis, dependency analysis, and risk-based analysis can be employed to achieve effective TCS. Multiple studies, including [10]-[15], have explored this method. Researchers have suggested the implementation of TCP and TCS to improve continuous testing efficiency. Legunsen et al. [18] compared class-level and method-level static TCS techniques on 985 revisions of 22 Java systems, and class-level static TCS techniques exhibited promising results.

Elbaum et al. [34] combines both TCP and TCS techniques to make continuous testing more cost-effective. Memon et al. [25] proposed an approach that utilizes test breakages or fixes to improve test case prioritization and decrease developer waiting time for feedback from continuous testing following a commit. Luo et al. [35] evaluated four static TCP techniques against a dynamic-based approach and found static techniques to be highly effective in terms of fault detection and cost reduction.

Engström et al. [36] classified 28 TCS techniques based on properties such as software language and granularity and found no technique to be superior to others. Zhu et al. [37] suggested re-prioritizing test cases based on historical failures. And companies such as Microsoft [19] and Facebook [38] use predictive test selection and test impact analysis to execute only impacted test cases, minimizing test execution overhead and scaling continuous testing for large codebases.

2.5 Test Case Dependencies and Test Case Quality

Understanding the dependencies between test cases and the codebase is essential for optimizing the testing process. Analyzing test case dependencies can help identify

redundant tests or tests that may have been missed. Ensuring high-quality test cases – that is, those that are effective in detecting defects and validating the functionality – is crucial for maintaining the overall quality of the software.

Several research studies analyze the interdependencies of test scenarios from various perspectives. In an examination of 58 Java systems, Luo et al. [35] compared four static TCP techniques with a state-of-the-art dynamic-based approach. Their findings indicated that static techniques could exhibit exceptional efficiency in terms of fault detection and cost reduction.

In an alternative study, Gambi et al. [38] presented a structured and data-oriented technique named PRADET for detecting test dependencies. In separate research, the researchers examined the evolution of test scenarios and uncovered that software developers frequently restructured and repaired test cases. The study unveiled that developers commonly employ mocking to replicate external dependencies, such as web services, and the maintenance of mock codes can lead to heightened overhead [39].

Test case quality, particularly regarding flaky tests, is another area that is currently receiving attention from researchers. As per the findings of Palomba et al. [41], more than 50% of unstable tests exhibit indicators of suboptimal test practices, and addressing these issues can result in improved software design and reduced test instability. In their empirical study, Vahabzadeh et al. [42] explored defects within test code and identified that instability, semantic flaws, and environment-related issues are commonly encountered problems. Luo et al. [43] identified the causes of unstable tests, highlighting that test execution order, concurrent operations, and asynchronous waits are the main factors contributing to test flakiness.

The next chapter, Experimental Settings, delves into the details of the studied systems, the methodology for constructing dependency graphs, and the way we define impacted test cases.



CHAPTER 3

EXPERIMENTAL SETTINGS

In this chapter, we explore our experimental settings and describe the systems being investigated, the methodology for detecting and constructing dependency graphs, and the guidelines for identifying affected test cases.

3.1 Studied Systems

Our study was conducted on seven Java projects and extends earlier research endeavors by focusing on TIA. The research investigates the effectiveness of a new TIA method that at the core maps test cases to classes by extracting the data from ASTs and compares it with the previous static class-level test impact techniques.

Based on prior research, our research presents a new technique for performing static test impact analysis at the class level and assesses its efficacy by evaluating it with state-of-the-art techniques. Our findings suggest that while the extent of code modifications in each change (i.e., commit) is typically limited, a considerable proportion of test cases are still impacted.

The studied systems consist of six open-source projects and one private project. Table 2 provides an overview of these systems and some insights into the relative size of the codebase and the test code. The table displays the subsequent columns:

- System contains name of the software system or project.
- Source files record the number of source files in the system. These are the primary files containing the code that makes up the system's functionality and include only the Java files.

- Test files record the number of test files in the system. These files contain the test code that verifies the functionality and correctness of the system. Same as the source files, we count only the Java files.
- LoC in source is the number of lines of code (LoC) in the source files. This metric provides a rough estimate of the system's size and complexity.
- LoC in test is the number of lines of code in the test files. This metric gives an indication of the testing effort put into the system.
- Source methods record the number of methods (or functions) in the source code. This metric provides insight into the modular structure of the code. A higher number of methods might indicate that the code is more modular, with small, focused methods handling specific tasks, which can make the code easier to understand and maintain.
- Test methods record the number of test cases. This metric shows the granularity of the testing effort, as each test method typically targets a specific functionality or behavior of the system. A higher number of test cases might suggest a more thorough testing effort, which can help uncover potential issues and improve the system's overall quality.

Figure 6 provides a visual representation of their comparative sizes. The projects encompass various domains, such as databases, distributed computing, cloud computing, and web services. To carry out our analysis, we thoroughly investigate all Java files within the seven systems. These projects were selected primarily due to their adherence to continuous testing methodologies, ongoing maintenance efforts, and previous examination in related research studies [16]. The studied systems strictly adhere to continuous integration (CI) practices and utilizes Jenkins, TravisCI, or GitLab as automation tools for code integration and test execution [20].

Table 2. Studied Systems Overview.

System	Source files	Test files	LoC in source	LoC in test	Source methods	Test methods
bookkeeper	1656	656	221359	139972	6828	3171
cucumber-jvm	449	358	25475	29368	2658	1257
hbase	2569	2234	633376	439158	22851	6734
hive	1167	658	215637	1665185	8911	3798
kafka	2422	1339	345014	343064	19558	10195
zookeeper	519	400	88304	71124	3880	1398
api-clients	870	53	52611	6203	400	303

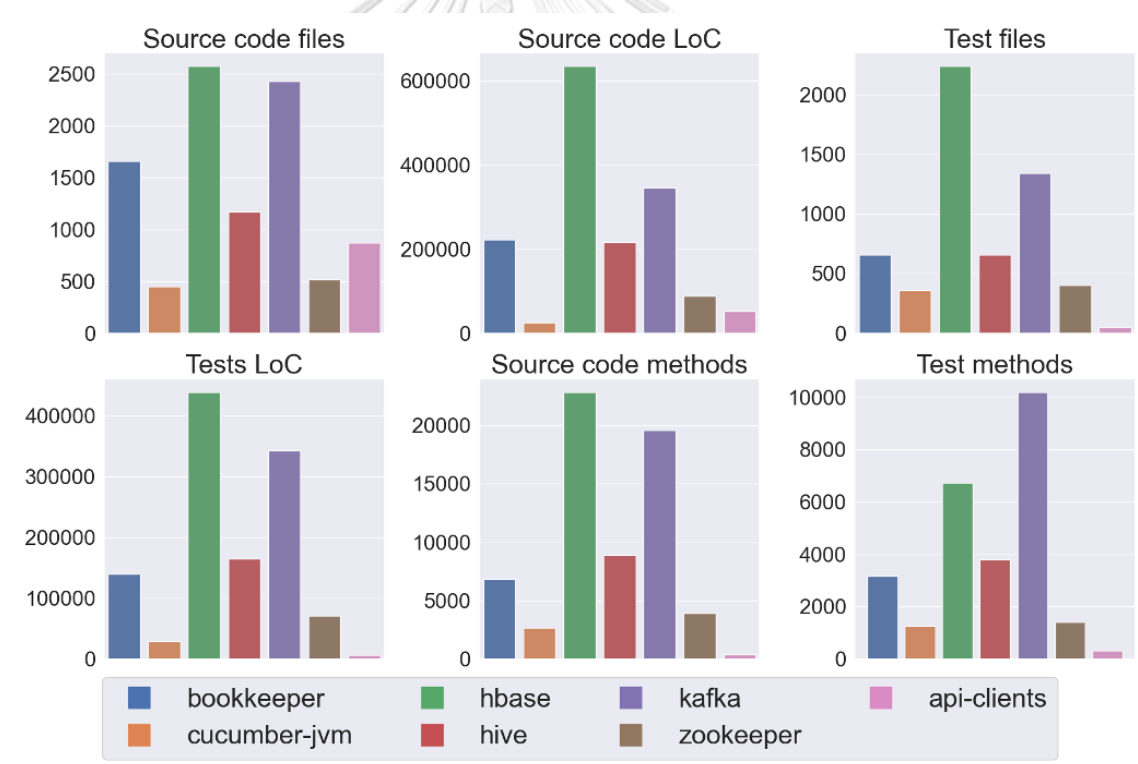


Figure 6. Comparison of the studied systems.

3.2 Dependency Graph Detection

To identify the dependency graph between test cases and source code in the examined systems, we employ JavaParser [41], an open-source Java library designed for static analysis. JavaParser is compatible with the latest Java version and allows for the representation of Java source code files as Abstract Syntax Trees (ASTs). Our objective is to create a tests-to-classes dependency graph that encompasses both test cases and source code files. This graph contains information regarding the presence or absence of dependencies between specific test cases and the corresponding source code. As an illustration, consider the "testModifyValue" test case example taken from the Kafka project:

```
@Test
public void testModifyValue() {
    SnapshotRegistry registry = new SnapshotRegistry(new LogContext());
    TimelineObject<String> object = new TimelineObject<>(registry, "default");
    assertEquals("default", object.get());
    assertEquals("default", object.get(Long.MAX_VALUE));
    object.set("1");
    object.set("2");
    assertEquals("2", object.get());
    assertEquals("2", object.get(Long.MAX_VALUE));
}
```

This test case resulted in a test-to-classes dependency graph as follows:

```
org.apache.kafka.timeline.TimelineObjectTest#testModifyValue:
[org.apache.kafka.timeline.Snapshot, org.apache.kafka.timeline.TimelineObject,
org.apache.kafka.common.utils.LogContext, org.apache.kafka.timeline.Revertable,
org.apache.kafka.timeline.Delta, org.apache.kafka.timeline.SnapshotRegistry]
```

This indicates that the "testModifyValue" test case is dependent on six classes, namely Snapshot, TimelineObject, LogContext, Revertable, Delta, and SnapshotRegistry, from the Kafka codebase. In the approach section under chapter 4, we will provide a

detailed explanation of the steps involved in creating this tests-to-classes dependency graph.

To determine whether a method is a test case, we check if it employs testing framework APIs like JUnit or TestNG (i.e., utilizing the `@Test` annotation). For instance, in Kafka, the method 'testNoPort' is classified as a test case because it uses JUnit's `@Test` annotation, as shown below:

```
@Test
public void testNoPort() {
    assertThrows(ConfigException.class, () -> checkWithoutLookup("127.0.0.1"));
}
```

Furthermore, we exclude binaries and restrict our analysis to dependencies that correspond to source code files within the system (i.e., we solely examine the subject system source code while disregard exterior source codes such as Java API and external Jars).

3.3 Impacted Test Cases

In line with previous studies [10], [11], [17], [18], we define a test scenario as impacted by a specific code modification (i.e., change in commit) if it necessitates execution due to its reliance on the modified code. Prior research [8] have shown that static techniques for test impact analysis are equally effective when compared to dynamic-based techniques.

Hence, our research presents a new technique for test impact analysis, which can be considered as static coverage analysis performed at the class level. In our evaluation, we compare the efficacy of this technique with existing state-of-the-art approach. Our proposed approach for TIA differs from the technique examined in previous studies [16], [17] in two primary aspects.

First, we construct a tests-to-class dependency graph that establishes connections between test cases and the classes they depend on, instead of associating test files with dependent classes. This ensures that each test case includes precise information regarding the relevant dependent classes. Considering that a test file can contain multiple test cases with varying dependent classes, a code modification may solely impact specific test cases within the file. Meaning that in our approach, we identify the subset of impacted test cases by creating a dependency graph that directly links each test case to its corresponding dependent classes, rather than assuming that all test cases within a file are impacted.

Secondly, we acquire pertinent data on dependent classes by examining distinct node types during the construction of the dependency graph. This approach enhances the accuracy and inclusiveness of the dependency graph linking test cases with their respective dependent classes. To do so, we extract specific node types (such as `ClassOrInterfaceType`, `FieldAccessExpr`, and `MethodCallExpr`), resolve them to get classes qualified names (i.e., the complete name of a class, including the package name and the class name itself), and filter only the dependent classes that belong to the project scope (i.e., part of the project's source code). We then recursively examine the usage of these classes for each test case, enabling us to filter out irrelevant information and focus on the pertinent dependencies.

The `ClassOrInterface` node type is mainly used to locate dependent classes by finding all corresponding nodes and filtering out those that are not within the project scope, meaning their resolved name does not start with the base project package. For instance, consider the following `LazyDownConversionRecords` constructor from the Kafka project:


```

package org.apache.kafka.common.record;

import ...

public class LazyDownConversionRecords implements BaseRecords {
    ...

    public LazyDownConversionRecords(TopicPartition topicPartition, Records
records, byte toMagic, long firstOffset, Time time) {
        this.topicPartition = Objects.requireNonNull(topicPartition);
        this.records = Objects.requireNonNull(records);
        this.toMagic = toMagic;
        this.firstOffset = firstOffset;
        this.time = Objects.requireNonNull(time);
        ...
    }
    ...
}

```

The class-to-classes dependency graph for the `LazyDownConversionRecords` class includes the following `ClassOrInterface` nodes that were located and resolved from the `LazyDownConversionRecords` constructor:

```

org.apache.kafka.common.record.LazyDownConversionRecords:
[org.apache.kafka.common.record.Records,
org.apache.kafka.common.TopicPartition,
org.apache.kafka.common.utils.Time, ...]

```

However, the 'Objects' `ClassOrInterface` node was filtered out and not included in the dependency graph since it belongs to the `java.util` package and its resolved name, `java.util.Objects`, does not match the base project package criteria, which in the case of Kafka is 'org.apache.kafka'.

In some cases, the `ClassOrInterface` node may not identify all dependent classes, so additional node types are included. To locate dependent classes that are accessed from static methods, we use the `MethodCallExpr` node type by locating and filtering static methods. For instance, in the `LazyDownConversionRecords` class mentioned previously, we identify `ConvertedRecords` as a dependent class by locating and resolving the `downConvert` static method's class name as shown below:

```
package org.apache.kafka.common.record;

import ...

public class LazyDownConversionRecords implements BaseRecords {
    ...
    private class Iterator extends AbstractIterator<ConvertedRecords<?>> {
        ...
        @Override
        protected ConvertedRecords makeNext() {
            ...
            while (batchIterator.hasNext()) {
                ...
                ConvertedRecords convertedRecords = RecordsUtil.downConvert(batches,
toMagic, firstOffset, time);
                ...
            }
            return allDone();
        }
    }
}
```

Therefore, we also include it in the class-to-classes dependency graph as follows:

```
org.apache.kafka.common.record.LazyDownConversionRecords:
[org.apache.kafka.common.record.RecordsUtil, ...]
```

To locate dependent classes that are accessed via their static fields or enum constants, we use the `FieldAccessExpr` node type to locate and resolve corresponding

nodes. Then, we include only nodes that are part of the project scope and are of type `ResolvedEnumConstantDeclaration` or `ResolvedFieldDeclaration`. If the node is of type `ResolvedFieldDeclaration`, we additionally filter only the static field declarations. For example, in the following `AlterPartitionReassignmentsResponse` class:

```
package org.apache.kafka.common.requests;

import ...

public class AlterPartitionReassignmentsResponse extends AbstractResponse {
    ...
    public
    AlterPartitionReassignmentsResponse(AlterPartitionReassignmentsResponseData
    data) {
        super(ApiKeys.ALTER_PARTITION_REASSIGNMENTS);
        this.data = data;
    }
    ...
}
```

We identified the `ALTER_PARTITION_REASSIGNMENTS` `FieldAccessExpr` node, which was resolved into a `ResolvedEnumConstantDeclaration` type. We then extracted the qualified name of the `ApiKeys` class and included it as a dependent class in the class-to-classes dependency graph as follow:

```
org.apache.kafka.common.requests.AlterPartitionReassignmentsResponse:
[org.apache.kafka.common.protocol.ApiKeys, ...]
```

Similarity, in the following ScramExtensions class:

```
package org.apache.kafka.common.security.scram.internals;

import ...

public class ScramExtensions extends SaslExtensions {
    ...
    public boolean tokenAuthenticated() {
        return
        Boolean.parseBoolean(map().get(ScramLoginModule.TOKEN_AUTH_CONFIG));
    }
}
```

After identifying the `TOKEN_AUTH_CONFIG` `FieldAccessExpr` node, we resolved it into a `ResolvedFieldDeclaration` type. We then verified that this field has static access and extracted the qualified name of the `ScramLoginModule` class. Finally, we included it as a dependent class in the class-to-classes dependency graph as follows:

```
org.apache.kafka.common.security.scram.internals.ScramExtensions:
[org.apache.kafka.common.security.scram.ScramLoginModule, ...]
```

3.4 Compared Approach

To explain in more detail, the compared approach [16], involves constructing a graph that represents the dependencies between test files and the code under test. In this approach, a node in the graph represents a class, and an edge between two nodes represents a dependency between them.

To identify the test cases impacted by code changes, the technique introduces the notions of ancestor and descendant. A node `_A` is considered an ancestor of node `_B` if there exists a path from node `_A` to node `_B`, indicating that node `_A` directly or indirectly invokes node `_B`. Leveraging the dependency graph, the approach initially gathers all the

ancestor nodes of the nodes representing the modified files within a commit, which are referred to as "all_ancestors".

Next, the set of "all_descendants" is derived by merging the descendant nodes from each node within "all_ancestors". Ultimately, the approach identifies only the nodes representing test cases within "all_descendants" as the impacted test cases with dependencies on the modified files. This means that only test files that depend on the changed code, either directly or indirectly, are considered as impacted, while all other test files are ignored.

To put it differently, the compared approach begins by identifying the nodes in the dependency graph that represent the changed source code files. It then proceeds to find all ancestor nodes of these changed files, followed by identifying all their descendant nodes. The next step is to combine all descendant nodes into a single set, which is referred to as "all_descendants". Finally, the approach identifies the nodes in "all_descendants" that represent test cases, as these are the ones that depend on the changed files and are therefore impacted.

For instance, consider a system with four source code files (A, B, C, D) and four test files (T1, T2, T3, T4) with the following dependency graph shows the relationships between them:

```
A → B → C → T1
D → T2
B → T3
```

Suppose a change is made to source code class B. The changed files would be the source code class B, and its ancestor node would be A. The descendant nodes of A are B, C, and T1, while the descendant nodes of C are T1, and the descendant nodes of B are T3. The union of all descendant nodes is {B, C, T1, T3}. The impacted test files are T3 (because

it depends on B) and T1 (because it depends on C). Consequently, the approach determines that test files T1 and T3 are impacted and should be executed to verify that the changes to source code class B do not affect any functionality.

To demonstrate the first contrast in constructing dependency graphs between our approach and the comparative method, we can analyze the `StripedReplicaPlacerTest` file from the Kafka project. Instead of constructing the dependency graphs on the test file level, we do it on the test case level. This test file comprises of 14 test cases as following:

```
package org.apache.kafka.metadata.placement;

import ...

@Timeout(value = 40)
public class StripedReplicaPlacerTest {

    private TopicAssignment place(ReplicaPlacer placer, int startPartition, int
numPartitions, short replicationFactor, List<UsableBroker> brokers) {
        ...
    }

    @Test
    public void testBrokerList() {
        ...
    }

    @Test
    public void testAvoidFencedReplicaIfPossibleOnSingleRack() {
        ...
    }

    @Test
    public void testMultiPartitionTopicPlacementOnSingleUnfencedBroker() {
        ...
    }
}
```

```
@Test
public void testPlacementOnFencedReplicaOnSingleRack() {
    ...
}

@Test
public void testRackListWithMultipleRacks() {
    ...
}

@Test
public void testRackListWithInvalidRacks() {
    ...
}

@Test
public void testAllBrokersFenced() {
    ...
}

@Test
public void testNotEnoughBrokers() {
    ...
}

@Test
public void testNonPositiveReplicationFactor() {
    ...
}

@Test
public void testSuccessfulPlacement() {
    ...
}

@Test
public void testEvenDistribution() {
```

```

    ...
}

@Test
public void testRackListAllBrokersFenced() {
    ...
}

@Test
public void testRackListNotEnoughBrokers() {
    ...
}

@Test
public void testRackListNonPositiveReplicationFactor() {
    ...
}
}

```

The generated dependency graph includes the test file name and 10 depended on classes (i.e., direct and indirect classes that referenced from the test code in the file) as follow:

```

org.apache.kafka.metadata.placement.StripedReplicaPlacerTest:
[org.apache.kafka.metadata.placement.StripedReplicaPlacer,
org.apache.kafka.metadata.placement.UsableBroker,
org.apache.kafka.server.util.MockRandom,
org.apache.kafka.metadata.placement.ReplicaPlacer,
org.apache.kafka.metadata.placement.TopicAssignment,
org.apache.kafka.metadata.placement.StripedReplicaPlacer.BrokerList,
org.apache.kafka.common.errors.InvalidReplicationFactorException,
org.apache.kafka.metadata.placement.PlacementSpec,
org.apache.kafka.metadata.placement.PartitionAssignment,
org.apache.kafka.metadata.placement.StripedReplicaPlacer.RackList]

```

However, in our case, based on the same test file we generated 14 distinct graphs illustrating the dependency between each test case and its associated dependent classes.

For instance, the `testRackListWithInvalidRacks` test case produced the following test-to-classes dependency graph:

```
org.apache.kafka.metadata.placement.StripedReplicaPlacerTest#testRackListWithInvalidRacks:
[org.apache.kafka.metadata.placement.ClusterDescriber,
org.apache.kafka.metadata.placement.UsableBroker,
org.apache.kafka.metadata.placement.TopicAssignment,
org.apache.kafka.server.util.MockRandom,
org.apache.kafka.metadata.placement.PlacementSpec,
org.apache.kafka.common.KafkaException,
org.apache.kafka.metadata.placement.StripedReplicaPlacer.RackList,
org.apache.kafka.common.errors.InvalidReplicationFactorException,
org.apache.kafka.metadata.placement.PartitionAssignment,
org.apache.kafka.metadata.placement.ReplicaPlacer,
org.apache.kafka.common.errors.ApiException]
```

To explain the second difference between our approach and the comparative method in constructing dependency graphs, we can examine the `testMultiPartitionTopicPlacementOnSingleUnfencedBroker` test case from the same `StripedReplicaPlacerTest` test file:

```
@Test
public void testMultiPartitionTopicPlacementOnSingleUnfencedBroker() {
    MockRandom random = new MockRandom();
    StripedReplicaPlacer placer = new StripedReplicaPlacer(random);
    ...
}
```

This test case resulted in the following dependency graph:

```
org.apache.kafka.metadata.placement.StripedReplicaPlacerTest#testMultiPartitionTopicPlacementOnSingleUnfencedBroker:
[org.apache.kafka.metadata.placement.ClusterDescriber,
org.apache.kafka.metadata.placement.TopicAssignment,
org.apache.kafka.metadata.placement.UsableBroker,
```

```
org.apache.kafka.metadata.OptionalStringComparator,
org.apache.kafka.server.util.MockRandom,
org.apache.kafka.metadata.placement.PlacementSpec,
org.apache.kafka.common.KafkaException,
org.apache.kafka.metadata.placement.StripedReplicaPlacer,
org.apache.kafka.common.errors.InvalidReplicationFactorException,
org.apache.kafka.metadata.placement.PartitionAssignment,
org.apache.kafka.metadata.placement.ReplicaPlacer,
org.apache.kafka.common.errors.ApiException]
```

We included the `ClusterDescriber` class in our dependency graph to account for additional node types, such as `MethodCallExpr`, and expose dependencies. This revealed a dependency from the `StripedReplicaPlacer` class to the `ClusterDescriber` class, as it is used as an injected dependency in the “place” method, as shown below:

```
package org.apache.kafka.metadata.placement;

import ...

public class StripedReplicaPlacer implements ReplicaPlacer {
    ...

    @Override
    public TopicAssignment place(PlacementSpec placement, ClusterDescriber cluster)
    throws InvalidReplicationFactorException {
        ...
    }
}
```

Comparable to the approach, our test impact analysis inspects class-level static dependencies among the changed source codes and test cases (i.e., test methods) and identifies impacted test cases if they directly or indirectly invoke the modified code.

Having discussed the methodology and findings in the previous chapter, the upcoming section will provide a comprehensive summary of the research and results obtained from the study.



CHAPTER 4

RESEARCH AND RESULTS

In this chapter, we aim to provide a comprehensive overview of our study, highlighting the key aspects of our research approach, the research question we seek to address, the Test Impact Analysis (TIA) configurations utilized, and the findings derived from our investigation. This chapter serves as a pivotal point, connecting the theoretical underpinnings and practical applications of our research.

4.1 Approach

To establish a dependency graph linking test cases and classes, we commence by compiling a set of all-classes, comprising solely the Java source code files found within the designated source code path. Subsequently, we represent these files as a set of compilation units, extracting the qualified name of each `CompilationUnit`. Within the `JavaParser` framework, `CompilationUnit` serves as a class that portrays the complete Java file as an Abstract Syntax Tree (AST) (refer to Figure 7 for a visual representation).

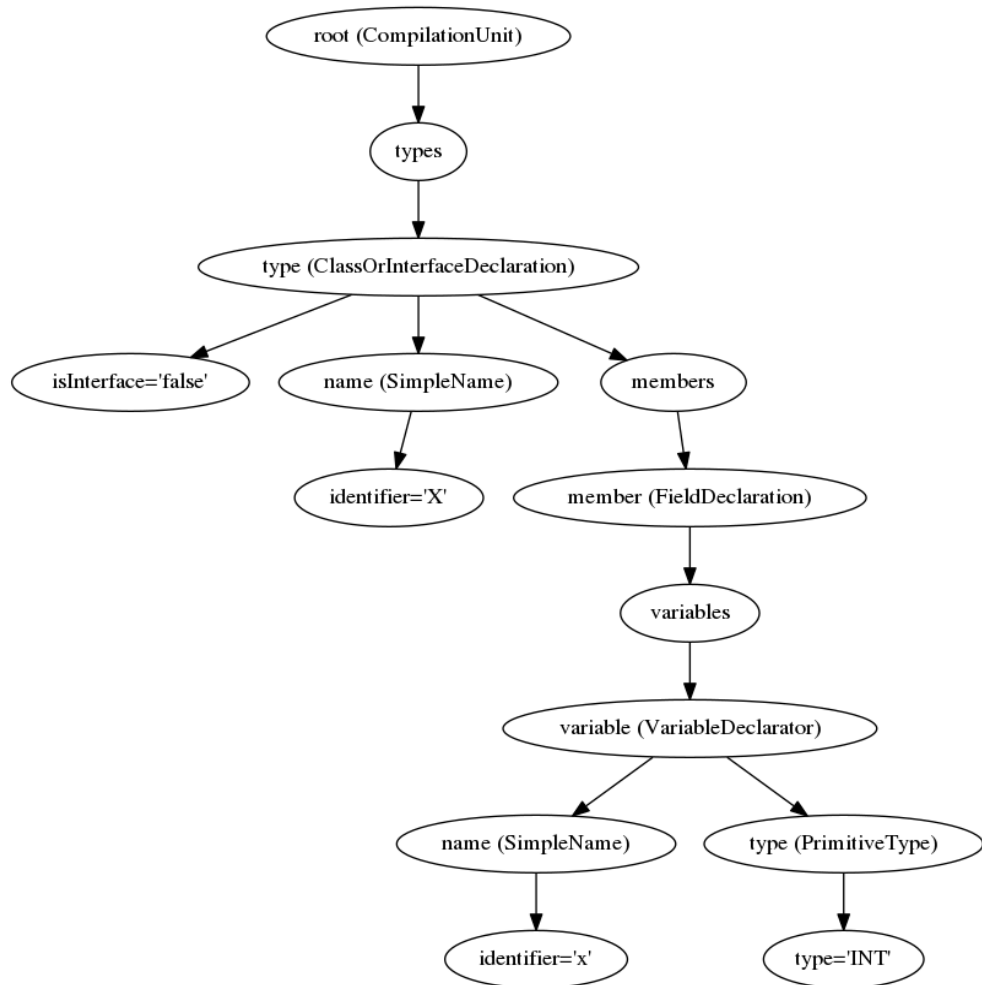


Figure 7. An example of a CompilationUnit instance in JavaParser.

Next, we construct a dependency graph connecting classes with their respective dependent classes by traversing AST and gathering significant node types (e.g., ClassOrInterfaceType, FieldAccessExpr, and MethodCallExpr). By employing the JavaParser JavaSymbolSolver, we resolve each node, obtaining the qualified name of the corresponding class. We then filter out only the nodes that belong to the project scope (i.e., are presented in the all-classes set). The JavaSymbolSolver is a package integrated with JavaParser, enabling the examination of the AST and identification of declarations associated with each element. For example, for the following ‘Callback’ interface in Kafka:

```

package org.apache.kafka.clients.producer;

public interface Callback {
    void onCompletion(RecordMetadata metadata, Exception exception);
}

```

Even though the CompilationUnit that corresponds to this interface contains two ClassOrInterface nodes, namely RecordMetadata and Exception (as seen in Figure 8), we exclude the Exception node from the final class-to-classes dependency graph for the Callback interface since it is not within the project's scope. Therefore, the dependency of the Callback interface will only be with the RecordMetadata class:

```

org.apache.kafka.clients.producer.Callback:
[org.apache.kafka.clients.producer.RecordMetadata]

```

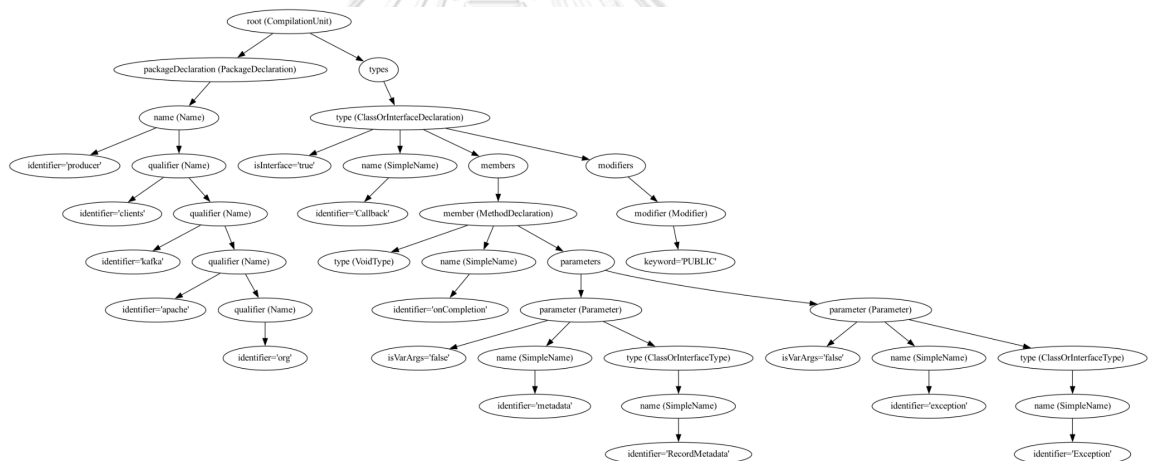


Figure 8. The representation of the Callback interface in Kafka as a CompilationUnit.

Upon completion of this stage, we obtain a dependencies map of class-to-classes. To derive the desired dependency graph between test cases and classes, we begin by aggregating all the test methods (i.e., test cases) by filtering out methods within the designated test path that include the `@Test` annotation. For each test method, we extract the qualified names of the relevant nodes using a similar approach as in the preceding step. Furthermore, considering that test cases often utilize shared resources like test fixture

methods (e.g., `@Before`, `@BeforeClass`), we iterate through the process of extracting class names for all non-test codes (i.e., code excluding methods annotated with the `@Test` annotation) within each test file.

To illustrate this point, we can consider the following `TestRackawareEnsemblePlacementPolicy` test class:

```
package org.apache.bookkeeper.client;

public class TestRackawareEnsemblePlacementPolicy extends TestCase {

    ...

    @Override
    protected void setUp() throws Exception {
        ...
        timer = new HashedWheelTimer(new
ThreadFactoryBuilder().setNameFormat("TestTimer-%d").build(),
            conf.getTimeoutTimerTickDurationMs(), TimeUnit.MILLISECONDS,
            conf.getTimeoutTimerNumTicks());
        ...
    }

    @Test
    public void testInitialize() throws Exception {
        String dnsResolverName = conf.getString(REPP_DNS_RESOLVER_CLASS,
ScriptBasedMapping.class.getName());
        DNSToSwitchMapping dnsResolver =
ReflectionUtils.newInstance(dnsResolverName, DNSToSwitchMapping.class);
        AbstractDNSToSwitchMapping tmp = (AbstractDNSToSwitchMapping)
dnsResolver;
        assertNull(tmp.getBookieAddressResolver());
        dnsResolver.setBookieAddressResolver(repp.bookieAddressResolver);
        assertNotNull(tmp.getBookieAddressResolver());
    }
}
```

```
...
}
```

When considering only the dependent classes in the testInitialize test case only the following classes will be included: ScriptBasedMapping, DNSToSwitchMapping, and AbstractDNSToSwitchMapping. However, because we extract class names from all the non-test methods as well, we also include the HashedWheelTimer class. Therefore, the final test-to-classes dependency graph of the testInitialize test case will also include the class names collected from the setUp method and will be presented as following:

```
org.apache.bookkeeper.client#testInitialize: [ScriptBasedMapping,
DNSToSwitchMapping, AbstractDNSToSwitchMapping, HashedWheelTimer, ...]
```

Finally, we build the test-to-classes dependency graph by iteratively traversing class names and their associated class dependencies for each test case, leading to the formation of the test-to-classes dependency graph. To illustrate this point, we can consider the following simplified example:

```
// ClassC.java
public class ClassC {
    public String getMessage() {
        return "Hello from ClassC";
    }
}

// ClassB.java
public class ClassB {
    private ClassC classC;

    public ClassB() {
        classC = new ClassC();
    }

    public String getMessage() {
        return "Hello from ClassB and " + classC.getMessage();
    }
}
```



```

    }
}

// ClassA.java
public class ClassA {
    private ClassB classB;

    public ClassA() {
        classB = new ClassB();
    }

    public String getMessage() {
        return "Hello from ClassA and " + classB.getMessage();
    }
}

// TestClass1.java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class TestClass1 {
    @Test
    public void testCase1() {
        ClassA classA = new ClassA();
        String expectedMessage = "Hello from ClassA and Hello from ClassB and Hello
from ClassC";
        assertEquals(expectedMessage, classA.getMessage());
    }
}

```

It is evident that while `testCase1` has a direct dependency solely on `ClassA`, the ultimate test-to-classes dependency graph encompasses `ClassB` and `ClassC` as well, due to the dependency between `ClassA` and `ClassB`, and the subsequent dependency between `ClassB` and `ClassC`. Consequently, the final test-to-classes dependency graph for `testCase1` is represented as follows: `testClass1: [ClassA, ClassB, ClassC]`. This approach allows us to

account for testCase1 as impacted if there is a code change in any of the dependent classes, rather than only considering its direct dependency.

To carry out test impact analysis (TIA) based on the test-to-classes dependency graph in the context of code changes, our research involves experimenting with two separate configurations:

- Source file level
- Class token level

4.1.1 Test Impact Analysis - Source File Level

In this configuration we collected the source code files that had changed from commits (see Table 3), filtering only those commits that had made some code changes to at least one Java source file. We then retrieved the corresponding qualified names for each source file in a similar manner to the previous steps (i.e., generation of the test-to-classes dependency graph). Using the test-to-classes dependency graph, we filtered the "impacted" test cases that contained the changed classes, resulting in a subset of "impacted" test cases for a given commit.

Table 3. Number of Commits, Mean Changed Files, and Median Percentage of Impacted Test Cases Using Test-to-Classes Dependency Graph (Changed Source Files Configuration).

System	Commits	Changed files (median)	Percentage of impacted tests (median)
bookkeeper	57	10.912	39.51%
cucumber-jvm	14	1.571	6.96%
hbase	45	3.511	44.85%
hive	24	4.125	26.20%
kafka	52	3.057	22.41%
zookeeper	48	3.312	62.39%
api-clients	67	2.761	43.77%

To accomplish this, we utilize the GitHub and GitLab APIs to obtain commit data and extract the modified Java source code files. For instance, if the system source code is hosted on GitHub, we initially use the following API to retrieve a list of commits:

```
curl -L \
-H "Accept: application/vnd.github+json" \
-H "Authorization: Bearer <YOUR-TOKEN>" \
-H "X-GitHub-API-Version: 2022-11-28" \
https://api.github.com/repos/OWNER/REPO/commits
```

Subsequently, for each commit, we acquire the data for modified source code Java files by making the following API call and providing the commit SHA as a reference:

```
curl -L \
-H "Accept: application/vnd.github+json" \
-H "Authorization: Bearer <YOUR-TOKEN>" \
-H "X-GitHub-API-Version: 2022-11-28" \
https://api.github.com/repos/OWNER/REPO/commits/REF
```

For instance, in the case of the following Kafka commit: 0bb05d8679b684ad8fbb2eb40dfc00066186a75a, we identified three qualified names of modified source code files:

```
[org.apache.kafka.controller.ClusterControlManager,
org.apache.kafka.common.requests.BrokerRegistrationRequest,
org.apache.kafka.metadata.BrokerRegistration]
```

which led to 6557 impacted test cases after filtering from the Kafka test-to-classes dependency graph.

From end-to-end high-level overview (see Figure 9), the source file level configuration can be described as follow:

- Construct test-to-classes dependency graphs:
 - Collect source code files - Gather all the relevant source code files from a given software project.
 - Transform source code files to Abstract Syntax Trees (ASTs) - Convert the source code files into ASTs, which represent the structure of the code in hierarchical manner.
 - Build class-to-classes dependency graph for each AST - Analyze each AST to identify dependent classes.
 - Construct dependency graphs that represent the relationships between root classes and their dependent classes.
 - Collect all test cases - Gather a list of all the test cases from the same given software project.
 - Collect common classes for each test case - Analyze common dependent classes in each test file and add them to the relevant test cases.
 - Build test-to-classes dependency graph for each test case - Using the class-to-classes dependency graphs, construct a graph that represents the

dependencies between the test case and the classes it interacts with directly or indirectly.

- Identify impacted test cases per code change:
 - Collect changed source files from commit - Identify specific source code files that have been modified in a commit.
 - Lookup each changed source file (i.e., changed class) in the test-to-class dependency graph - Examine each changed class and check if any test case depends on it in the test-to-class dependency graph.
 - Add the test case to a set of impacted test cases if a changed class is found - If a dependency between a test case and a modified class is identified, add that test case to a set of impacted test cases.
 - Return the set of impacted test cases per commit - After analyzing all the modified classes and their dependencies, return the set of test cases that are impacted by the code change (i.e., commit).

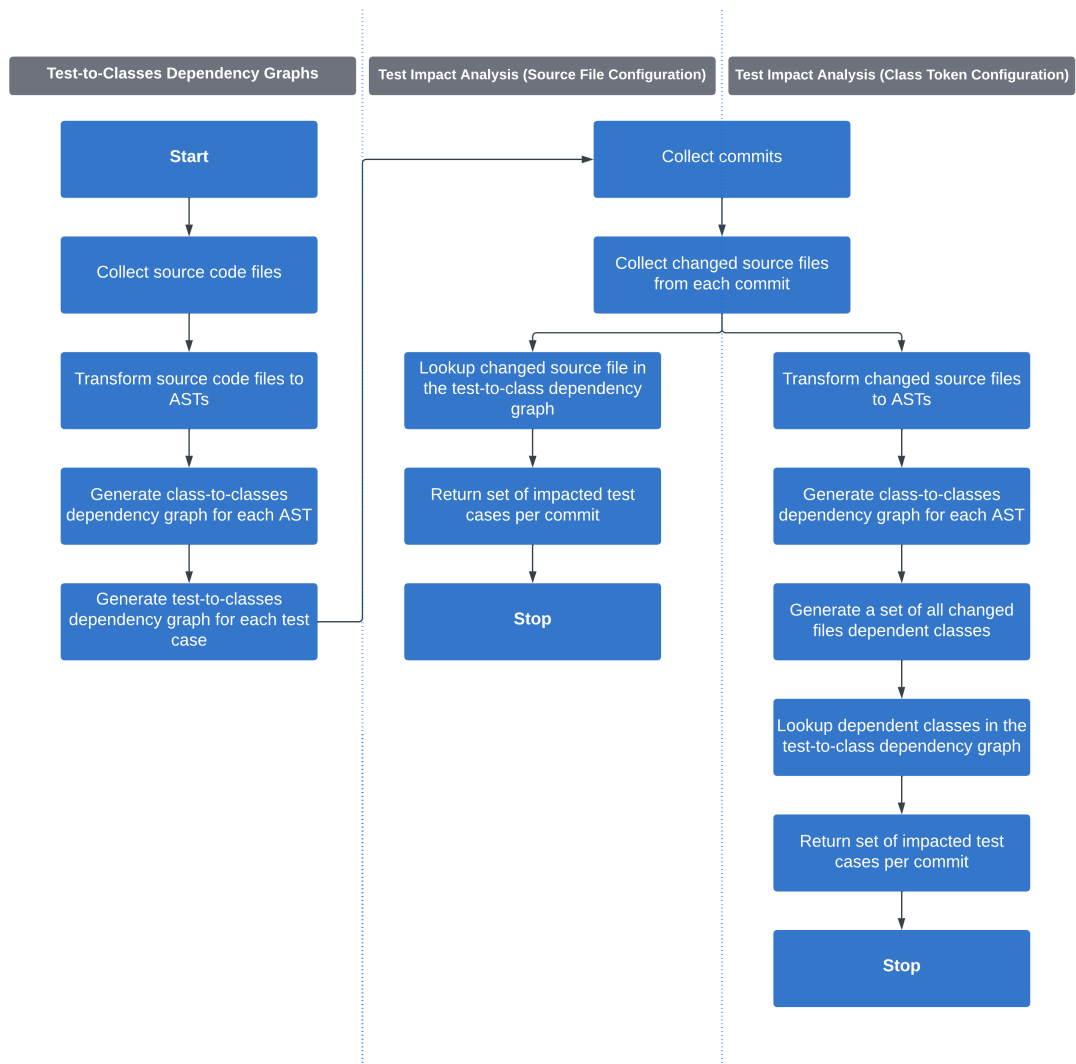


Figure 9. End-to-end diagram of source file level and class token level Test Impact Analysis approaches.

4.1.2 Test Impact Analysis - Class Tokens Level

We employed a configuration like the previous method, where we utilized the same filtered commits that involved modifications to at least one Java source file (as shown in Table 3). However, rather than acquiring the qualified names of each changed source file from the collected commits, we extracted all the modified class tokens within the changed source files. The rationale for this approach was to determine if we could optimize the selection of impacted test cases even further by narrowing the scope of

changes. To achieve this goal, we first simplified the test-to-classes dependency graph by modifying the qualified names to only include the names of the test cases and classes, without their respective packages. Consider the following test-to-classes dependency graph:

```
org.apache.kafka.timeline.TimelineObjectTest#testModifyValue:
[org.apache.kafka.timeline.Snapshot,
org.apache.kafka.timeline.TimelineObject,
org.apache.kafka.common.utils.LogContext, org.apache.kafka.timeline.Revertable,
org.apache.kafka.timeline.Delta,
org.apache.kafka.timeline.SnapshotRegistry]
```

After modification, it becomes:

```
testModifyValue: [Snapshot, TimelineObject, LogContext, Revertable, Delta,
SnapshotRegistry]
```

In the modified version, we removed the package information from the class names to simplify the test-to-classes dependency graph. As a result, only the class names are listed as dependencies for the test case "testModifyValue".

Afterward, we obtained changed lines of code by utilizing the "Git diff" command. Git diff is a command-line tool used in the Git version control system that compares two sets of code changes, typically between different versions of files or directories. The tool provides a summary of the differences between the two sets of codes, highlighting the lines that were added, modified, or deleted.

Subsequently, we processed the collected lines of code with changes, removing comments and imports, dividing the lines into individual tokens, and gathering only those tokens that belong to the project's scope (i.e., present in the all-classes set) (refer to Table 4). In the end, like the previous configuration, we employed the simplified test-to-classes dependency graph to identify the "impacted" test cases containing the altered token classes, resulting in a subset of "impacted" test cases for a given commit.

Table 4. Number of Commits, Mean Changed Class Tokens, And Median Percentage of Impacted Test Cases Using Simplified Test-to-Classes Dependency Graph (Changed Class Tokens Configuration).

System	Commits	Changed class tokens (median)	Percentage of impacted tests (median)
bookkeeper	57	3.142	53.470%
cucumber-jvm	14	5.692	27.850%
hbase	45	6.083	69.362%
hive	24	7.571	28.768%
kafka	52	8.295	63.235%
zookeeper	48	5.320	68.988%
api-clients	67	5.571	65.448%

For example, in the case of the following Kafka commit: 700947aa5a64a707264caac8959fe2dc6b7e7fd0, we identified two qualified names of modified source code files under the first source files configuration: `[org.apache.kafka.coordinator.group.GroupCoordinator, org.apache.kafka.common.requests.OffsetDeleteResponse]`, resulting in 6653 impacted test cases. However, using the class tokens configuration, we identified five class tokens: `[Errors, OffsetDeleteRequest, OffsetDeleteResponse, RequestContext, BufferSupplier]`, leading to 7399 impacted tests.

Although we managed to reduce the scope of changes from 9 classes to 5, which represents a 64.29% improvement when comparing the corresponding class-to-classes dependency graphs:

```
org.apache.kafka.coordinator.group.GroupCoordinator:
[org.apache.kafka.common.requests.RequestContext,
```



```
org.apache.kafka.common.TopicPartition,
org.apache.kafka.common.utils.BufferSupplier,
org.apache.kafka.common.requests.TransactionResult]
```

```
org.apache.kafka.common.requests.OffsetDeleteResponse:
[org.apache.kafka.common.requests.AbstractResponse,
org.apache.kafka.common.requests.OffsetDeleteRequest,
org.apache.kafka.common.protocol.ByteBufferAccessor,
org.apache.kafka.common.protocol.Errors,
org.apache.kafka.common.protocol.ApiKeys]
```

The resulting impacted test cases still increased from 6653 to 7399, representing an 11.21% decrease in the efficiency of the test impact analysis technique (i.e., an increase in selected impacted test cases) when comparing the class tokens configuration with the source files configuration. The rationale behind this will be elaborated further in the subsequent results section.

From end-to-end high-level overview (see Figure 9), the class tokens level configuration can be described as follow:

- Construct test-to-classes dependency graphs:
 - Collect source code files - Gather all the relevant source code files from a given software project.
 - Transform source code files to Abstract Syntax Trees (ASTs) - Convert the source code files into ASTs, which represent the structure of the code in a hierarchical manner.
 - Build class-to-classes dependency graph for each AST - Analyze each AST to identify dependent classes.
 - Construct dependency graphs that represent the relationships between root classes and their dependent classes.
 - Collect all test cases - Gather a list of all the test cases from the same given software project.

- Collect common classes for each test case - Analyze common dependent classes in each test file and add them to the relevant test cases.
- Build test-to-classes dependency graph for each test case - Using the class-to-classes dependency graphs, construct a graph that represents the dependencies between the test case and the classes it interacts with directly or indirectly.
- Identify impacted test cases per code change:
 - Collect changed source files from commit - Identify the specific source code files that have been modified in a commit.
 - Transform changed source code files to ASTs - Convert the changed source code files into ASTs.
 - Build class-to-classes dependency graph for each AST (i.e., changed source file) - Analyze the ASTs of the modified classes, determine their dependencies and construct a graph that represents the relationships between the modified source files and their dependent classes.
 - Build a list of changed source files dependent classes - Create a list that includes all the classes that the modified source files depend on.
 - Lookup each element of the list of changed source files dependent classes in the test-to-class dependency graph - Examine each class in the list of dependent classes, check if any test case depends on it and search in the test-to-class dependency graph for these dependencies.
 - Add the test case to a set of impacted test cases if an element (i.e., dependent class) is found - If a dependency between a test case and a modified class is identified, add that test case to a set of impacted test cases.
 - Return the set of impacted test cases per commit - After analyzing all the modified classes and their dependencies, return the set of test cases that are impacted by the code change (i.e., commit).

4.2 Results

In order to evaluate the effectiveness of our proposed technique, we assess the efficacy of our proposed technique by addressing the following research question (RQ).

RQ: How does our proposed static class-level test impact analysis technique perform compared to the state-of-the-art static class-level test impact analysis technique?

To address this question, we examine the techniques performance in terms of impacted test case selection. This approach allows us to identify any potential advantages or drawbacks associated with our proposed method, as well as to highlight its strengths and weaknesses in relation to the current state-of-the-art technique.

4.2.1 Test Impact Analysis - Source File Level

In most of the assessed systems, the average number of modified files remains below 4. However, the average proportion of impacted test cases can soar up to 62%. Table 3 provides insights into the number of commits, the median quantity of modified files, and the average percentage of impacted test cases for each system. The data reveals that in 5 out of the 7 scrutinized systems, the median count of modified files remains below 4. Considering the overall volume of source code and test files within the examined systems (depicted in Table 3), it becomes evident that, on average, developers modify less than 0.5% of the files daily.

Figure 10 showcases the distribution of impacted test cases based on code changes. The data demonstrates that, on average, the proportion of impacted test cases can soar as high as 62%. Across six of the seven analyzed systems, the average percentage of impacted test cases ranges from 22% to 62%. Notably, our findings indicate that cucumber-jvm exhibits a comparatively lower number of impacted test cases, with a median percentage of merely 6.96%. Further investigation reveals that this disparity can

be attributed to the relatively modest median number of modified files in cucumber-jvm, standing at just 1.5, in contrast to the other systems examined.

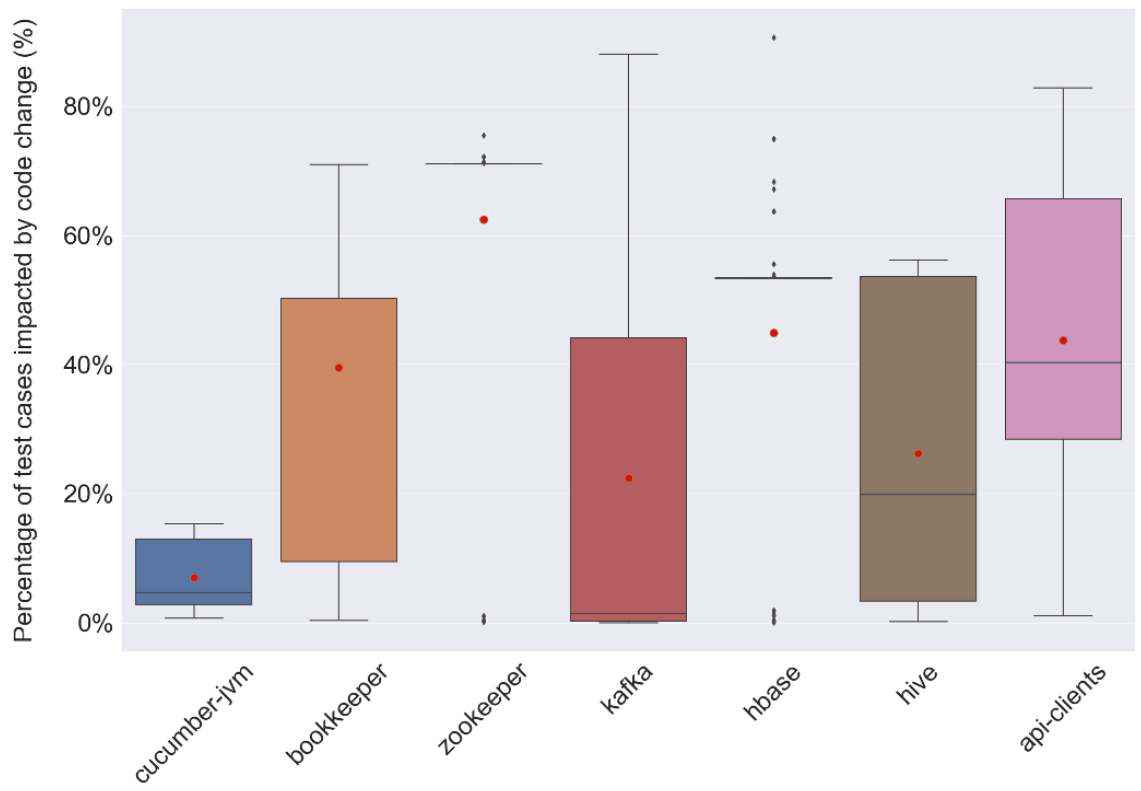


Figure 10. Distributions by percentage of impacted test cases using test to classes dependency graph (1st configuration - changed source files).

To enhance our comprehension of the findings, we undertook an additional examination of the interdependencies between classes (i.e., the number of references each class has to other classes) and the dependencies between test cases and classes (as indicated by the test-to-class dependency graph). Figure 11 illustrates the correlation between the percentage of classes and the distribution of dependent classes within various ranges.

Evidently, within the examined systems, the proportion of classes without any dependencies on other classes exceeds 15%, with the highest recorded value of 51.1%

observed in the api-clients system. Furthermore, our analysis unveiled that a relatively minor fraction of classes (averaging less than 5%) exhibit 20-40 dependencies on other classes.

Figure 12 depicts the association between test cases and their respective dependent classes, represented as a percentage. Across all systems analyzed, a minimum of 20% of test cases exhibit a dependency of less than 10% on the total number of classes within the system. However, in the case of hive, this percentage surpasses 70%.

Based on the findings, we can deduce that there exists a correlation between the percentage of dependent classes and the percentage of impacted test cases. The highest median percentage of impacted test outcomes (as indicated in Table 3), observed in zookeeper and hbase, is linked to the proportion of dependent classes. For instance, in the case of zookeeper, over 75% of test cases rely on 50%-70% of the total classes, while in hbase, more than 50% of test cases have a dependency on 60%-70% of the total classes. These results demonstrate that despite the minimal number of modified files in code modifications within the analyzed systems, the proportion of impacted test cases can be considerably high (with a median of approximately 47% of impacted test cases detected in four out of the seven systems investigated).

Nevertheless, simultaneously, the results also indicate that our static class-level test impact analysis technique holds promise in reducing the testing overhead by an average of approximately 60%. To establish a benchmark against the current state-of-the-art static class-level approach [16], we compared our results on commonly studied systems (as presented in Table 5). Our proposed approach demonstrated an average improvement of approximately 13% (represented by a reduction of roughly 13% in the selection of impacted test cases).

Yet, it is worth highlighting that none of the studies assessed the false negative ratio, which involves evaluating how many test cases influenced the modified code but were disregarded by the test impact analysis technique and can be another future research topic. Also, it should be noted that the median percentage of test results affected by the compared technique were estimated from a graph and thus may be slightly different.

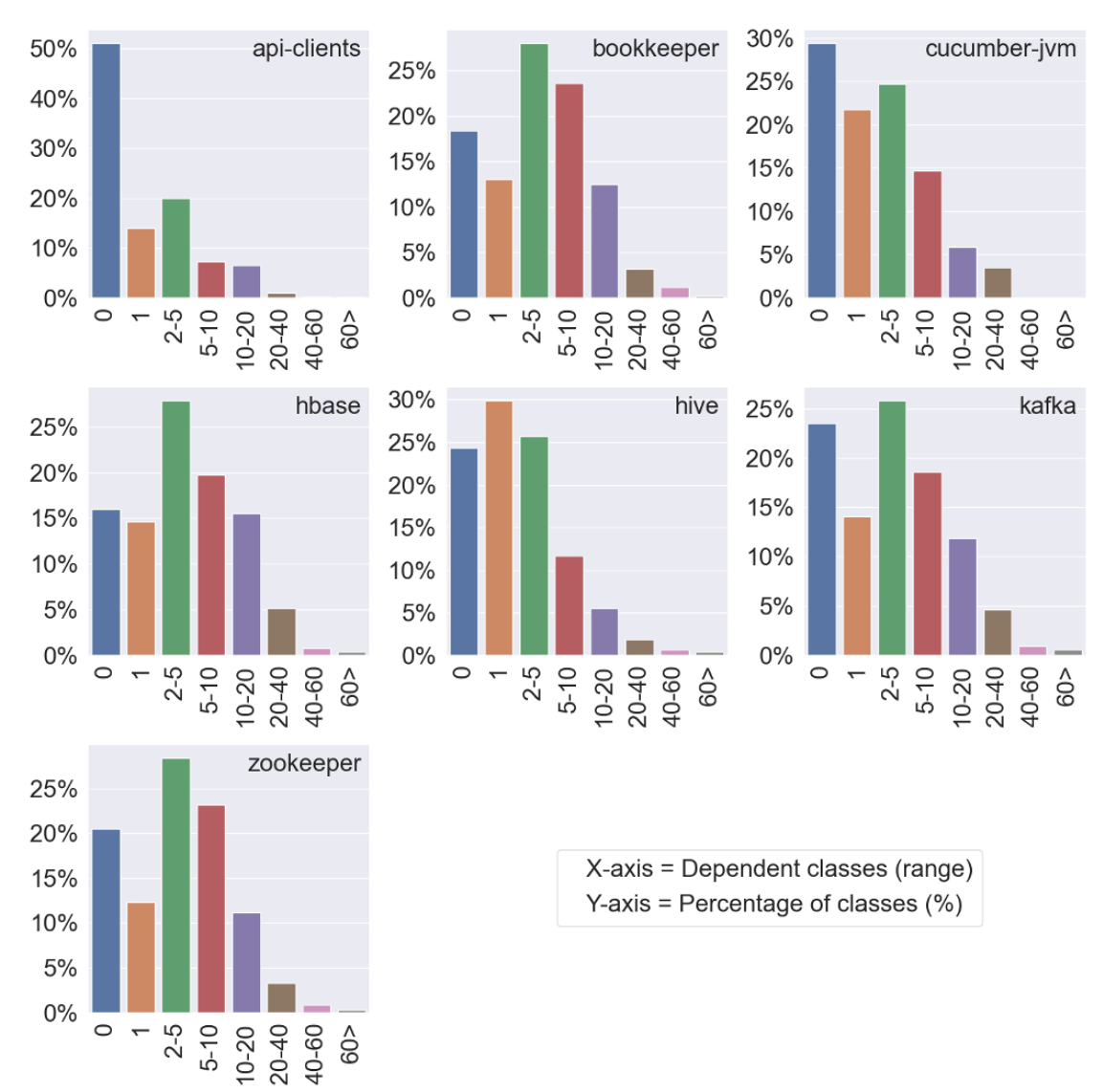


Figure 11. The percentage of classes with the distribution of dependent classes by range.

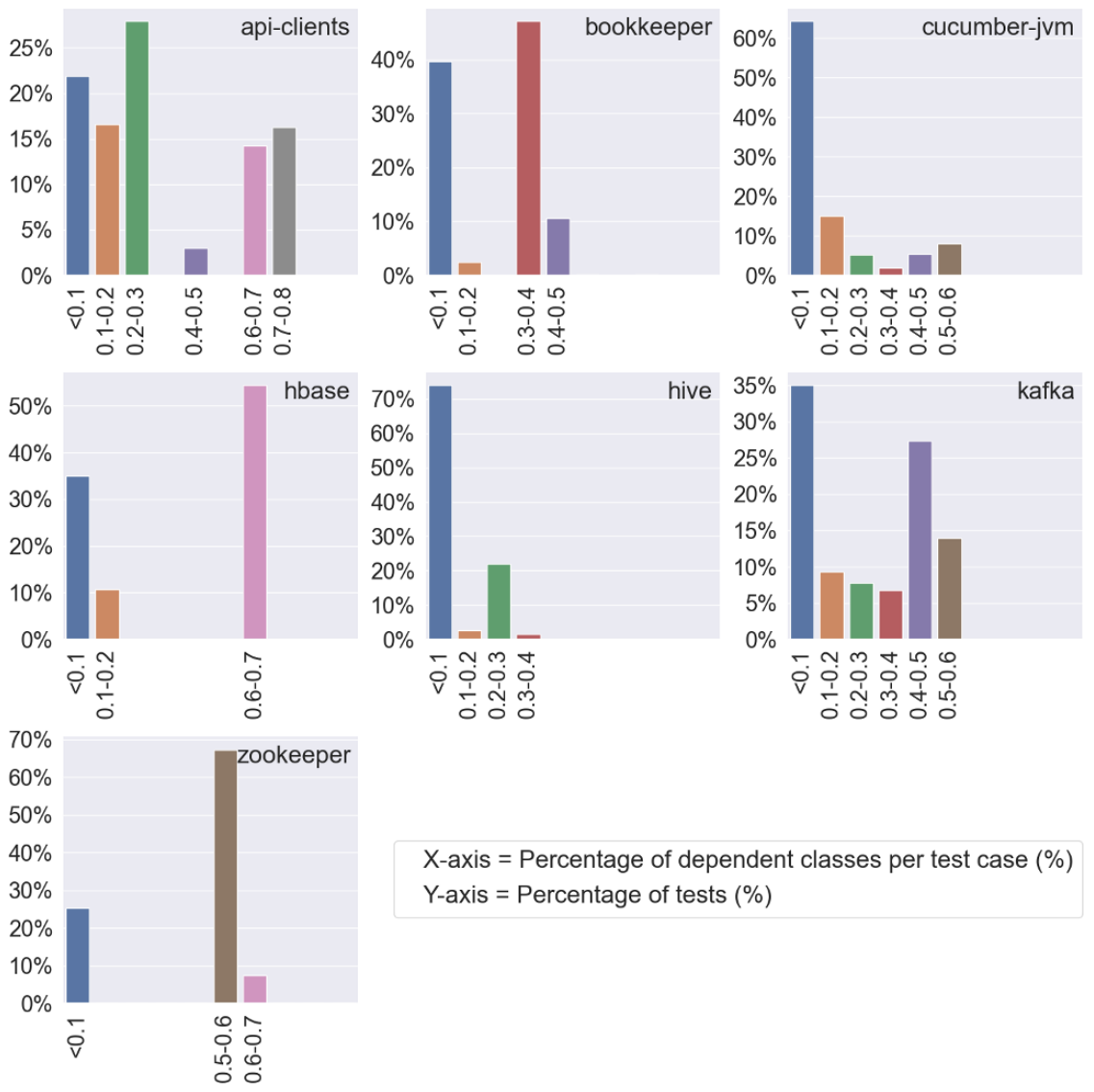


Figure 12. The percentage of test cases with the distribution of dependent classes by percentage.

Table 5. Comparison of Median Percentage of Impacted Tests Between Static Class-Level Techniques.

System	Proposed technique - 1st configuration (changed source files)	Proposed technique - 2nd configuration (changed class tokens)	Percentage of impacted tests (median) - compared state-of-the-art approach
bookkeeper	39.51%	53.47%	~40%
cucumber-jvm	6.96%	27.85%	~15%
hbase	44.85%	69.32%	~72%
hive	26.20%	28.76%	~48%
kafka	22.41%	63.23%	~33%

4.2.2 Test Impact Analysis - Class Tokens Level

In most systems examined, the average number of altered class tokens extracted is between 5 and 8 and the average percentage of impacted test cases can be as high as 69%. Table 4 displays the number of commits, the median number of modified class tokens extracted, and the average percentage of impacted tests for each system. The data reveals that for 6 out of 7 systems analyzed, the median count of class tokens modified is greater than 5.

The distribution of the proportion of impacted test cases resulting from code changes is presented in Figure 13. The results demonstrate that, on average, the percentage of impacted test cases can go up to 69%. In five of the seven systems analyzed, the percentage of impacted test cases ranges from 53% to 69%. Our findings further suggest that cucumber-jvm still has the lowest number of impacted test cases when compared to other systems. However, the median percentage of impacted test cases has significantly increased from 6.96% to 27.85% when compared to the configuration involving changed source files.

Table 4 displays that the highest median percentage of impacted test cases was still observed in Zookeeper and HBase, like the previous configuration. However, the median percentage of impacted test cases increased from 62.39% to 68.99% in Zookeeper and from 44.85% to 69.36% in HBase. Additionally, the average percentage of impacted test cases across all analyzed systems increased from 35.16% to 53.87%. These findings suggest that this setup is less effective in terms of selecting impacted tests compared to the previous configuration involving changed source files.

We conducted further analysis to better comprehend the outcomes. Our findings indicate that while the second configuration, involving changed class tokens, can aid in narrowing down the changed scope, there are drawbacks to this approach. This is because we removed the full identification of classes in the tests-to-classes dependency graph, which leads to a higher degree of dependency graph due to common class names. For example, we observed that in the Kafka project, the class name 'Builder' was repeated 93 times, and 'StatusData' was repeated 7 times. Similarly, in the HBase project, the 'Builder' class name was repeated 26 times, and 'Writer' was repeated 5 times.

Similarly, to the previous configuration (i.e., source code files level test impact analysis), we compared our findings to the current state-of-the-art static class-level approach [16] by examining commonly studied systems (as depicted in Table 5). Our proposed method using the class tokens configuration yielded poorer results, with an average increase of around 6.5% in selected impacted tests. Nonetheless, our results indicate that our static class-level test impact analysis approach using the class tokens configuration still has the potential to decrease test overhead by an average of approximately 42%.

As before, there are two important points to note. First, none of the studies evaluated the false negative ratio, which involves determining the number of test cases

that affected the modified code but were not considered by the test impact analysis technique. This can be a potential topic for future research. Second, it is important to mention that the median percentage of impacted test results in the compared technique was estimated from a graph, and thus there may be slight differences.

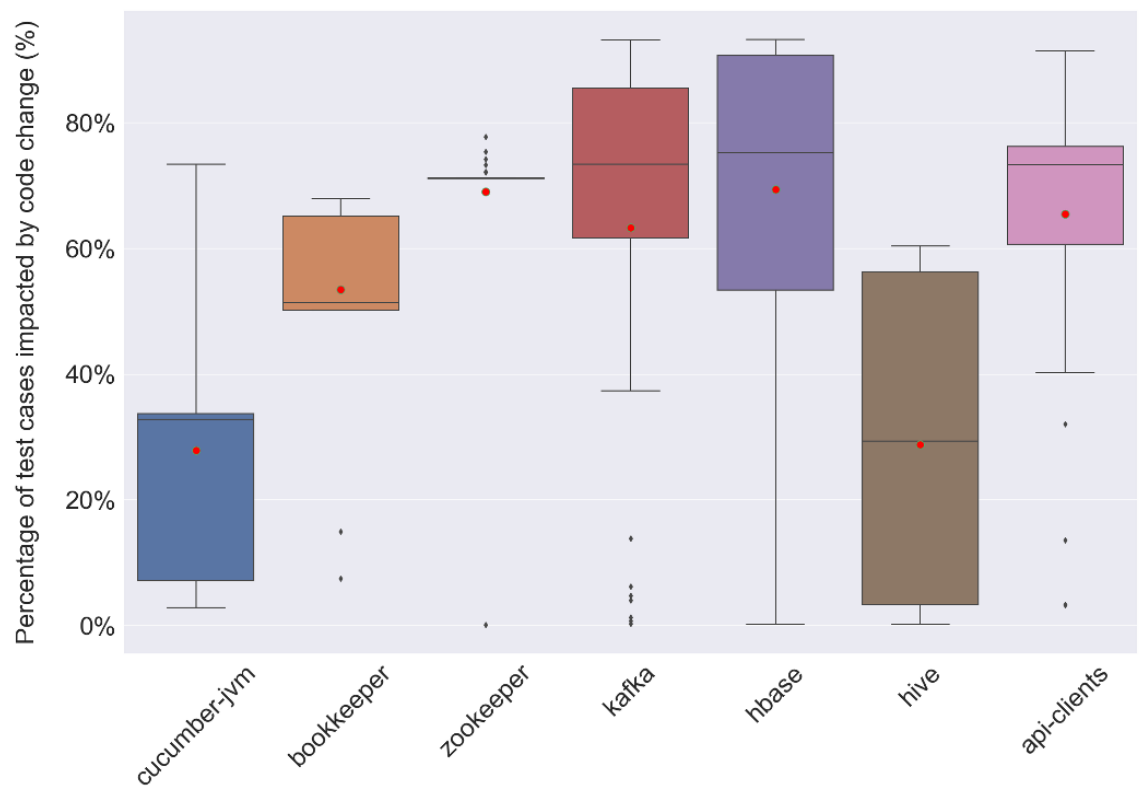


Figure 13. Distributions by percentage of impacted test cases using test to classes dependency graph (second configuration - changed class tokens).

The following chapter will encapsulate our conclusion and delve into the constraints of the current research, while accentuating potential avenues for future research.

CHAPTER 5

FUTURE WORK AND CONCLUSION

In this final chapter we aim to consolidate the key insights and findings of our research as well as reflecting on the implications of our study. We begin by revisiting limitations of our study, acknowledging any shortcomings or constraints that may have influenced the outcomes of our research. By addressing these limitations, we aim to provide a balanced and transparent account of our work. Next, we summarize the key outcomes and present our concluding thoughts.

5.1 Future Work

Our analysis encompassed seven Java-based systems spanning various domains, and we observed consistent results across all systems. However, it is important to note that since our study focused exclusively on Java implementations, the findings may not directly translate to software projects developed in different programming languages. To enhance the applicability of our results to other coding languages, future research should investigate their generalizability.

Moreover, our investigation was limited to a subset of code modifications (i.e., commits) and did not encompass the entirety of the initial stages of software system development. To overcome this constraint, future studies could undertake more comprehensive test impact analysis, expanding the scope to encompass the complete software history.

Our study relies on static analysis for identifying the dependency graph. However, it is important to acknowledge that static analysis may have inherent limitations in terms of accuracy, such as the resolution of nodes from abstract syntax trees, which could

potentially lead to inaccuracies in the results. Although we did not encounter such instances in our research, it is recommended that future studies validate our findings by applying them to different systems.

In this study, we employ static analysis to uncover the relationships between test cases and other source code elements. Our manual examination did not uncover any false positives resulting from our static analysis approach. However, it is crucial for future research to evaluate our proposed approach, as well as other static methods for test impact analysis, in terms of false positives. False positives refer to impacted test cases that are erroneously excluded by the test impact analysis implementation.

We opted for static analysis over dynamic analysis to capture class-level dependencies based on prior research [35], [37], which demonstrated comparable performance between static and dynamic test impact analysis methods. Moreover, the class-level dependency graph exhibited superior results when compared to the method-level dependency graph. To reassess our conclusions, future studies could explore the use of dynamic analysis or investigate method-level dependencies.

We classify a method as a test case when it contains invocations to testing libraries such as JUnit or TestNG, identified by the presence of the "@Test" annotation. Although we did not encounter any false detections during our examination, it is possible that certain test cases may be disregarded during execution due to annotations like "@Ignore" or "@Test(enabled=false)". Additional research is necessary to validate the accuracy of our method in identifying test cases.

This research primarily focuses on analyzing Java source code and its associated test cases. While most of the systems we examine are implemented in Java, there are cases where other programming languages are utilized. For example, in Kafka, Scala and Python constitute 21.8% and 2.5% of the code, respectively, while in Hbase, Perl and Ruby

contribute 0.9% and 1.8% of the code, respectively. Therefore, additional research is necessary to explore how different programming languages can impact the outcomes of TIA.

Based on our analysis, we observe a limited positive correlation between the number of modified files and the proportion of impacted test cases. This indicates that certain modified files may have a more substantial impact compared to others. To enhance the effectiveness of TIA, future research could explore various characteristics of modified files, including their susceptibility to change and significance within the dependency graph.

5.2 Conclusion

Numerous techniques have been suggested and assessed to minimize the overhead of test execution, including traditional software development settings incorporating TIA. This study investigates the efficacy of employing static class-level test impact analysis on seven Java systems under two distinct configurations within the framework of continuous testing. Our findings indicate that even with a limited quantity of changed files under the 1st configuration and limited changed class tokens in the 2nd configuration, a significant number of test cases (40% or more) are still impacted by code changes.

However, our results also suggest that implementing the proposed approach could potentially reduce accumulated testing overhead in continuous integration (CI) setups. Across all the systems we studied, we found that approximately 60% of tests could be excluded (i.e., not impacted by code changes) with the 1st configuration (i.e., changed source files), which shows promise for reducing testing overhead. To benchmark our results against the current state-of-the-art static class-level approach [16], we compared our findings and observed an average improvement of approximately 13% in the number of impacted tests with the 1st configuration (i.e., changed source files) and deterioration

of approximately 6.5% in the number of impacted tests when applied the 2nd configuration (i.e., changed class tokens).

In summary, our study's results suggest that integrating test impact analysis (TIA) into the development life cycle can bring benefits even with the current setup, considering the growing test frequency and associated overhead.



REFERENCES

- [1] Li W, Henry S. "Maintenance support for object-oriented programs". *Journal of Software Maintenance: Research and Practice* 1995, 7(2), pp. 131–147.
- [2] N. F. Schneidewind, "The State of Software Maintenance," in *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 303-310, March 1987, doi: 10.1109/TSE.1987.233161.
- [3] M. M. Lehman and L. A. Belady. 1985. *Program evolution: processes of software change*. Academic Press Professional, Inc., USA.
- [4] Bohner, Shawn A., and Robert S. Arnold. *Software Change Impact Analysis / Shawn A. Bohner, Robert S. Arnold*. IEEE Computer Society Press, 1996.
- [5] Bohner, "Impact analysis in the software change process: a year 2000 perspective," 1996 *Proceedings of International Conference on Software Maintenance*, Monterey, CA, USA, 1996, pp. 42-51, doi: 10.1109/ICSM.1996.564987.
- [6] P. Rovegård, L. Angelis and C. Wohlin, "An Empirical Study on Views of Importance of Change Impact Analysis Issues," in *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 516-530, July-Aug. 2008, doi: 10.1109/TSE.2008.32.
- [7] Turver, R.J. and Munro, M. (1994), An early impact analysis technique for software maintenance. *J. Softw. Maint: Res. Pract.*, 6: 35-52.
<https://doi.org/10.1002/smr.4360060104>.
- [8] Sue Black, Deriving an approximation algorithm for automatic computation of ripple effect measures, *Information and Software Technology*, Volume 50, Issues 7–8, 2008, pp. 723-736, ISSN 0950-5849, <https://doi.org/10.1016/j.infsof.2007.07.008>.
- [9] Xiaoxia Ren, B. G. Ryder, M. Stoerzer and F. Tip, "Chianti: a change impact analysis tool for Java programs," *Proceedings. 27th International Conference on Software Engineering*, 2005. ICSE 2005., St. Louis, MO, USA, 2005, pp. 664–665, doi: 10.1109/ICSE.2005.1553643.
- [10] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International*

Symposium on Software Testing and Analysis (ISSTA 2015). Association for Computing Machinery, New York, NY, USA, 211–222.

<https://doi.org/10.1145/2771783.2771784>.

[11] August Shi, Tiffany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).

Association for Computing Machinery, New York, NY, USA, pp. 237–247.

<https://doi.org/10.1145/2786805.2786878>.

[12] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (April 1997), pp. 173–210.

<https://doi.org/10.1145/248233.248262>.

[13] Zhang, Lingming & Marinov, Darko & Zhang, Lu & Khurshid, Sarfraz. (2011). An Empirical Study of JUnit Test-Suite Reduction. Proceedings - International Symposium on Software Reliability Engineering, pp. 170-179. 10.1109/ISSRE.2011.26.

[14] A. Vahabzadeh, A. Stocco and A. Mesbah, "Fine-Grained Test Minimization," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 2018, pp. 210-221, doi: 10.1145/3180155.3180203.

[15] Yoo, Shin & Harman, Mark. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability.* 22.

10.1002/stvr.430.

[16] Z. Peng, T. -H. Chen and J. Yang, "Revisiting Test Impact Analysis in Continuous Testing from the Perspective of Code Dependencies," in *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 1979-1993, 1 June 2022, doi:

10.1109/TSE.2020.3045914.

[17] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. *SIGSOFT Softw. Eng. Notes* 29, 6 (November 2004), pp. 241–251. <https://doi.org/10.1145/1041685.1029928>.

[18] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern

software evolution. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). Association for Computing Machinery, New York, NY, USA, pp. 583–594.

<https://doi.org/10.1145/2950290.2950361>.

[19] Microsoft, “Test impact analysis in visual studio test” [Online].

<https://learn.microsoft.com/en-us/azure/devops/pipelines/test/test-impact-analysis?view=azure-devops>, 14/7/2023.

[20] Gitlab, “Continuous Integration and Delivery”, [Online].

<https://about.gitlab.com/features/continuous-integration/>, 14/7/2023.

[21] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, pp. 805–816. <https://doi.org/10.1145/2786805.2786850>.

[22] Gradle, “Gradle Build Tool”, [Online]. <https://gradle.org/>, 14/7/2023.

[23] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003., Denver, CO, USA, 2003, pp. 281-292, doi: 10.1109/ISSRE.2003.1251050.

[24] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2013. Data debugging with continuous testing. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). Association for Computing Machinery, New York, NY, USA, pp. 631–634. <https://doi.org/10.1145/2491411.2494580>.

[25] Memon, Atif & Gao, Zebao & Nguyen, Bao & Dhanda, Sanjeev & Nickell, Eric & Siemborski, Rob & Micco, John. (2017). Taming Google-Scale Continuous Testing. 233-242. 10.1109/ICSE-SEIP.2017.16.

[26] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In Proceedings of the

- 39th International Conference on Software Engineering (ICSE '17). IEEE Press, pp. 700–711. <https://doi.org/10.1109/ICSE.2017.70>.
- [27] Z. Li, M. Harman and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," in *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, April 2007, doi: 10.1109/TSE.2007.38.
- [28] Mei, Hong & Hao, Dan & Zhang, Lingming & Zhang, Lu & Zhou, Ji & Rothermel, Gregg. (2012). A Static Approach to Prioritizing JUnit Test Cases. *Software Engineering, IEEE Transactions on*. 38. pp. 1258-1275. 10.1109/TSE.2011.106.
- [29] Rothermel, Gregg & Untch, Roland & Chu, Chengyun & Harrold, Mary. (2000). Prioritizing Test Cases for Regression Testing. *Software Engineering, IEEE Transactions on*. 27.
- [30] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, pp. 268–279.
- [31] Thomas, Stephen & Hemmati, Hadi & Hassan, Ahmed E. & Blostein, Dorothea. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*. 19. 10.1007/s10664-012-9219-7.
- [32] Shin Yoo, Mark Harman, and David Clark. 2013. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Trans. Softw. Eng. Methodol.* 22, 3, Article 19 (July 2013), 29 pages. <https://doi.org/10.1145/2491509.2491513>.
- [33] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, pp. 192–201.
- [34] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of*

- Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, pp. 235–245. <https://doi.org/10.1145/2635868.2635910>.
- [35] Q. Luo, K. Moran, L. Zhang and D. Poshyvanyk, "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects," in *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1054-1080, 1 Nov. 2019, doi: 10.1109/TSE.2018.2822270.
- [36] Y. Zhu, E. Shihab and P. C. Rigby, "Test Re-Prioritization in Continuous Testing Environments," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 69-79, doi: 10.1109/ICSME.2018.00016.
- [37] Machalica, Mateusz & Samylnin, Alex & Porth, Meredith & Chandra, Satish. (2019). Predictive Test Selection. 91-100. 10.1109/ICSE-SEIP.2019.00018.
- [38] Gambi, Alessio & Bell, Jonathan & Zeller, Andreas. (2018). Practical Test Dependency Detection. pp. 1-11. 10.1109/ICST.2018.00011.
- [39] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 33, pp. 1–11. <https://doi.org/10.1145/2393596.2393634>.
- [40] JavaParser, [Online]. <https://javapar.org>, 14/7/2023.
- [41] F. Palomba and A. Zaidman, "Notice of Retraction: Does Refactoring of Test Smells Induce Fixing Flaky Tests?," 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 1-12, doi: 10.1109/ICSME.2017.12.
- [42] A. Vahabzadeh, A. M. Fard and A. Mesbah, "An empirical study of bugs in test code," 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 2015, pp. 101-110, doi: 10.1109/ICSM.2015.7332456.
- [43] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*.

Association for Computing Machinery, New York, NY, USA, 643–653.

<https://doi.org/10.1145/2635868.2635920>.

[44] “The Rise of Test Impact Analysis” [Online]. <https://martinfowler.com/articles/rise-test-impact-analysis.html>, 14/7/2023.

[45] Li, Bixin & Sun, Xiaobing & Leung, Hareton & Zhang, Sai. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*. 23. 10.1002/stvr.1475.

[46] “CI/CD: Continuous Integration and Continuous Delivery”, [Online].

<https://www.geeksforgeeks.org/ci-cd-continuous-integration-and-continuous-delivery/>, 14/7/2023.

[47] “Improving Test Execution Efficiency with Test Impact Analysis”, [Online].

<https://docs.teamscale.com/tutorial/improving-test-efficiency/#setting-up-the-test-impact-analysis>, 14/7/2023.

[48] Ogheneovo, Edward. (2014). On the Relationship between Software Complexity and Maintenance Costs. *Journal of Computer and Communications*. 02. pp. 1-16. 10.4236/jcc.2014.214001.

[49] Ren, Yongchang & Xing, Tao & Chen, Xiaoji & Chai, Xuguang. (2011). Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method. *Intelligent System and Applications (ISA)*. 10.1109/ISA.2011.5873461.

[50] “What is Test Impact Analysis?”. [Online].

<https://www.launchableinc.com/blog/what-is-test-impact-analysis/>, 14/7/2023.

[51] Sommerville, I. (2011). *Software engineering* (9th ed.). Addison-Wesley.

[52] Pressman, R. S., & Maxim, B. R. (2015). *Software engineering: A practitioner's approach* (8th ed.). McGraw-Hill Education.

[53] Bishop, M. (2003). *Computer security: Art and science*. Addison-Wesley Professional.

[54] Fogel, K., & Wilson, B. (2018). *Producing open-source software: How to run a successful free software project*. O'Reilly Media, Inc.

- [55] Laudon, K. C., & Laudon, J. P. (2016). Management information systems: Managing the digital firm (14th ed.). Pearson Education.
- [56] Swanson, M., Bowen, P., Phillips, A., & Snyder, B. (2014). Computer security handbook (6th ed.). John Wiley & Sons.
- [57] Lientz, B. P., & Swanson, E. B. (2015). Software maintenance management: Evaluation and continuous improvement. John Wiley & Sons.
- [58] Roger S. P., & Bruce M. (2019). Software Engineering: A Practitioner's Approach. McGraw-Hill Education.



APPENDIX

BookKeeper test-to-classes Dependency Graph (with qualified names):

https://drive.google.com/file/d/1YRMLLTR6_2ozfbE6d-tnImXJ1Sz-Ejml/view?usp=sharing

BookKeeper test-to-class-tokens Dependency Graph (with simplified class names):

https://drive.google.com/file/d/1383B6v25bjVKTZlecXIMbWU_5rCa223x/view?usp=sharing



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

VITA

NAME : Alon Basin
DATE OF BIRTH : 2/9/1986
PLACE OF BIRTH : Israel
INSTITUTIONS ATTENDED : Bachelor of Economics, Ben Gurion University of
the Negev



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY