Prioritization and Evolution of Regression Tests: Data-Driven Solutions for Continuous Integration Contexts

by

Ahmadreza Saboor Yaraghi

Thesis submitted to the
Faculty of Engineering
In partial fulfillment of the requirements
For the Ph.D. degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Regression testing is a software testing method used to ensure that existing functionalities work correctly when changes are applied to a software system. It involves executing all or a subset of test cases. In Continuous Integration (CI) contexts, where code changes are frequent, regression testing is a critical process in ensuring software quality. However, the high time and resource costs of executing regression tests, coupled with the challenges of evolving test cases to align with rapidly changing software, pose significant challenges. This thesis addresses these challenges by optimizing the prioritization of regression tests and supporting their evolution following code changes.

The first part of the thesis focuses on Test Case Prioritization (TCP) to enhance the efficiency of regression testing in CI contexts. Many recent TCP studies employ Machine Learning (ML) techniques to address the dynamic and complex nature of CI. However, most of them use a limited number of features for training ML models and evaluate the models on subjects for which the application of TCP offers little practical value, due to their small regression testing time and low number of failed builds. This thesis begins by defining a data model that represents the data sources and their relationships within a CI environment. We used this model as the basis for identifying and collecting a comprehensive set of features, encompassing all features previously used in related studies. We collect these features from 25 open-source projects with significantly larger regression testing times and higher numbers of failed builds than existing benchmarks. Extensive experiments are conducted to assess the performance of ML-based TCP, evaluating its effectiveness, feature collection costs, and the decay rate of model performance over time. Additionally, this study examines the trade-offs between feature collection costs and TCP effectiveness, providing practical insights for deploying ML-based TCP in real-world CI environments.

The second part of the thesis introduces TaRGET (Test **R**epair **GE**nera**T**or), a novel approach for automating the repair of broken test cases. Ensuring the quality of software systems through regression testing is essential, yet maintaining test cases poses significant challenges and costs. The need for frequent updates to align with the evolving system under test (SUT) often entails high complexity and cost for maintaining these test cases. Broken test cases, if left unrepaired, degrade the quality of test suites, disrupt the development process, and waste developers' time. TaRGET leverages pre-trained code language models (CLMs) and treats test repair as a language translation task. It employs a two-step process that integrates essential contextual information, such as SUT code changes, and fine-tunes CLMs to enhance the accuracy and relevance of the generated repairs. To validate TaRGET, the thesis introduces TaRBench, a comprehensive benchmark containing over 45,000 test repair instances across 59 open-source projects, addressing the limitations of existing benchmarks, which often have a small number of instances or projects and lack diversity in test repair scenarios. Experimental results demonstrate that TaRGET achieves a 66.1% exact match accuracy in repairing test cases. The study further examines TaRGET's effectiveness across different test repair scenarios using both quantitative and qualitative analysis. Practical guidelines are also provided for identifying scenarios where the generated repairs might be less reliable. The study also examines the

effectiveness of repair generation for new projects without fine-tuning and evaluates the necessity of project-specific fine-tuning for achieving acceptable performance.

By combining advanced ML techniques with practical tools and benchmarks, this thesis provides a unified framework for supporting the prioritization and evolution of regression tests. To enable reproducibility and reusability, all experimental data, tools, and benchmarks are made publicly available for practitioners and researchers.

Acknowledgements

This research was supported by a grant from Huawei Technologies Canada, Mitacs Canada, as well as the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Research Ireland grant 13/RC/2094-2. Additionally, this research was enabled in part by computational support from the Digital Research Alliance of Canada [1].

First and foremost, I am sincerely grateful to my supervisor, Prof. Lionel Briand, for his invaluable mentorship. This work would not have been possible without his insightful feedback, expertise, and dedicated and attentive support. His constructive guidance has been essential in shaping the quality of this research.

I am also thankful to Dr. Mojtaba Bagherzadeh and Dr. Nafiseh Kahani for their invaluable advice and mentorship. Their suggestions and guidance have significantly enriched this work. In particular, I appreciate our brainstorming sessions, which provided clarity and direction during challenging phases of this research. I also extend my appreciation to Darren Holden for his meaningful contributions.

I am grateful to my Thesis Advisory Committee members, Prof. Baishakhi Ray, Prof. Olga Baysal, Prof. Herna Viktor, and Prof. Stéphane Somé, for their valuable feedback, which helped improve and strengthen this work.

Finally, I sincerely thank my colleagues and friends in the Nanda Lab for shaping a collaborative and friendly environment.

Dedication

All the strength, abilities, and blessings that have brought me to this stage come from Allah, the Almighty Creator, the Lord of the Earth and the Heavens, who created me, shaped me into who I am, and accompanied me in every moment of my existence. Only He knows how deeply His words have influenced and inspired me during this part of my life, a time that will forever remain a remarkable highlight.

This work is dedicated to my mother and father, whose love, care, and endless prayers only God fully knows. Their unwavering dedication and support to my success are beyond measure.

To my brother, who, despite being younger, has always inspired me with his wisdom and advice.

And to my amazing friends and to everyone whose encouraging presence have shaped my life throughout this journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Regression testing is a software testing method used to ensure existing functionalities work correctly when changes are applied to a software system. It involves executing all or a subset of test cases to confirm that recent code updates do not negatively impact other parts of the system. In modern software development, regression testing is a critical component of Continuous Integration (CI) processes, which automate the integration and testing of code changes from multiple developers. Although CI accelerates development and shortens release cycles, executing regression tests, particularly in large systems, can be time-consuming and resource-intensive, causing delays [2–4]. Therefore, optimizing regression test execution through techniques like Test Case Prioritization (TCP) is crucial.

Moreover, the rapid pace of code changes in CI environments necessitates continuous adaptation of regression tests to maintain their quality and coverage [5, 6]. Developers often need to update test cases as the system under test (SUT) evolves, which results in significant operational and maintenance costs. Automating the evolution of regression tests can reduce this burden by ensuring tests remain effective while minimizing manual intervention, ultimately reducing the overall costs and effort associated with evolving the test codebase.

Despite existing TCP and automated test evolution approaches, several challenges and limitations persist in these areas. Existing ML-based TCP techniques use limited feature sets and are evaluated on inadequate subjects with few failed test cases. There is limited reporting on feature collection costs and unclear guidelines for practical implementation [7]. Additionally, existing automatic test evolution approaches suffer from limited generalizability, often being project-specific, programming language-restricted, or targeting a limited group of test evolution scenarios. Many techniques demonstrate poor scalability for large codebases, especially with static or dynamic analysis-based approaches. Evaluations are typically conducted on small, non-diverse benchmarks with limited reproducibility [8].

This thesis addresses these challenges through two complementary contributions. First, it advances TCP techniques by developing a comprehensive data model with 150 features

across nine groups collected from all CI data sources, enabling more effective ML-based prioritization. Through extensive empirical analysis on 25 subjects with 21.5k builds, it provides practical guidelines on feature selection, data collection overhead, and model retraining frequency—aspects previously unexplored in the literature. Second, it addresses existing automatic test evolution challenges through a novel approach that leverages language models for automatic test repair, which is not limited to specific test evolution scenarios or programming languages. This thesis proposes the largest available benchmark (45,373 repairs across 59 projects) in this domain.

Together, these contributions significantly advance the state-of-the-art in regression testing by addressing scalability, generalizability, and quality challenges through reproducible, data-driven, ML-powered approaches with comprehensive empirical validation.

1.2 Thesis Objectives

The primary goal of this thesis is to develop practical techniques and provide insights for prioritizing and evolving regression tests in response to code changes within software systems. To achieve this goal, the research focuses on the following two objectives:

TO_1 (Test Case Prioritization in Continuous Integration Contexts): This objective seeks to optimize regression test execution by prioritizing test cases efficiently, reducing resource consumption, and providing faster feedback on system faults.

TO_2 (Automated Test Case Repair Using Language Models): This objective aims to enhance the evolution of regression test code through automation, leveraging language models to minimize manual effort while maintaining high testing quality.

The objectives address distinct phases of testing with different operational constraints. TO_1 prioritizes scalability and computational efficiency, which is critical for CI environments, whereas TO_2 emphasizes the correctness and effectiveness of the test code evolution over the solution's runtime performance, as test evolution typically occurs during development iterations rather than time-sensitive deployment phases.

1.3 Contributions

This thesis addresses critical challenges in regression testing through two novel approaches that improve test prioritization and automated test repair in continuous integration environments. The research makes four types of key advancements over existing approaches: (1) comprehensive methodology development, (2) large-scale empirical validation, (3) practical implementation guidelines, and (4) creation of reproducible and reusable benchmarks. The specific contributions are organized as follows:

TO_1: This contribution improves the efficiency of regression test execution through ML-based prioritization, aiming to detect faults as early as possible. The main contributions are:

- **Comprehensive Feature Model:** A data-driven feature model was created to enhance Machine Learning (ML)-based Test Case Prioritization (TCP). This model integrates features from multiple CI data sources (e.g., test execution history, source code metrics) to improve fault detection effectiveness.
- **Empirical Validation:** An evaluation was conducted on 25 open-source projects with over 21,500 builds and 2,500 failed builds, demonstrating the impact of different feature groups on TCP effectiveness.
- **Cost-Effectiveness Analysis:** An analysis of the trade-offs between feature collection cost and TCP effectiveness was performed, providing guidelines for optimizing prioritization in real-world CI settings.
- **Reproducibility and Benchmark Creation:** The thesis introduces a publicly available benchmark with real-world projects along with the code for experimentation and benchmark creation, allowing researchers to compare and evaluate new TCP techniques [9].

TO_2: This contribution reduces test evolution effort through context-aware automated test repair, aiming to maintain testing quality following frequent software updates. The key contributions include:

- **Introduction of TaRGET:** A novel transformer-based test repair generation approach using pre-trained code language models (CLMs) that effectively employs syntactic and semantic context prioritization [10] from the System Under Test (SUT).
- **Creation of TaRBench Benchmark:** A large and diverse dataset containing over 45,000 test repair instances from 59 open-source projects, addressing the limitations of existing benchmarks [11].
- **Empirical Validation:** Extensive experiments evaluating various input data formatting techniques and different CLMs demonstrate that TaRGET achieves a 66.1% exact match accuracy in test repair, significantly outperforming prior approaches.
- **Quantitative and Qualitative Analysis:** A detailed analysis reveals that TaRGET's performance degrades with more complex test repairs. A repair predictive model is proposed to enhance TaRGET's practical applicability, by allowing engineers to determine when to trust the repair. Furthermore, results show that TaRGET generalizes well to previously unseen projects without requiring project-specific fine-tuning.

By integrating ML-based TCP and automated test repair, this thesis holistically improves the efficiency and maintainability of regression testing in CI environments from both execution and development perspectives. All research outputs, including tools, benchmarks, and replication packages, are publicly available to support further advancements in the field.

1.4 Publications

The research presented in this thesis has led to the following peer-reviewed publications, both of which have been accepted for publication in *IEEE Transactions on Software Engineering (IEEE TSE)*:

1. Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C. Briand, "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts" in IEEE Transactions on Software Engineering, vol. 49, no. 04, pp. 1615-1639, April 2023.

   Presentation: This work was presented at the Journal First Track at ICSE 2023 in Melbourne, Australia.

   DOI: https://doi.org/10.1109/TSE.2022.3184842

2. Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel C. Briand, "Automated Test Case Repair Using Language Models" in IEEE Transactions on Software Engineering, vol. 51, no. 04, pp. 1104-1133, April 2025.

   Presentation: This work will be presented at the Journal First Track at FSE 2025 in Trondheim, Norway.

   DOI: https://doi.org/10.1109/TSE.2025.3541166

1.5 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 addresses TO_1 by presenting:
  - an introduction outlining the work's motivation and primary contributions;
  - background information, including key definitions to establish the context of ML-based TCP;
  - the detailed development of a feature model tailored to ML-based TCP;
  - the subject selection and data collection process;
  - an empirical evaluation of TCP's efficiency and accuracy, analyzing its practical application;
  - a comparative review of related work, highlighting our approach's unique aspects; and
  - a conclusion summarizing key findings and suggesting directions for future work.
- Chapter 3 aims to address TO_2 by presenting:
  - an introduction discussing the motivation and contributions of this work;

  - background covering concepts related to code language models relevant to our approach;
  - our proposed automated test repair technique, referred to as TARGET;
  - the rigorous and systematic creation of a large, diverse benchmark, named TARBENCH;
  - a large-scale empirical evaluation of the performance of our approach;
  - a review of related work and their limitations addressed by our work; and
  - a conclusion summarizing findings and potential areas for future improvement.
- Finally, Chapter 4 presents a summary of the thesis contributions, along with discussions and suggestions for future work.

Chapter 2

Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts

This chapter focuses on addressing TO_1, which involves optimizing the execution of regression tests. The findings and methodologies presented in this chapter have been published in the *IEEE Transactions on Software Engineering (TSE)* journal [12].

2.1 Introduction

Application of Continuous Integration (CI) significantly reduces integration problems, speeds up development time, and shortens release time by allowing software developers to integrate their work more frequently with the mainline codebase rather than with deferred integration. Each integration is automatically built and usually validated by regression testing (a CI cycle), upon completion of which the developers are provided feedback. The execution of regression tests may require significant computational resources and take hours or even days to be completed. The prolonged execution of regression tests can delay the CI cycles and prevent timely feedback to the developers.

Test Case Prioritization (TCP) techniques address the long execution time of regression testing by prioritizing (ranking) the execution of test cases such that faults can be detected as early as possible, i.e., the test cases with high fault detection probability and lower execution times are given higher execution priority. These techniques can be classified into heuristic-based and ML-based techniques. Heuristic-based techniques often make use of code coverage analysis and test execution history. Collecting coverage information is in general challenging and, furthermore, precise coverage information requires dynamic analysis, which is difficult to apply in practice, more particularly so in a CI context, mainly due to computational overhead and applicability issues [13–17]. Concerning heuristics based on the execution history, relying solely on such history for TCP may not lead to stable results for complex systems in a CI context [14, 18]. In addition to cost and effectiveness

issues, heuristics are often defined statically, and there is no standard procedure to tune them based on new changes. Adapting to new changes is critical for TCP in a CI context, due to the frequently-changing codebases.

ML-based TCP techniques train ML models based on various features collected from different sources, such as execution history, to prioritize test cases. In general, ML techniques enable the training of effective models based on imperfect features. They also can adapt to new changes either through incremental learning [19] or retraining. Thus, many researchers have relied on ML techniques to address TCP in the CI context. According to a recent survey [7], various ML-based TCP techniques have been applied in the CI context. However, existing work has not relied on a comprehensive set of features for training ML models, which is crucial to achieve high effectiveness. They also often used inadequate evaluation subjects that have a low number of failed test cases and a very short regression testing time. Evaluating the costs and benefits of a comprehensive set of features cannot be done on such subjects, as applying TCP techniques is practically inefficient in those cases. Further, none of the existing work reported the cost and time required for collecting features. Thus, despite significant progress, it is still not clear whether or not existing ML-based techniques have reached their full potential for TCP, as they do not take full advantage of all available data sources. Last, several practical questions regarding the application of ML-based techniques remain unanswered, notably what features can be collected and at what cost to support ML-based TCP. These questions include: What is the trade-off between using certain features in terms of data collection time and their impact on the effectiveness of ML models for TCP? How often do the ML models need to be retrained to remain useful? Our goal is to provide concrete recommendations regarding these questions.

To address the issues discussed above, we first define a data model characterizing the operational flow of a typical CI environment (e.g., Travis CI). The data model captures the available data sources and their relations at a high level. We then define a set of features based on the data model and a thorough review of the used features in related work. This results in 150 features across nine groups, which can be collected by analyzing five data sources, including build logs, the source code of test cases and its Version Control System (VCS) history, the code of the system under test, and its VCS history.

To investigate the benefits and costs of the 150 features, we conducted a large-scale empirical analysis. To do that, we first defined and developed methods to extract the features based on a popular CI tool, Travis CI, and projects written in the Java programming language. We then designed and conducted an extensive experiment, based on the latest 50 builds of 25 subjects with an adequate number of failed test cases and a regression testing time of at least five minutes, to answer the following questions.

- How does data collection time across feature groups compare, and are they significantly different?
  The result shows that data collection time ranges between 0.1 to 11.7 minutes across subjects for each build, the main portion of which is related to features that require static coverage analysis.

- How is the effectiveness of ML-based TCP using the defined comprehensive feature set? How does the use of each feature impact effectiveness?
  ML model trained using the full comprehensive set of features can reach promising results for TCP for most study subjects. Test execution history features have the greatest impact on TCP effectiveness.
- How often do the ML-based TCP models need to be retrained? What is the best trade-off between retraining frequency and model effectiveness?
  The result shows that retraining ML-based TCP models should be performed no less frequently than every 11 builds, and as frequently as possible to achieve the best TCP effectiveness.
- What are the trade-offs between the data collection time of features and their impact on the effectiveness of ML-based TCP?
  Depending on the acceptable degree of data collection overhead and the need for higher effectiveness, we suggest four alternative choices: using the comprehensive set of features for training the ML model, (1) with or (2) without retaining at each build, (3) using only features based on the execution history of test cases for training, and (4) rely on heuristics based on the failure history of test cases.

Overall, this work makes the following contributions:

- Collection and evaluation of a comprehensive feature set for training ML models for TCP. This set includes all features used by previous studies and addresses all data sources available in the CI context. No previous study is nearly as comprehensive as our work in this respect.
- Answering four important practical questions based on extensive empirical analysis, as discussed above.
- A benchmark of 25 subjects with 21.5k builds and 2.5k failed builds that enables a fair comparison and evaluation of future TCP techniques. We also made our data collection tools available, which can be used to extend and update the subjects[1].

2.2 Background

In this section, we first describe how ML-based test case prioritization (TCP) fits into a continuous integration (CI) context. We precisely define the ML problem that our study aims to solve. We then explain the methods that are used to extract dependencies between source code entities and test cases and among source code entities. Further, we present our approach for classifying software systems' changes (commits) into *defect-fix* and *non-defect* classes. The dependency and classification of changes are required for computing coverage-based test case features. More specifically, the dependency data is used for the change and impact analysis features, and the classification data is used for test case coverage features. Details regarding the definition of the features are discussed in Section 2.3.

[1] https://github.com/Ahmadreza-SY/TCP-CI

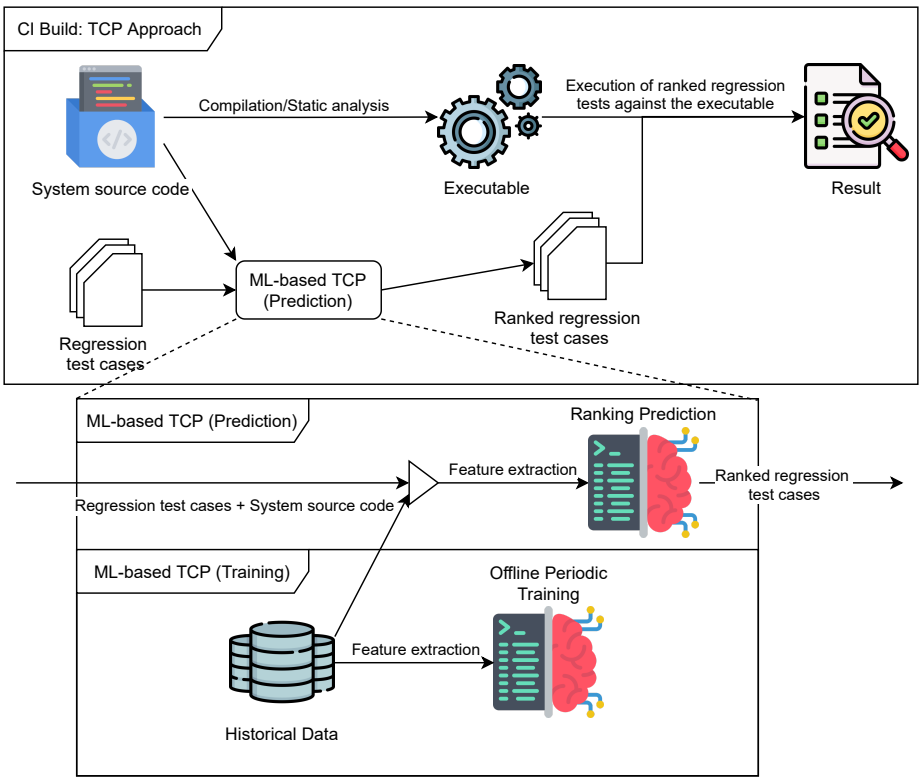

Figure 2.1: ML-based TCP in CI Context

2.2.1 ML-based Test Case Prioritization in CI Context

In a typical CI build, the system source code is built, and then its quality is validated by running a set of regression test cases, whose execution can be costly in terms of time and computation resources. Test Case Prioritization (*TCP*) techniques address this challenge by prioritizing (ranking) their execution such that test cases with a higher probability of fault detection and lower execution times are given higher execution priority. Depending on the execution budget, top-ranked test cases can be selected for execution or the regression testing stops once a test case fails. Figure 2.1 depicts a CI build process relying on ML-based TCP. First, similar to a typical CI build, the process begins with building the system. Second, test case features are extracted from several data sources (e.g., system source code) as discussed in Section 2.3. Third, the extracted features are passed to a pretrained ML ranking model for prediction. This model ranks regression test cases to be executed against the system. The pretrained ML ranking model is trained periodically in an offline environment and therefore does not cause computational overhead to the CI build.

The feature extraction step of ML-based TCP, which occurs before ranking regression test cases for each build, may significantly affect the overall CI runtime. Though the computation of some test case features only entails a simple analysis (e.g., a database query), other features require code analysis and the computation of metrics, thus possibly delaying the execution of a CI build. This motivates us to investigate the time needed for

extracting and computing regression test case features for each CI build and to measure the delay it causes. We investigate this issue in the first research question (RQ1) in Section 2.4.

Given a test suite T that contains a number of regression test cases, a feature set F that contains features for all test cases, and a ML ranking model M, we define an ML-based TCP as a function that takes M and F as input and produces an ordering of T called T^* that is intended to be as close as possible to the unknown optimal ordering. In this work, similar to existing TCP techniques [20], we assume that in the optimal ordering of test suite T (T^{opt}), test cases are first sorted by their verdict (i.e., fail/pass state after execution) with failed test cases at the beginning. Second, test cases are sorted by their execution time in ascending order. This ensures the detection of faults as early as possible during regression testing. Also, when the execution budget is limited, the execution of top-ranked test cases (Test Case Selection) may be the only option for an efficient use of the budget.

2.2.2 Dependency and Impact Analysis of Source Code Entities

Code coverage can be calculated in different ways depending on the selected analysis techniques. In this work, for practical reasons related in part to scalability as pointed out by the industry partner supporting this research, we use a mix of lightweight static analysis and association rule mining to calculate the dependency graph between source code files of the System Under Test (SUT) and its test cases, assuming that test cases are developed using the same language as the SUT. The dependency graph is a directed graph, each node of which refers to a source file. Each edge shows a dependency relation from the source node to the destination, i.e., the source node depends on the destination. The edges are also weighed with coverage scores calculated based on the association rule mining of co-changes of source files corresponding to the source and destination nodes. In the following, we refer to the source code file corresponding to a node by only referring to the latter.

A dependency graph is constructed for a version of the SUT and its test cases in a specific build according to the following steps. First, the algorithm that creates the dependency graph accepts a set of test cases and all changes (commits) of the system up to the target version. It then iterates over all source files and creates a node for each one of them. Edges from a node to other nodes are added if the former either calls a function from the latter or imports them. When the graph is completed, then a dependency score is calculated for each edge based on the association rule mining results of co-changes among nodes. The algorithm iterates over all commits of the development history of a given SUT, extracts all pairs of co-changes which are associated with edges of the dependency graph, and finally calculates the support, confidence, and lift for each edge as discussed below.

Assuming that CH is a **list** of change sets in the project's commit history: $CH = (\{f_1, f_2, f_3\}, \{f_1, f_3\}, \{f_2\}, \{f_1, f_2, f_3, f_4\}, ...)$ with f_i representing source files, let us define the following helper functions.

$$\text{p\_cnt}(f_i, f_j) = |\{(f_i, f_j) \subseteq E : E \in CH \wedge 1 \leq i, j \leq n \wedge i \neq j\}|$$

$$\text{cnt}(f_i) = |\{f_i \in E : E \in CH \wedge 1 \leq i \leq n\}|$$

where n is the number of source files. Function p_cnt(f_i, f_j) computes the number of commits in which f_i and f_j were changed together. Also, function cnt(f_i) computes the number of commits in which f_i was changed. We define support, confidence, and lift as follows.

$$\text{support}(f_i, f_j) = \frac{\text{p\_cnt}(f_i, f_j)}{|CH|}$$

$$\text{confidence}(f_i, f_j) = \frac{\text{p\_cnt}(f_i, f_j)}{\text{cnt}(f_i)}$$

$$\text{lift}(f_i, f_j) = \frac{\text{p\_cnt}(f_i, f_j)}{\text{cnt}(f_i) * \text{cnt}(f_j)}$$

Definition 1. *Coverage.* Here, we assume that the test cases are developed using the same language as the SUT. Thus, the source files covered by a test case refer to all source files that the test case depends on, i.e., the test case either calls a function in the related source files or imports them.

The coverage score refers to the confidence of an association rule between a test case and a source file that is computed using association rule mining. To be more precise, the coverage score function (cov_score) is defined as:

$$\text{cov\_score}(f, t) = \text{confidence}(f, t)$$

where f is a source file and t is a test case. In other words, cov_score(f, t) estimates the conditional probability of t being changed given that f is changed.

2.2.3 Classification of Defect-fix Commits

Definition 2. *Previously Detected Faults (PDF).* Let us define Previously Detected Faults (PDF) of a source file f as the number of faults detected in f according to its change history:

$$\text{PDF}(f) = |\{c \in C | f \in \text{chn}(c) \wedge \text{chn\_cls}(c) = \textit{defect-fix}\}|$$

where C is the set of all commits of a project, chn(c) is a function that returns a set of changed files for commit c, and chn_cls(c) is a commit classifier that classifies commit c into *defect-fix* or *non-defect* classes.

The classification of commits is still an active research area, initiated by Mockus et al. [21], who used three keyword-based rules to automatically classify modification requests' textual descriptions into four maintenance groups. Hindle et al. [22] applied machine learning models using project-dependent features (e.g., authors, modules, and file types) as well as commits' word distributions. Levin et al. [23] also used a keyword-based approach as well as source code changes from commits. In a recent effort, Zafar et al. [24] created a commit classifier that reached the high accuracy (proportion of correct predictions) of 92.2% for their test dataset. They used BERT [25], a state-of-the-art text classification method.

BERT [25] is a pretrained transformer-based language model which is trained on massive corpora including BooksCorpus [26] with 800M words and English Wikipedia with 2,500M words. BERT achieved state-of-the-art results on a number of natural language processing tasks. The original $BERT_{BASE}$ model has 110M parameters with 12 encoder layers, and it requires GPU resources for effective training and prediction. For this reason, BERT is computationally expensive and compared to other machine learning models, this is a major hurdle in a CI context, due to the resource and timing constraints of CI builds.

Thus, we aim to use TF-IDF (term frequency-inverse document frequency) and simpler classification techniques (e.g., Random Forests [27] and SVM [28]) to train a commit classifier that requires much less computation time than BERT [25], while retaining high classification performance. TF-IDF [29] measures the importance of a word based on its frequency in a corpus and its presence in individual documents. In the following, we discuss the details of training and evaluating a lightweight commit classifier.

Preparing the training dataset. We collected three publicly available datasets in the literature, including those of Zafar et al. [24], Levin et al. [23], and Berger et al. [30], and merged them into a unified training set. Each dataset includes commit messages annotated with binary labels indicating whether a commit is a defect-fix or not. This unified training set contains 3,681 commits, 35.2% of which are defect-fix commits. The minority class (defect-fix) is not extremely underrepresented and is unlikely to cause bias or overfitting. Nonetheless, we prioritize the F1 score for evaluation because it balances false positives and false negatives. Using accuracy may be misleading as a classifier could achieve high accuracy simply by mostly predicting the majority class. We applied several preprocessing techniques: converting all text to lowercase, replacing URLs with a placeholder token, removing non-alphanumeric characters, eliminating stop words, and applying stemming. We then transformed the processed commit messages into vector representations using the TF-IDF method.

Training and evaluation of classifiers. We evaluated four classifiers—Support Vector Machines (SVM) [28], Random Forest [27], XGBoost [31], and Logistic Regression [32]—using 10-fold cross-validation on the unified training set, with the F1 score as the primary performance metric. A Friedman test [33] revealed a statistically significant difference among the classifiers (p-value = 0.00021). Subsequent pairwise comparisons with the Nemenyi post hoc test [34] showed that SVM, Random Forest, and XGBoost each significantly outperformed Logistic Regression (all p-value < 0.05), while no statistically significant differences were observed among SVM, Random Forest, and XGBoost. Based on the average F1 scores across folds—80% for XGBoost, 79% for Random Forest, 78% for SVM, and 73% for Logistic Regression—XGBoost not only achieves the highest performance but also benefits from a highly optimized gradient boosting framework. This efficiency is particularly valuable in CI contexts. Therefore, we selected XGBoost as our classifier.

A key difference between XGBoost and BERT is their approach to feature representation. XGBoost relies on manually engineered features—in our case, TF-IDF vectors—whereas BERT does the feature engineering itself and generates contextual embeddings directly from the raw text, capturing deep semantic and syntactic relationships.

Dataset	BERT	XGBoost
Zafar et al. [24]	88.5%	83.8%
Levin et al. [23]	68.5%	62.9%
Berger et al. [30]	76.0%	75.4%
Berger et al. [30] (subset [24])	87.9%	85.7%

Table 2.1: **F1 scores** of BERT and XGBoost for defect-fix commit classification across four datasets. BERT represents the state-of-the-art model by Zafar et al. [24], while XGBoost is our proposed model.

Although BERT often yields superior performance on complex natural language tasks, it demands significantly more computational resources for both training and inference due to its deep transformer-based architecture. In contrast, XGBoost constructs an ensemble of decision trees sequentially, with each tree addressing the errors of its previous ones, and leverages regularization and parallelized execution to mitigate overfitting and speed up training.

Comparison with the state-of-the-art. To benchmark the XGBoost classifier against the BERT-based approach presented by Zafar et al. [24], we trained an XGBoost classifier using the training set proposed by Zafar et al. [24] and evaluated it on the same four test datasets used in their study. Although Zafar et al. [24] primarily reported accuracy along with detailed confusion matrices, we recalculated the F1 score—our metric of choice—to ensure a fair comparison. Since the test datasets were predefined, k-fold cross-validation was not applicable in this case. In Table 2.1, the *BERT* column presents the F1 scores derived from Zafar et al.'s results, while the *XGBoost* column shows the corresponding scores for our classifier. Although BERT outperforms XGBoost by 1–5 percentage points, this difference is acceptable given the substantial efficiency benefits offered by XGBoost. Thus, particularly in CI contexts, XGBoost provides a favorable balance between computational cost and predictive performance. We use this classifier to compute the PDF that is defined above.

2.3 Data Model

In this section, we first propose a data model based on the regression testing of a CI build. We then describe a feature model aimed at addressing the TCP problem with machine learning and show how our data model relates to the features.

Figure 2.2 shows a high-level class diagram of entities and properties that are relevant to the regression testing of a CI build. For each CI build, a set of source code files of the SUT are changed (*INC*). To verify whether or not existing functionalities are impacted by changes, a set of regression test cases are executed (*RUN*), the results of which are captured (*REC*) in the build log. When a test case detects a fault (*DET*), the relevant data is often captured using a fault tracking tool such as Bugzilla. A fault is fixed via changes to the source code (*CHN*) of the SUT and recorded in a code versioning system

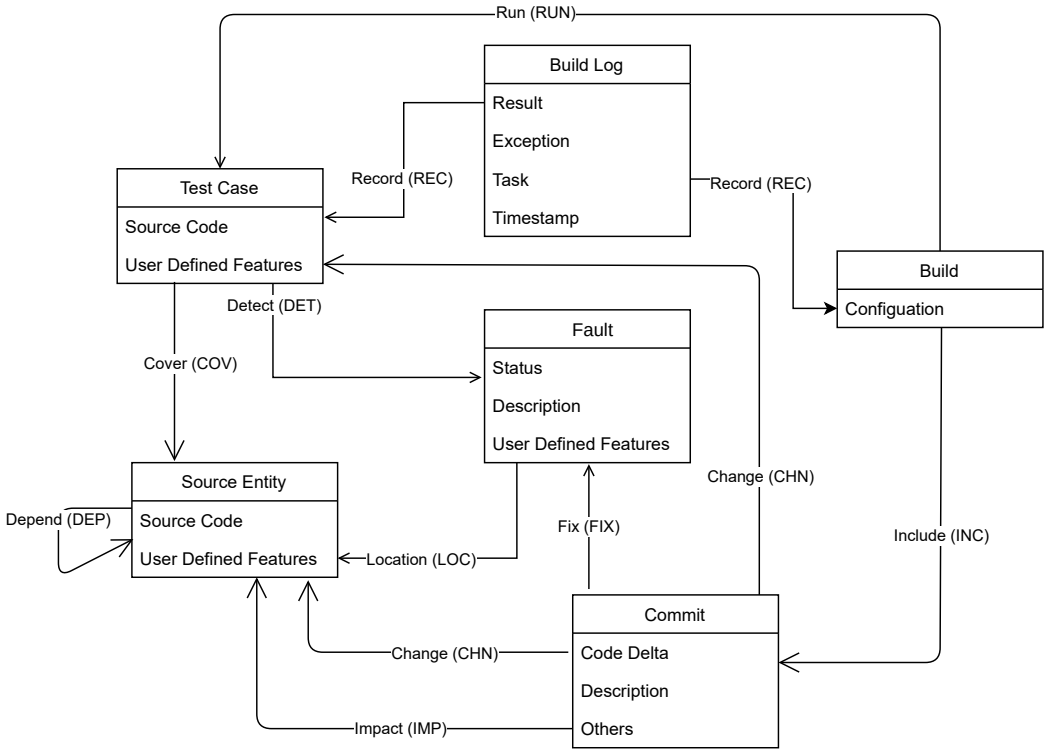

Figure 2.2: High-level Data Model of a CI Build

such as *Git*. A change of source code entity may impact other dependent entities (IMP). Moreover, each test case causes one or more source entities (files or methods) of the SUT to execute (*COV*).

2.3.1 High-level class diagram

The data represented by our data model is not necessarily available in a structured form and can be available as source code, text files, or even binary files. In the following, we define each of the entities and discuss how their corresponding data can be collected.

Definition 3. *Build.* This data source refers to the CI builds that are defined by scripts and configurations. The scripts are written using a Domain Specific Language (DSL) provided by build tools (e.g., Maven). The scripts specify the details (recipe) of the build, often in the form of rules. A build configuration is a collection of settings (e.g., compiler version) that guides how to run the build scripts. For a CI build to be completed (a build task), the build script needs to be executed based on a specific configuration. The build scripts and configurations contain metadata required to understand the build logs. Also, build scripts can be analyzed to extract dependencies between source files.

Definition 4. *Build Log.* This data source refers to the logs of a CI build. Depending on the underlying build technology, logs may contain different details. Despite such differences, existing build tools typically generate logs that contain the build id, timing information,

the result of the build, the output of executed test cases showing whether they passed, and the build configuration (e.g., platform and compiler version). The use of this data requires an analysis of log files that accounts for the specifics of each build tool.

Definition 5. *Test case.* This data source refers to the source code of test cases and sometimes their descriptions and user-defined properties. Such data can be collected via the analysis of the source code of test cases or their descriptions. Source code analysis typically requires static analysis tools (e.g., Understand [35]). Also, natural language processing (NLP) techniques can be applied to extract useful data from source code or test case descriptions.

Definition 6. *Source code.* This data source refers to the source code of the SUT, which can be accessed and analyzed at three levels of granularity: source file, class, method. Similar to the source of test cases, static analysis is required.

Definition 7. *Commits.* This data source refers to all changes to the SUT source code and test cases. A commit captures a change that is applied to a set of source code files. Existing code versioning tools (e.g., Git) provide APIs to access and analyze information about commits.

Definition 8. *Fault.* This data source refers to the information about detected faults in the SUT. Existing tools, such as Bugzilla, enable end-to-end tracking of faults that captures when and why a fault is introduced and how it is fixed. In a regression testing context, it is essential to know (1) if a test case reveals a fault during regression testing (regression fault), and (2) how a regression fault is fixed (i.e., commits). While fault tracking tools are widely used, the quality of faults' data is determined by the process followed by development teams to record all relevant details.

2.3.2 Feature model

An ML-based TCP model takes feature vectors of test cases as input and ranks these test cases. Thus, for any feature to be used for training, it must be a property of test cases. This often requires aggregating and recasting data collected at a different level of granularity. For example, coverage data shows which source files are covered by which test cases, but such data cannot be directly used to define test case features and must be aggregated and recast as a test case property.

We define a comprehensive set of test case features that are grouped into three main groups and nine subgroups, as shown in the class diagram in Figure 2.3. In the following, we discuss why each group is considered to be a potentially useful set of features for TCP and how features are calculated based on our data model. Although most of the features can be calculated at the three different granularity levels (method, class, and file), we only address the file level here because most open-source data sources (e.g., *RTPTorrent* [36] and *TravisTorrent* [37]) reported test case execution history for test files and classes rather than methods. Coverage features are also collected here at the file level as, for practical reasons invoked by our industry partner, we want to rely on scalable, light-weight static

analysis. File-level analysis can however overestimate test case coverage features as each test case file may contain multiple dependent or independent test case methods. A similar analysis and process can be applied for the class and method levels. Also, we use the naming convention *F_name* for features, where *name* refers to a meaningful name that is selected based on the source code metrics and the aggregation function (if any) used to calculate the feature.

Source code metrics that are grouped into three groups (*complexity*, *process*, and *change*) are presented in Figure 2.3 and described in Appendix A.1. *Complexity* metrics measure the complexity of source code as calculated by static analysis. Also, both *process* and *change* metrics concern how the source code has evolved. However, the former is calculated based on the entire change history of the source code, while the latter is only calculated based on the changes of the latest build. Notably, process metrics include human-oriented measurements such as *AllCommittersExperience*, which capture collaborative development dynamics. While our current analysis does not focus on human factors in TCP, this represents an interesting direction for future research.

Data collection for each metric can be done in two sequential steps: preprocessing and measurement. During preprocessing, all required raw data are computed and loaded in memory in a certain format (e.g., Abstract Syntax Tree (AST)) that is adequate for the measurement step, during which the metrics are calculated. For instance, to collect complexity metrics for a source file, the file is parsed during preprocessing and represented as an AST, based on which the metrics are calculated. Note that metrics in each group share the same preprocessing step and, as we discuss in Section 2.4, the cost of the measurement step is significantly less than that of preprocessing. This implies that, in practice, the cost of data collection for a metric in a group is close to that of its entire set of metrics. For fairness, costs associated with auxiliary steps (e.g., cloning Git repositories) are excluded from the metric data collection cost.

To facilitate the definition of features below, let us define functions *chn* and *imp* which take a build b as input and return two disjoint sets of strictly changed and impacted source code files, respectively. The latter correspond to files that may be affected by changes in other files though they were not changed themselves. Results of functions *chn* and *imp* correspond to relations CHN and IMP in the data model (Figure 2.2).

2.3.2.1 Test Case Source Code Features (TES)

These features are calculated based on the complexity, process, and change metrics (Figure 2.3) of the source code files of test cases. The source code features of a test case simply correspond to the metrics of the test case's source file. More details on the definition of metrics can be found in the Appendix A.1. Since this feature group is defined based on source code metrics, we categorize the features in this group into three subgroups, namely TES_COM, TES_PRO, and TES_CHN that correspond to *complexity*, *process*, and *change* metrics, respectively.

The main motivation for using TES features is based on the hypothesis that more complex test cases tend to be associated with longer execution times and a higher probability

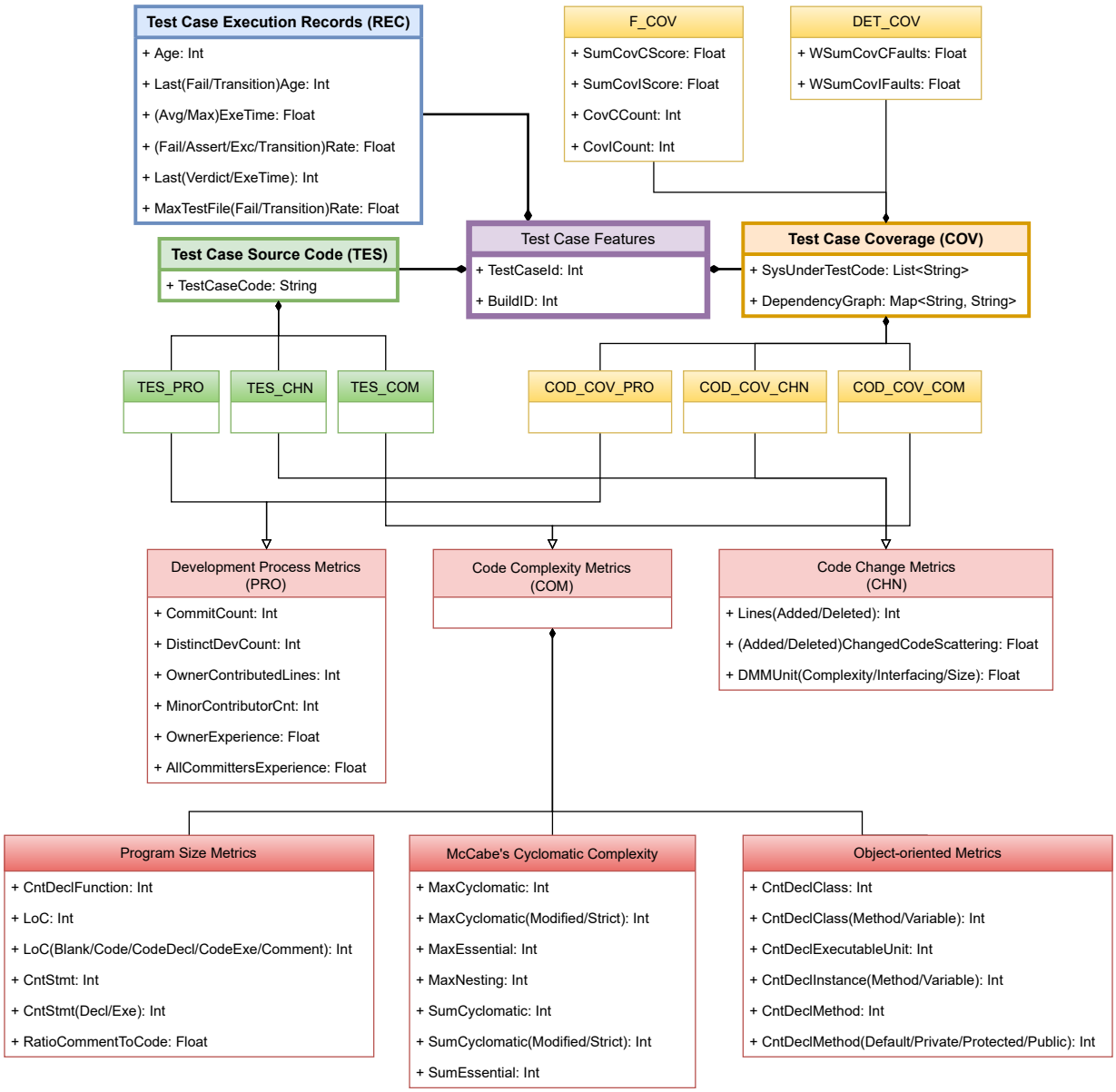

Figure 2.3: A comprehensive class diagram depicting our proposed feature group hierarchy (three high-level groups with nine low-level groups) and the relation between the feature groups and the metrics used for computing them.

of detecting faults. Indeed, such test cases are more likely to cover more of the SUT source code and thus have a higher probability of fault detection. Such data are collected through the static analysis of the source code of test cases, which is supported by several tools such as Understand [35], and corresponds to the *Test Case* entity in the data model (Figure 2.2).

2.3.2.2 Test Case Execution Record Features (REC)

We define test case features which are calculated based on previous execution time records and verdicts (i.e., failed or passed) of the test cases. These features correspond to relation REC between *Test case* and *Build Log* entities in the data model (Figure 2.2).

- **F_Age:** This feature captures the number of builds from the first execution of the test case (its introduction). Assuming test case t was first executed on build i, the age of the test case at build n will be equal to $n-i$. We adopt this feature from previous work [38] which reports that newer test cases fail more often since they exercise new and possibly changed source code.
- **F_LastFailAge:** This feature refers to the number of builds from the last failure of the test case [39]. Assuming that the latest failure of test case t occurred on build i, *F_LastFailAge* of t for build n is equal to $n-i-1$. *F_LastFailAge* of a test case that has never failed is set to -1 rather than 0, the latter being used for a test case that has failed in the previous build.
- **F_LastTransitionAge** [39]: This feature refers to the number of builds since the last change (transition) in the test case's verdict, from failed to passed or vice versa. It is computed in the same way as *F_LastFailAge* but based on the last transition.
- **F_AvgExeTime:** This feature refers to the average of the previous execution time records of the test case.
- **F_MaxExeTime:** This feature refers to the maximum value of execution time records of the test case.
- **F_FailRate:** This feature is defined as $\frac{f}{n}$, where f is the number of failed executions of the test case and n is the total number of executions of the test case. In this work, we use this rate rather than the failure count of a test case since the latter can be misleading as it is very much dependent on the number of times a test case was executed.
- **F_AssertRate:** An assertion failure of a test case refers to a failure due to an unexpected output of the SUT. This feature refers to the rate of assertion failures of the test case.
- **F_ExcRate:** An exception failure is caused by an exception that is not handled correctly in the source code of the test case. This feature refers to the rate of exception failure of a test case. An exception failure may indicate a fault in the source code of

t rather than that of the SUT. Thus, we distinguish assertion failures from exception failures.

- **F_TransitionRate** [39]: This feature refers to the rate of transitions of the test case verdicts.
- **F_LastVerdict:** This feature captures the verdict of the last execution of a test case.
- **F_LastExeTime:** This feature refers to the last execution time of a test case.
- **F_MaxTestFileFailRate** [39]: Assuming that $\mathrm{TFF}(t, f, k)$ refers to the number of builds before build k, in which the file f is changed and test case t has failed, and that $\mathrm{TF}(t, k)$ refers to the total number of builds before build k in which test case t has failed, *F_MaxTestFileFailRate* for test case t in build k with a set of changed files F is calculated as:

$$\frac{\text{Maximum of } \mathrm{TFF}(t, f, k), f \in F}{\mathrm{TF}(t, k)}$$

  If t has never failed, the feature is set to -1 rather than 0, the latter being used when no failure of t co-occurred with changes of the files that are changed in build k. The use of this feature is motivated by the fact that, if the previous changes in a file are associated with previous failures of a test case, then the future changes of the file are likely to be associated with failures of the test case.

- **F_MaxTestFileTransitionRate** [39]: This feature is exactly computed the same way as *F_MaxTestFileFailRate* except that it accounts for test case transitions instead of failures.

Due to the frequent execution of regression test cases in CI contexts, the volume of execution history is continuously and quickly growing. Thus, *REC* features such as *F_AvgExeTime*, *F_MaxExeTime*, *F_FailRate*, *F_AssertRate*, *F_ExcRate*, and *F_TransitionRate* are typically calculated based on the latest n test case executions, which are extracted by processing the logs of the latest n builds. Since the main goal of our work is to use a comprehensive set of features, we calculate two values for these six features in our final feature set: *Recent* and *Total*. The *Recent* value is computed based on the latest six builds, similar to previous studies [14,20], while the *Total* value is calculated based on all available builds. Previous work [14] reports that using long text execution history may lead to a reduction in performance.

The primary motivation for using *REC* features related to fault detection (i.e. *F_FailRate*, *F_AssertRate*, *F_ExcRate*, *F_TransitionRate*, and *F_LastVerdict*) is that test cases that detected more faults in the past are more likely to detect faults again in the future, as they tend to exercise complex and frequently changed features. Thus, we conjecture that past failed test cases should be executed again in new builds. Additionally, the hypothesis behind using execution time history is that test cases that take more time to run are more likely to execute more complex and compute-intensive code, as well as larger parts of the

code. Therefore, long-running test cases are more likely to detect faults in the SUT. Also, since the execution time of test cases is used as the second criteria for prioritizing test cases (when two test cases have the same verdict, the one with the lower execution time is ranked first), it is an important feature for training ML-based TCP techniques.

2.3.2.3 Test Case File Coverage Features (F_COV)

These features are calculated based on source files covered (exercised) by the test case and correspond to relation COV in the data model (Figure 2.2). These features capture the ability of test cases to cover source files that are changed (relation CHN) or impacted (relation IMP) in the build.

Let us define function *cov* that takes a source file and a test case as inputs and returns whether the test case covers (exercises) the source file. Also, let us define function *cov_score* that takes the same inputs and returns the normalized coverage score if the test case covers the source file, and zero otherwise. Since we are eventually going to use coverage scores for weighted summations in builds, we need to use normalized scores. Given a set of coverage scores $S = \{s_1, ..., s_n\}$ (e.g., coverage scores of all changed files covered by a test case in a build), we normalize them by dividing each score by the sum of all coverage scores. Hence, the normalized coverage score set, SN, is defined as follows.

$$SN = \{\frac{s_1}{\sum_{i=1}^{n} s_i}, ..., \frac{s_n}{\sum_{i=1}^{n} s_i}\}$$

The four features in this group refer to the number of covered files and coverage score as defined in the following.

- **F_SumCovCScore** of a test case t in build b refers to the sum of coverage scores of t w.r.t the changed source files in build b: $\sum_{f \in \text{chn}(b)} \text{cov\_score}(f, t)$.
- **F_SumCovIScore** of a test case t in build b refers to the sum of coverage scores of t w.r.t the impacted source files in build b: $\sum_{f \in \text{imp}(b)} \text{cov\_score}(f, t)$.
- **F_CovCCount** of a test case t in build b refers to the number of covered source files by t that are changed in build b: $|\{f : f \in \text{chn}(b) \wedge \text{cov}(f, t)\}|$.
- **F_CovICount** of a test case t in build b refers to the number of covered source files by t that are impacted in build b: $|\{f : f \in \text{imp}(b) \wedge \text{cov}(f, t)\}|$.

Impact analysis is relatively time-consuming and therefore separation of the features based on impacted and changed source files allows us to investigate whether or not including impacted files brings significant benefits. Using such coverage scores as features is motivated by the hypothesis that a test case with higher coverage is more likely to exercise faults, and is, therefore, more likely to detect them.

2.3.2.4 Features of Covered Source Code by Test cases (COD_COV)

This group includes the source code features of the changed and impacted source files (*Source Entity* in the data model in Figure 2.2) covered by the test cases. These features are calculated based on the complexity, process, and change metrics in Figure 2.3. Similar to the TES feature group, we categorize features of this group into three subgroups which are COD_COV_COM, COD_COV_PRO, and COD_COV_CHN, which correspond to *complexity*, *process*, and *change* metrics, respectively.

Since most test cases cover more than one source code entity, we use the normalized weighted sum of metrics based on the coverage score of the test case w.r.t to source files. Also, similar to F_COV features, we define separate features for changed and impacted source files. For example, the weighted summation of the *LoC* features of covered files of test case t in build b is calculated as follows, assuming that function $\text{loc}(f)$ computes *LoC* for source file f,

$$\text{F\_WSumCLoC} = \sum_{f \in \text{chn}(b)} \text{cov\_score}(f, t) * \text{loc}(f)$$

$$\text{F\_WSumILoC} = \sum_{f \in \text{imp}(b)} \text{cov\_score}(f, t) * \text{loc}(f)$$

where *F_WSumCLoC* and *F_WSumILoC* refer to the features based on changed and impacted files, respectively. As discussed above, other metrics can replace $\text{loc}(f)$ in the above equations.

The main motivation for using COD_COV features based on complexity metrics is due to the hypothesis that the cumulative complexity of covered source files by a test case is an indicator of the execution time and fault revealing power of the test case. Also, change metrics, specifically delta maintainability metrics (DMM) [40], assess the maintainability implications of changes by estimating the risk level of each change. Thus, features defined based on maintainability metrics are good indicators of risky changes, i.e., changes that are likely to be faulty. The same argument applies to features based on process metrics since they indicate the risk entailed by changes by relying on metrics that concern the development process. For example, a change by a new developer has a higher probability of being faulty than that of an experienced developer. Thus, the execution of test cases that cover source files including higher risk changes in the current build has a higher fault detection probability in the context of regression testing.

2.3.2.5 Test Case Coverage Fault Detection Features (DET_COV)

These features are defined based on the Previously Detected Faults (PDF, Definition 2) of source files that is captured by relations DET and LOC in the data model (Figure 2.2). We define the following features in this feature group:

- **F_WSumCovCFaults** refers to the weighted sum (weighted by coverage scores) of PDFs of the changed source files covered by test case t in build b.

$$\sum_{f \in \text{chn}(b)} \text{PDF}(f) * \text{cov\_score}(f, t)$$

- **F_WSumCovIFaults** refers to the weighted sum (weighted by coverage scores) of PDFs of the impacted source files covered by test case t in build b.

$$\sum_{f \in \text{imp}(b)} \text{PDF}(f) * \text{cov\_score}(f, t)$$

The main motivation for using DET_COV features is based on the hypothesis that a source file with a higher PDF is more likely to contain faults in the future. Similarly, a test case that covers files with higher PDF is more likely to detect faults.

2.4 Validation

This section reports on the experiments we conducted to assess the impact of features on the effectiveness and cost of TCP techniques. We first discuss and motivate four research questions. We then describe the subjects of the study and the evaluation metrics. Finally, we explain our experimental process, present the results, and discuss their practical implications.

2.4.1 Research Questions

2.4.1.1 RQ1. Data Collection Time

- **RQ1.1** How does data collection time across feature groups compare and are they significantly different?
- **RQ1.2** How does accounting for impacted files affect data collection time for each feature group?
- **RQ1.3** How do subject size (Source Lines of Code), the number of test cases, and the number of builds affect the data collection time?

In a CI context, data collection time is a particularly sensitive issue as time is usually strictly limited for regression testing. A feature can be used to train an ML-based TCP if it can be collected within much less time than that of regression test execution. Thus, we define RQ1 to investigate the data collection time of each feature group according to two modes, based on whether or not impacted files are considered (RQ1.1 and RQ1.2). Also, RQ1.3 investigates how the size of subjects in terms of SLOC (Source Lines of Code), the number of test cases, and the number of builds affect data collection times. In particular,

RQ1.3 investigates how data collection time increases as the size of the subject grows and whether scalability issues can be expected for large systems when collecting features. This RQ focuses on feature groups since all features in each group rely on the same preprocessing, which accounts for most of the data collection time.

2.4.1.2 RQ2. TCP Effectiveness

- **RQ2.1** How effective is test case prioritization when using the full feature set?
- **RQ2.2** How effective is test case prioritization when impacted files are not considered?
- **RQ2.3** How does the use of each feature group contribute to the effectiveness of the ML-based TCP?
- **RQ2.4** Which specific feature or subset of features has the highest impact on the effectiveness of ML-based TCP models?
- **RQ2.5** How does the effectiveness of heuristic-based TCP models compare to ML-based models?

RQ2.1-2.4 are particularly important in the context of CI as the features used for ML-based TCP models must be minimized, especially features that induce high data collection time, without significantly improving TCP effectiveness. Therefore, the goal is to investigate what TCP effectiveness can be achieved with all features (RQ2.1 and RQ2.2) and what feature groups are most important for TCP effectiveness (RQ2.3). Though the analysis in RQ2.3 is based on feature groups, within feature groups, even those with low impacts, some individual features may have a higher impact on effectiveness than others. This is addressed by RQ2.4. Finally, RQ2.5 compares the results of ML-based TCP with heuristic-based TCP to investigate whether or not using ML brings significant advantages that justify its use.

2.4.1.3 RQ3. How often do the ML-based TCP models need to be retrained?

ML-based TCP models, specifically in the context of CI, need to be retrained regularly based on newly collected data, to better reflect the current status of the system and its history. However, retraining can be expensive and incur delays, and thus we should investigate how the effectiveness of ML-based TCP models decays over time when features are not updated and the models not retrained. This analysis will provide us with insights on how often feature data needs to be collected and used for retraining ML-based TCP.

2.4.1.4 RQ4. Trade-off between data collection time and TCP effectiveness for features

RQ1 and RQ2 separately investigate the data collection time and effectiveness of the TCP model based on individual feature groups. For engineers to make informed decisions in specific contexts, regarding which feature groups should be used for ML-based TCP, a trade-off often needs to be made between data collection times and effectiveness. Thus, RQ4 conducts a comprehensive trade-off analysis with the objective of providing concrete guidelines regarding the use of feature groups, that will hopefully lead to acceptable effectiveness within reasonable time for a specific context.

2.4.2 Subjects

We ran our experiments on 25 subjects, which were selected in a 6-step process, as discussed in the following.

1. We started with the latest available database of open-source projects from *GHTorrent* [41] (dump 2021-03-06 with more than 100 million projects). We then selected active (i.e., not forked and deleted) and popular (with at least 50 stars) Java projects from the database that resulted in 22,551 projects. *GHTorrent* provides regularly updated databases of GitHub open-source repositories along with tooling to search in the databases of active (i.e., not forked or deleted) projects in GitHub.

2. We selected projects with at least 100 CI runs hosted on Travis CI that resulted in 3,323 projects. We then used *TravisTorrent* [37] to fetch CI build logs and commits of the selected open-source repositories. *TravisTorrent* provides scripts for fetching data from Travis CI and GitHub (for build commits).

3. We selected projects that use *Maven* as their test execution tool since it provides build logs that contain the required information regarding test case executions (i.e., verdict, duration, and class name). This further reduced the number of projects to 1,419.

4. From the resulting projects in step 3, we selected the union of the top 300 projects with the highest SLOC (Source Lines of Code) and the top 300 projects with the highest number of builds. This resulted in 434 projects.

5. We used *TravisTorrent* [37] to extract the build logs and build commits of the projects resulting from step 4. We analyzed the build logs to calculate the regression testing duration and failed builds of projects. We then selected projects with at least 5 minutes of average regression testing time and 50 failed builds, which resulted in 18 projects. There is no or little practical value in performing TCP when the regression testing time is less than 5 minutes, as most TCP techniques often require more than a few minutes for the data collection and prioritization of test cases. Also, we require at least 50 failed builds to create sufficiently balanced datasets.

S_{ID}	Subject	SLOC	JSLOC	Commits	Span	Builds	F_Builds	FR %	TC/Build	Duration
S_1	JMRI/JMRI	4.56M	1.05M	69.3k	5mo	1,481	65	4	4364	25min
S_2	apache/airavata	1.46M	731k	10.0k	15mo	236	83	35	49	6min
S_3	SonarSource/sonarqube	899k	398k	31.8k	18mo	4,286	230	5	1309	6min
S_4	apache/sling	695k	388k	47.4k	7mo	1,403	343	24	189	6min
S_5	camunda/camunda-bpm-platform	653k	395k	20.6k	34mo	822	125	15	569	23min
S_6	facebook/buck	586k	384k	26.3k	10mo	846	130	15	663	26min
S_7	apache/shardingsphere	422k	165k	29.6k	7mo	1,049	123	11	789	17min
S_8	b2ihealthcare/snow-owl	373k	212k	13.4k	2mo	277	21	7	46	10min
S_9	Angel-ML/angel	336k	204k	3.0k	23mo	308	124	40	33	20min
S_{10}	apache/logging-log4j2	313k	166k	12.7k	13mo	441	122	27	544	8min
S_{11}	eclipse/jetty.project	282k	199k	25.0k	2mo	192	89	46	137	6min
S_{12}	optimatika/ojAlgo	246k	84k	1.6k	22mo	254	72	28	136	9min
S_{13}	yamcs/Yamcs	229k	123k	5.6k	24mo	504	61	12	114	6min
S_{14}	eclipse/steady	221k	98k	2.0k	13mo	675	51	7	81	7min
S_{15}	Graylog2/graylog2-server	182k	85k	22.3k	53mo	3,668	124	3	110	20min
S_{16}	CompEvol/beast2	159k	83k	3.0k	85mo	415	115	27	65	6min
S_{17}	EMResearch/EvoMaster	158k	25k	4.1k	7mo	583	109	18	100	12min
S_{18}	apache/rocketmq	135k	100k	2.0k	16mo	536	56	10	182	17min
S_{19}	zolyfarkas/spf4j	125k	79k	32.6k	37mo	587	180	30	116	7min
S_{20}	spring-cloud/spring-cloud-dataflow	104k	54k	3.5k	9mo	408	19	4	115	17min
S_{21}	cantaloupe-project/cantaloupe	98k	77k	4.5k	29mo	450	65	14	148	11min
S_{22}	thinkaurelius/titan	85k	40k	5.1k	25mo	384	41	10	45	48min
S_{23}	apache/curator	84k	58k	3.1k	21mo	517	65	12	115	67min
S_{24}	jcabi/jcabi-github	61k	32k	2.8k	29mo	788	6	< 0.01	176	14min
S_{25}	eclipse/paho.mqtt.java	61k	34k	1.0k	16mo	378	77	20	37	15min

Table 2.2: The statistics of the 25 carefully selected subjects. The statistics include source lines of code (SLOC), Java SLOC (JSLOC), number of commits, length of the time period between the first and last build in months (Span), number of all builds and failed builds (F_Builds) with the failure rate (FR), average number of test cases per build (TC/Build), and average execution time of all test cases per build in minutes (Duration).

6. We selected another 7 projects with at least 5 minutes of average regression testing time and 50 failed builds from the 20 open-source projects provided by RTPTorrent's [36], a public dataset for the evaluation of TCP techniques. Thus, overall we selected 25 projects as the subjects of our studies that are shown in Table 2.2.

We investigated the failure frequency of test cases among failed builds. For most of the subjects, some of the test cases failed frequently across failed builds. We then investigated the reasons for the existence of such test cases by going through the build logs of a sample of the subjects and reading the error messages caused by test case failures. In the investigation sample, which included S_{24}, S_{20}, S_8, and S_7, we found test cases that failed in more than 65% of the failed builds due to the same reason. Such reasons included external exceptions which were not related to the SUT, such as errors in HTTP requests to external APIs due to authentication issues, invalid arguments, or unexpected responses. Common failure causes also included Java runtime errors, such as *ClassNotFoundException* or *FileNotFoundException*, which were due to missing classes or missing files in the project. Executing such frequently-failing (FF) test cases, which are also referred to as *known breakages*, has no practical value in regression testing as they tend to fail most of the time for the same reason, independently of changes. Since our focus is regression testing, we removed these test cases by performing outlier tests, with respect to failure frequency, using the three-sigma rule of thumb [42]. As suggested by our analysis, outlier test cases are highly likely to correspond to non-regression failures and therefore tend to blur our empirical results. Note that though such outlier tests may not remove all of FF test cases, we are confident that most of them were identified and excluded from our datasets.

Hence, the subject statistics, experiments, and results presented throughout the rest of the chapter are based on data in which the *FF* test cases are removed.

The final 25 subjects were selected by analyzing more than 20,000 popular open-source Java projects. The selection process assures that all subjects have an acceptable number of failed test cases and regression testing time, both of which are critical for the application and evaluation of TCP techniques. Table 2.2 shows the characteristics of the subjects in terms of line of codes, the number of (failed) builds, failure rate, commits, and the average regression testing time per build. The first column (S_{ID}) of the table shows the identifier of subjects that will be used to refer to them in the rest of this section.

Column *SLOC* of Table 2.2 shows the total number of code and comment lines of the subjects that were counted based on the latest build. *SLOC* ranges from 61k to 4.56M, with a median of 229k. Compared with the subjects that are used in most recent studies, our work relies on a high number of subjects (25) whose median SLOC is 229k compared to 37.4k [20] and 132k [39], respectively. Also, the average number of test cases per build across our subjects ranges from 33 to 4368, with a median of 117, which is similar to 18 previous studies reported by [7]. Thus, compared to the previous studies, we use a higher number of subjects whose size in terms of SLOC can be considered to be reasonably larger. A build can fail for several reasons, including compilation errors or a test case failure. However, in a TCP context, we are only interested in the latter, and failed builds are, in our subjects, builds with at least one failed test case. Column *# Failed Builds* of Table 2.2 shows the number of failed builds of each subject. This number ranges from 6 to 343, with a median of 83, representing a diverse set of subjects allowing us to conduct a large number of experiments to account for randomness and draw statistically valid conclusions. Pan et al. [7] reports that previous studies evaluate their work mainly based on subjects with a low number of failed builds that ranges between 1 and 70 with a median of 9, which results in unbalanced training datasets. Also, it is not meaningful to evaluate TCP techniques based on a subject with a very low number of failed builds as the main goal of TCP is the early detection of faults and most of the evaluation metrics are based on counting detected faults.

The last column of Table 2.2 shows the average of regression testing time per build across all subjects that ranges from 6 to 67 minutes with a median of 12. Compared to recent studies where 11 out of 23 subjects [39] have regression testing times below 3 minutes, or all subjects have regression testing times below 30 seconds [20], our subjects are significantly better. Recall that it is not meaningful to apply and evaluate TCP techniques in the context of projects whose regression testing times are small.

As discussed earlier, we removed the *FF* test cases from the subjects using the outlier test. Table 2.3 compares the statistics of subjects, which have at least one *FF* test case, before and after removing their *FF* test cases. Column *# FF TCs* corresponds to the number of *FF* test cases, and the rest of the columns show the statistics before (B) and after (A) removing all FF test cases. As visible, the number of failed builds for some subjects significantly drops when the FF test cases are removed. More specifically, for four of the subjects ($S_{8,20,22,24}$), the number of failed builds decreases to under 50 (our selection criterion) after removing *FF* test cases. This suggests that *FF* test cases were the main

S_{ID}	FF	F_Builds		FR %		TC/Build		Duration (min)	
		B	A	B	A	B	A	B	A
S_5	8	174	125	21	15	575	569	24	23
S_3	7	299	230	6	5	1315	1309	7	6
S_7	6	151	123	14	11	795	789	17	17
S_1	5	94	65	6	4	4368	4364	26	25
S_{17}	2	143	109	24	18	101	100	13	12
S_{21}	2	70	65	15	14	149	148	12	11
$\mathbf{S_{22}}$	2	60	**41**	15	10	45	45	48	48
$\mathbf{S_{24}}$	2	76	**6**	9	< 0.01	174	176	13	14
S_4	1	697	343	49	24	189	189	6	6
$\mathbf{S_8}$	1	58	**21**	20	7	47	46	22	10
S_{10}	1	231	122	52	27	545	544	8	8
S_{11}	1	150	89	78	46	138	137	6	6
S_{13}	1	72	61	14	12	115	114	7	6
S_{19}	1	277	180	47	30	117	116	7	7
$\mathbf{S_{20}}$	1	88	**19**	21	4	115	115	17	17
S_{23}	1	103	65	19	12	115	115	68	67

Table 2.3: The number of frequent-failing test cases (FF) and subject statistics before (B) and after (A) removing the FF. The statistics align with those in Table 2.2. Subjects with no FF are excluded. The subjects that have less than 50 failed builds after removing FF test cases are shown in bold.

cause of build failures across these subjects. However, the effect of removing *FF* test cases on the average number of test cases per build and the average regression testing time per build is negligible.

2.4.3 Evaluation Metrics

In this work, we use Cost-cognizant Average Percentage of Faults Detected (APFD_C) [43] as the evaluation metric to measure the effectiveness of prioritization techniques. APFD_C is a cost-aware variant of the well-known and widely used APFD [44] metric. APFD only measures the extent to which a certain ranking reveals faults early and does not take the execution time of test cases into account, which is important, especially in a CI context. Similar to prior work [39, 45, 46], since fault severity information is not available, we assume all faults have the same severity. Also, since our collected data does not include the mapping of faults and test cases, similar to prior work [39, 45, 46], we assume that each test case failure refers to a distinct fault in the SUT. In practice, obviously, this widely used assumption is not correct. However, we can expect the number of faults detected to be roughly proportional to the number of failures.

APFD_C of a test case ordering T^* is calculated as:

$$\text{APFD}_C = \frac{\sum_{i=1}^{m} \left(\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i} \right)}{\sum_{j=1}^{n} t_j \times m}$$

where m refers to the total number of faults, n refers to the total number of test cases in T^*, and TF_i refers to the position (starting from 1) of the first failed test case in T^* that

	COD_COV_COM			COD_COV_PRO			DET_COV			COD_COV_CHN			F_COV			TES_COM			TES_PRO			REC			TES_CHN		
S_{ID}	P	M	T	P	M	T	P	M	T	P	M	T	P	M	T	P	M	T	P	M	T	P	M	T	P	M	T
S_1	**508.9**	**0.4**	**509.3**	**455.8**	**5.5**	**461.3**	**452.8**	**4.1**	**456.9**	**451.4**	6.1	**457.5**	**451.4**	0.0	**451.4**	**57.5**	**0.1**	**57.6**	4.4	**28.9**	**33.3**	**0.4**	**139.1**	**139.4**	0.0	**5.6**	**5.6**
S_2	61.6	0.0	61.6	23.9	0.2	24.2	23.6	0.1	23.7	23.3	2.2	25.5	23.3	0.0	23.3	38.3	0.0	38.3	0.6	0.4	1.0	0.0	0.5	0.5	0.0	0.4	0.4
S_3	262.7	0.1	262.8	244.2	0.8	245.0	244.1	0.5	244.6	243.8	4.5	248.4	243.8	0.0	243.8	18.9	0.0	18.9	0.4	5.1	5.5	0.1	21.9	22.0	0.0	4.2	4.2
S_4	98.6	0.0	98.6	74.3	0.2	74.5	74.1	0.1	74.3	74.1	1.1	75.2	74.1	0.0	74.1	24.6	0.0	24.6	0.2	0.9	1.2	0.0	1.3	1.4	0.0	1.1	1.1
S_5	130.0	0.1	130.1	107.9	0.7	108.6	107.7	0.4	108.1	106.8	1.3	108.1	106.8	0.0	106.8	23.2	0.0	23.2	1.1	3.6	4.7	0.1	6.6	6.7	0.0	1.2	1.2
S_6	69.7	0.1	69.9	49.4	0.7	50.1	49.2	0.3	49.5	48.9	1.6	50.5	48.9	0.0	48.9	20.9	0.0	20.9	0.5	3.3	3.8	0.1	5.6	5.7	0.0	1.5	1.5
S_7	110.3	0.1	110.3	93.8	0.6	94.3	93.2	0.3	93.5	92.5	1.6	94.1	92.5	0.0	92.5	17.7	0.0	17.8	1.2	3.3	4.5	0.1	6.2	6.3	0.0	1.5	1.5
S_8	47.5	0.0	47.5	33.5	0.3	33.8	32.8	0.2	33.0	31.2	0.7	31.9	31.2	0.0	31.2	16.2	0.0	16.2	2.3	0.5	2.8	0.0	0.6	0.6	0.0	0.6	0.6
S_9	29.4	0.0	29.4	21.0	0.1	21.1	20.8	0.1	20.9	19.6	0.3	20.0	19.6	0.0	19.6	9.8	0.0	9.8	1.4	0.2	1.6	0.0	0.3	0.3	0.0	0.3	0.3
S_{10}	41.0	0.0	41.1	30.2	0.2	30.4	30.1	0.1	30.1	29.8	0.5	30.2	29.8	0.0	29.8	11.2	0.0	11.3	0.5	2.5	3.0	0.1	3.7	3.8	0.0	0.4	0.4
S_{11}	65.3	0.0	65.3	52.2	1.1	53.2	52.0	0.8	52.9	51.6	0.6	52.3	51.6	0.0	51.6	13.7	0.0	13.7	0.5	1.7	2.2	0.1	1.4	1.4	0.0	0.6	0.6
S_{12}	22.4	0.0	22.4	16.2	0.3	16.6	16.0	0.1	16.1	14.3	0.9	15.1	14.3	0.0	14.3	8.1	0.0	8.1	2.0	0.8	2.8	0.0	1.1	1.1	0.0	0.5	0.5
S_{13}	59.0	0.0	59.0	16.8	0.3	17.1	16.4	0.1	16.5	14.9	**10.4**	25.3	14.9	0.0	14.9	44.1	0.0	44.1	1.9	0.7	2.6	0.0	1.0	1.1	0.0	1.0	1.0
S_{14}	70.5	0.0	70.5	59.4	0.2	59.6	59.0	0.0	59.1	50.3	0.6	50.9	50.3	0.0	50.3	20.2	0.0	20.2	9.1	0.5	9.5	0.1	0.6	0.6	0.0	0.4	0.4
S_{15}	7.1	0.0	7.1	4.5	0.2	4.7	4.4	0.2	4.6	4.2	0.8	5.0	4.2	0.0	4.2	2.9	0.0	2.9	0.2	0.4	0.7	0.0	0.4	0.5	0.0	0.8	0.8
S_{16}	12.9	0.0	12.9	6.3	0.1	6.4	6.2	0.0	6.3	6.2	0.2	6.4	6.2	0.0	6.2	6.7	0.0	6.7	0.2	0.4	0.6	0.0	0.6	0.6	0.0	0.1	0.1
S_{17}	6.0	0.0	6.0	2.8	0.1	2.9	2.7	0.0	2.8	2.5	0.2	2.7	2.5	0.0	2.5	3.5	0.0	3.5	0.3	0.5	0.8	0.0	1.1	1.1	0.0	0.2	0.2
S_{18}	29.8	0.0	29.8	18.4	0.3	18.7	18.3	0.0	18.3	17.7	0.4	18.1	17.7	0.0	17.7	12.1	0.0	12.1	0.7	0.9	1.6	0.1	1.3	1.4	0.0	0.3	0.3
S_{19}	21.1	0.0	21.1	13.6	0.7	14.3	12.6	0.5	13.1	12.5	1.6	14.0	12.5	0.0	12.5	8.6	0.0	8.6	1.2	1.1	2.3	0.0	1.6	1.6	0.0	1.5	1.5
S_{20}	23.9	0.0	23.9	29.0	0.2	29.2	27.3	0.1	27.3	16.9	3.6	20.5	16.9	0.0	16.9	7.0	0.0	7.0	**12.1**	0.6	12.7	0.0	1.0	1.0	0.0	3.5	3.5
S_{21}	10.1	0.0	10.1	4.5	0.2	4.7	4.4	0.0	4.4	4.2	0.3	4.6	4.2	0.0	4.3	5.9	0.0	5.9	0.2	0.8	1.0	0.0	1.3	1.3	0.0	0.3	0.3
S_{22}	10.6	0.0	10.7	8.7	0.3	9.0	8.4	0.2	8.6	6.0	0.5	6.6	6.0	0.0	6.0	4.6	0.0	4.6	2.7	0.5	3.2	0.0	0.6	0.6	0.0	0.5	0.5
S_{23}	10.3	0.0	10.3	5.4	0.1	5.5	5.3	0.0	5.4	5.3	0.1	5.4	5.3	0.0	5.3	5.0	0.0	5.1	0.1	0.5	0.6	0.0	0.7	0.7	0.0	0.1	0.1
S_{24}	7.3	0.0	7.3	5.2	0.1	5.3	5.1	0.1	5.2	5.0	0.2	5.1	5.0	0.0	5.0	2.3	0.0	2.3	0.2	0.7	1.0	0.0	1.3	1.3	0.0	0.1	0.1
S_{25}	6.7	0.0	6.7	3.2	0.1	3.3	3.1	0.0	3.1	2.8	0.1	2.9	2.8	0.0	2.8	3.8	0.0	3.8	0.3	0.2	0.6	0.0	0.3	0.4	0.0	0.1	0.1
Avg	83.9	0.0	84.0	68.5	0.5	69.0	68.1	0.3	68.4	67.3	1.6	68.9	67.3	0.0	67.3	16.6	0.0	16.6	1.2	2.3	3.5	0.1	7.2	7.3	0.0	1.3	1.3

Table 2.4: Average preprocessing (P), measurement (M), and total (T) data collection time (in seconds) for all feature groups across subjects. For each column, the maximum value is shown in bold.

detects the ith fault, and t_j refers to the execution time of the jth test.

Both APFD and APFD$_C$ can only be computed for builds that contain failures. Thus, here we only report APFD$_C$ based on failed builds.

2.4.4 Experiment Design, Results, and Discussion

2.4.4.1 Data collection Time (RQ1)

Overview Table 2.4 reports the preprocessing (sub-column P) and measurement (sub-column M) times of feature groups across subjects. For computing each test case feature in a CI build, a number of preprocessing steps are required including static source code and dependency analysis, source code change history collection, and text classification. The preprocessed data is used for one or more feature groups. The measurement time refers to the computation of feature values using the preprocessed data. Most of the measurement times, for all feature groups, are less than a second and therefore negligible. In contrast, preprocessing is expensive, taking for many feature groups almost all the data collection time. Such groups require static source code and dependency analysis as well as textual classification. Therefore, in most production environments where systems tend to be as large or larger than the largest system we consider here (S_1), it might be more practical to do the preprocessing periodically rather than on each build, as we will discuss in RQ3. For this RQ, however, we performed the preprocessing for all builds, to account for the worst-case situation in terms of data collection time.

Also, Table 2.4 shows the average data collection times of feature groups for all builds, across subjects. The first result of practical importance is that the data collection times of certain feature groups are significantly higher than others: COD_COV_COM, COD_COV_PRO, DET_COV, COD_COV_CHN, and F_COV with total data collection time averages of 84, 69, 68.4, 68.9, and 67.3 seconds per build, respectively, with a maximum value (S_1) above 450 seconds. This can be explained by the fact that these feature groups require

S_{ID}	Testing	Collection	C/T %
S_1	26.6	**11.7**	44
S_2	6.0	1.1	18
S_3	7.0	5.0	**71**
S_4	6.9	1.7	25
S_5	24.7	2.4	10
S_6	26.3	1.4	5
S_7	17.6	2.1	12
S_8	22.4	0.9	4
S_9	20.7	0.5	3
S_{10}	8.1	0.8	10
S_{11}	6.6	1.2	18
S_{12}	9.5	0.5	5
S_{13}	7.5	1.2	16
S_{14}	7.7	1.4	18
S_{15}	20.7	0.2	1
S_{16}	6.3	0.2	4
S_{17}	13.3	0.1	1
S_{18}	17.6	0.6	3
S_{19}	7.1	0.5	7
S_{20}	17.7	0.7	4
S_{21}	12.3	0.2	2
S_{22}	48.8	0.3	1
S_{23}	**68.1**	0.2	< 1
S_{24}	13.7	0.2	1
S_{25}	15.5	0.1	1

Table 2.5: Average execution time of all test cases per build (*Testing*) compared to the average data collection time for all feature groups per build (*Collection*) across all subjects. The ratio of *Collection* over *Testing* is shown in the *C/T %* column. Times are reported in **minutes**, and the maximum value of each column is shown in bold.

static source code and dependency analysis. Conversely, TES_CHN, with an average of 1.3 seconds per build, shows the lowest data collection time, since this feature group is based on analyzing test case source code changes in a build, and does not require any historical data or preprocessing. Further, the source code of regression test cases is not frequently changed.

Further, Table 2.5 shows the ratio of the total data collection time over the regression testing time for all subjects. The results suggest that data collection time can increase the regression testing time between 1% and 71% with an average of 11% across subjects. As expected, larger systems and test suites tend to correspond to much larger percentages, e.g., 44% for S_1, 71% for S_3, and our analysis shows that the percentage of data collection time and subject size (SLOC) are strongly correlated (Spearman's ρ) with $\rho = 0.79$. This can cause practical issues in a CI context since regression testing should be fast enough to enable the code to be built and tested several times a day. As a result, this justifies our attempt to investigate whether all feature groups are required to achieve satisfactory TCP effectiveness, especially those groups entailing the largest preprocessing times.

EXP1.1 To answer RQ1.1, we collected data for all builds of each subject and stored the resulting datasets. We recorded the data collection time related to preprocessing and measurement for each feature group, which is used to answer RQ1.1.

Feature Group Pair		*p-value*	CL
COD_COV_COM ❶	TES_CHN	0.00	1.00
	TES_PRO	0.00	0.98
	REC	0.00	0.95
	TES_COM	0.00	0.80
	F_COV	0.00	0.60
	COD_COV_CHN	0.00	0.59
	COD_COV_PRO	0.00	0.59
	DET_COV	0.00	0.59
COD_COV_PRO ❷	TES_CHN	0.00	0.99
	TES_PRO	0.00	0.95
	REC	0.00	0.92
	TES_COM	0.00	0.71
	F_COV	0.00	0.51
	DET_COV	0.00	0.51
	COD_COV_CHN	< 0.01	0.50
DET_COV ❸	TES_CHN	0.00	0.99
	TES_PRO	0.00	0.95
	REC	0.00	0.92
	TES_COM	0.00	0.71
	F_COV	0.00	0.51
	COD_COV_CHN	< 0.01	0.50
COD_COV_CHN ❹	TES_CHN	0.00	0.99
	TES_PRO	0.00	0.95
	REC	0.00	0.92
	TES_COM	0.00	0.71
	F_COV	0.00	0.51
F_COV ❺	TES_CHN	0.00	0.98
	TES_PRO	0.00	0.95
	REC	0.00	0.92
	TES_COM	0.00	0.70
TES_COM ❻	TES_CHN	0.00	0.98
	TES_PRO	0.00	0.93
	REC	0.00	0.88
TES_PRO ❼	TES_CHN	0.00	0.74
	REC	< 0.01	0.50
REC ❽	TES_CHN ❾	0.00	0.73

Table 2.6: Results of pairwise comparison of the data collection time of feature groups using the Wilcoxon Signed-rank test and the Common Language (CL) effect size. Also, each feature group is indicated with a number that shows their rank based on average data collection time per build (higher ranks indicate higher data collection time).

RQ1.1 To check whether the differences between the average data collection times of feature groups (ref. Table 2.4) are statistically significant, we performed multiple pairwise Wilcoxon Signed-rank tests, to compare the data collection times of each feature group pair across the same builds. Note that the Wilcoxon Signed-rank test is a non-parametric test and does not make any distributional assumptions.

As shown in Table 2.6 in column *p-value*, the pairwise comparison indicates that the differences between all possible feature group pairs are statistically significant. As a result, we can meaningfully rank groups based on their average data collection times. The ranks are depicted in Table 2.6 with ⓫, where a lower rank means higher data collection time. Thus, COD_COV_COM has the highest data collection time, while TES_CHN has the lowest.

We also used the Common Language (CL) effect size analysis to investigate the practical significance of differences. CL is the probability of a randomly sampled item from a population being greater than a randomly sampled item from another population. Ta-

ble 2.6 shows the values of effect sizes for each pair across feature groups. The results show that feature groups that rely on coverage analysis (i.e., with COV in their name) have the smallest differences with each other and very large differences with other feature groups. For instance, the difference of COD_COV_COM with other coverage-based feature groups (e.g., F_COV) has a small effect size of around 0.6. Additionally, except for TES_PRO and REC, all feature pairs that do not rely on coverage analysis have large differences with each other (CL effect sizes above 0.7).

RQ1.1 Summary

The data collection time of feature groups that rely on coverage analysis is significantly higher than other feature groups. Feature groups that rely on test case code analysis have the second-highest, and the feature group that is based on test case execution history has the lowest data collection time.

EXP1.2 We conducted the same experiment as EXP1.1, but by only accounting for data collection time of the features for impacted files to address RQ1.2.

RQ1.2 To investigate the effect of accounting for impacted files in the data collection process, we measured the data collection time of all the features which are based on impacted files. Columns *Avg. Total Collection Time* and *Avg. Impacted Collection Time* in Table 2.7, show the average data collection time for all features per build and the average data collection time for features based on impacted files per build, respectively, across all subjects. Also, column *Impacted/Total* shows the percentage of the impacted collection time in the total collection time. The collection time of features based on impacted files takes between 7% and 38% of the total data collection time per build across subjects, with an average of 21%. This is a significant portion of the data collection time and, therefore, is a strong justification to investigate whether features that are based on impacted files are required to achieve satisfactory TCP effectiveness. Our analysis also shows that such percentage has a significant correlation (Spearman's ρ) with both subject size (SLOC, $\rho = 0.56$) and the number of test cases ($\rho = 0.57$), respectively.

RQ1.2 Summary

On average, 21% of the total data collection time per build is spent on collecting data for features that are based on impacted files, which is a significant portion of the collection time.

EXP1.3 To answer RQ1.3, we used Spearman's rank correlation to assess the relationship between the data collection time of features and subject characteristics. Spearman rank correlation is a non-parametric test without any conditions about the data distribution. It is used when the variables are monotonically related and measured on a scale that is at least ordinal. The variables (i.e., data collection time of features and subject characteristics) are ordinal and in monotonic order as shown in Figure 2.4. Thus, it is an appropriate choice in our context. We performed the correlation analysis between the data collection time of all features as well as each feature group, which was recorded in EXP1.1, and subject size (SLOC) as well as the number of test cases and builds.

S_{ID}	Total	Impacted	I/T %
S_1	**11.7**	**3.5**	29
S_2	1.1	0.1	8
S_3	5.0	1.9	**38**
S_4	1.7	0.5	31
S_5	2.4	0.8	34
S_6	1.4	0.3	25
S_7	2.1	0.7	35
S_8	0.9	0.2	25
S_9	0.5	0.1	26
S_{10}	0.8	0.2	27
S_{11}	1.2	0.4	32
S_{12}	0.5	0.1	21
S_{13}	1.2	0.1	7
S_{14}	1.4	0.4	28
S_{15}	0.2	< 0.1	10
S_{16}	0.2	< 0.1	15
S_{17}	0.1	< 0.1	8
S_{18}	0.6	0.1	22
S_{19}	0.5	0.1	12
S_{20}	0.7	0.1	14
S_{21}	0.2	< 0.1	9
S_{22}	0.3	< 0.1	13
S_{23}	0.2	< 0.1	16
S_{24}	0.2	< 0.1	20
S_{25}	0.1	< 0.1	12

Table 2.7: Average data collection time for features related to impacted files per build (*Impacted*) compared to the average data collection time for all feature groups per build (*Total*) across all subjects. The ratio of *Impacted* over *Total* is shown in the *I/T %* column. Times are reported in **minutes**, and the maximum value of each column is shown in bold.

RQ1.3 As shown in Table 2.4, the collection time of each feature group varies across subjects. To explain such variation, we analyzed correlations between subjects' characteristics and their feature groups' collection times. We used Spearman's rank correlation to assess the strength of such correlations as the inspections of scatterplots showed monotonic relationships.

As shown in Table 2.8, the subjects' SLOC strongly correlates with most of the feature groups' data collection times, which is not surprising since the time for coverage and code complexity analysis increases as SLOC increases. We also observe that the average number of test cases per build has a moderate correlation with the data collection time of most feature groups, and a strong correlation (0.95) with REC. REC computes features based on test execution history, and subjects with more test cases have more execution history, this results in higher collection times.

Figure 2.4 depicts six strong correlations between subjects' SLOC and data collection times of the feature groups. They suggest there is an increasing monotonic relationship between SLOC and data collection time with a correlation coefficient above 0.8. We can see that there is one outlier in all the scatterplots, which refers to subject S_1. Thus, we

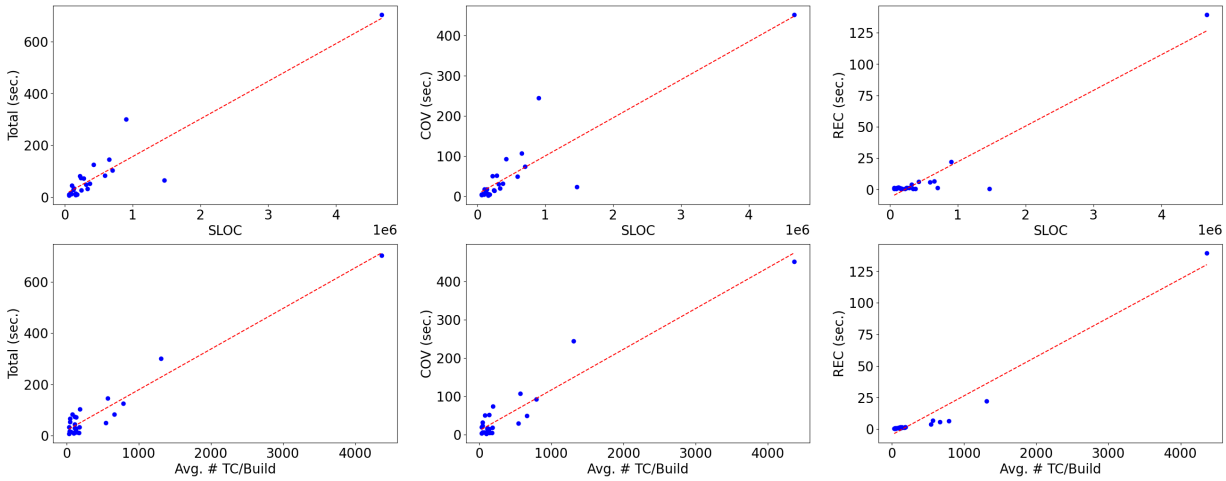

Figure 2.4: Linear correlation graphs between two characteristics of subjects (x-axis) and the total data collection time of all feature groups as well as two individual feature groups (y-axis).

Characteristic	COD_COV_COM	COD_COV_PRO	DET_COV	COD_COV_CHN	F_COV	TES_COM	TES_PRO	REC	TES_CHN	Total
SLOC	**0.84**	**0.81**	**0.81**	**0.83**	**0.83**	**0.83**	0.49	0.41	0.61	**0.83**
Java SLOC	**0.87**	**0.82**	**0.82**	**0.85**	**0.86**	**0.86**	0.46	0.40	0.57	**0.85**
# Commits	0.57	0.57	0.57	0.60	0.57	0.51	0.38	0.60	0.76	0.63
# Builds	0.31	0.30	0.30	0.29	0.28	0.18	0.23	0.54	0.44	0.35
# Failed Builds	0.29	0.26	0.26	0.27	0.30	0.24	-0.04	0.36	0.31	0.29
Failure Rate (%)	-0.02	-0.08	-0.08	-0.05	-0.01	0.11	-0.29	-0.06	-0.24	-0.06
Avg. # TC/Build	0.56	0.56	0.56	0.56	0.57	0.40	0.46	**0.95**	0.53	0.58
Avg. Test Time (min)	-0.18	-0.11	-0.11	-0.15	-0.14	-0.27	0.12	-0.04	-0.00	-0.16

Table 2.8: Spearman's rank correlation between characteristics of subjects and data collection time of feature groups. Strong correlations (values greater than 0.8) are shown in bold.

repeated the correlation analysis without such outlier to investigate whether it caused significant inflation in the correlation coefficients. Though the results showed a reduction in correlation coefficients, trends remained consistent.

RQ1.3 Summary

The size of the subjects and the average number of test cases per build are positively and significantly correlated with the data collection time of the majority of features.

2.4.4.2 Training and Testing of Ranking Models for TCP (RQ2)

In the following, we assume that the builds of each subject are assigned a unique id incrementally, according to their time of occurrence.

Model Selection Bertolino et al. [20] showed that Multiple Additive Regression Trees (MART) [47], a.k.a. Gradient boosted regression trees, is the best ML ranking model in the TCP context. Since our study uses a different evaluation metric and different subjects, we conducted an experiment to verify whether or not MART remains the best ML ranking model (M_{opt}) for our subjects as well and thus to obtain the best results. To this

end, we evaluated MART against five other ML ranking models, namely LambdaMART (LMART) [48], Random Forest (RF) [27], RankBoost [49], ListNet [50], and Coordinate Ascent (CA) [51]. They were selected as they cover pairwise and listwise ML ranking models, are considered the most popular techniques in the RankLib [52] library, and were already used by Bertolino et al. [20]. For all ranking models, we used an existing implementation found in the RankLib [52] library with the default values for model hyperparameters. For each failed build with id n, we used all feature records from the previous failed builds for training, i.e., all feature records whose build id is between 1 and $n-1$. We then evaluated the ranking model based on the feature records of build n. We conducted this experiment based on the latest 50 failed builds of each subject. For subjects with less than 50 failed builds (see Section 2.4.2), we used all failed builds. We then used the Friedman statistical test [33] to see if there is at least one ML model that is significantly better than others across the same builds and used the Nemenyi post hoc test [34] to single out the best model.

The results of our comparison showed that the RF ranking model performed significantly better than the other models including MART, in contrast to the results reported by Bertolino et al. [20]. Therefore, we decided to use RF for all the experiments throughout the rest of this study. It is worth reminding that the main focus of this study is assessing the impact of test case features on the effectiveness of TCP. Since we expect practitioners to use the best model, we report such results for RF only. A more detailed investigation of the impact of different sets of features for different ML algorithms is out of the scope of this work.

Hyperparameter Selection Hyperparameter selection can significantly affect the performance of ML models. Thus, we designed an experiment to find the best hyperparameters for the RF ranking model. We investigated the following RF hyperparameters: *rtype*, *srate*, *bag*, *frate*, *tree*, *leaf*, and *shrinkage*. Please refer to the RankLib [52] library for details regarding these hyperparameters. We defined a hyperparameter search space by using the following values for each hyperparameter:

$$\text{rtype} \in \{\text{MART}, \text{LMART}\}, \text{srate} \in \{0.5, 1.0\},$$

$$\text{bag} \in \{150, 300, 600\}, \text{frate} \in \{0.15, 0.3, 0.6\}, \text{tree} \in \{1, 3, 5\},$$

$$\text{leaf} \in \{50, 100, 200\}, \text{shrinkage} \in \{0.05, 0.1, 0.2\}$$

The above hyperparameter search space leads to 972 possible combinations. Since exploring all hyperparameter combinations was not computationally possible, we relied on covering arrays [53] to identify 42 combinations with the strength of 3, thus covering all hyperparameter combinations of size 3. We used all the builds (1133 builds) across subjects to evaluate the 42 combinations. Finally, we compared the results of the 42 combinations with the results of the default combination (HC_{def}), which was obtained from the model selection experiment, using the pairwise Wilcoxon Signed-rank test. The results showed that HC_{def} was not the best but was among the best. The best combination (HC_{opt}) achieved an average APFD_C of 0.824 that is higher than that of HC_{def} with 0.813. The difference between HC_{opt} and HC_{def} is statistically significant, though the magnitude of

the difference is small. Based on the results, the selected hyperparameter values (HC_{opt}) used through the rest of this study were:

$$\text{rtype} = \text{MART}, \text{srate} = 0.5, \text{bag} = 150, \text{frate} = 0.3,$$

$$\text{tree} = 5, \text{leaf} = 200, \text{shrinkage} = 0.2$$

Model and Hyperparameter Selection

The Random Forest (RF) ML ranking model performs significantly better than other investigated ranking models. In addition, the difference between the TCP effectiveness of RF's default and best hyperparameters is small though in favor of the latter.

EXP2.1 To answer RQ2.1, similar to the model selection experiment, we trained RF ranking models for the latest 50 failed builds of each subject using all feature records. Our decision for training the models only based on failed builds is based on the results of a recent study. Via empirical analysis, Elsner et al. [39] showed that training ML-based TCP models by including builds with no failures does not improve the effectiveness of the TCP models. Thus, including such builds only increases the training time without any benefit.

In practice, training ML models for each new build may not be practical or necessary, and we investigate this question in RQ3. However, since our focus is on investigating the impact of features on the effectiveness of ML-based TCP models, we train ranking models for all possible builds across subjects, to analyze results on the largest possible number of builds.

As discussed in Section 2.4.3, we report APFD_C only based on the failed builds of all subjects. Table 2.9 shows APFD_C averages and standard deviations of Random Forest ranking models across subjects.

RQ2.1 Column *Full_M* in Table 2.9 shows APFD_C averages for subjects, when the training data includes the full feature set. Average APFD_C values, for all subjects except the last two, are above 0.7, and the average APFD_C across all builds is 0.82. Thus, it is safe to conclude that including the full feature set in the training of ranking models for test case prioritization leads to promising results for most subjects. However, we can observe a considerable APFD_C variation in Table 2.9 across subjects (e.g., 0.56 for S_{25} and 0.98 for S_{15}). For this reason, we conducted a correlation analysis to measure potential correlations between APFD_C and subject characteristics. As a result, we did not find any strong or moderate correlations, and therefore we left this question for future work.

RQ2.1 Summary

Using the full feature set for training ML ranking models for TCP leads to promising results for most subjects. However, we can observe a considerable variation in results across subjects.

S_{ID}	*Full_M*	*IMP_M*	*TES_M*	*REC_M*	*COV_M*
S_{15}	0.98±0.05	0.97±0.05	0.97±0.07	**0.98±0.03**	0.73±0.24
S_{10}	0.94±0.14	0.94±0.14	0.92±0.17	**0.94±0.13**	0.66±0.21
S_{14}	**0.92±0.16**	0.92±0.17	0.92±0.18	0.90±0.20	0.60±0.28
S_{7}	0.91±0.17	**0.91±0.16**	0.72±0.23	0.89±0.20	0.46±0.24
S_{5}	**0.90±0.12**	**0.90±0.12**	0.89±0.17	0.88±0.16	0.60±0.17
S_{6}	0.88±0.23	0.88±0.22	0.76±0.29	**0.89±0.22**	0.62±0.29
S_{19}	**0.88±0.12**	0.88±0.13	**0.88±0.12**	0.87±0.12	0.57±0.25
S_{11}	0.86±0.26	0.86±0.26	**0.89±0.21**	0.84±0.28	0.62±0.27
S_{23}	**0.85±0.15**	**0.85±0.15**	0.85±0.16	0.84±0.16	0.50±0.29
S_{12}	0.85±0.16	0.85±0.16	0.82±0.17	**0.85±0.12**	0.62±0.22
S_{2}	0.84±0.04	0.84±0.04	**0.85±0.05**	0.84±0.04	0.60±0.14
S_{9}	**0.81±0.20**	0.81±0.21	0.79±0.24	0.80±0.22	0.52±0.12
S_{18}	0.81±0.24	0.83±0.22	**0.86±0.17**	0.78±0.28	0.58±0.23
S_{4}	0.79±0.20	0.79±0.20	0.84±0.19	**0.87±0.16**	0.53±0.17
S_{8}	**0.78±0.14**	0.78±0.15	0.77±0.14	0.68±0.25	0.59±0.23
S_{20}	0.78±0.24	0.78±0.24	**0.81±0.21**	0.78±0.18	0.66±0.26
S_{16}	0.78±0.22	**0.79±0.21**	0.77±0.24	0.78±0.23	0.58±0.19
S_{1}	0.77±0.25	0.78±0.24	0.65±0.25	**0.79±0.22**	0.42±0.23
S_{21}	0.77±0.23	0.77±0.23	**0.80±0.18**	0.74±0.24	0.66±0.23
S_{17}	**0.77±0.18**	0.76±0.18	0.76±0.18	0.73±0.20	0.50±0.12
S_{3}	**0.75±0.25**	**0.75±0.25**	0.68±0.24	0.74±0.26	0.56±0.21
S_{13}	0.73±0.24	0.74±0.24	**0.79±0.23**	0.66±0.27	0.56±0.22
S_{24}	**0.70±0.31**	0.67±0.31	0.69±0.29	0.69±0.31	0.60±0.23
S_{22}	0.66±0.29	0.67±0.29	**0.71±0.28**	0.63±0.28	0.51±0.23
S_{25}	0.56±0.27	0.57±0.28	**0.60±0.18**	0.54±0.20	0.54±0.09

Table 2.9: **APFD$_C$** averages and standard deviations across subjects for four different Random Forest ranking models. *Full_M* represents the model that was trained on data that **includes** the full feature set. *IMP_M* was trained on data that **excludes** features related to impacted files. *TES_M*, *REC_M*, and *COV_M* were trained only with features related to test case source code, execution history, and test code coverage, respectively. For each subject (row), the APFD$_C$ of the model with the highest average APFD$_C$ and the lowest standard deviation is shown in bold.

EXP2.2 To answer RQ2.2, we conducted the same experiments as EXP2.1, but we removed features related to impacted files to compare their results with those of the all-features experiment (EXP2.1) and measure their effect on TCP effectiveness.

RQ2.2 Column *IMP_M* in Table 2.9 reports average APFD$_C$ values for subjects when features related to impacted files are not included in the training data. As shown, the differences between the APFD$_C$ of *Full_M* and *IMP_M* are small across all subjects, and they range between 0.0 and 0.03.

We also ran a Wilcoxon Signed-rank test based on APFD$_C$ results for *Full_M* and *IMP_M* across all builds and subjects. The test results show that there is no statistically significant difference between the APFD$_C$ of *Full_M* and *IMP_M*; *p-value*=0.14. Hence, it is safe to conclude that accounting for the features of impacted files does not bring practical advantages as it does not have a significant effect, across all subjects, on the effectiveness of the ranking models in terms of APFD$_C$.

RQ2.2 Summary

Test case features that are related to impacted files do not have a significant effect on the effectiveness of TCP.

EXP2.3 To answer RQ2.3, we conducted two sets of experiments:

- **EXP2.3.1** We trained nine ranking models for each failed build using the same method as in EXP2.1 and EXP2.2. The nine models differ based on the feature groups used for their training since for each model, one feature group was left out.
- **EXP2.3.2** We trained three ranking models for each failed build using the same method as in EXP2.1 and EXP2.2. Again, the three models differ based on the features used for their training. The first model (*COV_M*) was trained using all coverage-related features, all of which are collected via source code coverage analysis of the system under test, i.e, F_COV, COD_COV_COM, COD_COV_PRO, COD_COV_CHN, and DET_COV. The second model (*TES_M*) was trained using features that are calculated via static analysis of test case source code, i.e., TES_COM, TES_PRO, and TES_CHN. The last model (*REC_M*) was trained only based on test case execution history, i.e., REC. The data collection time for features in the same high-level group is heavily driven by data preprocessing of common data sources, and therefore, removing or adding features in such a group does not make a practical difference. Therefore, understanding the impact of each high-level group can help practitioners select the best feature group(s) when the collection of all features is not possible.

As we explained in EXP2.3.2 above, the feature extraction cost of a single feature (group) within a high-level group is nearly the same as the cost of extracting all features in that group. As a complementary lower-level analysis, that is admittedly of less practical importance, we investigated the impact of removing one (low-level) feature group and the importance of each individual features in RQ2.4. Given the large volume of our data and the high execution time of the experiments when training Random Forest models, we could not train these models using all possible feature (group) combinations.

RQ2.3 The results of EXP2.3.1 show that excluding any one of the feature groups does not cause a significant decrease in effectiveness for the trained models. We conducted nine Wilcoxon Signed-rank tests on $APFD_C$ results for all builds across subjects. As shown in Table 2.10, among all nine feature groups, removing TES_COM, REC, or TES_PRO causes a statistically significant difference in $APFD_C$ with *Full_M*; TES_COM: *p-value*<0.01, REC: *p-value*<0.01, and TES_PRO: *p-value*$=0.01$. However, the CL effect size analysis shows that the magnitude of their impact is practically negligible (CL is around 0.5). Based on the above results, we can conclude that removing any of the nine feature groups does not have a practical impact on the $APFD_C$ of the ranking models.

Columns *TES_M*, *REC_M*, and *COV_M* in Table 2.9 report the average $APFD_C$ of ranking models trained with features related to test case source code, execution history, and test code coverage, respectively. Though the $APFD_C$ of *COV_M* is low, *REC_M* reaches

Experiment Pair		*p-value*	CL
Full	TES_COM	< **0.01**	0.51
Full	REC	< **0.01**	0.54
Full	TES_PRO	**0.01**	0.51
Full	COD_COV_COM	0.13	0.50
Full	COD_COV_CHN	0.18	0.50
Full	DET_COV	0.41	0.50
Full	COD_COV_PRO	0.52	0.50
Full	F_COV	0.56	0.50
Full	TES_CHN	0.97	0.50

Table 2.10: Pairwise $APFD_C$ comparison results across subjects between models that exclude individual feature groups and the model that includes the full feature set (*Full*). The comparison is done using the Wilcoxon Signed-rank test and the Common Language (CL) effect size. *p-values* below 0.05 are shown in bold.

Experiment Pair		*p-value*	CL
Full	COV_M	< **0.01**	0.79
Full	REC_M	< **0.01**	0.51
Full	TES_M	< **0.01**	0.53

Table 2.11: Pairwise $APFD_C$ comparison results across subjects between models that include individual high-level feature groups and the model that includes the full feature set (*Full*). The comparison is done using the Wilcoxon Signed-rank test and the Common Language (CL) effect size. *p-values* below 0.05 are shown in bold.

similar $APFD_C$ to *Full_M*. To analyze the result of EXP2.3.2, we conducted three Wilcoxon Signed-rank tests to assess the differences of the three models with *Full_M*. As shown in Table 2.11, the $APFD_C$ results of all three models are statistically different from *Full_M*; *p-value*$\leq$0.01. The CL effect size analysis for $APFD_C$ results shows that the difference with *Full_M* for *COV_M*, with a value of 0.79, is practically significant, as opposed to the CL effect sizes of *TES_M* and *REC_M*, which are 0.53 and 0.51, respectively. Thus we can conclude that, among all high-level feature groups, *TES_M*—which is trained with features related to the source code of test cases— and *REC_M*—which is trained with features related to the execution history of test cases— achieve the best results, with nearly the same $APFD_C$ as the models trained with the full feature set. Further, the models trained with only the coverage-based feature groups achieve the worst $APFD_C$ and are not reliable individually.

Intuitively, coverage-based features (*COV_M*) are expected be effective for test prioritization since they directly measure which code parts are exercised by tests. However, our results demonstrate otherwise. This finding stems from TCP's dual objective: detecting faults early while minimizing execution costs. Tests with high coverage often have higher computational costs but may not target the majority of fault-prone code areas effectively.

Feature Group	Feature	Avg. Freq.
REC	Age	17034
TES_PRO	OwnersExperience	10618
TES_PRO	AllCommitersExperience	7643
REC	TotalMaxExeTime	5409
REC	LastExeTime	4095
TES_PRO	OwnersContribution	3997
TES_PRO	CommitCount	3986
REC	TotalAvgExeTime	3977
REC	RecentAvgExeTime	3850
REC	RecentMaxExeTime	3741
TES_COM	RatioCommentToCode	3695
TES_COM	CountStmtDecl	3383
TES_COM	CountLineCodeDecl	3342
TES_COM	CountLineBlank	3149
TES_COM	CountStmtExe	2825

Table 2.12: Top 15 frequently used individual features in models that were trained on the full feature set. The *Avg. Freq.* column shows the average usage frequency of a feature across all ranking models.

Therefore, using coverage as the sole prioritization criterion leads to suboptimal results compared to other feature groups.

RQ2.3 Summary

Each of the nine feature groups does not have a significant impact on TCP effectiveness, individually. This is different for high-level feature groups. Features related to test case source code or execution history have nearly the same impact on TCP effectiveness as all features, whereas coverage-based features have a significantly lower impact.

EXP2.4 To address RQ2.4, we rely on RankLib [52] as it provides model feature statistics for the RF model. It captures how frequently each feature is used to split nodes of trained regression trees. Such decision trees use information gain, which is based on the concept of entropy, to determine which feature to use to further split the tree at each training step. Therefore, a higher usage frequency in the tree for a feature indicates that it tended to yield higher information gains at training time. We use feature usage frequencies to measure the impact of individual features in our feature groups.

RQ2.4 Table 2.12 reports the average usage frequency of the top 15 features in *Full_M* for models across all subjects. Top features belong to the three feature groups REC, TES_PRO, and TES_COM, all of which are identified as significant feature groups by RQ2.3. For more details, we added the usage frequency of all individual features in Appendix A.2.

From the TES_COM group, features capturing the size of test cases show higher usage, which is to be expected as large test cases tend to have a higher execution time and coverage. Also, from the TES_PRO group, features capturing the experience of test case

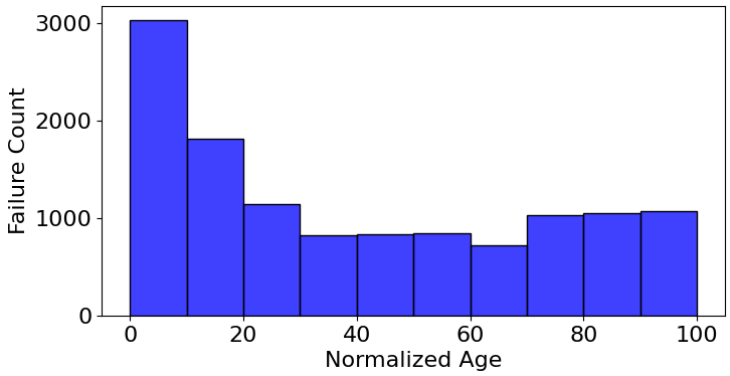

Figure 2.5: The histogram shows the relationship between the age of all test cases and their failure frequency. The x-axis shows the normalized age between 0 and 100 and the y-axis shows the number of failures of all test cases in a given age period.

developers and the number of commits are more important. We conjecture that experienced developers may better understand the system under test to develop high-quality test cases compared to less experienced developers. Also, higher numbers of commits for a test case suggest that the test case is under active development and is being updated to detect faults.

From the REC feature group, features related to previous execution times and the age of test cases are the most frequently used. The former are good indicators of future test case execution times and explain why execution time-related features have high usage. However, seeing the age of a test case as the most important among all features may seem surprising at first glance. To explain this, we analyzed the relation between the age of test cases and their failure occurrences across all subjects. As shown in Figure 2.5, failures drop sharply after the first 10% and 20% of the builds on which they are executed. Thus, the age of a test case is a good predictor of its failure probability. Further, this suggests that all subjects contain obsolete test cases that are not actively maintained and not kept updated. When a new functionality of the SUT is developed, a number of test cases are developed to test it, and once the functionality works as expected and becomes stable over time, their corresponding test cases often pass across builds.

RQ2.4 Summary

The most frequently used test case features in the ML ranking models belong to REC, TES_PRO, and TES_COM feature groups. These features include test case size, development history, failure history, and age.

EXP2.5 A heuristic-based TCP approach prioritizes test cases based on single features of test cases [39]. Given a test suite T and a test case feature f_i, we define two heuristic-based ranking models $M_{f_i,a}$ and $M_{f_i,d}$. $M_{f_i,a}$ produces a test case ordering T_a^* in which test cases are sorted by their corresponding value of f_i in ascending order. Conversely, $M_{f_i,d}$ produces T_d^* in which test cases are sorted in descending order.

Elsner et al. [39] reported that heuristic-based TCP models that are based on *F_MaxTestFileFailRate*

(defined in 2.3.2.2) outperform other heuristics. We first investigated if the same results held with the subjects of this study. To do that, for each test case feature f_i ($1 \leq i \leq 150$), we applied the $M_{f_i,a}$ and $M_{f_i,d}$ heuristic-based models (300 models in total) and compared their results. We observed that the $M_{f_{56},d}$ model, which is based on the *F_FailRate(Total)* feature (Section 2.3.2.2), performed best for the majority of subjects, in contrast to previous work [39]. Elsner et al. [39] also concluded that heuristic-based TCP models are better than ML-based TCP models in terms of average APFD_C, though not statistically different. Thus, in this experiment, we compared the results of our ML-based TCP model, RF, with the best heuristic, i.e., *F_FailRate(Total)*.

RQ2.5 To compare the effectiveness of the best *H_M* (heuristic-based) model with the *Full_M* ML-based model, we conducted a Wilcoxon Signed-rank test for all builds across subjects. The test results showed that the difference between the two models is statistically significant (*p-value*< 0.01), and the *Full_M* model, with an average APFD_C of 0.82, clearly outperforms the *H_M* model, with an average APFD_C of 0.71.

For further analysis, we compared *H_M* and *Full_M* by conducting Wilcoxon Signed-rank tests for each subject. Table 2.13 shows the statistical test results as well as APFD_C averages and standard deviations across subjects. Based on Table 2.13, *H_M* outperformed *Full_M* in only four subjects in terms of average APFD_C, and in only one of these four subjects, the difference was statistically significant. For the rest of the subjects, *Full_M* achieved a higher average APFD_C, and in 13 subjects, the difference was statistically significant.

Further, using Spearman's rank correlation coefficient (ρ), we investigated the relationship between subject characteristics and their *Full_M* and *H_M* average APFD_C. The analysis showed that a moderate correlation, with $\rho = -0.28$, exists between subject failure rate and the difference between *Full_M* and *H_M* in terms of average APFD_C. Thus, the *H_M* model tends to perform better than *Full_M* in subjects that have higher failure rates, especially in the presence of a large number of failed builds (e.g., S_4), which can be explained since *H_M* is based on the failure rate heuristic. However, Beller et al. [54] conducted a comprehensive analysis of TravisCI projects and showed that for all 1,108 Java projects with test executions, the ratio of builds with at least one failed test case has a median of 2.9% and a mean of 10.3%. Therefore, since high failure rates (e.g., failure rates above 30% in our subjects) are not common in CI contexts, *H_M* is probably not a good option in practice.

To conclude, our empirical analysis shows that the Random Forest TCP ranking model, when trained on the full feature set, performs overall significantly better than the best heuristic-based model based on the *F_FailRate (Total)* feature. Nevertheless, the heuristic-based model performs significantly better for a few subjects. Though this remains to be confirmed by further studies, an analysis of the characteristics of these projects suggests this is due to a combination of large numbers of failed builds with high failure rates, which are unlikely in a CI context.

S_{ID}	*Full_M*	*H_M*	*p-value*	CL
S_2	**0.84±0.04**	0.82±0.05	< **0.01**	0.86
S_{16}	**0.78±0.22**	0.58±0.16	< **0.01**	0.78
S_{19}	**0.88±0.12**	0.50±0.38	< **0.01**	0.74
S_{21}	**0.77±0.23**	0.49±0.33	< **0.01**	0.73
S_6	**0.88±0.23**	0.68±0.30	< **0.01**	0.72
S_{18}	**0.81±0.24**	0.60±0.33	< **0.01**	0.69
S_{20}	**0.78±0.24**	0.57±0.32	< **0.01**	0.68
S_{15}	**0.98±0.05**	0.86±0.30	< **0.01**	0.66
S_1	**0.77±0.25**	0.58±0.32	< **0.01**	0.66
S_7	**0.91±0.17**	0.84±0.25	0.13	0.65
S_3	**0.75±0.25**	0.54±0.34	< **0.01**	0.64
S_{17}	**0.77±0.18**	0.69±0.23	0.07	0.62
S_9	0.81±0.20	**0.83±0.12**	0.30	0.62
S_{13}	**0.73±0.24**	0.64±0.29	0.43	0.61
S_{12}	**0.85±0.16**	0.77±0.21	**0.02**	0.60
S_5	**0.90±0.12**	0.74±0.34	**0.01**	0.59
S_{22}	**0.66±0.29**	0.55±0.35	0.08	0.58
S_8	**0.78±0.14**	0.71±0.18	0.07	0.56
S_{14}	**0.92±0.16**	0.84±0.28	< **0.01**	0.56
S_{11}	0.86±0.26	**0.86±0.21**	0.39	0.55
S_{24}	**0.70±0.31**	0.66±0.32	0.44	0.54
S_{25}	0.56±0.27	**0.59±0.28**	0.81	0.48
S_{10}	**0.94±0.14**	0.94±0.22	0.53	0.46
S_{23}	**0.85±0.15**	0.85±0.22	0.27	0.45
S_4	0.79±0.20	**0.89±0.16**	**0.04**	0.42

Table 2.13: Pairwise $APFD_C$ comparison across subjects between the heuristic model (*H_M*) based on *F_FailRate Total* and ML-based models that were trained on the full feature set (*Full_M*). The comparison is done using the Wilcoxon Signed-rank test and the Common Language (CL) effect size. For each subject (row), the $APFD_C$ of the model with the higher average $APFD_C$ and the lower standard deviation is shown in bold. Also, *p-values* below 0.05 are shown in bold.

RQ2.5 Summary

The Random Forest ranking model, when trained on the full feature set, significantly outperforms the best heuristic-based model for most subjects. However, the heuristic-based model performs significantly better for a few subjects, all of which have high build failure rates.

2.4.4.3 Effectiveness Decay of ML-Based TCP Models (RQ3)

EXP3 To answer RQ3, we used all of the pretrained models from EXP2.1 and tested each model on the subsequent builds using features that were computed based on preprocessed data (i.e., dependency graph, source code, and process metrics) of the build related to the model. Assume we have n builds ($B = \{b_1, b_2, \cdots, b_n\}$) and n pretrained models ($M = \{m_1, m_2, \cdots, m_n\}$), and the pretrained model of b_i is m_i where $1 \leq i \leq n$. In

	m_1	m_2	m_3	m_4	m_5
b_1	0	-	-	-	-
b_2	1	0	-	-	-
b_3	2	1	0	-	-
b_4	3	2	1	0	-
b_5	4	3	2	1	0

Table 2.14: An example of retraining window (RW) values when the number of builds is equal to 5. Each cell shows the RW when TCP is applied to the build of the cell's row using the pretrained model of the cell's column. For instance, if we use m_1 for b_4, then $RW = 3$.

this experiment, we used b_k and m_k ($1 \leq k \leq n$ and $n = 50$ for the latest 50 failed builds) from EXP2.1 and all its relevant preprocessed data. Then, for each b_i after b_k, i.e., $\{b_{k+1}, b_{k+2}, \cdots, b_n\}$, we recomputed its test case features based on the preprocessed data available in b_k, ranked its test cases using m_k, and evaluated the ranking. The main motivation for reusing pretrained ranking models is to reduce the cost of data preprocessing and model training, and for this reason, we wanted to assess the impact of such reuse on APFD_C decay. When the preprocessed data of b_k did not contain enough information to compute some of the test case features in b_i (i.e., new test cases or new code coverage graphs), we used the mean substitution imputation method for missing values, as it is a simple, efficient, and effective method [55], and assigned those missing features with the average value across all other test cases with such feature values.

Results As we discussed in *EXP3*, we have tested the pretrained ranking model for each failed build on its following failed builds to measure how the APFD_C of the pretrained models decreases over time. Here we assume a retraining window (RW) as the distance (i.e., number of builds) between the builds based on which the model is trained and tested, respectively, e.g., when m_k, that is trained based on the data captured up to b_k, is used for TCP in builds up to b_{k+4} without retraining, then RW = 4. Table 2.14 shows an example of RWs when the number of builds is equal to 5. Each cell with row b_i and column m_j in Table 2.14 is equal to the RW when the pretrained model m_j is used to prioritize the test cases of b_i.

We computed the APFD_C for all valid builds and pretrained models across subjects. Figure 2.6 shows the APFD_C of pretrained models, based on all builds across subjects, for RWs ranging from 0 to 45. The x-axis is the RW, and the y-axis is the average APFD_C of all pretrained models with the same RW. Since each subject had a total of 50 builds, we excluded RW $\geq$ 46 to ensure at least five APFD_C values per subject, thus enabling meaningful average calculations. Notably, as shown in Table 2.2, four subjects ($S_{8,20,22,24}$) had fewer than 50 failed builds. For these subjects, we computed APFD_C only for the feasible RW values given their respective build counts.

As shown in Figure 2.6, there is a steady decreasing APFD_C trend until RW is about 11. The two points RW = 0 and RW = 11 are connected with a red dotted line in Figure 2.6. The slope of this line, which is the average decrease in APFD_C after each RW increment, is equal to -0.005. However, APFD_C values above RW = 11 display unstable trends. Hence, we can conclude that retraining ranking models with a RW of less than 11 builds

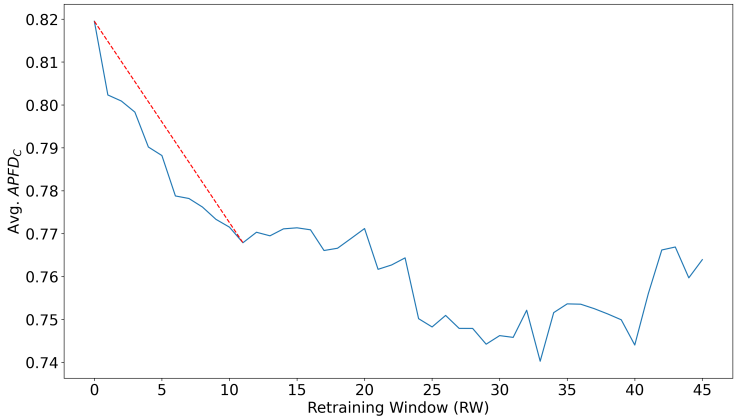

Figure 2.6: This graph shows the relationship between the retraining window (RW) of ranking models across all subjects and their average $APFD_C$.

is necessary to obtain predictable results across all subjects. However, given the available budget for data collection and model retraining, we also conclude that ML ranking models should be retrained as frequently as possible to achieve the best possible TCP effectiveness.

The precision of the RW recommendations can be further enhanced through advanced techniques that monitor feature distributions across consecutive builds. By detecting significant changes in these distributions, the approach could dynamically adjust the RW rather than relying on fixed values. Although this method introduces some computational overhead due to frequent feature collection, it offers benefits when retraining costs are high.

RQ3 Summary

Retraining ranking models with a retraining window (RW) of less than 11 builds is necessary to obtain predictable results. Also, the RW should be as small as possible to obtain the best TCP effectiveness.

2.4.4.4 Trade-off Between Data Collection Time and TCP Effectiveness for Features (RQ4)

Based on the results of RQ1-3, three different decisions can be taken concerning the application of TCP techniques for a software system to achieve the best trade-off, in a given context, between data collection overhead and TCP effectiveness. In the following, we provide guidelines regarding such decisions.

Use ML model relying on the full feature set with retraining at each build or every 11 builds. When higher TCP effectiveness is essential, and the cost of data collection can be afforded, the use of the full feature set for training ML-based TCP models is recommended since, as shown in RQ2, it yields the best results. If necessary, based on the results of RQ3, to decrease data collection time at the cost of a slight effectiveness loss, model retraining can be conducted every 11 builds instead of each build.

Use ML model relying on the REC feature set. When the overhead of the full feature set cannot be afforded, regardless of the training strategy, and when a small decrease in model effectiveness is acceptable given a much lower data collection time, an ML model relying on the REC feature set is recommended. As discussed in RQ1, the data collection time of the REC feature group (i.e., features based on test execution history) is in the order of seconds and negligible. Further, RQ2 results show that not only removing the REC feature group causes statistically significant differences in model effectiveness, but furthermore training the model with only the REC features achieves results that are close to those obtained with the full feature set. Finally, our comparison between ML models based on REC features and heuristics models shows that the former's effectiveness is higher than the latter's.

Use failure-based heuristics. When no data collection overhead is acceptable, the use of failure-based heuristics can be effective, especially in the context of systems with a high number of builds and high failure rates, thus potentially enabling the definition of effective heuristics.

RQ4 Summary

To achieve the best TCP effectiveness when data collection cost is affordable, we recommend using Random Forest (RF) ranking models that are trained on the full feature set. When the data collection is not affordable, we suggest an RF model relying on the REC feature set. Finally, the fastest and least effective approach is the use of failure-based heuristics.

2.4.4.5 Configuration

Data collection for each subject was conducted using one CPU core (Intel Xeon Gold 6234 CPU @ 3.30GHz) with a memory usage ranging between 1GB to 60GB, depending on the size of the subjects. Also, our data collection tool, all results, and datasets are made publicly available[2]. We used PyDriller [56] for mining git repositories, Understand [35] for static and dependency analysis of source code, and RankLib [52] for training our TCP machine learning ranking models.

2.4.5 Threats to Validity

2.4.5.1 Internal Threats

As explained in Section 2.2.2, for scalability reasons invoked by the industry partner supporting this research, the test case code coverage that we computed is an estimation based on static code coverage, which is computed using Understand [35], and the history of file changes in the system's code repository. This, however, can lead to overestimating test case code coverage. Also, based on Section 2.3.2, our proposed test case coverage is at the

[2] https://github.com/Ahmadreza-SY/TCP-CI

file level, which can further worsen the overestimation issue. Future work should systematically investigate trade-offs between finer-grained coverage measurement, scalability, and TCP effectiveness.

In Travis CI, each CI cycle (build) includes one or more jobs, each of which tests the source code for different configurations and platforms (e.g., a project may have two jobs to test the code for Java 1.6 and 1.8). In most cases, the jobs share the same test cases (98% of test cases for 24 out of 25 of our subjects are the same across jobs of the same build), resulting in more than one execution record for the repeated test cases that affect the quality of the datasets negatively. To mitigate this issue, we only focused on a job with the highest number of test cases. This reduces our data sets size in terms of the number of test cases around 2% that may have a negligible effect on our experiments since the large number of builds that are used in the experiments compensate for this reduction. An alternative way to deal with this issue is to include extra features concerning the platform. However, we left this to future work.

Also, as discussed in Section 2.4.4.2, we removed frequent-failing test cases using the three-sigma rule technique that may deflate the results of the ML models. To investigate this, we measured the difference between the effectiveness of our ML models with/without outlier test cases that showed there is no statistically significant difference between the two cases, i.e., removing outlier test cases does not deflate the results of the ML models.

2.4.5.2 External Threats

We relied on *GHTorrent* [41] to search for open-source projects. In addition, *TravisTorrent* [37] was used to fetch build logs from Travis CI and to find the commits that are related to builds. We also included a subset of test case execution record datasets that were published by *RTPTorrent* [36]. There might be flaws in these tools and datasets—though it is unlikely given how widely used they are—that impact the quality of our data collection.

2.5 Related work

Many studies have addressed Test Case Prioritization (TCP) in the regression testing context. Overall, these studies can be classified into two groups: heuristic-based and ML-based techniques. Since our work is ML-based, in the following, we only briefly discuss the heuristic-based group.

Heuristic-based TCP. The studies in this group apply heuristics to TCP that are mainly defined based on test coverage and execution history.

Coverage-based heuristics are based on the principle that increasing structural test coverage of the System Under Test (SUT) increases the chances of fault detection [57]. Structural coverage measures include statement coverage [44], functions/methods coverage [58], and modified condition/decision coverage [59]. Overall, coverage-based heuristics

can be grouped into two sub-groups: total coverage and additional coverage [60]. At each prioritization step, the former selects the test case with the highest coverage, while the latter selects the test case with the highest coverage of entities not yet covered by higher-priority test cases.

Extracting precise coverage information at a reasonable cost is the major drawback of coverage-based heuristics. Coverage information can be collected either by static or dynamic analysis. Static analysis techniques are more easily amenable to collecting coverage and conducting impact analysis. However, they overestimate the coverage and impact of test cases, and relying solely on them may undermine the effectiveness of TCP. Dynamic analysis is the most precise technique and requires the execution of tests on the instrumented (i.e., binary or source code instrumentation) version of the SUT. Dynamic analysis techniques are platform-dependent, time-consuming for large codebases, and may not be applicable for real-time systems because code instrumentation may cause timeouts or interrupt normal test execution. Overall, dynamic analysis techniques are difficult or even impossible to apply in practice, specifically in the CI context, mainly due to computational overhead and applicability [13–17]. Thus, our work only investigates the use of static coverage information. We conjecture that mixing static coverage information with other relevant information, such as test cases' execution history, can compensate for the static analysis techniques' tendency to overestimate.

Heuristics that are based on test case execution history assume that previously failed test cases are more likely to fail again. For example, Kim and Porter [57] proposed an execution history heuristic that calculates ranking scores based on the average failure rate of test cases. The execution history is easily accessible, and some studies such as [14, 39] reported that the effectiveness of execution history heuristics is almost the same as ML-based TCP. However, as discussed by Pan et al. [7], the reported training of ML models is problematic due to (1) limited feature sets, (2) unbalanced datasets containing only a small number of failed test cases, and (3) suboptimal ML techniques in the TCP context. To further investigate, we performed a comparison between the best heuristics based on the failure history of test cases and the best ML models, as discussed in Section 2.4.

ML-based TCP: Work in this group trains ML models based on features collected from different sources, such as test execution history, to prioritize test cases. Compared to heuristics that are often defined based on a single feature, ML models can be trained on a set of features from different sources. This enables ML models to deal with the complexities of the TCP problem for complex systems in the CI context. Also, ML techniques either support online training or can be retrained to account for the dynamic nature of CI [61]. However, heuristics are static, and there is no standard procedure to tune them based on new changes.

Pan et. al [7] surveyed ML-based TCP techniques and reported that existing ML-based papers investigated a wide variety of ML techniques including reinforcement learning [14–16, 20, 62–64], clustering [65–71], ranking models [20, 38, 46, 72–81], and natural language processing [38,68,74,82,83]. However, each study only used a small number of features that are either easily collected or publicly published by existing work. Also, most existing works evaluated the proposed TCP techniques based on subjects with very low numbers of failed

test cases (e.g., 2% failure rate across subjects in [20]) and short regression testing time (average regression testing below 90 seconds). Having a sufficient number of failed builds is important for evaluating such TCP techniques and creating the balanced datasets required by many ML techniques. Further, applying TCP techniques to systems whose regression testing takes a short time is not practically beneficial and therefore not representative in an experimental context. Our study focuses on these three mentioned issues by investigating the use of a comprehensive feature set for training ML TCP models based on 25 realistic subjects whose test failure rates have a median of 14% and their regression testing time takes at least 5 minutes with a median of 12 minutes. In the following, we discuss the three studies closest to ours.

Bertolino et al. [20] analyzed the performance of ten ML algorithms, including three RL algorithms, for test prioritization in CI. Through experimental analysis, they showed that RL-based approaches to test case prioritization can adapt to new changes as their TCP effectiveness is less affected by changes. They also reported that MART [52], a pairwise ranking model that is trained using an ensemble of boosted regression trees, reaches the highest effectiveness among the ten ML and RL techniques. However, their study suffers from the issues discussed above. More specifically, they only use limited features that are defined based on the execution history of test cases and their complexities. They also evaluated their work only based on six subjects, all of which include very low numbers of failed test cases (only 49 failed cycles among 2.6k builds across all subjects) and short regression testing time (maximum of 22 seconds across subjects).

In a similar work to the previous one, Bagherzadeh et al. [18] conducted a thorough analysis of ten state-of-the-art deep reinforcement learning techniques in the TCP context. Through extensive empirical analysis, they showed that the best result from RL techniques can reach similar effectiveness as MART in the TCP context. This study is based on the same data set as that of Bertolino et al. [20]. Thus it suffers from the same issues (limited number of features and subjects) as discussed above.

Elsner et al. [39], investigated a more comprehensive set of features for training ML models in the TCP context. They found that features from test execution history work better than other types of features. They also showed that well-known and simple heuristics based on the failure rate of test cases often outperform complex ML models, which was unexpected. While the paper used a relatively large number of subjects with an adequate number of failed test cases, they only used subjects containing only 16 features and thus this study is far from being comprehensive. Indeed, it missed many essential features, including features based on static coverage analysis and the code complexity of test cases and the SUT. Our study shows that complexity features play an important role in training highly effective ML models for TCP. They also did not investigate the data collection time associated with features, which is necessary to perform trade-off analysis on the cost and benefits of features. Finally, they failed to use the state-of-the-art ML models in a TCP context (MART) as reported by previous studies [20]. Instead, they trained a point-wise ranking model that, according to existing studies (Bertolino et al. [20] and Bagherzadeh et al. [18]), provides lower effectiveness compared to pairwise ranking models such as MART. We believe that all these elements explain why the effectiveness of their ML models is lower than simple heuristics. We further investigated (Section 2.4) this issue by comparing the

results of MART models against the heuristics based on failure rate, which reaches the best effectiveness as reported by Elsner et al. [39]. The results show that the models trained either based on the full feature set or features related to the execution history outperform the heuristic model.

Overall, our work complements existing ML-based TCP studies by performing a comprehensive investigation of features (150), obtained from various sources, for training ML TCP models. This allows us to provide practical insights on the benefits and costs of ML-based techniques through extensive empirical analysis. Also, we provide a tool and a set of diverse subjects that can be used as a benchmark for future studies.

2.6 Conclusion

In this work, focused on ML-based Test Case Prioritization (TCP), we have carefully defined a comprehensive set of features aimed at predicting the probability of regression failure, based on related studies, and a data model that captures entities and their relations in a typical CI environment. The features are categorized according to common data collection sources and steps into three high-level groups: *REC* containing features capturing the test execution history, *TES* containing features characterizing the complexity of the test case source code, their changes, and their development process, and *COV* containing features that capture test case code coverage of changes in the system under test (SUT). We have developed a tool to collect these features for 25 carefully selected open-source projects in order to carry out an extensive experimental study to analyze the cost and benefits of the feature groups for ML-based TCP. The ultimate goal is to provide concrete and precise recommendations regarding what data to collect to effectively support TCP in a scalable way.

The results of our study show that:

- The data collection time of all features ranges between 0.1 to 11.7 minutes across subjects for each build. Also, the trained models for TCP based on the full feature set can achieve promising results in terms of APFD_C across most subjects, with an average of 0.82.
- Coverage-related features (i.e., feature groups with *COV* in their name) are the most expensive features to collect, while they have the least impact on the effectiveness of ML TCP models.
- *REC* features are the least expensive to collect, while models relying on them achieve an APFD_C close to models trained with the full set of features.
- Not retraining models for more than 11 builds leads to unstable APFD_C. Also, to achieve the best TCP effectiveness, the models should be retrained as frequently as possible.

Access to high-quality datasets is one of the main issues for devising and evaluating new TCP techniques. Thus, we also made our datasets publicly available, which can serve as a benchmark for future studies. In addition, we suggest to extend our data collection tool to support the data collection at the method level and investigate how using method-level data can impact the TCP techniques in terms of data collection cost and effectiveness. Further, extending the data collection and analysis to other CI tools such as GitHub Actions and other programming languages would be a useful endeavor.

Chapter 3

Automated Test Case Repair Using Language Models

This chapter focuses on addressing TO_2, which involves reducing test code evolution development and maintenance costs. The findings and methodologies presented in this chapter have been accepted by the *IEEE Transactions on Software Engineering (TSE)* journal [84].

3.1 Introduction

Testing is crucial for ensuring the quality of software systems [85,86]. However, maintaining test cases can be costly as developers often need to update them when the source code of the System Under Test (SUT) evolves, posing a maintainability challenge [5, 87–89]. Failing to do so can result in broken test cases that are no longer valid and, without proper maintenance, these broken test cases might be discarded, ultimately reducing the overall quality of the test suites. Two recent analyses of test case failures of 211 Apache software foundation projects [6] and 61 open-source projects [5] reported that broken test cases account for 14% and 22% of failures, respectively. Additionally, false alarms resulting from broken test cases, which are failures caused by the test code rather than the SUT, disrupt software build processes and waste valuable developer time [88, 90].

To illustrate the motivation and scope of the problem, we present two representative examples of broken test cases from real-world software projects in Figure 3.1. The first example (Figure 3.1a) demonstrates a scenario where the test fails at runtime due to a change in the type of exception thrown by the SUT. In this case, the test originally expected a *FailedPreconditionRuntimeException*, but after the code change, the SUT threw an *InternalRuntimeException* instead. The second example (Figure 3.1b) showcases a compilation error caused by a change in method visibility. In this case, the method *Columns.add* was modified from public to non-public, causing a test that relied on this method to fail compilation. These examples cover two main types of test breakages: runtime errors that arise during execution and compilation errors that prevent tests from running in the first place.

```
// Updates in the system under test (change of exception type)
- return new FailedPreconditionRuntimeException ( t ) ;
+ return new InternalRuntimeException ( t ) ;

// Repairing the broken test case
@Test
public void close() throws Exception {
  ByteBuffer buf = BufferUtils.getIncreasingByteBuffer(TEST_BLOCK_SIZE);
  Assert.assertEquals(TEST_BLOCK_SIZE, mWriter.append(buf));
  mWriter.close();
  - mThrown.expect(FailedPreconditionRuntimeException.class);
  - mWriter.append(buf);
  + Assert.assertThrows(InternalRuntimeException.class, () -> mWriter.append(buf));
}
```

(a) A broken test example causing a runtime error, from the *alluxio/alluxio* project at commit *6982d6c759*

```
// Updates in the system under test (Columns.add is not public anymore)
- public Columns add ( Column < ? > column ) {
+ Columns add ( Column < ? > column ) {

// Repairing the broken test case
@Test
public void testValidateColumns() {
  Schema schema = SchemaBuilders.schema().mapper("field1", stringMapper()).build();
  - Columns columns = new Columns().add(Column.builder("field1").buildWithComposed("value" ,
      UTF8Type.instance));
  + Columns columns = new Columns().addComposed("field1", "value", UTF8Type.instance);
  schema.validate(columns);
  schema.close();
}
```

(b) A broken test case example causing a compilation error, from the *stratio/cassandra-lucene-index* project at commit *fdd34e53*

Figure 3.1: Two real-world broken test examples causing compilation and runtime errors.

Overall, the failure of broken test cases accounts for a significant portion of test case failures, negatively impacting the quality of testing. To address this challenge, researchers have developed two complementary groups of techniques [8]: detection and repair. The main focus of our study is the latter. It aims at automatically repairing broken tests and often uses source code analysis and search-based techniques [88,90–93]. On the other hand, detection focuses on determining whether a test failure results from a broken test case and pinpoints the specific location of the breakage within the test code [94–98]. Apart from the aforementioned studies on broken tests, which necessitate access to and analysis of the source code of the SUT, there exist methods to repair broken black-box GUI tests solely through GUI analysis [99, 100].

Existing automated test repair studies show limitations from both methodological and evaluation perspectives, restricting their applicability across diverse software systems and repair scenarios. Methodologically, these approaches can be classified into heuristic, rule-based, or search-based, and static or dynamic analysis-based techniques. However, these

methods often struggle with generalization (they rely on project-specific data), scalability, and broad applicability. They are typically tailored to specific programming languages or focus narrowly on particular types of test repairs, limiting their application. For instance, *ReAssert* [90] focuses solely on repairing test oracles (assertions). While static or dynamic analysis-based techniques, such as symbolic execution in *TRIP* [93], involve a more generic approach with a broader scope, these methods are often challenging to implement for different programming languages and are expensive to scale for large codebases. From an evaluation standpoint, existing studies rely on benchmarks that are limited in size and diversity, raising concerns about the validity of their evaluations. For example, *TRIP* [93] is evaluated on only 91 broken tests across four projects. Further, most of these studies are not fully reproducible because their source code or benchmarks are not publicly accessible.

A recent adaptation of language models for source code shows satisfactory results for different software engineering tasks such as code generation, code completion, and program repair [101]. Motivated by these studies, we propose TARGET (TEST **R**EPAIR **GE**NERA**T**OR) [10], an approach that leverages language models to automatically repair broken test cases. Our solution is based on the insight that certain modifications in the source code of a SUT warrant corresponding updates in the associated test case source code. Thus, we formulate the test repair problem as a language translation task via a two-step approach. The first step focuses on identifying and collecting essential data that characterize the test breakage. The second step utilizes the gathered information to shape inputs and outputs for fine-tuning a given language model for the test repair task.

We conduct an extensive experimental analysis and assess TARGET by creating TARBENCH [11], a benchmark comprising 45,373 broken test repairs across 59 distinct projects, making it by far the most comprehensive benchmark to date in this application context. We demonstrate that fine-tuning a language model using TARGET for repairing broken tests achieves an exact match accuracy (EM) of 66.1% and a plausible repair accuracy (PR) of 80%. The EM denotes repairs perfectly matching the ground truth, while the PR denotes repairs that execute successfully. TARGET significantly outperforms the baseline, which is the best language model when it is not fine-tuned with crucial context information for the test repair task, by a substantial margin of 37.4 EM percentage points. To maximize the practicality of TARGET, we introduce and evaluate a model to predict the reliability of repairs generated by TARGET, helping developers to decide whether to trust them or not. Furthermore, we investigate the generalizability of TARGET by showing that a model fine-tuned on specific projects can be effectively applied to other software projects with an acceptable margin of EM loss. This is in contrast to existing work on broken test repair that necessitates project-based analysis.

Overall, the main contributions of this work are as follows.

- **TaRGET: An automated broken test case repair technique using language models.** TARGET's fine-tuned language model achieves an EM and PR of 66.1% and 80%, respectively, a significant achievement given the state of the art. This outperforms the application of existing language models without leveraging test repair context data, which only yield an EM of 28.7%. Moreover, TARGET overcomes

the methodological limitations of prior studies, as it is not restricted to specific programming languages and supports a broader range of repair types. Unlike existing approaches, TARGET directly utilizes code changes in the SUT through effective prioritization and selection, an aspect that has remained unexplored in previous research.

- **TaRBench: A large and comprehensive benchmark of broken tests and their repair.** As previously discussed, existing benchmarks in the literature are limited in both size and diversity, making them insufficient for effectively training language models. To overcome this limitation, we developed a tool capable of extracting broken tests and their corresponding repairs from open-source projects. Using this tool, we constructed TARBENCH, a comprehensive and high-quality benchmark with including 45,373 instances, which serves as the foundation of our study. We believe that TARBENCH will be valuable to researchers in this field, and we have made both the tool and dataset publicly available for future use[1,2].

- **Empirical evaluation of TaRGET.** The study addresses three research questions that concern evaluating TARGET using different configurations, analyzing instances where TARGET fail to generate a correct repair, investigating ways to maximize the practical usage of TARGET by predicting whether repairs can be trusted, and assessing the generalizability of TARGET and the amount of training data needed to fine-tune it for test repair.

The rest of this chapter is organized as follows. Section 3.2 introduces the foundational concepts which our work relies on. Following that, Section 3.3 describes our approach in detail. In Section 3.4, we introduce our benchmark and present the results of our experimental study, addressing three key research questions. Section 3.5 examines and compare existing studies relevant to our work. Lastly, Section 3.6 summarizes our findings and outlines potential paths for future enhancements.

3.2 Background

We address the problem of repairing broken test cases by utilizing language models. This is motivated by the idea that automated program repair, which is similar to automated test repair, can be approached by translating faulty code snippets into repaired code snippets [102]. The rest of this section will elaborate on this idea and act as a primer on the concepts of language models, code language models, pre-training, fine-tuning, and other concepts upon which our work relies.

[1]TARGET: https://github.com/Ahmadreza-SY/TaRGet

[2]TARBENCH: https://doi.org/10.6084/m9.figshare.25008893

3.2.1 Language Models

Language models aim to perform language-based tasks by modeling the likelihood of token sequences in order to predict an output, which is often a generated sequence of tokens [103]. Code Language Models (CLMs) are a specialized category of language models designed for code-related tasks, forming the focus of our study. CLMs are often trained on multiple programming languages alongside natural language texts [104], [105], [106]. Consequently, they are well suited for code-related tasks, such as code summarization, code generation, program repair, and code translation.

3.2.1.1 Language Model Architecture

The architecture of language models are largely inspired by the pairing of a self-attention layer with a feed forward network, originally proposed by Vaswani et al. [107]. These self-attention layer and feed forward network pairs are then used to construct encoders and decoders, which can be combined to construct language models. There are two variations of language model architectures that are relevant to this study: encoder-decoder models, and decoder-only models. Encoder-only models also exist but they are not utilized in our work, since they require an additional decoder to perform generative tasks [102], and are thus not described here.

Encoder-decoder models are models which use both an encoder and a decoder. The sequence-to-sequence Transformer [107] is an example of such a model, and is the basis on which many modern language models were built. In an encoder-decoder model, the encoder takes the input token sequence, and converts it into a hidden, intermediary representation, which is then used by the decoder to generate the output token sequence [101], [102].

The encoder generally consists of several layers, each of which has two sub-layers: a multi-head self-attention mechanism, and a fully connected feed-forward network. The decoder is also constructed of several layers, each of which has three sub-layers: a masked multi-head attention mechanism, a multi-head self-attention mechanism, and a fully connected feed-forward network [107]. The multi-head self-attention mechanism performs linear transformations on the inputs, resulting in several projections of the the input, which are recombined to give the final output [107]. The decoder generates the final output by building a sequence of tokens one token at a time in an auto-regressive process, selecting the next output token based on the model input and the current, incomplete output sequence. An example of an encoder-decoder model that we use in this study is CodeT5+ [105].

Decoder-only models are models which only have a decoder with no encoder, as the name suggests. These models start from an initial state, and gradually build an output sequence of token, attending to the previously generated output tokens [101], [108]. The decoders in this type of model work similarly to those used in the encoder-decoder architecture. They rely on an ability to understand the target language and its nuances [101]. An example of a decoder-only model that we use in this study is CodeGen [106].

3.2.2 Pre-Training, Fine-Tuning, and Inference of Code Language Models

It is common for language models to be pre-trained on large corpora of general-purpose data for a variety of tasks, resulting in a general model that can then be adapted to specific down-stream tasks [109]. For language models, there are five general types of pre-training tasks: mask language modeling, denoising autoencoder, replaced token detection, next sentence prediction, and sentence order prediction [109]. These pre-trained models are then adapted to specific tasks through the process of fine-tuning, during which the model is trained on additional data which is focused on the tasks of interest.

Fine-tuning a model is simply performing further training on a pre-trained model so that it learns to perform specific tasks. Fine-tuning a model this way has benefits, as the model retains the vocabulary and any insights it learned during pre-training, and thus can be trained on a more limited dataset [102]. Fine-tuning is commonly performed with the goal of minimizing the model's cross-entropy loss on a specific task [110]. This is done by training the decoder on iterations of the output sequence. The decoder considers the input representation and the previously generated output sequence to determine the next token in the output sequence. The loss is determined based on what token the decoder determines is next in the sequence, and is minimal when the decoder selects the correct token.

As a practical example of pre-trained models being fine-tuned, consider the PLBART CLM, which was pre-trained on a corpus of data to perform the task of denoising both natural language and programming languages (Java and Python in particular) [104]. This pre-trained model was then fine-tuned several different times to perform several tasks, including code generation, code summarization, and code translation, amongst other tasks [104].

At the inference stage, given that the model output can consist of any sequence of tokens from the target language, the output space is quite large. To address this, generative models use different techniques to strategically select the output. In line with previous program repair studies [102], we adopt the beam search ranking strategy to prioritize the model output. The beam search algorithm operates by maintaining a set of k sequences, where k refers to the beam size. At each generation step, the decoder predicts the probability distribution of tokens in the vocabulary based on the previous tokens for each of the k sequences. Next, the beam search selects the top-k tokens with the highest scores. The score for a token is calculated by multiplying its probability with the probabilities of the previous tokens in the sequence. This search process continues until either the end-of-sequence (EOS) token is encountered or a maximum output sequence length is reached.

3.3 Approach: TaRGET

Let V_i and V_{i+1} denote two consecutive versions of a SUT. Let T_i denote a test case that is broken due to the code changes introduced in V_{i+1} relative to V_i, and let L be the set

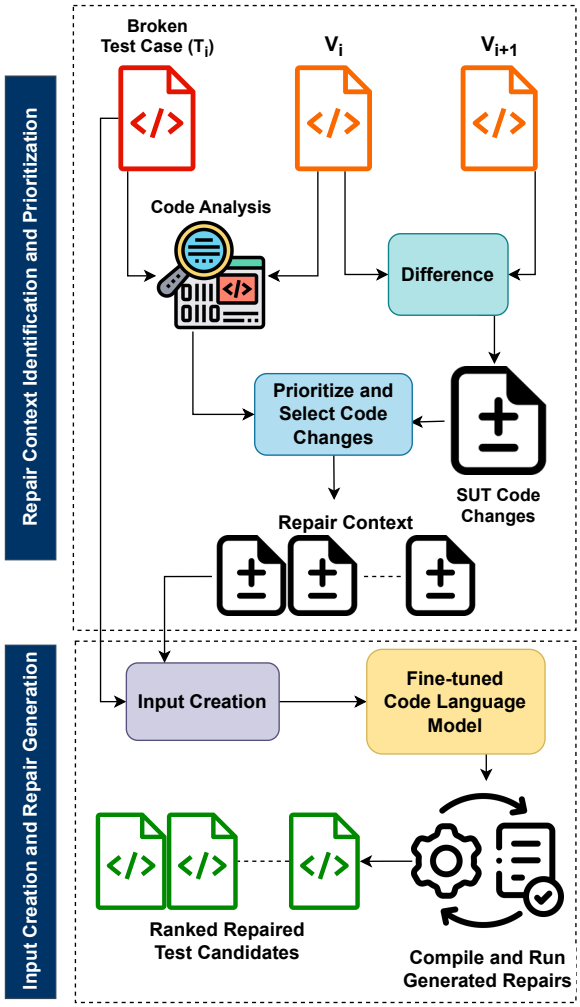

Figure 3.2: An overview of TARGET given a broken test case and two versions (V_i and V_{i+1}) of the system under test (SUT).

of breakage lines, which are the specific lines in T_i that require repair. We define our approach, TARGET, as follows.

$$\text{TARGET}(T_i, L, V_i, V_{i+1}) \rightarrow T_{i+1}$$

where T_{i+1} is the repaired version of T_i. Note that we distinguish between *error lines* and *breakage lines*. Error lines are the lines in T_i where compile or runtime errors occur, whereas breakage lines refer to the lines containing the root cause of the test case breakage and where repairs should be made. These two sets of lines may not necessarily overlap.

Figure 3.2 presents an overview of TARGET, encompassing two high-level steps: (1) repair context identification and prioritization, and (2) application of CLMs to repair tests. The first step focuses on identifying and gathering essential data that characterize the breakage of tests. The second step uses the collected information from the first step

to shape the inputs and outputs for utilizing a given CLM (fine-tuning and inference), specifically for test repair. It also addresses the generation of test repairs. In this section, we discuss the details of each step.

Figure 3.3 illustrates an example of test repair for the *testDeposit* test case, testing the *deposit* method within the *BankAccount* class. Figures 3.3a and 3.3b display the original source code for *deposit* and *testDeposit*, respectively. Also, Figure 3.3c contains modifications to the *BankAccount* class, introducing currency support. These changes include adding a new field—*currency*—to the class, and updating the account balance based on the exchange rate during each deposit. These changes caused *testDeposit* to fail at lines 3 and 4 in Figure 3.3b, indicating the breakage location. Consequently, as shown in Figure 3.3d, *testDeposit* is repaired by including the currency value in both the *BankAccount* constructor and the *deposit* method. This example serves as a running example of test repair throughout the chapter, and we will illustrate TARGET using this example whenever possible.

Admittedly, an ideal automated test repair solution should address both breakage localization and repair, which are both challenging tasks. As shown in Section 3.4.2.6, breakage localization, an active area of research often termed "fault localization", is not straightforward. TARGET contributes to a comprehensive automated solution, by effectively addressing the automated repair problem.

Even when knowing the breakage lines, determining the appropriate repair is often complex. In the example shown in Figure 3.1b, although the breakage line (line 9) is given, developers face multiple repair options, necessitating careful consideration. Furthermore, our analysis of TARBENCH in Section 3.4.5 highlights cases with a large search space for potential repairs. Finally, TARGET, regardless of repair complexity, provides automation which reduces the time required to interpret breakage lines, test code, and SUT code, alleviating the repetitive and tedious nature of the task.

3.3.1 Repair Context Identification and Prioritization

Merely examining the test case code and breakage location without considering the changes in the SUT is insufficient to repair a broken test. It is essential to examine all changes made to the SUT, and detect the ones that led to the test case breakage. We refer to these changes in the SUT source code as the *repair context*. Adding additional context is also a common practice when language models are applied to automated program repair, where the context will consist of code related to the faulty system code [102].

While we employ a comprehensive approach to gather all relevant information for the repair context, it is important to note that current CLMs come with limitations regarding input size, also known as *context window*. This necessitates a thoughtful selection of the repair context for input. Therefore, we introduce prioritization techniques to ensure that CLMs receive the most crucial information.

While test repair context identification and fault localization share similarities, a critical distinction lies in the latter's reliance on oracles; fault localization depends on test suites

```java
public class BankAccount {
  private int balance = 0;
  public int getBalance() { return
     balance; }
  public int deposit(int amount) {
    balance += amount;
  }
}
```

(a) System Under Test (V_i)

```java
@Test
void testDeposit() {
  Bank account = new BankAccount();
  account.deposit(500);
  assertEquals(500,
     account.getBalance());
}
```

(b) Test Case (T_i)

```
public class BankAccount {
  private int balance = 0;
  + private final String currency;
  + public BankAccount(String currency) { this.currency = currency; }
  public int getBalance() { return balance; }
  - public int deposit(int amount) {
    - balance += amount;
  + public int deposit(int amount, String depositCurrency ) {
    + balance += amount * Exchange.getRate(depositCurrency, currency);
  }
}
```

(c) Code changes resulting in V_{i+1}

```
@Test
void testDeposit() {
  - Bank account = new BankAccount();
  - account.deposit(500);
  + Bank account = new BankAccount("USD");
  + account.deposit(500, "USD");
  assertEquals(500, account.getBalance());
}
```

(d) Test Case Repair resulting in T_{i+1}

Figure 3.3: Test case repair example

to identify faults, whereas our context lacks a reliable oracle. Moreover, fault localization aims to pinpoint faulty statements, while our approach focuses on identifying correct code changes that cause obsolete test cases to fail. Given these differences, we opted for simple and widely validated heuristics based on static code analysis and text similarity to identify repair contexts, which we describe in this section.

We recognize that SUT code changes alone may sometimes be insufficient to repair a test case. However, since SUT changes are the root cause of the test case breakage, they inherently contain useful information and are relevant. The extent to which additional repair context beyond SUT changes is required remains uncertain. Furthermore, the unchanged project-level context is significantly larger than the changed context, making the identification and selection of the most relevant context a challenging task. While this issue lies beyond the scope of our study, it represents a promising avenue for future research.

In the following sections, we shed light on the details of TARGET for repair context identification and prioritization.

3.3.1.1 Repair Context Identification

First, we automatically identify the information linked with the repair context of a broken test case by following the steps below, assuming (1) a broken test case T_i is caused by changes made to the SUT from version V_i to V_{i+1}, and (2) a *hunk* defines a specific section within a source code file, consisting of one or more adjacent lines that have been deleted, added, or changed.

Identifying Method- and Class-Level Hunks. We identify method-level hunks, denoted as M, by comparing the code of methods within the same class between V_i and V_{i+1}. Changes that occur within the class but outside of methods are identified as class-level hunks, denoted as C. When comparing source files, even if a change spans multiple methods, we treat it as separate hunks, each corresponding to a single method. According to this definition, class-level hunks do not intersect with method-level hunks ($M \cap C = \emptyset$). We identify methods using their fully qualified names, which include the package name, class name, method name, and argument types. If a method's name changes, we match it with the most similar fully qualified name within the same class, using the *Jaro–Winkler* string distance metric [111] to measure similarity. This measure is effective for matching strings with minor variations, and it emphasizes similarities at the beginning of the string, making it ideal for minor method name changes or method signatures with argument changes at the end of the string.

In the context of the running example (Figure 3.3c), we have identified two hunks within the *BankAccount* class: (1) h_1, which includes two added lines (lines 3 and 4), and (2) h_2, which consists of two deleted lines (lines 6 and 7) and two added lines (lines 8 and 9). The method-level hunk set of the running example is $M = \{h_2\}$, and the class-level hunk set is $C = \{h_1\}$.

Generating Method and Class Level Call Graphs. We use the Spoon library [112] to conduct static code analysis on V_i, generating two call graphs, namely method-level (G_m) and class-level (G_c) call graphs, for T_i. G_m encompasses all SUT methods or constructors that are directly or indirectly invoked by T_i. In this graph, T_i acts as the root, and each method invocation forms an edge that connects the calling method to the called method. In the context of the running example, the root node of G_m is *testDeposit()* method. This root node directly connects to three nodes: *BankAccount.deposit(int)*, *BankAccount()*, and *BankAccount.getBalance()*.

Similarly, G_c encompasses all classes that are directly or indirectly used by T_i. Similar to G_m, T_i is the root of G_c. However, in G_c, non-root nodes represent classes, and each usage of a class (i.e., calling a method or a constructor of the class) is considered as an edge connecting the using node to the used class. The G_c of the running example only has the test as the root node connected to the *BankAccount* class as the non-root node.

Creating the Repair Context Sets. We create three sets of hunks to serve as the repair context for our approach: R, R_m, and R_c. R includes all changes in the SUT, including both method-level and class-level hunks, and is defined as:

$$R = M \cup C$$

$R_m \subset R$ is a set of method-level hunks, where each hunk corresponds to a method that is directly or indirectly called by T_i, as indicated by G_m. It is defined as:

$$R_m = \{h \in M \mid h \text{ is covered by } T_i \text{ in } G_m\}$$

$R_c \subset R$ is a set of both method-level and class-level hunks that are within classes covered by T_i, as indicated by G_c. However, R_c specifically excludes any hunks already in R_m. It is defined as:

$$R_c = \{h \in (M \cup C) \mid h \notin R_m \wedge h \text{ is covered by } T_i \text{ in } G_c\}$$

In the context of the running example (Figure 3.3c), and based on the two hunks identified earlier, $R = \{h_1, h_2\}$. Since the *BankAccount.deposit(int)* method is included in the G_m of *testDeposit()*, $R_m = \{h_2\}$, as h_2 corresponds to the *deposit* method. Furthermore, since the *BankAccount* class is part of the G_c of *testDeposit()*, $R_c = \{h_1\}$, as h_1 is within that class and is not part of R_m.

We utilize the repair context sets (R, R_m, and R_c) and the call graphs (G_m and G_c) to prioritize and select the hunks most relevant to the repair task. These elements enable us to identify the structural code relationships between the test cases and the SUT. Further details on the hunk prioritization process are provided in the following section.

3.3.1.2 Repair Context Prioritization

As discussed, CLMs have limitations on their input size, which entail the careful selection of the repair context (R) as input. In certain scenarios, the token size of R may exceed the input size limit. Also, CLMs may produce different outputs based on the order of the repair context provided in the input. Since exploring all possible orderings is computationally infeasible, we prioritize placing the information that we consider most important at the beginning of the input sequence. Therefore, we employ a prioritization strategy for the hunks in R to select the top k most relevant hunks ($1 \leq k < |R|$) and create R'. We define the hunk prioritization criteria in the following.

Call Graph Depth. To determine the call graph depth (d) for a given hunk $h \in (R_m \cup R_c)$, we examine its associated node n within the call graph G (G_m or G_c) related to the broken test case T_i. The value of d is calculated by counting the number of edges between the root node of G and n. When d equals 1, it shows that the code within h is directly invoked by T_i and any modifications in h are highly relevant to T_i. Thus, lower call graph depth values indicate a closer relationship and are given higher priority, with 1 being the best value (direct relationship) for d. Higher values refer to indirect relationships, making them less relevant. For the running example, both h_1 and h_2 have a d value of 1. This corresponds to the *deposit* method and the *BankAccount* class, both directly accessed from the *testDeposit()* test case.

Context Type. The context type (CT) for h determines whether $h \in R_m$ or $h \in R_c$. If h belongs to R_m, CT represents a *method*; if h belongs to R_c, CT represents a *class*. When selecting hunks based on context types for effective test case repair, hunks with the

method context type offer greater potential value compared to those with the *class* context type. This is because method-level relations are more fine-grained, reducing the selection of hunks that are irrelevant to the test code. Therefore, we assign a higher priority to hunks with a *method* context type. In the running example, h_2 has a method context type, while h_1 has a class context type.

TF-IDF Similarity. In addition to the structural associations observed between the test case and the SUT, there are textual similarities among variables, methods, classes, and literal values that are not captured by call graphs. Thus, these similarities can play a crucial role in identifying the most relevant repair context. We measure text similarity by comparing the changed lines of the hunks in the SUT against (1) L, the breakage lines of T_i (*Breakage TF-IDF Similarity*), or (2) the complete code of T_i (*Test TF-IDF Similarity*), using the TF-IDF (term frequency-inverse document frequency) algorithm [29]. First, we use the tokenizer of the code language model to convert the code to tokenized vectors. This tokenization process produces $|R|$ documents, each corresponding to a hunk $h \in R$, alongside an additional document for the selected lines of T_i (whether breakage lines or all lines). Then, for each document, TF-IDF creates a normalized vector of vocabulary size (the count of unique tokens) by evaluating the significance of the document's individual tokens. This evaluation is based on token frequency across all documents and its occurrence within individual documents. We compute the cosine similarity between the TF-IDF vector of each $h \in R$ and the selected lines of T_i, prioritizing the hunks based on the highest similarity to the lowest.

3.3.2 Input Creation and Repair Generation

At this stage, we have gathered all the necessary information required to create the input and the output to fine-tune a CLM and generate repairs for broken tests. There is no single way for shaping the input and output format (IO) for CLMs and the selection of a suitable IO for CLMs necessitates experimental analysis. Thus, in this work, we explore four different approaches to shape the IO of test case repairs. An overview of the characteristics of the IOs is shown in Table 3.1. In the following, we first provide details for the IOs that we explored, explaining the input's repair context, the hunk prioritization and representation, and the output. Finally, we explain the generation process of test repairs.

3.3.2.1 The Input

For IO_1 (*Base*), we select the repair context from $R_m \cup R_c$ and adopt three criteria sorted by their respective priorities: *Call Graph Depth*, *Context Type*, and *Breakage TF-IDF Similarity*. As discussed, call graph depth and context type specifically capture the structural code relations between the test code and the repair context, while TF-IDF similarity captures the text-based code relations. We integrated the three criteria to differentiate between hunks in cases where two hunks share the same value for one criterion. Specifically, when the call graph depth and context type of two hunks are equal, their prioritization

	Input-output Format	Repair Context	Hunk Prioritization	Hunk Represenation	Output
IO_1	Base	$R_m \cup R_c$	CGD, CT, BrkSim	Line-level	Code Sequence
IO_2	Text-based Similarity Hunk Ordering	R	BrkSim, Rep, TSim	Line-level	Code Sequence
IO_3	Word-level Hunk Representation	R	BrkSim, Rep, TSim	Word-level	Code Sequence
IO_4	Edit Sequence Output	R	BrkSim, Rep, TSim	Word-level	Edit Sequence

Table 3.1: Overview of input and output formats (***IO***s) in test repair generation models. Call Graph Depth (CGD), Context Type (CT), Breakage TF-IDF Similarity (BrkSim), Repetition (Rep), and Test TF-IDF Similarity (TSim) are the characteristics used for hunk prioritization. For detailed definitions of these characteristics and others, please refer to Section 3.3.

is determined by their similarity criterion. We used this order under the assumption that structural code relations are consistently accessible, given the inherent nature of programming languages. However, text-based code relations depend on developers' coding styles and the use of meaningful names across various sections of the code. Consequently, structural relations generally serve as a more dependable indicator of associations between hunks and test code compared to text-based relations.

For IO_2 (*Text-based Similarity Hunk Ordering*), IO_3 (*Word-level Hunk Representation*), and IO_4 (*Edit Sequence Output*), we adopt three different criteria sorted by their respective priorities: *Breakage TF-IDF Similarity*, *Repetition*, and *Test TF-IDF Similarity*. This prioritization emphasizes text-based similarity between the test code and the repair context. The repetition criterion measures how frequently a hunk is repeated within the repair context. When there are multiple occurrences of the same change across the source code, it is even more likely to be valuable for repairing the broken test case.

The repair context selection for IO_2, IO_3, and IO_4 is not limited to R_m or R_c; instead, we select the hunks from R. We do this to mitigate potential imprecision in using call graphs generated through static analysis. Unlike code execution and dynamic analysis, static analysis can introduce imprecision in type or method inference as well as in pointer analysis. Additionally, we aim to investigate scenarios where SUT changes beyond the scope of call graphs are required for repairing the test. Including all hunks may introduce irrelevant repair context. We mitigate this issue by prioritizing hunks based on their similarity to the test code. Additionally, removing crucial context can be more detrimental than including irrelevant context. Therefore, we experimented with a more inclusive repair context selection that involves all hunks.

For all ***IO***s, the input is created per each test case T_i as a sequence consisting of two parts: the test context (TC) and the repair context (RC). Each part is prefixed with a special token. The input sequence I is thus formatted as follows:

$$I = \text{[<TESTCONTEXT>]}\ TC\ \text{[<REPAIRCONTEXT>]}\ RC$$

The special tokens [<TESTCONTEXT>] and [<REPAIRCONTEXT>] mark the start of their respective parts, indicating their position within the input. The TC consists of the complete source code of the broken test method, with the breakage lines enclosed by the two special tokens [<BREAKAGE>] and [</BREAKAGE>]. Meanwhile, the RC consists of code hunks

```
1  [<TESTCONTEXT>]
2  @Test
3  public void test() {
4    [<BREAKAGE>]
5    BankAccount account = new BankAccount();
6    account.deposit(500);
7    [</BREAKAGE>]
8    assertEquals(500, account.getBalance());
9  }
10 [<REPAIRCONTEXT>]
11 [<HUNK>]
12 public int deposit(int amount [<ADD>] , String depositCurrency [</ADD>]) {
13   balance += amount [<ADD>] * Exchange.getRate(depositCurrency, currency) [</ADD>] ;
14 } [</HUNK>]
15 [<HUNK>] [<ADD>]
16 private final String currency;
17 public BankAccount(String currency) { this.currency = currency; } [</ADD>] [</HUNK>]
```

(a) Input Format

```
BankAccount account = new BankAccount("USD");
account.deposit(500, "USD");
```

(b) Output Format

Figure 3.4: Example input format with word-level hunk representation and the expected code sequence output.

```
[<HUNK>] [<DEL>]
public int deposit(int amount) {
  balance += amount;
[<ADD>]
public int deposit(int amount, String depositCurrency) {
   balance += amount * Exchange.getRate(depositCurrency, currency); [</HUNK>]
```

Figure 3.5: Example of the line-level representation of a code hunk (h_2 in our running example) in the input.

from R' (defined in Section 3.3.1.2), arranged in order of priority as discussed above. In Figure 3.4a, the lines 1-9 show the TC input format, and lines 10-17 show the RC for our running example.

The TC part is consistent across all IOs, while the hunk representation in the RC varies. For IO_1 and IO_2, we use a line-level hunk representation, whereas for IO_3 and IO_4, we use a word-level representation. In both line-level and word-level cases, the hunk is enclosed with the special tokens [<HUNK>] and [</HUNK>].

For line-level representation, the sequence starts with [<DEL>], listing all deleted lines of the hunk, followed by [<ADD>], listing all added lines. If a hunk solely comprises deleted or added lines, only the existing set, along with its corresponding special token, is included. Figure 3.5 illustrates the line-level representation of h_2.

In the line-level representation, deleted and added lines often share many tokens, caus-

ing redundancy in the input and requiring the CLM to discern changed tokens. In contrast, the word-level representation combines deleted and added lines, enclosing only the changed words (tokens) with special tokens. This representation uses special tokens to enclose deleted words within [<DEL>] and [</DEL>] and added words within [<ADD>] and [</ADD>] for each group of changed words. Lines 11-14 and lines 15-17 in Figure 3.4a show the word-level representations of hunks h_2 and h_1, respectively.

3.3.2.2 The Output

For IO_1, IO_2, and IO_3, the code sequence of the repaired lines, derived from applying changes to the breakage lines, constitute the expected output of the CLM. Figure 3.4b shows the code sequence output of our running example. Figure 3.4b is a code transformation of lines 5 and 6 in Figure 3.4a.

For IO_4, we use the edit sequence encoding as the output. This is a series of replacements that, when applied to the breakage lines of code, results in the repaired lines, and is derived from the unambiguous edit sequence formulation proposed by Zhang et al. [108]. They found that modeling code changes as edit sequences significantly outperforms state-of-the-art approaches for the multilingual code co-evolution task. Given the similarity between the task they addressed and our test case repair task in this study, we decided to experiment with edit sequences. This output format includes solely the edit sequence, which is then applied to the test breakage lines to create the repair candidate. The key component of this formulation is selecting a sequence of tokens to be replaced from the breakage lines, such that the selected sequence is unique within the breakage lines. This ensures that each replacement can be performed without needing additional location information. An edit sequence is assumed invalid and discarded if the sequence of tokens to be replaced is not uniquely identifiable, as this introduces ambiguity. Each replacement E in an edit sequence with the following format:

$$E = \text{[<replaceOld>]}\ s\ \text{[<replaceNew>]}\ n\ \text{[<replaceEnd>]}$$

The special tokens [<replaceOld>], [<replaceNew>], and [<replaceEnd>] define the replacement, indicating its components. Since an edit sequence can consist of multiple replacements, [<replaceEnd>] is used to indicate that the preceding replacement is completely defined. Component s indicates the target token sequence to be replaced, which must be uniquely identifiable within the breakage lines of code. Component n indicates the token sequence that s is to be replaced with. A simple example of an edit sequence encoding can be seen in Figure 3.6a.

On top of [<replaceOld>] and [<replaceNew>], we used additional pairs of special tokens to give hints about the replacement when the sequence of changed tokens is not uniquely identifiable within the breakage lines of code. To find a unique s within the breakage lines, additional tokens may need to be prepended and/or appended to the changed code. In these cases, s and n will have a shared set of tokens that remain the same throughout the edit sequence, and the special replace tokens indicate the relative position of the shared tokens. An example of this scenario can be seen in Figure 3.6b.

Ground Truth Test Repair

```
- account.deposit (amount, "EUR" ) ;
+ account.deposit (new Money ( amount ), "EUR" ) ;
```

Edit Sequence Output

```
[<replaceOld>] amount
[<replaceNew>] new Money ( amount )
[<replaceEnd>]
```

(a) A simple edit sequence where the changed tokens are uniquely identifiable.

Ground Truth Test Repair

```
- BankAccount account = new BankAccount( ) ;
+ ChequingAccount account = new ChequingAccount( ) ;
+ account.setCurrency ( "USD" ) ;
```

Edit Sequence Output

```
[<replaceOldKeepAfter>] BankAccount account
[<replaceNewKeepAfter>] ChequingAccount account
[<replaceEnd>]
[<replaceOldKeepBefore>] new BankAccount
[<replaceNewKeepBefore>] new ChequingAccount
[<replaceEnd>]
[<replaceOldKeepBefore>] ;
[<replaceNewKeepBefore>] ;
account.setCurrency ( "USD" ) ;
[<replaceEnd>]
```

(b) A complex edit sequence requiring extra tokens for unique identification, also showing the handling of new lines.

Figure 3.6: Edit sequence output encoding examples

The example displays a code change where the *account* variable's type is changed to *ChequingAccount*. This requires updating the *BankAccount* text in two places, necessitating a clear definition of where each replacement should occur. As each replacement must identify a unique sequence of tokens, both instances of replacing the *BankAccount* token require additional tokens to ensure uniqueness. Since a replacement specifying *[<replaceOld>] BankAccount [<replaceNew>] ChequingAccount [<replaceEnd>]* would be ambiguous (as it could be applied to multiple locations and it is unclear which locations it should be applied to), it is considered to be invalid. Instead, the two instances of *BankAccount* are each specified with a unique replacement. In the first instance, the *account* token follows *BankAccount* to form a uniquely identifiable sequence, using the special tokens [<replaceOldKeepAfter>] and [<replaceNewKeepAfter>] to indicate that a sub-sequence of the end part remains unchanged. This results in a replacement of *[<replaceOldKeepAfter>] BankAccount account*

[<replaceNewKeepAfter>] ChequingAccount account [<replaceEnd>], which is a valid replacement since it can only be applied to one location in the test. Similarly, the second replacement of the *BankAccount* token includes the *new* token before *BankAccount* to create a distinct sequence, using the special tokens [<replaceOldKeepBefore>] and [<replaceNewKeepBefore>] to signify that a sub-sequence of the beginning part remains unchanged. These special tokens also guide the insertion of new statements. Since new statements do not change any old tokens, the edit sequence replaces the end of the preceding statement (the semicolon) with the semicolon followed by the new line.

It should be noted that while a valid edit sequence can be constructed using only [<replaceOld>] and [<replaceNew>], the other special tokens (such as [<replaceOldKeepBefore>]) are used to delineate between location indicators and actual repairs. These special tokens are used with the goal of helping the model focus on the tokens which comprise the actual repair, rather than the tokens which are used to indicate the repair's location. As an example, *[<replaceOldKeepAfter>] BankAccount account [<replaceNewKeepAfter>] ChequingAccount account [<replaceEnd>]* is used over *[<replaceOld>] BankAccount account [<replaceNew>] ChequingAccount account [<replaceEnd>]* as the former specifies that the *account* token is not significant to the repair and is only a location marker, whereas the latter can be interpreted as indicating that *account* is an important part of the repair.

3.3.2.3 Input-Output Format Selection

In Sections 3.3.2.1 and 3.3.2.2, we described the four IOs selected for our experiments. As outlined in Table 3.1, each IO is defined by four parameters, with each parameter having two possible values, as shown in Table 3.2. In this section, we explain (1) the rationale behind selecting these four parameters and assigning two values to each, and (2) why, out of the 16 possible combinations, we chose these four IOs. For clarity, we refer to the values for each parameter using the IDs specified in Table 3.2. For instance, using covered hunks as the repair context is denoted as rc_1. For example, IO_1 is defined by (rc_1, hp_1, hr_1, o_1).

Fine-tuning CLMs on 36k training instances is resource-intensive, particularly when it involves multiple iterations for different parameter value combinations. This makes it necessary to pre-select parameter value combinations that are likely to be effective. The performance of language models during fine-tuning depends significantly on the input context, the expected output, and the representation of this information. We carefully considered these aspects and identified four key parameters: the provided context (rc), the order of the context (hp), the representation of the context (hr), and the expected output (o). The rationale for the selection of each parameter is detailed in Sections 3.3.2.1 and 3.3.2.2. Also, formatting inputs or outputs based on the selected parameter values is computationally efficient, since it is relying on static analysis and straightforward text processing, while leading to diverse formats.

Not all 16 parameter combinations are feasible. (rc_2, hp_1) cannot be used together because some of the hunks in R, derived from using rc_2, are not covered by the test's

Parameter		ID	Values
RC	Repair Context	rc_1	Covered hunks ($R_m \cup R_c$)
		rc_2	All hunks (R)
HP	Hunk Prioritization	hp_1	Sorts by CGD, CT, BrkSim
		hp_2	Sorts by BrkSim, Rep, TSim
HR	Hunk Representation	hr_1	Line-level
		hr_2	Word-level
O	Output	o_1	Code sequence
		o_2	Edit sequence

Table 3.2: The parameters used to define IOs in Table 3.1. An IO is characterized by one value for each parameter.

call graph, making them impossible to prioritize with hp_1. This constraint excludes four combinations. While using (rc_1, hp_2) is technically feasible, its effectiveness is limited. hp_2 prioritizes hunks based on text similarity, which is designed to explore and prioritize all hunks effectively. However, using it to prioritize only the hunks covered by the test method's call graph contradicts its primary purpose, leading us to exclude combinations involving these two values. As a result, eight combinations remained to investigate.

Of these remaining eight, to reduce the enormous cost of our experiments, four were eliminated based on preliminary experimental results, which were done on a smaller dataset. For instance, $IO_2(rc_2, hp_2, hr_1, o_1)$ showed improved performance over $IO_1(rc_1, hp_1, hr_1, o_1)$, leading to the exclusion of the remaining formats that utilized (rc_1, hp_1). Similarly, $IO_4(rc_2, hp_2, hr_2, o_2)$ showed decreased performance compared to $IO_3(rc_2, hp_2, hr_2, o_1)$, leading us to discard other formats utilizing o_2. Consequently, our experiments ended up covering four parameter value combinations.

3.3.2.4 Repair Generation

We utilize CLMs with both encoder-decoder and decoder-only transformer architectures [107] that are pre-trained on code (e.g., CodeT5+ [105]). In both architectures, the decoder generates output tokens in an auto-regressive manner. To teach test repair patterns and repair generation to the model, we fine-tune CLMs using a test repair training set. We formulate the fine-tuning task as a neural machine translation task where the broken test case along with the required context are translated into the repaired test case. During fine-tuning, the model is presented with the inputs (broken test code and repair context) and their corresponding outputs (repaired test code). The model learns to minimize the difference between its predicted repairs and the ground truth repairs in the training data. Subsequently, we employ the fine-tuned model to generate repairs for broken test cases. Language models have a predefined vocabulary comprising various tokens. Therefore, the input source code goes through tokenization before being provided as input to the model, and the resulting outputs are also in token format.

As described in Section 3.2, we use the beam search ranking strategy to generate candidate test case repairs. Finally, we replace the breakage lines of the original test code with the candidate repair patches. We compile and run the repaired test cases against the new version of the SUT (V_{i+1}). By following this process, we identify the plausible test repairs, i.e., candidate test repairs that compile and pass. These plausible test repairs are then recommended as the outcome of TARGET.

3.4 Study Design and Results

In this section, we present the experimental study to evaluate our automatic test case repair technique leveraging language models (TARGET) and answer the following research questions.

RQ1.1 How do different combinations of pre-trained code language models (CLMs) and input formats perform in repairing test cases using TARGET?

As outlined in Section 3.3, our methodology employs multiple input-output formats (*IOs*) to adapt CLMs for test case repair. RQ1.1 seeks to evaluate the effectiveness of different *IOs* across three selected CLMs, with the goal of identifying the conditions under which our approach yields the best results. This evaluation is critical to understanding the adaptability and effectiveness of our approach in various contexts. Furthermore, the best configuration identified in RQ1.1 is subsequently applied in the remaining RQs to investigate and discuss other aspects of the approach.

RQ1.2 How does the best configuration of TARGET from RQ1.1 perform against baselines?

RQ1.2 investigates whether our proposed approach, in its best configuration, offers any advantages over one relevant existing study and two simple baselines. The goal of this comparison is to thoroughly evaluate the effectiveness and practicality of our approach in relation to baseline methods, complementing its assessment in RQ1.1.

RQ2.1 How effective is TARGET at repairing test cases based on the test repair characteristics found in our benchmark (TARBENCH)?

Test breakages can arise from a variety of causes. RQ2.1 seeks to examine whether TARGET's effectiveness varies depending on the specific type of test breakage being repaired. Understanding this is crucial for the practical application of our approach, as it allows us to pinpoint the scenarios where it performs optimally and highlights challenging areas to direct future work in addressing diverse test failure scenarios.

RQ2.2 Can we accurately predict, at the time of repair, the situations in which TARGET cannot effectively repair a test in practice?

Although a perfect solution for test repair may be achievable in the future, we have not currently achieved it yet. Consequently, being able to predict when TARGET will deliver highly accurate repairs and when it might not is essential for its effective use. This predictive capability would help practitioners avoid unnecessary effort by suggesting them

to avoid low-quality repairs. RQ2.2 seeks to explore whether, for a given broken test case, we can predict the model's ability to generate correct repairs without executing the model itself.

RQ3.1 What is the impact of the amount of data available for fine-tuning on test repair performance?

CLMs require extensive datasets for effective pre-training. Although fine-tuning these models requires significantly less data, the quality and relevance of the data become critical, making the process of collecting a suitable fine-tuning dataset both resource-intensive and costly. Understanding the impact of the size of the fine-tuning dataset on model accuracy is important for performing a cost-benefit analysis. RQ3.1 specifically investigates how varying the size of the fine-tuning dataset impacts model performance.

RQ3.2 How does the performance of the fine-tuned model generalize to unseen projects?

RQ3.2 explores whether TARGET can effectively generalize to repair tests from projects whose data was not utilized during fine-tuning. Addressing this question will show whether TARGET can be applied to new projects, thereby reducing or potentially eliminating the need to gather project-specific historical data for fine-tuning.

In the following, we describe the evaluation metrics we employ and introduce TARBENCH. We then elaborate on the experimental process, present the results, and finally discuss their practical implications.

3.4.1 Evaluation Metrics

Based on existing studies on code generation tasks [113–115], we adopt four evaluation metrics to assess the performance of test case repair generation. Specifically, we rely on: exact match accuracy (EM), BLEU [116], CodeBLEU [117], and plausible repair accuracy (PR). These metrics evaluate the quality of generated repair candidates by comparing them with the ground truth and assessing their execution results. In the context of test case repair, we define each metric based on the assumption that our evaluation set (E) contains the broken test cases, for each $t \in E$, the ground truth repair equals to $gt(t)$, and for each $t \in E$, our model generates k repair candidates, i.e., $C_t = \{c_1, c_2, \ldots, c_k\}$.

Exact Match Accuracy (EM). The EM metric [101], also known as perfect accuracy, refers to the percentage of test cases that include at least one repair candidate with source code exactly matching the ground truth. The EM metric is defined as follows:

$$\text{EM} = \frac{1}{|E|} \sum_{t \in E} \begin{cases} 1, & \text{if } gt(t) \in C_t \\ 0, & \text{otherwise} \end{cases}$$

While the EM metric is very strict, it fails to consider repair candidates that have similar semantics but differ in syntax. Therefore, we also adopted other metrics to provide a comprehensive evaluation.

BLEU and CodeBLEU. The BLEU metric [116] was originally designed for evaluating machine translation tasks. It measures the extent of overlapping n-grams between

a candidate translation and a set of reference translations. However, BLEU falls short in evaluating code-related tasks as it fails to account for the tree structure and logic inherent in code. To address this limitation, Ren et al. introduced CodeBLEU [117], which extends BLEU by incorporating code-specific information into its measurements. Both BLEU and CodeBLEU are computed based on a single repair candidate per test case. Thus, for each $t \in E$, we choose the optimal repair candidate from the available k candidates. The best repair candidate is determined either by an exact match with the ground truth or, if no exact match exists, by selecting the candidate with the highest score in the beam search ranking. It is worth noting that we use the BLEU-4 variation, which is widely adopted and is the geometric mean of all n-gram precision values up until 4-gram [116]. Throughout the remainder of this chapter, any references to BLEU will be referring to this variant.

Plausible Repair Accuracy (PR). The PR metric [102] evaluates the effectiveness of test case repairs by evaluating the percentage of test cases that contain at least one repair candidate that compiles successfully and passes on the updated SUT version. In other words, a repair is considered successful if it avoids both compile-time and runtime errors, and the test execution logs confirm a pass. This metric offers a practical assessment of the syntactic and, to some extent, semantic correctness of the repair by validating it within the real execution environment. The PR metric is defined as follows:

$$\text{PR} = \frac{1}{|E|} \sum_{t \in E} \begin{cases} 1, & \exists c \in C_t : c \text{ compiles and passes} \\ 0, & \text{otherwise} \end{cases}$$

3.4.2 Benchmark: TaRBench

Current research on test repair lacks a public, sufficiently large and diverse benchmark for evaluating and comparing their methods. In short, existing test repair benchmarks suffer from their small size, limited scope regarding repair categories, and their lack of reproducibility and reusability. We discuss these limitations in great detail in Section 3.5. To enable high-quality research on test repair, we have undertaken the major initiative to construct a large, high-quality test repair benchmark, which we name TaRBench [11].

To create the dataset used in this study, we followed a multi-step process, which involved the selection of subjects, test repair collection, data preprocessing, and data splitting. An overview of the dataset creation steps is shown in Figure 3.7. We provide the details for each step in this section.

3.4.2.1 Subject Selection

To initiate the dataset creation process for this study, we began by following these steps to select the study's subjects.

Selecting Projects. We began by using the Java projects from CodeSearchNet [118], which provided us with 4,662 Java projects. CodeSearchNet's projects, while used for code retrieval tasks, were selected from a comprehensive search of open-source projects to meet quality criteria, thus forming a good initial set for our purpose.

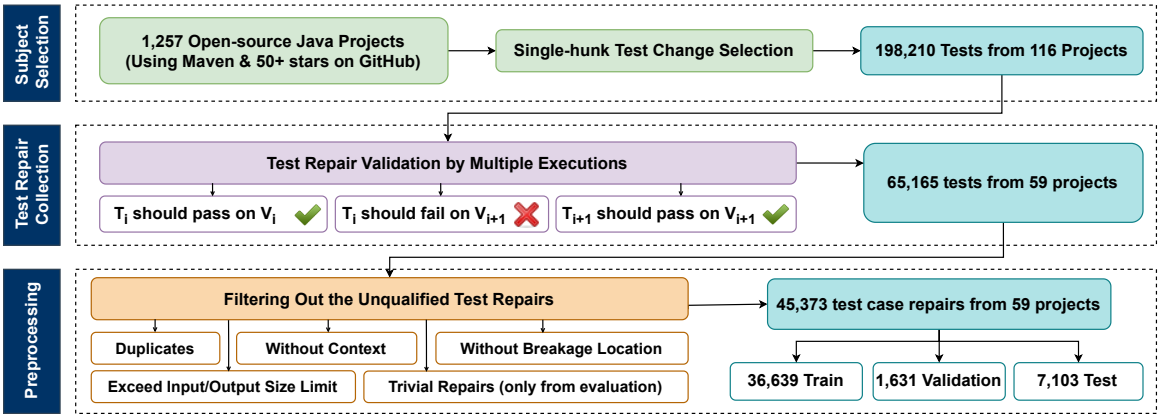

Figure 3.7: Overview of the creation steps of our test case repair benchmark (TARBENCH).

Next, we fetched the projects from GitHub and selected the ones using Maven with a minimum of 50 stars. GitHub hosts a vast number of repositories, and many of them may not be actively maintained or may not have a clear purpose. Therefore, filtering by stars helps to reduce this kind of noise. This step led to a selection of 1,257 projects.

Selecting Test Cases. We analyzed each commit, C, to identify code changes within unit test methods across all projects. Recall from Section 3.3 that we defined V_i and V_{i+1} to represent two consecutive versions of the SUT, where T_i is a broken test case in V_i, and T_{i+1} is its repaired version in V_{i+1}. Assuming V_{i+1} corresponds to the code at commit C and V_i to C's parent, we examined the code changes in C. We then selected all available (T_i, T_{i+1}) pairs in C as potential test case repairs, and automatically matched T_i with T_{i+1} using their fully qualified names, as outlined in Section 3.3.1.1. We selected test method changes with only one hunk (single-hunk method changes), where the code change is confined to a single chunk of code. This step led to the selection of 116 projects with 198,210 test method changes.

Our motivation for selecting single-hunk test method changes (potential test repairs) is to facilitate the learning process of CLMs. Single-hunk changes represent isolated modifications that can be easily associated with specific intentions, providing clarity and focus for learning. This approach is also a common practice in most studies [110, 119–122], especially in the program repair domain. Note that single-hunk test repairs are not necessarily simple. For example, a multi-hunk test repair might involve a simple renaming of a method repeated multiple times across the test method. In contrast, a single-hunk test repair might involve adding a zero at the end of a literal integer, which requires a deep understanding of the SUT. Therefore, the number of hunks in a test repair does not necessarily correlate with its complexity.

While our focus is on single-hunk test repairs, our approach can be adapted to handle multiple hunks by either iteratively addressing each one or repairing them in one step by marking multiple breakage locations in the input. However, extending support to multi-hunk repairs requires additional effort to modify our single-hunk technique and gather relevant multi-hunk data. We believe that expanding to multi-hunk repairs is a valuable

direction for future research.

Note that when collecting test repairs from projects, we focus on the default branch rather than examining every branch. We begin with the most recent commit on the default branch and work backwards through its commit history, inspecting all reachable commits. This approach is based on the observation that most projects maintain a primary branch-often named *"main"* or *"master"*—into which most changes are merged after successful code reviews and CI/CD checks. Thus, changes in the default branch have typically undergone thorough review processes and automated testing, ensuring their reliability.

3.4.2.2 Test Case Repair Collection

Not all test method changes are valid test repairs since they can be test refactorings or unsuccessful repair attempts. Following the selection of projects, we proceeded to analyze the commits associated with the test method changes to identify valid test repairs. To determine whether a test method change is a test repair and to ensure the quality of our data, we conducted three test executions [3].

To determine the result of each test execution, we analyzed the Maven logs and categorized executions as either valid or invalid. Valid executions are those where the test passes or fails due to issues within the test code itself. Invalid executions result from factors unrelated to the test code. We identified valid executions based on the three Maven log patterns outlined in Table 3.3, which cover successful executions and failures due to compilation or runtime errors within the test code. Any execution log that does not match these patterns is considered an invalid execution and is discarded. Examples of such invalid scenarios include dependency resolution failures and compilation or runtime errors external to the test code.

We conducted the following three types of test executions to identify test repairs among the test method changes.

The Test Case Initially Passes. We executed T_i and verified its successful execution against V_i. In the event of T_i failing, we excluded the test, because if T_i is already failing before any change to the SUT, even if the commit repairs the test, we are unable to extract the necessary repair context for our approach to function effectively. This step resulted in the exclusion of 91,059 tests.

We conducted a detailed analysis of the test execution results during this initial phase. The failing executions were categorized into four groups: (1) compile or runtime errors from the SUT, accounting for 23% of cases; (2) compile or runtime errors from the test cases, representing 16%; (3) dependency errors related to libraries or plugins, despite using Maven and the correct Java version, which made up 33%; and (4) unknown failures, where the cause could not be automatically identified from the execution logs, contributing 28%. Notably, only 16% of the failures were directly linked to issues with the test cases. The

[3]To execute the test cases, we use Maven version 3.6.3 and Java Development Kit (JDK) versions 1.8.0_192, 11.0.16_8, or 17.0.2, depending on the compiler version specified in the project's pom.xml file.

Valid Text Execution Scenarios

1. Successful test case execution
Pattern:
`"Tests run:  1, Failures:  0, Errors:  0, Skipped:  0"`
Example:
`[INFO] Running TestClass.testMethod`
`[INFO] Tests run:  1, Failures:  0, Errors:  0, Skipped:  0`
`[INFO] BUILD SUCCESS`

2. Compilation error within the test case
Pattern:
`"COMPILATION ERROR"` **or** `"Compilation failure"`
And `"[ERROR] /.+/<test_rel_path>:[(\d+),\d+].*"`
Example:
`[ERROR] COMPILATION ERROR :`
`[ERROR] /path/to/TestClass.java:[25,13] cannot find symbol`

3. Runtime failure within the test case
Pattern:
`"[ERROR] <test_class>.<test_method>:(\d+)"` **or**
`"at .+<test_class>.<test_method>(<test_class>.java:(\d+))"`
Example:
`[ERROR] TestClass.testMethod:15 NullPointerException`
`[ERROR] at TestClass.testMethod(TestClass.java:15)`

Table 3.3: Valid test execution scenarios identified by Maven log patterns, with examples showing successful execution, compilation errors, and runtime failures in test case code.

majority were caused by environment-related problems or faults in the SUT, potentially due to ongoing development.

SUT Changes Break the Original Test Case. We subsequently executed T_i against V_{i+1} and assessed whether it failed specifically due to problems in the test code. In cases where the test did not fail, or its execution was found to be invalid as previously described, we excluded the test. This decision was based on our aim to identify valid broken test cases. This step resulted in the exclusion of 33,977 tests.

The Test Case Change Repairs the Broken Version. We executed T_{i+1} against V_{i+1} and verified whether T_{i+1} passed, ensuring that the commit repaired the test. In cases where T_{i+1} did not pass, indicating that the changes did not repair the test, we excluded the test. This step resulted in the exclusion of 1,592 tests.

After applying the above-mentioned analysis based on the test method changes, we ended up with 71,582 test case repairs across 112 projects. We found that, in certain projects, significant test repair activity was undetected by our approach. This determination was made through a two-step process:

Test Case Execution Analysis. We examined the test cases to identify those that did not meet the first two conditions of the test executions (T_i passing on V_i and T_i failing on V_{i+1}). When a substantial percentage of potential test repairs (test method changes) in a project did not meet these conditions, it suggested limitations such as the need for a specialized testing environment or unresolved dependencies by Maven. Projects where over 50% of test method changes did not meet these conditions were skipped. This criterion led

to the exclusion of 55 projects.

Test Repair and Commit Count. From the 55 skipped projects, we reviewed the number of test repairs and commits. We selected two projects for inclusion in our dataset due to their significantly higher numbers of valid test repairs and commits, averaging 2,500 and 450, respectively, compared to averages of 120 and 25 in the other 53 skipped projects.

To summarize, this process ensured that only projects with detectable and significant test repair activities were included in our dataset. Due to potential instability in testing or the need for special testing environments, projects with lower detectable test activities are expected to lead to many experimental problems. Indeed, using these projects would contribute a limited number of test cases while requiring the execution of test cases in a specialized environment that we may not have access to. In this step, 53 projects, which had a total of 6,417 test repairs, were skipped, while 65,165 test repairs across 59 projects were retained.

To complete our benchmark, we also needed to collect the changes in the SUT for creating the repair context. Therefore, for each test case repair commit, we collected all the changes in the SUT source files, i.e., files that do not include "*src/test*" in their path and do not include the JUnit test annotation (*@Test*). In collecting SUT changes, we considered code changes within the existing source files of the SUT. We excluded changes that involved the addition or removal of an entire file. The rationale behind this exclusion is that complete file additions or removals typically result in lengthy code sequences, with most of the code in these files being irrelevant for repairing test cases. Enriching the repair context with relevant code from complete file additions or removals is a task that can be explored in future work.

3.4.2.3 Preprocessing and Splitting

In the next step, we conducted preprocessing to eliminate redundant repairs, large tests which do not fit the model's input, and repairs for which we could not obtain complete information, as detailed below. The following preprocessing steps were applied to the dataset derived from the previous section, which included 65,165 test repairs across 59 projects.

Duplicate Test Repairs. Identical test repairs may exist in different commits or branches. Since we analyze every commit as detailed in Section 3.4.2.1, we may encounter duplicate test repairs, as a test repair may appear both in a merge commit and in the original commit from its source branch. This step resulted in exclusion of 1,039 tests.

Empty Repair Context. We were unable to collect the repair context for a test repair when the Spoon library, a static code analysis tool [112] we utilized, failed to process the project's code. We excluded such cases as our approach relies on having a repair context to provide to our model. This step resulted in the exclusion of 3,748.

Absence of Test Breakage Location. In accordance with our problem definition outlined in Section 3.3, we assume that the test breakage location is included in the input. During our data analysis, we found that, in certain cases, the test breakage location could

not be identified. Specifically, in 1,248 tests, the test repair consisted solely of adding new lines. In other words, the repair introduced new lines of code to address the problem without changing any of the existing lines from the test that was initially broken. As a result, the breakage location remains undetermined in these cases. Furthermore, we identified 19 test cases in which the breakage location could not be determined using the Spoon library and, in 7 other test cases, the broken lines were actually comments. As a result of these findings, we excluded a total of 1,274 tests in this preprocessing step.

Maximum Input and Output Length. We set the maximum input length to 512 tokens and the maximum output length to 256 tokens. While the language models we use can handle larger inputs and outputs, we were constrained by available computing resources (Section 3.4.9). Also, we used the 512 limit to be consistent across models as the maximum token limit varies across models. We excluded test cases that exceeded the limit, considering both the test code and at least one code change hunk in the SUT. It is worth noting that with the expected advancements in language models and hardware, this practical limitation and constraint is going to become less significant in the future. This step resulted in excluding 5,383 test cases.

3.4.2.4 Quality Checking

In addition to analyzing test executions (as detailed in Section 3.4.2.2), to significantly enhance data quality by only including passing test cases or those that fail due to problems in the test code, we implemented two additional criteria to further improve data quality and remove low quality repairs and noise.

Test Refactoring and Relocation. We excluded cases where the repair only involved the relocation or refactoring of test code, with no changes to the SUT. Examples include renaming test utility methods or classes and upgrading testing library versions, such as JUnit. These cases occur when all changes in the commit are confined to test source code, specifically files containing "*src/test*" in their path. A manual review of randomly-selected cases confirmed that such test repairs were consistently unrelated to the SUT. Consequently, they were classified as refactoring activities and treated as noise in our dataset. This step resulted in the exclusion of a total of 6,302 tests.

While our heuristic may overlook some valid test repairs—such as cases where production code changes occur in one commit and corresponding test repairs happen in subsequent commits—it achieves a balance between minimizing noise and maximizing repair instance count. Given the relatively small proportion of test-change-only instances (6k out of 65k) and the complexity associated with more nuanced heuristics, this approach offers a practical trade-off for this study.

Trivial Test Repairs. Trivial repairs refer to repairs that can be accomplished with development tools, such as code editors or IDEs. We define trivial repairs as repairs that only include the renaming of class or method names. We designed a simple technique to automatically detect trivial repairs. First, we detect renamed classes and methods in the SUT using RefactoringMiner [123]. Second, we identify the classes and methods used in

the broken test case using Spoon [112]. Finally, if there is at least one common class or method name between the first and second steps, we identify the test repair as trivial.

Trivial repairs fail to capture the challenges of the test case repair problem. Thus, we removed them from our evaluation set to obtain more realistic results based on substantial and complex test case repairs and prevent the risk of artificially inflating our performance metrics due to trivial repairs.

Though excluded for evaluation purposes, during our preliminary experiments, we observed that including trivial repairs in the training set enhanced our model's performance. We provide additional details to support the effectiveness of keeping trivial repairs in the training set in the results section. Trivial repairs, although simple, provide valuable information about basic patterns and structures in the test and production code that the model needs to learn. By including these repairs, the model gains a foundational understanding of syntax and semantics, which can enhance its capability to handle more complex repairs.

We identified 9,242 trivial test repairs in our dataset, removed 2,046 of them from the evaluation set, and kept 7,196 in the training set. We will provide further details on the training and evaluation sets in the following section.

3.4.2.5 Data Splitting

Following the above steps, we divided the data into the three splits: the training set (80%), the validation set (5%), and the test set (15%). To ensure a fair and realistic evaluation, it is crucial that the more recent test case repairs are not exposed to the model during training. This is important because newer code may contain the ground truth of older test repairs. Thus, for each project, we kept older commits in the training set and included newer commits in both the validation and test sets. We use the validation set to determine when to stop training, preventing the model from overfitting.

Following the completion of all steps, TARBENCH includes 45,373 test repairs across 59 distinct projects, making it notably larger and more diverse than the largest existing test repair dataset in the literature, which only consists of 235 test repairs across 14 projects (see Section 3.5). In this dataset, we have **36,639** training instances, **1,631** validation instances, and **7,103** test instances.

3.4.2.6 Analysis of Breakage and Error Lines

While tools such as IDEs and CI systems can capture error locations in test code, identifying the actual breakage lines that lead to those errors is not always straightforward. We analyzed the test repairs in our benchmark to clarify the distinction between breakage lines and error lines, which are defined in Section 3.3. These two sets of lines do not always overlap, as repairs may occur in locations different from where the errors originated. The results of this analysis are presented in Table 3.4. The dataset consists of 82.1% compilation failures and 17.9% runtime failures, reflecting two primary types of test failures.

Case	Compile (%)	Runtime (%)	Total (%)
No Intersection	37.0 (13786)	60.3 (4899)	41.2 (18685)
Exact Match	55.3 (20608)	26.0 (2110)	50.1 (22718)
Some Intersection	7.7 (2851)	13.7 (1119)	8.7 (3970)

Table 3.4: Breakdown of the intersections between the breakage lines and error lines. The values in brackets indicate the count of instances for each cell.

Regarding the relationship between the error lines and the breakage lines, in 50.1% of the failures, the breakage lines exactly match the error lines. In the remaining cases, however, they do not match perfectly: 41.2% of the failures show no intersection between breakage and error lines, while 8.7% have a partial intersection without a complete match. This indicates that in nearly half of the dataset, identifying breakage lines is not a straightforward task, specifically for IDEs or developers, as error and breakage lines do not fully match. Furthermore, we found that the breakage lines and error lines match exactly in 26% of the runtime error instances and 55.3% of the compilation error instances. This demonstrates that identifying breakage lines associated with runtime errors is more challenging than with compilation errors, though it is not always straightforward for compilation errors either.

The analysis highlights that identifying breakage lines is inherently complex and cannot be solely determined by error lines. This suggests that test repair tools must employ more sophisticated techniques, beyond relying on error line information alone, to accurately pinpoint breakage locations. This is particularly crucial for addressing runtime failures, where the challenge is more noticeable.

3.4.3 Test Repair Performance of CLMs and Input Formats (RQ1.1)

In recent years, CLMs (language models that are pre-trained on code) have emerged that can be fine-tuned specifically for test code repair purposes. While some studies have compared CLMs in different program generation tasks [113–115, 124], to the best of our knowledge, this study is one of the first to examine their performance in the test repair context. Therefore, answering this research question will offer valuable practical insights and contribute to our understanding of test case repair using CLMs.

Additionally, we have explored multiple approaches to format input and output data for CLMs. This research question also seeks to evaluate how different input formats influence model performance in comparison to each other. Overall, this research question investigates the combinations of models and input formats that produce the best results in the context of test repair, which is essential for the effective utilization of CLMs.

3.4.3.1 Selection of Models

In general, CLMs can be categorized into three groups based on their architectures: encoder-only, decoder-only, and encoder-decoder. A systematic review conducted by Hou et al. [101]

explored the application of CLMs in software engineering tasks and identified more than 50 models that can be applied in various software engineering tasks. However, no study has applied these models specifically to test repair or has demonstrated that a single CLM outperforms others in software development tasks, particularly in program repair. Thus, since evaluating all available CLMs is computationally impossible, we have established the following filtering criteria to select CLMs used to answer RQ1.1.

Direct Applicability. Similar to the approach taken by Jiang et al. [124], our focus is exclusively on models that can be seamlessly applied to repair source code without requiring additional modifications or architectural changes. Consequently, we exclude all models with an encoder-only architecture, as they necessitate the addition of an extra decoder component to generate repairs. Two notable examples of such models include CodeBERT [125] and GraphCodeBERT [126].

Open Source Models. We exclude all models that are not publicly available, as fine-tuning them with test code repair data is essential for our study. Examples of such models are Codex [127], AlphaCode [128], PaLM-Coder [129], and GPT-4 [130].

Pre-training Data. As our primary focus is on code generation, it is crucial that the models we employ have been pre-trained on large code datasets. Consequently, we do not consider models that have been pre-trained on general natural language datasets. For example, we exclude T5 [131], GPT-Neo [132], and GPT-J [133] based on this criterion.

Old Models. We excluded models that have been superseded by newer versions. For instance, CodeT5 [115] has been excluded because it is superseded by CodeT5+ [105].

Model Size. To enhance practicality and broaden the applicability of fine-tuning, we excluded models exceeding one billion (1B) parameters in size. Utilizing larger models for fine-tuning often demands substantial computational resources, which may not be readily available in many development contexts. For instance, consider StarCoder [134], a 15.5B decoder-only CLM, which underwent pre-training on 512 *A100 80 GB GPUs* across 64 nodes. We fully realize that larger models will become easier to train over time and that therefore our repair performance results probably represent a lower bound in terms of what to expect in the future.

Model Performance. While no study has identified top-performing models for the specific test repair context, we nevertheless reviewed related studies and excluded models that exhibited relatively lower performance compared to others. Our prioritization was based on the widely-recognized HumanEval [127] benchmark, which is designed for assessing code generation capabilities. This benchmark closely aligns with the challenges posed by the test case repair task. For example, we excluded PolyCoder [135] due to its underperformance in HumanEval when compared to CodeGen [106].

By applying the exclusion criteria outlined above, we narrowed down our selection down to two CLMs: **CodeT5+ 770M [105]** (CT5+), an encoder-decoder model with 770 million parameters, and **CodeGen Multi 350M [106]** (CG), a decoder-only model with 350 million parameters. Furthermore, **PLBART Base 140M [104]** (PLB), a relatively small encoder-decoder model that was not evaluated on the HumanEval benchmark, was selected as well to assess how a smaller model impacts test repair performance.

Input-output Format (*IO*)		EM			PR			CodeBLEU			BLEU		
		CT5+	PLB	CG	CT5+	PLB	CG	CT5+	PLB	CG	CT5+	PLB	CG
IO_1	Base	61.0	54.0	51.8	77.2	78.0	73.8	76.7	76.4	72.4	77.7	77.8	73.1
IO_2	Text-based Similarity Hunk Ordering	**66.1**	57.9	57.7	80.0	79.2	77.0	**79.3**	77.9	74.7	**80.0**	79.2	75.7
IO_3	Word Level Hunk Representation	63.2	59.1	58.1	**80.2**	79.2	77.0	77.2	78.4	75.6	78.3	79.6	76.4
IO_4	Edit Sequence Output	52.9	47.6	45.7	66.4	64.4	56.9	66.8	61.2	56.9	67.5	61.8	57.0

Table 3.5: Comparison of test case repair performance among the CodeT5+ 770M (CT5+), PLBART Base 140M (PLB), and CodeGen Multi 350M (CG) code language models, and across four different input-output formats.

The beam size, which determines the number of repair candidates generated, is set to 200 for PLB and 40 for both CT5+ and CG. Our goal was to select the largest beam size feasible to maximize the exploration of the potential repair candidates' search space, thereby improving performance. The selected beam sizes represent the maximum we could utilize given our computational resources.

3.4.3.2 Input and Output Formatting

Based on our problem definition in Section 3.3, we perform test repair given the broken test code, the location of the breakage within the test code, and a set of code changes made to the SUT. With these inputs, we can construct an input sequence for a CLM in various ways. Additionally, the expected output that the model is trained to generate can have different formats. We defined four distinct input and output formats (*IO*s) in Section 3.3 and, in this research question, we explore the impact of these *IO*s on the performance of test case repair.

3.4.3.3 RQ1.1 Results

Table 3.5 presents the results of the three selected CLMs across various *IO*s, measured using the four evaluation metrics. The CT5+ model achieves the highest EM, when utilizing Text-based Similarity Hunk Ordering (IO_2). Furthermore, in the majority of cases, CT5+ outperforms other models across different metrics and IOs. This superiority can be justified due to the larger number of parameters in CT5+.

Interestingly, we observed that PLB, despite having only 140M parameters, outperforms CG with 350M parameters. Furthermore, its performance (with IO_2) closely aligns with that of CT5+, with losses of 8.2, 0.8, 1.4, and 0.8 percentage points (pp) in terms of EM, PR, CodeBLEU, and BLEU, respectively. This observation suggests that the number of parameters is not the sole determinant of model performance. It implies that a high-performing CLM can be trained at a significantly lower cost without significantly compromising performance. Considering the substantial size difference between PLB and CT5+, sacrificing a few percentage points of performance for substantial cost reduction during training and inference seems reasonable. Thus, we recommend PLB over CG in the test repair context, and the choice between PLB and CT5+ is a trade-off between training cost and a marginal performance loss.

```
[<TESTCONTEXT>]
@Test
public void testValidateColumns() {
   Schema schema = SchemaBuilders.schema().mapper("field1", stringMapper()).build();
   [<BREAKAGE>]
   Columns columns = new
       Columns().add(Column.builder("field1").buildWithComposed("value",
       UTF8Type.instance));
   [</BREAKAGE>]
   schema.validate(columns);
   schema.close();
}
[<REPAIRCONTEXT>]
[<HUNK>]
[<DEL>]
  columns.add(Column.builder(name).buildWithDecomposed(value, valueType));
[<ADD>]
  columns.addDecomposed(name, value, valueType);[</HUNK>]
[<HUNK>]
[<DEL>]
  columns.add(builder.withUDTName(itemName).buildWithNull(itemType));
[<ADD>]
  adder.withUDTName(itemName).addNull(itemType);[</HUNK>]
[<HUNK>]
[<DEL>]
  columns.add(builder.buildWithNull(valueType));
[<ADD>]
  adder.addNull(valueType);[</HUNK>]
```

(a) Input Format

```
Columns columns = new Columns().addComposed("field1", "value", UTF8Type.instance);
```

(b) Output Format

Figure 3.8: An example from TARBENCH showcasing the best-performing input-output format, IO_2. TARBENCH ID: *stratio/cassandra-lucene-index:485*.

Moreover, concerning IOs, as depicted in Table 3.5, IO_2 and IO_3 yield the best results across all models and metrics. Specifically, IO_2 provides the best results for CT5+, while IO_3 yields better results for PLB and CG. Overall, based on the results, IO_3 exhibits the highest performance in 7 out of 12 cases, outperforming IO_2, which achieves the highest performance in 3 out of 12 cases, with both being equal in the remaining 2 cases. As further discussed below, neither Base (IO_1) nor Edit Sequence Output (IO_4) achieves favorable results in any case, with IO_4 reaching the lowest performance in all instances. We provide an example of IO_2, the best-performing IO, in Figure 3.8.

By definition, the difference between IO_2 and IO_3 lies in their hunk representation: IO_2 uses line-level context, while IO_3 uses word-level context. The results indicate that the larger model, CT5+, performs better with line-level context, whereas the two smaller models, PLB and CG, demonstrate improved performance with word-level context. This suggests that larger models may be better at leveraging the additional information provided by line-level context, while smaller models might benefit more from the focused and concise nature of word-level information. However, it is not possible to draw a definitive conclusion

based solely on this data, as the models vary in more than just size; they also differ in pre-training datasets and tasks. Further investigation is required to explore the relationship between model size and performance across different *IOs*. For instance, comparing CT5+ with 770 million parameters to versions with 220 million or 2 billion parameters could provide additional insights.

We have explored a number of *IO* formats and the fact that some of them perform worse should be reported to inform researchers and practitioners. Further, while we recommend against the use of IO_4 and IO_1, they remain viable options. For instance, IO_4 remains viable as the differences in EM and PR between IO_4 and IO_2 (the best-performing *IO*) are only 10.3% and 13.8%, respectively. The choice between the other two techniques, IO_2 and IO_3, is context-dependent and may require prior analysis and experimentation. For instance, IO_2 is recommended for CT5+, whereas IO_3 is preferred for PLB.

We acknowledge that more advanced prompt engineering techniques, such as Chain-of-Thought prompting, Few-shot learning, or iterative refinement, as well as the larger context windows available in commercial LLMs, could further enhance the test case repair task.

To assess the impact of trivial repairs on model performance, as discussed in Section 3.4.2.4, we conducted an additional experiment using our best-performing strategy: the CT5+ model with IO_2. In this experiment, we removed 7,196 trivial repairs from the training set and repeated the fine-tuning and evaluation processes. The results indicated a performance decline of 5.6 percentage points in EM and 2.8 percentage points in CodeBLEU. These findings highlight the positive contribution of trivial repairs to the overall model performance, reinforcing the value of including them in the training set.

RQ1.1 Summary

The CT5+ model with IO_2 delivers the best performance for the test repair task, with 66.1% exact match accuracy and 80.0% plausible repair accuracy, making it our top recommendation. For resource-limited environments, the PLB model with IO_3 offers a good balance between performance and efficiency, with faster fine-tuning and inference speeds.

3.4.4 Test Repair Performance Against Baselines (RQ1.2)

In this research question, we aim to compare TaRGET against a recent state-of-the-art method and two simple baselines. Specifically, we compare the best configuration of our approach (CT5+ with IO_2) with Ceprot [136], which addresses the automatic detection and updating of obsolete tests and focuses solely on the SUT method that the test code targets. Further, we also compare with two baselines that implement basic solutions: SUTCopy and NoContext. This comparison aims to provide a comprehensive evaluation of the effectiveness and practicality of our approach against both advanced and basic baselines. In the following subsections, we detail the comparison between TaRGET and Ceprot, define the SUTCopy and NoContext baselines, and report the results of TaRGET's

performance against these three baselines. We also provide a technical comparison between TARGET and CEPROT in Section 3.5.1.

3.4.4.1 Collecting Ceprot's Data

CEPROT is an approach that addresses both the automatic detection and updating of obsolete tests. This method fine-tunes the CodeT5 language model to handle both tasks efficiently. The authors have also constructed a dataset for the update task, containing 5,196 instances, with 520 instances selected as the evaluation (test) set. In their study, the authors define an obsolete test case as one that is modified within a commit, given that the production method from the SUT, which is covered by the test, also undergoes changes. Although this definition is similar to our benchmark's definition, TARBENCH encompasses a broader spectrum of scenarios and does not require a change in the production method. TARBENCH defines broken test cases based on the test execution results before and after changes in the test code, as detailed in Section 3.4.2.

CEPROT achieves a 12.3% EM and a 63.1% CodeBLEU score in the test update generation task. To assess the generalizability of TARGET and ensure a fair comparison with CEPROT, we applied TARGET to CEPROT's evaluation set. TARGET utilizes data from all code changes in a commit, and therefore, the raw data provided by CEPROT was insufficient for a direct application of TARGET. However, CEPROT provided the GitHub repository name and the commit hash for each instance. Using this information, we collected all necessary data to enable TARGET to function correctly. We could not apply CEPROT to our benchmark as CEPROT did not provide the code needed to collect and construct the required data for their approach.

After our data collection, we determined that TARGET could only be applied to a subset of 214 instances (41%) from CEPROT's evaluation set, which we refer to as compatible instances. This decision was based on several key factors described below.

Invalid Cases. We found 30 instances to be invalid or impossible to collect or repair: 10 instances were excluded because the specified commit was not available on GitHub; 2 instances were excluded as the repository did not exist on GitHub; 12 instances were excluded because the source code of the test and the target code were identical, indicating no changes in the test; 2 instances were excluded as the identified test case was actually production code; 2 instances were excluded because the commits involved only test code changes without production code changes, likely indicating test code refactoring rather than repair; and 2 instances were excluded due to our data collection tool's inability to detect repairs for unknown reasons. These invalid cases were removed from our comparison.

Test Method Name Changes. During TARBENCH's data collection, we matched test methods between versions based on their full class and method names. Consequently, we did not identify test method name change scenarios as repair instances. However, 38 instances in CEPROT's evaluation set involved repairs that included changes to the test method name. CEPROT did not clarify how these methods were matched initially to ensure they pointed to the same test case. Therefore, we did not include these cases in our comparison.

No Source Changes. In 24 instances of CEPROT's evaluation set, the repairs involved only added lines without changing any existing lines. As mentioned in Section 3.4.2.3, TARBENCH excludes these cases due to the absence of a test breakage location. TARGET requires the test breakage location to function, and without source changes, there is no breakage location. For this reason, we excluded these cases from our comparison.

Multi-hunk Repairs. We found that 214 test instances involved multi-hunk test repairs, i.e., repairs with code changes in multiple chunks of the test method. As described in Section 3.4.2.1, our work focuses on single-hunk test repairs, which is a common practice in many program repair studies and provides clarity and focus in the model's learning process. To evaluate CEPROT's performance on multi-hunk instances, we replicated CEPROT and computed its performance on these cases, finding an EM of 7.9% and a CodeBLEU score of 42.6%. This indicates poor performance by CEPROT on multi-hunk instances, showing ineffective support for these cases, which clearly should be addressed by future research through iterative approaches. Consequently, we excluded multi-hunk instances from our comparison.

3.4.4.2 SUTCopy and NoContext

In SUTCOPY, we examine the SUT changes in an arbitrary order and update the test code when we encounter an SUT changed part that is uniquely identifiable in the broken part of the test code. For example, if the string "*value1*" is changed to "*value2*" in the SUT, and "*value1*" exists in the broken test code, SUTCOPY replaces "*value1*" with "*value2*" to repair the test case. In NOCONTEXT, we fine-tune the best performing CLM by excluding the repair context from the input, meaning we include only the broken test as input and expect the repaired test as output.

3.4.4.3 RQ1.2 Results

We compute and report both exact match accuracy (EM) and CodeBLEU, as used by CEPROT for evaluating the test update generation task. However, there are two key differences in how we computed these metrics for TARGET when compared to CEPROT.

First, we used beam search with a beam size of 40 for our best model, CT5+. Consequently, TARGET generates 40 repair candidates for each test repair instance, whereas CEPROT does not use beam search and generates only one repair candidate per instance. This affects the computation of EM. TARGET considers there is an exact match if at least one of the 40 candidates matches the ground truth, while CEPROT only compares the ground truth with a single repair candidate.

Second, CEPROT generates the full test code as output, whereas TARGET generates only the repaired part of the test code. For computing CodeBLEU, CEPROT compares the full test code between the prediction and the ground truth, while TARGET compares only the repaired part. This difference affects the CodeBLEU scores because including the full test code inflates the CodeBLEU values. Indeed, since most of the test code remains unchanged in a repair, this leads to higher scores.

Baseline	EM	CodeBLEU
CEPROT [136]	21	60.4
TARGET (CT5+ & IO_2)	40.6	91.1

Table 3.6: Comparison of CEPROT with TARGET's best model, CodeT5+ using IO_2, on CEPROT's benchmark.

To ensure a fair comparison, we applied TARGET to CEPROT's test set and computed EM and CodeBLEU in the same manner as CEPROT. Specifically, we used only the first repair candidate out of our 40 candidates to compute EM and placed the repaired part within the full test code for computing CodeBLEU. Although CEPROT stated to have provided a script for CodeBLEU in their replication package, we were unable to locate it in their repository. Therefore, we used our own CodeBLEU implementation, which is included in our replication package.

Table 3.6 presents the results of two techniques on the 214 compatible test instances. According to the results, TARGET achieved a CodeBLEU score of 91.1 and an EM score of 40.6, significantly outperforming CEPROT, which scored 60.4 in CodeBLEU and 21 in EM.

To investigate the reasons behind TARGET's superior performance, we performed a comparative analysis, manually reviewing instances where TARGET successfully repaired the test cases while CEPROT failed. Details of five handpicked comparisons are provided in Appendix B.2. Our analysis identified several recurring failure patterns in CEPROT's repair attempts: failing to make any modifications to the test case (Figure B.9), deleting significant portions of the test case (Figures B.10 and B.11), making superficial changes, such as modifying only whitespace characters (Figure B.13), and making irrelevant or minor adjustments that did not repair the test case (Figure B.12).

We attribute TARGET's superior performance to three key factors. First, TARGET effectively utilizes SUT changes by prioritizing and selecting those most relevant to the test code. In contrast, CEPROT focuses exclusively on changes to the production method under test. Though this information is beneficial for repair, it may fail to capture the broader context needed for accurate test repair. This limitation is evident in Figures B.11 and B.13 in Appendix B.2. Second, CEPROT represents code changes by including the two versions of the full production method along with the change's edit sequence. In contrast, TARGET includes only the changed lines, significantly reducing the token count and allowing more test code and SUT changes to fit within the input. Figure B.10 illustrates how this difference enables TARGET to better utilize contextual information. Third, TARGET benefits from being fine-tuned on a substantially larger dataset of test repair instances (36.6k) compared to CEPROT (4.6k). Also, TARGET leverages a larger and more recent language model, CodeT5+ (770M parameters), whereas CEPROT uses the smaller and older CodeT5-base (220M parameters). These advantages are highlighted in Figures B.9 and B.12, where both approaches had sufficient repair contexts.

Further, Table 3.7 shows the result of baselines. The results indicate that CT5+ with

Baseline	EM	PR	CodeBLEU	BLEU
SUTCopy	10.8	13.8	61.9	52.1
CT5+ with NoContext	28.7	55.9	65.5	66.2
TaRGET (CT5+ & IO_2)	66.1	80.0	79.3	80.0

Table 3.7: Comparison of baseline models with TaRGET's best model, CodeT5+ using IO_2, on TaRBench.

NoContext consistently outperforms SUTCopy across all metrics, with improvements of 17.9, 42.1, 3.6, and 14.1 pp in terms of EM, PR, CodeBLEU, and BLEU, respectively. This outcome aligns with expectations and clearly indicates that CLMs are indeed effective at automating test repair.

Finally, TaRGET's best configuration (CT5+ with IO_2) consistently outperforms CT5+ with NoContext across all metrics with improvements of 37.4, 24.1, 13.8, and 13.8 pp in terms of EM, PR, CodeBLEU, and BLEU, respectively. This confirms that using the repair context proposed by our approach significantly enhances CLMs' performance. Also, this implies that depending only on a generic CLM for test repair may not produce effective results. It emphasizes the critical need for context-dependent fine-tuning and customization of CLMs.

RQ1.2 Summary

The best configuration of TaRGET (CT5+ with IO_2) consistently and significantly outperforms the baselines—Ceprot, NoContext, and SUTCopy—across all evaluated metrics. This finding highlights the effectiveness of using CLMs for test repair, provided that the model's input and output are carefully engineered and fine-tuned. Incorporating comprehensive and relevant information, as done in our approach by integrating the repair context, is crucial for achieving optimal performance.

3.4.5 Test Repair Performance on Test Repair Characteristics (RQ2.1)

Test cases can break for various reasons, each necessitating a specific category of repair. This research question investigates whether TaRGET's performance varies across repair categories. By addressing this question, we can gain a more detailed understanding of TaRGET's performance in test repair tasks, allowing us to determine their suitability for specific repair categories. Also, the results can highlight areas where future improvements are needed to effectively apply CLMs in the context of test repair.

3.4.5.1 Test Repair Categories

To answer RQ2.1, we examined the code changes involved in repairing test cases for all instances in TaRBench, and (1) categorized these changes into three main categories as discussed in the following, and (2) measured their complexity based on their number of abstract syntax tree (AST)-level edits.

Test repair categorization requires understanding the semantic intent behind broken and repair code differences. While traditional line-based diff tools compare textual changes at line granularity [137], they fail to capture structural edits. We instead leveraged AST differencing, which generates edit sequences reflecting structural transformations between two ASTs [138]. We computed AST-level edits using the GumTree Spoon AST Diff tool [139], a prominent AST differencing tool utilized in prior empirical studies [140]. For each test repair instance, we parsed the broken and repaired test code into ASTs, applied GumTree [139] to generate a sequence of edit actions transforming the broken AST into the repaired version, and recorded the total number of AST-level edits as a complexity metric, which is one of our measurements of repair complexity in this RQ.

To classify repairs into categories, we implemented a rule-based classifier mapping AST edit actions to semantic repair categories. For each AST edit action derived from GumTree [139], we analyzed three attributes: *Action Type* (Insert, Delete, Update, or Move), *Source AST Node Type*, and *Target AST Node Type*. AST node types represent programming elements such as *Invocation*, *ConstructorCall*, *Literal*, and *Assignment*. Inspired by existing repair categories from the test repair literature [8], we conducted a manual analysis and identified recurring patterns characterized by specific combinations of action types and node types. These patterns are detailed in our public repository [4]. We then categorized these patterns into three main repair categories:

1. **Invocation argument or return type change (ARG).** Adding, removing, or modifying arguments that are passed to an invocation, i.e., a method call or a constructor call, in the test code. This category also includes changing the expected return type of an invocation in the test code.

2. **Invocation change (INV).** Adding or deleting an invocation, such as adding a new method call to the test code, falls into this category. It also includes the replacement of an existing invocation. For instance, the repair might replace the constructor *Foo()* with *Bar()*. The key distinction between this category and the previous one is that the former involves changes in the arguments and the return types for the same method or constructor. However, in this category, the invocation itself is changed, regardless of the arguments and return type.

3. **Test oracle change (ORC).** Modifications to sections where test oracles are compared to the SUT's output, which include changes to assertions or expected exceptions. Modifications including changing the type of an expected exception or altering a line containing an assert statement—either changes to the type of assertion or its parameters—are counted in this category.

[4] https://github.com/Ahmadreza-SY/TaRGet/blob/master/repair-collection/repair_catg.py

Category	Runtime (%)	Compile (%)
ARG	8.3 (3781)	38.6 (17504)
ORC	6.2 (2798)	10.7 (4857)
INV	0.9 (408)	13.8 (6251)
ARG+INV	0.8 (354)	11.0 (4972)
ARG+ORC	0.7 (333)	3.0 (1349)
INV+ORC	0.3 (140)	1.7 (788)
ARG+INV+ORC	0.3 (130)	1.3 (603)
OTH	0.4 (184)	2.0 (921)

Table 3.8: Distribution of runtime and compilation failures across different repair categories

Note that a test repair may include more than one change category. In such cases, we assign it to a new category that combines multiple categories. Also, if a test repair does not belong to any of the categories above, we categorize it as Other (OTH). Finally, we computed performance metrics obtained from the best-performing model identified in RQ1.1 for each repair category and compared the results.

Across categories, repairs may stem from runtime or compilation errors. Table 3.8 presents a detailed breakdown of failures by category. Take the ARG category, for example. Depending on the code change in the SUT, ARG repairs can result from either a runtime or compilation failure. If an argument is added or removed from a method, it leads to a compilation failure in the test code that calls this method. However, if the method's logic changes to expect a specific format for an existing argument or return value, it could trigger a runtime exception from within the SUT method or an assertion failure from the test code.

3.4.5.2 RQ2.1 Results

Figure 3.9 shows the distribution of repair categories and the count of AST-level edit actions for 7,103 test case repairs in the TaRBench benchmark (test set). The figure shows that that the majority (80%) of repairs belong to a single category. Additionally, about 75% of repairs entail either a single or double AST-level edit, implying that the vast majority test repairs do not require extensive structural changes. This may be explained by test methods being inherently simple, in terms of logic and size.

Figure 3.10 presents two heatmaps showing the performance of the best model (CT5+ with IO_2) across repair categories and AST actions in terms of EM and PR metrics. The x-axis denotes the AST-level edit action count, and the y-axis denotes the repair category. Each cell in the heatmaps has three attributes: (1) the performance metric value, shown as the top number, (2) the color as determined by the performance metric value, with darker color indicating lower performance, and (3) the proportion of test repair instances (out of the 7,103 in the test set), shown as the bottom number.

Repairing a test involving multiple repair categories or AST actions is inherently more complex than repairing one focused on a single category or action. This complexity is con-

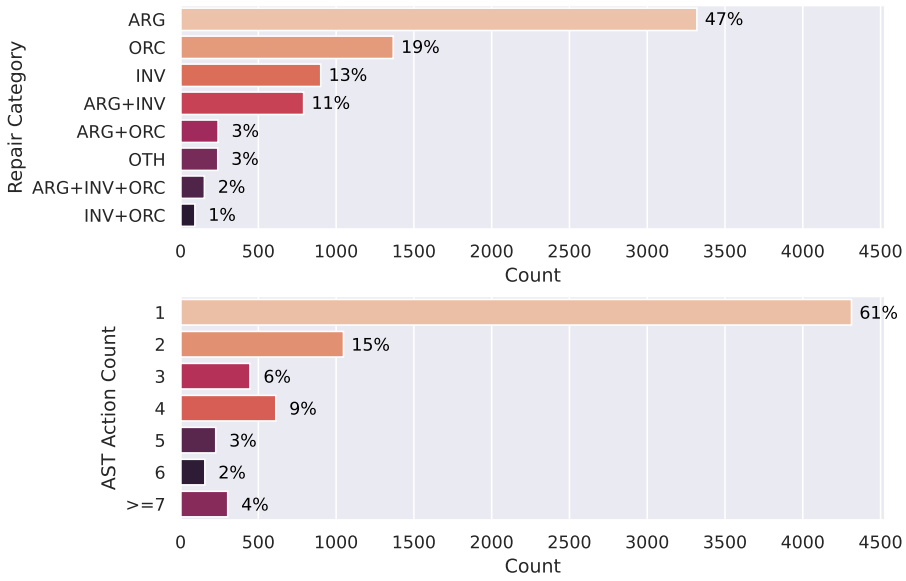

Figure 3.9: The distribution of test repair categories and AST-level edit action count within TaRBench's test set.

firmed by the results shown in Figure 3.10, which reveals a reduction in the model's performance as the number of repair categories and actions increases. Also, in Figures 3.10a and 3.10b, darker colors are notably concentrated towards the bottom-right of the heatmaps, where the number of repair categories and AST actions are higher. Thus it is safe to conclude that the model's performance degrades with more complex test case repairs, aligning with our expectations.

When comparing Figures 3.10a and 3.10b, we see a noticeable distinction: the bottom-right areas in Figure 3.10a are around the 20% spectrum and appear considerably darker than their counterparts in Figure 3.10b, which fall within the 60% spectrum. This suggests that in more complex repair scenarios, although the model shows a lower success rate in generating exact match repairs (approximately 20% of the time), it manages to generate repairs that execute successfully about 60% of the time. Hence, the model remains applicable to more complex repairs, though its accuracy decreases compared to simpler repair tasks. Arguably, certain plausible repairs might hold semantic equivalence to the exact match repairs, despite differences in syntax. However, performing a semantic comparison on these repairs is a costly task. Identifying an applicable and scalable solution within this context should be addressed by future research.

Analyzing the proportions of repair instances shows that nearly 70% of these instances are concentrated within the six top-left cells, identified by the presence of one or two AST actions alongside a single repair category. Focusing on this majority, the results highlight the model's effectiveness in repairing test oracles (ORC), achieving higher scores compared to the ARG category. Regarding EM, the model achieves 83% and 65% for the ORC category with one and two AST actions, respectively, surpassing the results for the ARG category with EM values of 72% and 58%. Similar results are achieved in terms of PR, with

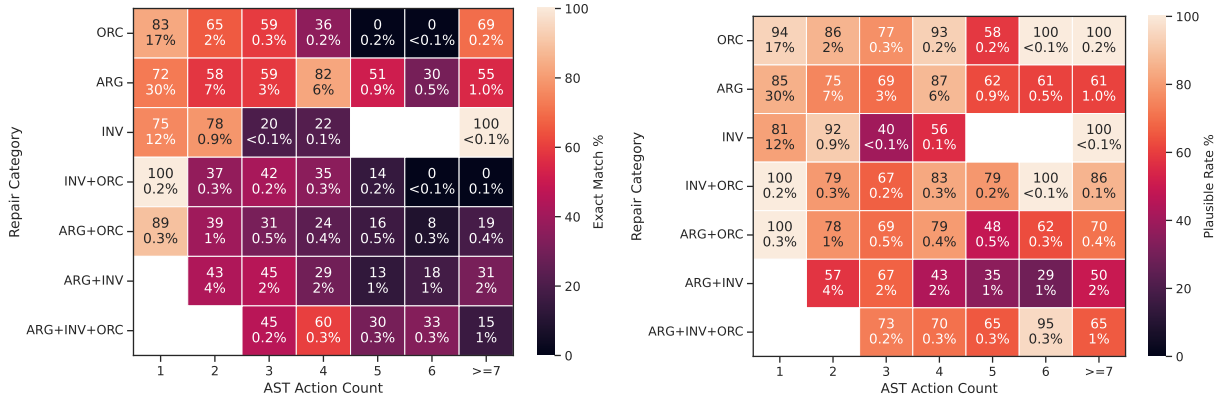

Figure 3.10: Performance analysis of CodeT5+ using IO_2 (the best model) across repair categories, correlating with AST-level edit actions. Each cell displays the performance metric value (top) with the respective proportion of instances (bottom).

the model achieving 94% and 86% for the ORC category, surpassing those for the ARG category with 85% and 75%, respectively. Our observations reveal that ORC test repairs primarily involve changing literal values or the test's expected exceptions. Conversely, in ARG repairs, the changes show a larger variety, including changes, additions, or deletions of arguments, which could range from constructor calls, method calls, to literal values, among others. Consequently, the scope of potential repairs within the ARG category is broader than that within the ORC category, rendering it a more challenging repair category.

We believe that an iterative repair approach, as a future direction, could enhance the performance of TARGET, particularly in managing more complex test repairs. For instance, although ITER [141] addresses program repair rather than test repair and is not a direct alternative to our approach, it is an influential technique that utilizes a specialized training method for iterative repair. Moreover, ITER supports multi-hunk repairs (also known as multi-location repairs [141]), allowing it to address complex scenarios involving modifications across multiple locations. In addition, generating synthetic training instances could further improve TARGET's performance. As mentioned earlier, complex repair types are underrepresented in our dataset. By generating synthetic data, we could introduce a greater variety of repair complexities in our training set, potentially enhancing the model's effectiveness in handling complex repairs. For example, a previous study [142] generated one million synthetic training samples for the program repair task, which demonstrated superior performance compared to traditional learning methods.

3.4.5.3 Qualitative Analysis

To complement the quantitative results and gain deeper insight into TARGET's performance in real-world scenarios, we conducted a qualitative analysis. By examining both successful and failed cases, we aimed to identify factors that contribute to TARGET's repair effectiveness and highlight areas for improvement. This analysis not only strengthens

the validation of our approach but also informs future advancements in test case repair. We manually analyzed many examples from different repair categories and projects, and selected eight representative and insightful cases that illustrate successes and common failures. To keep the main text concise, details regarding these examples are provided in Appendix B.1.

We present four successful examples that demonstrate both exact-match and plausible repairs, underscoring the effectiveness of TARGET and its intelligent repairing of test cases. In some examples (Figures B.5 and B.6), TARGET generated exact-match repairs by accurately capturing the relevant context of the SUT and identifying key code change patterns. In other examples (Figures B.7 and B.8), TARGET proposed a functionally equivalent repair, even though it did not exactly match the ground truth, maintaining the intent of the original test case. These examples highlight the model's ability to prioritize crucial information and apply learned patterns, even in challenging scenarios.

Our analysis of failed cases identified two main factors contributing to unsuccessful repairs: (1) when multiple distinct changes are required (Figures B.2 and B.3), and (2) when deeper contextual understanding of the SUT is necessary (Figures B.1 and B.4), especially from both unchanged and changed code. In one failure for instance (Figure B.1), the model was unable to repair an assertion due to a lack of contextual information about the underlying unchanged logic and API of the SUT. In another case (Figure B.3), the model partially predicted the correct repair but introduced syntax errors due to the high volume of the required changes.

Overall, the qualitative analysis highlights the strengths of TARGET in selecting, representing, and learning the repair context while revealing areas for future improvement, such as enhancing its capacity to process broader contextual information and manage multi-step repairs more effectively. These insights can guide future research to refine test case repair methodologies and address unresolved challenges.

RQ2.1 Summary

TARGET's performance declines as the complexity of repairs increases, particularly when the repairs involve a higher number or variety of code edits. While the model achieves a exact match accuracy of around 20% even for more complex tasks, its accuracy is notably higher for simpler repairs. We also observed that the model's performance in the ARG category is relatively lower compared to that in other simpler repair instances, for reasons explained above.
These limitations highlight areas of improvement in future work. A multi-step and iterative approach is a reasonable avenue for exploration. Also, diversifying the training set with synthetic data to better balance the frequency of repair categories and complexity is a promising direction.

3.4.6 Test Repair Trustworthiness Prediction (RQ2.2)

Being able to determine, in practice, whether to trust a proposed repair would make TARGET much more practical as practitioners would not waste time considering low quality repairs. The question is then whether we can accurately predict the performance of TARGET for a given broken test case.

3.4.6.1 Creating the Prediction Model

To address this research question, we first propose the following features, which we hypothesize may be good indicators of repair correctness to be used by the prediction model.

- **Similarity between the input's test context and the repair context.** We compute the similarity between the broken lines of the test code and the SUT's changed lines before the change, that are included in the input. We extract four features that relate to this similarity. The two initial features include the maximum and average cosine similarities derived from TF-IDF vectors. The third and fourth features involve the count of common AST nodes at hunk and node levels, respectively.
- **Overall complexity of changes in SUT.** We calculate the total number of changed files in the SUT as well as the total number of added and deleted lines.
- **Complexity of the test code.** We compute the number of lines (LOC) of the broken test method.

Secondly, we constructed a dataset using the above-mentioned features to train models for predicting whether a test is likely to be properly repaired. To achieve this, we used two distinct labels as the ground truth: (1) exact match repair and (2) plausible repair, hence creating two distinct prediction models. The labels indicating the model's success in generating exact match or plausible repairs were extracted from the results of the best-performing model. For instance, if the model generates an exact match repair for a particular broken test case, we identify it as a positive exact match label for that test. Since we have an imbalanced dataset (66% exact match repairs and 80% plausible repairs), we used random oversampling to improve performance for the minority class.

Thirdly, we conducted a 5-fold cross validation on the test set of TARBENCH. For each of the five folds, we trained a Random Forest (RF) model. We chose RF due to its explainability and ability to generate accurate models while being robust to overfitting [143]. RF enables us to measure the importance of each feature on the decision-making process. Our RF model incorporates all available features and predicts the probability that our model correctly repairs a given test. Finally, we report and analyze the average performance across folds, including average precision, recall, and F1 score.

Label	Precision	Recall	F1
Exact Match	87	88	88
Plausible	90	94	92

Table 3.9: Test repair effectiveness prediction results using the Random Forest classifier.

3.4.6.2 RQ2.2 Results

Table 3.9 presents the performance of the RF model, achieving precision, recall, and F1 scores of 87%, 88%, and 88% for the exact match label, and 90%, 94%, and 92% for the plausible label. The RF models are therefore highly accurate in predicting whether our best test repair CLM (CT5+ with IO_2) can effectively repair a given broken test case. Notably, the RF model operates exclusively on input information, without utilizing any data related to the output of CLMs. This model characteristic enhances its applicability since there is no requirement for additional prompting of the CLM to get extra information. Furthermore, enhancing the accuracy of the RF model should be possible with richer training data including a greater number of repair instances and a wider variety of repairs. Indeed, as developers use our CLM and offer feedback on the correctness of the generated test repairs in various test repair scenarios, there is significant potential for refining the RF model.

Let us discuss the practical implications of the RF model's results. Regarding precision, when the RF model predicts that our CLM's repair is trustworthy, developers would benefit from correct test repairs in 87% and 90% of the cases in terms of exact match and plausible repair, respectively. In other words, by relying on the RF model, in only 13% and 10% of the cases would developers need to further repair incorrectly generated CLM repairs. Therefore, such predictions provide useful guidance to developers.

Examining the recall values allows us to understand the number of missed opportunities in terms of generated correct repairs. In 12% of cases for exact match repairs and 6% for plausible repairs, the RF model misleads developers in suggesting that correct automatic repairs should not be trusted. Such cases do not entail extra effort for developers, as they would anyway have to repair such test cases manually without the CLM.

Last, we employed the *Permutation Feature Importance* technique to measure the impact of the various features on the RF model's performance. This method assesses the impact of individual features by shuffling each feature's values while keeping others constant and observing the resulting changes in the model's predictive accuracy. By averaging the feature importance scores obtained across 5 cross-validation folds, we identified that, the most important features, in order of importance, are (1) the average similarity between TF-IDF vectors of test broken lines and the pre-change SUT lines, (2) the count of common AST nodes between the two, and (3) the LOC of the broken test method. This suggests that the most important features in predicting the success of our CLM's test repair are related to the relationship between the broken lines of the test case and the pre-change SUT lines. Therefore, it is crucial to provide the most relevant SUT changes to the model's input while filtering out irrelevant changes. Further exploration focusing on this aspect, as well as incorporating data beyond the scope of SUT changes, remains a potential area

for future research.

RQ2.2 Summary

We introduced a simple machine learning technique that accurately predicts whether TARGET will successfully repair a given broken test case. This capability enhances the application of TARGET by identifying instances where developer intervention is necessary. Additionally, our feature importance analysis highlights the critical role of the repair context, suggesting that refining this aspect could lead to a higher success rate in automated repairs.

3.4.7 Impact of Fine-tuning Data Size on Test Repair Performance (RQ3.1)

Often, CLMs are initially trained for general tasks and then fine-tuned for specific tasks such as test case repair. However, fine-tuning is a time-consuming process that requires substantial amounts of data, especially in the context of test case repair where test breakages are not common [144,145]. Moreover, extensive fine-tuning datasets may be necessary to enhance the performance of a large CLM with hundreds of millions of parameters when compared to its pre-trained version. This RQ aims to analyze how the performance of test repair in CLMs is affected by the amount of fine-tuning data. By answering this question, we can gain valuable insights into the data requirements for fine-tuning to achieve effective test case repair.

3.4.7.1 Downsizing Fine-tuning Data

To address this research question, we utilized the training set from TARBENCH. We created four distinct fine-tuning data subsets, each containing 20%, 40%, 60%, and 80% of the original training data, achieved by excluding older data, for each project, based on commit dates. This process ensured a reduction in data size while preserving project diversity. Using these subsets, we fine-tuned four models using the best-performing settings identified in RQ1.1. Subsequently, we conducted an analysis to examine how varying sizes of the fine-tuning data impact the performance of test case repair.

3.4.7.2 RQ3.1 Results

Figure 3.11 depicts the impact of fine-tuning data sizes on test case repair performance. Initially, as the fine-tuning data size increases, we observe rising trends in both the EM and PR metrics, as expected. However, the effect is more noticeable on the EM than on the PR. For instance, increasing the fine-tuning data size from 20% to 100% results in an improvement of 7.4 pp in the EM, compared to a 2.7 pp improvement in the PR.

As a result, if we consider the PR as our criterion, our findings suggest that using approximately 7,000 fine-tuning instances yields a test repair performance similar to using

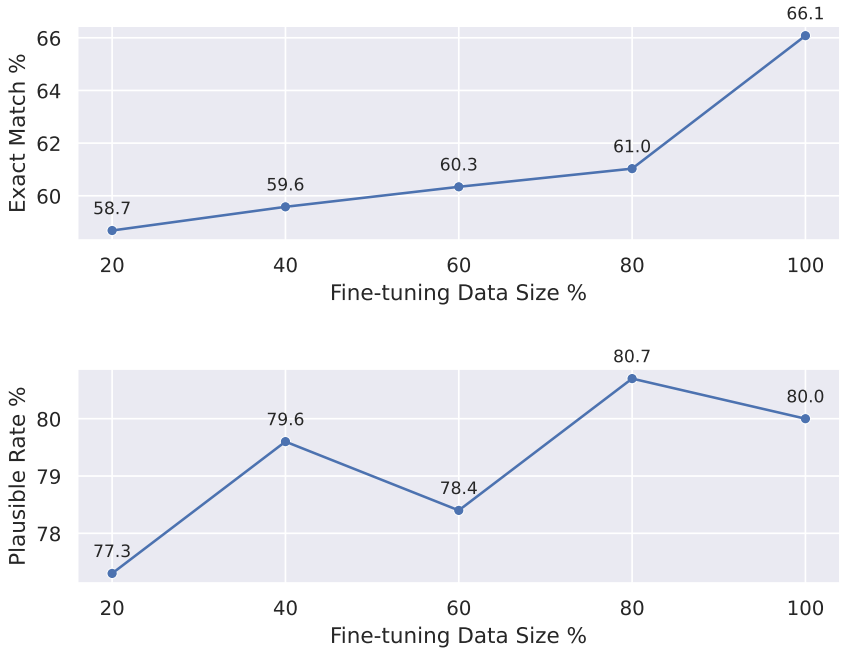

Figure 3.11: Test case repair performance on varying fine-tuning data sizes.

around 36,000 instances. On the contrary, when considering the more conservative metric, EM, as our criterion, a trade-off between performance and data size becomes evident. Moving from 7,000 fine-tuning instances to 36,000 and starting from an initial 58.7% EM, each addition of 7,000 instances leads to an average 1.85 pp improvement in EM.

RQ3.1 Summary

Our results show that having a larger fine-tuning dataset improves TARGET's test repair performance, particularly in terms of exact match accuracy, which is the most conservative evaluation metric. While collecting high-quality training data can be expensive, generating synthetic training data may be a beneficial and cost-effective approach worth exploring in future research.

3.4.8 Generalization of Fine-tuned Test Repair CLMs (RQ3.2)

To complement RQ3.1, this RQ aims to determine whether models that are fine-tuned on a number of projects can be applied to other unseen projects without further fine-tuning. The results can reveal whether a more generic approach can be employed to address test code repair, eliminating or minimizing the need for project-specific fine-tuning.

3.4.8.1 Excluding Projects from Fine-tuning Data

To address this research question, our approach involved creating random subsets of projects. Each subset was chosen to be excluded from the training set while keeping

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
Project-agnostic EM	55.4	70.3	60.7	69.7	62.9	67.6	55.7	64.8	34.3	49.2
Best (CT5+ & IO_2) EM	56.3	72.2	65.1	69.9	65.5	71.8	60.7	70.9	55.8	51.0
Evaluation Test Instances	453	327	476	1169	437	632	458	2135	534	482
Excluded Train Instances	2583	1819	2613	3445	2023	3915	2681	13049	1919	2592

Table 3.10: Generalization analysis results of fine-tuned test case repair models: The table shows the exact match accuracy (EM) and the counts of training and test instances of the selected projects across 10 folds. In each fold, six random projects were selected and excluded from the training data. EM values are specifically computed on the testing data solely from the excluded projects in each fold.

the test set and validation set unchanged. More precisely, we formed 10 folds by using a stratified sampling method, each fold comprising a specific number of projects. Six projects were sampled for all folds, except for the 10th fold, which included five projects due the dataset containing a total of 59 projects. The stratified sampling approach ensured that each fold represented the overall project population, categorized into three segments based on the number of test repair instances: small, medium, and large. The data in the last two rows of Table 3.10 displays the count of test instances from the chosen projects in each fold, along with the number of train instances that were excluded from the training set.

Subsequently, for each fold, we fine-tuned a new model based on the best-performing settings (CT5+ & IO2) identified in RQ1.1. The newly fine-tuned models were trained on the modified dataset, excluding the subset of projects corresponding to their respective fold. Finally, to assess the effectiveness and generalizability of these fine-tuned models, we conducted evaluations on the excluded projects. This evaluation process provided a measure of the models' performance on unseen project data, offering insights into their ability for generalization.

3.4.8.2 RQ3.2 Results

The results presented in Table 3.10 report the EM values of two models across 10 folds, as outlined earlier. We excluded the PR metric from this analysis due to the substantial computational resources required to execute all candidate repaired test cases across all folds; the EM metric is sufficient to address this research question. The *Project-agnostic* model refers to the model that was not exposed to the selected projects during fine-tuning in each fold, whereas the *Best* model denotes the top-performing model identified in RQ1.1, fine-tuned on the entire training dataset. As shown in Table 3.10 , the difference between the two models in terms of EM is relatively small, with an average EM difference of 4.87 pp.

To assess the statistical significance of the difference between the Project-agnostic model and the Best model across folds, we conducted a paired t-test. Our dataset satisfied the normality assumption, confirmed by the Shapiro-Wilk normality test, with p-values of 0.16 and 0.25 ($p > 0.05$) for the Project-agnostic EM data and the Best EM data, respectively. The paired t-test yielded a statistically significant difference (p-value = $0.034 < 0.05$)

between the two models. Moreover, to measure the observed difference's magnitude, we computed Cohen's d as the effect size, resulting in a value of 0.513. According to Cohen's guidelines [146], this effect size falls within the moderate range, leaning closer to a small effect size, given that values below 0.5 are categorized as indicative of a small effect.

The EM values of the Project-agnostic models, except for two of the folds, are above 50%, averaging 59.1% across all folds. Also, despite the statistical significance in the differences observed across folds, we note a small to moderate effect size and an average difference of 4.87 EM pp. These findings suggest that Project-agnostic models represent an acceptable alternative. Although Project-agnostic models do not fare quite like models fine-tuned on the entire training data, their results imply that TARGET is not restricted solely to project-specific fine-tuning and can generalize to unseen projects.

The above results have significant practical implications regarding applicability. Currently, the most effective test repair methods [93] rely on dynamic symbolic execution, requiring project-specific analysis and execution, which poses a considerable challenge in practice. In contrast, as suggested by the above results, fine-tuned CLMs can produce satisfactory outcomes for unseen projects without the need for any dedicated project analysis.

The primary goal of this RQ is to evaluate the impact of cross-project training by comparing the performance of our best model against project-agnostic models, regardless of the specifically performance level observed of each individual fold. Notably, our best model achieves an EM of around 51% and 56% on folds 1 and 10, respectively, while achieving around 72% and 71% on folds 6 and 8.

To further investigate performance variability across different folds, we employed the same metrics used in RQ2.1, specifically the number of AST-level edit actions and repair categories (ARG, INV, ORC). Both the AST actions metric, outlined in Section 3.4.5, and the number of repair categories, ranging from 1 to 3, measure the complexity of the repair. For each fold, we calculated the average value of these metrics across repair instances and computed the Pearson correlation coefficient (r) between each metric and the EM. For our best-performing model, we found negative correlations of $r = -0.57$ for AST actions and $r = -0.59$ for repair categories. The project-agnostic models showed $r = -0.21$ and $r = -0.51$, respectively. These results indicate a moderate but significant negative correlation, suggesting that as AST actions or repair categories increase, model performance tends to decrease. This trend explains the lower EM values in certain folds, a pattern consistent in both project-agnostic and our best-performing model. For instance, in fold 10, where the EM is 51%, the average number of AST actions are 3.28 with a standard deviation of 3.22, whereas in fold 8, where the EM is 71%, this average is 1.90 with a standard deviation of 2.07. This analysis aligns with the findings of RQ2.1, reinforcing the observed relationship between repair complexity and performance without altering our conclusion in this RQ regarding the limited impact of project-agnostic models on performance.

RQ3.2 Summary

Our findings show that fine-tuning on project-specific data has a relatively limited impact on performance. Despite the challenges introduced by complex test repair tasks in certain projects, this suggests that TARGET provides a relatively generalizable solution that is not project-specific.

3.4.9 Implementation and Resources

Implementation. We implemented CLMs using the Hugging Face Transformers [147] library and the Accelerate [148] library. In each fine-tuning setting, we trained the model for 4 epochs and employed an early-stopping strategy, which involved monitoring the validation loss after each epoch. If the validation loss did not decrease after 1 epoch, we stopped fine-tuning and selected the model with the smallest validation loss. We configured a beam size of 200 for PLB and 40 for CT5+ and CG, along with batch sizes of 8 for PLB, 1 for CT5+, and 2 for CG. We used the Adam optimizer with weight decay (AdamW) and set a learning rate of 1×10^{-5}, in combination with a cosine learning rate scheduler.

We anticipate that the primary contributions of our work, including relevant hunk selection, representation, and fine-tuning of CLMs, are not dependent on any specific programming language and can therefore be widely adopted. The CLMs we utilize are pre-trained on multiple languages, such as Java and Python, and our most effective hunk selection strategy is language-agnostic, as it relies on the textual content of code changes. However, the current implementation of our approach focuses specifically on Java, particularly for repairing JUnit tests. To achieve optimal results for other programming languages, we recommend additional efforts in gathering language-specific test repair data and selecting a CLM with pre-training that is more tailored to the target language.

Computation Resources. We conducted our fine-tuning experiments on a machine with two Nvidia Quadro RTX 6000 GPUs (each with 24GB GPU memory), an Intel Xeon Gold 6234 16-Core CPU, and 187GB of RAM. We used the Digital Research Alliance of Canada [1] servers for concurrently executing test cases at a large scale.

3.4.10 Threats to Validity

External Validity. Two threats to the external validity of our evaluation are the use of possibly simple test repairs and the applicability of TARGET to real-world development scenarios. To mitigate the former, we have taken steps to ensure our evaluation set is fair and realistic by excluding simple test repairs that can be addressed by an IDE or CI automation tools. Specifically, we identified and excluded trivial repairs (discussed in Section 3.4.2.4), such as renaming class or method names, as their inclusion would have inflated our results. The number of these trivial repairs was not negligible, reinforcing the importance of their exclusion for an accurate assessment.

The applicability of TARGET to real-world development environments was carefully considered in our data collection process. We aimed to replicate real-world conditions

by sourcing high-quality data samples from over 1,200 representative Java open-source projects on GitHub, focusing on test case repairs extracted from actual commits. We also executed the test cases to further ensure the authenticity and reliability of the repairs. No synthesized elements (e.g., faults and repairs) were introduced; all data were derived from real-world development environments.

Internal Validity. We acknowledge that the PR metric has a limitation in not always accounting for the preservation of the tests' semantics and intent. For instance, the removal of assertions could be considered a plausible repair, even if the ground truth repair does not involve such a change (though there are cases, such as when a feature is removed from the SUT, where this is necessary). In the following paragraphs, we discuss the strategies employed to mitigate this issue.

Our approach focuses on repairing only the breakage lines, which is discussed Section 3.3. This limits the capability of the model to change the semantics of the test code as it can only change a few lines. Also, the PR metric is not the only metric we used for evaluation. We also employed three other metrics: exact match accuracy (EM), CodeBLEU, and BLEU. These three metrics are highly sensitive to unwanted changes, such as removing assertions, because they are text-based and not execution-based. Therefore, the diversity of the metrics we use mitigates the limitation of the PR metric.

Further, to assess the impact of the PR metric limitation on our best-performing model (CT5+ with IO_2), we conducted a detailed analysis. We focused on predictions where the model generated at least one plausible repair candidate without producing any exact matches, as exact matches are definitely correct and irrelevant to this particular analysis. From these non-exact-match and plausible instances, we identified a subset of 377 cases where the keyword "assert" appeared in either the ground truth repair or the model's plausible repair candidate. In the most conservative scenario, we considered the possibility that all 377 of these instances might be misclassified as plausible when they are actually non-plausible. For instance, 123 of these cases involved the deletion of assertions, while the ground truth did not involve such deletions. Even under this worst-case assumption, where all 377 instances are reclassified as non-plausible, the PR value for our model would only decrease by 5.3 percentage points, from 80.0% to 74.7%. This small reduction demonstrates that the overall impact on the model's performance is limited.

Another potential threat to validity is data leakage, where the test set we used might have been previously seen by the code language models during their original pretraining phase. To measure the impact of this potential threat, we identified the release dates of each model that we used and tested our best performing configuration on test data that was created after the release date, comparing it with results on data before the release date. The results showed that there were no significant differences, and the model actually performed better on the data after the release date, indicating that data leakage from pretraining is unlikely to have influenced our results.

3.5 Related work

In this section, we discuss existing research related to the automated repair of broken tests, along with an analysis of their challenges. Additionally, we provide a concise overview of automated program repair studies that use language models. These two areas of research share similarities in their use of language models for repairing source code, even though their purposes differ.

3.5.1 Automatic Repair of Broken Test Cases

Despite the importance of repairing broken tests, there are only a few studies that focus on this problem, as discussed below. The first work in this context is a tool called *ReAssert* [90] that repairs broken JUnit test cases using a set of heuristic repair strategies (e.g., replacing the expected value in an assertion statement). Daniel et al. [88] extended *ReAssert* by using symbolic execution to repair broken tests. In order to repair the tests this way, the expected value of the assertion is modified using a solution to symbolic constraints. These constraints are built based on the literals which contribute only to the assertion's expected value. In this way, the actual value of the assertion remains unmodified. The solution to the symbolic constraints are used to update literal values in the test case, and if no solution exists, the test is deemed unrepairable [88].

Mirzaaghaei et al. [91] developed a framework named *TestCareAssistant* (TCA) to repair broken tests or generate new tests based on existing ones. TCA repair algorithms focus on repairing tests whose covering code undergoes specific types of changes, such as adding parameters to a method being covered by a test. The repair algorithms work by replacing variables and values within the test to identify potential candidate repairs [91].

Xu et al. [92] proposed *TestFix* that uses a genetic algorithm to repair broken test cases by finding a sequence of adding or deleting method calls which results in a repaired test. *TestFix* can only repair tests with a single assert statement [92].

Li et al. [93] developed *TRIP* that focuses on preserving the intent of the test during its automatic repair [93]. *TRIP* generates candidate repairs by modifying the test using a search-based technique. The test case is modified while considering the test-accessible elements (e.g., a public method). The search algorithm replaces broken calls to elements that are no longer test-accessible with one or more calls to elements that are accessible in the updated SUT [93]. *TRIP* uses dynamic symbolic execution of the original test and repair candidates to determine the similarity between their intent, which is used to rank candidates [93].

Our analysis of the aforementioned existing work reveals the following three main issues that our work (TARGET) aims to address.

Repair Categories: Overall, as discussed in Section 3.4.5, a repair category may cover one or more change categories including: (1) Invocation argument or return type change (ARG), (2) Invocation change (INV), and (3) Test oracle change (ORC).

Project	Repair Categories	Evaluation Benchmark		Reproducibility	
		# Projects	# Tests	Repeatable	Reusable
ReAssert [90]	Test oracles	6	170	No	No
Symbolic Test Repair [88]	Repairs that require modifying literal values	14	235	No	No
TestCareAssistant [91]	Method parameters and return types	5	138	No	No
TestFix [92]	Single assertion statements	1	6	No	No
TRIP [93]	All except repairs requiring long sequences of functional calls or repairs involving generic types	4	91	Yes	No
TARGET	Any repair category	59	45,373	Yes [10]	Yes

Table 3.11: Criteria for comparing our work with existing work

Unlike existing work, thanks to our reliance on language models, TARGET is not limited to handling specific repair categories, with the caveat that it will of course perform better on repair categories that are more common in the training dataset. As shown in column *Repair Categories* of Table 3.11, TARGET is capable of repairing failing test oracles, in contrast to *TestFix* [92]. *ReAssert* [90] and *Symbolic Test Repair* [88] are unable to modify method calls in the broken test, which is something that TARGET can accomplish. Finally, unlike *TestCareAssistant* [91], TARGET is able to modify the literal values of variables. Thus, TARGET is a more generic approach compared to those discussed above. TARGET is also a comprehensive test repair tool capable of addressing a combination of repair types covered by existing studies, while the other described approaches would need to be combined with one another to achieve similar comprehensiveness.

Evaluation Benchmark: As shown in the *Evaluation Benchmark* column of Table 3.11, existing studies utilize different benchmarks to evaluate their proposed approaches, with none of theses studies using common benchmarks to enable comparisons. Moreover, reported accuracy percentages range from 44.7% to 100% in terms of plausible repair accuracy (proportion of tests that run without a failure after being repaired), using benchmarks containing less than 250 broken tests. Such limited benchmark size raises concerns about the validity of their evaluation regarding the effectiveness and applicability of these approaches across various software projects and repair scenarios.

To address this limitation, we evaluated TARGET using a significantly larger and more comprehensive benchmark, TARBENCH, which is constructed from 59 open-source Java projects, comprising a total of 45,373 broken tests. In this evaluation, TARGET generated plausible repairs for 80% of the broken tests, demonstrating its effectiveness and scalability on a much larger and more diverse set of real-world scenarios.

Reproducibility and Future Research: As shown in the *Reproducibility* column of Table 3.11, all studies, except one, do not provide their benchmarks and the source code of their implementations and experiments, thus making their replication impossible. Also, in the case of *TRIP* [93], only a dataset comprising 91 broken tests is made available. However, *TRIP*'s source code is unavailable and that makes it unusable and inapplicable to other projects. Consequently, none of the existing studies are deemed reproducible. To address this issue and provide a foundation for future research in the context of broken test repairs, we provide the implementation of TARGET, TARBENCH, and the script for

reproducing all experiments [10, 11].

Additionally, extending existing solutions may prove challenging due to limited access to their source code and their primary focus being Java and JUnit, with the exception of *Symbolic Test Repair* which was also evaluated on .NET tests [88]. Similarly, TaRGET was fine-tuned specifically to repair tests written in Java with the JUnit framework. Nonetheless, it is worth noting that in contrast with existing solutions, fine-tuning TaRGET for repairing tests in other languages and testing frameworks can be easily achieved. This is because the underlying language models are pre-trained on multiple programming languages. For example, *PLBART* underwent pre-training on both Python and Java [104].

We highlighted above reproducibility issues, noting that most works do not provide their benchmarks and implementation details, making it impossible to compare them with our approach. Though *TRIP* [93] made its dataset and repair tool available, we encountered significant challenges in comparing TaRGET with *TRIP*. Firstly, the *TRIP* repair tool failed to execute on our dataset. Since the authors only provided executables without the accompanying source code, we were unable to troubleshoot and resolve the failure. Also, we were unable to get assistance from the authors. Secondly, our methodology requires access to the Git repositories of the projects to collect test repair data. However, the TRIP dataset does not include links to Git repositories, commits, or tags. Consequently, we were unable to perform a meaningful comparison between TaRGET and TRIP.

Additionally, being the most recent and closest work to ours, we empirically compared Ceprot [136] with TaRGET in Section 3.4.4 to evaluate their respective performance. In the following, we analyze in depth the issues and limitations of Ceprot and its empirical validation in comparison to TaRGET and TaRBench. This analysis includes the examination of the dataset, approach, and experimental design.

Dataset Diversity and Quality: Ceprot aligns test methods with their corresponding production methods by matching file paths and method names. It then defines a test repair as a co-evolving production-test pair within the same commit. When both a test method and its associated production method are modified in a commit, Ceprot labels the pair as an obsolete test case repair. However, this approach limits the diversity of the dataset. A test method might interact with multiple classes or methods in the production code for purposes such as initialization, which are not directly tested in the specific test case. These additional classes or methods may also need to evolve alongside the production code, but Ceprot fails to identify these cases.

Additionally, this approach can potentially collect instances that are not genuine test repairs. For example, Ceprot might mistake test code refactoring cases for obsolete test repairs. If a variable is renamed in both the test and production code, Ceprot incorrectly identifies it as a repair instance, even though it is not a true repair. Ceprot can also capture unsuccessful or partial repairs because it does not verify the success of the repair.

In contrast, TaRBench addresses these limitations by ensuring greater dataset diversity and quality through a three-step test execution process. First, TaRBench verifies that the test case works correctly in its original state by executing it on the original production code and confirming its success. Second, it confirms that the test case has become

obsolete by executing the original test code on the updated production code and expecting it to fail. Finally, it ensures that the updated test code executes successfully on the updated production code. This method effectively overcomes the limitations of CEPROT by providing a dataset with higher quality and more diversity.

Trivial Repairs: TARBENCH identifies and excludes trivial repairs, i.e., renaming class or method names, which can be automatically accomplished by code editors and IDEs. This ensures that its evaluation results are realistic. In contrast, CEPROT does not follow this practice. In our application of TARGET to CEPROT's subset of 214 test instances (Section 3.4.4), we identified 37 (17%) trivial repairs. Including these trivial repairs produces overly optimistic results.

Dataset Size: TARGET is fine-tuned and evaluated on a significantly larger dataset compared to CEPROT. Specifically, CEPROT includes 4,676 training instances, whereas TARBENCH comprises 36,639 training instances. This substantial difference enhances the effectiveness of the fine-tuning process for TARGET. Additionally, CEPROT has only 520 evaluation instances, in contrast to the 7,103 evaluation instances in TARBENCH. This larger and more varied evaluation set provides a more diverse and realistic assessment of the model's performance.

Dataset Issues: In addition to the data limitations previously mentioned, our analysis of CEPROT's data revealed other issues. First, we observed discrepancies in the commit attribution between test cases and focal (production) methods. Specifically, in 511 instances (11%) within the training set and 60 instances (11%) within the test set, the commit attributed to the test case differed from that of the focal method. This contradicts the authors' assertion that the test and focal methods are changed in the same commit. Moreover, 33 instances from the training set and 2 instances from the test set even differed in the project associated with the commits.

Second, we identified instances where the "test_src" and "test_tgt" values, which are preprocessed code and are utilized for training and testing, were completely identical. This occurred in 234 instances (5%) of the training set and 22 instances (4%) of the testing set, indicating no code change in the test repair, potentially leading the model to learn to generate outputs that exactly match the inputs.

Third, we found that in 355 training instances (7%) and 36 testing instances (7%), the method names of the source and target test code were different. Since the authors did not provide clarification on how they matched these cases, they are potentially mismatches.

These findings highlight inconsistencies in the data, which could impact the validity of the authors' results and the overall reliability of their model.

Limitations: We identified two limitations in CEPROT's approach and experiments that TARGET effectively addresses. First, CEPROT's authors state that their train-test split is done randomly. However, for a realistic evaluation, it is crucial that more recent test case repairs are not included in the training data. This ensures that the model is not exposed to future information, which could contain the solutions to older test repairs. To address this, in each project within TARBENCH, we ensured that older commits were included in the training set while newer commits were reserved for the test set.

Second, CEPROT's implementation has a restrictive token limit of 150 for the test method, focal method, and edit sequence, truncating any instances that exceed this limit. Additionally, CEPROT imposes a 150-token limit for output generation. In contrast, TARGET supports a much larger token limit of 512 for both input and output. This allows TARGET to process the full test code and all relevant SUT (System Under Test) code changes, ensuring a more comprehensive and accurate analysis.

To conclude, in addition to outperforming (Section 3.4.4) the most recent and relevant approach (CEPROT), TARGET offers a more comprehensive and adaptable solution for repairing broken test cases. TARGET leverages the flexibility of language models to address a wider range of repair categories. Additionally, in contrast to CEPROT, TARGET is fine-tuned and evaluated using an extensive dataset (TARBENCH), which provides a large-scale, realistic assessment of repair effectiveness. This also addresses the scalability and validity concerns present in other studies. Finally, TARGET distinguishes itself by being fully reproducible and extensible, with its codebase and dataset openly available, thereby setting a new baseline for future research in the field.

3.5.2 Automated Program Repair

Automated program repair (APR) is a closely related research field to automated test repair, as both aim to automatically repair source code. However, APR focuses on repairing faulty code by taking a defective program along with a test suite that exposes the program's faults. The goal of APR is to generate a repair that corrects the program's faults to ensure that the provided test suite passes comprehensively.

While APR techniques are effective at repairing faulty code, they are not applicable to repairing broken test cases due to (1) APR's correctness and training relying on a set of tests, whereas in test case repair the tests themselves are flawed, (2) APR addressing problems arising from flawed behavior in the SUT code. In contrast, broken tests result from changes in the SUT. This necessitates the provision of SUT changes along with the broken tests to the model to learn repairs as we will discuss in Section 3.3. Finding and providing SUT changes is challenging given the limited input size of language models and large sets of SUT code changes.

Over the last few years, there has been significant work put into the application of language models to repair program faults. Most applications tend to achieve a 15% to 20% repair rate on program faults, with the best results reach repair rates as high as 23%. Fortunately, there are a couple of widely used benchmark datasets designed for APR, meaning that different studies are easily comparable. One of the more commonly used datasets is *Defects4J* [149], which contains hundreds of bugs from Java programs. There are two widely used versions of Defects4J, v1.2 and v2.0 which contain 393 bugs and 837 bugs, respectively [150]. The other commonly used dataset is *QuixBugs* [151], which contains both Java and Python bugs. *QuixBugs* is a much smaller dataset, containing only 40 bugs.

Liu et al. [152] built a tool called *TBar*, which uses templates to repair bugs in Java programs. These templates are patterns of common bugs which specify a strategy to use

for repair (e.g. inserting a null check before a buggy statement). Liu et al. found that *TBar* could repair 18.7% of the considered bugs [152]. *TBar* is commonly used as a point of comparison for other subsequent works in this area.

Jiang et al. [153] built a neural machine translation model for APR called *CURE*, which uses novel beam search strategies to select repair candidates. *CURE* was able to repair 19.2% of analyzed bugs, and the authors noted that some of those repairs were cases where state-of-the-art techniques failed due to having no applicable repair patterns [153].

Drain et al. [154] presented *DeepDebug*, a model which uses pre-trained transformers to generate fixes for faulty functions written in Java. *DeepDebug* achieves a 14.9% exact match accuracy overall. Drain et al. also note that datasets for fault-related changes are collected via heuristics (e.g. all changes in a project repository which contain specific words such as "bug"), resulting in noisy datasets that impact their accuracy [154]. Zhu et al. [120] developed *Recoder*, which uses a tree-based Transformer [107] in order to generate program repairs. *Recoder* achieved a 16.7% repair rate, which outperformed state-of-the-art models on the same dataset [120].

Ye et al. [110] produced a neural translation model, *RewardRepair*, that was designed to produce compilable patches, as they noted that previous models tend to generate patches which do not compile. They do this by factoring compilation results in the training loss function. *RewardRepair* was able to repair 23% of the assessed bugs, and had a 45.3% compilation rate in the top 30 candidate patches [110]. Xia and Zhang [155] proposed *AlphaRepair*, which was designed to produce patches without fine-tuning the underlying pre-trained model on repair datasets. This was done with the intent of avoiding common issues regarding the quality and quantity of program repair datasets [155] required for fine-tuning. The buggy line is completely masked before being fed into *AlphaRepair* along with surrounding code context, so that the APR problem is framed as a code generation problem. *AlphaRepair* was able to repair 18.9% of the utilized buggy programs [155]. Jiang et al. [156] developed an APR approach called *KNOD*, which aims to use domain knowledge to inform candidate patch generation. Domain knowledge is used to train *KNOD* by using grammar logic formulae as part of the loss function. *KNOD* achieved a 16.8% repair rate on the analyzed benchmarks [156].

Jiang et al. [124] recently studied the performance of language models and DL-based APR techniques in order to compare the performance of program repair techniques. They applied four different language models to the APR problem: *PLBART* [104], *CodeT5* [115], *CodeGen* [157], and *InCoder* [158]. These language model-based techniques were compared to four DL-based APR techniques: *CURE* [153], *RewardRepair* [110], *Recoder* [120], and *KNOD* [156]. Results showed that language models are able to be competitive with and surpass the DL-based techniques [124]. The latter generated 11.6% correct repairs on average, while the former were able to generate 10.7% correct repairs on average before fine-tuning, and 22.3% correct repairs on average after fine-tuning. Jiang et al. specify that data leakage is a point of concern for language model-based techniques, as there is a risk that code from open-source code repositories is used in the training dataset, and later contained in the test dataset when the models are evaluated [124].

We can perform a comparison of our results to the results of APR techniques by com-

paring their rate of plausible repairs with ours. For APR, a plausible repair is defined as a repair which passes the test suite revealing the fault [152], [153], [110], [120], [155]. Although we do not have a test suite to determine plausible repairs in the context of test repair, this is similar to how we define plausible test repairs. A plausible repaired test passes on the new SUT version. The APR methodologies achieve a PR that ranges from 21.1% to 32.1%, with an average of 26.1% [152], [153], [120], [155], [156]. In contrast, we were able to achieve a PR of 80%. Though we cannot use the same benchmarks and there are significant differences in the problems being addressed between APR and test repair, this comparison provides a rough indication of how comparatively good TARGET's results are.

3.6 Conclusion

This chapter introduces TARGET (TEST **R**EPAIR **GE**NERA**T**OR), a new method that utilizes language models to automatically repair broken test cases. TARGET formulates test repair as a language translation task and fine-tunes pre-trained CLMs. The best-performing CLM (CodeT5+ 770M) achieves a 66.1% exact match accuracy (EM) and an 80% plausible repair accuracy (PR) on an extensive benchmark.

Beyond TARGET, we developed TARBENCH, a benchmark comprising 45,373 broken test repairs across 59 distinct projects, making it by far the most comprehensive benchmark publicly available. Additionally, we address crucial research questions through a large-scale empirical study to explore different configurations of TARGET, analyze its effectiveness across repairs, introduce a model for predicting the repair reliability of TARGET, and assess its generalizability across diverse software projects.

We show that leveraging the test repair context in our best configuration outperforms the performance obtained without utilizing this context. We observe notable improvements of 37.4, 24.1, 13.8, and 13.8 percentage points in terms of EM, PR, CodeBLEU, and BLEU, respectively. Additionally, we noted a small difference between the performance of PLBART 140M and CodeT5+ 770M, the smallest and largest CLMs in our study, respectively. Thus, a high-performing CLM can be fine-tuned and used at a significantly reduced cost without significantly sacrificing effectiveness.

Our findings also show that the effectiveness of TARGET degrades when handling more complex test repairs, particularly those involving multiple repair categories or AST edit actions. Furthermore, our approach seems more effective at repairing test oracles when compared to other repair categories. Moreover, our repair reliability predictive model helps make TARGET more applicable in practice, offering practical guidance to developers by filtering out low-quality generated repairs, thus saving valuable time and effort. As it may be intuitively expected, we found that a decrease in fine-tuning data size negatively impacts test repair effectiveness in terms of EM. Finally, our results demonstrate that TARGET is not limited to project-specific fine-tuning and has the capability to generalize across previously unseen projects.

In conclusion, the findings detailed above strongly suggest that TARGET can be an

effective and practical solution to support test repair. Our study is the first that applies CLMs for repairing broken tests. With the promising results obtained, we anticipate several avenues of research for refining this approach. Potential improvements include, but are not limited to: (1) an investigation of plausible repairs that are not an exact match for their semantic equivalence with the ground truth, measuring their quality and reliability, (2) exploring ways to further improve the repair context by expanding the search scope beyond SUT changes and by eliminating irrelevant context, and (3) extending the diversity of programming languages in the benchmark.

Chapter 4

Conclusion

In this chapter, we summarize the contributions of the thesis and discuss potential directions for future work.

4.1 Summary of Contributions

This thesis advances the state-of-the-art through four key contributions: (1) development of comprehensive methodologies, (2) extensive empirical validation, (3) practical implementation guidelines, and (4) creation of reproducible and reusable benchmarks. The specific research objectives are organized as follows:

TO_1: Our work introduces a data-driven feature model that enhances ML-based TCP by integrating multiple CI data sources, including test execution history and source code metrics. Through extensive validation on 25 open-source projects with over 21,500 builds, we explore how different feature combinations affect prioritization effectiveness. The research also examines the practical trade-offs between the cost of collecting various features and their impact on prioritization performance, providing actionable insights for practitioners. To foster further research in this area, we develop and publish a comprehensive benchmark containing real-world projects and the necessary tooling for experimentation, enabling meaningful comparison of future prioritization approaches.

TO_2: We propose a novel transformer-based approach that leverages pre-trained code language models with specialized context prioritization techniques from the SUT. Our work includes developing a new benchmark containing over 45,000 test repair instances across 59 diverse open-source projects, significantly expanding the evaluation scope beyond existing datasets. The research explores various input formatting strategies and model architectures, quantifying their effectiveness through experiments. Additionally, we investigate the relationship between repair complexity and model performance, developing a predictive framework to help engineers assess repair reliability in practical settings. Our approach is designed with generalizability in mind, evaluating performance on previously unseen projects without requiring project-specific model adaptations.

This thesis takes a holistic approach to improving regression testing in CI environments by addressing both execution efficiency through ML-based test prioritization and development productivity through automated test repair. All research tools, benchmarks, and replication packages are made publicly available to support further research in the field.

4.2 Discussion: Research Implications and Broader Context

This section examines the broader implications of our research on test case prioritization and test code evolution. We discuss practical implications, model architecture considerations, and the applicability of our techniques beyond the scope of this work.

4.2.1 Practical Implications

In the following, we provide a summary of the practical implications of our findings for practitioners and researchers working in the same area.

Test Case Prioritization (TCP): Our findings show that different types of features used for TCP have varying costs of collection and differing impacts on effectiveness. Ideally, the most effective TCP models are achieved by using all available feature types. However, this may not always be feasible due to resource constraints. We suggest the following based on practical trade-offs:

- When feature collection cost is a concern or some features are inaccessible, practitioners can rely solely on execution history features (e.g., past failures, execution times). This approach significantly reduces overhead while still maintaining an acceptable level of TCP effectiveness.
- When even the cost of training and deploying machine learning models is prohibitive, we recommend using the failure rate of test cases as a heuristic—prioritizing tests in descending order of their failure rate. While this heuristic performs worse than ML-based approaches on most subjects, it remains the most effective single-feature heuristic and offers the lowest overhead, making it a practical alternative strategy.

Test Code Evolution: Our results also show that fine-tuning code language models (CLMs) for test code evolution is promising, especially for simple and common changes. These models can automate test maintenance tasks efficiently and accurately under moderate complexity.

- As the complexity of test changes increases, the reliability of CLMs decreases. However, they can still be useful when applied selectively. We propose using a test repair trustworthiness predictor, such as the one we developed, to determine when model-generated changes are likely to be correct. This enables practitioners to delegate a subset of repairs to the model while reserving more complex edits for expert developers.

- Furthermore, fine-tuned models trained on data from other projects can still generalize well, making them useful even when historical data from the target project is limited or unavailable. This promotes reuse and lowers the barrier for adopting AI-assisted test maintenance.

4.2.2 Model Scale Trade-offs: Fine-tuned vs. Foundation Models

Discussing the trade-offs between using small, task-specific fine-tuned models and large general-purpose foundation models is important and valuable for future research. Fine-tuning smaller models comes with significant costs——including the need for high-quality labeled data, time-consuming data preparation, and the resource-intensive fine-tuning process. These models also face context window limitations. However, once trained, they offer high effectiveness tailored to the target domain and benefit from improved data privacy, especially for organizations with sensitive codebases.

Conversely, large foundation models offer much larger context windows and remove the need for fine-tuning, which lowers the upfront cost. Yet, their general-purpose nature often leads to lower task-specific performance, particularly in specialized domains like test repair. Additionally, while API usage is simpler, it still incurs ongoing cost and introduces potential data privacy concerns.

Overall, while our current work focuses on fine-tuned models, we recognize the potential and practicality of large foundation models and see them as a promising direction for future research. As an additional effort, we experimented with GPT-3.5 Turbo—a significantly larger foundation model accessed via API—compared to our fine-tuned CodeT5+ 770M model. We designed four different prompt variants, adjusting elements such as instructions, special tokens used for representing breakage lines and hunk representation, and few-shot examples. For consistency, we used the same test code and repair context previously collected for TARGET. We then evaluated the prompts on a 10% stratified random sample of our test set and compared the outcomes against TARGET's best-performing configuration.

Our preliminary results showed that, despite the scale and generality of GPT-3.5 Turbo, none of the prompt designs outperformed TARGET. These findings are not conclusive and suggest that extensive further exploration is needed to fully leverage large foundation models. Potential directions include making better use of large context windows by incorporating additional contextual information (e.g., broader code changes, unchanged but relevant code), iterative repair generation and execution loops, and even fine-tuning the foundation models if supported via APIs.

4.2.3 Generalizability Beyond Java and Travis CI

Limiting the scope of this thesis to Java-based projects and specifically to Travis CI environments may reduce the immediate applicability of some findings to broader ecosystems. However, these limitations are largely technical and implementation-specific, rather than conceptual or methodological.

The feature groups introduced in Chapter 2 are designed with generalizability in mind. These include features derived from test execution records—such as test verdicts and execution times—which are universally present across CI environments regardless of the programming language or CI tool used. Similarly, features related to development activity, code coverage, and code complexity are based on widely accepted concepts in software engineering and can be collected or adapted in most development contexts. While certain language-specific metrics—such as object-oriented code metrics—might require adaptation or redefinition for non-Java languages, and while some environments might require the development of alternative tooling (e.g., for coverage analysis), the overall taxonomy of feature groups and the ML-based TCP techniques we propose are broadly applicable. With minor adjustments, our methods can be extended to a variety of languages and CI environments.

Regarding the work in Chapter 3, we emphasize that the core contributions——such as relevant hunk selection and representation and fine-tuning of Code Language Models (CLMs)——are fundamentally language-agnostic. The CLMs we employ have been pre-trained on a diverse corpus of programming languages, including Java and Python. Moreover, our most effective hunk selection strategy leverages the textual characteristics of code changes rather than language-specific syntax, further supporting generalizability.

Nevertheless, the core techniques and methodologies introduced are applicable well beyond the Java/Travis CI context and can serve as a foundation for further adaptation and development in more diverse CI/CD environments.

4.3 Future Work

This section discusses and outlines several directions for future work. We first reflect on TO_1 (ML-based TCP), and then we discuss the insights and potential extensions for TO_2 (automated test repair).

4.3.1 Test Case Prioritization in CI Contexts

Our proposed TCP study, which leverages a comprehensive data-driven feature model, has demonstrated significant improvements in fault detection cost-effectiveness. However, several avenues remain open for further exploration:

1. **Granularity of Coverage Measurement:** The current approach measures test case coverage at the file level. Future work should systematically investigate the trade-offs between finer-grained (e.g., method-level) coverage measurement, scalability, and overall TCP effectiveness. Such analysis would reveal whether a more granular approach can further enhance fault detection while keeping data collection costs manageable.

2. **Incorporation of Human Factors:** Our study focused on feature groups based on technical metrics, leaving human-oriented feature groups that capture the dynamics of collaborative development for future work. Analyzing these human factors could offer valuable insights to refine development practices and improve the effectiveness of TCP.

3. **Platform-Specific Considerations:** In many CI environments, the same test cases are executed across different configurations and platforms. Although we mitigated the potential negative impact of duplicated execution records by focusing on the configuration with the highest number of test cases, future research could integrate platform-specific features.

4. **Variation in TCP Effectiveness Across Subjects:** Our analysis showed notable variations in the $APFD_C$ metric across subjects. While initial correlation analysis did not show strong or moderate correlations with subject characteristics, further investigation is needed. Future research should examine factors beyond those analyzed here to uncover relationships that could suggest adaptive TCP strategies.

5. **Extension to Diverse CI Tools and Languages:** The current data collection and analysis framework is tailored to specific CI environments and programming languages. Extending the tool to support other CI systems (e.g., GitHub Actions) and programming languages could enhance the generalizability of our findings and provide broader applicability for TCP techniques.

4.3.2 Automated Test Repair Using Language Models

The automated test repair contribution is a significant step toward addressing the problem. Nonetheless, several research challenges and opportunities remain:

1. **Integrated Breakage Localization and Repair:** A fully automated test repair pipeline must combine both breakage localization and repair. While this study focuses on the repair task, we acknowledge that breakage localization (broadly referred to as fault localization) is a complex and ongoing research challenge. Our tool, TARGET, contributes to the broader vision by addressing the repair component. Future work should explore full integration with CI/CD pipelines and IDEs.

2. **Addressing Black-Box Scenarios:** In some settings, access to different versions of the SUT is limited. Future studies could explore the adaptation of context selection and prioritization techniques proposed here to black-box scenarios by leveraging test execution logs and traces as alternative sources of information.

3. **Support for Multi-Hunk Repairs:** Our current approach focuses on single-hunk test repairs. Extending the methodology to handle multiple hunks—either iteratively or in a single-step process—requires further refinement of the technique and the collection of relevant multi-hunk data. Such an extension would be valuable for addressing more complex test repair scenarios.

4. **Support for Cross-Boundary Test Repairs:** Our current approach focuses on test breakages and repairs within individual test methods. However, test repairs can span beyond test method boundaries, such as in setup/teardown methods, helper functions, utility classes, or abstract superclasses from which tests inherit. In this work, we addressed repairing test methods directly, but comprehensive test repair should handle breakages wherever they occur in the test code. Such an extension would be valuable for maintaining complex test code with complex structure and shared testing practices.

5. **Enriching Context through Repository Understanding:** While changes in the SUT code are the primary trigger for test breakages, additional context from unchanged portions of the codebase might be beneficial. Future research could investigate methods for efficiently identifying and incorporating relevant project-level context into the repair process, despite the challenge posed by the large volume of unchanged code.

6. **Moving Beyond the Plausible Repair Accuracy Metric:** While the plausible repair accuracy metric identifies repairs that pass, it does not guarantee semantic correctness. Additionally, although human-conducted, semantic-aware evaluations are ideal, they are expensive. Future research should prioritize the development of scalable, semantic-aware evaluation metrics to better assess repair quality. We propose two key future directions for improving semantic validation: (1) using large language models (LLMs) to compare non-exact-match repairs with the ground truth, leveraging their advanced code understanding capabilities, and (2) designing new evaluation metrics that better capture test intent. For instance, one such metric could compare the coverage of the original, unbroken test case with the repaired version, identifying substantial divergences that may indicate a loss of semantic logic.

7. **Adapting to other Programming Languages:** Although the primary focus in this thesis has been on Java (particularly for JUnit tests), the underlying principles of hunk selection, representation, and fine-tuning of CLMs are language-agnostic. Future work should explore the adaptation of the approach to other programming languages, supported by the collection of language-specific repair data and the selection of suitably pre-trained CLMs.

8. **Iterative and Agentic Repair Approaches:** Future research could explore iterative repair strategies, such as multi-step or agentic approaches that follow a repair-execute-feedback loop. Additionally, shifting from fine-tuning to leveraging larger commercial foundation models with expanded context sizes might further improve repair effectiveness and adaptability.

9. **Data Augmentation and Synthetic Repair Samples:** To better handle the variation in repair complexity, enhancing the fine-tuning process with data augmentation techniques and synthetic repair samples could balance the dataset. This augmentation would likely lead to improved performance across a wider range of test repair scenarios.

References

[1] "Digital research alliance of canada," https://alliancecan.ca, 2023. [Online]. Available: https://alliancecan.ca

[2] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[3] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2102–2139, 2019.

[4] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020.

[5] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of Java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 821–830.

[6] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 101–110.

[7] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *arXiv e-prints*, pp. arXiv–2106, 2021.

[8] J. Imtiaz, S. Sherin, M. U. Khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," *Information and Software Technology*, vol. 113, pp. 1–19, 2019.

[9] "TCP-CI," 2024. [Online]. Available: https://github.com/Ahmadreza-SY/TCP-CI

[10] "TaRGet," 2024. [Online]. Available: https://github.com/Ahmadreza-SY/TaRGet

[11] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, "TaRBench: A Comprehensive Benchmark for Automated Test Case Repair," 2025. [Online]. Available: https://doi.org/10.6084/m9.figshare.25008893

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

[13] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

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

[15] J. A. P. Lima, W. D. Mendonça, S. R. Vergilio, and W. K. Assunção, "Learning-based prioritization of test cases in continuous integration of highly-configurable software," in *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, 2020, pp. 1–11.

[16] J. A. do Prado Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, 2020.

[17] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.

[18] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, pp. 1–1, Apr 2021.

[19] A. Gepperth and B. Hammer, "Incremental learning algorithms and applications," in *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

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

[21] Mockus and Votta, "Identifying reasons for software changes using historic databases," in *Proceedings 2000 International Conference on Software Maintenance*, 2000, pp. 120–130.

[22] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 30–39.

[23] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of*

the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, ser. PROMISE. New York, NY, USA: Association for Computing Machinery, 2017, p. 97–106. [Online]. Available: https://doi-org.proxy.bib.uottawa.ca/10.1145/3127005.3127016

[24] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[26] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 19–27.

[27] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[28] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[29] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing and Management*, vol. 24, no. 5, pp. 513–523, 1988.

[30] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the impact of programming languages on code quality: A reproduction study," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 4, Oct. 2019. [Online]. Available: https://doi-org.proxy.bib.uottawa.ca/10.1145/3340571

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi-org.proxy.bib.uottawa.ca/10.1145/2939672.2939785

[32] P. McCullagh and J. A. Nelder, *Generalized linear models*. CRC press, 1989, vol. 37.

[33] M. Friedman, "The use of ranks to avoid the assumption of normality implicit in the analysis of variance," *Journal of the American Statistical Association*, vol. 32, no. 200, pp. 675–701, 1937.

[34] P. B. Nemenyi, "Distribution-free multiple comparisons," Ph.D. dissertation, Princeton University, 1963. [Online]. Available: https://www.proquest.com/docview/302256074

[35] Scientific Toolworks, Inc., "Understand™," https://www.scitools.com/, 2020, retrieved October 14, 2020.

[36] T. Mattis, P. Rein, F. Dürsch, and R. Hirschfeld, "Rtptorrent: An open-source dataset for evaluating regression test prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 385–396. [Online]. Available: https://doi-org.proxy.bib.uottawa.ca/10.1145/3379597.3387458

[37] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[38] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 975–980.

[39] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 491–504.

[40] M. di Biase, A. Rastogi, M. Bruntink, and A. van Deursen, "The delta maintainability model: Measuring maintainability of fine-grained code changes," in *Proceedings of the Second International Conference on Technical Debt*, ser. TechDebt '19. IEEE Press, 2019, p. 113–122. [Online]. Available: https://doi.org/10.1109/TechDebt.2019.00030

[41] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[42] Wikipedia contributors, "68–95–99.7 rule — Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule

[43] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* IEEE, 2001, pp. 329–338.

[44] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, 1999, pp. 179–188.

[45] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 324–336. [Online]. Available: https://doi.org/10.1145/3395363.3397383

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

[47] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *The Annals of Statistics*, vol. 29, no. 5, pp. 1189 – 1232, 2001. [Online]. Available: https://doi.org/10.1214/aos/1013203451

[48] Q. Wu, C. J. C. Burges, K. M. Svore, and J. Gao, "Adapting boosting for information retrieval measures," *Information Retrieval*, vol. 13, no. 3, pp. 254–270, Jun 2010. [Online]. Available: https://doi.org/10.1007/s10791-009-9112-1

[49] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *Journal of machine learning research*, vol. 4, no. Nov, pp. 933–969, 2003.

[50] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, "Learning to rank: From pairwise approach to listwise approach," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 129–136. [Online]. Available: https://doi.org/10.1145/1273496.1273513

[51] D. Metzler and W. Bruce Croft, "Linear feature-based models for information retrieval," *Information Retrieval*, vol. 10, no. 3, pp. 257–274, Jun 2007. [Online]. Available: https://doi.org/10.1007/s10791-006-9019-z

[52] "Ranklib," https://sourceforge.net/p/lemur/wiki/RankLib/, 2020.

[53] J. Torres-Jimenez and I. Izquierdo-Marquez, "Survey of covering arrays," in *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013, pp. 20–27.

[54] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[55] R. Somasundaram and R. Nedunchezhian, "Evaluation of three simple imputation methods for enhancing preprocessing of data with missing values," *International Journal of Computer Applications*, vol. 21, 05 2011.

[56] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

[57] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[58] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[59] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[60] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1. IEEE, 2006, pp. 411–420.

[61] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[62] J. A. P. Lima and S. R. Vergilio, "Multi-armed bandit test case prioritization in continuous integration environments: A trade-off analysis," in *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, 2020, pp. 21–30.

[63] L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, "Xcs as a reinforcement learning approach to automatic test case prioritization," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 1798–1806.

[64] T. Shi, L. Xiao, and K. Wu, "Reinforcement learning based test case prioritization for enhancing the security of software," in *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2020, pp. 663–672.

[65] R. Almaghairbe and M. Roper, "Separating passing and failing test executions by clustering anomalies," *Software Quality Journal*, vol. 25, no. 3, pp. 803–840, 2017.

[66] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study." in *ICSM*, vol. 11, 2011, pp. 382–391.

[67] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 1–10.

[68] P. Kandil, S. Moussa, and N. Badr, "Cluster-based test cases prioritization and selection technique for agile regression testing," *Journal of Software: Evolution and Process*, vol. 29, no. 6, p. e1794, 2017.

[69] Z. Khalid and U. Qamar, "Weight and cluster based test case prioritization technique," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 2019, pp. 1013–1022.

[70] Y. Wang, Z. Chen, Y. Feng, B. Luo, and Y. Yang, "Using weighted attributes to improve cluster test selection," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 138–146.

[71] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 201–212.

[72] M. Hasnain, M. F. Pasha, C. H. Lim, and I. Ghan, "Recurrent neural network for web services performance forecasting, ranking and regression testing," in *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2019, pp. 96–105.

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

[74] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016, pp. 361–368.

[75] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, 2020.

[76] S. Mirarab and L. Tahvildari, "An empirical study on bayesian network-based approach for test case prioritization," in *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 278–287.

[77] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 2–11.

[78] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, "An improvement to test case failure prediction in the context of test case prioritization," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2018, pp. 80–89.

[79] M. M. Sharma and A. Agrawal, "Test case design and test case prioritization using machine learning," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 1, pp. 2742–2748, 2019.

[80] A. Singh, R. K. Bhatia, and A. Singhrova, "Machine learning based test case prioritization in object oriented testing," *International Journal of Recent Technology and Engineering*, vol. 8, no. 3, pp. 700–707, 2019.

[81] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 123–133.

[82] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "A comparative study of vectorization-based static test case prioritization methods," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 80–88.

[83] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[84] A. Saboor Yaraghi, D. Holden, N. Kahani, and L. Briand, "Automated test case repair using language models," *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 1104–1133, 2025.

[85] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[86] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *37th IEEE International Conference on Software Engineering*. IEEE, 2015, pp. 483–493.

[87] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: Empirical observations," *Advances in Software Engineering*, vol. 2010, 2010.

[88] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010, pp. 207–218.

[89] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of Travis CI," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 647–658.

[90] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 433–444.

[91] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatic test case evolution," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 386–411, 2014.

[92] Y. Xu, B. Huang, G. Wu, and M. Yuan, "Using genetic algorithms to repair junit test cases," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 1. IEEE, 2014, pp. 287–294.

[93] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 217–227.

[94] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393634

[95] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 602–628.

[96] Y. Gao, H. Liu, X. Fan, Z. Niu, and B. Nyirongo, "Analyzing refactorings' impact on regression test cases," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, 2015, pp. 222–231.

[97] E. J. Rapos and J. R. Cordy, "Examining the co-evolution relationship between simulink models and their test cases," in *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, ser. MiSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 34–40. [Online]. Available: https://doi.org/10.1145/2896982.2896983

[98] H. A. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. V. Nguyen, "Interaction-based tracking of program entities for test case evolution," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 433–443.

[99] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 910–929, 2020.

[100] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *International Conference on Software Testing, Verification and Validation*, 2010, pp. 245–254.

[101] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2023.

[102] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[103] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[104] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[105] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," 2023.

[106] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.

[107] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[108] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," *arXiv preprint arXiv:2307.14991*, 2023.

[109] C. Zhou, Q. Li, C. Li, J. Yu, Y. Liu, G. Wang, K. Zhang, C. Ji, Q. Yan, L. He *et al.*, "A comprehensive survey on pretrained foundation models: A history from bert to chatgpt," *arXiv preprint arXiv:2302.09419*, 2023.

[110] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.

[111] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proceedings of the 2003 International Conference on Information Integration on the Web*, ser. IIWEB'03. AAAI Press, 2003, p. 73–78.

[112] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[113] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the*

31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[114] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. [Online]. Available: https://openreview.net/forum?id=6lE4dQXaUcb

[115] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[116] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[117] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020.

[118] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.

[119] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," 2023. [Online]. Available: https://arxiv.org/abs/2303.18184

[120] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[121] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 602–614.

[122] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[123] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[124] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," *arXiv preprint arXiv:2302.05020*, 2023.

[125] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[126] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[127] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[128] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[129] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[130] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[131] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2019.

[132] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow," 2021. [Online]. Available: http://github.com/eleutherai/gpt-neo

[133] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[134] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis,

S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[135] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *arXiv preprint arXiv:2202.13169*, 2022.

[136] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "Identify and update test cases when production code changes: A transformer-based approach," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1111–1122.

[137] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 790–823, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09772-z

[138] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, Sep. 2014, pp. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[139] ——, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering*, 2014, pp. 313–324. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01054552/file/main.pdf

[140] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[141] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623337

[142] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556926

[143] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research*, vol. 15, no. 90, pp. 3133–3181, 2014. [Online]. Available: http://jmlr.org/papers/v15/delgado14a.html

[144] Y. Kashiwa, K. Shimizu, B. Lin, G. Bavota, M. Lanza, Y. Kamei, and N. Ubayashi, "Does refactoring break tests and to what extent?" in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 171–182.

[145] N. A. Nagy and R. Abdalkareem, "On the co-occurrence of refactoring of test and source code," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 122–126. [Online]. Available: https://doi.org/10.1145/3524842.3528529

[146] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Hillsdale, N.J.: L. Erlbaum Associates Hillsdale, N.J., 1988.

[147] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[148] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Mangrulkar, M. Sun, and B. Bossan, "Accelerate: Training and inference at scale made simple, efficient and adaptable." https://github.com/huggingface/accelerate, 2022.

[149] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[150] R. Just, "defects4j," https://github.com/rjust/defects4j, 2023.

[151] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.

[152] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[153] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[154] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *ACM SIGPLAN International Symposium on Machine Programming*, 2021, pp. 1–8.

[155] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[156] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," *arXiv preprint arXiv:2302.01857*, 2023.

[157] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv e-prints*, pp. arXiv–2203, 2022.

[158] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[159] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 432–441.

[160] M. di Biase, A. Rastogi, M. Bruntink, and A. van Deursen, "The delta maintainability model: Measuring maintainability of fine-grained code changes," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 113–122.

[161] D. Di Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia, "On the role of developer's scattered changes in bug prediction," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 241–250.

APPENDICES

Appendix A

Complementary Details for TO_1

A.1 Metric Definitions

In this section, we provide definitions of the metrics used in our test case feature model. We use the standard definitions from the relevant sources [35, 56, 159–161] and highlight and justify our definitions when they deviate from the standard ones. The metrics are defined based on a source code file. However, similar definitions can be provided based on a class or a method. In Table A.1 and Table A.2, we also provide summarized descriptions of the feature groups and the metrics, respectively.

Group(s)	Description
TES_COM, **TES_PRO**, **TES_CHN**	Features characterizing the source code of test cases that are calculated based on code metrics (complexity, process, and change metrics).
REC	Features characterizing test cases based on their execution records (time and verdict) in builds.
F_COV	Features capturing the file coverage of test cases in changed and impacted files.
COD_COV_COM, **COD_COV_PRO**, **COD_COV_CHN**	Features characterizing the source code of the changed and impacted files covered by test cases, which are calculated based on code metrics (complexity, process, and change metrics).
DET_COV	Features based on the history of faults present in source files and detected by test cases.

Table A.1: Test case feature groups.

A.1.1 Complexity Metrics

A.1.1.1 Program Size

- **CntDeclFunction:** Number of functions in a file.
- **LoC:** Number of all lines in a file. [aka NL]

Group	Metrics	Description
Complexity (COM)	**Program Size**: CntDeclFunction, LoC, LoCBlank, LoCCode, LoCCodeDecl, LoCCodeExe, LoCComment, CntStmt, CntStmtDecl, CntStmtExe, RatioCommentToCode **McCabe's Cyclomatic Complexity**: MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxNesting, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential **Object-oriented Metrics**: CntDeclClass, CntDeclClassMethod, CntDeclClassVariable, CntDeclExecutableUnit, CntDeclInstanceMethod, CntDeclInstanceVariable, CntDeclMethod, CntDeclMethodDefault, CntDeclMethodPrivate, CntDeclMethodProtected, CntDeclMethodPublic	This metric group includes: 1) Size metrics related to the number of lines of code, declarations, statements, and files. 2) Complexity metrics related to the control flow graph of methods. 3) Metrics based on object-oriented constructs.
Process (PRO)	CommitCnt, DistinctDevCnt, OwnerContributedLines, MinorContributorCnt, OwnerExperience, AllCommittersExperience	Metrics based on the history of the development process of files in the SUT.
Change (CHN)	LinesAdded, LinesDeleted, ChangedCodeScattering, DMMUnitComplexity, DMMUnitInterfacing, DMMUnitSize	Metrics focused on changes of the last version of the system under test.

Table A.2: Complexity, process, and change metrics of source code.

- **LoCBlank:** Number of blank lines in a file. [aka BLoC]
- **LoCCode:** Number of lines containing source code in a file. [aka LoC]
- **LoCCodeDecl:** Number of lines containing declarative source code in a file.
- **LoCCodeExe:** Number of lines containing executable source code in a file.
- **LoCComment:** Number of lines containing comments in a file. [aka CLoC]
- **CntStmt:** Number of statements in a file.
- **CntStmtDecl:** Number of declarative statements in a file.
- **CntStmtExe:** Number of executable statements in a file.
- **RatioCommentToCode:** Ratio of comment lines to code lines in a file.

A.1.1.2 McCabe's Cyclomatic Complexity

- **MaxCyclomatic:** Maximum cyclomatic complexity[1] of all nested functions or methods in a file.
- **MaxCyclomaticModified:** Maximum modified cyclomatic complexity of nested functions or methods in a file.

[1] SciTools: Understanding McCabe Cyclomatic Complexity

- **MaxCyclomaticStrict:** Maximum strict cyclomatic complexity of nested functions or methods in a file.
- **MaxEssential:** Maximum essential complexity of all nested functions or methods in a file.
- **MaxNesting:** Maximum nesting level of control constructs in a file.
- **SumCyclomatic:** Sum of cyclomatic complexity of all nested functions or methods in a file. [aka WMC]
- **SumCyclomaticModified:** Sum of modified cyclomatic complexity of all nested functions or methods in a file.
- **SumCyclomaticStrict:** Sum of strict cyclomatic complexity of all nested functions or methods in a file.
- **SumEssential:** Sum of essential complexity of all nested functions or methods in a file.

A.1.1.3 Object-oriented Metrics

- **CntDeclClass:** Number of classes in a file.
- **CntDeclClassMethod:** Number of class methods in a file.
- **CntDeclClassVariable:** Number of class variables in a file.
- **CntDeclExecutableUnit:** Number of program units with executable code in a file.
- **CntDeclInstanceMethod:** Number of instance methods in a file. [aka NIM]
- **CntDeclInstanceVariable:** Number of instance variables in a file. [aka NIV]
- **CntDeclMethod:** Number of local methods in a file.
- **CntDeclMethodDefault:** Number of local default methods in a file.
- **CntDeclMethodPrivate:** Number of local private methods in a file. [aka NPM]
- **CntDeclMethodProtected:** Number of local protected methods in a file.
- **CntDeclMethodPublic:** Number of local public methods in a file. [aka NPRM]

A.1.2 Process Metrics

- **Commit Count:** The number of commits made to a file.
- **Distinct Dev Cnt:** The cumulative number of distinct developers who contributed to this file up to this build/release.
- **Owner's Contributed Lines:** The percentage of lines authored by the highest contributor of a file (the contributor with the most authored lines). Both added and deleted lines are counted as authored lines.
- **Minor Contributor Cnt:** The number of contributors who authored less than 5% of the code in the file.
- **Owner's Experience:** Measures the experience of the highest contributor of the file using the percent of lines he/she authored in the project up to this build/release.
- **All Committer's Experience:** The geometric mean of experience of all the developers of the file.

A.1.3 Change Metrics

- **Lines Added and Deleted:** The added and deleted lines in the file in this build.
- **Added and Deleted Change Scattering:** Given the added and deleted changes of a source file in a build, this metric measures the scattering of changes in the file. Assuming $CH = \{ch_1, ch_2, ..., ch_n\}$ is the set of added/deleted code chunks in file f, we define Change Scattering (CS) as follows.

$$CS(f) = \frac{|CH|}{\binom{|CH|}{2}} \times \sum_{\forall ch_i, ch_j \in CH} \text{dist}(ch_i, ch_j)$$

where $\text{dist}(ch_i, ch_j)$ computes the distance between the two code chunks ch_i and ch_j and is defined as follows.

$$\text{dist}(ch_i, ch_j) = |line(ch_i) - line(ch_j)|$$

where $\text{line}(ch_i)$ computes line number of the beginning of code chunk ch_i. If there is only one code chunk in the changed file f (i.e., $|CH| = 1$), then $CS(f) = 0$.

The multiplication factor at the beginning of CS's formula has two objectives: (i) normalizing the distances between the code chunks by the number of pairs of code chunks modified in a build (reflected in the denominator), (ii) assigning a higher scattering to files including a higher number of changed code chunks (reflected in the numerator).

This metric is inspired by previous work [161] which proposes structural scattering for computing a developer's change scattering in a given period of time.

- **Delta Maintainability Metric (DMM)**: In one sentence, DMM is the proportion of low-risk change in a commit. The resulting value ranges from 0.0 (all changes are risky) to 1.0 (all changes have a low risk). DMM was originally calculated at the method level, and to calculate it at the commit level, we use

$$DMM = \frac{lr\_chn}{lr\_chn + hr\_chn}$$

  where lr_chn and hr_chn stand for the number of high-risk and low-risk changes, respectively. A low-risk change is defined as adding low-risk code or removing high-risk code, whereas a high-risk change is defined as adding high-risk code or removing low-risk code.

  The DMM can be used on arbitrary properties that can be determined at the method (unit) level. The PyDriller [56] OS-DMM implementation supports three properties:

  - **DMMUnitComplexity:** Cyclomatic complexity of method; low risk threshold 5.
  - **DMMUnitInterfacing:** The number of method's parameters: low risk threshold 2.
  - **DMMUnitSize:** Method size in lines of code; low risk threshold 15.

  For instance, removing lines of code from a 20-line-method would be a low-risk change. However, adding lines of code to a method with three parameters counts as a high-risk change.

A.2 Usage Frequencies of All Features

As a supplement to research question 2.4, which is discussed and analyzed in Section 2.4.4.2 of the thesis, Table A.3 shows the average usage frequency for all individual features in this study across all trained RF ranking models for all builds.

Table A.3: Usage frequency of individual features in models that were trained on the full feature set. The *Avg. Freq.* column shows the average usage frequency of a feature across all trained RF ranking models for all builds.

Feature Group	Feature	Avg. Freq.	Feature Group	Feature	Avg. Freq.
REC	Age	17034	COD_COV_COM	CountDeclClass	309
TES_PRO	OwnersExperience	10618	COD_COV_COM	CountLineCodeDecl	302
TES_PRO	AllCommitersExperience	7643	COD_COV_COM	CountLineBlank	293
REC	TotalMaxExeTime	5409	COD_COV_COM	CountDeclMethodPublic	292
REC	LastExeTime	4095	COD_COV_COM	CountStmtDecl	291
TES_PRO	OwnersContribution	3997	COD_COV_COM	CountDeclFunction	287
TES_PRO	CommitCount	3986	COD_COV_COM	CountLine	277
REC	TotalAvgExeTime	3977	COD_COV_COM	CountDeclMethod	273
REC	RecentAvgExeTime	3850	COD_COV_COM	CountLineCodeExe	272
REC	RecentMaxExeTime	3741	REC	RecentExcRate	270
TES_COM	RatioCommentToCode	3695	COD_COV_COM	CountDeclExecutableUnit	261
TES_COM	CountStmtDecl	3383	COV	ChnScoreSum	261
TES_COM	CountLineCodeDecl	3342	COD_COV_COM	SumEssential	255
TES_COM	CountLineBlank	3149	COD_COV_COM	CountLineCode	248
TES_COM	CountStmtExe	2825	COD_COV_COM	CountStmtExe	248
TES_COM	CountLine	2792	COD_COV_COM	CountStmt	248
TES_COM	CountLineCodeExe	2688	COD_COV_PRO	AllCommitersExperience	243
TES_COM	CountLineComment	2668	COD_COV_COM	SumCyclomaticStrict	238
TES_COM	CountStmt	2658	TES_COM	CountDeclMethodDefault	238
TES_COM	CountLineCode	2453	COD_COV_COM	SumCyclomatic	230
TES_COM	CountDeclMethodPublic	1529	REC	RecentAssertRate	230
TES_COM	CountDeclInstanceVariable	1489	COD_COV_COM	SumCyclomaticModified	228
TES_PRO	DistinctDevCount	1374	COD_COV_PRO	OwnersExperience	228
TES_COM	SumCyclomaticStrict	1315	COD_COV_CHN	LinesAdded	224
REC	LastTransitionAge	1284	COD_COV_COM	RatioCommentToCode	207
TES_COM	SumCyclomatic	1281	COD_COV_PRO	OwnersContribution	196
TES_COM	SumCyclomaticModified	1280	COD_COV_CHN	LinesDeleted	192
TES_COM	CountDeclInstanceMethod	1273	COD_COV_CHN	AddedChangeScattering	168
REC	LastFailureAge	1268	TES_COM	MaxEssential	157
REC	TotalTransitionRate	1141	COD_COV_CHN	DeletedChangeScattering	150
TES_PRO	MinorContributorCount	1134	COD_COV_COM	CountDeclClassVariable	131
TES_COM	CountDeclMethod	1068	COD_COV_COM	MaxNesting	130
REC	TotalFailRate	1058	COD_COV_COM	CountDeclInstanceVariable	128
TES_COM	SumEssential	1055	DET_COV	Faults	126
TES_COM	CountDeclExecutableUnit	1039	COD_COV_PRO	CommitCount	125
TES_COM	CountDeclFunction	1035	COD_COV_COM	MaxEssential	124
TES_COM	CountDeclClassVariable	995	COD_COV_CHN	DMMSize	119
COD_COV_PRO	OwnersExperience	971	COD_COV_COM	CountLineComment	119
COD_COV_PRO	AllCommitersExperience	908	COD_COV_COM	CountDeclClassMethod	119
TES_COM	MaxCyclomaticStrict	845	COD_COV_COM	MaxCyclomaticStrict	115
REC	TotalExcRate	843	COV	ChnCount	115
COD_COV_COM	RatioCommentToCode	797	COD_COV_COM	MaxCyclomaticModified	114
TES_COM	MaxCyclomatic	784	COD_COV_COM	MaxCyclomatic	114
TES_COM	CountDeclMethodPrivate	777	COD_COV_COM	CountDeclMethodPrivate	112
TES_COM	MaxCyclomaticModified	769	COD_COV_COM	CountDeclInstanceMethod	108
COV	ImpScoreSum	761	TES_CHN	LinesAdded	108
TES_COM	CountDeclClass	749	COD_COV_CHN	DMMComplexity	107
TES_COM	MaxNesting	740	COD_COV_PRO	DistinctDevCount	105
TES_COM	CountDeclClassMethod	661	COD_COV_PRO	MinorContributorCount	105
REC	MaxTestFileFailRate	645	COD_COV_COM	CountDeclClass	101
REC	TotalAssertRate	612	COD_COV_COM	CountDeclMethodPublic	100
REC	MaxTestFileTransitionRate	597	TES_CHN	LinesDeleted	100
COD_COV_PRO	OwnersContribution	597	COD_COV_COM	CountDeclMethodDefault	96
COV	ImpCount	507	COD_COV_CHN	DMMInterfacing	94
COD_COV_PRO	CommitCount	458	COD_COV_COM	CountDeclMethodProtected	94
COD_COV_COM	CountDeclClassVariable	455	COD_COV_COM	CountLineCodeExe	93
COD_COV_COM	CountDeclMethodDefault	455	COD_COV_COM	CountLineCodeDecl	92
COD_COV_COM	CountDeclClassMethod	451	COD_COV_COM	CountLine	90
REC	LastVerdict	443	COD_COV_COM	SumEssential	90
REC	RecentFailRate	441	COD_COV_COM	CountStmtExe	88
COD_COV_COM	CountDeclInstanceVariable	437	COD_COV_COM	CountLineBlank	88
COD_COV_COM	CountLineComment	418	COD_COV_COM	CountLineCode	87
DET_COV	Faults	403	COD_COV_COM	SumCyclomaticStrict	87
COD_COV_PRO	DistinctDevCount	389	COD_COV_COM	CountDeclFunction	87
COD_COV_PRO	MinorContributorCount	386	COD_COV_COM	CountDeclExecutableUnit	86
TES_COM	CountDeclMethodProtected	360	COD_COV_COM	CountStmtDecl	86
COD_COV_COM	CountDeclInstanceMethod	359	COD_COV_COM	CountStmt	86
REC	RecentTransitionRate	355	COD_COV_COM	SumCyclomatic	85
COD_COV_COM	MaxEssential	352	COD_COV_COM	CountDeclMethod	85
COD_COV_COM	CountDeclMethodPrivate	351	TES_CHN	AddedChangeScattering	85
COD_COV_COM	MaxCyclomaticStrict	332	COD_COV_COM	SumCyclomaticModified	84
COD_COV_COM	MaxNesting	331	TES_CHN	DeletedChangeScattering	78
COD_COV_COM	MaxCyclomatic	325	TES_CHN	DMMSize	70
COD_COV_COM	CountDeclMethodProtected	317	TES_CHN	DMMInterfacing	47
COD_COV_COM	MaxCyclomaticModified	309	TES_CHN	DMMComplexity	46

Appendix B

Complementary Details for TO_2

B.1 Examples of Test Case Repairs Generated by TaRGET on TaRBench: Successes and Failures

In this section, we present eight detailed examples—four successful repairs and four failed repairs—to complement the qualitative analysis of TaRGET discussed in Section 3.4.5.3.

Successful and failed repairs are determined using two well-established evaluation metrics: Exact Match Accuracy (EM) and Plausible Repair Accuracy (PR). These metrics are clearly defined, justified, and consistently applied throughout our study. To reiterate, a repair is a successful exact match repair if it is identical to the ground truth repair. A repair is a successful plausible repair if it compiles and runs to completion without any failures. In selecting the examples, we followed these criteria. Additionally, we ensured diversity in the repair characteristics and avoided plausible examples that did not uphold the semantics and intent of the test case (see limitations of PR outlined in Section 3.4.10).

B.1.1 Failure Examples

B.1.1.1 Failure Example 1

The analysis of the failure case in Figure B.1 reveals several critical insights. According to the ground truth in Figure B.1a, no changes were initially expected in the output based on the provided input. However, following the repair, the system expected the addition of a new annotation, *@CanIgnoreReturnValue* (*@CIRV*), which was expected to appear above a method declaration, along with its corresponding import.

The predictions generated by TaRGET, as seen in Figure B.1b, indicate that the model did not attempt to change the expected output lines and continued to expect no changes. This misunderstanding is evident in the method calls of the model's predictions, where the model, in one instance, even removed a method call. Although the relevant repair context, shown in Figure B.1c in line 6, hinted at a behavioral change related to *@CIRV*, this context alone was insufficient for the model to correctly perform the repair.

A deeper analysis shows that the model lacked critical contextual information. Specifically, it needed details about the system under test (SUT) and the test code that were not provided. Firstly, the model required awareness of the API associated with the *helper* variable and the presence of the *addOutputLines* method. Secondly, it needed a more comprehensive understanding of the SUT's logic, including how annotations like *@CIRV* are applied.

In conclusion, this failure can be attributed to missing contextual information. To improve repair performance in such cases, a strategy that incorporates repository-level repair context-—considering both unchanged and changed parts of the codebase-—would likely perform better.

Figure B.1: **Failure Example 1** – An example of missing essential repository-level unchanged contextual information from the model's input.
TaRBench ID: *google/error-prone:25*
GitHub Commit Reference: *428aa98972*

```
@Test
public void testBuilder_abstractClass() {
  helper
    .addInputLines(
      "Builder.java",
      "package com.google.frobber;",
      "public abstract class Builder {",
      " public abstract Builder setName(String name);",
      " public abstract Builder clone();",
      " public abstract Builder copy();",
      "}")
    - .expectUnchanged()
    + .addOutputLines(
    + "Builder.java",
    + "package com.google.frobber;",
    + "import com.google.errorprone.annotations.CanIgnoreReturnValue;",
    + "public abstract class Builder {",
    + " @CanIgnoreReturnValue",
    + " public abstract Builder setName(String name);",
    + " public abstract Builder clone();",
    + " public abstract Builder copy();",
    + "}")
    .doTest();
}
```

(a) **Ground Truth** Repair

```
1  - .expectUnchanged ( )
2  + .expectNoChanges ( )
3
4  ----------------------------------
5
6  - .expectUnchanged ( )
7  + // Deleted
8
9  ----------------------------------
10
11 - .expectUnchanged ( )
12 + .expectNoChange ( )
13
14 ----------------------------------
15
16 - .expectUnchanged ( )
17 + .expectNoChanged ( )
```

(b) Selected Top **Repair Candidates** Generated by TARGET

```
1  + // skip @AutoValue and @AutoBuilder methods
2  + if ( isAbstractAutoValueOrAutoBuilderMethod ( methodSymbol , state ) ) {
3    + return Description.NO_MATCH ;
4  + }
5  - // OK, now the real implementation: For each possible return branch, does the expression
6  - // returned look like "this" or instance methods that are also @CanIgnoreReturnValue.
7  - if (methodReturnsIgnorableValues(methodTree, state)) {
8  + // if the method looks like a builder, or if it always returns this, then make it @CIRV
9  + if (methodLooksLikeBuilder(methodSymbol) || methodReturnsIgnorableValues(methodTree,
       state)) {
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.1.2 Failure Example 2

Figure B.2 presents the second failure example. As seen in Figure B.2a, the ground truth repair involves modifying two assertions in the test case. In the first assertion, the *OpenTelemetryConstants* class is replaced with the *AttributeConstants* class in two instances. In the second assertion, the method for accessing the *span kind* has changed from direct access to accessing it via its *attributes*.

As shown in Figure B.2b, TARGET successfully repaired the first assertion in two of the five provided candidates (lines 9 and 21 of Figure B.2b). However, it failed to repair the second assertion in any candidate. Notably, in two candidates (lines 15 and 20), the second assertion was not generated at all. Upon reviewing the repair context included in the input (Figure B.2c), we observe that line 7 provides a hint about the *span kind* becoming an attribute. Additionally, the repair context excluded due to token limits (line 4 of Figure B.2d) could have offered further insights, particularly about the *SPAN_KIND_CLIENT*, if it had been included.

In conclusion, we believe this failure stems from the wide range of changes required for the repair, which likely reduced the model's focus on each individual change. While

additional repair context offers valuable clues, employing a multi-step or iterative repair process, which concentrates the model's attention on each assertion separately, could enhance repair success in such cases.

Figure B.2: **Failure Example 2** – A repair involving a wide variety of code changes, which reduces the model's focus on repairing each assertion.
TaRBench ID: *apache/shardingsphere:10788*
GitHub Commit Reference: *b5230f5c681*

```
@Test
public void assertMethod() {
  OpenTelemetryCommandExecutorTaskAdvice advice = new
     OpenTelemetryCommandExecutorTaskAdvice();
  advice.beforeMethod(getTargetObject(), null, new Object[]{}, "OpenTelemetry");
  advice.afterMethod(getTargetObject(), null, new Object[]{}, null, "OpenTelemetry");
  List<SpanData> spanItems = COLLECTOR.getSpanItems();
  assertThat(spanItems.size(), is(1));
  SpanData spanData = spanItems.get(0);
  assertThat(spanData.getName(), is("/ShardingSphere/rootInvoke/"));
- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
     OpenTelemetryConstants.COMPONENT)), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
     AttributeConstants.COMPONENT)), is(AttributeConstants.COMPONENT_NAME));
- assertThat(spanData.getKind(), is(SpanKind.CLIENT));
+ assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
     AttributeConstants.SPAN_KIND)), is(AttributeConstants.SPAN_KIND_CLIENT));
}
```

(a) **Ground Truth** Repair

```
- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    OpenTelemetryConstants.COMPONENT )), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeConstants.COMPONENT)), is (
    AttributeConstants.COMPONENT_NAME));
- assertThat(spanData.getKind(), is(SpanKind.CLIENT));
+ assertThat(spanData.getKind(), is(SpanKind.INTERNAL));

---------------------------------

- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    OpenTelemetryConstants.COMPONENT )), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    AttributeConstants.COMPONENT)) , is(AttributeConstants.COMPONENT_NAME));
- assertThat(spanData.getKind(), is(SpanKind.CLIENT));
+ assertThat(spanData.getKind(), is(SpanKind.INTERNAL));

---------------------------------

- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    OpenTelemetryConstants.COMPONENT )), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeConstants.COMPONENT)), is (
    AttributeConstants.COMPONENT_NAME));

---------------------------------

- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    OpenTelemetryConstants.COMPONENT )), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    AttributeConstants.COMPONENT)) , is(AttributeConstants.COMPONENT_NAME));

---------------------------------

- assertThat(spanData.getAttributes().get(AttributeKey.stringKey(
    OpenTelemetryConstants.COMPONENT )), is(OpenTelemetryConstants.COMPONENT_NAME));
+ assertThat(spanData.getAttributes().get(AttributeConstants.COMPONENT)), is (
    AttributeConstants.COMPONENT_NAME));
- assertThat(spanData.getKind(), is(SpanKind.CLIENT));
+ assertThat(spanData.getKind(), is(AttributeConstants.SPAN_KIND_INTERNAL));
```

(b) Selected Top **Repair Candidates** Generated by TARGET

```
- .setAttribute ( OpenTelemetryConstants.COMPONENT , OpenTelemetryConstants.COMPONENT_NAME )
+ .setAttribute ( AttributeConstants.COMPONENT , AttributeConstants.COMPONENT_NAME )
- .setAttribute ( OpenTelemetryConstants.DB_TYPE , OpenTelemetryConstants.DB_TYPE_VALUE )
- .setAttribute ( OpenTelemetryConstants.DB_STATEMENT , sql ) ;
+ .setAttribute ( AttributeConstants.DB_STATEMENT , sql )
- if ( null != rootSpan ) {
+ .setAttribute ( AttributeConstants.SPAN_KIND , AttributeConstants.SPAN_KIND_INTERNAL ) ;
- spanBuilder.setParent ( Context.current ( ) .with ( rootSpan ) ) ;
+ spanBuilder.setParent ( Context.current ( ) .with ( parentSpan ) ) ;
- }
- return spanBuilder.startSpan ( ) ;
+ Span result = spanBuilder.startSpan ( ) ;
+ target.setAttachment ( result ) ;
+ return result ;
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

```
1  - .setAttribute ( OpenTelemetryConstants.COMPONENT , OpenTelemetryConstants.COMPONENT_NAME )
2  + .setAttribute ( AttributeConstants.COMPONENT , AttributeConstants.COMPONENT_NAME )
3  - .setSpanKind ( SpanKind.CLIENT ) ;
4  + .setAttribute ( AttributeConstants.SPAN_KIND , AttributeConstants.SPAN_KIND_CLIENT ) ;
5  - Span result = spanBuilder.startSpan ( ) ;
6  + return spanBuilder.startSpan ( ) ;
7  - target.setAttachment ( result ) ;
8  - return result ;
```

(d) Relevant **Repair Context** Not Selected by TARGET and Excluded from the Model's Input

B.1.1.3 Failure Example 3

The third failure example, illustrated in Figure B.3, highlights a repair that primarily involves changing the way fields are created and retrieved from the *GeoShapeMapper* class. The ground truth repair, shown in Figure B.3a, replaces the use of the *Document* class and the *mapper.addFields* method with the *mapper.indexableFields* method. Additionally, the first *assertEquals* statement in the test case is removed, as it becomes redundant following these changes, while the second *assertEquals* is updated accordingly to reflect the new implementation.

In reviewing TARGET's top five repair candidates (Figure B.3b), we observe that all candidates correctly employ the new *mapper.indexableFields* method, remove the obsolete field handling logic, remove the redundant assertion, and update the remaining assertion. Despite these successes, none of the repair candidates are fully correct, as they introduce issues that result in compilation errors. Specifically, the first, third, and fourth candidates (lines 1, 11, and 16 in Figure B.3b) fail to close the double quotation marks for a long string. Meanwhile, the second and fifth candidates (lines 6 and 21) misuse the API of the *Columns* class.

Examining the relevant repair context in Figure B.3c, we see that the first three items (lines 2, 7, and 12) provide the model with information to understand the new *indexableFields* method. Additionally, the last item (line 17) demonstrates an alternative approach for adding columns. Although adding columns is not the central focus of the required repair, this extra information about the API usage contributes to the failure of these candidates.

In conclusion, similar to the example in Section B.1.1.2, this failure appears to result from the wide range of changes required. We hypothesize that a multi-step or iterative repair approach–where the model sequentially addresses each distinct aspect of the repair (e.g., the core field change and the column API change)—-could improve its success rate in cases involving multiple changes.

Figure B.3: **Failure Example 3** – A repair involving a wide variety of code changes changes.
TaRBench ID: *stratio/cassandra-lucene-index:571*
GitHub Commit Reference: *f9234609*

```
@Test
public void testAddFieldsWithValidUnionMultiPolygon2() {
  GeoShapeMapper mapper = geoShapeMapper().column("column").maxLevels(10).build("field");
- Columns columns = new Columns();
- columns.add("column",
    - "MULTIPOLYGON (((40 40, 2045, 4530, 4040)),((20 35, 1030, 1010, 305, 4520, 2035),
     (30 20, 2015, 2025, 3020)))");
- Document document = new Document();
- mapper.addFields(document, columns);
- assertEquals("Fields are not properly created", 1, document.getFields("field").length);
- assertEquals("Fields are not properly created", 1, document.getFields().size());
+ Columns columns = new Columns()
  + .add("column",
    + "MULTIPOLYGON (((40 40, 2045, 4530, 4040)),((20 35, 1030, 1010, 305, 4520, 2035),
     (30 20, 2015, 2025, 3020)))");
+ List<IndexableField> fields = mapper.indexableFields(columns);
+ assertEquals("Fields are not properly created", 1, fields.size());
}
```

(a) **Ground Truth** Repair

```
+ List < IndexableField > fields = mapper.indexableFields(new Columns()
      .add(Column.apply("column" ).withValue("MULTIPOLYGON (((40 40, 2045, 4530, 4040)),
      (20 35, 1030, 1010, 305, 4520, 2035),(30 20, 2015, 2025, 3020)))) ));
+ assertEquals("Fields are not properly created", 1, fields.size());

---------------------------------

+ List < IndexableField > fields = mapper.indexableFields(new Columns()
      .with("field" ) .withValue( "MULTIPOLYGON (((40 40, 2045, 4530, 4040)),
      ((20 35, 1030, 1010, 305, 4520, 2035),(30 20, 2015, 2025, 3020)))" ));
+ assertEquals("Fields are not properly created", 1, fields.size());

---------------------------------

+ List < IndexableField > fields = mapper.indexableFields(new Columns()
      .add(Column.apply("column" ).withValue("MULTIPOLYGON (((40 40, 2045, 4530, 4040)),
      ((20 35, 1030, 1010, 305, 4520, 2035),(30 20, 2015, 2025, 3020)))) ));
+ assertEquals("Fields are not properly created", 1, fields.size());

---------------------------------

+ List < IndexableField > fields = mapper.indexableFields(new Columns()
      .add(Column.apply("column" ).withValue("MULTIPOLYGON (((40 40, 2045, 4530, 4040)),
      (20 35, 1030, 1010, 305, 4520, 2035),(30 20, 2015, 2025, 3020)))));
+ assertEquals("Fields are not properly created", 1, fields.size());

---------------------------------

+ List < IndexableField > fields = mapper.indexableFields(new Columns()
      .with("column" ) .withValue( "MULTIPOLYGON (((40 40, 2045, 4530, 4040)),
      ((20 35, 1030, 1010, 305, 4520, 2035),(30 20, 2015, 2025, 3020)))" ));
+ assertEquals("Fields are not properly created", 1, fields.size());
```

(b) Selected Top **Repair Candidates** Generated by TARGET

```
-addFields(new Document(), columns);
+indexableFields(columns);

---------------------------------

-public void addFields ( Document document , Columns columns){
+public List < IndexableField > indexableFields ( Columns columns){

---------------------------------

-mapper.addFields(document , columns);
+fields.addAll(mapper.indexableFields ( columns)) ;

---------------------------------

- columns.add(ColumnsMapper.column(name, value, valueType));
+ columns = columns.add(Column.apply(name).withValue(ColumnsMapper.compose(
      value, valueType)));
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.1.4 Failure Example 4

In the fourth failure example shown in Figure B.4, the ground truth repair (Figure B.4a) reveals that four assertions in the test case were repaired. Specifically, the leading zero values in the expected outputs of the *Utils.formatDuration* method were removed.

Examining the top repair candidates in Figure B.4b, we observe that none of the generated solutions move toward removing the leading zeros, as seen in the ground truth repair. Moreover, the model introduces incorrect or redundant changes in some candidates. For instance, in line 4, it changes *"ms"* to *"s"*, despite the string already containing *"1sec"*. Similarly, in line 13, the time value is altered from *100* to *1000* milliseconds in both the input and expected output, which does not repair the test case. Additionally, in the second and fourth candidates (lines 13 and 29), the model changes only one of the four required assertions.

The relevant repair context provided to the model, shown in Figure B.4c, indicates that the SUT changed its logic for formatting the duration. Previously, the method used *String.format* to generate a string based on a predefined format stored in the *DURATION_FORMAT* variable. After the change, it relies on conditional logic to decide whether each time component should be included in the output. Despite the fact that this repair context should have been sufficient for repairing the test case, the model failed to fully grasp the change in logic, solely from the code differences.

In conclusion, this failure can be attributed to insufficient contextual understanding, similar to the example discussed in Section B.1.1.1. To enhance repair performance in cases like this, a more robust strategy that incorporates repository-level repair context—-such as considering the unchanged parts of the *formatDuration* method or analyzing usage examples of that method elsewhere in the SUT-—would likely yield better results.

Figure B.4: **Failure Example 4** – An example of missing additional repository-level unchanged contextual information from the model's input.
TaRBench ID: *j-easy/easy-batch:13*
GitHub Commit Reference: *2882d33b*

```
@Test
public void testFormatDuration() throws Exception {
- assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
      "0d 0hr 0min 0sec 100ms");
+ assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
      "100ms");
- assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
      "0d 0hr 0min 1sec 0ms");
+ assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
      "1sec 0ms");
- assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
      "0d 0hr 2min 0sec 0ms");
+ assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
      "2min 0sec 0ms");
- assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
      "0d 2hr 0min 0sec 0ms");
+ assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
      "2hr 0min 0sec 0ms");
  assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.DAYS))).isEqualTo(
     "2d 0hr 0min 0sec 0ms");
  // in the extremely unlikely event that the job takes more than a year
  assertThat(Utils.formatDuration(Duration.of(366, ChronoUnit.DAYS))).isEqualTo(
     "366d 0hr 0min 0sec 0ms");
}
```

(a) **Ground Truth** Repair

```
1  - assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 100ms");
2  + assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.SECONDS))).isEqualTo(
       "0d 0hr 0min 0sec 100s");
3  - assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
       "0d 0hr 0min 1sec 0ms");
4  + assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
       "0d 0hr 0min 1sec 0s");
5  - assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
       "0d 0hr 2min 0sec 0ms");
6  + assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
       "0d 0hr 2min 0sec 0s");
7  - assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
       "0d 2hr 0min 0sec 0ms");
8  + assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
       "0d 2hr 0min 0sec 0s");
9
10 ----------------------------------
11
12 - assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 100ms");
13 + assertThat(Utils.formatDuration(Duration.of(1000, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 1000ms");
14
15 ----------------------------------
16
17 - assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 100ms");
18 + assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0 d 0hr 0min 0sec 100ms");
19 - assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
       "0d 0hr 0min 1sec 0ms");
20 + assertThat(Utils.formatDuration(Duration.of(1, ChronoUnit.SECONDS))).isEqualTo(
       "0 d 0hr 0min 1sec 0ms");
21 - assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
       "0d 0hr 2min 0sec 0ms");
22 + assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.MINUTES))).isEqualTo(
       "0 d 0hr 2min 0sec 0ms");
23 - assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
       "0d 2hr 0min 0sec 0ms");
24 + assertThat(Utils.formatDuration(Duration.of(2, ChronoUnit.HOURS))).isEqualTo(
       "0 d 2hr 0min 0sec 0ms");
25
26 ----------------------------------
27
28 - assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 100ms");
29 + assertThat(Utils.formatDuration(Duration.of(100, ChronoUnit.MILLIS))).isEqualTo(
       "0d 0hr 0min 0sec 0ms");
```

(b) Selected Top **Repair Candidates** Generated by TARGET

```
- long millis = duration.toMillis ( ) ;
- return String.format ( DURATION_FORMAT , MILLISECONDS.toDays ( millis )
    , MILLISECONDS.toHours ( millis ) % 24,
  - MILLISECONDS.toMinutes ( millis ) % 60, MILLISECONDS.toSeconds ( millis )
      % 60, millis % 1000) ;
+ long days = duration.toDays ( ) ;
+ long hours = duration.toHours ( ) ;
+ long minutes = duration.toMinutes ( ) ;
+ long seconds = duration.getSeconds ( ) ;
+ long milliseconds = duration.toMillis ( ) ;
+ return ( days > 0? days + "d " : "" ) +
  + ( hours > 0? ( hours - TimeUnit.DAYS.toHours ( days ) ) + "hr " : "" ) +
  + ( minutes > 0? ( minutes - TimeUnit.HOURS.toMinutes ( hours ) ) + "min " : "" ) +
  + ( seconds > 0? ( seconds - TimeUnit.MINUTES.toSeconds ( minutes ) ) + "sec " : "" ) +
  + ( milliseconds - TimeUnit.SECONDS.toMillis ( seconds ) ) + "ms" ;
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.2 Successful Examples

B.1.2.1 Successful Example 1

TARGET generated an exact-match repair candidate in the first successful example shown in Figure B.5. As shown in Figure B.5a, the repair involves multiple changes, including replacing the *add* method with *addComposed* and removing the usage of the *Column.builder* and *buildWithComposed* methods.

Among the nine hunks included in the model's input by TARGET, the most relevant is highlighted in Figure B.5b. This particular hunk represents a change in the SUT that provides the necessary pattern for repairing the test case. Although the hunk utilizes a different method (*addDecomposed*) and is not directly invoked during construction, the model effectively grasps the essential concept without merely replicating it.

The commit associated with this example comprised a total of 47 hunks. Due to input size constraints, TARGET could include only 9 of these in the input. Nevertheless, the hunk selection and prioritization process ensured that the critical hunk required for the repair was selected first, accompanied by eight other relevant hunks. This successful exact-match example demonstrates the efficacy of our hunk selection strategy and the model's fine-tuning, which enables it to learn patterns rather than blindly replicating SUT changes.

Figure B.5: **Successful Example 1** – An exact-match repair where TARGET prioritizes the critical hunk and effectively applies its change pattern.
TaRBench ID: *stratio/cassandra-lucene-index:485*
GitHub Commit Reference: *fdd34e53*

```
1  @Test
2  public void testValidateColumns() {
3    Schema schema = SchemaBuilders.schema().mapper("field1", stringMapper()).build();
4  - Columns columns = new Columns().add(Column.builder("field1").buildWithComposed("value",
       UTF8Type.instance));
5  + Columns columns = new Columns().addComposed("field1", "value", UTF8Type.instance);
6    schema.validate(columns);
7    schema.close();
8  }
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
1  - columns.add ( Column.builder ( name ) .buildWithDecomposed ( value , valueType ) ) ;
2  + columns.addDecomposed ( name , value , valueType ) ;
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.2.2 Successful Example 2

Another exact-match example is presented in Figure B.6. As depicted in Figure B.6a, one of the assertion statements (lines 7 and 8) has been changed. Initially, the assertion checked if the singleton instance *UnauthorizedAction.INSTANCE* was exactly equal to the *action* variable. After the repair, the assertion was updated to verify only that the type of the *action* variable was *UnauthorizedAction*. This change occurred because the SUT update removed the singleton implementation of the instance.

TARGET included 10 hunks in the repair context for this input, with the highest-ranked hunk shown in Figure B.6b. This particular hunk illustrates that the return value of the singleton instance was replaced by a new instance of the class. The remaining nine hunks in the repair context either represented the same change for other classes or were irrelevant to the repair.

In this case, the commit involved 44 hunks. Once again, TARGET successfully identified and prioritized the most relevant hunk as the top selection in the input. Moreover, there were no explicit examples or clues in the input regarding the use of *instanceof*, yet the model accurately predicted it solely based on the provided hunk. This demonstrates the effectiveness of the model's learned knowledge, acquired through pre-training and fine-tuning.

Figure B.6: **Successful Example 2** – An exact-match repair example where TARGET correctly prioritizes the most critical hunk, demonstrating the model's ability to leverage its learned knowledge for an effective repair.
TaRBench ID: *pac4j/pac4j:397*
GitHub Commit Reference: *5f7dfe653*

```
@Test
public void testBuildUnauthenticated403WithHeader() {
  HttpActionHelper.setAlwaysUse401ForUnauthenticated(false);
  final WebContext context = MockWebContext.create();
  context.setResponseHeader(HttpConstants.AUTHENTICATE_HEADER, VALUE);
  final var action = HttpActionHelper.buildUnauthenticatedAction(context);
  - assertEquals ( UnauthorizedAction.INSTANCE , action ) ;
  + assertTrue ( action instanceof UnauthorizedAction ) ;
  assertEquals(VALUE, context.getResponseHeader(HttpConstants.AUTHENTICATE_HEADER).get());
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
-return UnauthorizedAction.INSTANCE ;
+return new UnauthorizedAction ( ) ;
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.2.3 Successful Example 3

The ground truth repair for this example (Figure B.7) occurs in the final assert statement. As illustrated in Figure B.7a, the last assertion originally checks if the *httpRequestMatchers* list is empty. The repair involves invoking the *toSortedList* method on this attribute, which is necessary because the type of *httpRequestMatchers* was changed in the update to the SUT.

The relevant repair context, provided in the input (Figure B.7c), indicates the change in how the SUT handles *httpRequestMatchers*. Specifically, instead of treating it as a list, the *stream* method is now invoked on it. However, the changed type is not explicitly stated. In response to this change, TARGET generates a plausible repair, as shown in Figure B.7b. The plausible repair adopts a more generalized strategy: instead of converting the variable to a list to check for emptiness, it directly calls the *size* method and compares the result to zero. This approach is broadly applicable, as the *size* method is typically implemented for most collection types, and it maintains semantic equivalence with the ground truth repair without altering the intent of the test case.

This example demonstrates TARGET's capability to explore diverse, yet correct, repair candidates. It also underscores the importance of metrics, such as plausible repair accuracy (introduced in our study), which evaluate model performance when the prediction does not exactly match the ground truth.

Figure B.7: **Successful Example 3** – An example of a plausible repair generated by TARGET, which is an equivalent repair to the ground truth that preserves the original test's intent.
TaRBench ID: *mock-server/mockserver:2081*
GitHub Commit Reference: *8ddaa46ea*

```
@Test
public void shouldNotRemoveAfterTimesCompleteOrExpired() {
  // when
  requestMatchers.add(new Expectation(new HttpRequest().withPath("somepath"),
    Times.exactly(0), TimeToLive.unlimited(), 0
    ).thenRespond(response().withBody("someBody")), API);
  requestMatchers.add(new Expectation(request().withPath("someOtherPath"),
    Times.unlimited(), TimeToLive.exactly(TimeUnit.MICROSECONDS, 0L), 0
    ).thenRespond(response().withBody("someOtherBody")), API);
  requestMatchers.add(new Expectation(request().withPath("someOtherPath"),
    Times.exactly(0), TimeToLive.exactly(TimeUnit.MICROSECONDS, 0L), 0
    ).thenRespond(response().withBody("someOtherBody")), API);
  // then
  assertThat(requestMatchers.firstMatchingExpectation(new
    HttpRequest().withPath("somepath")), nullValue());
  assertThat(requestMatchers.firstMatchingExpectation(new
    HttpRequest().withPath("someOtherPath")), nullValue());
  - assertThat(requestMatchers.httpRequestMatchers, empty());
  + assertThat(requestMatchers.httpRequestMatchers.toSortedList(), empty());
}
```

(a) **Ground Truth** Repair

```
- assertThat ( requestMatchers.httpRequestMatchers , empty ( ) ) ;
+ assertThat ( requestMatchers.httpRequestMatchers.size ( ) , is ( 0) ) ;
```

(b) Top **Plausible Repair Candidate** Generated by TARGET

```
- return new ArrayList < > ( httpRequestMatchers ) ;
+ return httpRequestMatchers.stream ( ) .collect ( Collectors.toList ( ) ) ;

---------------------------------

- new ArrayList < > ( httpRequestMatchers ) .forEach ( httpRequestMatcher ->
    removeHttpRequestMatcher ( httpRequestMatcher , cause , false ) ) ;
+ httpRequestMatchers.stream ( ) .forEach ( httpRequestMatcher ->
    removeHttpRequestMatcher ( httpRequestMatcher , cause , false ) ) ;

---------------------------------

- private List < HttpRequestMatcher > getHttpRequestMatchersCopy ( ) {
- httpRequestMatchersCopy.compareAndSet ( null , httpRequestMatchers.toSortedList ( ) ) ;
- return httpRequestMatchersCopy.get ( ) ;
+ private Stream < HttpRequestMatcher > getHttpRequestMatchersCopy ( ) {
+ return httpRequestMatchers.stream ( ) ;
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.1.2.4 Successful Example 4

In this successful case (Figure B.8), the ground truth repair involves two changes to the point where the test case expects an exception to be thrown. As illustrated in Figure B.8a, the first change replaces the exception type *FailedPreconditionRuntimeException* with *InternalRuntimeException*. The second change updates the API used by the testing framework to check for the thrown exception: the new version of the test case uses *Assert.assertThrows* instead of *mThrown.expect*, which was used in the earlier version. It is important to note that this second change is related to the testing framework itself and is unrelated to any changes in the SUT.

The relevant repair context, depicted in Figure B.8c, provides clear guidance for the model. In this instance, TARGET generates a plausible repair candidate, shown in Figure B.8b. This candidate repair only addresses the change to the exception type. Similar to the example discussed in Section B.1.2.3, this repair is functionally equivalent to the ground truth, preserving the testing intent. Furthermore, the additional change present in the ground truth is a test refactoring and is thus unnecessary for the repair. This type of plausible repair is advantageous because it remains simple, effectively isolating the required code repair from any unrelated test refactoring.

Figure B.8: **Successful Example 4** – An example of a plausible repair generated by TARGET, which is an equivalent repair to the ground truth that isolates test repair from refactoring.
TaRBench ID: *alluxio/alluxio:1164*
GitHub Commit Reference: *6982d6c759*

```
@Test
public void close() throws Exception {
  ByteBuffer buf = BufferUtils.getIncreasingByteBuffer(TEST_BLOCK_SIZE);
  Assert.assertEquals(TEST_BLOCK_SIZE, mWriter.append(buf));
  mWriter.close();
  - mThrown.expect(FailedPreconditionRuntimeException.class);
  - mWriter.append(buf);
  + Assert.assertThrows(InternalRuntimeException.class, () -> mWriter.append(buf));
}
```

(a) **Ground Truth** Repair

```
- mThrown.expect ( FailedPreconditionRuntimeException.class ) ;
+ mThrown.expect ( InternalRuntimeException.class ) ;
```

(b) Top **Plausible Repair Candidate** Generated by TARGET

```
1  - return new FailedPreconditionRuntimeException ( t ) ;
2  + return new InternalRuntimeException ( t ) ;
```

(c) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

B.2 A Comparative Analysis of Test Case Repairs Generated by TaRGET and Ceprot

This section supplements the discussion in Section 3.4.4.3 by presenting five detailed examples where TaRGET outperformed Ceprot, the baseline approach analyzed in our study. These examples illustrate the limitations of Ceprot and highlight the strengths of TaRGET. Explanations for each example are provided in the corresponding figure captions.

Figure B.9: **Comparison 1** – Test repair where TaRGET generated an exact match repair while Ceprot made no changes to the broken test, despite having sufficient repair context.
Ceprot *test_db* ID: 4244531
GitHub Commit Reference: *7c693db546*

```
@Test
+ public void writeStream() throws IOException
{
  WriteCallback callback = new WriteCallback()
  {
    @Override
    public void writeData(Attributes attributes)
    {
    }
  };
  ByteArrayResponse response = new ByteArrayResponse();
  Attributes attributes = new Attributes(new MockWebRequest(new Url()), response);
  byte[] srcData = new byte[5000];
  for (int i = 0; i < srcData.length; i++)
  {
    srcData[i] = (byte)i;
  }
  InputStream in = new ByteArrayInputStream(srcData);
  callback.writeStream(attributes, in);
  assertTrue("Content not equal", Arrays.equals(response.getBytes(), srcData));
  }
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TaRGET

```
- protected final void writeStream(Attributes attributes, InputStream stream)
+ protected final void writeStream(Attributes attributes, InputStream stream)
      throws IOException
```

(b) Relevant **Repair Context** Selected by TaRGET and Included in the Model's Input

```
// No change to broken test
```

(c) Incorrect Repair Generated by Ceprot. Ceprot Makes No Change to the Broken Test.

```
-protected final void writeStream(Attributes attributes, InputStream stream) {
    final Response response = attributes.getResponse(); try { Streams.copy(stream,
    response.getOutputStream()); } catch(IOException e) { throw new WicketRuntimeException(
    e); } }
+protected final void writeStream(Attributes attributes, InputStream stream)
    throws IOException { final Response response = attributes.getResponse(); Streams.copy(
    stream, response.getOutputStream()); }
```

(d) CEPROT Repair Context. Contains Sufficient Context to Perform the Required Repair.

Figure B.10: **Comparison 2** – Test repair where TARGET generated an exact match repair while CEPROT removed much of the broken test without generating a replacement, despite having sufficient repair context.
Ceprot *test_db* ID: 262742
GitHub Commit Reference: *b90ffa1af1*

```
@Test
public void deleteTest() throws Exception {
  CreateDirectoryOptions recMkdir = CreateDirectoryOptions.defaults().setRecursive(true);
  DeleteOptions recDelete =
    - new DeleteOptions.Builder().setRecursive(true).build();
    + DeleteOptions recDelete = DeleteOptions.defaults().setRecursive(true);
  for (int i = 0; i < 10; i ++) {
    String dirPath = "/i" + i;
    mTfs.createDirectory(new TachyonURI(dirPath), recMkdir);
    for (int j = 0; j < 10; j ++) {
      CreateFileOptions option = CreateFileOptions.defaults().setBlockSizeBytes((i + j + 1)
     * 64);
      String filePath = dirPath + "/j" + j;
      mTfs.createFile(new TachyonURI(filePath), option).close();
      if (j >= 5) {
        mTfs.delete(new TachyonURI(filePath), recDelete);
      }
    }
    if (i >= 5) {
      mTfs.delete(new TachyonURI(dirPath), recDelete);
    }
  }
  mLocalTachyonCluster.stopTFS();
  deleteTestUtil();
  deleteFsMasterJournalLogs();
  deleteTestUtil();
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
- DeleteOptions options = new DeleteOptions.Builder().setRecursive(true).build();
+ DeleteOptions options = DeleteOptions.defaults().setRecursive(true);

---------------------------------------------------

- DeleteOptions options = new DeleteOptions.Builder().setRecursive(recursive).build();
+ DeleteOptions options = DeleteOptions.defaults().setRecursive(recursive);
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

```
- @Test public void deleteTest()throws Exception { CreateDirectoryOptions recMkdir =
      CreateDirectoryOptions.defaults().setRecursive(true); DeleteOptions recDelete = new
      DeleteOptions.Builder().setRecursive(true).build(); for(int i = 0; i < 10; i ++ ) { String dirPath = "/i" + i;
      mTfs.createDirectory(new TachyonURI(dirPath), recMkdir); for(int j = 0; j < 10; j ++ ) { CreateFileOptions
      option = CreateFileOptions.defaults().setBlockSizeBytes((i + j + 1) * 64); String filePath = dirPath + "/j" + j;
      mTfs.createFile(new TachyonURI(filePath), option).close(); if(j >= 5) { mTfs.delete(new TachyonURI(filePath),
      recDelete); } } if(i >= 5) { mTfs.delete(new TachyonURI(dirPath), recDelete); } } mLocalTachyonCluster.stopTFS();
      deleteTestUtil(); deleteFsMasterJournalLogs(); deleteTestUtil(); }
+ @Test public void deleteTest()throws Exception { CreateDirectoryOptions recMkdir =
      CreateDirectoryOptions.defaults().setRecursive(true); DeleteOptions recDelete = new
      DeleteOptions.Builder().setRecursive(true).build(); for(int i = 0; i < 10; i ++ ) { String dirPath = "/i" + i;
      mTfs.createDirectory(new TachyonURI(dirPath), recMkdir); for(int j = 0; j < 10; j ++ ) { CreateFileOptions
      option = CreateFileOptions.defaults().setBlockSizeBytes((i + j + 1) * 64); String filePath = dirPath + "/j" + j;
      mTfs.createFile(new T
```

(c) Incorrect Repair Generated by CEPROT. CEPROT Deletes Much of the Test.

```
- @Override public boolean delete(Path cPath, boolean recursive)throws IOException { LOG.info("delete({}, {})", cPath,
      recursive); if(mStatistics != null) { mStatistics.incrementWriteOps(1); } TachyonURI path = new
      TachyonURI(Utils.getPathWithoutScheme(cPath)); DeleteOptions options = new
      DeleteOptions.Builder().setRecursive(recursive).build(); try { mTFS.delete(path, options); return true; }
      catch(InvalidPathException e) { LOG.info("delete failed: {}", e.getMessage()); return false; }
      catch(FileDoesNotExistException e) { LOG.info("delete failed: {}", e.getMessage()); return false; }
      catch(TachyonException e) { throw new IOException(e); } }
+ @Override public boolean delete(Path cPath, boolean recursive)throws IOException { LOG.info("delete({}, {})", cPath,
      recursive); if(mStatistics != null) { mStatistics.incrementWriteOps(1); } TachyonURI path = new
      TachyonURI(Utils.getPathWithoutScheme(cPath)); DeleteOptions options =
      DeleteOptions.defaults().setRecursive(recursive); try { mTFS.delete(path, options); return true; }
      catch(InvalidPathException e) { LOG.info("delete failed: {}", e.getMessage()); return false; }
      catch(FileDoesNotExistException e) { LOG.info("delete failed: {}", e.getMessage()); return false; }
      catch(TachyonException e) { throw new IOException(e); } }
```

(d) CEPROT Repair Context. Contains Sufficient Context to Perform the Required Repair.

Figure B.11: **Comparison 3** – Test repair where TARGET generated an exact match repair while CEPROT removed much of the broken test without generating a replacement, possibly because of its insufficient repair context.
Ceprot *test_db* ID: 2229339
GitHub Commit Reference: *83d814eefb*

```
@Test
public void addAllAsync_manyTimesRoundTheRing() throws Exception {
  RingbufferConfig c = config.getRingbufferConfig(ringbuffer.getName());
  Random random = new Random();
  for (int iteration = 0; iteration < 1000; iteration++) {
    List<String> items = randomList(max(1,    random.nextInt(c.getCapacity())));
    long previousTailSeq = ringbuffer.tailSequence();
    - long result = ringbuffer.addAllAsync(items, OVERWRITE).get();
    + long result = ringbuffer.addAllAsync(items, OVERWRITE).toCompletableFuture().get();
    assertEquals(previousTailSeq + items.size(),    ringbuffer.tailSequence());
    if (ringbuffer.tailSequence() < c.getCapacity()) {
      assertEquals(0, ringbuffer.headSequence());
    } else {
      assertEquals(ringbuffer.tailSequence() - c.getCapacity() + 1,
     ringbuffer.headSequence());
    }
    assertEquals(ringbuffer.tailSequence(), result);
    long startSequence = previousTailSeq + 1;
    for (int k = 0; k < items.size(); k++) {
      assertEquals(items.get(k), ringbuffer.readOne(startSequence + k));
    }
  }
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
- return ringbuffer.addAsync(message, OverflowPolicy.OVERWRITE).get();
+ return ringbuffer.addAsync(message, OverflowPolicy.OVERWRITE).toCompletableFuture().get();
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

```
- @Test public void addAllAsync_manyTimesRoundTheRing()throws Exception { RingbufferConfig c =
    config.getRingbufferConfig(ringbuffer.getName()); Random random = new Random(); for(int iteration = 0; iteration
    < 1000; iteration ++ ) { List < String > items = randomList(max(1, random.nextInt(c.getCapacity()))); long
    previousTailSeq = ringbuffer.tailSequence(); long result = ringbuffer.addAllAsync(items, OVERWRITE).get();
    assertEquals(previousTailSeq + items.size(), ringbuffer.tailSequence()); if(ringbuffer.tailSequence() <
    c.getCapacity()) { assertEquals(0, ringbuffer.headSequence()); } else { assertEquals(ringbuffer.tailSequence()
    - c.getCapacity() + 1, ringbuffer.headSequence()); } assertEquals(ringbuffer.tailSequence(), result); long
    startSequence = previousTailSeq + 1; for(int k = 0; k < items.size(); k ++ ) { assertEquals(items.get(k),
    ringbuffer.readOne(startSequence + k)); } } }
+ @Test public void addAllAsync_manyTimesRoundTheRing()throws Exception { RingbufferConfig c =
    config.getRingbufferConfig(ringbuffer.getName()); Random random = new Random(); for(int iteration = 0; iteration
    < 1000; iteration ++ ) { List < String > items = randomList(max(1, random.nextInt(c.getCapacity()))); long
    previousTailSeq = ringbuffer.tailSequence(); long result = ringbuffer.addAllAsync(items, OVERWRITE).get();
    assertEquals(previousTailSeq + items.size(), ringbuffer.tailSequence()); if(ringbuffer.tailSequence() <
    c.getCapacity()) { assertEquals(0, ringbuffer.head
```

(c) Incorrect Repair Generated by CEPROT. CEPROT Deletes Much of the Test.

```
- ICompletableFuture<Long> addAllAsync(@Nonnull Collection<? extends E> collection,
    @Nonnull OverflowPolicy overflowPolicy);
+ CompletionStage<Long> addAllAsync(@Nonnull Collection<? extends E> collection,
    @Nonnull OverflowPolicy overflowPolicy);
```

(d) CEPROT Repair Context. Contains Insufficient Context to Perform the Required Repair.

Figure B.12: **Comparison 4** – Test repair where TARGET generated an exact match repair while CEPROT produced a repair that, although compilable, was incorrect, despite having sufficient repair context.
Ceprot *test_db* ID: 3675646
GitHub Commit Reference: *7138c860e24*

```
@Test(expectedExceptions=NullPointerException.class)
public void test_appendValueReduced_null() throws Exception {
- builder.appendValueReduced(null, 2, 2000);
+ builder.appendValueReduced(null, 2, 2, 2000);
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
- appendValueReduced ( field , 2, 2000) ;
+ appendValueReduced ( field , 2, 2, 2000) ;
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

```
- @Test(expectedExceptions = NullPointerException.class)
    public void test_appendValueReduced_null() throws Exception {
    builder.appendValueReduced(null, 2, 2000); }
+ @Test(expectedExceptions = NullPointerException.class)
    public void test_appendValueReduced_null() throws Exception {
    builder.appendValueReduced(null, 2, 2000, 2); }
```

(c) Incorrect Repair Generated by CEPROT. CEPROT Generates an Incorrect Repair.

```
- public DateTimeFormatterBuilder appendValueReduced(TemporalField field, int width, int baseValue) {
    Objects.requireNonNull(field, "field"); ReducedPrinterParser pp = new ReducedPrinterParser(field, width,
    baseValue); appendFixedWidth(width, pp); return this; }
+ public DateTimeFormatterBuilder appendValueReduced(TemporalField field, int width, int maxWidth, int baseValue) {
    Objects.requireNonNull(field, "field"); ReducedPrinterParser pp = new ReducedPrinterParser(field, width,
    maxWidth, baseValue); if(width == maxWidth) { appendFixedWidth(width, pp); } else { appendInternal(pp); }
    return this; }
```

(d) CEPROT Repair Context. Contains Sufficient Context to Perform the Required Repair.

Figure B.13: **Comparison 5** – Test repair where TARGET generated an exact match repair while CEPROT only modified whitespace, possibly due to irrelevant repair context.
Ceprot *test_db* ID: 4640902
GitHub Commit Reference: *5570aea8cb*

```
@Test public void serialize() {
  for(final String[] test : TOCSV) {
    final String query = test[1].isEmpty() ? _CSV_SERIALIZE.args(test[0]) :
    _CSV_SERIALIZE.args(test[0], " {" + test[1] + "}");
    if(test.length == 2) {
      - error(query, Err.BXCS_CONFSEP, Err.ELMOPTION);
      + error(query, Err.BXCS_CONFSEP, Err.INVALIDOPT);
    } else {
      query(query, test[2]);
    }
  }
}
```

(a) **Ground Truth** Repair and **the Exact Match Repair Candidate** Generated by TARGET

```
- if(mode == null) ELMOPTION.thrw(info, md);
+ if(mode == null) INVALIDOPT.thrw(info, Util.info(Text.INVALID_VALUE_X_X, "mode", md));

--------------------------------------------------

- ELMOPTION(FORG, 6, "&^Option '^&%&^' is invalid^&."),
+ INVALIDOPT(FORG, 6, "%."),
+ / ** FORG0006. * /
+ INVALIDOPTX(FORG, 6, Text.UNKNOWN_OPTION_X),
```

(b) Relevant **Repair Context** Selected by TARGET and Included in the Model's Input

```
- @Test public void serialize() { for(final String[]test : TOCSV) { final String query = test[1].isEmpty()?
      _CSV_SERIALIZE.args(test[0]) : _CSV_SERIALIZE.args(test[0], " {" + test[1] + "}"); if(test.length == 2) {
      error(query, Err.BXCS_CONFSEP, Err.ELMOPTION); } else { query(query, test[2]); } } }
+ @Test public void serialize() { for(final String[]test : TOCSV) { final String query = test[1].isEmpty()?
      _CSV_SERIALIZE.args(test[0]) : _CSV_SERIALIZE.args(test[0], " {" + test[1] + "}"); if(test.length == 2) {
      error(query, Err.BXCS_CONFSEP, Err.ELMOPTION); } else { query(query, test[2]); } } }
```

(c) Incorrect Repair Generated by CEPROT. CEPROT Generates a Repair that Only Contains Whitespace Changes.

```
- public final void serialize(final Item item)throws IOException { openResult(); if(item instanceof ANode) { final
      Type type = item.type; if(type == NodeType.ATT)SERATTR.thrwSerial(item); if(type ==
      NodeType.NSP)SERNS.thrwSerial(item); serialize((ANode)item); } else if(item instanceof FItem) {
      SERFUNC.thrwSerial(item.description()); } else { finishElement(); atomic(item); } closeResult(); }
+ public final void serialize(final Item item)throws IOException { openResult(); if(item instanceof ANode) { final
      Type type = item.type; if(type == NodeType.ATT)SERATTR.thrwIO(item); if(type == NodeType.NSP)SERNS.thrwIO(item);
      serialize((ANode)item); } else if(item instanceof FItem) { SERFUNC.thrwIO(item.description()); } else {
      finishElement(); atomic(item); } closeResult(); }
```

(d) CEPROT Repair Context. Contains Insufficient Context to Perform the Required Repair.