Should This Code Get Tested? A Study into Code and Non-Code Characteristics Leading to Test-Suite Modifications

Mirko Sander Boon

Supervisors: Burcu Kulahcioglu Ozkan, Carolin Brandt

Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Master of Computer Science
June 18, 2026

Name of the student: Mirko Sander Boon
Thesis committee: Burcu Kulahcioglu Ozkan, Carolin Brandt, Gosia Migut

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Software testing is essential for maintaining software quality, yet determining which code changes require corresponding test modifications remains a challenging and time-consuming task. This thesis investigates whether test-suite co-evolution can be predicted from a combination of code-, coverage-, repository-level and non-code project and pull request characteristics and studies which characteristics are most informative in both within-project and cross-project settings.

To answer this question, we constructed a dataset containing 72,534 modified lines extracted from 1,303 pull requests across 18 open-source Java projects. Using coverage information from the modified test suites, we derived line-level co-evolution labels and extracted characteristics from multiple scopes, including line, method, class, repository, pull request and project-level metrics. Several machine learning models were evaluated, with Gradient Boosting achieving the strongest overall performance.

The results show that test-suite modifications can be predicted with moderate accuracy. The best-performing model achieved an average MCC of 0.357 and an AUC of 0.805 in the within-project setting, and a MCC of 0.454 and an AUC of 0.839 in the cross-project setting. Historical coverage was the strongest individual predictor, but repository-level and non-code characteristics provided substantial additional predictive value. Furthermore, broader developmental and repository-level characteristics proved more informative than localized code metrics, and cross-project prediction consistently outperformed within-project prediction.

These findings demonstrate that software repositories contain meaningful signals regarding future testing behavior and that test-suite co-evolution can be predicted using information available during development.

Contents

1 Introduction **3**
- 1.1 Research Questions . . . 3
- 1.2 Approach . . . 4
- 1.3 Contributions . . . 4

2 Related Work **4**
- 2.1 Co-evolution . . . 4
  - Detecting co-evolution pairs . . . 5
- 2.2 Empirical studies on code coverage . . . 5
- 2.3 Defining characteristics and metrics . . . 5
  - Code characteristics . . . 5
  - Non-code project characteristics . . . 5
- 2.4 Test coverage prediction . . . 6
  - SITAR . . . 6
  - DRIFT . . . 6
  - Brandt and Ramirez . . . 6
- 2.5 Alternatives for reducing test bloat . . . 6
- 2.6 Explainable machine learning models . . . 7
  - Defining interpretability . . . 7
  - Interpretability in practice . . . 7

3 Methodology **7**
- 3.1 Data collection . . . 7
  - Setting the project criteria . . . 7
  - Creating the dataset . . . 8
- 3.2 Feature selection . . . 8
- 3.3 Feature extraction . . . 10
  - Detecting co-evolution . . . 10
  - Code metrics . . . 10
  - Non-code metrics . . . 11
- 3.4 Creating the dataset . . . 11
- 3.5 Data pre-processing . . . 11
  - Removing correlated features . . . 12
  - Normalizing metrics . . . 12
- 3.6 Evaluation . . . 12
  - Feature importance . . . 12

4 Results **13**
- 4.1 Dataset creation . . . 13
- 4.2 Dataset analysis . . . 13
  - Feature correlations . . . 13
- 4.3 Cross-Project and Within-Project Predictive Performance Evaluation . . . 14
  - Model selection . . . 14
  - Repository performance . . . 14
  - Characteristics of predictable projects . . . 16
  - Baseline comparison . . . 17
  - Comparison between within project and cross-project predictions . . . 17
- 4.4 Feature Importance Evaluation . . . 17
  - Individual Feature Analysis . . . 18
  - Category importance . . . 18

5 Discussion **18**
- 5.1 Prevalence of co-evolution . . . 18
- 5.2 Buildability and testability . . . 21
- 5.3 Feasibility of test suite modification prediction . . . 21
- 5.4 Feature importance . . . 22
  - Partial Dependence Analysis . . . 23
- 5.5 Differences between within- and cross-project prediction . . . 23
- 5.6 Threats to validity . . . 24
  - Construct validity . . . 24
  - Internal validity . . . 24
  - External validity . . . 25
- 5.7 Implications and applicability . . . 25

6 Conclusions and Future Work **26**
- 6.1 Conclusion . . . 26
- 6.2 Contributions . . . 26
- 6.3 Future work . . . 26
  - Semantic based features . . . 26
  - Improving co-evolution detection . . . 27
  - Validating on other scopes . . . 27
  - Integration in developer workflow . . . 27

1 Introduction

According to the OECD the ICT sector grew 3 times as much as the rest of the economy during the period from 2013 to 2023 [1], showing the ever-increasing importance of software. With the increasing use of software systems, there is also an increasing dependence on software systems. When these systems fail, the consequences can be devastating. An example of this is the CrowdStrike bug of July 2024 that caused outages for roughly 8.5 million PC's, most of which were used in vital sectors [2]. In addition to the major disruption across multiple industries, the financial consequences were enormous, with damages estimated in the billions of dollars [3, 4]. CrowdStrike noted three reasons for the bug occurring: latent bugs caused by inadequate testing, a mismatch in integration between two parts of the system, and a missing test case for a specific scenario [5].

This example and the accompanying analysis show us the importance of software verification, and in particular, the importance of software testing. This importance is further strengthened by the fact that discovering bugs early in the development cycle can drastically reduce software development costs [6]. There are many ways in which one can verify the correctness of the code, such as manual testing, static analysis and formal verification. Another common method used is automated software testing, with 61.7% of open source projects employing the method [7]. This popularity of automated software testing as a testing method can be explained by the finding that increasing the amount of code that is covered by automated software tests leads to a lower number of field-reported problems [8].

Automated testing checks the functional correctness of code on different levels using unit-, integration-, and system tests. Each of these has different purposes in bug prevention. Unit tests are made to isolate small blocks of logic and verify the behavior of this logic for the possible set of inputs, while integration tests focus on the communication between different components. System tests aim to model a real user experience, testing the software behavior instead of the implementation.

Modern development practices rely heavily on automated test suites consisting of these unit, integration and system tests. One of these development methods is continuous integration. In continuous integration the goal is to detect issues early by running automated checks whenever new code is added. This is commonly done using an automated test suite. Another popular modern development method reliant on automated test suites is Test-driven development (TDD), which entails writing tests specifying the desired behavior before writing the corresponding production code. The goal is to guide design decisions and improve test coverage from the outset [9].

As software systems evolve, test code must evolve alongside production code to remain effective, a phenomenon commonly referred to as co-evolution between production and test code. Empirical studies have shown that changes to production code frequently necessitate corresponding changes to test cases, either to extend coverage for new functionality or to repair tests that break due to changed behavior [10–14].

Code bases generally tend to grow larger over time, and as the amount of production code increases, so does the test suite size [10]. Running test suites for these larger projects can take days or even weeks [15]. This can drastically reduce employee productivity, thus highlighting the importance of reducing the time spent running the test suite.

In literature, there are a couple of different areas trying to tackle long-running test suites. These are roughly described as test case selection, test case prioritization and test suite minimization [15]. Test case selection attempts to only run that part of the test suite that is relevant to the production changes, while test case prioritization still runs the entire test suite; however, test cases which are more likely to detect faults are prioritized. Lastly, in the case of test suite minimization, redundant tests are permanently removed from the test suite, effectively shrinking the test suite.

Developing tests is quite expensive, with the share of testing costs of total production costs estimated at 50% or more [16]. Furthermore, it is suggested that the optimal level of coverage is well short of 100% [8]. Preventing the development of unnecessary test cases can thus both reduce the development costs as well as reduce the issue of long-running test suites. An issue with all the methods listed above is that they only work once the test cases are already written. Thus the question of which parts of the codebase should be covered does not yet have a definitive answer.

Some prior empirical work has been done to determine the characteristics of code that make it more likely to lead to test changes or test cases. These characteristics include properties of the code change itself, such as the types of modifications performed, the affected program elements, and the size and complexity of the change [11, 13, 17]. One such work is the work of Wang et al. [11] where they propose SITAR, a classifier that predicts whether or not a code change on the commit level should lead to corresponding test changes. They tested this method both to predict coverage in a within-project as well as a cross-project setup. This approach solely considers unit tests and only considers patch changes, not the context of the changes, leading to a possible incomplete view of the problem.

Another study looked into the influence of the line and method properties on the probability of that line being covered [17]. In this case, the method context is taken into account, but the context of other scopes is disregarded. Furthermore, this work considers only a single commit from a single project, making it impossible to generalize the answer to other projects.

However, previous research found that testing and quality assurance standards differ across projects [18, 19]. The question then arises: are there non-code project characteristics that we can use to predict coverage? Furthermore, knowing that there are differences in practices across projects, are there differences between the important characteristics in a within and cross-project setting for predicting coverage?

1.1 Research Questions

We hypothesize that certain code characteristics can be used to predict whether or not code needs to be tested. Furthermore, we believe that the characteristics of the surrounding

scope of the code also have an impact. In this context, scope refers to the structural surroundings of a code fragment, including the method in which it appears, the class that defines it, and the repository it belongs to. Lastly, this thesis aims to study whether there are any external non-code characteristics that can be used to predict if code should be tested.

To better understand which characteristics of production changes lead to test changes, we present a study that, for multiple projects, analyzes multiple pull requests (PRs) and investigates the relation between production changes and changes in the test suite. In this paper, we aim to answer the following research questions:

RQ 1. **How accurately can we predict test suite modifications from a combination of pull request code characteristics and project non-code characteristics?**

RQ 2. **What is the influence of differently scoped characteristics on the prediction of test suite modifications in both a within as well as a cross-project setting?**

- *RQ 2.1* **What is the importance of different line-level characteristics?**
- *RQ 2.2* **What is the importance of different method-level characteristics?**
- *RQ 2.3* **What is the importance of different class-level characteristics?**
- *RQ 2.4* **What is the importance of different repository-level characteristics?**
- *RQ 2.5* **What is the importance of different non-code project characteristics?**

RQ 3. **What are the differences in which characteristics are important in a cross-project setting compared to the within-project setting?**

1.2 Approach

We begin with an exploratory literature study to identify existing code characteristics, commonly used metrics and prior work that investigates the relationship between code changes and testing activities. This study was used for the selection of metrics and helps position our work within the existing body of research.

Next, we construct a dataset of PRs mined from open-source GitHub projects written in Java and built using Gradle. Only PRs that contain at least one production code change are considered. For each PR, we extract a range of line, repository, class and method-level metrics describing the modified production code. We then merge these metrics with the line-level metrics, resulting in a dataset where each modified line of production code is associated with the characteristics of its surrounding scope.

To obtain the target labels for this dataset, we analyze test changes introduced in the same PR. We instrument the projects to collect test coverage data at the level of individual test files and execute the modified tests to determine which lines of production code they cover. A production line is labeled as a positive sample if it is covered by at least one of the changed tests, and as a negative sample otherwise.

Finally, the resulting dataset is divided into training, validation, and test sets and used to train and evaluate a set of explainable machine learning models. These models enable us to assess not only the predictive power of different code characteristics, but also their relative importance, thereby providing insight into which properties of production code changes are most strongly associated with test additions in PRs.

1.3 Contributions

In summary, this paper makes the following contributions:

- A comprehensive dataset containing code characteristics on the line, method, class and repository level as well as non-code project characteristics across multiple PRs and projects.
- Insights into the feasibility of pull-request-based predictions of test suite modification, including insight into the importance of the different characteristics.
- A replication package capable of reproducing the results from this study, that can also serve as the basis for future research.

2 Related Work

This chapter reviews the theoretical and empirical work that forms the foundation of this study, as well as related research addressing similar problems. We first discuss the co-evolution of production and test code, then present an overview of empirical studies on code coverage. Next, we define code characteristics and survey commonly used code metrics, and discuss non-code characteristics and their relevance. We later review recent work on test coverage prediction and examine alternative approaches for reducing test bloat. Finally, we provide an overview of explainable machine learning models.

2.1 Co-evolution

When production code changes, developers often need to update the associated test methods to maintain effective quality control. This leads to the common practice of updating test cases either as changes happen or soon afterward. This phenomenon is often referred to as the co-evolution of production and test code. In practice, testing often happens in bursts. As a result, test code tends to lag slightly behind production code [10]. However, most developers update their test suite on a per-PR basis, with the most common other intervals being every few commits and every commit [20]. This suggests that most co-evolution data can be captured by analyzing projects on a PR level. This motivates the use of PR-level analysis in this thesis, as it is expected to capture the majority of co-evolution behavior in practice.

Detecting co-evolution pairs

Many studies detect production-test co-evolution by relying on common Java test naming conventions. In Java projects, unit tests for a source file usually appear in a corresponding test file with the same name plus the 'Test' suffix within the test directory. Researchers frequently use this mechanism.

However, this method is also often criticized across the literature. As not all developers strictly adhere to the naming conventions [12, 21, 22]. White et al. [21] found the recall of this method to range between 69% and 89%. This implies that a significant number of positive matches are missed. They implemented TCTracer, a method that combines and modifies multiple techniques to produce a ranking and score to determine co-evolution pairs. TCTracer provides a higher recall than solely relying on naming conventions, but also results in a lower precision. Liu et al. [12] make use of the call relationship of the test case.

Sun et al. [22] introduce the notion of production test non-co-evolution, describing cases where test and production classes change in the same commit (or soon after), but where the changes are independent. This reveals a limitation in co-evolution detection: methods based solely on temporal proximity can misidentify these cases as true co-evolution. To address this, they propose CHOSEN, which enhances the naming convention method by analyzing whether recent changes are true behavioral modifications rather than maintenance. Nonetheless, reliance on recent changes could still miss some complex dependency scenarios. Importantly, when examining small time intervals, this limitation lessens, as 88.66% of samples with simultaneous changes are true co-evolution cases.

These prior works generate co-evolution samples only at the file and method levels. This thesis addresses this limitation using a novel coverage-based approach to detect co-evolution samples.

2.2 Empirical studies on code coverage

Prior empirical studies show that code coverage varies between projects and between patches within the same project.

Hilton et al. [23] did a large-scale evaluation of code coverage changes on 7,816 builds of 47 projects. They built 29 projects themselves, for which they extracted coverage data from 5,360 builds and merged that with a dataset created by fetching data for 18 projects with 2,456 builds from the Coveralls API[1]. They found that patch coverage is not correlated with the coverage across the entire project. They also found that there was a large variability in patch coverage between patches within the same project, suggesting that there are certain non-project-dependent characteristics to patches that influence whether they get tested in practice. Furthermore, they also note a stark difference in the patch coverage profiles between projects, and since there was no correlation between project coverage and patch coverage, this suggests that certain non-PR-specific influences exist.

Hora studied which characteristics lead to code being explicitly excluded from test coverage analysis in 20 popular Python projects [24]. They used the pragma feature of coverage.py[2] to detect lines that were purposefully excluded from coverage calculations. They then used comments and commit messages to determine why code was not tested. The most common reasons were: the code was already untested, the code was low-level code, and the code was too complex. Other reasons were the code being deprecated, featuring parallelism, being trivial or behaving non-deterministically. Some of these characteristics might also influence which code gets tested.

These findings show that both project-level and local code factors influence coverage outcomes. This motivates closer examination of the context in which code exists.

2.3 Defining characteristics and metrics

As mentioned before, there are both code characteristics and non-code project characteristics. We will first discuss code-level characteristics and then project-related non-code characteristics.

Code characteristics

Fenton and Norman make a distinction between internal and external characteristics, where internal characteristics can be examined by studying the code without studying the behavior, while external characteristics study the behavior without studying the product [25]. Furthermore, they express the difficulty in measuring the attributes that we are interested in. They propose using a group of different metrics to approximate the characteristic.

Non-code project characteristics

Zhang et al. [26] did an empirical study on decisions for PRs, studying which properties influence the decision. They identified a set of 59 attributes, divided into the categories of the developer, the PR itself, and the surrounding project. This work focuses on data that can be gathered through existing API's or low-effort processing of the source code. These include things such as the presence and number of comments, PR description length, as well as the project age and total lines of code of the repository.

Khatami and Zaidman investigated the use of Automated Static Analysis Tools (ASAT), CI tools, and testing practices across 1,454 open-source projects [19]. They also propose a code review intensity metric, which normalizes the amount of comments and reviews by the number of line and file changes. They found that high code review intensity has a weak, positive correlation with code coverage, but did not find a correlation between the use of ASAT and CI tools with code coverage.

Zampetti et al. [27] investigated the effects of CI practices on the review process. They looked at both process factors as well as build factors. Process factors include many of the characteristics described above, such as comments and the number of changes. Additionally, they include the age of the PR before a decision is made, whether test code is included in the PR, and whether the author is a core member. For build factors, they use the number of builds for the PR history and

[1] https://coveralls.io/

[2] https://coverage.readthedocs.io

the percentage of failed builds. They found that the process-related features were enough to predict the acceptance of PRs.

Additionally, they found that when a build failure is discussed, the most common discussion point is test case failure, signifying the importance of the test suite to reviewers. The most prominent feature that was discussed was failures due to external dependencies, especially those that are hard to find. Other features that were often discussed were misconfiguration of the test suite, as well as a misalignment between the production and the test code.

These studies motivated the use of multiple code metrics as a proxy for code characteristics and the use of review-based non-code features.

2.4 Test coverage prediction

Recent work on test coverage prediction has focused on using machine learning models to capture relationships between code changes, structural characteristics, and testing behavior. We will discuss three recent studies, including both their major takeaways and their limitations.

SITAR

In 2021, Wang et al. [11] proposed a ML based approach called SITAR to facilitate the co-evolution of production and test code. SITAR focuses on predicting whether the unit test class should be updated when its corresponding production class is changed. SITAR makes use of the naming convention method discussed above to identify production-test co-evolution samples. Wang et al. argue that there is no single-factor heuristic that can predict whether or not a test class should be updated, and that a machine-learning-based approach is thus necessary.

SITAR is a model that learns the interrelations of changed code components using the within-project history and uses this to make predictions on added code. It extracts the number of lines added and removed from various language constructs in a file and transforms them to vector representations as the input data for the prediction. In their work, Wang et al. evaluate the model using only within-project history and using cross-project training on 20 open-source Java projects. They used recursive feature elimination to determine the relative importance of the characteristics. They found that adding parameter list lines was the most important feature for prediction, with using just that feature resulting in an accuracy of nearly 70%.

DRIFT

Liu et al. [12] had some criticism of the SITAR method, mostly on the coarseness of the method and on the fact that SITAR relies on naming conventions to determine matches. This led to the creation of DRIFT in 2023. DRIFT allows the prediction to happen on a method-level granularity, showing exactly which test case needs to be updated. Furthermore, they improve the matching between production and test changes by extending the patterns that match the file name. Furthermore, they refine the matches by looking at which production modules are imported in the test files.

DRIFT also uses the number of lines added and removed using the abstract syntax tree (AST), similarly to Wang et al. [11]. Liu et al. [12] evaluated the model using solely the project history as well as cross-project training on 40 projects. DRIFT generally outperforms SITAR on precision, recall and F1 score. While DRIFT does improve the prediction quality on SITAR, its methods do not produce an importance ranking of the features.

Brandt and Ramirez

In 2025, Brandt and Ramirez proposed a method to predict which lines should be covered by a test case [17]. A large difference between their work and DRIFT and SITAR is that the work focuses on the code base as a whole, analyzing all lines in a single commit, instead of focusing on changes. Their approach combines line and method-level characteristics using JaCoCo to determine whether lines should be covered, CK [28] to extract method-level statistics and tree-sitter to analyze the AST. They found that the features of the method the line belongs to were quite relevant to the classifier, while the line's individual features were less relevant.

Overall, prior work indicates that effective test coverage prediction depends on combining multiple features that capture both code changes and their surrounding context. In particular, the importance of method-level characteristics observed by Brandt and Ramirez suggests that the broader structural context of code plays a significant role, while existing models remain limited by coarse representations, matching heuristics, and limited insight into which contextual factors are most influential.

2.5 Alternatives for reducing test bloat

Ivankovic et al. [29] propose an alternative to conventional test coverage called productive coverage. It works by detecting similar code and then accumulating the coverage results for the similar fragments. The idea is that it is better at identifying crucial pieces of code with a risk of introducing defects, which can then be selected to be tested. This could thus potentially result in lower test bloat, as testing low-risk code is mitigated.

Yoo and Harman [30] extensively discuss the three primary methods to reduce test bloat that were also discussed in Noemmer and Haas [15]. We will briefly discuss the methods of test selection, test prioritization and test suite minimization.

In the case of test selection, all algorithms are based on identifying tests that execute code that is changed or execute code that is influenced by other changed code. There are a large number of different approaches that aim to tackle this problem. Yoo and Harman identify methods using: Integer Programming, data flow analysis, symbolic execution, dynamic slicing, CFG graph-walking, textual difference in source code, SDG slicing, path analysis, modification detection, firewall, CFG cluster identification and design-based testing [30].

Test code prioritization works by ordering the tests in a way such that failures happen early. The idea behind this is that if testing is stopped prematurely, we will still have received as much information as possible. Fault detection information is often not known preemptively, so to mitigate this fact, methods make use of surrogate properties. Most prioritization models actually aim to get early coverage, as coverage

is seen as the best indicator of early fault detection. Another surrogate that is often chosen is test dissimilarity, the idea being that you can cluster similar tests together and then prioritize tests from different clusters. Using this method, a large number of different behaviors are tested early, before running other similar tests.

Test suite minimization is most often achieved using greedy approaches, which function by finding a subset of tests that still fulfill the original requirements by repeatedly picking the test that satisfies most of the remaining requirements [31]. Marking tests that are not needed to create a subset of redundant tests, which are safe to remove. There also exist heuristic-based methods, which reason the other way around, where you keep tests that cover requirements that are only covered by that respective test [32].

2.6 Explainable machine learning models

Seeing as we are not just interested in whether we can predict test changes from production changes, but also want to say something about the feature importance, it is important that the created model is interpretable. First, we will discuss what model interpretability entails exactly, then afterward we will dive into the interpretability of the machine learning methods used in the works of Wang et al. [11], Liu et al. [12], and Brandt and Ramirez [17]. Then we will discuss some methods to produce interpretable models from black-box models using model-agnostic methods.

Defining interpretability

So what is model interpretability, often also called explainability? Miller formulates model explainability as: "the degree to which an observer can understand the cause of a decision" [33]. There are two types of interpretability: local interpretability and global interpretability. They differ in the scope of their explanation. A local explanation can for a single outcome explain why the result was formed, but can not say anything about all other outcomes. If an explanation applies to all outcomes, it is a global explanation [34]. A good example can be the neighbors in the k-Nearest Neighbors model. The labels of the neighbors are an explanation for that one node, but do not explain the global feature influences.

As we are interested in answers as to how features influence the predicted target for all outcomes, we are interested in global explanations. There are two main categories of models: glass-box and black-box models. Glass-box models are inherently interpretable, while black-box models are not. However, these models can be made interpretable after training using a large number of techniques [35–41]. These techniques give either the feature importance or a breakdown of how different values for a feature influence the outcome.

Interpretability in practice

Both the DRIFT [12] and SITAR [11] papers evaluate using the same machine learning methods: Logistic Regression, Random Forest, Gradient Boosting, and Naive Bayes. DRIFT additionally adds the Support Vector Machine method to the methods for classification. Brandt and Ramirez also use the Random Forest and Naive Bayes models as well as a Decision Tree and k-Nearest Neighbors model. Some of these methods offer inherent explainability.

The most interpretable model is the Decision Tree model, as this provides hard rules in a human-readable format. Logistic Regression models are also highly interpretable, as they output a formula containing coefficients that can be used to determine the feature importance, as well as whether they have a positive or negative effect. Another interpretable model is Naive Bayes, as it produces prior class probabilities as well as per-feature likelihoods, allowing the model to be interpreted. All of these models are glass-box models, as described in 2.6.

Random Forest, k-Nearest Neighbors, Support Vector Machine and Gradient Boosting are not interpretable on their own; they are so-called black-box models. SITAR [11] extracts feature importance using recursive feature importance as described in [41]. Brandt and Ramirez [17] extract feature importance using the importance permutation method [40].

Another common way to gain a better understanding of the working of a model is Friedman's Partial Dependence Plots [35]. These plots show the impact of a selected set of features on the outcome of the model when setting the rest of the features to marginal values. While these do not give the full importance, they can say something about how outcomes change when certain input features change.

3 Methodology

In this chapter, we describe the experimental setup used to investigate which code and non-code characteristics can be used to predict whether production code changes should be covered by tests. Our experimental setup can be split into three stages: Data collection, feature extraction and model evaluation. In the data collection stage, we make a selection of the projects and PRs we want to analyze. In the feature extraction stage, we extract the useful PR metrics and project metrics. In the evaluation stage, we tune different machine learning models and evaluate the predictive performance of the different models. An overview of the entire experimental setup can be found in Figure 2.

3.1 Data collection

We first set some selection criteria, which we afterward integrated into a pipeline using the GitHub GraphQL and GitHub REST API to create our final set of projects and PRs. A summary of our project selection can be found in Figure 1.

Setting the project criteria

To construct a meaningful and reliable dataset for answering our research questions, our goal was that the selected projects (i) enable consistent and feasible data collection, (ii) avoid data duplication, (iii) represent mature projects with substantial developer interaction and activity, and (iv) contain enough data points to be able to run reproducible analysis.

As the projects needed to be built and their test suites executed, we aimed to simplify our data collection. To simplify our data collection, we restricted our analysis to projects written in Java and using Gradle as their build system. Java provides a large ecosystem of well-maintained open-source projects and is commonly used in related research, while focusing on Gradle projects allows for a uniform approach to

collecting test coverage data without the need to support multiple build systems.

To prevent data duplication, we excluded forked repositories. Forks often share a substantial portion of their history with the original repository, which could otherwise lead to redundant data in our dataset.

To ensure that the selected projects reflect realistic and collaborative development environments, we focused on repositories with a sufficient level of activity and team size. We only included projects with at least 100 PRs and more than 11 contributors. Zhang et al. [26] classify projects with more than ten contributors as large-team projects, which are more likely to encounter challenges such as coordinating test changes and avoiding test bloat.

Finally, to ensure that each project contributes a meaningful number of relevant observations, we applied several PR–level filters. We only considered merged PRs created between 2023 and 2026 that modify at least one production file. Restricting the time frame reduces the likelihood of build and dependency incompatibilities when collecting coverage data. Requiring production code changes ensures that each PR is relevant to our research focus on the relationship between production and test code. Considering only merged PRs reflects developer-approved changes, meaning that uncovered lines represent implicit decisions that test coverage was not required. Additionally, we required at least 20 such qualifying PRs per project to ensure sufficient data for within-project analysis.

Creating the dataset

Based on the previously defined selection criteria, we constructed the dataset using a multi-step data collection pipeline combining data from both the GitHub GraphQL and REST APIs, as shown in Figure 1.

We first queried the GitHub GraphQL API to retrieve all Java projects with more than 50 stars, recent activity on the main branch after September 1st 2025, and excluding forked repositories. For each project, we collected metadata including the repository name and owner, total number of PRs, number of merged PRs, and the presence of either a `build.gradle` or `build.gradle.kts` file in the root directory of the most recent commit.

Next, we identified Gradle-based projects by checking for the presence of a Gradle build file. Projects without either a Groovy- or Kotlin-based Gradle file were excluded. We then filtered projects based on PR activity using the previously retrieved metadata. Contributor information was subsequently retrieved using the GitHub REST API, after which projects with fewer than 11 contributors were removed.

For each remaining project, we retrieved all PRs created between January 1st 2023 and January 1st 2026 that had been merged. For each PR, we collected the file paths of all modified files. These files were then classified using a name-based heuristic. Files located in the `src/test` directory or with names ending in `Test.java`, `Tests.java`, or `IT.java` were classified as test files. Files ending in `.java` that were not identified as test files were classified as production files, while all remaining files were considered non-code files.

PRs were then filtered to retain only those containing at least one production file change. Finally, projects with fewer than 20 qualifying PRs were excluded. The resulting dataset consists of 756 projects and 140,691 PRs, which form the basis for the experiments conducted in this study.

3.2 Feature selection

Inspired by the work of Briand et al. [42], we aimed to select metrics that can be used as proxies for size, complexity, coupling and cohesion.

Metric Family	Line	Method	Class	Repo.
CBO		✓	✓	Average
Fan-in		✓	✓	
Fan-out		✓	✓	
WMC		✓	✓	Average
RFC		✓	✓	Average
LCOM			✓	Average
LOC		✓	✓	✓
DIT			✓	
Coverage percentage		✓	✓	✓
Covered lines	✓	✓	✓	✓
Operator count	✓			
Unique operator count	✓			
Rel. position in file	✓			
Line length	✓			
Dependency count				✓
Sub module count				✓
Comment lines				✓

Table 1: Overview of metric families and the scopes at which they are used.

To construct a feature set that captures both structural and semantic properties of code changes, we build on metrics proposed in prior work, particularly the method-level features introduced by Brandt and Ramirez [17], as they found promising results using these metrics. These include object-oriented and size-related metrics such as method start line, Coupling Between Objects (CBO), modified CBO, fan-in, fan-out, Weighted Methods per Class (WMC), and Lines of Code (LoC).

We adopt this set as a foundation, as it provides a well-established representation of method-level structure and dependencies. However, instead of using a binary indicator for the presence of JavaDoc, we quantify documentation more precisely by measuring the number of comment lines, including both JavaDoc and inline comments. This provides a more fine-grained representation of documentation density. In addition, we include the line length as proposed by Brandt and Ramirez, as it captures aspects of code readability and local complexity.

To better capture fine-grained characteristics of code changes, we extend the feature set with additional line-level and method-level metrics. At the line level, we include the line number within the file, as well as the total number of operators and the number of unique operators, which reflect local syntactic complexity. At the method level, we also include RFC.

Additionally, we selected a number of features on the class level. These metrics describe inheritance structure, size, cou-

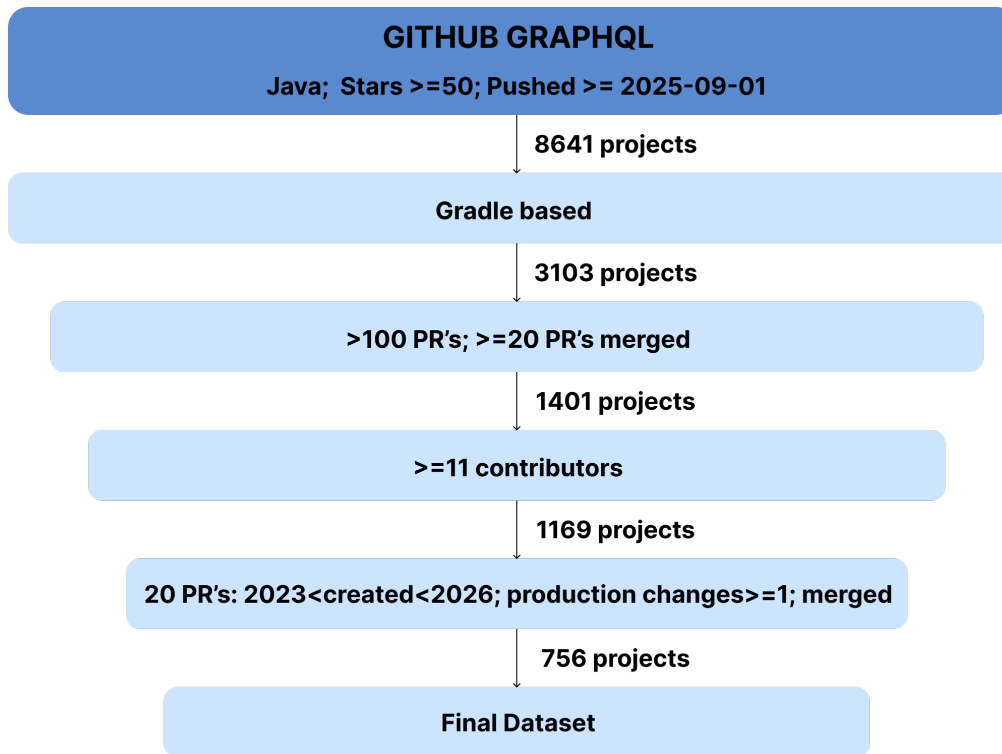


Figure 1: Overview of the data collection process, including the number of projects left after each step.

pling, and cohesion at the class level. Most of these features are shared with the method-level features, as can be seen in Table 1. As the features performed well on the method level in the work from Brandt and Ramirez [17], we assume these features could also be beneficial to use on the class level. We expand these class features with the Depth of the Inheritance Tree.

At the repository level, we incorporate aggregate metrics to capture project-wide properties. These include total Lines of Code (LoC), average class-level RFC and WMC, average CBO, average LCOM, and total number of comment lines. These aggregated features reflect overall system size, complexity, and maintainability, enabling the analysis to account for differences between projects. We also include the number of dependencies on other projects and the number of submodules.

Finally, we incorporate historical test coverage information to capture the prior testing state of the code. Specifically, we include whether a given line was already covered by existing tests, as well as the percentage of covered lines within its surrounding scope prior to the changes introduced in a PR. These features provide contextual information about existing test adequacy, which may influence the likelihood that additional tests are added or modified. Furthermore, we assume that lines that were already covered will more often require test changes, as modifying the behavior of a line that is covered will have a high chance of modifying the test outcome.

Lastly, we are interested in answering questions regarding the influence of the non-code characteristics on the likelihood of test suite modification. This includes both metrics related to the singular PR we are currently analyzing, as well as metrics that describe the project as a whole. These metrics are more focused on determining the influence of types of development, how engaged the contributors are, but also the influence of the number of people collectively working together. Furthermore, it is focused on CI and testing practices.

We believe that products with a well-defined CI process, including large amounts of automatic tests, intensive code reviews, and the presence of a CI pipeline, would increase the chance of lines being covered. Similarly, we believe that more people working together on a project will lead to more testing taking place, and thus there being a higher chance of line coverage.

For this reason, on the PR level we picked the features: contributors to the code in the PR, the amount of total comments, the amount of issue comments, the amount of review comment (these are inline comments on the code level), the amount of reviews (this is the amount of review actions opened, which can contain multiple review comments or can be the acceptation or rejection of the PR), the amount of unique people who left a comment, the amount of code suggestions, the amount of commits related to the PR, the time between the creation and merging the PR, the amount of changed lines, the amount of additions and the amount of deletions.

On the project level, we picked the following features: the number of PRs created and merged between 1-1-2023 and 1-1-2026, the age of the project, the number of people who contributed to the project, the presence of a CI pipeline, the number of stars, and the number of forks of the project.

3.3 Feature extraction

For the historical coverage and external dependency metrics, as well as determining co-evolution, the projects needed to be built and their test suites needed to be executed. We first verify if this works for the first PR of the projects, Figure 2 step 1. If this is not the case, we discard the project as being unfeasible Figure 2 step 2. As not all projects and PRs compile with the same JDK version, we need to determine the working JDK version Figure 2 step 3. To do this, we implemented a JDK version fallback. We start with versions 17-21, then we check versions 22-25 and finally we check versions 8-17. The reasoning is that most builds use a recent but not the most recent JDK version. When we try to determine the feasibility of using a JDK version, we run the build command. This ensures that all dependencies can be resolved and the project can be compiled using the current JDK version.

Although developed independently for this work, our JDK fallback strategy is consistent with recent findings that incorrect Java versions are a major source of build failures and that attempting multiple JDK versions can recover a substantial fraction of otherwise failing builds [43].

To speed up the process, we implemented two methods to reduce the number of versions probed. The first is that if a PR was successfully processed using a JDK version, we will use that as the initial input for the next PR. Making use of the fact that neighboring PRs will likely share the same JDK version used. Additionally, we inspect the error for errors related to major version issues and if this is the case, we will use the corresponding JDK to the major file version.

An overview of the whole extraction pipeline can be seen in Figure 2.

Detecting co-evolution

Then we have to determine whether the line belongs to the positive class (it is a line that is covered by the modified tests) or to the negative class (a line that was not covered by the modified tests). We do this by running just the modified tests while generating coverage using the JaCoCo framework[3] Figure 2 step 5. As not every project is configured in the same way, we inject the JaCoCo dependency into the build file, ensuring that it also gets injected into the dependencies of any submodules.

Then, to ensure support for projects with unconventional module layouts (for example, where test cases are not located in the same module as the production code), we set the source material equal to all classes while generating the coverage report. Additionally, we ignore failing test cases and also ignore them if no matching tests are found. We also disable treating warnings as errors and disable predictive test selection. This is needed, as some projects have these automatically enabled when running the test command.

We generate a coverage file per test file, wiping all test execution data between tests to ensure no pollution of the data takes place. Then, at the end, we parse all the generated XML files, resulting in a mapping between the test file name and a list of production file and line number tuples. Then, for all of the lines that were extracted from the patches, we check if their file name and line number are present in any of the test file mappings, and if so, consider them to be a positive sample and otherwise a negative sample.

Unfortunately, this methodology does not work for all projects. One of the common issues we ran into was an incompatible Gradle version. To circumvent this, once the coverage collection failed, we tried running the coverage collection again while changing the Gradle wrapper version to 8.14.3. This version was chosen as it was the most recent version that is not Gradle 9. Gradle 9 became much stricter with enforcing compatibilities and removing deprecated features.

Unfortunately, not all the issues could be fixed this way. This resulted in a lot of projects that could either not build, not run the tests, or where the tests resulted in 0 lines covered. This is consistent with the work of Khatami et al. [19], who found that they could build 56% of Gradle-based projects and were only successful in generating full coverage results in 28.7% of the cases.

There were a large number of reasons for these projects failing to build. The most major ones were broken dependencies, requiring manual actions in the testing process, requiring software to be installed on the PC, broken toolchain repositories and rare project layouts.

Code metrics

We make use of the CK tool [28] to extract object-oriented metrics on the class and method level, similarly to the method used in Brandt and Ramirez [17], Figure 2, step 6. We extended the CK tool to offer the end line of methods and the start and end lines of classes. This allows us to precisely determine to which class and method a changed production line belongs. Additionally, we extended the tool to also add the number of commented lines to the class and method metrics. Furthermore, CK normally separates classes, inner classes and anonymous classes. We modified the tool to merge these results. For the additive metrics such as WMC, RFC, fanin, and fanout, the sum of all inner classes is simply used, and for the non-additive metrics, LCOM, DIT, and CBO, the maximum value is used.

Normally, CK does not provide any repository-level metrics. To be more efficient, we implemented a feature to provide the LoC and number of comments for the full repository. This cannot simply be done by aggregating all the data from classes and methods, as some code does not belong to any class or method, such as the import statements. We then process the method and class metrics to extract the average repository WMC, RFC, CBO and LCOM.

For the number of external dependencies, we ran a Gradle script that counts the declared dependencies across all submodules Figure 2 step 7. We split this into internal and external dependencies. We purposefully only include declared dependencies instead of the size of the resolved dependency graph, as the resolved dependency graph will also measure all dependencies of external dependencies recursively. Our assumption is that the dependencies of the external dependency have no influence on the likelihood of test coverage, thus we ignore those. Additionally, we extract the historical coverage data by running the full test suite, with predictive tests turned

[3] https://github.com/jacoco/jacoco

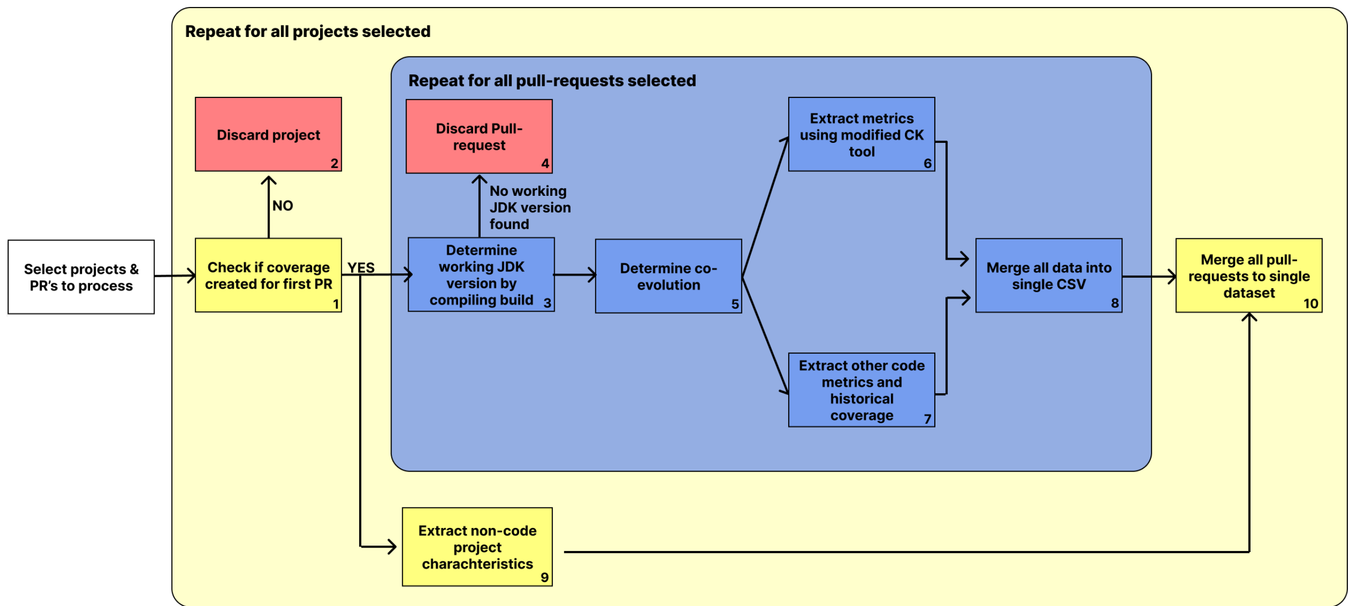


Figure 2: Overview of the feature extraction pipeline.

off. This results in an XML file containing the coverage data.

After creating all the metrics, we extract all modified and added lines from the PR patches and merge them with their corresponding class and method, making use of the start and end line numbers of classes and methods and the line number of the production line. Additionally, we merge the line with the repository characteristics. This results in a table of lines with all their line, method, class, and repository features Figure 2, step 8.

We remove all lines that do not belong to a class or are comment lines from the dataset. Furthermore, we remove lines of code that solely consist of "}", as these are not meaningful lines of code and pollute the dataset.

Non-code metrics

For all PRs for which data could be extracted, we fetch the data described in Figure 3.2 using the GitHub GraphQL API Figure 2, step 9. We request the general PR data such as the creation dates and changed file counts. We additionally request all review threads, comments and commits related to this PR. This allows us to get the number of unique commenters and the commit count, as well as the amount of code suggestions.

For all projects that resulted in a dataset with at least 100 positive and negative samples, we extended the dataset by fetching the respective project data using the GitHub GraphQL. We normalized the PR creation and merging rates to a monthly rate.

3.4 Creating the dataset

Because we need sufficient samples from both the positive and negative classes, and because we need variance in the PRs that the samples come from, we set 4 additional parameters to select viable projects. Those being that there need to be at least 100 samples for both classes and that there need to be 3 unique PRs containing samples from the different classes. The limit of at least 3 PRs is chosen to support stratified grouped cross-validation, as is further discussed in section 3.6.

Additionally, a limit of at least 100 samples per class was chosen to ensure enough data exists per class, as our high-dimensional dataset requires many samples to reduce the risk of the curse of dimensionality. This is even more important as several projects showed substantial class imbalance. For example, in the *spring-framework* project, only 24% of the extracted samples were labeled as positive co-evolution samples. The requirement of at least 100 samples per class therefore helps ensure adequate representation of both classes across the selected projects.

3.5 Data pre-processing

The quality of the conclusions one can make about the output of machine learning models is in a large part dependent on the quality of the input data. It can make the work more interpretable, generalizable and can reduce noise that can lead to false conclusions.

One approach for improving data quality is the removal of redundant features. Feature reduction can improve model interpretability and efficiency while mitigating the curse of dimensionality [44]. Since interpretability is a major objective of this work, and the dataset contains a large number of features extracted across multiple scopes, feature reduction provides a useful mechanism for identifying the most informative characteristics.

Another important method is scaling features. Using non-normalized features can lead to features with larger variance and mean dominating the learning process. As the extracted characteristics range from single-digit values to tens of thousands, scaling helps place the features on a comparable scale and ensures a fairer evaluation of their predictive value.

Removing correlated features
We analyzed both the correlation between the features and the target label as well as the correlations between the features. An unimportant feature is categorized as having redundancy (a high correlation between the input features) or low relevance (a low correlation with the target label).

We used the Spearman correlation to determine the within-feature correlations as well as the correlation with the target label. We set a threshold of redundancy at 0.8. This means any feature pair with a correlation higher than 0.8 is marked as redundant, leading to one of the pair being removed. We choose the feature with the lower correlation with the target label, as that is determined to be the less relevant feature of the two.

Normalizing metrics
We scaled all continuous non-boolean features except the line number using Z-score normalization, resulting in the features having 0 mean and a standard deviation of 1. We scaled the line number by normalizing it relative to the file length of the class that it is in. This resulted in a value between 0 and 1, showing its relative file position. We do this as the relevant information we want to retain is the position in the file. If we applied Z-score normalization, this information would not be retained in the scaled data.

3.6 Evaluation

We evaluated our predictive model in a within- and cross-project setting. For this, we use precision, recall, Matthews correlation coefficient (MCC) and Area Under the Receiver Operating Characteristic Curve (ROC-AUC). We then compared the performance of our methods to each other as well as to two baselines using the Wilcoxon signed-rank test. The first baseline randomly assigns predictive targets based on the distribution of the class labels. The second baseline assigns targets based on whether the line was covered before. If a line is already covered by the test suite, it is predicted that this line will have co-evolution in this PR. Note that for the random baseline, an MCC score of 0 is expected.

We use MCC, which is a balanced classification metric that takes true and false positives and negatives into account. MCC is particularly suitable for imbalanced datasets, as it provides a more reliable indication of overall classification performance than accuracy alone. MCC values range from -1 to 1, where 1 indicates perfect prediction and -1 indicates perfectly predicting the opposite outcome.

In their work, Zampetti et al. [27] define the meaning of different MCC scores. Their definitions are summarized in Table 2. We will use these ranges to interpret our results.

MCC Range	Interpretation
< 0	Worse than random guessing
$= 0$	Similar to random guessing
$0 < \text{MCC} < 0.2$	Low correlation
$0.2 \leq \text{MCC} < 0.4$	Fair correlation
$0.4 \leq \text{MCC} < 0.6$	Moderate correlation
$0.6 \leq \text{MCC} < 0.8$	Strong correlation
≥ 0.8	Very strong correlation

Table 2: Interpretation of MCC scores based on Zampetti et al. [27]

MCC is mathematically equivalent to the Pearson correlation between the predicted value and the class label of the target, as can be seen in formula 1.

$$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (1)$$

We evaluated our ability to predict coverage using the Random Forest, Naive Bayes, Logistic Regression and Gradient Boosting models as implemented in sklearn. For the within-project predictions, we used a stratified grouped 3-fold cross-validation, grouped on PR number. This ensures that all lines from the same PR are in either the test or train set, preventing data leakage. As lines in a PR share similar metrics, as they often share the same class, method and repository statistics, the model would learn how to discern to which PR the line belongs. In conjunction, if some lines in a PR are covered, it is much more likely that the other lines in the PR are also covered. Thus, not using grouped cross-validation would lead to an overestimation of the model strength. We use stratification to obtain a roughly equal class distribution across the train and test set. Due to the grouping taking place, the distributions are not entirely equal, which is especially true in smaller datasets. The choice for k=3 was made due to the limited amount of PRs available. Raising K could lead to some projects having a majority of the folds not containing any positive samples, even when using stratification. For the cross-project setting, we used all samples of all other projects for the training set and all samples from the project we wanted to evaluate as the test set.

Feature importance
We used two methods to determine feature importance. The first method uses the internal feature importances in the chosen model. These importance scores are computed using the Mean Decrease in Impurity (MDI), which measures how much a feature contributes to reducing classification impurity. The importance scores are then normalized such that the sum of scores is equal to 1, meaning that the score corresponds to the relative contribution of that feature.

Then, we applied Recursive Feature Elimination (RFE) to analyze the impact of different feature subsets on predictive performance. This analysis was performed for all repositories in both the within-project and cross-project settings. First, the features were ranked based on the feature importance scores obtained using the method described above. Subsequently, two types of feature subsets were evaluated. The first subset contains the top k most important features, where k ranges from 1 up to the number of features with an importance score greater than 0.001. The second subset consists of the bottom k features using the same values of k. This approach allowed us to evaluate both the predictive performance obtained when using only the most important features and the performance impact of excluding those features from the model.

Furthermore, we analyzed the performance using all features, as well as the performance from subsets of features. The subsets used were line features, method features, class features, repository features, all features except the non-code

features (only-code) and all features except the coverage-based features (non-coverage). All sets except only-code also included the non-code features and all scoped sets include their respective coverage features. This allowed us to determine the performance of different feature scopes. Additionally, we analyzed the top 10 most important features using Partial Dependence Plots, to determine how the features influence the model.

4 Results

To answer the research questions posed in this thesis, we conducted our experiment as described in the previous chapter. This chapter will present the results from these experiments. First, we will describe the created dataset, as well as analyze differences between projects we were and were not able to extract coverage from. Then, we analyze the dataset, investigating class imbalance as well as data distribution. We also discuss the feature correlations in our dataset and their effect on prediction quality. Then we provide an overview of the predictive performance of our method. Lastly, we discuss the results from the feature importance experiment.

4.1 Dataset creation

Due to time constraints, we were only able to extract the data from 50 out of the 84 viable projects. From these 50 projects, we were able to build and extract all metrics for at least 1 PR for 33 of these projects. From these 33 projects, we were able to extract sufficient data from 18 projects. The reduction of viable data is discussed in Section 3.3. This resulted in a dataset that consists of 18 projects with a total of 1303 PRs consisting of 72534 lines.

We first analyzed whether projects with more stars or contributors were more likely to result in at least one successful build. To do so, we compared projects with and without a successful build using a Welch t-test. The test did not provide sufficient evidence to reject the null hypothesis, yielding p-values of 0.3 and 0.5 for stars and contributors, respectively. The distributions of both the processed and failing projects with regard to the number of stars and contributors can be found in Figure 3.

We additionally investigated whether stars and contributors were related to two continuous project-level outcomes: the percentage of PRs that could successfully be extracted and the prevalence of co-evolution within a project. Using Pearson correlation, we found all correlations to be below 0.1, suggesting that stars and contributors are not strongly associated with extraction success or co-evolution prevalence. This reduces the likelihood that the resulting dataset is systematically biased toward larger or more popular projects.

4.2 Dataset analysis

The final resulting dataset size and positive rate per project can be seen in Table 3. The minimum number of lines that could be extracted was 902, while the maximum number of samples extracted was 12832. The rate of positive samples ranges from 6.6% to 40%, showing a wide gap in testing practices across projects consistent with the findings of Hilton et al. [23].

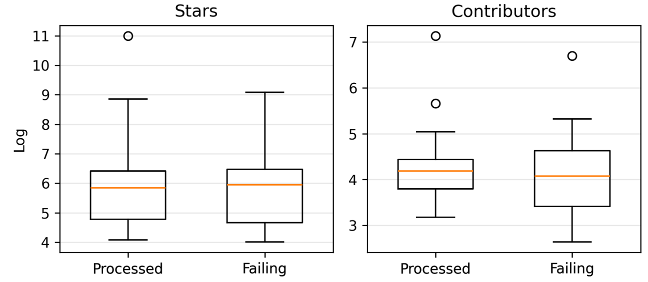


Figure 3: Star and contributor distribution of processed and failing projects.

Project	Samples	Positive Rate
tiered-storage-for-apache-kafka	7218	35%
netcdf-java	1403	26%
tds	1382	9%
allure-java	1756	57%
artio	839	15%
extender	4387	7%
dwh-migration-tools	8021	18%
inspectit-ocelot	1900	38%
junit-framework	3550	35%
sechub	2285	6%
fixture-monkey	22383	15%
tritium	564	40%
spring-amqp	1762	10%
spring-framework	1072	24%
springwolf-core	1482	32%
substrait-java	3354	36%
twitch4j	4406	5%
WALA	4770	9%

Table 3: Project sample statistics

We found co-evolution not being uncommon, with a weighted average of 38.2% of PRs containing co-evolution, with a standard deviation of 21.5%. A breakdown of per-project PR distribution can be found in Table 4. The table shows that a majority of PRs do not contain co-evolution. The table shows that most projects have between 30% and 60% of their PRs containing co-evolution; however, some projects have much fewer examples of co-evolution. The Extender project, for instance, has less than 10% of its PRs containing co-evolution.

Feature correlations

Figure 4 shows the correlations between the target variable, whether a line is covered by tests in the PR and the extracted features. Historical coverage exhibits the strongest correlation with the target variable, with a Pearson correlation coefficient slightly above 0.5. This indicates a moderate positive relationship between prior coverage and future test-suite modifications. In practice, this suggests that code that was already covered by existing tests is substantially more likely to receive additional or modified test coverage in subsequent PRs.

Project	Positive PRs	%
tiered-storage-for-apache-kafka	87 / 134	64.93%
netcdf-java	18 / 35	51.43%
tds	4 / 27	14.81%
allure-java	17 / 23	73.91%
artio	7 / 20	35.00%
extender	8 / 92	8.70%
dwh-migration-tools	68 / 182	37.36%
inspectit-ocelot	17 / 23	73.91%
junit-framework	53 / 84	63.10%
sechub	8 / 15	53.33%
fixture-monkey	94 / 238	39.50%
tritium	6 / 14	42.86%
spring-amqp	9 / 32	28.12%
spring-framework	24 / 71	33.80%
springwolf-core	21 / 36	58.33%
substrait-java	25 / 40	62.50%
twitch4j	8 / 97	8.25%
WALA	23 / 140	16.43%

Table 4: PR statistics per project

The result also implies that historical coverage alone already contains considerable predictive power when the features are evaluated independently. This finding aligns with the intuition that developers are more likely to extend tests around code that is already part of the tested codebase, while previously uncovered code may remain untested.

At the same time, a correlation of approximately 0.5 also indicates that historical coverage alone is insufficient to fully explain the co-evolution patterns. While it is clearly the strongest individual predictor, a substantial portion of the variance remains unexplained. This leaves room for additional code and non-code characteristics to contribute complementary predictive information.

Figure 5 shows a heatmap showing the correlations between all features used. There are clear regions of high correlation, these are the correlations between features in the same scope. For instance, the class features have a high correlation with the other class features. Some of the features had a correlation above our threshold and were thus removed when evaluating to keep the model as simple as possible. These features are: the number of prod files changed, the LoC of the method, the WMC of the class, the amount of comments on a PR, the LoC of the class, the amount of operators in the line, the amount of comments in a review, the number of changed lines and the class fanout.

4.3 Cross-Project and Within-Project Predictive Performance Evaluation

Model selection

To decide which model should be used for our predictions, we evaluated the four models, Linear Regression, Naive Bayes, Random Forest, and Gradient Boosting, by generating a PR-curve on the full dataset as well as on the individual projects. As the dataset exhibits a substantial class imbalance, PR curves provide a more informative evaluation metric than accuracy-based metrics.

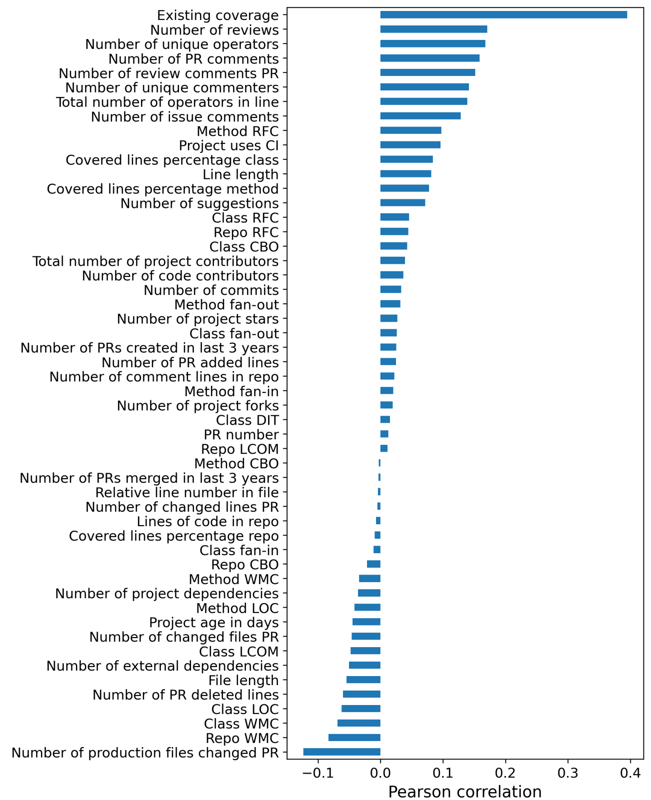


Figure 4: Correlations between features and the target feature.

Figure 6 shows that the Random Forest and Gradient Boosting classifiers substantially outperform the other evaluated models across nearly the entire recall range. Notably, performance for Naive Bayes and Logistic Regression experiences significant drops in precision starting at very low recall values. This indicates that these models have some samples that they can strongly separate, but then experience difficulty separating the noise from the data for all other samples.

In contrast, Random Forest and Gradient Boosting show a considerably more stable Recall-Precision trade-off. Furthermore, they substantially outperform the other evaluated models across nearly the entire recall range. As Gradient Boosting slightly outperforms Random Forest and is more stable across the different within-project PR-curves, we decided to use the Gradient Boosting model for the further analysis.

Repository performance

The within-project results shown in Table 5 demonstrate that the model achieves generally moderate predictive performance across the evaluated repositories, with an average MCC of 0.357 and an average ROC-AUC of 0.805. The MCC score indicates a fair-to-moderate correlation between the prediction labels and the target labels. The recall being much higher than the precision shows that the model is effective at retrieving the positive samples, but also results in many false positives.

The performance varies wildly across the different repositories. The strongest results are observed for repositories

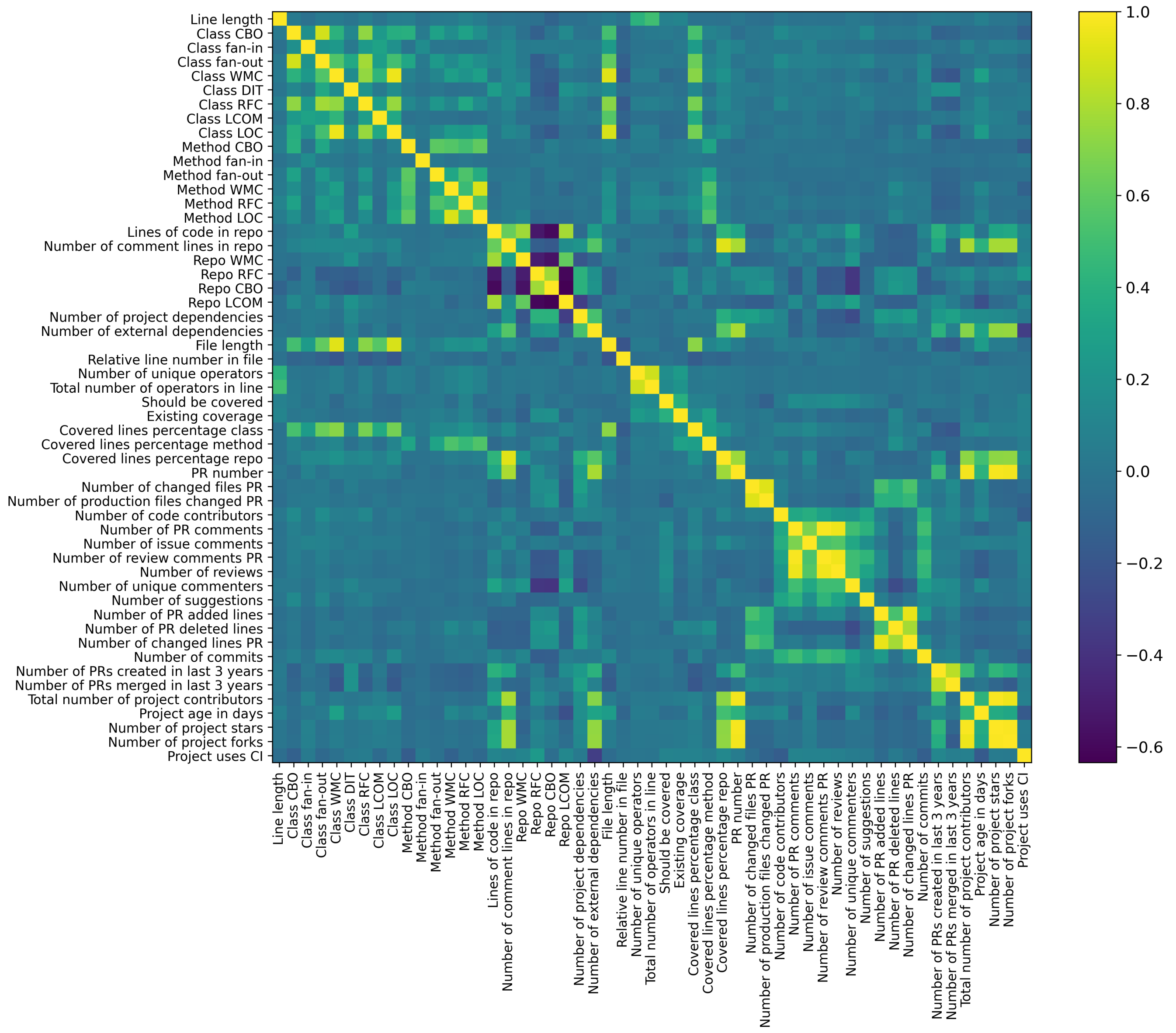


Figure 5: Between-feature correlation heatmap.

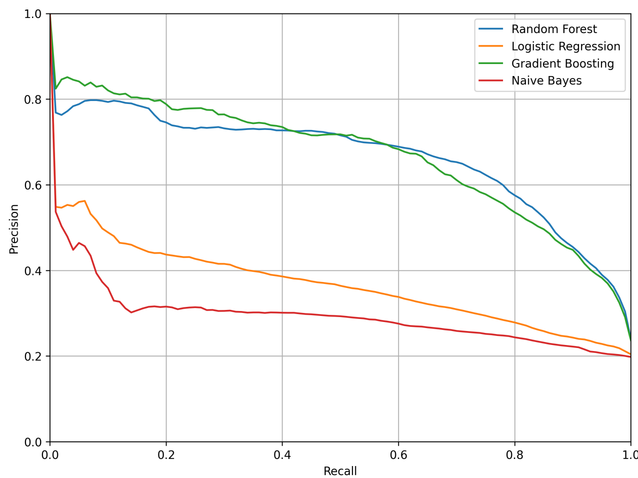


Figure 6: PR curves for the four different machine learning methods on the aggregated dataset

Repository	Precision	Recall	MCC	ROC-AUC
spring-framework	0.533	0.847	0.495	0.831
tds	0.073	0.667	0.094	0.765
substrait-java	0.652	0.768	0.498	0.886
tiered-storage-for-apache-kafka	**0.684**	0.810	**0.592**	0.879
inspectit-ocelot	0.564	0.641	0.262	0.716
netcdf-java	0.460	0.435	0.200	0.685
allure-java	0.634	**0.904**	0.508	0.788
artio	0.307	0.652	0.259	0.828
sechub	0.572	0.629	0.492	0.877
fixture-monkey	0.464	0.767	0.492	**0.898**
spring-amqp	0.154	0.376	0.082	0.583
springwolf-core	0.475	0.599	0.383	0.834
twitch4j	0.032	0.667	-0.004	0.793
wala	0.177	0.548	0.235	0.847
junit-framework	0.612	0.877	0.542	0.839
dwh-migration-tools	0.545	0.624	0.479	0.869
extender	0.390	0.550	0.407	0.708
tritium	0.574	0.563	0.412	0.868
Average	**0.439**	**0.662**	**0.357**	**0.805**

Table 5: Within-project performance per repository. Maximum values per metric are bolded, while minimum values are underlined.

such as *tiered-storage-for-apache-kafka*, *junit-framework*, *substrait-java* and *allure-java*, which all achieve MCC scores close to or above 0.5, indicating moderate predictive performance. Simultaneously, repositories such as *twitch4j*, *spring-amqp* and *tds* achieve very low MCC, some even negative, indicating a worse than random predictive performance.

The cross-project results shown in Table 6 demonstrate that the model is capable of generalizing across repositories, achieving an average MCC of 0.454 and an average ROC-AUC of 0.839. Here, the MCC score indicates an overall moderate correlation between the predicted labels and the target labels. Similar to the within-project setting, the recall is substantially higher than the precision. This indicates that the model is effective at retrieving positive samples, but simultaneously produces a considerable number of false positives.

The predictive performance again varies substantially across repositories. The strongest results are observed for repositories such as *allure-java*, *twitch4j*, *junit-framework*,

Repository	Precision	Recall	MCC	ROC-AUC
spring-framework	0.552	0.737	0.501	0.860
tds	0.221	0.593	0.260	0.820
substrait-java	0.666	0.612	0.446	0.874
tiered-storage-for-apache-kafka	0.611	0.856	0.539	0.824
inspectit-ocelot	0.678	0.421	0.344	0.792
netcdf-java	0.592	0.471	0.389	0.754
allure-java	**0.884**	0.913	**0.759**	0.924
artio	0.450	**0.959**	0.579	0.903
sechub	0.335	0.591	0.400	0.799
fixture-monkey	0.315	0.781	0.356	0.793
spring-amqp	0.161	0.626	0.168	0.670
springwolf-core	0.586	0.881	0.546	0.877
twitch4j	0.461	0.869	0.610	**0.950**
wala	0.248	0.745	0.339	0.884
junit-framework	0.676	0.795	0.569	0.842
dwh-migration-tools	0.475	0.509	0.374	0.831
extender	0.458	0.658	0.511	0.921
tritium	0.566	0.942	0.481	0.780
Average	**0.496**	**0.720**	**0.454**	**0.839**

Table 6: Cross-project performance per repository. Maximum values per metric are bolded, while minimum values are underlined.

artio and *springwolf-core*, all of which achieve MCC scores above 0.5, indicating moderate predictive performance. Notably, *allure-java* achieves the strongest overall performance with an MCC of 0.759 and an ROC-AUC of 0.924, corresponding to a strong correlation according to the MCC interpretation guidelines. In contrast, repositories such as *spring-amqp*, *tds* and *wala* achieve considerably weaker MCC scores, indicating that the model struggles to effectively distinguish positive and negative samples for these repositories.

Interestingly, *twitch4j* shows one of the largest differences between settings, achieving the weakest MCC in the within-project setting while simultaneously achieving one of the strongest MCC scores in the cross-project setting.

Characteristics of predictable projects

The substantial variation in predictive performance between repositories raises the question of which project characteristics make co-evolution easier or harder to predict. To investigate this, we divided projects into two groups based on the median MCC score. Projects above the median MCC were considered easier to predict, while projects below the median MCC were considered harder to predict. We then compared the feature distributions between both groups using statistical tests.

It should be noted that this analysis is exploratory in nature. We evaluated 57 different features while using a significance threshold of $\alpha = 0.05$. Under these conditions, approximately three statistically significant results would be expected by chance alone. Furthermore, the analysis is based on only 18 projects, resulting in groups of nine projects each. Consequently, the statistical power of the tests is limited and the reported results should be interpreted as indicators of potential trends rather than conclusive evidence.

In the cross-project setting, projects with above-median predictive performance exhibited significantly longer median merge times, fewer changed files per PR, fewer modified production files per PR, longer lines of code, and lower average repository-level WMC. Together, these results suggest

Scope	MCC	Δ MCC	Wilcoxon p	Δ Accuracy (pp)
line	0.303	-0.071	0.946	-1.0
method	0.190	-0.184	**1.000**	-12.3
class	0.136	-0.238	**1.000**	-15.3
only-code	0.320	-0.054	0.862	-2.3
non-coverage	0.228	-0.145	**0.990**	-5.2
all	0.357	-0.016	0.568	1.6
repository	0.131	-0.243	**1.000**	-14.8

Table 7: Within-project comparison against the existing coverage baseline across feature scopes. Δ Accuracy is reported in percentage points (pp). Bold p-values indicate significance at $\alpha = 0.05$. Black values indicate significance in the tested direction, while red values indicate significance in the opposite direction.

Scope	MCC	Δ MCC	Wilcoxon p	Δ Accuracy (pp)
line	0.397	-0.008	0.247	0.1
method	0.210	-0.196	**1.000**	-16.1
class	0.172	-0.234	**1.000**	-22.9
only-code	0.318	-0.087	**0.963**	0.1
non-coverage	0.218	-0.187	**0.999**	-7.4
all	0.454	0.049	**0.033**	2.6
repository	-0.166	-0.571	**1.000**	-27.3

Table 8: Cross-project comparison against the existing coverage baseline across feature scopes. Bold p-values indicate significance at $\alpha = 0.05$. Black values indicate significance in the tested direction, while red values indicate significance in the opposite direction.

Scope	MCC cross	MCC within	Δ MCC	Wilcoxon p
line	0.397	0.303	0.095	**0.010**
method	0.210	0.190	0.020	0.517
class	0.172	0.136	0.036	0.185
only-code	0.318	0.320	-0.001	0.432
non-coverage	0.218	0.228	-0.010	0.551
all	0.454	0.357	0.097	**0.019**
repository	-0.166	0.131	-0.296	0.804

Table 9: Cross-project and within-project comparison across feature scopes. Bold p-values indicate significance at $\alpha = 0.05$. Black values indicate significance in the tested direction, while red values indicate significance in the opposite direction.

that cross-project prediction performs better for projects with more focused changes and lower average code complexity. Several additional features approached significance, including repository coverage, method coverage, repository comment density, method size, and class inheritance depth. These trends further suggest that code complexity and developmental practices contribute to predictive performance.

For the within-project setting, projects with above-median predictive performance contained significantly more positive samples, more review comments, and lower method-level WMC. This indicates that within-project prediction benefits from having a larger number of co-evolution examples available for training and from lower code complexity. Several additional features were near significance, including merge time, PR discussion activity, review count, class coverage, method coverage, positive sample rate, positive PR rate, and repository-level WMC. Collectively, these observations suggest that both the amount of available co-evolution data and the intensity of the review process may contribute to prediction performance.

Overall, the analysis indicates that projects that are easier to predict tend to have lower complexity, more available examples of co-evolution, and stronger review or testing practices. However, given the small sample size and large number of statistical tests performed, these findings should primarily be viewed as exploratory observations that warrant validation in future studies on larger datasets.

Baseline comparison

The within-project predictions outperformed the random baseline significantly, with the random baseline indeed having an MCC score of 0. Table 7 compares the within-project performance of the different feature scopes against the existing coverage baseline. The results show that within-project predictions are worse than our coverage-based baseline, with the coverage-based baseline significantly outperforming the method, class, repository and non-coverage subsets. The full feature set and the line and only-code subsets perform slightly better, as the p-value is not significant, but they still perform worse than the baseline. Lastly, the full feature set is the only set with a positive MCC Δ and a positive accuracy Δ. However, the Wilcoxon p-value of 0.568 shows that these results are non-significant.

In the case of the cross-project prediction, the results are fairly similar, as can be seen in Table 8. Our model still significantly outperforms the random baseline, with the sole exception being the repository subset, but gets beaten on most feature subsets by the coverage-based baseline. However, the full feature set does outperform the coverage-based baseline significantly, with a p-value of 0.03, an MCC difference of 0.05, and an accuracy difference of 2.6 percentage points. Furthermore, the line-based subset of features appears to be performing better than the coverage-based metric. However, these results were not strong enough to support rejecting the null hypothesis.

Comparison between within project and cross-project predictions

We also compared the performance of our within-project setup to the cross-project setup, as can be seen in Table 9. Cross-project outperforms within-project significantly in the line subset and full feature set scenarios, with a p-score of 0.01 and 0.02, respectively, and a ΔMCC of 0.07 and 0.09, respectively. For all other feature subsets, there were no additional significant results.

4.4 Feature Importance Evaluation

Table 10 shows the 10 most important features by the average feature rank per project in both the within-project and the cross-project settings. In both the cross-project setting and the within-project setting, existing coverage is the top feature by average rank and also appears in the top 10 for all projects. We see that there is quite a big difference in the features that appear in the top 10, with the cross-project setting having 8 features from the non-code and repository scope, while the within setting has only 1 non-code feature and no repository features. The only features that are shared in the top 10 of both settings are the line already being covered, the number of unique operators in a line and the number of lines added in the PR.

Figure 7 shows the feature importance's in both settings for the top 15 features in the cross-project setting. As we can see, often the differences are small, however there are some features where there is a large difference. For instance, the number of reviews for a PR and the number of external dependencies have a much larger impact in the cross-project setting than in the within-project setting.

Simultaneously, we see that features on the class, method, and line scope appear to have a bigger impact in the within-project setting. For instance, the relative line number in a file is not shown in figure 7 as it is the 26th most important feature in the cross-project setting, while it is the 4th most important feature in the within-project setting, reaching an importance of more than 0.1 for some projects.

Inspired by the work of Wang et al. [11], we analyzed how predictive performance changes when using progressively larger feature subsets based on Recursive Feature Elimination (RFE) rankings for the top 20 features. We evaluated two strategies. In the prefix strategy, the model is first trained using only the most important feature, after which features are incrementally added in descending order of importance. For example, the model is first trained using only existing coverage, after which features such as number of PR added lines are added sequentially. In the postfix strategy, the process is reversed, starting with the least important feature and incrementally adding more important features, moving from right to left in the ranking. For example, the model is first trained using only line length, after which features such as unique commenters are added.

The results show that existing coverage alone already achieves an accuracy of approximately 78%, whereas line length alone achieves only around 73%. Additionally, using only the top 5 features already results in an accuracy of roughly 81%, after which adding additional features yields only marginal improvements.

Individual Feature Analysis

We also investigated how the 5 most important features in the cross-project setting influence the predictive strength with Partial Dependence Plots, as can be seen in Figure 9. Partial dependence plots show what the predicted value is at different values of the feature, when all other features are kept stable. In our experiment, this thus shows how likely the model is to predict a line should be covered based solely on the selected feature.

In the figure, we can see that a line not being covered yet has only a 10% chance of being predicted to be covered, while if the line was already covered, this chance jumps to 40%. This again shows the strong influence of the coverage-based metric. Additionally, we see positive trends between the number of reviews and the number of added lines in the PR. We also see a negative trend for the amount of deleted lines. We see that for the unique operator count, values are very unlikely to be higher than 2. We also see that there is a massive difference between 0 and 1 operator, while the difference above 1 operator is marginal.

Unfortunately, due to the limited dataset sizes of the within-project datasets, it was not possible to create stable partial dependency plots. Most plots were not useful because the data were noisy, with trends varying greatly across projects.

Category importance

The importances of the different scopes can be seen in Figure 10. Here we see that the line, method and class features are the least important scopes in both settings for the model. The coverage metric is the most important scope for the within-project setting and the second strongest scope in the cross-project setting. The non-code category is the most important scope for the cross-project setting, with it being the second most important in the within-project setting. Lastly, the repository scope is the third most important scope for both settings.

The local scopes, such as line, method, and class, are relatively more important in the within-project setting than in the cross-project setting, while the non-code and repository scope are more important in the cross-project setting compared to the within-project setting. The coverage-based category is equally important in both settings.

To determine if the non-code features strengthen the model, we ran a Wilcoxon signed-rank test between the model with just the code-based features and the model including all features, including the non-code features. The results of this comparison can be seen in Table 11.

The non-code features strengthen both the within-project as well as the cross-project model, with p-values of 0.043 and 0.004, respectively. In the within-project setting, the median MCC changes by 0.004, while in the cross-project setting the median MCC changes by 0.086.

5 Discussion

In this chapter, we interpret the obtained results and discuss their implications for predictive software testing and software engineering research. We begin by comparing the observed prevalence of co-evolution with findings from existing literature, followed by a discussion of the challenges encountered during dataset construction. Next, we evaluate the feasibility of predicting test-suite modifications using a combination of code, coverage, repository-level, and non-code characteristics. We then examine the importance of the different features and feature scopes, as well as the differences between within-project and cross-project prediction. Subsequently, we discuss the threats to validity of this study. Finally, we consider the broader implications of our findings for both practical software testing and the wider field of software engineering.

5.1 Prevalence of co-evolution

Our work found that on average, 38.2% of PR's contain co-evolution. This is substantially lower than what would be expected based on the survey results of Sterk et al. [20], in which developers reported that test modifications frequently accompany production code changes. In contrast, our findings are more consistent with the empirical repository mining studies of both Beller [45] and Levin and Yehudai [13]. Beller [45] found that there is a weak to moderate correlation between production code changes and test changes, while Levin and Yehudai [13] found an average per-commit rate of

Within-project			Cross-project		
Feature	Top 10 frequency	Avg. rank	Feature	Top 10 frequency	Avg. rank
Existing coverage	100.0%	2.00	Existing coverage	100.0%	1.00
Number of unique operators	83.3%	7.22	Number of PR added lines	100.0%	2.50
Number of PR added lines	61.1%	9.28	Number of reviews	100.0%	2.67
Relative line number in file	50.0%	12.61	Number of PR deleted lines	100.0%	4.61
Method RFC	44.4%	15.06	Number of unique operators	100.0%	6.17
Covered lines percentage class	38.9%	12.56	Number of external dependencies	100.0%	7.33
Class CBO	38.9%	12.89	Repo WMC	94.4%	5.33
Class LCOM	38.9%	13.83	Number of commits	94.4%	7.56
Class fan-in	38.9%	15.56	Number of project dependencies	88.9%	9.28
Covered lines percentage method	38.9%	15.72	Number of changed files PR	50.0%	10.89

Table 10: Top 10 features by rank frequency for the within-project and cross-project settings [all].

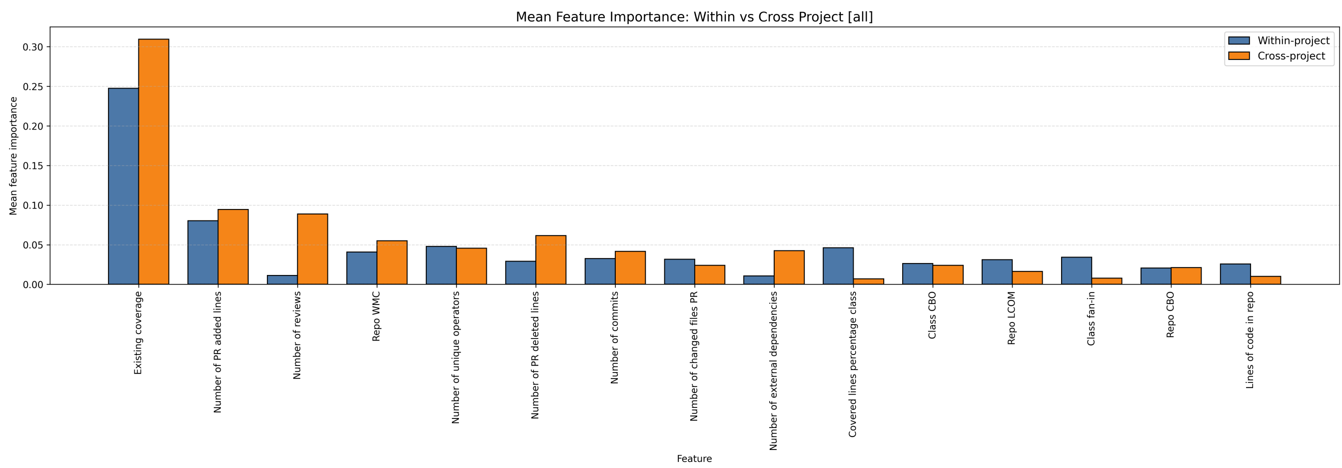


Figure 7: Feature importances in both the within-project setting and the cross-project setting for the top 15 features in the cross-project setting.

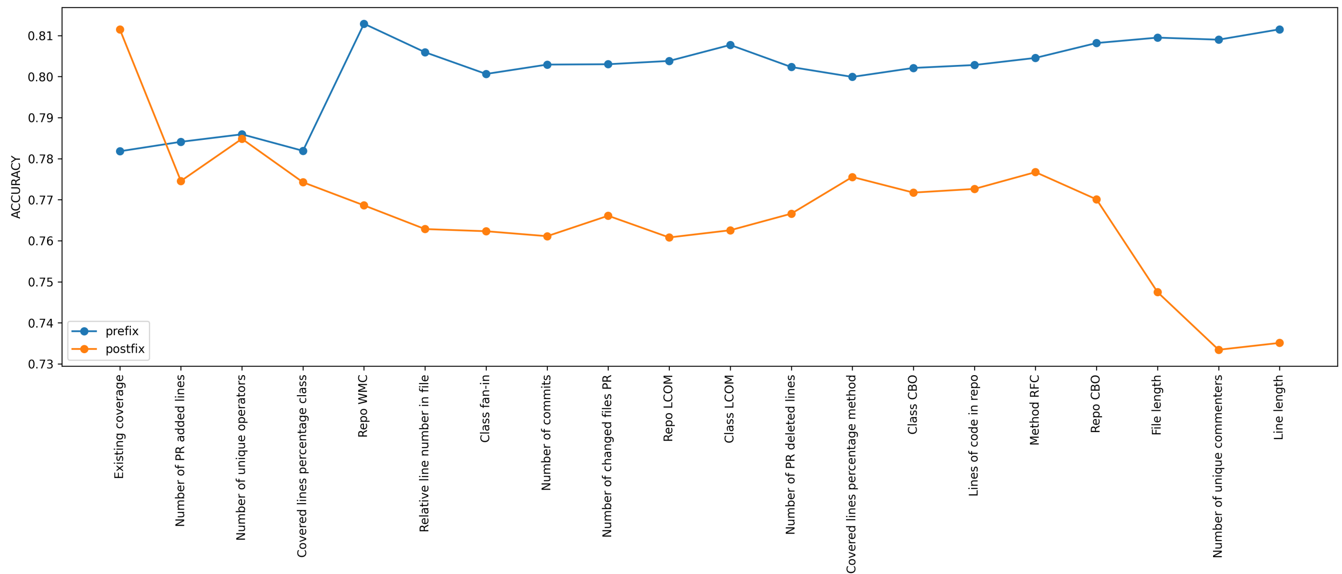
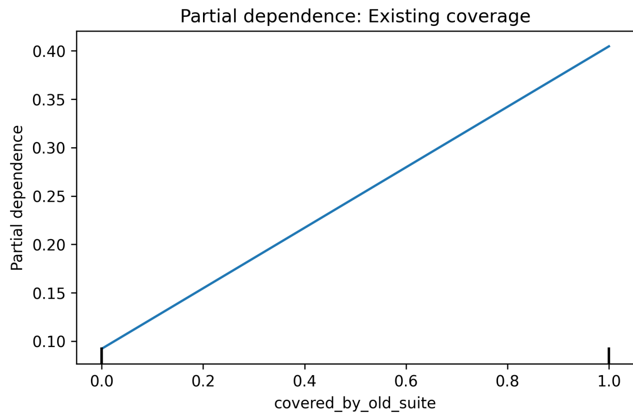
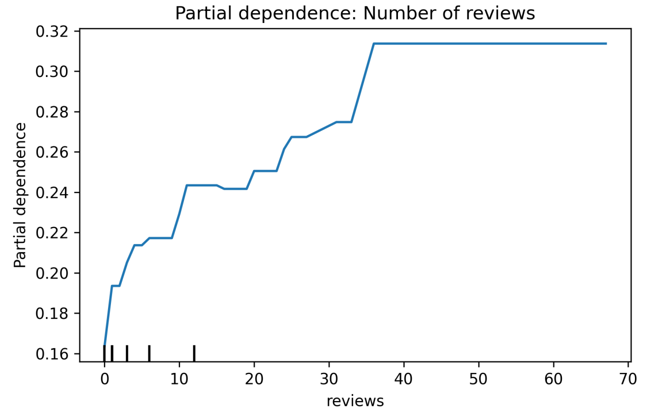


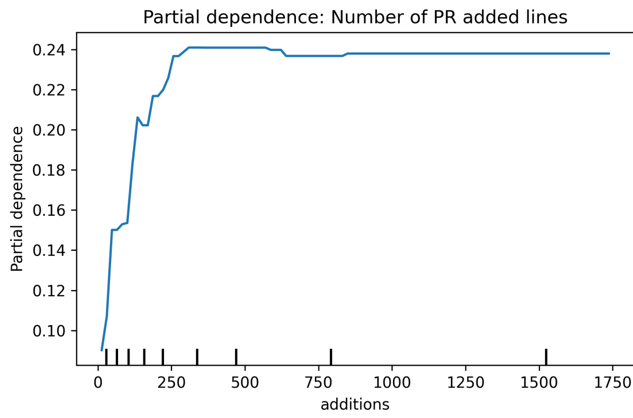
Figure 8: Recursive feature elimination showing accuracy in both the pre-fix and post-fix methods. The prefix method shows accuracy of the subset of features from left to right, while the postfix method shows the accuracy of the subset of features going right to left.



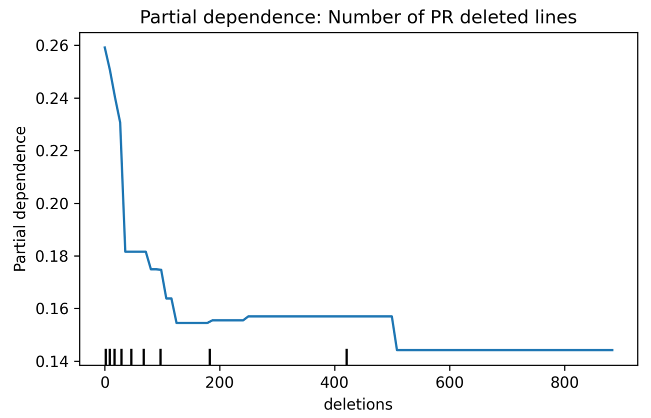
(a) Existing coverage, binary variable, only data at 0 and 1 are accurate.



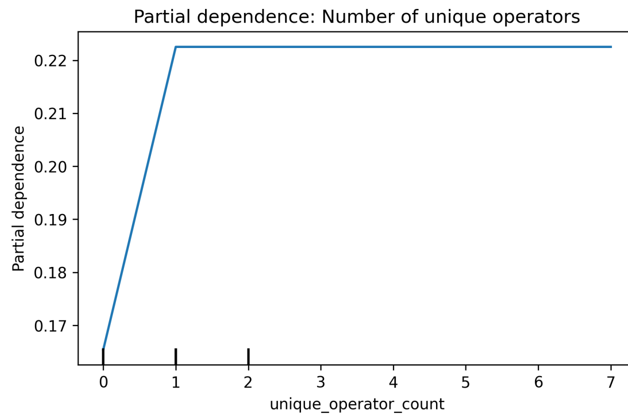
(b) Number of reviews



(c) Number of PR added lines



(d) Number of PR deleted lines



(e) Number of unique operators

Figure 9: Partial dependence plots of the most important features in the cross-project setting. The black marks along the x-axis indicate the distribution of observed feature values.

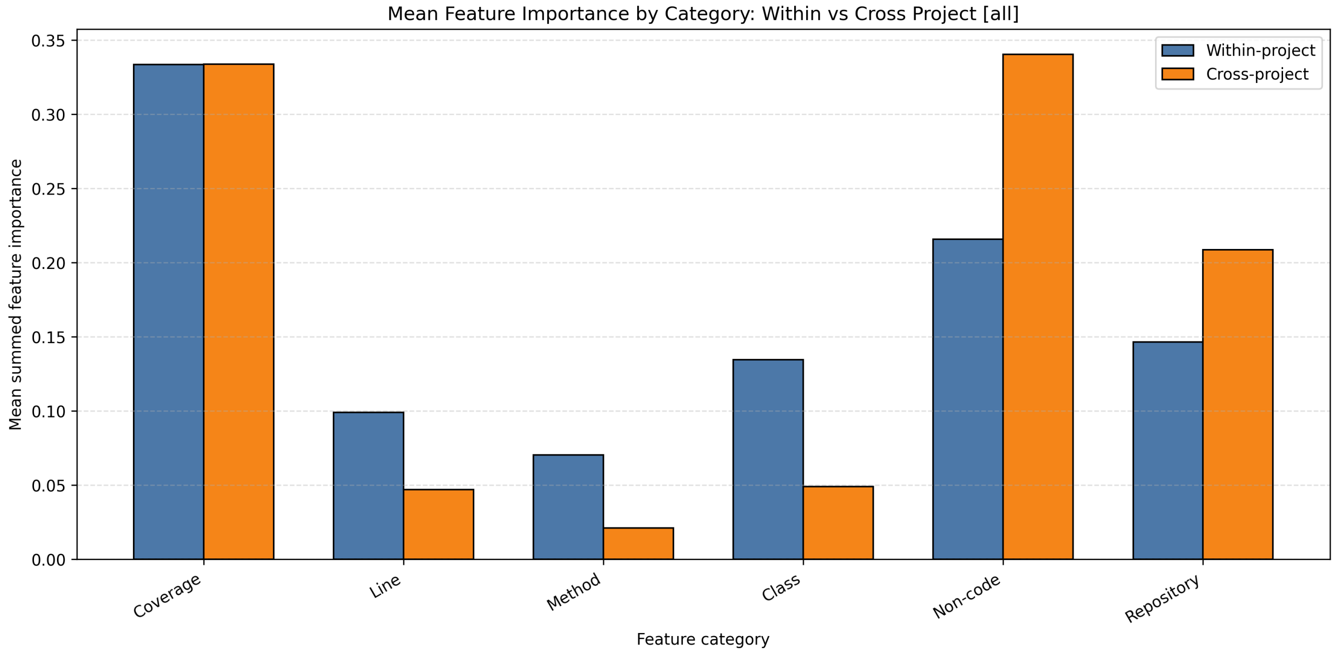


Figure 10: Feature category importance in both the within-project setting as well as the cross-project setting

Table 11: Comparison between the full feature model and the only-code feature model. Bold p-values indicate significance at $\alpha = 0.05$. Black values indicate significance in the tested direction, while red values indicate significance in the opposite direction.

Setting	MCC all	MCC only-code	Δ MCC	Wilcoxon p MCC
Within-project	0.357	0.320	0.038	**0.043**
Cross-project	0.454	0.318	0.136	**0.004**

co-evolution of 32.3% with a standard deviation of 17.1% and a maximum of 68.5%. These results closely match the results presented in Table 4.

One possible explanation for why our findings align more closely with Beller [45] and Levin and Yehudai [13] than with Sterk et al. [20] is the difference between perceived and empirically observed development practices. Prior work by Devanbu et al. showed that strongly held software engineering beliefs do not always align with empirical observations [46]. Additionally, this discrepancy could partially be explained by social desirability bias, where respondents overreport behavior that is considered desirable or expected. Van de Mortel found that only a very small fraction of self-report-based studies explicitly investigate social desirability bias, while a substantial portion of those studies that did investigate it found evidence that responses were influenced by it [47]. Developers may thus be overestimating their adherence to co-evolution.

The substantial variation in co-evolution rates and positive sample distributions across repositories indicates that testing practices are highly project-dependent. This could mean that the predictive patterns learned within one repository may not directly transfer to another repository. This variability can make cross-project prediction challenging, as models trained on one set of repositories may not be able to generalize to projects with different testing behaviors, development practices, or class distributions.

5.2 Buildability and testability

A substantial number of projects and PRs could not be successfully extracted or built. Common causes included broken dependencies, manual testing practices, external software dependencies, unavailable toolchain repositories and uncommon project layouts. Similar challenges have been reported in recent large-scale studies of Java projects, which found that build failures remain common and that JDK incompatibilities are a major contributing factor [43], additionally, they found that Gradle projects in particular struggle to build. This supports our decision to implement an automated JDK fallback strategy during data collection.

While an automated data collection pipeline enables large-scale analysis, it also introduces the risk of dataset bias, as projects with more standardized and reproducible build processes are more likely to be included successfully. This may affect the representativeness of the dataset, potentially under representing projects with complex build environments or non-standard testing practices. However, the analysis presented in Section 4.1 showed no strong relationship between project popularity or contributor count and extraction success, suggesting that the resulting bias is not primarily driven by repository size or popularity. Still, we cannot say with certainty that there is no bias in the dataset.

5.3 Feasibility of test suite modification prediction

All evaluated models significantly outperform the random baseline based on class prevalence, demonstrating that the ex-

tracted feature sets contain meaningful predictive information regarding test suite co-evolution. The strongest performance was achieved using the full feature set in the cross-project setting, which had a moderate performance (MCC 0.454) with a relatively high recall and low precision.

In the within-project setting, the best-performing feature set was also using the full feature set, with a slightly weaker performance compared to the cross-project setting (MCC 0.357). Again, while recall is relatively high, precision is even lower than in the cross-project case, showing that an even larger fraction of samples is classified as a false positive.

The combination of relatively high ROC-AUC and recall values together with comparatively low precision suggests that the models are reasonably effective at ranking positive samples above negative samples, but struggle to establish a reliable binary classification boundary. This is likely influenced by the class imbalance present in many repositories, heterogeneous testing practices between projects, and noise in the co-evolution labels. Since co-evolution labels are derived using coverage information, some changes may be unintentionally covered, introducing inaccuracies into the extracted labels. We discuss this limitation further in Section 5.6. As a result, the models appear more effective at identifying potentially relevant samples than at making highly precise binary decisions.

Historical coverage information proved to be highly predictive across both experimental settings. In the within-project setting, the full feature model did not significantly outperform the coverage-only baseline, suggesting that existing coverage information alone already captures a substantial amount of the predictive signal for line-level co-evolution.

In contrast, the cross-project setting showed that combining code and non-code characteristics with historical coverage information significantly improves predictive performance over the coverage-only baseline. This indicates that while historical coverage is the single strongest predictor, additional code and non-code characteristics do improve model performance and are thus a worthwhile addition.

These findings provide a clear **answer to RQ1:** test-suite modifications can be predicted with fair-to-moderate accuracy using a combination of code, coverage, repository-level, and non-code characteristics when predicting on the PR-level. Performance is strongest in the cross-project setting, where the full-feature set outperforms the coverage-only baseline. Simultaneously, using only historical coverage information already yields strong predictive performance while requiring substantially less effort to deploy in practice. Most modern projects already collect coverage data as part of their CI pipelines, while extracting the remaining code and non-code characteristics requires significantly more preprocessing and analysis effort.

5.4 Feature importance

Historical coverage information consistently emerged as the strongest predictor across both experimental settings. This suggests that previous testing behavior is highly indicative of future testing behavior, meaning that developers tend to extend or modify tests in areas that were already covered previously. This is not entirely unexpected, as production changes often alter the behavior of the system, requiring tests to be updated to validate the modified behavior. Furthermore, code that was previously considered important enough to test is likely to be regarded as important to test again in future changes. This is supported by the finding of Yu et al. [48] that code with a higher amount of historical defects is more likely to contain new defects. This finding also helps explain why the relatively simple coverage-only baseline already achieved strong predictive performance.

Other features that ranked as important in both features were the number of added lines in a PR, the repository WMC and the number of unique operators that the line contained.

In the cross-project setting, the number of PR reviews, the number of deleted lines in a PR and the number of external dependencies are among the most important features, while these features are comparatively less important in the within-project setting. The within-project setting assigns higher importance to more localized features, such as class coverage percentage, class CBO, method RFC, class LCOM and the relative position of a line within a file.

When analyzing feature importance at the scope level, the coverage, non-code and repository scopes are the most influential categories in both settings, while the more localized line, method and class scopes contribute much less. This provides a clear **answer to RQ2:** Broader development and project-level characteristics are more predictive of test suite co-evolution than the specific local code changes themselves, with the exception of previous line coverage, which is the strongest feature.

It should be noted, however, that interpreting development-process characteristics is not always straightforward. For example, the number of reviews emerged as one of the most important features in the cross-project setting. One possible explanation is that PRs receiving more reviews tend to be more complex and therefore require additional testing. Alternatively, the feature may act as a proxy for project-level quality assurance practices, where projects with stricter review processes also place greater emphasis on testing. In this case, the model may be learning differences in testing culture rather than characteristics of the individual code change. These examples serve to provide caution: feature importance should not necessarily be interpreted as evidence of causality.

In the field of defect prediction, prior work has consistently shown that a combination of development-process characteristics and structural code metrics is effective for predicting defect-prone code, with development and process-related features being the most important [49, 50]. As automated testing is closely tied to defect detection and software quality assurance, it is reasonable that similar feature categories are also predictive for test-suite co-evolution.

An important implication of these findings is that future work on co-evolution prediction should not focus exclusively on structural code characteristics. While existing approaches primarily rely on code-based metrics [11, 12, 17], our results show that development-process characteristics contain substantial predictive information and may be equally important for understanding when test-suite modifications are required. Furthermore, most existing work focuses on local

code scopes, such as line-, method-, and class-level characteristics, whereas our findings indicate that repository-level characteristics are often more informative for predicting co-evolution. Together, these observations suggest that the need for test-suite modifications is influenced not only by the changed code itself, but also by the broader development context in which the change occurs. Finally, although additional features improve predictive performance, historical coverage information alone already captures a large portion of the predictive signal. This raises the question of whether the additional complexity required to extract and process large numbers of code and process metrics is justified by the resulting performance gains in practical applications.

Partial Dependence Analysis

The partial dependence analysis provides additional insight into the findings for RQ2 by illustrating how the most influential features affect the model predictions. The strongest effect can be seen for historical coverage, where the predicted probability of co-evolution increases from roughly 10% to more than 40% when a line was already covered by the existing test suite. This again highlights that previous testing behavior is a very strong indicator for future testing behavior and explains why the coverage-only baseline already performs quite well.

We also observe positive trends for the number of reviews and the number of added lines in a PR. This suggests that larger PRs and PRs receiving more discussion are more likely to require accompanying test modifications. This matches the intuition that larger or more complex changes generally require more verification effort. In contrast, the number of deleted lines shows a negative relationship with the predicted probability of co-evolution. One possible explanation is that deletions are much more common in cleanup or refactoring work, which may require fewer additional tests than added functionality.

Lastly, the plot for the number of unique operators shows that the largest increase in predicted probability occurs between zero and one operator, while additional operators only have a small effect. This implies that the presence of operational logic on a line is more important than the exact amount of operational complexity. The rug plots also show that higher feature values occur relatively infrequently in the dataset, meaning that the model behavior in those regions should be interpreted more carefully.

5.5 Differences between within- and cross-project prediction

Interestingly, the cross-project setting achieved substantially better predictive performance than the within-project setting. This contradicts findings from prior work, such as Wang et al., who observed significantly lower performance in the cross-project setting [11]. One possible explanation for this difference is the distinct feature sets and prediction granularity used in our work. Wang et al. perform prediction on the file level and primarily rely on statistics derived from source code modifications, such as the number of added, modified, or deleted language constructs, while we predict on the line level and use more granular features across multiple scopes and including developmental features as well.

In our experiments, using only code-related features resulted in no statistically significant difference between the within-project and cross-project settings. However, after adding coverage, non-code and repository-level characteristics, the cross-project setting improved substantially, resulting in an average MCC improvement of 0.084 over the within-project setting. This suggests that these broader developmental and project-level characteristics generalize more effectively across repositories than localized code characteristics. Taken together, these findings provide a clear **answer to RQ3:** the primary differences between within-project and cross-project prediction lie not only in predictive performance, but also in the types of characteristics that are most informative. At the same time, within-project prediction relies more heavily on localized code characteristics, while cross-project prediction benefits substantially more from coverage, repository-level, and development-process characteristics.

This interpretation is further supported by the feature importance analysis shown in Figure 10, where non-code and repository-level features consistently contributed more strongly than line, method and class-level features. This may indicate that broader development patterns such as review behavior, project maturity, contributor activity and repository-level structure are more stable across projects than local coding styles or implementation details, making them more suitable for cross-project prediction.

This interpretation is also consistent with our exploratory analysis of repository performance. In the cross-project setting, repositories that were easier to predict exhibited significantly longer merge times and fewer changed files and production files per PR. Furthermore, review-related metrics and repository complexity measures were among the characteristics that differentiated easier and harder repositories. Together, these findings suggest that development-process and repository-level characteristics are associated not only with individual predictions, but also with the overall predictability of a repository.

Another possible explanation for the relatively weaker within-project performance is the limited amount of training data available per repository. This explanation is supported by the exploratory analysis of project-level performance in Section 4.3. Within the within-project setting, repositories with above-median MCC scores contained significantly more positive samples than repositories with below-median MCC scores (1266 versus 320 positive samples on average). Additionally, positive sample rate and positive PR rate both approached statistical significance. These findings suggest that the availability of co-evolution examples plays an important role in determining prediction performance. Since the within-project models are restricted to learning from the historical data of a single repository, projects with relatively few positive examples may not provide sufficient training data to learn robust predictive patterns. In contrast, the cross-project setting benefits from aggregating positive examples across multiple repositories, reducing this limitation. While the cross-project setting is able to learn from a substantially larger and more diverse collection of PRs, the within-project setting is constrained to the historical evolution of a single reposi-

tory. This may limit the model's ability to learn robust developmental patterns and increase the risk of overfitting to repository-specific behavior.

However, this analysis should be interpreted cautiously. The comparison was performed on only 18 repositories, resulting in two groups of nine projects each. Furthermore, 57 features were evaluated using a significance threshold of $\alpha = 0.05$, implying that several significant findings could be expected by chance alone. Consequently, these results should be regarded as exploratory evidence rather than definitive conclusions.

An important implication of our findings is that cross-project co-evolution prediction may be more feasible than previously assumed. Prior work has generally treated cross-project prediction as the more challenging setting [11], yet our results suggest that incorporating development-process and repository-level characteristics can substantially improve generalizability across projects. This suggests that future research should place greater emphasis on cross-project co-evolution prediction, as the inclusion of development-process and repository-level characteristics appears to mitigate some of the generalization challenges observed in earlier work.

5.6 Threats to validity

This study is subject to several threats to validity. In the following subsections, we discuss the main threats related to construct validity, internal validity, and external validity, as well as the steps taken to mitigate their potential impact.

Construct validity

In our work, we use a line being covered by the test cases that are modified in the PR as a proxy that this line should be covered. This proxy, however, is not completely accurate, as lines can be covered accidentally and assertions can be weak, thus not verifying the line behavior. Consequently, some lines may be labeled as requiring testing even though the modified tests do not meaningfully validate their functionality. This introduces noise into the target labels used for model training and evaluation, which may reduce predictive performance and affect the estimated importance of individual features. We partially mitigate this threat by relying on coverage information from the modified test cases rather than the full test suite, increasing the likelihood that the observed coverage is functionally related to the production changes. Furthermore, coverage-based definitions of test relevance are commonly used in test-selection and co-evolution research. Nevertheless, the risk of label noise remains.

Additionally, we only consider automatic testing in this work. However, some projects also use manual testing practices. This leads to samples potentially being labeled as not needing testing, while they would still benefit from testing. The impact here is limited, as we only include work which produces coverage and we require at least 100 changed lines with co-evolution and at least 3 PR's containing co-evolution in the 3-year period of PR's. This suggests that the included projects make at least moderate use of automated testing practices.

A core assumption of our work is that co-evolution occurs within the same PR. In practice, developers may introduce production code changes and corresponding test modifications in separate PRs or commits. As a result, some production changes in our dataset may be incorrectly labeled as not requiring test modifications, while the corresponding test changes occurred outside the analyzed PR. This could lead to an underestimation of the prevalence of co-evolution and introduce noise into the target labels used for prediction. We partially mitigate this threat by following assumptions commonly made in the literature. Sterk et al. [20] found that most developers create tests at the PR level or more frequently, while Wang et al. [11] reported that most test modifications occur within 48 hours of the corresponding production code changes. These findings suggest that a substantial portion of co-evolution activity occurs close in time to the associated production changes. However, we only find co-evolution in 38,2% of PRs, implying that co-evolution could be happening outside of the pull-request scope.

Internal validity

Our results are gathered through JaCoCo injection into Gradle projects automatically. This method can fail or behave inconsistently across projects. To prevent this from affecting the study, we only include PR's for which the entire data collection passes, including the coverage generation. We also only include samples from the PR if there are actually covered lines produced by the coverage task, meaning that the entire injection must have been at least partially successful. However, it could be the case that certain submodules did not produce coverage data, without resulting in an error, while some others did. In this case, the lines of the PR are included in the dataset. We believe that this is unlikely, as this would mean the injection was successful for at least one submodule, but not the others. Nevertheless, partial failures without explicit errors may still introduce some measurement noise, although manual inspection of dozens of PRs across multiple projects did not reveal this behavior.

As we exclude projects which fail to generate coverage, this could lead to selection bias toward projects which are more modern, well-structured, and more maintained. To determine the effect of this, we analyzed whether there was a difference between the populations of our failed and successfully extracted projects in the number of contributors and the number of stars. Our assumption was that projects with more contributors and more stars are more mature projects with a better setup, since more new people will have to work on the project, this could have then resulted in a biased dataset. However, our analysis showed no significant difference across the two populations.

When conducting predictive machine-learning-based methods, there is a risk of data leakage. This occurs when the model gets access to training data that can be directly linked to the test data. In our case, we have a risk of data leakage when lines from the same PR appear in both the training and test set. To mitigate this risk, we decided to make use of grouped splits in the within-project experiment and, of course, project-level splits in the cross-project experiments.

Finally, some of the discussion relies on exploratory analyses comparing repositories with above- and below-median predictive performance. Because these analyses evaluated 57

features on only 18 repositories, some observed differences may represent false positives arising from multiple comparisons. Consequently, these findings should be interpreted as indicative trends rather than definitive evidence.

External validity

For our dataset, we selected open-source Java projects that use the Gradle build system. Consequently, our findings may not generalize to projects written in other programming languages, projects using different build systems, or proprietary software systems. In particular, different programming languages and ecosystems may exhibit different testing practices, development workflows, and co-evolution patterns, which could influence both predictive performance and the relative importance of the extracted features.

Furthermore, we selected projects that exhibit active development, are relatively mature, and have at least 11 contributors. As a result, our findings may not generalize to legacy systems, startup projects, or smaller development teams. Such projects often follow different development and testing practices, potentially affecting both the prevalence of co-evolution and the usefulness of the considered characteristics. Additionally, software engineering practices evolve over time. Since our dataset only contains PRs created after 2023, the observed relationships between code changes, development processes, and test-suite modifications may not be representative of older development practices. Therefore, caution should be exercised when generalizing our findings to projects developed in substantially different contexts or time periods.

The increasing adoption of AI-assisted software development may affect the generalizability of our findings. Our study does not distinguish between human-written and AI-generated code when constructing the dataset. As a result, we cannot directly assess whether the identified predictors of test-suite modification apply equally to both categories of code.

Furthermore, our dataset spans PRs created between 2023 and 2025, a period during which the use of AI coding assistants increased substantially. Prior work has shown that AI-generated code exhibits different structural and stylistic characteristics from human-written code, including differences in complexity, repetitiveness, and code organization [51]. These differences may influence both the extracted features and the resulting testing behavior.

Although it is likely that part of our dataset contains AI-generated code, the extent of this influence is unknown. Consequently, it is unclear to what degree the observed relationships between code characteristics and test-suite modifications generalize to modern AI-generated code. Future work could explicitly identify AI-generated contributions and investigate whether separate predictive models are required for human-written and AI-generated code.

5.7 Implications and applicability

This research contributes to the broader field of software engineering research, specifically within the domains of software testing, mining software repositories and predictive software analytics. Over the past years, increasing attention has been given to the use of machine learning techniques to support software quality assurance tasks such as defect prediction, test selection, flaky test detection and automated code review support. This thesis aligns with this broader trend by investigating whether machine learning models can predict when production code changes require accompanying modifications to the test suite.

A notable contribution of this work is the combination of multiple feature scopes, ranging from local code characteristics to repository-level and developmental features. Existing research in test coverage prediction often focuses primarily on local source code changes or static code metrics. In contrast, the findings of this thesis suggest that broader developmental patterns, such as review behavior, repository characteristics and project activity, may generalize better across repositories than localized implementation details.

The results also highlight the growing importance of historical software engineering data within modern development environments. Many CI/CD pipelines already collect large amounts of metadata, including test coverage, PR activity and review statistics. This thesis demonstrates that such historical development data can be leveraged to support predictive quality assurance techniques. In particular, the strong predictive power of historical coverage information suggests that software testing behavior is highly influenced by existing testing structures.

From a practical perspective, predictive models for test suite co-evolution could support developers during code review and testing activities. Such models could potentially be integrated into CI pipelines or development environments to highlight production changes that are likely to require additional or modified tests. This may help developers prioritize testing effort, improve awareness of potentially under-tested changes, and reduce the likelihood of bugs entering production systems.

At the same time, this work also demonstrates several limitations and risks associated with predictive software engineering systems. Although the models achieve fair-to-moderate predictive performance, precision remains relatively low, meaning that many predicted positive samples are false positives. In practice, excessive false positives could reduce developer trust in such systems and lead to warning fatigue. Furthermore, development and testing practices vary substantially between repositories, which may limit the applicability of generalized prediction models in certain contexts.

Different stakeholders might also view these kinds of predictive systems differently. Developers may benefit from additional guidance regarding testing, but could simultaneously view these systems as intrusive, especially when a fair share of predictions are incorrect. While organizations might benefit from improving software quality and reducing development costs, the method also brings additional computational overhead, one of the things our method was aiming to limit. Furthermore, it would add additional complexity to current CI/CD structures. Lastly, not all projects will be able to benefit from this method. Repositories lacking reliable test coverage infrastructure or consistent development workflows may not benefit equally from these techniques.

Finally, this research reflects a broader trend within Com-

puter Science toward data-driven software engineering and AI-assisted development practices. As software repositories continue to grow in size and complexity, automated support systems are increasingly being explored to assist developers in decision-making processes. This thesis contributes to this broader movement by showing that machine learning techniques can capture meaningful signals related to test suite evolution, while simultaneously illustrating the challenges of balancing predictive performance, interpretability, generalizability and practical usability in real-world software engineering environments.

6 Conclusions and Future Work

In this chapter, we summarize the main findings of this thesis and discuss their implications for predictive software testing and software engineering research. We first present the conclusions drawn from the conducted experiments and analyses, after which we discuss directions for future work.

6.1 Conclusion

This thesis investigated whether test-suite modifications can be predicted from a combination of code, coverage, repository-level, and non-code PR characteristics, and studied which characteristics are most important in both within-project and cross-project settings.

RQ1: How accurately can we predict test-suite modifications from a combination of PR code characteristics and project non-code characteristics? Our results show that test-suite modifications can be predicted with fair-to-moderate accuracy using ML techniques. While historical coverage information already provides strong predictive power, the inclusion of additional code, repository-level, and non-code characteristics further improves performance, particularly in the cross-project setting.

RQ2: What is the influence of differently scoped characteristics on the prediction of test-suite modifications in both a within- and cross-project setting? We find that broader feature scopes are substantially more informative than localized code characteristics. Coverage, repository-level, and non-code characteristics consistently contribute more to predictive performance than line-, method-, and class-level metrics, indicating that test-suite co-evolution is strongly influenced by the broader development context.

RQ3: What are the differences in which characteristics are important in a cross-project setting compared to the within-project setting? Cross-project prediction relies more heavily on repository-level and development-process characteristics, whereas within-project prediction places relatively greater importance on localized code characteristics. Furthermore, the cross-project setting consistently outperformed the within-project setting, suggesting that broader developmental characteristics generalize effectively across repositories.

Overall, this thesis demonstrates that test suite co-evolution can be predicted using ML techniques and that broader developmental and repository-level characteristics play a central role in this prediction task. The findings contribute to the growing field of data-driven software engineering by showing that software repositories contain meaningful signals regarding testing behavior and test evolution. At the same time, the results also highlight that challenges remain to make the method ready for real-world usage.

6.2 Contributions

In summary, this thesis makes the following contributions:

- We construct a large-scale dataset containing line, method, class, repository and non-code PR characteristics extracted from 1303 pull requests spanning 18 open-source Java projects. The dataset additionally includes historical coverage information and co-evolution labels derived from test suite modifications.
- We perform an extensive empirical study on predicting test suite modifications in both within-project and cross-project settings. Using explainable ML models, we analyze the predictive importance of differently scoped characteristics and study how these importance patterns differ between settings.
- We show that broader developmental and repository-level characteristics contribute substantially to cross-project prediction performance, while historical coverage information is the strongest overall predictor of test suite co-evolution.
- We provide a fully reproducible replication package containing the data collection pipeline, preprocessing steps, experimental setup and analysis scripts. Enabling reproduction of the results and facilitating future research on predictive software testing and test suite co-evolution. Available at https://github.com/anoncodeartifact/coverage-prediction
- We provide our modified version of the CK project, which is modified to incorporate repository-level metrics, as well as the start and end lines of methods and classes, and which reports comment metrics for classes and methods. Available at https://github.com/anoncodeartifact/ck-modified

6.3 Future work

While our results demonstrate that predicting test suite co-evolutions is feasible using a combination of code and non-code characteristics, further research is needed to improve predictive performance, especially by reducing false positives. Additionally, further validation is needed to determine whether the observed findings are generalizable to other types of projects and development environments. Future research should also investigate the possibility of integrating our predictive approach into CI/CD pipelines and study how developers perceive and interact with such integrations in practice. Based on these findings, we propose the following directions for future work:

Semantic based features

Our work focused on structural code metrics across different scopes alongside non-code project and PR metrics. While these metrics function as a proxy for complexity, coupling, historical testing practices, and developmental patterns, they do not capture the semantic meaning of the production code

changes. Our work shows that the semantics of the code change might be important, as the amount of unique operators per line was one of the top features in both predictive settings. Future work could investigate whether these semantic features could be used to complement the existing features. Such features may help reduce false positives and improve cross-project generalizability, as semantic information is potentially less dependent on project-specific structures and conventions. This hypothesis is supported by prior work showing that topic-based representations can be effective for defect prediction [52], suggesting that similar approaches may also be beneficial for predicting test-suite co-evolution.

Improving co-evolution detection

Our work uses a novel approach by conflating coverage of production code by the modified test suite to identify co-evolution. While this does allow us to make predictions on the line level instead of the file and method level, this method does have the disadvantage of potentially overestimating co-evolution, as lines of code can be non-meaningfully covered by a lack of assertions, or lie in the path of other code that is the targeted code of the test change. Future work could improve on our co-evolution detection by incorporating assertion-aware coverage as described in Chen et al. [53] as well as the dynamic execution traces used in the work of Liu et al. [12].

Validating on other scopes

This study focuses exclusively on open-source Gradle-based Java projects. While this allowed easier large-scale automatic data collection, it remains unclear to what extent our findings are generalizable to industrial-size closed-source projects, as well as to projects written in other programming languages. As one of the motivations of our study was to reduce test suite bloat in large-scale industrial systems, future work should validate whether the findings of this study are transferable to other environments.

Integration in developer workflow

While this study shows the theoretical viability of test suite modification prediction, the practical usefulness in real-world developmental workflows remains unclear. Future work could investigate how predictive test suite modification could be integrated into developer tooling such as IDE's and CI/CD pipelines. Such studies should evaluate the effects on testing practices when using the framework, including whether it leads to increased developer productivity. Additionally, it should investigate how developers perceive the recommendations made by such a framework in practice, especially in the trust of the framework as well as alarm fatigue relating to the generation of false positive predictions.

Use of Generative AI Tools

Generative AI tools were used during the development and writing process of this thesis. These tools were primarily used to support brainstorming, finding relevant literature, improve readability and language quality, assist with LaTeX formatting, generate table and figure formatting suggestions, and provide debugging assistance for analysis scripts and data-processing pipelines.

The tools used were ChatGPT for brainstorming, literature collections and debugging of the code. And Grammarly for writing assistance.

The author retains full responsibility for the research design, implementation, data collection, experimental evaluation, interpretation of results, and all conclusions presented in this work.

References

[1] OECD, Ed., *OECD Digital Economy Outlook 2024 (Volume 1): Embracing the Technology Frontier*, eng. OECD Publishing, 2024, ISBN: 978-92-64-64012-2 978-92-64-65426-6 978-92-64-60685-2. DOI: 10.1787/a1689dc5-en.

[2] D. Weston, *Helping our customers through the CrowdStrike outage*, Jul. 2024. Accessed: Nov. 30, 2025. [Online]. Available: https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/.

[3] Parametrix, *Crowdstrikes impact on the fortune 500*. Accessed: Jul. 1, 2026. [Online]. Available: https://www.parametrixinsurance.com/reports-white-papers/crowdstrikes-impact-on-the-fortune-500.

[4] FitchRatings, *(Re)Insurers Could Withstand Preliminary Loss Estimates from CrowdStrike Chaos*, Jul. 2024. Accessed: Jul. 1, 2026. [Online]. Available: https://www.fitchratings.com/research/insurance/re-insurers-could-withstand-preliminary-loss-estimates-from-crowdstrike-chaos-22-07-2024.

[5] CrowdStrike, "Channel file 291 incident: Root cause analysis," CrowdStrike, Tech. Rep. Channel-File-291, Jun. 2024, Incident root cause analysis report.

[6] G. Tassey, "The economic impacts of inadequate infrastructure for software testing. national institute of standards and technology," *RTI Project*, vol. 7007, no. 11, pp. 1–309, 2002.

[7] P. S. Kochhar, T. F. Bissyande, D. Lo, and L. Jiang, "An Empirical Study of Adoption of Software Testing in Open Source Projects," in *2013 13th International Conference on Quality Software*, IEEE, 2013, pp. 103–112, ISBN: 978-0-7695-5039-8. DOI: 10.1109/QSIC.2013.57.

[8] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE, Oct. 2009, pp. 291–301, ISBN: 978-1-4244-4842-5. DOI: 10.1109/ESEM.2009.5315981.

[9] S. Mäkinen and J. Münch, "Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies," en, in *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, D. Winkler, S. Biffl, and J. Bergsmann, Eds., vol. 166, Series Title: Lecture Notes in Business Information Processing, Springer International Publishing, 2014, pp. 155–169, ISBN: 978-3-319-03601-4 978-3-319-03602-1. DOI: 10.1007/978-3-319-03602-1_10.

[10] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," en, *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, Jun. 2011, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-010-9143-7.

[11] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and Facilitating the Co-Evolution of Production and Test Code," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Mar. 2021, pp. 272–283, ISBN: 978-1-7281-9630-5. DOI: 10.1109/SANER50967.2021.00033.

[12] L. Liu, S. Wang, Y. Liu, J. Deng, and S. Liu, "Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning," en, in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, ACM, Aug. 2023, pp. 227–237, ISBN: 979-8-4007-0894-7. DOI: 10.1145/3609437.3609449.

[13] S. Levin and A. Yehudai, "The Co-evolution of Test Maintenance and Code Maintenance through the Lens of Fine-Grained Semantic Changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Sep. 2017, pp. 35–46, ISBN: 978-1-5386-0992-7. DOI: 10.1109/ICSME.2017.9. [Online]. Available: http://ieeexplore.ieee.org/document/8094407/.

[14] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, IEEE, May 2009, pp. 151–154, ISBN: 978-1-4244-3493-0. DOI: 10.1109/MSR.2009.5069493. [Online]. Available: http://ieeexplore.ieee.org/document/5069493/.

[15] R. Noemmer and R. Haas, "An Evaluation of Test Suite Minimization Techniques," en, in *Software Quality: Quality Intelligence in Software and Systems Engineering*, D. Winkler, S. Biffl, D. Mendez, and J. Bergsmann, Eds., vol. 371, Springer International Publishing, 2020, pp. 51–66, ISBN: 978-3-030-35509-8 978-3-030-35510-4. DOI: 10.1007/978-3-030-35510-4_4.

[16] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost," en, in *Proceedings of the 2006 international workshop on Automation of software test*, ACM, May 2006, pp. 85–91, ISBN: 978-1-59593-408-6. DOI: 10.1145/1138929.1138946.

[17] C. Brandt and A. Ramírez, "Towards Refined Code Coverage: A New Predictive Problem in Software Testing," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, Mar. 2025, pp. 613–617, ISBN: 979-8-3315-0814-2. DOI: 10.1109/ICST62969.2025.10989028.

[18] A. Khatami and A. Zaidman, *State-Of-The-Practice in Quality Assurance in Java-Based Open Source Software Development*, Version Number: 1, 2023. DOI: 10.48550/ARXIV.2306.09665.

[19] A. Khatami and A. Zaidman, "Quality Assurance Awareness in Open Source Software Projects on GitHub," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, Oct. 2023, pp. 174–185, ISBN:

979-8-3503-0506-7. DOI: 10.1109/SCAM59687.2023.00027.

[20] A. Sterk, M. Wessel, E. Hooten, and A. Zaidman, “Running a Red Light: An Investigation into Why Software Engineers (Occasionally) Ignore Coverage Checks,” en, in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, ACM, Apr. 2024, pp. 12–22, ISBN: 979-8-4007-0588-5. DOI: 10.1145/3644032.3644444.

[21] R. White, J. Krinke, and R. Tan, “Establishing multilevel test-to-code traceability links,” en, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ACM, 2020, pp. 861–872, ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380921.

[22] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, “Revisiting the Identification of the Co-evolution of Production and Test Code,” en, *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, Nov. 2023, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3607183.

[23] M. Hilton, J. Bell, and D. Marinov, “A large-scale study of test coverage evolution,” en, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, Sep. 2018, pp. 53–63, ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238183.

[24] A. Hora, “Excluding code from test coverage: Practices, motivations, and impact,” en, *Empirical Software Engineering*, vol. 28, no. 1, p. 16, Jan. 2023, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-022-10259-7.

[25] N. E. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach* (Innovations in software engineering and software development), eng, 3rd edition. CRC Press, 2015, ISBN: 978-1-4398-3822-8 978-1-4398-3823-5. DOI: 10.1201/b17461.

[26] X. Zhang, Y. Yu, G. Gousios, and A. Rastogi, “Pull Request Decisions Explained: An Empirical Overview,” *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 849–871, Feb. 2023, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2022.3165056.

[27] F. Zampetti, G. Bavota, G. Canfora, and M. D. Penta, “A Study on the Interplay between Pull Request Review and Continuous Integration Builds,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China: IEEE, Feb. 2019, pp. 38–48, ISBN: 978-1-7281-0591-8. DOI: 10.1109/SANER.2019.8667996.

[28] M. Aniche, *Java code metrics calculator (ck)*, Available in https://github.com/mauricioaniche/ck/, 2015.

[29] M. Ivankovic et al., “Productive Coverage: Improving the Actionability of Code Coverage,” en, in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ACM, Apr. 2024, pp. 58–68, ISBN: 979-8-4007-0501-4. DOI: 10.1145/3639477.3639733.

[30] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: A survey,” en, *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012, ISSN: 09600833. DOI: 10.1002/stv.430.

[31] T. Chen and M. Lau, “A new heuristic for test suite reduction,” en, *Information and Software Technology*, vol. 40, no. 5-6, pp. 347–354, Jul. 1998, ISSN: 09505849. DOI: 10.1016/S0950-5849(98)00050-0.

[32] M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for controlling the size of a test suite,” en, *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, Jul. 1993, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/152388.152391.

[33] T. Miller, “Explanation in artificial intelligence: Insights from the social sciences,” en, *Artificial Intelligence*, vol. 267, pp. 1–38, Feb. 2019, ISSN: 00043702. DOI: 10.1016/j.artint.2018.07.007.

[34] P. Linardatos, V. Papastefanopoulos, and S. Kotsiantis, “Explainable AI: A Review of Machine Learning Interpretability Methods,” en, *Entropy*, vol. 23, no. 1, p. 18, Dec. 2020, ISSN: 1099-4300. DOI: 10.3390/e23010018.

[35] J. H. Friedman, “Greedy function approximation: A gradient boosting machine.,” *The Annals of Statistics*, vol. 29, no. 5, Oct. 2001, ISSN: 0090-5364. DOI: 10.1214/aos/1013203451.

[36] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems*, I. Guyon et al., Eds., vol. 30, Curran Associates, Inc., 2017.

[37] A. Goldstein, A. Kapelner, J. Bleich, and E. Pitkin, “Peeking Inside the Black Box: Visualizing Statistical Learning With Plots of Individual Conditional Expectation,” en, *Journal of Computational and Graphical Statistics*, vol. 24, no. 1, pp. 44–65, Jan. 2015, ISSN: 1061-8600, 1537-2715. DOI: 10.1080/10618600.2014.907095.

[38] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, “Permutation importance: A corrected feature importance measure,” en, *Bioinformatics*, vol. 26, no. 10, pp. 1340–1347, May 2010, ISSN: 1367-4811, 1367-4803. DOI: 10.1093/bioinformatics/btq134.

[39] D. W. Apley and J. Zhu, “Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models,” en, *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 82, no. 4, pp. 1059–1086, Sep. 2020, ISSN: 1369-7412, 1467-9868. DOI: 10.1111/rssb.12377.

[40] P. Biecek and T. Burzykowski, *Explanatory Model Analysis: Explore, Explain and Examine Predictive Models*, en, 1st ed. Chapman and Hall/CRC, Feb. 2021, ISBN: 978-0-429-02719-2. DOI: 10.1201/9780429027192.

[41] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene Selection for Cancer Classification using Support Vector Machines," en, *Machine Learning*, vol. 46, no. 1-3, pp. 389–422, Jan. 2002, ISSN: 0885-6125, 1573-0565. DOI: 10.1023/A:1012487302797.

[42] L. Briand, S. Morasca, and V. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, Jan. 1996, ISSN: 00985589. DOI: 10.1109/32.481535.

[43] M. Sulír, J. Porubän, and S. Chodarev, "Local software buildability across Java versions," en, *Empirical Software Engineering*, vol. 31, no. 3, p. 78, May 2026, ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-026-10806-6. [Online]. Available: https://link.springer.com/10.1007/s10664-026-10806-6.

[44] M. Büyükkeçeci and M. C. Okur, "A Comprehensive Review of Feature Selection and Feature Selection Stability in Machine Learning," *Gazi University Journal of Science*, vol. 36, no. 4, pp. 1506–1520, Dec. 2023, ISSN: 2147-1762. DOI: 10.35378/gujs.993763.

[45] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," en, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo Italy: ACM, Aug. 2015, pp. 179–190, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786843.

[46] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," en, in *Proceedings of the 38th International Conference on Software Engineering*, Austin Texas: ACM, May 2016, pp. 108–119, ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884812.

[47] T. F. van de Mortel, "Faking it: Social desirability response bias in self-report research," *Australian Journal of Advanced Nursing*, vol. 25, no. 4, pp. 40–48, 2008.

[48] T.-J. Yu, V. Shen, and H. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1261–1270, Sep. 1988, ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/32.6170.

[49] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, Jul. 2000, ISSN: 00985589. DOI: 10.1109/32.859533.

[50] M. Mukelabai, S. Strüder, D. Strüber, and T. Berger, "Feature-Oriented Defect Prediction: Scenarios, Metrics, and Classifiers," 2021, Version Number: 1. DOI: 10.48550/ARXIV.2104.06161.

[51] D. Cotroneo, C. Improta, and P. Liguori, "Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity," in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2025, pp. 252–263, ISBN: 979-8-3503-9302-6. DOI: 10.1109/ISSRE66568.2025.00035. [Online]. Available: https://ieeexplore.ieee.org/document/11229706/.

[52] T.-H. Chen, S. W. Thomas, H. Hemmati, M. Nagappan, and A. E. Hassan, "An Empirical Study on the Effect of Testing on Code Quality Using Topic Models: A Case Study on Software Development Systems," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 806–824, Sep. 2017, ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2017.2699938.

[53] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How Do Assertions Impact Coverage-Based Test-Suite Reduction?" In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Tokyo, Japan: IEEE, Mar. 2017, pp. 418–423, ISBN: 978-1-5090-6031-3. DOI: 10.1109/ICST.2017.45.