Politecnico di Torino

Politecnico di Torino

Computer Engineering

Academic Year 2024/2025

Graduation Session July 2025

Application of Large Language Models in Software Testing

An Analysis of Method-Level Bug Detection

Supervisors:

Prof. Flavio Giobergia

Prof. Alexander Felfernig

Prof. Denis Helic

Candidate:

Simone Bugni Duch

Abstract

Software testing is a crucial phase in the software development life cycle, essential for delivering secure and reliable software systems. Detection of software bugs is critical in order to deploy robust systems to end users, avoiding unexpected behaviors and maintaining high software quality. However, traditional testing methods frequently rely on manual effort or static rule-based tools, approaches that can be time-consuming and resource-intensive.

With the advancement of Artificial Intelligence (AI), Large Language Models (LLMs) have emerged as a breakthrough, demonstrating impressive capabilities on various scenarios, such as machine translation, text summarization, and natural language understanding. Motivated by these promising results, researchers have begun exploring the potential of LLMs for a wide range of software engineering tasks, including applications within software testing.

This thesis explores the application of LLMs to the task of method-level bug detection in source code. Specifically, it examines the influence of different prompting scenarios, including zero-shot and few-shot approaches, on six state-of-the-art open-source LLMs, and it investigates the impact of enriching model prompts with additional context, such as raised exceptions and method call graphs. The evaluations, conducted on the widely adopted Defects4J dataset, offer valuable insights into how model size, prompting strategy, and contextual information affect bug detection performance. Overall, this research provides a comprehensive analysis on LLM-based bug detection, highlighting the potential and limitations of these approaches and identifying directions for future work.

Acknowledgements

I would like to express my sincere gratitude to all those who supported and guided me throughout the course of this thesis.

First and foremost, I am deeply grateful to my supervisor, Prof. Flavio Giobergia, for his valuable feedback and his continuous support and encouragement throughout the development of this work.

I would also like to extend my heartfelt thanks to my co-supervisors from TU Graz, Prof. Alexander Felfernig, who initially proposed this research topic, and Prof. Denis Helic, for their insightful guidance. I am especially thankful for the opportunity they gave me, as an Erasmus student from another university, to work with them and be part of their research. I would also take this opportunity to thank the entire research team for their support and for welcoming me into their group.

Special thanks go to the institutions at Graz University of Technology, in particular the nvCluster team, for providing the computational resources that made the experiments possible. I am also especially grateful to the International Office, and in particular to Gitte Cerjak, for her unwavering patience and invaluable help.

I am truly thankful to my friends for their presence over the years and for all the unforgettable moments we have shared. Your friendship has filled this time with laughter, support, and warmth. I am truly grateful to have you in my life.

Finally, I would like to express my deepest gratitude to my family for their constant and unconditional support throughout this journey. Even if I may not always show it, I am fully aware of your irreplaceable role in my life. Thank you, from the bottom of my heart.

Table of Contents

List of Tables VII

List of Figures XI

List of Listings XII

1 Introduction 1
- 1.1 Structure of the Thesis . . . 2

2 Background 3
- 2.1 Software Testing Description . . . 3
- 2.2 LLMs Description . . . 5
  - 2.2.1 Model Architectures . . . 6
  - 2.2.2 Optimization Approaches . . . 6

3 Experimental Setup 8
- 3.1 Methodology . . . 8
- 3.2 Dataset Description . . . 9
  - 3.2.1 Dataset Preparation . . . 10
  - 3.2.2 Dataset Partitioning . . . 13
- 3.3 Model Descriptions . . . 15
  - 3.3.1 Common Parameters . . . 16
  - 3.3.2 Models List . . . 16
- 3.4 Prompting Strategies . . . 18
- 3.5 Computational Environment . . . 26

4 Results And Discussion 27
- 4.1 Implementation Details . . . 27
  - 4.1.1 Few-Shot Examples . . . 28
  - 4.1.2 Exceptions Information . . . 30
  - 4.1.3 Call Graph Construction . . . 30

4.1.4 Evaluation Metrics . . . 32
4.2 Partition 1 . . . 33
4.2.1 Zero-Shot Approach . . . 34
4.2.2 Few-Shot Approach . . . 37
4.2.3 Processing Time . . . 42
4.2.4 Model Choice . . . 43
4.3 Partition 2 . . . 45
4.3.1 Results . . . 47
4.4 Partition 3 . . . 56
4.4.1 Results . . . 57
4.5 Threats To Validity . . . 69

5 Related Work 72
5.1 Automated Program Repair . . . 72
5.2 Vulnerability Detection . . . 73
5.3 Code Analysis . . . 74

6 Conclusion and Future Work 76

A Detailed Results for Ablation Studies 79
A.1 Ablation study: position of the explanation in Few-Shot examples . 79
A.2 Ablation study: analyzing different Few-Shot variants . . . 82
A.2.1 Partition 2 . . . 82
A.2.2 Partition 3 . . . 83
A.3 Ablation study: position of the reasoning field in Chain-of-Thought approaches . . . 86

Bibliography 88

List of Tables

3.1 Overview of Defects4J dataset 9
3.2 Number of bugs within static initialization blocks across the dataset 12
3.3 Final state of the dataset after the filter 13
3.4 Combined total of bugs and buggy methods per project 14
3.5 Three resulting partitions 15
3.6 Characteristics of the tested models 18

4.1 Models size with 16384 tokens context window 28
4.2 Few-shot examples for Partitions 1 and 2 29
4.3 Few-shot examples for Partition 3 30
4.4 Codes assigned to each model 33
4.5 Partition 1 project weights 33
4.6 Project: **Codec**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 34
4.7 Project: **Gson**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 35
4.8 Project: **Collections**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 35
4.9 Project: **Mockito**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 36
4.10 Project: **Closure**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 36
4.11 **Weighted Averages**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold** 37
4.12 F1 scores for different position of the explanation (Top: explanation *before* the `has_bug` field; Bottom: explanation *after* the `has_bug` field); (**P**: Projects; **1**: Codec; **2**: Gson; **3**: Collections; **4**: Mockito; **5**: Weighted Average); Format: value ± confidence interval; Best results in **bold** 38
4.13 Project: **Codec**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 39

4.14 Project: **Gson**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 39
4.15 Project: **Collections**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 40
4.16 Project: **Mockito**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 40
4.17 Project: **Closure**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 41
4.18 **Weighted Averages**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold** 41
4.19 Average processing time per method on Partition 1 projects with zero-shot approach; Results expressed in s/m (seconds/method) . . 42
4.20 Average processing time per method on Partition 1 projects with few-shot approach; Results expressed in s/m (seconds/method) . . . 43
4.21 Code assigned to each prompting strategy 45
4.22 Partition 2 project weights 46
4.23 Partition 2 project weights for the zero-shot-whole-class approach . 46
4.24 Partition 2 project weights for the call graph approach 47
4.25 Project: **Csv**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 48
4.26 Project: **JxPath**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 49
4.27 Project: **JacksonCore**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 50
4.28 Project: **Compress**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 51
4.29 Project: **Lang**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 52
4.30 Project: **JacksonDatabind**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 53
4.31 **Weighted Averages**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 54
4.32 Average processing time per method on Partition 2 projects with different approaches; Results expressed in s/m (seconds/method); (**JCore**: JacksonCore; **JData**: JacksonDatabind); Best results in **bold** 56
4.33 Partition 3 project weights 56
4.34 Partition 3 project weights for the zero-shot-whole-class approach . 57
4.35 Partition 3 project weights for the call graph approach 57
4.36 Project: **JacksonXml**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 58

4.37 Project: **Chart**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 59
4.38 Project: **Time**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 60
4.39 Project: **Cli**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 61
4.40 Project: **Jsoup**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 61
4.41 Project: **Math**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 62
4.42 **Weighted Averages**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 63
4.43 Average processing time per method on Partition 3 projects with different approaches; Results expressed in s/m (seconds/method); (**JXml**: JacksonXml); Best results in **bold** 65
4.44 F1 scores of different few-shot variants on Partition 2 projects (**W. Average**: Weighted Average); Format: value ± confidence interval; Best results in **bold** 66
4.45 F1 scores of different few-shot variants on Partition 3 projects (**W. Average**: Weighted Average); Format: value ± confidence interval; Best results in **bold** 67
4.46 Average processing time per method across few-shot variants; Results expressed in s/m (seconds/method); Best results in **bold** 68
4.47 F1 scores for different position of the reasoning field (**Before**: `reasoning` field **before** the `has_bug` field; **After**: `reasoning` field **after** the `has_bug` field); Format: value ± confidence interval; Best results in **bold** 69
4.48 Average processing time per method based on position of `reasoning` field; Results expressed in s/m (seconds/method); (Top: `reasoning` field **before** the `has_bug` field; Bottom: `reasoning` field **after** the `has_bug` field); Best results in **bold** 69

A.1 Project: **Codec**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 79
A.2 Project: **Gson**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 80
A.3 Project: **Collections**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 80

A.4 Project: **Mockito**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 81
A.5 **Weighted Averages**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 81
A.6 Project: **Csv**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 82
A.7 Project: **JxPath**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 82
A.8 Project: **JacksonCore**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 82
A.9 Project: **Compress**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 82
A.10 Project: **Lang**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 83
A.11 Project: **JacksonDatabind**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 83
A.12 **Weighted Averages**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 83
A.13 Project: **JacksonXml**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 83
A.14 Project: **Chart**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 84
A.15 Project: **Time**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 84
A.16 Project: **Cli**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 84
A.17 Project: **Jsoup**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 84
A.18 Project: **Math**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 85
A.19 **Weighted Averages**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold** 85
A.20 **Weighted Averages**; Top: `reasoning` field **before** the `has_bug` field; Bottom: `reasoning` field **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold** 86

List of Figures

3.1 Number of bugs excluded by the filtering operation 12

4.1 Zero-Shot vs. Few-shot approach on Partition 1 projects (Precision) 43
4.2 Zero-Shot vs. Few-shot approach on Partition 1 projects (Recall) . 44
4.3 Zero-Shot vs. Few-shot approach on Partition 1 projects (F1 Score) 45
4.4 F1 Scores of different approaches on Partition 2 projects 55
4.5 F1 Scores of different approaches on Partition 3 projects 64

List of Listings

3.1 Example of included bug (project: Chart, bug: 1) 10
3.2 Example of included bug (project: Math, bug: 6) 11
3.3 Example of excluded bug (project: JacksonDatabind, bug: 87) . . . 11
3.4 Example of excluded bug (project: Codec, bug: 14) 11
3.5 Zero-Shot prompt 19
3.6 Zero-Shot-Motivation prompt 19
3.7 Zero-Shot-Whole-Class prompt 20
3.8 Zero-Shot-Exceptions prompt 21
3.9 Call Graph prompt 22
3.10 Few-Shot prompt 23
3.11 Few-Shot-Motivation prompt 23
3.12 Zero-Shot Chain-of-Thought prompt 24
3.13 Few-Shot Chain-of-Thought prompt 25
4.1 Few-shot example used in the prompt (Project: Chart; Bug: 11) . . 28
4.2 Exception information used in the prompt (Project: Chart; Bug: 11) 30
4.3 Generation of a call graph (Project: Csv; Bug: 2) 31

Chapter 1

Introduction

Software bugs are a persistent challenge in software development, often recognized as inevitable and destructive. They can lead to unexpected behavior, security issues, and increasing maintenance costs in terms of both financial burden [1] and time consumption [2]. As software systems grow in size and complexity, the number of bugs tends to rise accordingly, reaching hundreds or even thousands within a single system [3]. Detecting such defects early is essential to ensure software reliability and to minimize the cost associated with downstream debugging and patching.

Software testing is a crucial activity to ensure the quality and reliability of software products before they are deployed to end users. It is defined as the set of practices and processes that verify whether the software's actual results match the expected ones, based on the given specifications and requirements [4]. The goal of this process is to check the characteristics of a software item for differences between its actual and expected behavior: these differences indicate errors or bugs. Traditionally, testing methodologies relied on manual efforts by human testers or static rule-based programs, which, while effective, can be time-consuming and resource-intensive [5].

The rapid advances in Artificial Intelligence (AI), particularly in Natural Language Processing (NLP), have positioned Large Language Models (LLMs) as a major breakthrough. In recent years, they have received praise worldwide for showing remarkable capabilities when applied across tasks such as machine translation [6], text summarization [7], or classification [8], generating considerable interest within the research community. Thanks to their ability to process both natural and programming languages, LLMs have been leveraged to automate software engineering tasks, such as code generation [9, 10, 11], code summarization [12, 13], and automated test generation [14, 15]. LLMs' ability to reason about code syntax and semantics has led to increasing interest in their application to software testing tasks. This opens new possibilities for automating different aspects of software testing, enhancing efficiency and improving overall software quality.

This thesis explores the application of LLMs in the software testing domain, particularly in the bug detection phase. Specifically, it explores how different prompting strategies (including zero-shot, few-shot, and chain-of-thought techniques) affect the performance of large language models in detecting bugs at the method level of source code. This investigation is conducted using a real-world dataset and explores how context, examples, and structures influence model behavior.

The primary contribution of this work is a systematic evaluation and comparative analysis of multiple state-of-the-art (SOTA) large language models in method-level bug detection. Furthermore, it evaluates the effectiveness of different approaches and prompting strategies to determine which performs best under various circumstances. Finally, it discusses the strengths and limitations of LLM-based bug detection, identifying key challenges and outlining directions for future improvement.

1.1 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 provides essential background on software testing and large language models;
- Chapter 3 describes the experimental setup, including the dataset, models, and prompting strategies used;
- Chapter 4 presents and analyzes the results obtained through experimentation;
- Chapter 5 offers an overview of the existing related research;
- Chapter 6 summarizes the findings and outlines directions for future work.

Chapter 2

Background

2.1 Software Testing Description

Software testing [16] is an essential part of the software development process, as it evaluates the quality of a software product. The main goal of software testing is to assess the presence of defects in software systems, which may lead to incorrect or unexpected behavior. As stated in the introduction (see Chapter 1), testing used to be performed manually by human testers, relying heavily on their experience and capabilities. This approach was resource-intensive and prone to errors. Automated testing, by contrast, involves writing scripts and using dedicated software and tools to perform tests. This significantly reduces both testing time and the cost of software development and maintenance.

The software testing life cycle typically includes the following phases:

- **requirement analysis**: analyze the software requirements and identify the testing objectives, scope, and criteria;
- **test planning**: define the testing plan, including the strategy, goals, and schedule;
- **test development**: develop and review test cases to ensure they align with the plan and requirements defined in the previous stages;
- **test execution**: execute the test cases and record the results to assess the system's behavior against expected outcomes;
- **test reporting**: analyze the results and produce reports that describe the testing process and identify any discovered defects;
- **regression testing**: fix the reported defects and execute the test cases again to ensure that the changes did not introduce new issues;

- **software release**: release the software to customers or end users once all phases are completed and defects are resolved.

Depending on the requirements and complexity of the software under test, the testing process may involve multiple cycles of the above steps. During the testing phase, different types of tests can be performed, including:

- **unit testing**: test individual units or components (e.g., methods or classes), focusing on the functionality to ensure correct behavior;
- **integration testing**: integrate different modules or components and test their interactions to ensure they function correctly together;
- **system testing**: test the software system as a whole, including all the integrated components and the external dependencies;
- **acceptance testing**: test the whole application to guarantee it meets the business requirements and is ready for deployment.

This thesis falls within the scope of unit testing. It focuses specifically on bug detection at the method level, that is, assessing whether individual methods contain defects. It is important to clarify how this task relates to, and differs from, other research areas in software quality assurance, such as bug prediction and fault localization:

- **bug detection** is the process of determining whether a given piece of code (e.g., a method) contains a bug. It is usually framed as a binary classification task and often serves as an entry point for Automated Program Repair (APR) pipelines;
- **bug prediction** is typically a statistical or machine learning task that analyzes historical features (such as code metrics, commit history, and developer activity) to estimate the likelihood that a software component will contain bugs in the future [17]. It is often used to prioritize testing or inspection efforts;
- **fault localization** is the process of identifying the specific lines or expressions in a program that are likely to contain the bug, typically after a failure is observed. Techniques include spectrum-based methods [18], which analyze both failed and passed test cases to rank suspicious lines.

This work targets bug detection, with a focus on applying LLMs to analyze individual methods and assess whether they contain bugs. While some approaches are purely static (i.e., based on code alone), one variant incorporates runtime failure information, such as failed test cases and exception traces, to enrich the model's

reasoning. Although this introduces an element of fault analysis, the goal remains to determine whether a method is buggy, rather than to pinpoint the exact faulty line. A more detailed discussion of the prompting strategies is provided in Section 3.4.

2.2 LLMs Description

Pre-trained Language Models (PLMs) refer to language models trained on large-scale corpora that demonstrate strong performance across a variety of Natural Language Processing (NLP) tasks. When scaling up the number of parameters, these models show not only improved performance, but also new capabilities, such as in-context learning. To distinguish between models with different parameter sizes, researchers coined the term *Large Language Models* (LLMs). These models typically contain billions of parameters and are trained on massive text corpora, enhancing their ability to process natural language in a human-like manner [19]. Currently, there is no consensus in existing literature on a precise threshold that defines a language model as *large* [20].

The roots of LLMs date back to 1986, when Rumelhart et al. [21] introduced Recurrent Neural Networks (RNNs), opening up the possibility of processing sequential data by retaining previous input in internal states. In 1997, Hochreiter et al. [22] proposed Long Short-Term Memory networks (LSTMs), an extension of RNNs that addressed the long-range dependency problem by relying on a cell state and three types of gates. In 2003, Bengio et al. [23] introduced the concept of a language model based on a neural network, which paved the way for a new paradigm in natural language processing. In 2015, Dai et al. [24] expounded key ideas that underlie large language models, such as leveraging unsupervised pre-training followed by supervised fine-tuning. In 2017 Vaswani et al. [25] presented the Transformer network architecture, which weighs the importance of different parts of the input data using a self-attention mechanism. This was the innovation that led to the birth of modern Large Language Models as they are known today. LLMs are based on the Transformer architecture and follow a pre-training-and-fine-tuning approach. Firstly, they are trained on a large corpus of unlabeled data through unsupervised learning; then, they are fine-tuned for downstream tasks using a smaller amount of labeled data [26]. Some tasks lack high-quality fine-tuned data; even in these scenarios, LLMs are able to achieve good performance through prompt engineering [27]. This technique involves providing the model with natural language descriptions and demonstrations of the desired task before presenting the actual input. LLMs can be trained on both natural and programming languages, and some are capable of handling both [28].

2.2.1 Model Architectures

Large Language Models are usually classified into three categories based on their model architecture:

- **encoder-only LLMs**: these LLMs only use the encoder part of the Transformer architecture. Generally, these models are pre-trained with a Masked Language Modeling (MLM) approach, where a small percentage of tokens in the input are randomly masked. The model learns to predict the original values of these masked tokens based on their bidirectional context. Since these models only have an encoder component, which is able to generate context-aware representations for inputs, they are well-suited for code understanding tasks (i.e., code search), but are less suitable for code generation tasks (i.e., program repair). Models like CodeBERT [29] are built upon this architecture;

- **decoder-only LLMs**: these LLMs only use the decoder part of the Transformer architecture. Usually, these models are pre-trained using a Causal Language Modeling (CLM) approach, learning to predict the next token of a sequence based on previous tokens. Therefore, such LLMs are used to produce sequences of words in an autoregressive fashion, producing one token at a time and using what has been generated so far as context for subsequent tokens. Given this, these models are capable of naturally performing a variety of tasks just by relying on a few examples or proper instructions, without the need for fine-tuning. Notable model families belong to this category, such as GPT-series models, including the well-known ChatGPT [30];

- **encoder-decoder LLMs**: these LLMs utilize both the encoder and decoder parts of the Transformer architecture. The encoder's role is to capture the semantics and meaning of the input sequence by encoding it into a fixed-size hidden state. The decoder then produces the corresponding output by processing this hidden state and using attention mechanisms to refer to part of the input sequence when needed. As a result, these models are particularly well-suited for transforming one sequence into another, thus being used in tasks like program repair. CodeT5 [31] is among the models that is built on top of this architecture.

2.2.2 Optimization Approaches

LLMs typically acquire general knowledge from extensive datasets. A fundamental research question is how to adapt general-purpose models for specific domains and tasks to improve their performance. This goal can be achieved through two main strategies:

- **fine-tuning**: as briefly discussed earlier, fine-tuning refers to the process in which LLMs are further trained on a smaller, specific dataset, adjusting their parameters through supervised learning to improve performance on the desired task. Fine-tuning was (and remains) a widely used strategy, especially during the early emergence of LLMs, when off-the-shelf models with millions of parameters lacked sufficient capacity and required further adaptation to achieve better performance;

- **prompt engineering**: in recent years, there has been considerable research related to the generation of prompts (i.e., the inputs given to the models) that could improve LLMs' performance across different tasks. Modern LLMs demonstrate strong reasoning and generalization abilities, thanks to their exponential growth in size [32]. Therefore, researchers have focused on prompt engineering as a way to steer model behavior. This approach allows models to produce the desired outputs without modifying their internal weights. According to Wang et al.'s survey [20], the most commonly used approaches are the following:

  - **zero-shot learning**: this strategy consists of providing the model with a description of the desired task and requesting an output. The model solely rely on its prior knowledge and understanding to generate a proper response;
  - **few-shot learning**: with this approach [33], prompts include a task description along with a few examples containing both inputs and corresponding outputs. By observing these examples, the model can better understand what the expected answers are.
  - **chain-of-thought**: Chain-of-Thought (CoT) prompting [34] consists of generating a sequence of short sentences that describe the reasoning step-by-step, to guide the model towards the correct final answer. Usually, prompts include questions followed by reasoning and their respective results.

Chapter 3

Experimental Setup

Several datasets have been proposed in the literature to support empirical research in software testing. Notable examples include QuixBugs [35], TrickyBugs [36], and Vul4J [37], each targeting different types of bugs and evaluation settings. Among them, Defects4J [38] has emerged as one of the most widely adopted and influential benchmarks. Over the last decade, it has become a common standard within the research community, providing a consistent basis to reflect on the strengths and weaknesses of proposed approaches in the software testing field [26]. For this reason, this thesis relies on it as the evaluation dataset.

Section 3.1 describes the experimental methodology, while Section 3.2 gives an overview of the Defects4J dataset and explains the criteria used to create a filtered version of the database. Section 3.3 presents the characteristics of the large language models evaluated in this thesis. Section 3.4 describes the prompting strategies adopted to test the above-mentioned models. Finally, Section 3.5 describes the available hardware and software resources.

3.1 Methodology

Starting from the Defects4J dataset, a manual filtering operation is performed to extract only the bugs that fall within the scope of this research. Afterwards, each selected bug is manually analyzed to construct the ground truth files for each project. These files contain information about the buggy methods that meet the requirements defined in the previous phase. Specifically, for each method, the following information is reported: the method name, the name of the file in which it is located, and its occurrence within that file. Sobreira et al. [39] analyzed six projects in their study. Additionally, the fault localization dataset by Just et al. [40] provides a partial list of buggy methods from Defects4J. Both sources offer faster access to the data required for building the ground truth files.

After filtering out unrelated bugs, the resulting dataset is split into three partitions. In the first experimental phase, six open-source large language models are evaluated on the first partition of the dataset. They are prompted with both zero-shot and few-shot strategies, and the best-performing model is selected for the second phase. During this phase, the selected model is tested using nine different prompting approaches on the second dataset partition. The goal is to explore how performance can be improved, leveraging both standard strategies and auxiliary information commonly available when working in software testing, such as source code, exception traces, or failed test cases. Finally, the experiments are repeated on the third partition to validate and confirm the model's observed behavior.

3.2 Dataset Description

Defects4J [41] is a well-known dataset that provides a collection of reproducible bugs from real-world open-source programs written in the Java programming language. It also features a framework that facilitates the access to faulty and fixed program versions, along with their corresponding test suites. This framework provides a high-level interface that enables researchers to perform various tasks in software testing, such as test generation, test execution, and code coverage analysis.

Defects4J contains 854 bugs, plus 10 deprecated ones that are no longer reproducible due to behavioral changes introduced in newer Java versions. These bugs span 17 different open-source projects:

Identifier	Project name	Number of active bugs	Active bug ids	Deprecated bug ids
Chart	jfreechart	26	1-26	None
Cli	commons-cli	39	1-5, 7-40	6
Closure	closure-compiler	174	1-62, 64-92, 94-176	63, 93
Codec	commons-codec	18	1-18	None
Collections	commons-collections	28	1-28	None
Compress	commons-compress	47	1-47	None
Csv	commons-csv	16	1-16	None
Gson	gson	18	1-18	None
JacksonCore	jackson-core	26	1-26	None
JacksonDatabind	jackson-databind	110	1-64, 66-88, 90-112	65, 89
JacksonXml	jackson-dataformat-xml	6	1-6	None
Jsoup	jsoup	93	1-93	None
JxPath	commons-jxpath	22	1-22	None
Lang	commons-lang	61	1, 3-17, 19-24, 26-47, 49-65	2, 18, 25, 48
Math	commons-math	106	1-106	None
Mockito	mockito	38	1-38	None
Time	joda-time	26	1-20, 22-27	21

Table 3.1: Overview of Defects4J dataset

The goal of the dataset is to identify real bugs (i.e., bugs fixed by a developer) and to obtain, for each bug, a faulty and a fixed version that differ only by the bug fix. In particular, each bug of the Defects4J dataset meets the following requirements:

- **relation to source code**: the commit corresponding to the bug fix must be explicitly labeled as a bug-fixing commit, and the fix must directly affect source code. Bug fixes involving configuration files, tests, or documentation are excluded;
- **reproducibility**: the bug must be accompanied by at least one test that fails on the faulty version and succeeds on the fixed one. Additionally, the bug must be reproducible using the project's build system and a Java Virtual Machine (JVM);
- **isolation**: the only difference between the faulty and fixed versions of the bug must be the bug fix itself (that is, the bug fix does not include irrelevant changes, such as features or refactorings).

The following sections describe the operations performed on the dataset. Section 3.2.1 outlines the filtering procedure, while Section 3.2.2 explains how the dataset is partitioned.

3.2.1 Dataset Preparation

Since this thesis focuses on bug detection at the method level, the first step is to extract buggy methods from the dataset. In the Defects4J context, a *bug* refers to a set of related faults and the corresponding Java classes in which they occur. Each Java class contains multiple methods and may include more than one buggy method. Due to this discrepancy between *bugs* and *buggy methods*, the focus of this section will be on *buggy methods*, since they are the target of this evaluation.

To adhere to the research scope, only defects located within methods and constructors are considered, excluding bugs affecting other parts of the source code, such as static initialization blocks or missing imports.

The following list illustrates the types of bugs that are either included in or excluded from the dataset based on the filtering criteria:

- **bug within a method**

```
public LegendItemCollection getLegendItems() {
    (...)
    int index = this.plot.getIndexOf(this);
    CategoryDataset dataset = this.plot.getDataset(index);
    -if (dataset != null) {
    +if (dataset == null) {
        return result;
    } (...)
```

Listing 3.1: Example of included bug (project: Chart, bug: 1)

In the fixed version of the project, the line labeled with '-' is removed, while the one labeled with '+' is inserted. The bug is located within the body of the method `getLegendItems`, which is therefore labeled as a buggy method.

- **bug within a constructor**

```
protected BaseOptimizer(ConvergenceChecker<PAIR> checker) {
    this.checker = checker;

    evaluations = new Incrementor(0, new MaxEvalCallback());
    -iterations = new Incrementor(0, new MaxIterCallback());
    +iterations = new Incrementor(Integer.MAX_VALUE, new
   MaxIterCallback());
}
```

Listing 3.2: Example of included bug (project: Math, bug: 6)

Constructors are a special type of methods used to initialize objects and share syntactic similarities with regular methods. Due to this correspondence, they are included in the scope of this research.

- **bug outside methods or constructors**

```
(...)
protected final static DateFormat DATE_FORMAT_ISO8601;
protected final static DateFormat DATE_FORMAT_ISO8601_Z;
+protected final static DateFormat DATE_FORMAT_ISO8601_NO_TZ;
   // since 2.8.10

protected final static DateFormat DATE_FORMAT_PLAIN;
(...)
```

Listing 3.3: Example of excluded bug (project: JacksonDatabind, bug: 87)

This example illustrates a bug excluded during the preliminary filtering phase. The difference between the buggy and the fixed versions lies in the added line labeled with '+', which corresponds to a missing global static field declaration. The model is fed Java methods or constructors, but this error happens outside of these structures and is therefore discarded.

- **bug within a static initialization block**

```
-private static final String LANGUAGE_RULES_RN = "org/apache/
   commons/codec/language/bm/lang.txt";
+private static final String LANGUAGE_RULES_RN = "org/apache/
   commons/codec/language/bm/%s_lang.txt";

static {
    for (final NameType s : NameType.values()) {
```

```
        -Langs.put(s, loadFromResource(LANGUAGE_RULES_RN,
    Languages.getInstance(s)));
        +Langs.put(s, loadFromResource(String.format(
    LANGUAGE_RULES_RN, s.getName()), Languages.getInstance(s)))
    ;
     }
}
```

Listing 3.4: Example of excluded bug (project: Codec, bug: 14)

This example shows a bug within a static initialization block. Although these blocks share structural similarities with regular methods, they do not have explicit names by which they can be referenced. Initially, placeholder names were considered to label these cases; however, such bugs are often related to broader issues, such as missing static imports, as seen in this example, making them less suitable for isolated evaluation. For this reason, they were excluded by the filtering operation. In total, nine such bugs were identified across the projects, and they are listed in Table 3.2:

Codec	**JacksonDatabind**	**Jsoup**	**Mockito**	**Time**	**Total**
1	4	2	1	1	**9**

Table 3.2: Number of bugs within static initialization blocks across the dataset

Using the dataset's framework, a manual comparison between faulty and fixed versions of each bug in the dataset has been performed. These comparisons led to the extraction of each buggy method. The number of bugs excluded by the filtering operation is reported in the following list:

- **Chart**: 1
- **Cli**: 2
- **Closure**: 1
- **Codec**: 2
- **Collections**: 4
- **Compress**: -
- **Csv**: 1
- **Gson**: -
- **JacksonCore**: 1
- **JacksonDatabind**: 8
- **JacksonXml**: 1
- **Jsoup**: 4
- **JxPath**: -
- **Lang**: 2
- **Math**: 2
- **Mockito**: 1
- **Time**: 1
- **Total**: 31

Figure 3.1: Number of bugs excluded by the filtering operation

As reported in Figure 3.1, the filtering operation excluded 31 bugs out of the

original 854. The *JacksonDatabind* project was the most affected, with 8 bugs removed, while some other projects remained unaltered.

Table 3.3 shows the final composition of the filtered dataset:

Identifier	Number of bugs	Number of buggy methods
Chart	25	39
Cli	37	52
Closure	173	313
Codec	16	21
Collections	24	44
Compress	47	74
Csv	15	17
Gson	18	29
JacksonCore	25	56
JacksonDatabind	102	202
JacksonXml	5	5
Jsoup	89	159
JxPath	22	44
Lang	59	87
Math	104	149
Mockito	37	78
Time	25	51
Total	**823**	**1420**

Table 3.3: Final state of the dataset after the filter

One project stands out for its size: *Closure*. In total, four projects contain more than 100 buggy methods, which are *Closure*, *JacksonDatabind*, *Jsoup*, and *Math*. The other projects comprise tens of buggy methods, ranging from 17 to 87. The smallest project is *JacksonXml*, which includes only 5 buggy methods.

3.2.2 Dataset Partitioning

The outcome of the filtering phase is shown in Table 3.3. This filtered dataset serves as the foundation of the experiments conducted in this thesis. As described in the methodology (see Section 3.1), the dataset is divided into three partitions. The goal is to create dataset partitions that are as balanced as possible in terms of the total number of *bugs* and *buggy methods*. To achieve this, a greedy partitioning algorithm is applied. Projects are sorted in descending order based on the sum of their *bugs* and *buggy methods* (a higher number of *bugs* implies a larger number of total methods). They are then sequentially assigned to the partition with the

currently smallest total sum, aiming to balance both buggy and total methods across partitions.

Table 3.4 lists each project along with the combined total of *bugs* and *buggy methods*:

Identifier	**Sum of bugs and buggy methods**
Closure	486
JacksonDatabind	304
Math	253
Jsoup	248
Lang	146
Compress	121
Mockito	115
Cli	89
JacksonCore	81
Time	76
Collections	68
JxPath	66
Chart	64
Gson	47
Codec	37
Csv	32
JacksonXml	10

Table 3.4: Combined total of bugs and buggy methods per project

The algorithm is outlined below:

Greedy Partitioning Algorithm

Require: Project list $P = \{p_1, p_2, \ldots, p_n\}$ with associated weights $w(p_i)$
Ensure: Three balanced partitions P_1, P_2, P_3
1: Sort P in descending order by $w(p_i)$
2: Initialize partitions $P_1 \leftarrow \emptyset$, $P_2 \leftarrow \emptyset$, $P_3 \leftarrow \emptyset$
3: **for all** project p in P **do**
4: Identify partition P_k with minimum total weight
5: Assign project p to P_k
6: **end for**
7: **return** P_1, P_2, P_3

A few execution steps are reported below. Each entry follows the format: Partition(Projects)[Total sum of bugs and buggy methods]:

- **Starting point**: **P1**( )[0]; **P2**( )[0]; **P3**( )[0]
- **Step 1**: **P1**((Closure, 486))[486]; **P2**( )[0]; **P3**( )[0]
- **Step 2**: **P1**((Closure, 486))[486]; **P2**((JacksonDatabind, 304))[304]; **P3**( )[0]
- **Step 3**: **P1**((Closure, 486))[486]; **P2**((JacksonDatabind, 304))[304]; **P3**((Math, 253))[253]
- **Step 4**: **P1**((Closure, 486))[486]; **P2**((JacksonDatabind, 304))[304]; **P3**((Math, 253), (Jsoup, 248))[501]
- (...)

Based on the data in Table 3.4, the application of the above-mentioned greedy algorithm produces the following three partitions:

	Partition 1	**Partition 2**	**Partition 3**
Projects	Closure Mockito Collections Gson Codec	JacksonDatabind Lang Compress JacksonCore JxPath Csv	Math Jsoup Cli Time Chart JacksonXml
Bugs	268	270	285
Buggy methods	485	480	455
Total	**753**	**750**	**740**

Table 3.5: Three resulting partitions

The adoption of this strategy results in three partitions that are well-balanced in terms of both *bugs* and *buggy methods*. The maximum imbalance between any two partitions is only 17, and the difference in buggy methods is at most 30, making these partitions suitable for comparative analysis in the subsequent experiments.

3.3 Model Descriptions

As reported in [26], [20], and [42], the majority of studies conducted in the software testing research area rely on GPT models, like ChatGPT [30] and Codex [43]. In order to broaden the evaluation in this field, six different open-source models from the Ollama [44] library are tested. These models are chosen according to the available hardware resources described in Section 3.5.

This section presents the characteristics of each model, along with a description of the parameters that are kept consistent across all models.

3.3.1 Common Parameters

To ensure a fair comparison, some parameters are kept the same across all model evaluations:

- **instruction tuning**: the instruction-following variant (`instruct`) is used for each model;
- **quantization**: all models are run in 4-bit standard quantized mode (`q4_0`);
- **temperature**: each model uses the default temperature setting to ensure a uniform degree of randomness in the generated answers.

3.3.2 Models List

To ensure diversity regarding model size, two *small* models (between 1B and 25B parameters), two *medium* models (between 25B and 50B parameters), and two *big* models (above 50B parameters) were selected. Moreover, three of them are specifically tailored for code-related tasks, while the remaining represent general-purpose models.

The models evaluated within this thesis are reported in the following list. The *default size* refers to the memory occupation of models with a context window of 2048 tokens, which is the default value set by Ollama.

- **Codellama**

  Codellama is a family of code-based LLMs built on top of Llama2 architecture. It supports code generation and discussion across multiple programming languages, such as Java, Python, and C++. It is available in several parameter sizes (7B, 13B, 34B, and 70B), each released in three variants:

  - **instruct**: an instruction-following variant fine-tuned to follow user prompts;
  - **python**: a specialized variant further fine-tuned on Python code;
  - **code**: a variant optimized for code completion.

  The version used in this thesis is the following:

  - **name**: `codellama:34b-instruct-q4_0`
  - **parameters**: 33.7B

  - **maximum context length**: 16384 tokens
  - **default size**: 19 GB

- **Codestral**

  Codestral is a model designed for code generation tasks, such as writing tests or completing partially written code, that leverages a fill-in-the-middle mechanism. It has 22B parameters.

  The version used in this thesis is the following:

  - **name**: `codestral:22b-v0.1-q4_0`
  - **parameters**: 22.2B
  - **maximum context length**: 32768 tokens
  - **default size**: 13 GB

- **Llama3.1**

  Llama3.1 is a family of models available in three different sizes: 8B, 70B, and 405B. These models are known for their strong reasoning capabilities as well as their proficiency on multilingual translation and in-depth general knowledge.

  The two versions used in this thesis are the following:

  - Version 1:
    * **name**: `llama3.1:8b-instruct-q4_0`
    * **parameters**: 8.03B
    * **maximum context length**: 131072 tokens
    * **default size**: 4.7 GB
  - Version 2:
    * **name**: `llama3.1:70b-instruct-q4_0`
    * **parameters**: 70.6B
    * **maximum context length**: 131072 tokens
    * **default size**: 40 GB

- **Llama3.3**

  Llama3.3 is a pre-trained, instruction-tuned generative model with 70B parameters.

  The version used in this thesis is the following:

  - **name**: `llama3.3:70b-instruct-q4_0`
  - **parameters**: 70.6B

  - **maximum context length**: 131072 tokens
  - **default size**: 40 GB

- **Qwen2.5-Coder**

  Qwen2.5-Coder is a family of models available in various sizes: 0.5B, 1.5B, 3B, 7B, 14B, and 32B. They are particularly recommended for tasks like code generation, code repair, and code reasoning, which is the ability to predict the model's inputs and outputs by learning the process of code execution. These models support multiple programming languages, including Java, Rust, and Haskell.

  The version used in this thesis is the following:

  - **name**: `qwen2.5-coder:32b-instruct-q4_0`
  - **parameters**: 32.8B
  - **maximum context length**: 32768 tokens
  - **default size**: 19 GB

The following table sums up the characteristics of the chosen models:

Name	Version	Quantization	Parameters	Max. Context Length	Size
codellama	instruct	q4_0	33.7B	16384	19 GB
codestral	v0.1	q4_0	22.2B	32768	13 GB
llama3.1	instruct	q4_0	8.03B	131072	4.7 GB
llama3.1	instruct	q4_0	70.6B	131072	40 GB
llama3.3	instruct	q4_0	70.6B	131072	40 GB
qwen2.5-coder	instruct	q4_0	32.8B	32768	19 GB

Table 3.6: Characteristics of the tested models

3.4 Prompting Strategies

As briefly introduced in Section 2.2.2, various strategies exist for generating input prompts aimed at improving the performance of LLMs. This thesis explores nine different approaches, described in the following list:

- **Zero-Shot**

  Listing 3.5 shows the zero-shot prompt used in the experiments. It begins by setting the context for the model: it is tasked with analyzing Java methods to detect potential bugs. Then, it instructs the model to evaluate a specific method (indicated by `method_name`) and determine whether it contains any

bugs. The model is asked to produce a binary decision and is warned not to assume that all methods are buggy, which helps mitigate potential biases in its output. The source code of the method under evaluation is then included. Finally, the model is instructed to generate output in JSON format to ensure structured responses, which facilitate automatic evaluation of the results.

```
You are analyzing Java methods to determine whether they
    contain bugs.

Analyze the method named '{method_name}' to identify any bugs.
If you find any, set "has_bug" to true; otherwise, set "
    has_bug" to false. Do not assume that all methods have bugs
    .

Method code:
{method_code}

Provide your response as a JSON object with the following
    structure:
{
\t"method_name": "string",
\t"has_bug": true or false
}
Do not include any other comments or explanations.
```

Listing 3.5: Zero-Shot prompt

The next prompts are based on the structure of this zero-shot prompt.

- **Zero-Shot-Motivation**

  The prompt shown in Listing 3.6 extends the basic zero-shot formulation by requiring the model to provide a short justification for its decision. While the overall structure remains the same, the model is now asked to include a brief motivation within the JSON response object. This requirement encourages the model to explicitly articulate its internal decision-making process, even though the explanation is expected to remain concise. This approach, that could be seen as a lightweight version of Chain-of-Thought strategy (see Listing 3.12), aims to improve both prediction quality and transparency, offering insights into whether the model's rationale aligns with human expectations.

```
You are analyzing Java methods to determine whether they
    contain bugs.
Analyze the method named '{method_name}' to identify any bugs.
If you find any, set "has_bug" to true; otherwise, set "
    has_bug" to false. Do not assume that all methods have bugs
    .
Explain briefly your reasoning in the "motivation" field.
```

```
Method code:
{method_code}

Provide your response as a JSON object with the following
    structure:
{
\t"method_name": "string",
\t"has_bug": true or false,
\t"motivation": "string"
}
Do not include any other comments or explanations.
```

Listing 3.6: Zero-Shot-Motivation prompt

- **Zero-Shot-Whole-Class prompt**

  Listing 3.7 shows the first prompt that investigates the impact of adding additional context on the model's performance. In particular, the model is fed the source code of the entire Java class from which the target method is extracted. It is explicitly stated that this serves as context to aid the model's analysis. However, the evaluation still focuses on a single method, as enforced by the prompt. This strategy aims to determine the influence of a realistic class-level code environment on the model's ability to detect possible bugs.

```
You are analyzing Java methods to determine whether they
    contain bugs.

The full Java class code is provided below to give you context
    :
{class_code}

Now focus on the method named '{method_name}' extracted from
    the class above. Analyze this method to identify any bugs.
If you find any, set "has_bug" to true; otherwise, set "
    has_bug" to false. Do not assume that all methods have bugs
    .

Method code:
{method_code}

Provide your response as a JSON object with the following
    structure:
{
\t"method_name": "string",
\t"has_bug": true or false
}
Do not include any other comments or explanations.
```

Listing 3.7: Zero-Shot-Whole-Class prompt

- **Zero-Shot-Exceptions prompt**

  The prompt shown in Listing 3.8 is the second variant to provide contextual information, this time in the form of runtime failure reports. Specifically, the prompt includes information about the failing test cases and the exceptions raised when running the test suite on the class containing the target method. These details are extracted by parsing the output of the `defects4j info` command (see Section 4.1.2 for an example). The model is asked to determine whether the method under evaluation could be responsible for the observed failures. This approach investigates the role of dynamic feedback (i.e., failing behavior from test suites) and mimics real-world debugging scenarios, in which developers start from failing test cases to locate faults in the source code.

```
You are analyzing Java methods to determine whether they
   contain bugs.

Below is a report of the exception(s) or assertion failure(s)
   triggered by executing the test suite for this class:
{exception_info}

Now analyze the method named '{method_name}' from this class
   and determine whether it could be responsible for the
   failure(s) described above.
If you find a bug in this method, set "has_bug" to true;
   otherwise, set "has_bug" to false. Do not assume that all
   methods have bugs.

Method code:
{method_code}

Provide your response as a JSON object with the following
   structure:
{
\t"method_name": "string",
\t"has_bug": true or false
}
Do not include any other comments or explanations.
```

Listing 3.8: Zero-Shot-Exceptions prompt

- **Call Graph prompt**

  The prompt shown in Listing 3.9 introduces contextual information in the form of a call graph. Alongside the method code, it also provides the source code of

the methods directly called by the method under investigation. This context is introduced solely as supporting information to aid the model's understanding. The prompt clearly states that these called methods should not be evaluated. This approach aims to help the model reason about a method's behavior in relation to its dependencies, potentially offering key insights for detecting bugs. Section 4.1.3 describes how call graphs are constructed.

```
You are analyzing Java methods to determine whether they
   contain bugs.

Analyze the method named '{method_name}' to identify any bugs.
If you find any, set "has_bug" to true; otherwise, set "
   has_bug" to false. Do not assume that all methods have bugs
   .
To help you better understand how this method works, the code
   of the methods it directly calls is also provided. These
   called methods are included for context only --- do not
   evaluate them for bugs.

Method code:
{method_code}

Called methods (context only):
{method_calls}

Provide your response as a JSON object with the following
   structure:
{
\t"method_name": "string",
\t"has_bug": true or false
}
Do not include any other comments or explanations.
```

Listing 3.9: Call Graph prompt

- **Few-Shot prompt**

  Listing 3.10 presents the first prompt built using a few-shot learning strategy. While the overall structure mirrors that of the previous prompts, this prompt also includes six illustrative examples of Java methods, each paired with its corresponding evaluation and a brief justification. These examples are designed to implicitly teach the model the expected reasoning process and output format. This approach tests whether in-context learning [45], which is the ability of LLMs to learn from examples provided directly in the input prompt, can enhance the model's performance by evaluating its ability to generalize from a small set of labeled instances. The examples included in the prompt are reported in Section 4.1.1.

```
You are analyzing Java methods to determine whether they
    contain bugs.

Below are six examples of Java methods with explanations on
    whether they contain bugs and their corresponding
    evaluations:
{examples}

Now analyze the method named '{method_name}' to identify any
    bugs.
If you find any, set "has_bug" to true; otherwise, set "
    has_bug" to false. Do not assume that all methods have bugs
    .

Method code:
{method_code}

Provide your response as a JSON object with the following
    structure:
{
\t"method_name": "string",
\t"has_bug": true or false
}
Do not include any other comments or explanations.
```

Listing 3.10: Few-Shot prompt

- **Few-Shot-Motivation prompt**

  The prompt displayed in Listing 3.11 builds upon the few-shot structure by incorporating the same enhancement introduced in Listing 3.6. The model is required to provide a brief justification for its prediction by filling the additional `motivation` field of the JSON object. This formulation combines the benefits of few-shot learning and lightweight reasoning, potentially leading to better performance.

```
You are analyzing Java methods to determine whether they
    contain bugs.

Below are six examples of Java methods with explanations on
    whether they contain bugs and their corresponding
    evaluations:
{examples}

Now analyze the method named '{method_name}' to identify any
    bugs.
```

```
If you find any, set "has_bug" to true; otherwise, set "
   has_bug" to false. Do not assume that all methods have bugs
   .
Explain briefly your reasoning in the "motivation" field.

Method code:
{method_code}
Provide your response as a JSON object with the following
   structure:
{
\t"method_name": "string",
\t"has_bug": true or false,
\t"motivation": "string"
}
Do not include any other comments or explanations.
```

Listing 3.11: Few-Shot-Motivation prompt

- **Chain-of-Thought Zero-Shot prompt**

  Listing 3.12 introduces the first Chain-of-Thought (CoT) prompting strategy, which explicitly instructs the model to perform step-by-step reasoning before reaching a final decision. Unlike previous similar approaches, where only a brief explanation was required (see Listings 3.6 and 3.11), this prompt clearly encourages the model to reflect on various aspects of the target method (such as control flow and data handling) and to reason aloud before answering. This step-by-step reasoning can reduce hallucination (i.e., when a model generate nonsensical or unsupported responses [46]) and improve the model's performance.

```
You are analyzing Java methods to determine whether they
   contain bugs.

Analyze the method named '{method_name}' step-by-step to
   identify any bugs. If you find any, set "has_bug" to true;
   otherwise, set "has_bug" to false.
Think carefully about the control flow, data handling, method
   calls, and edge cases. Reason aloud before making a final
   decision. Do not assume that all methods have bugs.

Method code:
{method_code}

Explain your step-by-step reasoning in the "reasoning" field.
Provide your response as a JSON object with the following
   structure:
{
\t"method_name": "string",
```

```
\t"has_bug": true or false,
\t"reasoning": "string"
}
Do not include any other comments or explanations.
```

Listing 3.12: Zero-Shot Chain-of-Thought prompt

- **Chain-of-Thought Few-Shot prompt**

  Listing 3.13 presents the final tested approach: a few-shot version of the previous CoT prompt. Here, the instructions on how to conduct the reasoning are preceded by a set of examples. To ensure a fair comparison among the approaches, the few-shot examples are the same as those used in previous few-shot strategies, but the explanations are replaced with step-by-step reasoning, to align with the model's expected output. This approach aims to combine the benefits of both in-context learning and structured reasoning, potentially guiding the model towards more accurate predictions.

```
You are analyzing Java methods to determine whether they
   contain bugs.

Below are six examples of Java methods with step-by-step
   reasoning and final evaluations:
{examples}

Now analyze the method named '{method_name}' step-by-step to
   identify any bugs. If you find any, set "has_bug" to true;
   otherwise, set "has_bug" to false.
Think carefully about the control flow, data handling, method
   calls, and edge cases. Reason aloud before making a final
   decision. Do not assume that all methods have bugs.

Method code:
{method_code}

Explain your step-by-step reasoning in the "reasoning" field.
Provide your response as a JSON object with the following
   structure:
{
\t"method_name": "string",
\t"has_bug": true or false,
\t"reasoning": "string"
}
Do not include any other comments or explanations.
```

Listing 3.13: Few-Shot Chain-of-Thought prompt

3.5 Computational Environment

The experiments are conducted on the nvCluster infrastructure at TU Graz, a computing cluster comprising four compute nodes. Each node features eight Nvidia Quadro RTX 8000 GPUs, each providing 48 GB of memory. These GPUs are designed for high-performance workloads and large-scale inference tasks, making them well-suited for the demands of LLM evaluation. More information on their specifications is available in the official datasheet [47].

On the software side, the experimental pipeline is implemented primarily in Python. Model interaction is performed via the `Ollama` client, which communicates with the examined large language models.

Java code analysis is enabled using `JavaParser` through `JPype`, allowing seamless integration between Java and Python. This setup is used to extract individual methods and constructors from source files for model evaluation.

Chapter 4

Results And Discussion

This chapter reports the results obtained through the conducted experiments. Section 4.1 describes the selection of the examples used in the few-shot prompting approaches and the construction of call graphs. It also presents the evaluation metrics adopted to measure the models' performance and an example of exception-related information. The results obtained on the three dataset partitions are reported and discussed in Sections 4.2, 4.3 and 4.4. Section 4.5 discusses the potential threats to the validity of this study.

4.1 Implementation Details

As reported in Section 3.2.1, in the Defects4J context, each *bug* represents one or more Java classes that contain one or more actual defects. Since this analysis focuses on method-level bug detection, the experiments are conducted by extracting individual methods from the buggy classes and prompting the models with each method separately. The corresponding JavaDoc comment (if present) immediately preceding each method is retained during extraction, ensuring that the model receives not only the method implementation but also its accompanying documentation. This can provide valuable semantic context that may aid in bug detection.

To ensure a fair comparison among the evaluated models, the context window size is uniformly set to 16384 tokens, which is the smallest maximum context length supported across all models considered in this study, as reported in Table 3.6. Table 4.1 reports the models' memory usage with the context window set to 16384 tokens. Each model fits within the available resources described in Section 3.5.

Model	Size
Codellama:34b-instruct-q4_0	41GB
Codestral:22b-v0.1-q4_0	35GB
Llama3.1:8b-instruct-q4_0	8.3GB
Llama3.1:70b-instruct-q4_0	48GB
Llama3.3:70b-instruct-q4_0	48GB
Qwen2.5-Coder:32b-instruct-q4_0	41GB

Table 4.1: Models size with 16384 tokens context window

The experiments follow the evaluation workflow described in Section 3.1, where models are tested across three dataset partitions and multiple prompting strategies.

4.1.1 Few-Shot Examples

The zero-shot approach measures the inherent capabilities of the models, hence serving as the baseline for the experiments. Regarding the few-shot approach, the examples are selected from actual buggy methods within the Defects4J dataset, ensuring alignment with the methods being evaluated. In order to avoid data leakage, the examples used in the prompts for Partitions 1 and 2 are selected from projects in Partition 3, while those used for Partition 3 are picked from projects in Partitions 1 and 2. With the aid of the Defects4J dissection by Sobreira et al. [39], the selection process aims to cover a diverse range of bug types. The number of examples included in each prompt is empirically set to six, with an equal split between buggy and non-buggy methods, to avoid biasing the model's behavior. Since Defects4J bugs are non-trivial, each example shows, alongside its code and evaluation, a brief explanation justifying the label. This additional reasoning is meant to help the model understand why a method is considered buggy or not, thereby preserving the effectiveness of the few-shot approach.

Listing 4.1 shows one of the few-shot examples actually used in this thesis, along with both its evaluation and explanation, to clarify how such examples are presented to the model. The entire source code of the method is omitted due to space constraints.

```
Example 1
Method code:
(...)
if (p1.getWindingRule() != p2.getWindingRule()) {
    return false;
}
PathIterator iterator1 = p1.getPathIterator(null);
PathIterator iterator2 = p1.getPathIterator(null);
double[] d1 = new double[6];
```

```
double[] d2 = new double[6];
(...)
Result:
{
    "method_name": "equal",
    "has_bug": true
}
Explanation: This method is intended to compare two 'GeneralPath'
   objects for equality. However, it mistakenly uses 'p1.
   getPathIterator(null)' for both 'iterator1' and 'iterator2',
   meaning it compares 'p1' to itself instead of comparing 'p1' to
    'p2'. This logic causes the method to always return true when
   'p1.equals(p1)', even if 'p1' and 'p2' are different.
```

Listing 4.1: Few-shot example used in the prompt (Project: Chart; Bug: 11)

The characteristics of the selected examples are reported in Tables 4.2 and 4.3. The column *Repair Patterns* refers to the terminology used in the above-mentioned dissection [48].

Project	**Bug Number**	**Repair Patterns**	**Version**
Chart	11	Wrong variable reference Single line	Buggy
Math	80	Arithmetic expression modification Single line	Fixed
Chart	15	Conditional block addition with return statement Missing non-null check addition Missing null check addition Wraps-with if statement	Buggy
Time	15	Conditional block addition with exception throwing	Buggy
Math	53	Conditional block addition with return statement	Fixed
Math	92	Conditional block addition with exception throwing Conditional block addition Conditional block removal Conditional block addition with return statement Arithmetic expression modification Wrong method reference	Fixed

Table 4.2: Few-shot examples for Partitions 1 and 2

The repair patterns reported in Table 4.3 follow the same categorization used in Table 4.2.

Project	Bug Number	Repair Patterns	Version
Lang	61	Arithmetic expression modification Single line Wrong variable reference	Buggy
Closure	97	Arithmetic expression modification Single line	Fixed
Mockito	4	Conditional block addition with return statement Missing null check addition Wraps-with method call	Buggy
Lang	11	Conditional block addition with exception throwing Conditional block addition	Buggy
Lang	49	Conditional block addition Conditional block addition with return statement	Fixed
Closure	32	Conditional block addition Conditional block removal	Fixed

Table 4.3: Few-shot examples for Partition 3

4.1.2 Exceptions Information

As described in the Zero-Shot-Exceptions prompt (see Listing 3.8), the information regarding failed test cases and the corresponding raised exceptions is parsed from the output of the `defects4j info` command. Listing 4.2 contains an example of how this information is inserted into the prompt. To maintain consistency with the previous example, the following entry also refers to bug 11 of the *Chart* project:

```
(...)

Below is a report of the exception(s) or assertion failure(s)
   triggered by executing the test suite for this class:
Test case: ShapeUtilitiesTests::testEqualGeneralPaths
Failure: AssertionFailedError

(...)
```

Listing 4.2: Exception information used in the prompt (Project: Chart; Bug: 11)

4.1.3 Call Graph Construction

A call graph is a data structure that represents invocation relationships between functions (or methods) within a program. Call graphs can be built in two directions: downstream graphs (also known as callee graphs), which represent all the methods invoked by the target method, and upstream (caller) graphs, which capture all the methods that invoke the target method.

In this thesis, call graphs are used as part of the call graph prompting approach (see Listing 3.9), and are generated using the Soot [49] framework for static analysis of Java programs. This analysis only focuses on downstream calls, i.e., method calls from the target method to other methods, as they provide insights into the internal logic of the method being analyzed.

Since LLMs are already trained on vast amounts of code, each call graph includes only method calls that target other methods within the same project, excluding external calls, such as those to the standard library or third-party dependencies. This design choice ensures that the model focuses only on project-specific logic, which is more relevant in this context.

The following example, extracted from bug 2 of the *Csv* project, illustrates this situation:

```
Method code:

/**
 * Checks whether a given column is mapped and has a value.
 *
 * @param name
 *            the name of the column to be retrieved.
 * @return whether a given columns is mapped.
 */
public boolean isSet(final String name) {
    return isMapped(name) && mapping.get(name).intValue() < values
   .length;
}

Called methods:

/**
 * Checks whether a given column is mapped.
 *
 * @param name
 *            the name of the column to be retrieved.
 * @return whether a given columns is mapped.
 */
public boolean isMapped(final String name) {
    return mapping != null ? mapping.containsKey(name) : false;
}
```

Listing 4.3: Generation of a call graph (Project: Csv; Bug: 2)

The call graph of the `isSet` method shown in Listing 4.3 includes only the method `isMapped`, as it is the only method that belongs to the same project. Other method calls, such as `get` and `intValue`, are excluded since they refer to standard library classes (e.g., `Map` and `Integer`).

4.1.4 Evaluation Metrics

Since this bug detection analysis can be seen as a binary classification task, model performance is evaluated using precision, recall, and F1 score (which is the harmonic mean of the two former metrics). These metrics rely on the standard classification outcomes:

- **True Positives (TP)**: correctly identified buggy methods;
- **False Positives (FP)**: non-buggy methods that are incorrectly labeled as buggy;
- **False Negatives (FN)**: buggy methods that are incorrectly labeled as non-buggy;
- **True Negatives (TN)**: correctly identified non-buggy methods.

The metrics are defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

95% Confidence Intervals (CI) are computed to estimate the variability of the results. They are calculated using the following formula:

$$\text{CI}_{95\%} = \bar{x} \pm t_{0.025,n-1} \cdot \frac{s}{\sqrt{n}}$$

where:

- $\bar{x}$ is the sample mean across runs;
- s is the sample standard deviation;
- n is the number of runs (which are five in this context);
- $t_{0.025,n-1}$ is the critical value from the Student's distribution with *n-1* degrees of freedom.

In order to mitigate the intrinsic randomness of LLMs, each experiment is repeated across five independent runs. The scores obtained by each run are aggregated using `NumPy` and `SciPy`. The values reported in the following tables represent the arithmetic mean of each metric across those five runs.

As reported in Tables 4.5, 4.22, and 4.33, the proportion of buggy methods relative to the total number of methods is low for every project (only one project exceeds a 10% ratio). This implies that models have to evaluate a large number of non-buggy methods, while only a few are actually buggy. This significant class imbalance increases the difficulty of the task, making low precision values (and consequently low F1 scores) expected, since even a small number of false positives can significantly impact the metric.

4.2 Partition 1

As briefly reported in Section 4.1, the six selected models are tested on the projects from Partition 1 using two approaches: zero-shot and few-shot. The two prompts are shown in Listings 3.5 and 3.10. The goal of this initial evaluation is to identify the best-performing model, which will be used for subsequent experiments.

For clarity in the result tables, Table 4.4 provides the shorthand codes assigned to each model:

Code	Model
CDL	`Codellama:34b-instruct-q4_0`
CST	`Codestral:22b-v0.1-q4_0`
LM8	`Llama3.1:8b-instruct-q4_0`
LM70	`Llama3.1:70b-instruct-q4_0`
LM3	`Llama3.3:70b-instruct-q4_0`
QWC	`Qwen2.5-Coder:32b-instruct-q4_0`

Table 4.4: Codes assigned to each model

Table 4.5 reports the number of buggy and total methods for each project in Partition 1, along with the percentage of buggy methods.

Project	Number of buggy methods	Number of total methods	Ratio (%)
Codec	21	460	4.57%
Gson	29	627	4.63%
Collections	44	981	4.49%
Mockito	78	630	12.4%
Closure	313	10016	3.12%

Table 4.5: Partition 1 project weights

4.2.1 Zero-Shot Approach

Table 4.6 shows the performance of the six models on the *Codec* project using the zero-shot strategy. CDL and CST achieve relatively high precision (14.12% and 19.30%, respectively), but perform very poorly in recall, particularly CST, which has the lowest recall across all models (8.57%). LM8 achieves the highest recall (70.48%) but suffers from the lowest precision (6.20%), resulting in the lowest F1 score (11.40%). LM70, LM3, and QWC exhibit more balanced performance, with F1 scores of 13.69%, 12.95%, and 14.32%, respectively.

Model	Precision	Recall	F1 Score
CDL	14.12% ± 7.76	21.90% ± 12.26	**17.16% ± 9.48**
CST	**19.30% ± 9.52**	8.57% ± 4.95	11.84% ± 6.54
LM8	6.20% ± 0.74	**70.48% ± 7.71**	11.40% ± 1.35
LM70	7.94% ± 1.74	49.52% ± 10.74	13.69% ± 2.98
LM3	8.04% ± 1.13	33.33% ± 4.18	12.95% ± 1.76
QWC	9.36% ± 1.60	30.48% ± 5.29	14.32% ± 2.44

Table 4.6: Project: **Codec**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.7 reports the performance of the six models on the *Gson* project. The results highlight the difficulty of this project: every model achieves low values in terms of F1 score. CDL results in a modest F1 score (6.94%), reporting moderate precision (6.52%) and poor recall (7.59%). LM8 and LM70 achieve higher recall values (42.07% and 38.62%, respectively), but are penalized by their low precision, resulting in modest F1 scores (7.70% and 8.78%, respectively). LM3 offers the best F1 score (10.80%), benefiting from a slightly better trade-off between precision (6.61%) and recall (29.66%). CST and QWC are the worst-performing models on this project, reporting the lowest F1 scores (4.93% and 4.18%, respectively).

Model	Precision	Recall	F1 Score
CDL	6.52% ± 4.23	7.59% ± 5.58	6.94% ± 4.65
CST	3.40% ± 1.82	8.97% ± 4.88	4.93% ± 2.66
LM8	4.24% ± 1.03	**42.07% ± 9.76**	7.70% ± 1.86
LM70	4.95% ± 1.06	38.62% ± 7.04	8.78% ± 1.84
LM3	**6.61% ± 1.19**	29.66% ± 6.49	**10.80% ± 2.01**
QWC	2.62% ± 0.78	10.34% ± 3.03	4.18% ± 1.24

Table 4.7: Project: **Gson**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.8 displays the performance of the six models on the *Collections* project. Overall, performance remains modest across all models, with F1 scores below 14%. CST achieves the highest F1 score (13.94%) thanks to the top precision (10.35%), while its recall is the second-lowest (21.36%). LM8 still gets the best recall (75.91%), but at the same time it exhibits the worst precision (5.43%), resulting in an F1 score of 10.14%. LM70, LM3, and QWC show modest results in terms of recall (each above 43%) and are penalized by low precision values (around 6%). CDL demonstrates poor performance with balanced yet low precision and recall (7.60% and 10.45%, respectively), resulting in the lowest F1 score (8.79%) among all models.

Model	Precision	Recall	F1 Score
CDL	7.60% ± 3.52	10.45% ± 4.72	8.79% ± 3.99
CST	**10.35% ± 1.03**	21.36% ± 2.52	**13.94% ± 1.41**
LM8	5.43% ± 0.48	**75.91% ± 7.08**	10.14% ± 0.90
LM70	5.92% ± 0.71	57.73% ± 8.13	10.74% ± 1.31
LM3	5.74% ± 0.38	43.64% ± 3.09	10.14% ± 0.68
QWC	6.69% ± 0.60	48.64% ± 4.72	11.76% ± 1.07

Table 4.8: Project: **Collections**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.9 reports the results obtained by the six models on the *Mockito* project. This time, LM8 achieves the highest F1 score (26.70%), thanks to its leading recall (50.77%) and solid precision (18.12%). LM70 reports the highest precision (18.79%), but, due to its modest recall (36.67%), it results in an F1 score of 24.83%. LM3 and QWC follow closely, with F1 scores of 21.10% and 19.29%, respectively, showing slightly lower values for precision and recall. While CDL exhibits a relatively high precision (18.13%), its recall is the lowest across all models (8.21%), resulting in the weakest F1 score (11.27%). CST pairs the lowest precision (12.61%) with poor

recall (16.41%), achieving a modest F1 score of 14.26%. As shown in Table 4.5, this project features the highest proportion of buggy methods relative to total methods in Partition 1, which likely contributes to higher precision values, especially compared to other projects.

Model	Precision	Recall	F1 Score
CDL	18.13% ± 6.51	8.21% ± 3.49	11.27% ± 4.53
CST	12.61% ± 2.45	16.41% ± 3.45	14.26% ± 2.84
LM8	18.12% ± 0.87	**50.77% ± 0.87**	**26.70% ± 1.01**
LM70	**18.79% ± 3.14**	36.67% ± 5.91	24.83% ± 3.99
LM3	15.56% ± 1.16	32.82% ± 2.89	21.10% ± 1.58
QWC	15.56% ± 0.72	25.38% ± 1.74	19.29% ± 1.04

Table 4.9: Project: **Mockito**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.10 presents the performance of the six models on the *Closure* project, which is the last one in Partition 1. Overall, F1 scores are low across all models, highlighting the challenges posed by the project's size and the large number of analyzed methods. CDL achieves the highest F1 score (9.53%), despite modest values for both precision (7.90%) and recall (12.01%). LM8, LM70, and LM3 show relatively high recall, ranging from 35% to 52%, but their F1 scores remain below 8% due to poor precision. QWC, despite a lower recall (19.62%) than those models, exhibits a better F1 score (7.78%), benefiting from a more balanced trade-off between precision and recall. CST is by far the worst-performing model, with both precision and recall below 3%, yielding an F1 score of just 2.29%.

Model	Precision	Recall	F1 Score
CDL	**7.90% ± 1.69**	12.01% ± 3.11	**9.53% ± 2.20**
CST	1.87% ± 0.44	2.94% ± 0.65	2.29% ± 0.53
LM8	3.81% ± 0.15	**51.63% ± 2.11**	7.10% ± 0.28
LM70	4.21% ± 0.19	40.89% ± 2.30	7.63% ± 0.34
LM3	4.09% ± 0.30	35.78% ± 2.59	7.34% ± 0.54
QWC	4.85% ± 0.25	19.62% ± 1.14	7.78% ± 0.42

Table 4.10: Project: **Closure**; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.11 presents the weighted average results for the zero-shot strategy across all projects in Partition 1. The weights are based on the total number of methods per project, as reported in Table 4.5, to reflect each project's relative size and contribution to the evaluation.

The results show that CDL achieves the highest weighted F1 score (9.71%) thanks to the highest precision (8.54%), despite a low recall (11.84%).

LM8 reaches the highest recall (53.67%) by a large margin, but is penalized by its low precision (4.75%), the second-lowest among all models. A similar trend is observed for LM70, LM3, and QWC, which show good recall but suffer from limited precision, resulting in comparable F1 scores around 9%.

CST is the worst-performing model, with the lowest values in precision (3.77%), recall (5.53%) and F1 score (4.26%). Moreover, its confidence intervals are the highest among all models, indicating that its performance is not only poor but also highly inconsistent across runs. This highlights the model's unreliability in the zero-shot setting.

Model	Precision	Recall	F1 Score
CDL	**8.54% ± 3.11**	11.84% ± 2.88	**9.71% ± 2.00**
CST	3.77% ± 5.38	5.53% ± 6.96	4.26% ± 5.17
LM8	4.75% ± 3.86	**53.67% ± 9.51**	8.49% ± 5.34
LM70	5.23% ± 3.97	42.18% ± 6.09	9.00% ± 4.80
LM3	5.05% ± 3.21	35.85% ± 3.35	8.61% ± 3.95
QWC	5.58% ± 3.14	22.08% ± 10.32	8.71% ± 3.75

Table 4.11: Weighted Averages; Approach: **Zero-shot**; Format: value ± confidence interval; Best results in **bold**

4.2.2 Few-Shot Approach

Ablation study: position of the explanation in few-shot examples

As reported in Section 4.1.1, each few-shot example includes the method's source code, its evaluation (i.e., whether it is buggy or not) and a brief explanation justifying the assigned label. In order to determine the optimal placement of the explanation (either before or after the classification result), a small ablation study is conducted on a subset of Partition 1 projects (specifically *Codec*, *Gson*, *Collections*, and *Mockito*), considering all models except `Codestral`, due to its poor performance.

Table 4.12 summarizes the results of this study, reporting the obtained F1 scores over five independent runs. Within each cell, the upper value represents the configuration where the explanation is placed *before* the classification result, while the bottom value corresponds to the explanation placed *after* the result.

The complete results for each metric are reported in Section A.1 of Appendix A.

P	CDL	LM8	LM70	LM3	QWC
1	9.83% ± 5.76 11.76% ± 2.14	9.68% ± 2.93 9.37% ± 2.86	13.77% ± 3.29 12.95% ± 2.94	12.72% ± 3.61 11.24% ± 1.75	10.08% ± 2.71 12.21% ± 3.78
2	9.66% ± 5.31 9.08% ± 2.31	6.24% ± 2.28 7.42% ± 2.64	7.68% ± 5.27 10.69 ± 1.57%	6.54% ± 3.29 5.59% ± 1.82	5.92% ± 0.98 7.55% ± 1.98
3	9.73% ± 3.49 9.16% ± 1.70	9.86% ± 1.18 9.90% ± 1.87	11.98% ± 1.37 11.21% ± 1.54	15.45% ± 2.27 14.62% ± 1.49	12.27% ± 1.43 13.33% ± 1.61
4	13.95% ± 5.63 20.10% ± 7.88	19.79% ± 2.24 21.88% ± 1.72	23.52% ± 3.60 24.32% ± 2.91	17.53% ± 1.11 17.14% ± 3.55	22.81% ± 0.98 22.28% ± 1.97
5	10.72% ± 2.84 **12.14% ± 7.16**	11.31% ± 7.79 **12.03% ± 8.78**	13.98% ± 8.99 **14.45% ± 8.75**	**13.40% ± 6.47** 12.53% ± 6.75	12.88% ± 9.51 **13.88% ± 8.17**

Table 4.12: F1 scores for different position of the explanation (Top: explanation *before* the `has_bug` field; Bottom: explanation *after* the `has_bug` field); (**P**: Projects; **1**: Codec; **2**: Gson; **3**: Collections; **4**: Mockito; **5**: Weighted Average); Format: value ± confidence interval; Best results in **bold**

The results show that positioning the explanation *after* the result generally leads to better performance, as four out of the five tested models achieve higher F1 scores with this configuration. It also tends to produce narrower confidence intervals, making the model's behavior more consistent across repeated runs. In addition to its empirical advantages, this ordering mimics the natural reasoning flow: evaluation first, justification after. Based on this analysis, the configuration in which the explanation follows the result is adopted in all subsequent few-shot experiments.

Results

Table 4.13 presents the results of the six models on the *Codec* project under the few-shot prompting strategy. LM70 achieves the best F1 score (12.95%), combining a moderate precision (7.61%) with a high recall (43.81%). Both CDL and QWC obtain competitive F1 scores (11.76% and 12.21%, respectively), though their recall is lower than that of LM70. LM3 has a slightly lower precision than LM70 (7.19% vs. 7.61%) and lower recall (25.71%), leading to lower F1 score (11.24%). LM8 obtains the highest recall (44.76%) even in this few-shot setting, but it is again penalized by a low precision value (5.23%), which results in a modest F1 score of 9.37%. Interestingly, CST shows the highest average precision (60.00%), but it is accompanied by an extremely large confidence interval (±68.01) and a very low recall (4.76%), leading to the lowest F1 score (8.77%) among all models. This suggests that CST may be highly inconsistent in its predictions across runs.

Model	Precision	Recall	F1 Score
CDL	7.85% ± 1.38	23.81% ± 5.91	11.76% ± 2.14
CST	**60.00% ± 68.01**	4.76% ± 5.91	8.77% ± 10.80
LM8	5.23% ± 1.62	**44.76% ± 12.26**	9.37% ± 2.86
LM70	7.61% ± 1.81	43.81% ± 7.71	**12.95% ± 2.94**
LM3	7.19% ± 1.19	25.71% ± 3.24	11.24% ± 1.75
QWC	8.11% ± 2.38	24.76% ± 8.77	12.21% ± 3.78

Table 4.13: Project: **Codec**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.14 shows the performance of the six models on the *Gson* project. Overall, all models obtain low scores, confirming the struggle on this project already observed with the zero-shot setting (see Table 4.7). LM70 achieves the highest F1 score (10.69%), thanks to a relatively strong recall (40.69%) and a modest precision (6.15%). This time, LM8 does not have the highest recall (33.10%) and its precision remains low (4.18%), resulting in a modest F1 score of 7.42%. CDL and QWC obtain similar results, with the same recall (20.00%) and comparable precision values (5.88% and 4.65%, respectively). LM3 performs worse with this project, with both low precision (3.61%) and recall (12.41%), leading to an F1 score of just 5.59%. CST completely fails to identify any buggy methods across all five runs, resulting in 0% score for each metric. This suggests either a complete lack of generalization from the few-shot examples or a failure in its prediction strategy for this specific project.

Model	Precision	Recall	F1 Score
CDL	5.88% ± 1.50	20.00% ± 5.58	9.08% ± 2.31
CST	0.00% ± 0.00	0.00% ± 0.00	0.00% ± 0.00
LM8	4.18% ± 1.49	33.10% ± 11.57	7.42% ± 2.64
LM70	**6.15% ± 0.94**	**40.69% ± 4.69**	**10.69% ± 1.57**
LM3	3.61% ± 1.19	12.41% ± 3.83	5.59% ± 1.82
QWC	4.65% ± 1.25	20.00% ± 4.69	7.55% ± 1.98

Table 4.14: Project: **Gson**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.15 reports the results of the six models on the *Collections* project. This time LM3 exhibits the best F1 score (14.62%), with strong results both in precision (8.45%, the second-highest) and recall (54.09%). QWC follows closely with an F1 score of 13.33%, demonstrating a good balance between precision (7.56%) and recall (56.36%). LM8 and LM70 perform well in terms of recall (64.09% and 57.73%,

respectively), but low precision values (around 6%) limit their F1 scores to 9.90% and 11.21%, respectively. CDL delivers a modest F1 score (9.16%), driven mainly by its low recall (20.45%). CST continues to be the worst-performing model, with consistently low values across all three metrics.

Model	Precision	Recall	F1 Score
CDL	5.90% ± 1.08	20.45% ± 3.99	9.16% ± 1.70
CST	**8.51% ± 3.32**	6.82% ± 2.00	7.55% ± 2.51
LM8	5.36% ± 1.02	**64.09% ± 12.20**	9.90% ± 1.87
LM70	6.21% ± 0.86	57.73% ± 7.88	11.21% ± 1.54
LM3	8.45% ± 0.86	54.09% ± 6.12	**14.62% ± 1.49**
QWC	7.56% ± 0.89	56.36% ± 7.83	13.33% ± 1.61

Table 4.15: Project: **Collections**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.16 reports the results obtained by the six models on the *Mockito* project. LM70 obtains the best F1 score (24.32%), showing a good balance between precision (19.24%) and recall (33.08%). LM8 and QWC also perform well (with F1 scores of 21.88% and 22.28%, respectively), with LM8 once again achieving the highest recall (37.18%). CDL offers a solid performance with the F1 score of 20.10%, thanks to the highest precision value (20.24%) across all models. LM3, while having lower recall (19.23%) compared to the others, still obtains a reasonable F1 score (17.14%). CST remains the weakest performer across these models, with the lowest precision (13.74%), recall (3.08%), and F1 score (5.02%).

Model	Precision	Recall	F1 Score
CDL	**20.24% ± 7.37**	20.00% ± 8.47	20.10% ± 7.88
CST	13.74% ± 4.46	3.08% ± 1.42	5.02% ± 2.18
LM8	15.51% ± 1.34	**37.18% ± 2.52**	21.88% ± 1.72
LM70	19.24% ± 2.17	33.08% ± 4.42	**24.32% ± 2.91**
LM3	15.47% ± 3.18	19.23% ± 4.06	17.14% ± 3.55
QWC	18.00% ± 1.73	29.23% ± 2.36	22.28% ± 1.97

Table 4.16: Project: **Mockito**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

Table 4.17 shows the performance of the six models on the *Closure* project. This time, LM3 has the highest F1 score (9.32%), combining a decent recall (34.82%) with moderate precision (5.38%). LM70 achieves the best recall (47.99%) across all models, while LM8 struggles, with a recall of 30.99%, low precision (3.27%), and an F1 score of 5.92%. CDL and QWC obtain similar F1 scores (between 8%

and 9%), benefiting from a decent precision (5.60% and 4.96%, respectively). CST, once again, shows the worst performance across models, with extremely low values for precision (2.99%), recall (1.66%), and, consequentially, F1 score (2.14%).

Model	Precision	Recall	F1 Score
CDL	**5.60% ± 0.97**	21.60% ± 4.28	8.89% ± 1.57
CST	2.99% ± 0.57	1.66% ± 0.33	2.14% ± 0.41
LM8	3.27% ± 0.10	30.99% ± 1.09	5.92% ± 0.19
LM70	5.03% ± 0.34	**47.99% ± 3.22**	9.10% ± 0.62
LM3	5.38% ± 0.44	34.82% ± 2.72	**9.32% ± 0.76**
QWC	4.96% ± 0.33	23.96% ± 1.32	8.22% ± 0.53

Table 4.17: Project: **Closure**; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

Finally, Table 4.18 reports the weighted average performance of the six models across all Partition 1 projects under the few-shot setting. The weights are based on the total number of methods for each project reported in Table 4.5.

Among all models, LM70 achieves the best F1 score (10.24%), having the highest recall (47.49%) and a moderate precision (5.97%).

LM3 and CDL follow closely, as they combine the highest values for precision (6.10% and 6.44%, respectively) with moderate recall, resulting in F1 scores of 10.00% and 9.58%, respectively. QWC exhibits a slightly lower F1 score (9.43%), due to its modest precision (5.91%) and recall (26.56%).

LM8 performs poorly in precision (4.16%) and F1 score (7.22%), despite having a moderate recall (34.45%).

CST is confirmed as the worst-performing model, with particularly low values in recall and F1 score (2.16% and 2.83%, respectively).

Model	Precision	Recall	F1 Score
CDL	**6.44% ± 3.95**	21.43% ± 0.88	9.58% ± 3.06
CST	5.87% ± 13.48	2.16% ± 1.92	2.83% ± 2.49
LM8	4.16% ± 3.32	34.45% ± 11.21	7.22% ± 4.43
LM70	5.97% ± 3.83	**47.49% ± 5.74**	**10.24% ± 4.15**
LM3	6.10% ± 2.92	34.10% ± 10.23	10.00% ± 2.92
QWC	5.91% ± 3.60	26.56% ± 10.86	9.43% ± 4.11

Table 4.18: Weighted Averages; Approach: **Few-shot**; Format: value ± confidence interval; Best results in **bold**

4.2.3 Processing Time

To provide further insights into the characteristics of the models, the time required to process each method under both prompting strategies has been measured. The average processing time per method is expressed as the number of seconds required to process a single method, and is computed with the following formula:

$$\text{Processing Time (s/method)} = \frac{\text{Total time (s)}}{\text{Total number of processed methods}}$$

Table 4.19 reports the average processing time per method for each model under the zero-shot setting. As expected, smaller models (CST and LM8) are the fastest, with average latencies below 1.2 seconds per method. On the other hand, larger models like LM70 and LM3 require more than twice as long on average, with latencies over 2.5 seconds per method.

Model	Codec	Gson	Collections	Mockito	Closure	Average
CDL	1.53	1.42	1.37	1.43	2.04	1.56
CST	1.09	1.12	1.01	1.03	1.58	1.17
LM8	0.82	0.85	0.61	0.88	1.14	**0.86**
LM70	2.67	2.56	2.46	2.83	2.87	2.68
LM3	2.57	2.45	2.22	2.17	3.10	2.50
QWC	1.70	1.61	1.37	2.21	2.45	1.87

Table 4.19: Average processing time per method on Partition 1 projects with zero-shot approach; Results expressed in s/m (seconds/method)

Table 4.20 presents the corresponding values for the few-shot approach. As expected, this strategy results in a slightly slower processing across all models due to longer prompts. Despite this, the relative ranking of models remains the same, with smaller models that continue to be the fastest, while larger ones remain the slowest. Overall, these values highlight a clear trade-off between model complexity and execution time.

Model	Codec	Gson	Collections	Mockito	Closure	Average
CDL	1.72	1.60	1.60	1.60	1.79	1.66
CST	1.30	1.22	1.21	1.23	1.42	1.28
LM8	0.88	0.69	0.94	0.71	1.10	**0.86**
LM70	3.19	3.03	2.91	2.93	3.20	3.05
LM3	3.03	2.88	2.74	2.67	2.99	2.86
QWC	1.94	1.79	1.80	1.77	2.09	1.88

Table 4.20: Average processing time per method on Partition 1 projects with few-shot approach; Results expressed in s/m (seconds/method)

4.2.4 Model Choice

Figure 4.1 compares the precision scores obtained by each model under zero-shot and few-shot configurations across Partition 1 projects. In general, precision values remain low across all models, with all scores falling below 10%. Only two models (CDL and LM8) experience a drop in precision when transitioning from the zero-shot to the few-shot setting. The other models show slight improvements, with CST exhibiting the most notable gain (+2.1%). However, the overall impact of changing the prompting strategy remains limited. CDL stands out as the most precise model in both configurations, achieving 8.54% in the zero-shot and 6.44% in the few-shot setting. The other models perform similarly, with values ranging from around 4% to 6% for both configurations.

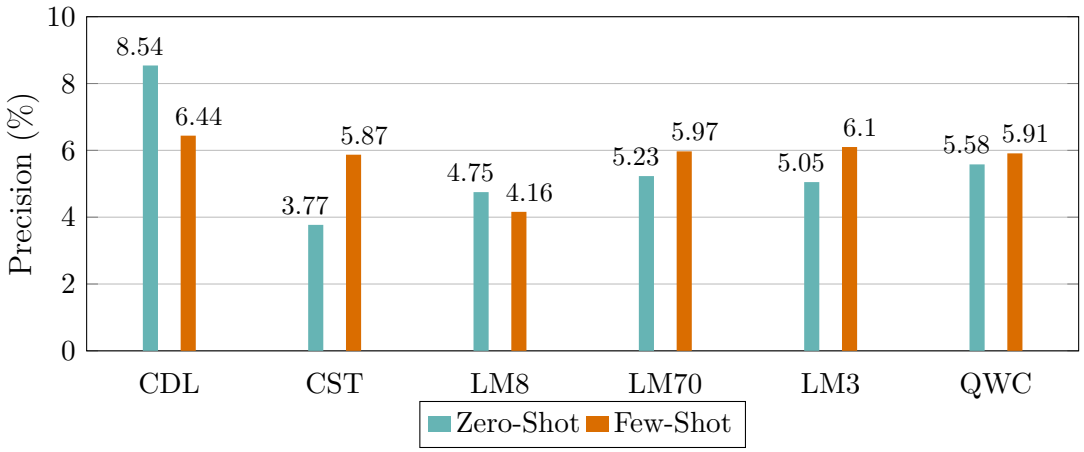

Figure 4.1: Zero-Shot vs. Few-shot approach on Partition 1 projects (Precision)

Figure 4.2 shows the recall scores achieved by each model under zero-shot and few-shot settings across Partition 1 projects. In contrast to precision, recall values

show greater variability and, in most cases, higher absolute values. LM8 and LM70 obtain the best results in the zero-shot setting, reaching 53.67% and 42.18%, respectively. Under the few-shot configuration, LM70 further improves to 47.49%, achieving the highest recall overall, while LM8 experiences a notable decrease to 34.45%. LM3 also records a slight reduction in performance. CDL and QWC show moderate improvements when moving from zero-shot to few-shot, although their scores remain modes, ranging between 10% and 20%. CST consistently underperforms across both settings with scores below 6%.

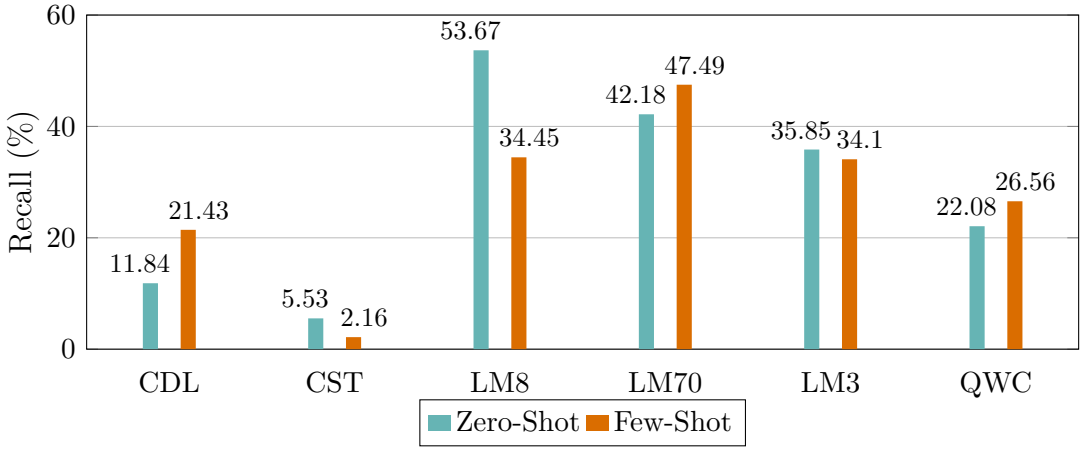

Figure 4.2: Zero-Shot vs. Few-shot approach on Partition 1 projects (Recall)

Figure 4.3 displays the F1 scores achieved by each model under zero-shot and few-shot configuration across Partition 1 projects. CDL produces consistent results across both settings, although its performance reflects a balanced trade-off between relatively low precision and recall values, rather than strong results in either metric. LM8 performs better under the zero-shot configuration, with a drop when moving to the few-shot setting. LM70 demonstrates the most consistent and reliable performance, improving from 9.00% in the zero-shot setting to 10.24% in the few-shot one, which is the highest F1 score overall. LM3 and QWC follow similar trends, each showing a modest improvement with the few-shot strategy, though their scores remain slightly lower than LM70's ones. CST is confirmed as the weakest performer across all models, with both F1 scores remaining below 5%.

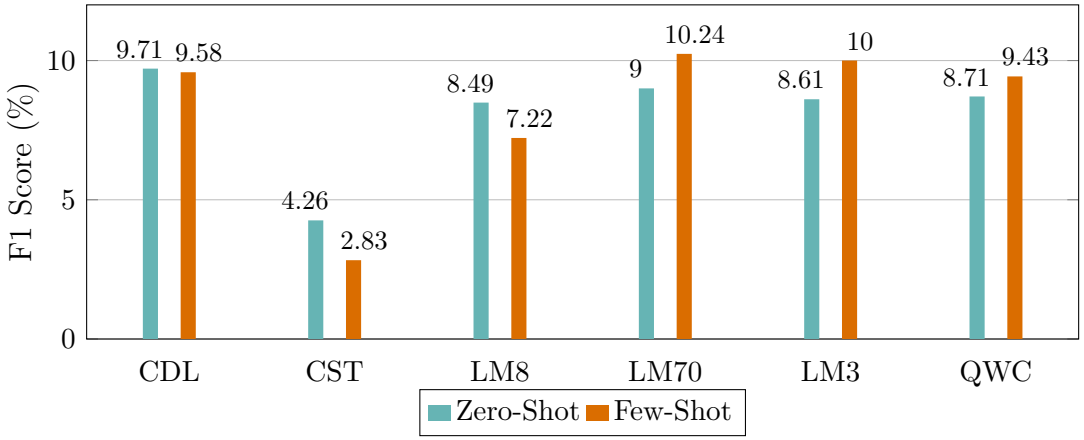

Figure 4.3: Zero-Shot vs. Few-shot approach on Partition 1 projects (F1 Score)

Given these results, **LM70** stands out as the best-performing model overall, offering strong and reliable F1 scores in both prompting scenarios. Therefore, it is selected for further experiments, despite being the slowest model, as reported in Section 4.2.3.

4.3 Partition 2

For clarity in the result tables, each prompting strategy (listed in Section 3.4) is associated with a corresponding code, as reported in Table 4.21:

Code	Prompting Strategy
ZS	Zero-Shot
ZS-Mot	Zero-Shot-Motivation
ZS-Class	Zero-Shot-Whole-Class
ZS-Exc	Zero-Shot-Exceptions
CG	Call Graph
FS	Few-Shot
FS-Mot	Few-Shot-Motivation
CoT-ZS	Chain-of-Thought Zero-Shot
CoT-FS	Chain-of-Thought Few-Shot

Table 4.21: Code assigned to each prompting strategy

Table 4.22 shows the number of buggy and total methods for each project in Partition 2, along with the percentage of buggy methods.

Project	Number of buggy methods	Number of total methods	Ratio (%)
Csv	17	382	4.45%
JxPath	44	730	6.03%
JacksonCore	56	1889	2.96%
Compress	74	1368	5.41%
Lang	87	3483	2.50%
JacksonDatabind	202	5252	3.85%

Table 4.22: Partition 2 project weights

Among the evaluated prompting strategies, two approaches require special consideration due to their structural constraints: zero-shot-whole-class and call graph (see Listings 3.7 and 3.9).

In the zero-shot-whole-class approach, the entire Java class is included in the prompt. To accomodate this, the context window is increased from 16384 to 20480 tokens, which is the maximum supported by the LM70 model on a single GPU. However, some classes still exceed this limit and are thus excluded from the evaluation. Table 4.23 presents the number of buggy and total methods after excluding the affected classes. Three projects are impacted: *JacksonCore*, *Lang*, and *JacksonDatabind*.

Project	Number of buggy methods	Number of total methods	Ratio (%)
Csv	17	382	4.45%
JxPath	44	730	6.03%
JacksonCore	38	1140	3.33%
Compress	74	1368	5.41%
Lang	70	2034	3.44%
JacksonDatabind	196	4441	4.41%

Table 4.23: Partition 2 project weights for the zero-shot-whole-class approach

In the call graph approach, only methods with a valid call graph representation are taken into account. Since not all methods invoke other methods within the same project, those without a valid call graph are excluded. Table 4.24 shows the number of buggy and total methods under this setting, with all the projects being affected.

Project	Number of buggy methods	Number of total methods	Ratio (%)
Csv	11	258	4.26%
JxPath	44	552	7.97%
JacksonCore	49	1498	3.27%
Compress	60	857	7.00%
Lang	53	1791	2.59%
JacksonDatabind	188	4174	4.50%

Table 4.24: Partition 2 project weights for the call graph approach

These two approaches are marked with an asterisk (*) in the following tables.

4.3.1 Results

Table 4.25 presents the results obtained by LM70 on the *Csv* project. ZS-Exc (zero-shot with exception context) achieves the best overall performance, with a precision of 15.62%, an outstanding recall of 80.00%, and the highest F1 score (26.13%), nearly doubling that of the second-best approach. Among the other zero-shot approaches, ZS-Mot (which asks the model to briefly justify its decision) outperforms the basic zero-shot setting, achieving a more stable recall (62.35%) and a higher F1 score (13.80% vs. 11.61%). The call graph variant performs slightly better than the basic ZS setting, with an F1 score of 12.80%, while the setting with the whole class inserted into the prompt has the lowest F1 score (10.97%) among zero-shot variants. Few-shot variants follow a similar trend: the motivated version (FS-Mot) outperforms the plain few-shot setup, with a consistently higher recall (58.82% vs. 27.06%) and F1 score (13.50% vs. 9.50%). Chain-of-Thought approaches, which incorporate step-by-step reasoning, yield modest recall performance, with both zero-shot and few-shot variants reaching only 34.12%. The zero-shot variant gains a higher F1 score than the few-shot one (12.59% against 11.88%) thanks to a slightly higher precision (7.73% vs. 7.19%).

Approach	Precision	Recall	F1 Score
ZS	6.73% ± 2.83	42.35% ± 20.92	11.61% ± 5.00
ZS-Mot	7.76% ± 0.66	62.35% ± 4.00	13.80% ± 1.11
ZS-Class*	6.58% ± 2.64	32.94% ± 12.22	10.97% ± 4.33
ZS-Exc	**15.62% ± 1.14**	**80.00% ± 8.33**	**26.13% ± 2.01**
CG*	7.63% ± 2.96	40.00% ± 15.14	12.80% ± 4.93
FS	5.76% ± 1.97	27.06% ± 8.33	9.50% ± 3.18
FS-Mot	7.63% ± 1.93	58.82% ± 13.66	13.50% ± 3.37
CoT-ZS	7.73% ± 1.29	34.12% ± 6.11	12.59% ± 2.07
CoT-FS	7.19% ± 1.96	34.12% ± 9.52	11.88% ± 3.22

Table 4.25: Project: **Csv**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.26 shows the results obtained by the LM70 model on the *JxPath* project. The trend observed in the *Csv* project continues here. The ZS-Exc strategy stands out once again, having the highest F1 score (21.00%) and the best precision (12.57%). The call graph approach (CG) also performs strongly, achieving an F1 score of 16.14% thanks to a solid recall of 61.36%. ZS-Mot shows a 10% improvement in recall over the zero-shot baseline, but their F1 scores are nearly identical (a difference of just 0.01%) due ZS-Mot's lower precision. ZS-Class performs better than in the previous case, with an F1 score of 13.95%. Among few-shot variants, FS-Mot slightly improves over the basic few-shot setup (13.32% vs. 12.84%), mainly due to its recall, which is the highest across all the approaches (74.09%). CoT-ZS is the weakest-performing approach, combining the lowest precision (6.55%) with modest recall (43.18%), which results in the lowest F1 score (11.37%). This time the few-shot CoT variant obtains a higher F1 score (13.44%) due to better precision and recall (7.57% and 60.00%, respectively).

Approach	Precision	Recall	F1 Score
ZS	6.87% ± 1.59	52.73% ± 12.53	12.16% ± 2.82
ZS-Mot	6.73% ± 0.49	62.73% ± 4.28	12.15% ± 0.88
ZS-Class*	8.09% ± 0.96	50.45% ± 5.05	13.95% ± 1.60
ZS-Exc	**12.57% ± 0.70**	63.64% ± 3.46	**21.00% ± 1.12**
CG*	9.30% ± 1.43	61.36% ± 10.56	16.14% ± 2.51
FS	7.27% ± 0.90	55.00% ± 7.30	12.84% ± 1.60
FS-Mot	7.32% ± 1.19	**74.09% ± 10.67**	13.32% ± 2.15
CoT-ZS	6.55% ± 0.97	43.18% ± 6.91	11.37% ± 1.69
CoT-FS	7.57% ± 1.60	60.00% ± 12.56	13.44% ± 2.84

Table 4.26: Project: **JxPath**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.27 presents the results obtained on the *JacksonCore* project using the LM70 model. Overall, the scores are noticeably lower than in previous projects, likely due to project-specific challenges, such as subtle bug patterns or code structures that are more difficult for the model to interpret. ZS-Exc is confirmed as the best-performing approach, achieving the highest F1-score (11.55%) by combining the best precision (6.24%) with the best recall (77.14%). All the other approaches struggle with precision, showing values around 4%. ZS-Class and CG both perform relatively well, with F1 scores between 8% and 9%, and achieving solid recall values (50.00% and 59.18%, respectively). Baseline zero-shot and few-shot approaches offer modest performance, with F1 scores of 6.81% and 7.27% each. Their motivation variants (ZS-Mot and FS-Mot) reach similar F1 scores, at least with better recall (55.00% and 63.21%, respectively). The zero-shot Chain-of-Thought approach yields a modest F1 score (6.95%), penalized by poor precision (3.85%) and low recall (35.36%). However, the worst-performing approach is the few-shot version of the CoT strategy, which records the lowest scores in precision, recall, and F1 score (3.17%, 34.29% and 5.81%, respectively).

Approach	Precision	Recall	F1 Score
ZS	3.67% ± 0.46	47.14% ± 6.76	6.81% ± 0.86
ZS-Mot	3.56% ± 0.49	55.00% ± 7.42	6.69% ± 0.91
ZS-Class*	4.88% ± 0.88	50.00% ± 8.00	8.89% ± 1.57
ZS-Exc	**6.24% ± 0.45**	**77.14% ± 5.06**	**11.55% ± 0.83**
CG*	4.33% ± 0.55	59.18% ± 8.40	8.07% ± 1.04
FS	3.94% ± 0.26	47.14% ± 2.53	7.27% ± 0.47
FS-Mot	3.71% ± 0.42	63.21% ± 7.45	7.02% ± 0.80
CoT-ZS	3.85% ± 0.75	35.36% ± 7.08	6.95% ± 1.35
CoT-FS	3.17% ± 0.74	34.29% ± 8.21	5.81% ± 1.36

Table 4.27: Project: **JacksonCore**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.28 displays the results obtained by the LM70 model on the *Compress* project. The best result is once again achieved by the ZS-Exc strategy, which reaches an F1 score of 18.93%, driven by its strong recall (62.43%) and the highest precision (11.15%). Other context-enriched strategies also perform well. Call graph and ZS-Class reach good F1 scores, 14.30% and 12.63%, respectively. As in previous cases, ZS-Mot achieves higher recall than the baseline zero-shot approach (57.30% vs. 45.68%), but, due to its lower precision, it results in similar F1 scores (around 12%). The same goes for the few-shot counterparts: the baseline has a lower recall than the motivation-oriented approach, but their F1 scores are comparable (13.50% vs. 13.11%), as the baseline few-shot has higher precision. Interestingly, both versions of the CoT strategy continue to struggle, recording the lowest values in precision (6.32% for the zero-shot version, 5.99% for the few-shot one), recall (31.62% and 35.41%, respectively) and, consequently, F1 score (10.54% and 10.25%, respectively).

Approach	Precision	Recall	F1 Score
ZS	7.06% ± 0.92	45.68% ± 5.09	12.23% ± 1.55
ZS-Mot	6.76% ± 0.76	57.30% ± 7.28	12.10% ± 1.37
ZS-Class*	7.21% ± 1.37	50.81% ± 10.04	12.63% ± 2.40
ZS-Exc	**11.15% ± 0.86**	62.43% ± 5.62	**18.93% ± 1.49**
CG*	8.34% ± 1.12	50.00% ± 6.21	14.30% ± 1.89
FS	7.77% ± 0.76	51.35% ± 6.28	13.50% ± 1.36
FS-Mot	7.26% ± 0.38	**67.84% ± 4.34**	13.11% ± 0.68
CoT-ZS	6.32% ± 0.42	31.62% ± 3.48	10.54% ± 0.77
CoT-FS	5.99% ± 0.92	35.41% ± 6.21	10.25% ± 1.60

Table 4.28: Project: **Compress**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.29 reports the results obtained on the *Lang* project by the LM70 model. The trend observed in previous projects holds, with ZS-Exc once again emerging as the best-performing approach. It achieves a remarkable recall of 83.45%, along with the highest precision (9.29%) and F1 score (16.72%). The call graph strategy also performs well, with an F1 score of 14.79%, combining good precision (8.53%) and recall (55.47%). ZS-Class achieves slightly lower performance, with an F1 score of 10.98%, although showing a modest precision (6.09%). The baseline approaches (zero-shot and few-shot) remain stable but unremarkable, with F1 scores around 9%. Their motivation-enhanced versions consistently achieve higher recall (64.14% for ZS-Mot and 65.06% for FS-Mot), but are again limited by lower precision, which results in similar F1 scores. CoT-ZS exhibits a good F1 score (9.52%), although having the lowest recall (38.39%) across all approaches. The few-shot CoT version also struggles on this project, showing the weakest F1 score (8.01%) due to low precision (4.40%) and modest recall (44.83%).

Approach	Precision	Recall	F1 Score
ZS	4.79% ± 0.43	51.03% ± 4.69	8.77% ± 0.79
ZS-Mot	4.72% ± 0.50	64.14% ± 7.44	8.80% ± 0.94
ZS-Class*	6.09% ± 0.63	56.00% ± 5.23	10.98% ± 1.12
ZS-Exc	**9.29% ± 0.57**	**83.45% ± 6.27**	**16.72% ± 1.04**
CG*	8.53% ± 1.10	55.47% ± 8.54	14.79% ± 1.95
FS	5.12% ± 0.77	55.63% ± 8.11	9.37% ± 1.40
FS-Mot	4.18% ± 0.32	65.06% ± 5.85	7.86% ± 0.62
CoT-ZS	5.43% ± 0.77	38.39% ± 5.94	9.52% ± 1.36
CoT-FS	4.40% ± 0.75	44.83% ± 8.86	8.01% ± 1.38

Table 4.29: Project: **Lang**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.30 presents the results obtained by the LM70 model on the *Jackson-Databind* project. The best-performing approach is once again ZS-Exc, with an F1 score of 17.81%, by combining the highest recall (60.30%) with the strongest precision (10.45%). ZS-Class and CG also perform well, reaching F1 scores of 13.45% and 13.74%, respectively. Baseline methods deliver solid yet low performance, with F1 scores around 12%, mainly due to moderate recall (both reach only 41.58%) and limited precision. FS-Mot achieves a high recall (62.57%) but is penalized by low precision (6.40%), resulting in an F1 score of 11.61%, which is lower than its corresponding baseline strategy. The same is observed for ZS-Mot, which achieves a modest F1 score of (9.69%), lower than the basic zero-shot approach. CoT-ZS records both the lowest recall and F1 score (25.74% and 9.24%, respectively), while also having the second-lowest precision (5.63%; only ZS-Mot performs worse, with 5.43%). The corresponding few-shot version performs slightly better, though obtaining a weak recall (40.30%) and a modest F1 score (10.27%).

Approach	Precision	Recall	F1 Score
ZS	6.65% ± 0.66	41.58% ± 4.60	11.47% ± 1.15
ZS-Mot	5.43% ± 0.27	44.75% ± 2.56	9.69% ± 0.49
ZS-Class*	7.78% ± 0.49	49.49% ± 2.72	13.45% ± 0.83
ZS-Exc	**10.45% ± 0.45**	60.30% ± 2.44	**17.81% ± 0.75**
CG*	8.15% ± 0.68	43.94% ± 4.30	13.74% ± 1.18
FS	7.29% ± 0.42	41.58% ± 2.61	12.40% ± 0.71
FS-Mot	6.40% ± 0.19	**62.57% ± 1.54**	11.61% ± 0.34
CoT-ZS	5.63% ± 0.30	25.74% ± 1.68	9.24% ± 0.52
CoT-FS	5.89% ± 0.69	40.30% ± 4.45	10.27% ± 1.19

Table 4.30: Project: **JacksonDatabind**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.31 reports the weighted average performance of the LM70 model across all projects in Partition 2. The weights are based on the total number of methods of each project reported in Tables 4.22, 4.23, and 4.24, ensuring that each project contributes proportionally to the results.

Among all strategies, ZS-Exc (zero-shot with exception context) achieves the highest F1 score (17.15%), combining the best recall (69.86%) with the highest precision (9.88%). This confirms its dominance across individual projects and demonstrates that information about raised exceptions and failed tests is among the most effective contextual knowledge for LLM-based bug detection.

Other context-enriching approaches perform well: call graph approach and ZS-Class (where the entire class is included in the input prompt) achieve F1 scores of 13.19% and 12.27%, respectively. Both approaches combine reasonable recall (50.21% for the CG strategy, 50.48% for the ZS-Class one) with decent values for precision (7.67% and 7.01%, respectively). These results highlight the importance of structural and semantic context in enhancing model performance, with information about method calls appearing more useful than including the entire class in the input prompt.

The two baseline approaches (zero-shot and few-shot) perform similarly, with F1 scores of 10.20% and 10.91%, respectively. This highlights that the addition of examples in the input prompt does not significantly improve performance in such complex code bases. The motivational variants yield mixed results: both achieve high recall (especially FS-Mot, with 64.41%) but suffer from low precision (5.25% for the zero-shot variant, 5.60% for the few-shot one), resulting in modest F1 scores of 9.53% and 10.26%, respectively. This suggests that, while brief explanations help with recall, they do not consistently improve overall detection accuracy.

Chain-of-Thought strategies remain the least effective. The zero-shot version

has the lowest recall (32.32%) and a weak F1 score of 9.34%, while the CoT-FS strategy records the lowest overall F1 score (9.25%) among all approaches. They both achieve similar F1 scores to their lightweight versions (ZS-Mot and FS-Mot), with values between 9% and 10%, although showing lower values for recall. This highlights that unguided reasoning alone does not aid complex tasks as bug detection.

Overall, the results reveal a clear pattern: strategies that incorporate contextual information in the prompt (e.g., exceptions, class structure, or call relationships) consistently outperform standard or purely reasoning-based prompts, particularly in large and complex projects.

Approach	Precision	Recall	F1 Score
ZS	5.79% ± 1.27	45.97% ± 4.41	10.20% ± 1.98
ZS-Mot	5.25% ± 1.07	54.21% ± 8.72	9.53% ± 1.79
ZS-Class*	7.01% ± 1.07	50.48% ± 4.48	12.27% ± 1.67
ZS-Exc	**9.88% ± 2.01**	**69.86% ± 10.72**	**17.15% ± 3.01**
CG*	7.67% ± 1.58	50.21% ± 7.20	13.19% ± 2.48
FS	6.23% ± 1.43	47.46% ± 7.31	10.91% ± 2.20
FS-Mot	5.60% ± 1.44	64.41% ± 3.15	10.26% ± 2.45
CoT-ZS	5.51% ± 0.86	32.32% ± 6.22	9.34% ± 1.30
CoT-FS	5.24% ± 1.26	41.04% ± 6.24	9.25% ± 2.05

Table 4.31: Weighted Averages; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

To provide a visual comparison, Figure 4.4 shows the F1 scores achieved by each prompting strategy on Partition 2 projects. ZS-Exc visibly stands out as the top-performing strategy, with an F1 score of 17.15%. ZS-Class and CG are also distinguishable for their relatively high positions. On the other hand, the shortest bars correspond to CoT-based methods, reflecting their weaker performance. The chart also clearly shows how baseline approaches (ZS and FS) slightly outperform their motivation-enhanced variants (ZS-Mot and FS-Mot), which are themselves ahead of the CoT-based prompts. Finally, the plot highlights a slight improvement when moving from the ZS configuration to the FS one, suggesting the benefit offered by the examples in the few-shot setting.

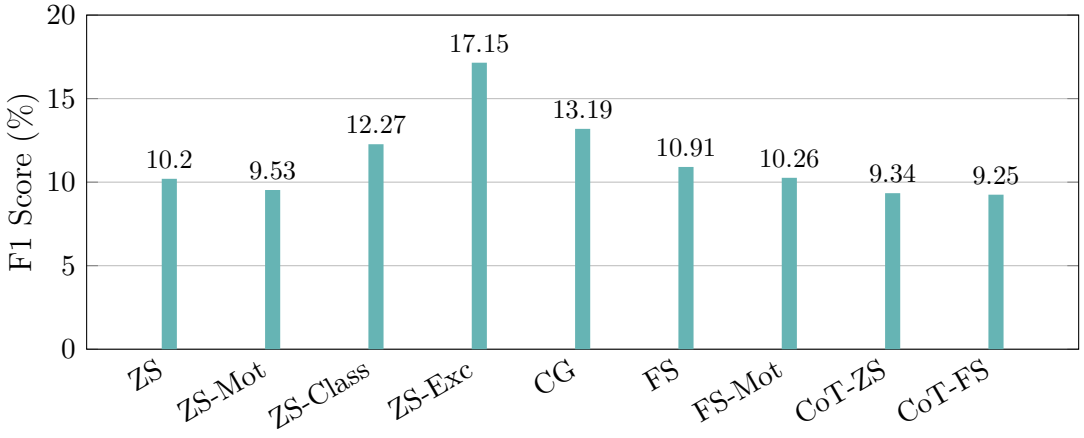

Figure 4.4: F1 Scores of different approaches on Partition 2 projects

Table 4.32 presents the average processing time for each approach on the Partition 2 projects. These results reveal significant differences among the evaluated prompting strategies. As expected, Chain-of-Thought approaches are the most time-consuming, requiring on average over 17 seconds per method, due to their step-by-step reasoning generation. The ZS-Mot and FS-Mot strategies, which include a short motivational statement, can be considered lightweight versions of CoT. While faster than their CoT counterparts, they still require over 8 seconds per method on average. On the other hand, the baseline ZS approach is more efficient, requiring around 3 seconds per method. Naturally, approaches with longer prompts, such as ZS-Class, CG, and FS, require more time, averaging nearly 5 seconds per method. Interestingly, ZS-Exc not only delivers the best performance but is also the fastest approach, with an average processing time of under 3 seconds per method. This suggests that providing exception-related context helps the model narrow its search space when assessing the presence of a bug.

Approach	Csv	JxPath	JCore	Compress	Lang	JData	Average
ZS	2.79	2.67	**3.05**	3.66	3.10	2.95	3.04
ZS-Mot	8.80	7.87	9.00	9.01	8.77	8.31	8.63
ZS-Class	5.14	3.63	5.81	4.12	5.64	4.77	4.85
ZS-Exc	**2.64**	**2.60**	3.11	**2.89**	**2.98**	**2.82**	**2.84**
CG	4.25	3.19	5.66	4.60	4.22	3.87	4.30
FS	3.71	3.58	3.70	3.99	4.04	3.76	3.80
FS-Mot	7.45	8.20	10.1	9.26	9.20	8.59	8.80
CoT-ZS	18.7	16.0	17.8	17.4	17.1	16.7	17.3
CoT-FS	18.8	18.2	21.1	19.8	19.1	18.4	19.2

Table 4.32: Average processing time per method on Partition 2 projects with different approaches; Results expressed in s/m (seconds/method); (**JCore**: JacksonCore; **JData**: JacksonDatabind); Best results in **bold**

4.4 Partition 3

To ensure clarity in presenting the results, each approach is again assigned to a code, as shown in Table 4.21. Table 4.33 shows the number of buggy and total methods for each project in Partition 3, along with their corresponding ratios.

Project	Number of buggy methods	Number of total methods	Ratio (%)
JacksonXml	5	152	3.29%
Chart	39	1503	2.59%
Time	51	1753	2.91%
Cli	52	769	6.76%
Jsoup	159	4097	3.88%
Math	149	3046	4.89%

Table 4.33: Partition 3 project weights

As previously discussed in Section 4.3, two approaches, due to their structural constraints, require different weighting than those listed above. Table 4.34 reports the number of buggy and total methods for Partition 3 projects when using the zero-shot-whole-class approach. This time four projects are affected: *Chart*, *Time*, *Jsoup*, and *Math*.

Project	Number of buggy methods	Number of total methods	Ratio (%)
JacksonXml	5	152	3.29%
Chart	30	628	4.78%
Time	50	1575	3.17%
Cli	52	769	6.76%
Jsoup	157	4091	3.84%
Math	143	2741	5.22%

Table 4.34: Partition 3 project weights for the zero-shot-whole-class approach

Table 4.35 reports the number of buggy and total methods for each project when using the call graph strategy.

Project	Number of buggy methods	Number of total methods	Ratio (%)
JacksonXml	5	92	5.43%
Chart	35	1108	3.16%
Time	37	1061	3.49%
Cli	40	432	9.26%
Jsoup	109	2515	4.33%
Math	122	2182	5.59%

Table 4.35: Partition 3 project weights for the call graph approach

4.4.1 Results

Table 4.36 presents the performance of the LM70 model on the *JacksonXml* project. Due to the small size of this project (only 5 buggy methods among 152 total), the confidence intervals are wide, especially for recall, making it a particularly unstable evaluation scenario. The best overall performance is achieved by the call graph approach, with an F1 score of 16.34%, obtained by combining the highest precision (9.39%) with a strong recall (64.00%). The second-best approach is ZS-Exc, that achieves the highest recall (96.00%) and a good precision (7.59%), resulting in an F1 score of 14.07%. The base few-shot approach also performs well, reaching an F1 score of 13.22%, suggesting that well-chosen examples can effectively guide the model in small-scale project settings. ZS-Class shows reasonable performance, with an F1 score of 11.25%. The zero-shot strategy and its motivation-enhanced variant perform similarly, with F1 scores of 10.13% and 10.14%, respectively. In contrast, FS-Mot underperforms compared to its baseline counterpart, penalized

by low precision (5.02%) despite achieving a higher recall (64.00% against 52.00%). Both the zero-shot and few-shot CoT approaches once again deliver the weakest performance, combining the lowest recalls (both with 36.00%) with the poorest precision values (4.34% and 4.60%, respectively).

Approach	Precision	Recall	F1 Score
ZS	5.54% ± 2.45	60.00% ± 30.41	10.13% ± 4.53
ZS-Mot	5.48% ± 1.49	68.00% ± 13.60	10.14% ± 2.70
ZS-Class*	6.17% ± 3.04	64.00% ± 32.38	11.25% ± 5.56
ZS-Exc	7.59% ± 0.64	**96.00% ± 11.11**	14.07% ± 1.19
CG*	**9.39% ± 2.12**	64.00% ± 20.78	**16.34% ± 3.88**
FS	7.57% ± 1.73	52.00% ± 13.60	13.22% ± 3.07
FS-Mot	5.02% ± 1.65	64.00% ± 20.78	9.32% ± 3.05
CoT-ZS	4.34% ± 2.50	36.00% ± 20.78	7.74% ± 4.46
CoT-FS	4.60% ± 2.10	36.00% ± 20.78	8.15% ± 3.83

Table 4.36: Project: **JacksonXml**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.37 shows the results obtained by the LM70 model on the *Chart* project. The most effective prompting strategy is ZS-Class, which achieves an F1 score of 16.28%, thanks to the highest precision (9.37%) and a strong recall (62.00%). ZS-Exc also performs well, combining a relatively good precision (7.45%) with the best recall (75.90%), for a resulting F1 score of 13.57%. Both the zero-shot and few-shot baselines perform moderately well, with F1 scores of 10.71% and 11.32%, respectively. The corresponding motivation-enhanced versions show better recalls, but their overall F1 scores are substantially lower due to poor precision values (4.85% for ZS-Mot and 5.04% for FS-Mot). Surprisingly, the call graph approach underperforms here, with an F1 score of just 7.91%, caused by both weak precision and recall (4.30% and 49.14%, respectively). This time CoT variants are not the worst-performing approaches (as they are surpassed by ZS-Mot and FS-Mot), hovering around 10% F1 score. Their recalls remain mediocre, particularly in the few-shot version (41.54%), which represents the lowest value among all approaches.

Approach	Precision	Recall	F1 Score
ZS	5.85% ± 0.88	63.59% ± 9.92	10.71% ± 1.61
ZS-Mot	4.85% ± 0.27	66.67% ± 5.03	9.04% ± 0.51
ZS-Class*	**9.37% ± 0.78**	62.00% ± 6.28	**16.28% ± 1.34**
ZS-Exc	7.45% ± 0.51	**75.90% ± 5.33**	13.57% ± 0.92
CG*	4.30% ± 0.70	49.14% ± 7.69	7.91% ± 1.28
FS	6.38% ± 0.70	50.26% ± 6.21	11.32% ± 1.25
FS-Mot	5.04% ± 0.71	72.31% ± 10.89	9.42% ± 1.33
CoT-ZS	5.85% ± 1.18	49.23% ± 8.83	10.46% ± 2.07
CoT-FS	5.41% ± 0.79	41.54% ± 7.26	9.56% ± 1.41

Table 4.37: Project: **Chart**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.38 presents the results obtained by the LM70 model on the *Time* project. As in other projects, ZS-Exc is the best-performing approach, achieving an F1 score of 16.40% by combining the highest precision (9.33%) and recall (67.84%). The other approaches appear to struggle on this project. CG shows relatively good precision (6.53%) along with a modest recall (48.11%), resulting in an F1 score of 11.50%. ZS-Class ranks third with 9.54% F1 score, combining modest values for precision and recall (5.30% and 47.20%, respectively). The zero-shot and few-shot baseline strategies yield mediocre results, with F1 scores around 8%. Their motivation-based counterparts perform even worse, with F1 scores of 7.45% for ZS-Mot and 7.62% for FS-Mot. While they consistently achieve higher recall, the accompanying drop in precision heavily impact their resulting F1 scores. Both CoT strategies rank among the worst-performing, especially the few-shot variant, which exhibits the lowest precision (3.49%) and recall (25.49%), resulting in a poor F1 score of 6.14%.

Approach	Precision	Recall	F1 Score
ZS	4.42% ± 0.45	43.53% ± 4.68	8.03% ± 0.82
ZS-Mot	4.02% ± 0.67	50.98% ± 8.26	7.45% ± 1.24
ZS-Class*	5.30% ± 0.54	47.20% ± 4.16	9.54% ± 0.96
ZS-Exc	**9.33% ± 0.49**	**67.84% ± 2.78**	**16.40% ± 0.83**
CG*	6.53% ± 2.07	48.11% ± 17.01	11.50% ± 3.68
FS	4.66% ± 0.80	34.12% ± 5.86	8.20% ± 1.39
FS-Mot	4.10% ± 1.20	53.73% ± 15.07	7.62% ± 2.23
CoT-ZS	4.18% ± 0.80	32.94% ± 6.06	7.41% ± 1.41
CoT-FS	3.49% ± 1.09	25.49% ± 7.70	6.14% ± 1.90

Table 4.38: Project: **Time**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.39 presents the performance of the LM70 model on the *Cli* project. This is one of the most favorable cases, as all the approaches show relatively high values compared to other projects, especially for precision. ZS-Exc is the best-performing strategy once again, thanks to the highest values for precision (17.62%), recall (76.54%), and, consequentially, F1 score (28.64%). CG and ZS-Class also demonstrate strong performance, with F1 scores of 23.41% and 21.90%, respectively. ZS and FS baseline approaches achieve good results, especially the few-shot version, with an F1 score of 18.60%. ZS-Mot results in a slightly higher F1 score than ZS (15.71% vs. 15.42%), primarily due to a stronger recall (72.31% against 59.23%). The FS-Mot strategy also significantly improves recall compared to FS (67.31% against 45.38%), but is penalized by relatively low precision, resulting in an F1 score of 16.99%. The CoT variants also show competitive results, with F1 scores around 17%, although the few-shot version still struggles with recall, as it only reaches 42.69%.

Approach	Precision	Recall	F1 Score
ZS	8.86% ± 1.20	59.23% ± 7.44	15.42% ± 2.06
ZS-Mot	8.82% ± 0.66	72.31% ± 4.34	15.71% ± 1.15
ZS-Class*	13.27% ± 1.34	62.69% ± 5.99	21.90% ± 2.12
ZS-Exc	**17.62% ± 0.80**	**76.54% ± 3.54**	**28.64% ± 1.21**
CG*	14.28% ± 1.87	65.00% ± 7.91	23.41% ± 2.97
FS	11.71% ± 2.83	45.38% ± 9.93	18.60% ± 4.38
FS-Mot	9.73% ± 0.68	67.31% ± 5.85	16.99% ± 1.20
CoT-ZS	9.44% ± 0.55	53.08% ± 4.34	16.03% ± 0.99
CoT-FS	10.89% ± 1.37	42.69% ± 5.95	17.35% ± 2.22

Table 4.39: Project: **Cli**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.40 displays the results for the LM70 model on the *Jsoup* project. Here, the ZS-Exc configuration clearly outperforms the other approaches, achieving the highest values for precision (11.41%), recall (72.08%), and F1 score (19.70%). CG also performs competitively, reaching an F1 score of 13.34%, while the ZS-Class variant only obtains an F1 score of 10.67%, mainly due to a modest recall (47.64%). The FS baseline achieves a better performance than its zero-shot counterpart (11.28% against 9.68%) thanks to its higher precision (6.46% for FS, 5.34% for ZS). Their motivation-enhanced versions follow the trend observed so far: they achieve higher recall at the cost of lower precision, leading to reduced F1 scores. Both CoT strategies achieve decent F1 scores (9.74% for the zero-shot version, 10.10% for the few-shot one), although they struggle with recall, exhibiting the lowest values (43.40% and 40.25%, respectively).

Approach	Precision	Recall	F1 Score
ZS	5.34% ± 0.42	51.70% ± 4.11	9.68% ± 0.75
ZS-Mot	5.19% ± 0.41	64.15% ± 5.18	9.61% ± 0.75
ZS-Class*	6.01% ± 0.36	47.64% ± 2.59	10.67% ± 0.63
ZS-Exc	**11.41% ± 0.67**	**72.08% ± 4.51**	**19.70% ± 1.16**
CG*	7.53% ± 0.65	58.17% ± 6.37	13.34% ± 1.18
FS	6.46% ± 0.54	44.53% ± 3.96	11.28% ± 0.94
FS-Mot	5.17% ± 0.40	62.26% ± 5.30	9.55% ± 0.74
CoT-ZS	5.49% ± 0.38	43.40% ± 3.54	9.74% ± 0.69
CoT-FS	5.77% ± 0.50	40.25% ± 3.36	10.10% ± 0.87

Table 4.40: Project: **Jsoup**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Table 4.41 reports the performance of the LM70 model on the *Math* project. The trend observed so far is confirmed. ZS-Exc is once again the best-performing approach, achieving the highest values for all the three metrics, with 12.03% for precision, 77.05% for recall, and 20.81% for F1 score. CG performs well, combining strong recall (60.98%) with relatively high precision (10.18%), resulting in an F1 score of 17.45%. ZS-Class follows closely, with an F1 score of 15.93%. Zero-shot and few-shot baseline configurations show similar performance, with F1 scores around 14%, while their motivation-based variants perform slightly worse, having F1 scores around 13%. Despite higher recall, these variants tend to sacrifice precision, suggesting that the motivation component may cause the model to over-predict. Chain-of-Thought approaches yield decent results, with F1 scores around 13%, despite showing the worst recall values (41.88% for the zero-shot version, 38.93% for the few-shot one).

Approach	Precision	Recall	F1 Score
ZS	8.26% ± 0.61	58.39% ± 5.10	14.47% ± 1.10
ZS-Mot	7.25% ± 0.45	62.01% ± 3.61	12.99% ± 0.80
ZS-Class*	9.17% ± 0.47	60.42% ± 3.76	15.93% ± 0.83
ZS-Exc	**12.03% ± 0.63**	**77.05% ± 3.20**	**20.81% ± 1.05**
CG*	10.18% ± 1.15	60.98% ± 7.57	17.45% ± 1.99
FS	8.14% ± 0.24	44.83% ± 2.46	13.78% ± 0.45
FS-Mot	7.31% ± 0.37	65.23% ± 3.56	13.15% ± 0.67
CoT-ZS	7.63% ± 0.88	41.88% ± 4.19	12.91% ± 1.46
CoT-FS	8.17% ± 0.59	38.93% ± 1.77	13.51% ± 0.88

Table 4.41: Project: **Math**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Finally, Table 4.42 summarizes the overall performance of the different prompting strategies on Partition 3 projects. The weights are represented by the total number of methods of each project reported in Tables 4.33, 4.34, and 4.35.

As already observed in Partition 2 (see Table 4.31), the best-performing approach is ZS-Exc, which consistently outperforms the other strategies and reaches an average F1 score of 19.21%, combining the best precision (11.10%) with the highest recall (73.89%). This result confirms that exception-related information is crucial for enhancing LLMs performance in bug detection tasks.

Call graph and ZS-Class approaches again show competitive performance, with F1 scores of 12.00% and 13.17%, respectively. These results align with what has been observed in Partition 2: structural or class-level context provides valuable information that helps the model in its evaluations.

The baseline ZS and FS strategies achieve moderate results, with F1 scores of

11.25% and 12.00%, respectively. Here, as it happened in Partition 2, the few-shot version offers a slightly better performance, highlighting the benefit of adding meaningful examples in the input prompt, though the impact remains minimal. The motivation-enhanced versions confirm the patterns observed in Partition 2: they both heavily improve recall, especially in the few-shot variant, at the expense of lower precision, which ultimately result in lower F1 scores. This indicates that the lightweight CoT variant is not as effective as expected.

The struggle of reasoning-based approaches is confirmed by the fact that, just like in Partition 2, both Chain-of-Thought strategies present mediocre results. The zero-shot version reaches an F1 score of 10.73%, with a poor recall of 42.70%, while the few-shot version, although it has the lowest recall across all approaches (37.89%), performs slightly better, with an F1 score of 10.80%. They both achieve comparable results to their lightweight versions (ZS-Mot and FS-Mot), with F1 scores around 10%, despite showing consistently lower recall values.

In conclusion, Partition 3 results confirm the key findings already observed in Partition 2: prompts enriched with contextual information, such as exception traces or program structure, lead to better model performance than those relying only on reasoning strategies, like chain-of-thought prompting. This suggests that, in real-world scenarios, models alone are not sufficiently effective, even when explicitly prompted to reason carefully about the code. However, when provided with relevant contextual information about the target method, their performance improves significantly.

Approach	Precision	Recall	F1 Score
ZS	6.29% ± 1.62	54.44% ± 6.51	11.25% ± 2.68
ZS-Mot	5.77% ± 1.46	62.48% ± 5.83	10.53% ± 2.47
ZS-Class*	7.54% ± 2.41	53.41% ± 7.14	13.17% ± 3.83
ZS-Exc	**11.10% ± 2.48**	**73.89% ± 4.34**	**19.21% ± 3.73**
CG*	8.10% ± 2.61	56.67% ± 5.75	14.10% ± 4.11
FS	6.99% ± 1.78	43.92% ± 4.88	12.00% ± 2.64
FS-Mot	5.87% ± 1.60	63.44% ± 5.50	10.71% ± 2.68
CoT-ZS	6.16% ± 1.54	42.70% ± 5.62	10.73% ± 2.43
CoT-FS	6.35% ± 2.06	37.89% ± 5.70	10.80% ± 3.10

Table 4.42: Weighted Averages; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

To complement the numerical results, Figure 4.5 visually compares the F1 scores obtained by each prompting strategy on Partition 3 projects. As in the previous partition, ZS-Exc clearly dominates, showing a 5.11% margin on the second-best approach. Both ZS-Class and CG also stand out for their relatively high bars,

reinforcing the benefit of structural context in prompting. The other approaches range between 10% and 12%, with the chart highlighting slight differences that might be less evident in tabular format.

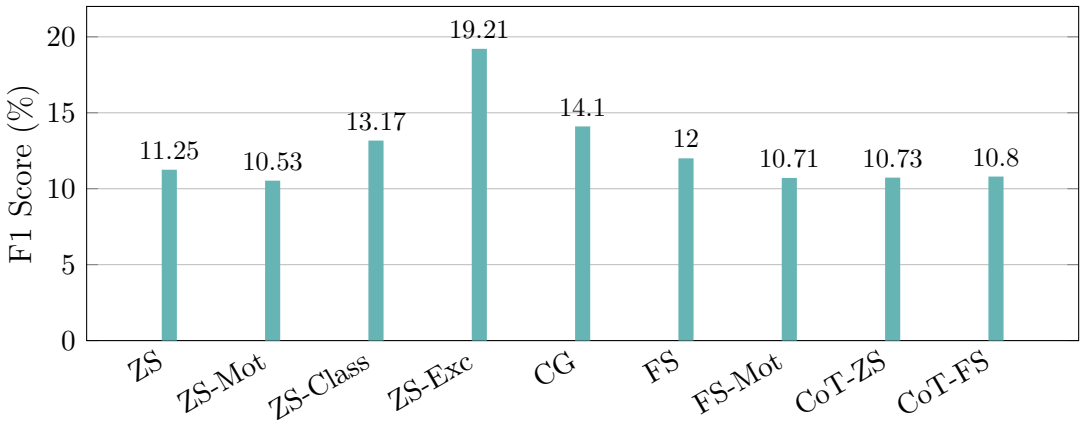

Figure 4.5: F1 Scores of different approaches on Partition 3 projects

Table 4.43 shows the average processing time per method for each approach across the Partition 3 projects. The results are consistent with those observed in Partition 2 (see Table 4.32), with the CoT strategies remaining the most time-consuming (especially the few-shot variant, reaching nearly 20 seconds per method). The motivation-enhanced approaches also require considerable time, with around 8 seconds on average, but they are still faster than full CoT prompting. The ZS baseline requires 3.15 seconds per method on average, while approaches with longer prompts, like ZS-Class, CG, and FS, intuitively require more time (taking around 4 seconds). Once again, the fastest approach is ZS-Exc, averaging 2.83 seconds per method. This reinforces the idea that exception-related context can simplify the model's task by better aligning with its reasoning process.

Approach	JXml	Chart	Time	Cli	Jsoup	Math	Average
ZS	3.24	3.30	3.09	3.29	2.80	3.20	3.15
ZS-Mot	6.53	7.83	7.39	8.01	7.26	7.87	7.48
ZS-Class	3.55	5.43	4.71	3.42	3.58	4.93	4.27
ZS-Exc	**2.71**	**3.03**	**2.95**	**2.70**	**2.63**	**2.97**	**2.83**
CG	3.77	4.70	3.97	3.94	3.38	4.93	4.11
FS	3.31	3.68	3.47	3.47	3.08	3.78	3.46
FS-Mot	8.51	8.60	8.90	8.57	8.68	9.54	8.80
CoT-ZS	15.2	16.0	16.8	15.7	16.9	17.6	16.4
CoT-FS	18.4	19.5	20.6	18.4	20.3	21.1	19.7

Table 4.43: Average processing time per method on Partition 3 projects with different approaches; Results expressed in s/m (seconds/method); (**JXml**: JacksonXml); Best results in **bold**

Ablation study: analyzing different few-shot variants

As described in Section 4.1.1, the main few-shot prompting strategy adopted in this work includes six examples, each accompanied by a textual explanation that explicitly states why the corresponding code is labeled as buggy or non-buggy. This ablation study investigates two key aspects of that setup: the impact of the explanation field and the effect of the number of examples included in the prompt.

For the first case, a variant called **P-FS** (pure few-shot) is defined, which omits the explanation. In this setup, the examples only consist of method code and its corresponding evaluation, reflecting a more traditional few-shot format. To ensure a fair comparison, the examples are kept the same as the ones used in the original few-shot strategy, reported in Tables 4.2 and 4.3.

The second variant is defined as **FS-LE** (few-shot with less examples), which reduces the number of in-context examples from six to three. These examples retain the explanation field and are selected among the ones used in the original few-shot strategy to ensure comparability. The chosen examples are listed below:

- **Examples for partitions 1 and 2**:
  - **Project:** Chart; **Bug Number:** 11; **Version:** Buggy
  - **Project:** Chart; **Bug Number:** 15; **Version:** Buggy
  - **Project:** Math; **Bug Number:** 92; **Version:** Fixed
- **Examples for partition 3**:
  - **Project:** Lang; **Bug Number:** 61; **Version:** Buggy

– **Project:** Mockito; **Bug Number:** 4; **Version:** Buggy
– **Project:** Lang; **Bug Number:** 49; **Version:** Fixed

For more details on their characteristics, refer to Tables 4.2 and 4.3. All these experiments were conducted with the LM70 model.

Table 4.44 displays the results obtained on the Partition 2 projects. Overall, the performance difference among the three variants is relatively small, with only minor gains or losses observed in most cases. The pure few-shot variant achieves the highest F1 scores on the smaller projects, which are *Csv* and *JxPath*, both with 13.37%. On the other hand, in projects like *JacksonCore*, *Compress*, and *Lang*, the original few-shot configuration remains slightly superior to both alternatives. Regarding the FS-LE approach, reducing the number of examples from six to three does not lead to a drop in performance, with an F1 score similar to FS in projects like *JacksonCore* and *Lang*. Overall, the weighted average across all projects reveals only a minimal gap between the variants, with the original FS configuration reaching the highest average F1 score of 10.91%.

The complete results for each metric are reported in Section A.2.1 of Appendix A.

Project	FS	P-FS	FS-LE
Csv	9.50% ± 3.18	**13.37% ± 2.17**	11.23% ± 5.82
JxPath	12.84% ± 1.60	**13.37% ± 0.88**	12.45% ± 2.08
JacksonCore	**7.27% ± 0.47**	6.62% ± 0.41	7.15% ± 0.44
Compress	**13.50% ± 1.36**	12.79% ± 1.27	12.26% ± 2.02
Lang	**9.37% ± 1.40**	8.27% ± 0.79	9.08% ± 0.63
JacksonDatabind	12.40% ± 0.71	**12.54% ± 0.19**	12.01% ± 0.72
W. Average	**10.91% ± 2.20**	10.65% ± 2.65	10.56% ± 1.99

Table 4.44: F1 scores of different few-shot variants on Partition 2 projects (**W. Average**: Weighted Average); Format: value ± confidence interval; Best results in **bold**

Table 4.45 reports the results obtained by the three few-shot variants on the Partition 3 projects. The differences in performance across these configurations are again limited, with no single variant outperforming the others. The pure few-shot variant once again performs well on small projects like *JacksonXml*, with a resulting F1 score of 13.35%. In all the other projects, the FS-LE version surprisingly achieves the best results, especially in *Cli* project, with an F1 score of 20.61%. However, in most projects the results remain very close, with minimal variation across the three variants. The weighted averages show that FS-LE performs slightly better than the others, with a final F1 score of 12.49%, against 12.00% for FS and 11.96% for the pure few-shot strategy.

The complete results for each metric are reported in Section A.2.2 of Appendix A.

Project	FS	P-FS	FS-LE
JacksonXml	13.22% ± 3.07	**13.35% ± 4.97**	10.49% ± 4.33
Chart	11.32% ± 1.25	12.29% ± 1.62	**12.54% ± 1.35**
Time	8.20% ± 1.39	**8.47% ± 1.07**	8.44% ± 0.89
Cli	18.60% ± 4.38	16.32% ± 1.71	**20.61% ± 1.58**
Jsoup	11.28% ± 0.94	10.90% ± 0.67	**11.31% ± 0.54**
Math	13.78% ± 0.45	14.06% ± 0.90	**14.45% ± 1.37**
W. Average	12.00% ± 2.64	11.96% ± 2.30	**12.49% ± 3.08**

Table 4.45: F1 scores of different few-shot variants on Partition 3 projects (**W. Average**: Weighted Average); Format: value ± confidence interval; Best results in **bold**

Overall, the results indicate that including an explanation alongside the examples offers a performance benefit with respect to the pure few-shot strategy, as shown by the higher average F1 score across both partitions. Therefore, the initial design choice is supported by these results, although the impact remains limited. As for the number of examples, reducing the count from six to three does not lead to a drop in performance, indicating that using fewer examples can still be effective.

Table 4.46 reports the average processing time per method of the three few-shot variants, measured across the projects in both Partition 2 and 3. The results confirm the expectations: the original few-shot approach (FS), which includes six examples along with their explanation, has the highest average processing time due to its longer prompt length. The pure few-shot variant, which excludes explanations, performs slightly faster, averaging 3.54 seconds per method compared to 3.63 seconds per method, on average. As expected, FS-LE, which uses only three examples, is the fastest variant, requiring 3.06 seconds on average to process a single method.

Project	FS	P-FS	FS-LE
Csv	3.71	3.67	**3.07**
JxPath	3.58	3.20	**1.99**
JacksonCore	3.70	3.43	**3.42**
Compress	3.99	4.83	**3.24**
Lang	4.04	3.51	**3.37**
JacksonDatabind	3.76	3.41	**3.19**
JacksonXml	3.31	3.29	**2.86**
Chart	3.68	3.57	**3.22**
Time	3.47	3.32	**3.05**
Cli	3.47	3.31	**3.01**
Jsoup	3.08	3.23	**2.96**
Math	3.78	3.69	**3.34**
Average	3.63	3.54	**3.06**

Table 4.46: Average processing time per method across few-shot variants; Results expressed in s/m (seconds/method); Best results in **bold**

Ablation study: position of the reasoning field in Chain-of-Thought approaches

The prompt presented in Listing 3.12 describes the zero-shot variant adopted for the Chain-of-Thought strategy. This ablation study investigates the impact of the position of the `reasoning` field in the model's JSON response. Specifically, the goal is to evaluate the model's performance when it has to produce tokens before its final evaluation (the `has_bug` field), compared to placing it after. Apart from this, the remainder of the prompt remains unchanged. This experiment resembles the earlier ablation study discussed in Section 4.2.2.

Table 4.47 reports the F1 scores with the CoT-ZS variant. The analysis was conducted on a smaller portion of the dataset, covering six projects from both Partition 2 and 3. The results highlight that positioning the `reasoning` field after the `has_bug` field generally leads to better performance, yielding higher F1 score overall. Moreover, this ordering results in smaller confidence intervals, suggesting more stable and reliable predictions across runs. These findings show a measurable effect of field ordering on the model's performance, although the impact remains minimal.

The complete results for each metric are reported in Section A.3 of Appendix A.

Project	Before	After
Csv	**13.23% ± 4.53**	12.59% ± 2.07
JxPath	8.50% ± 2.74	**11.37% ± 1.69**
JacksonCore	**7.33% ± 2.17**	6.95% ± 1.35
Compress	10.10% ± 1.60	**10.54% ± 0.77**
JacksonXml	6.16% ± 8.30	**7.74% ± 4.46**
Cli	15.75% ± 3.68	**16.03% ± 0.99**
Weighted Average	9.82% ± 3.11	**10.24% ± 3.24**

Table 4.47: F1 scores for different position of the reasoning field (**Before**: `reasoning` field **before** the `has_bug` field; **After**: `reasoning` field **after** the `has_bug` field); Format: value ± confidence interval; Best results in **bold**

Table 4.48 reports the average processing time per method of the two ZS-CoT variants. Interestingly, placing the `reasoning` field after the `has_bug` field consistently reduces average processing time, with improvements ranging from 3 to 5 seconds per method. This ordering offers both improved performance and reduced processing time, making it the more efficient of the two. Therefore, it was adopted for the reported experiments.

Csv	JxPath	JacksonCore	Compress	JacksonXml	Cli	Average
18.7	20.0	22.1	21.5	18.2	21.5	20.3
15.6	**16.0**	**17.8**	**17.4**	**15.2**	**15.7**	**16.3**

Table 4.48: Average processing time per method based on position of `reasoning` field; Results expressed in s/m (seconds/method); (Top: `reasoning` field **before** the `has_bug` field; Bottom: `reasoning` field **after** the `has_bug` field); Best results in **bold**

4.5 Threats To Validity

This section discusses the potential threats to validity that could affect the results presented in this thesis.

Internal Threats

The experimental pipeline relies on a combination of Python scripts and Java programs to extract methods from the dataset, interact with large language models, and evaluate their outputs. Although these components were thoroughly tested

during development, undetected bugs may still be present. To support transparency, these scripts can be made available upon request.

Another source of potential threats lies in the manual selection of buggy methods from the dataset and their manual annotation to build the ground truth files. This process, while carefully repeated to ensure accuracy, can still be prone to human error or subjective judgment.

Furthermore, the explanations and reasoning included in the few-shot examples were initially generated using ChatGPT and subsequently refined manually. Alternative formulations may be more effective and lead to better model performance.

External Threats

The experiments were conducted exclusively on Defects4J dataset, which contains only Java programs. While this dataset is widely adopted and represents a solid benchmark for evaluation, the results may not generalize to other programming languages or codebases.

Given Defects4J's reputation, it is possible that some of its bugs were included in the training data of the evaluated LLMs. While this bias cannot be ruled out, it would be unfeasible to re-train these models to eliminate it. However, the results highlight that context-enriched approaches, which provide additional code-level information, outperform baseline strategies, suggesting that the models are not simply memorizing exact instances from the dataset.

Construct Threats

In this work, the ability of LLMs to detect bugs was evaluated as a binary classification task. Therefore, model performance has been measured using precision, recall and F1 score, in line with standard practices in related work.

Conclusion threats

In order to mitigate the intrinsic randomness of LLMs, each prompting strategy was evaluated over five independent runs, and results were aggregated using 95% confidence intervals. This approach aligns with best practices in recent literature.

Additional Considerations

Beyond the above-mentioned threats, several other factors deserve discussion.

Prompt engineering was performed manually, guided by intuition and insights from existing literature. However, it may not represent the optimal way to extract reasoning from LLMs. Other more sophisticated strategies, such as automatic prompt generation [50], could potentially lead to better results.

LM70 was the model that was chosen for the subsequent experiments on projects of Partition 2 and 3. It achieved the best results, but its high processing time and large resource requirements may limit its feasibility in many real-world scenarios. Practical applications may prefer smaller and more efficient models, accepting a trade-off between performance and resource constraints.

Moreover, this study did not cover every available model or every size configuration, focusing on a limited set of LLMs whose parameters sizes ranges from 8B to 70B. Other models which were not evaluated in this work, especially newer ones, could potentially obtain better results.

Chapter 5

Related Work

In recent years, Artificial Intelligence (AI), and particularly Large Language Models (LLMs), have been extensively applied to various software engineering tasks. This chapter presents a review of relevant literature across this area, with a focus on topics that are related to the objectives of this thesis, such as automated program repair (APR), vulnerability detection, and code analysis.

5.1 Automated Program Repair

Automated Program Repair (APR) refers to the process of automatically fixing software bugs without requiring human intervention, and is widely recognized as a key step toward software automation. The goal is to produce a patched version of the code based on the original program and the identified buggy location.

Hossain et al. [51] proposed Toggle, a program repair framework that exploits LLMs both to pinpoint the location of the bug at the token level and to fix the corresponding buggy code. They explore various prompting strategies for the bug-fixing model, to determine which yields the best performance. Their framework achieves state-of-the-art performance on the CodeXGLUE [52] benchmark, and obtains strong results on other datasets too (such as Defects4J).

Xia et al. [53] evaluate nine state-of-the-art LLMs on their ability to generate patches across three different repair settings, in order to identify the most effective configuration. Their study is conducted on five datasets and shows that directly applying LLMs to APR tasks outperforms traditional techniques.

Fan et al. [54] explore the application of existing APR techniques to fix code produced by LLMs for complex programming tasks. Their study revealed that the code generated by LLMs shares common mistakes with the human-written one, highlighting the potential of APR techniques to fix the automatically generated code.

Prenner et al. [55] evaluated Codex [43] model on the QuixBugs [35] dataset, finding that it performs competitively on Python programs, but encounters greater challenges with Java.

Kolak et al. [56] also evaluated Codex (alongside two smaller models) on the same dataset, assessing its ability to generate correct patches based on a given code prefix. The results confirm the scaling effect of LLMs, where performance improves with model size.

5.2 Vulnerability Detection

Vulnerabilities represent a particular category of bugs that, if exploited, can compromise the security of a system. In recent years, several studies have investigated the use of LLMs for the detection and remediation of such security flaws.

Pearce et al. [57] conducted an analysis into the use of LLMs for vulnerability repair. Their work focused on the challenges involved in designing effective zero-shot prompts to guide models in generating secure versions of insecure code.

Lin et al. [58] evaluated different models on Vul4J [37] and Big-Vul [59] datasets for the task of vulnerability detection. They experimented with both zero-shot and few-shot prompting strategies and analyzed the impact of various factors, such as model size, quantization methods, and context window length. Their main findings are the following: LLMs tend to perform better on Java code rather than on C/C++; few-shot prompting often leads to worse results than zero-shot across most models; larger context windows generally yield better performance.

Steenhoek et al. [60] examined the capabilities of eleven LLMs to detect vulnerabilities in isolated functions, rather than entire source files. Their study explored various prompting strategies, including in-context learning and chain-of-thought reasoning. They observed that LLMs struggled with this task, failing to distinguish between buggy and fixed versions in 76% of cases.

In another work, Steenhoek et al. [61] further explored the limitations of LLMs in vulnerability detection. They observed that LLMs often struggle to reason about code semantics, especially on variable relationships or multi-step logic. Their results suggest that simply scaling up models size or design better prompts is not sufficient, indicating that new modeling and training approaches may be necessary.

Khare et al. [62] evaluated several pre-trained LLMs across five security datasets, each covering distinct vulnerability classes. Their experiments, conducted on short code snippets rather than full source files, highlight that step-by-step reasoning can significantly improve model performance.

Gao et al. [42] introduced VulBench, a benchmark suite that combines six different datasets, to standardize evaluation across different programming languages. Their experiments confirm that LLMs can outperform traditional techniques in

certain cases; however, they also note that models still struggle with more complex, real-world scenarios, particularly when analyzing large-scale software systems.

This trend is further confirmed by the study of Ullah et al. [63]. They developed a framework for the evaluation of LLMs on vulnerability identification and code reasoning across multiple scenarios, including real-world CVEs (Common Vulnerabilities and Exposures). They tested eight different LLMs using various prompting techniques and they identified some limitations, such as non-deterministic outputs and consistently poor performance on real-world scenarios. Their findings confirm that current LLMs remain unreliable for the automated identification of vulnerabilities.

Purba et al. [64] analyzed four LLMs to assess their ability to detect software vulnerabilities using two datasets: a code gadgets dataset [65], where code gadgets consist of semantically related lines of code, and the vulnerabilities are related to buffer overflows and memory management issues, and the CVEfixes [66] dataset, which focuses on SQL injection and buffer overflow problems. Although LLMs have high false positive rates compared to traditional static analysis tools, the study highlighted their promising capabilities in recognizing patterns that are typically associated with security vulnerabilities.

5.3 Code Analysis

Code analysis is a fundamental activity in software engineering that involves the examination of source code, bytecode, or binary code to ensure software quality and reliability. It plays a critical role in helping developers identify and address issues early in the software development life cycle.

Fang et al. [67] conducted an analysis on five popular LLMs, including models from GPT and LlaMa families, to assess their ability to understand both obfuscated and non-obfuscated input source code. Their findings show that, for non-obfuscated code, larger models are more likely to produce detailed explanations of code snippets. On the other hand, smaller models, even when fine-tuned on code-specific data, often fail to deliver correct outputs. When analyzing obfuscated code, both large and small models struggle to generate accurate results. Overall, their findings show that larger LLMs are a safer choice for code analysis tasks.

Chapman et al. [28] proposed an approach that integrates static analysis with LLMs to enhance code analysis. This approach interleaves calls to a traditional static analyzer with queries to LLMs. Specifically, the intermediate results from the static analysis are inserted into the LLMs queries, and the models responses are used to further refine subsequent analysis steps. This hybrid approach outperforms static analysis alone, highlighting the potential benefits of combining LLMs with other existing tools.

Li et al. [68] introduced LLift, a framework that combines several prompting strategies with static analysis results to identify Use Before Initialization (UBI) bugs in the Linux kernel. This framework combines path-sensitive analysis, guided by insights on variable usage specific to Linux, with prompts to LLMs which are carefully designed to complement the analysis. The result is a system that achieves high precision in bug detection, and it effectively scales across millions of lines of code. LLift discovered four previously undocumented UBI vulnerabilities in the Linux kernel, which were subsequently acknowledged by the Linux community. Their work demonstrates how combining semantically aware static analysis with LLM reasoning can offer improvements in real-world bug discovery.

Ma et al. [69] conducted an analysis on four state-of-the-art LLMs (specifically, GPT4, GPT3.5, StarCoder and CodeLlama-13b-instruct) to assess their ability to understand code syntax and semantic structures, such as abstract syntax trees (ASTs), call graphs (CGs) and control flow graphs (CFGs). Their evaluation focused on three dimensions: syntax understanding, static behavior understanding and dynamic behavior understanding. Their findings reveal that, while LLMs are good in correctly recognizing code syntax, they struggle in reasoning about code semantics, especially dynamic semantics. This may help explain why code generated by LLMs is often syntactically correct yet vulnerable to security issues, highlighting the need to check the correctness of LLMs outputs.

Chapter 6

Conclusion and Future Work

This thesis explored the application of Large Language Models (LLMs) to the task of method-level bug detection. The experimental setup involved prompting LLMs with individual methods and asking them whether the target method contained a bug. The study was conducted on Defects4J, a widely adopted open-source dataset containing real-world Java bugs. An initial filtering operation was performed to retain only those bugs that met specific requirements, and the resulting dataset was split into three partitions.

Subsequently, six open-source LLMs were evaluated on the first partition using both zero-shot and few-shot prompting strategies.

`Llama3.1:70b-instruct-q4_0` emerged as the best-performing model in terms of precision, recall, and F1 score. Consequently, it was selected for subsequent experiments on the second and third partitions of the dataset. The model was tested on nine different prompting strategies, ranging from standard zero-shot and few-shot to more elaborate ones, like chain-of-thought and context-enriched prompts.

The results demonstrate that prompts enriched with contextual information (such as exception traces, full-class source code, or structural elements like call graphs) significantly improve model performance, outperforming baseline approaches. Interestingly, the best-performing approach (the one with information about failed test cases and exceptions) also proved to be the fastest, making it appealing from both effectiveness and efficiency standpoints.

Surprisingly, chain-of-thought prompts (which involve step-by-step reasoning) consistently underperformed compared to simpler or context-aware strategies. This suggests that, in complex software scenarios, explicit reasoning alone is not sufficient to achieve acceptable performance.

Several ablation studies were also conducted. A comparison of different few-shot variants highlighted that reducing the number of examples (from six to three) lowers computational overhead without a significant drop in performance. Another

ablation study conducted on the prompt formatting showed that even the position of the output fields (e.g., placing the reasoning before or after the actual bug evaluation) can influence both performance and response generation speed.

Although this thesis evaluates a considerable number of different prompting approaches, there are still open questions for further research.

One area of interest is the impact of the examples in few-shot learning. This study used fixed and manually chosen examples; future work could explore how the choice of examples affects model performance. Similarly, research could also explore the effect of varying model parameters like quantization and temperature.

The call graph approach can be extended by analyzing different graph configurations. For instance, one could incorporate in the prompt caller methods rather than callees, or include indirect calls (multi-hop relationships) to expose deeper program dependencies.

Other combinations of strategies also merit attention. For instance, combining a reasoning-based approach with exception-related information could bridge the gap between abstract reasoning and context-aware insights. Furthermore, retrieval-augmented generation (RAG) could be applied to provide relevant knowledge dynamically.

Appendix A

Detailed Results for Ablation Studies

A.1 Ablation study: position of the explanation in Few-Shot examples

Model	Precision	Recall	F1 Score
CDL	7.30% ± 4.16	15.24% ± 9.72	9.83% ± 5.76
	7.85% ± 1.38	**23.81% ± 5.91**	**11.76% ± 2.14**
LM8	**5.46% ± 1.62**	42.86% ± 15.07	**9.68% ± 2.93**
	5.23% ± 1.62	**44.76% ± 12.26**	9.37% ± 2.86
LM70	**8.63% ± 2.16**	34.29% ± 7.71	**13.77% ± 3.29**
	7.61% ± 1.81	**43.81% ± 7.71**	12.95% ± 2.94
LM3	**8.47% ± 2.14**	25.71% ± 9.89	**12.72% ± 3.61**
	7.19% ± 1.19	**25.71% ± 3.24**	11.24% ± 1.75
QWC	6.87% ± 2.01	19.05% ± 4.18	10.08% ± 2.71
	8.11% ± 2.38	24.76% ± 8.77	**12.21% ± 3.78**

Table A.1: Project: **Codec**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

Model	Precision	Recall	F1 Score
CDL	**6.72% ± 3.85**	17.24% ± 8.56	**9.66% ± 5.31**
	5.88% ± 1.50	**20.00% ± 5.58**	9.08% ± 2.31
LM8	3.57% ± 1.30	24.83% ± 9.28	6.24% ± 2.28
	4.18% ± 1.49	**33.10% ± 11.57**	**7.42% ± 2.64**
LM70	4.65% ± 3.19	22.07% ± 15.32	7.68% ± 5.27
	6.15% ± 0.94	**40.69% ± 4.69**	**10.69% ± 1.57**
LM3	**4.78% ± 2.41**	10.34% ± 5.24	**6.54% ± 3.29**
	3.61% ± 1.19	**12.41% ± 3.83**	5.59% ± 1.82
QWC	3.64% ± 0.62	15.86% ± 2.35	5.92% ± 0.98
	4.65% ± 1.25	**20.00% ± 4.69**	**7.55% ± 1.98**

Table A.2: Project: **Gson**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

Model	Precision	Recall	F1 Score
CDL	**6.73% ± 2.27**	17.73% ± 7.83	**9.73% ± 3.49**
	5.90% ± 1.08	**20.45% ± 3.99**	9.16% ± 1.70
LM8	**5.38% ± 0.64**	59.09% ± 7.19	9.86% ± 1.18
	5.36% ± 1.02	**64.09% ± 12.20**	**9.90% ± 1.87**
LM70	**6.77% ± 0.77**	51.82% ± 6.12	**11.98% ± 1.37**
	6.21% ± 0.86	**57.73% ± 7.88**	11.21% ± 1.54
LM3	**9.25% ± 1.41**	46.82% ± 5.85	**15.45% ± 2.27**
	8.45% ± 0.86	**54.09% ± 6.12**	14.62% ± 1.49
QWC	6.97% ± 0.80	51.36% ± 6.50	12.27% ± 1.43
	7.56% ± 0.89	**56.36% ± 7.83**	**13.33% ± 1.61**

Table A.3: Project: **Collections**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

Model	Precision	Recall	F1 Score
CDL	16.24% ± 5.46	12.31% ± 5.70	13.95% ± 5.63
	20.24% ± 7.37	**20.00% ± 8.47**	**20.10% ± 7.88**
LM8	14.47% ± 1.56	31.28% ± 4.00	19.79% ± 2.24
	15.51% ± 1.34	**37.18% ± 2.52**	**21.88% ± 1.72**
LM70	**21.06% ± 2.99**	26.67% ± 4.56	23.52% ± 3.60
	19.24% ± 2.17	**33.08% ± 4.42**	**24.32% ± 2.91**
LM3	**17.37% ± 0.93**	17.69% ± 1.33	**17.53% ± 1.11**
	15.47% ± 3.18	**19.23% ± 4.06**	17.14% ± 3.55
QWC	17.95% ± 0.88	**31.28% ± 1.42**	**22.81% ± 0.98**
	18.00% ± 1.73	29.23% ± 2.36	22.28% ± 1.97

Table A.4: Project: **Mockito**; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

Model	Precision	Recall	F1 Score
CDL	9.05% ± 6.33	15.92% ± 3.46	10.72% ± 2.84
	9.58% ± 9.44	**20.81% ± 2.19**	**12.14% ± 7.16**
LM8	7.10% ± 6.59	41.87% ± 22.60	11.31% ± 7.79
	7.43% ± 7.13	**47.31% ± 21.01**	**12.03% ± 8.78**
LM70	**9.93% ± 9.98**	36.04% ± 19.93	13.98% ± 8.99
	9.48% ± 8.61	**45.64% ± 15.61**	**14.45% ± 8.75**
LM3	**9.97% ± 7.05**	27.94% ± 23.97	**13.40% ± 6.47**
	8.75% ± 6.59	**31.43% ± 28.06**	12.53% ± 6.75
QWC	8.74% ± 8.36	32.91% ± 23.85	12.88% ± 9.51
	9.42% ± 7.80	**36.19% ± 24.78**	**13.88% ± 8.17**

Table A.5: Weighted Averages; Top: explanation **before** the `has_bug` field; Bottom: explanation **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

A.2 Ablation study: analyzing different Few-Shot variants

A.2.1 Partition 2

Approach	Precision	Recall	F1 Score
FS	5.76% ± 1.97	27.06% ± 8.33	9.50% ± 3.18
P-FS	**7.83% ± 1.17**	**45.88% ± 10.83**	**13.37% ± 2.17**
FS-LE	7.10% ± 3.58	27.06% ± 16.00	11.23% ± 5.82

Table A.6: Project: **Csv**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	7.27% ± 0.90	55.00% ± 7.30	12.84% ± 1.60
P-FS	**7.41% ± 0.47**	**68.18% ± 5.99**	**13.37% ± 0.88**
FS-LE	7.13% ± 1.18	49.09% ± 8.61	12.45% ± 2.08

Table A.7: Project: **JxPath**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	3.94% ± 0.26	47.14% ± 2.53	**7.27% ± 0.47**
P-FS	3.51% ± 0.22	**58.21% ± 3.71**	6.62% ± 0.41
FS-LE	**3.94% ± 0.25**	38.93% ± 2.43	7.15% ± 0.44

Table A.8: Project: **JacksonCore**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	**7.77% ± 0.76**	51.35% ± 6.28	**13.50% ± 1.36**
P-FS	7.14% ± 0.70	**61.35% ± 6.67**	12.79% ± 1.27
FS-LE	7.20% ± 1.20	41.35% ± 6.56	12.26% ± 2.02

Table A.9: Project: **Compress**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	**5.12% ± 0.77**	55.63% ± 8.11	**9.37% ± 1.40**
P-FS	4.41% ± 0.42	**65.52% ± 5.80**	8.27% ± 0.79
FS-LE	5.02% ± 0.35	48.05% ± 3.09	9.08% ± 0.63

Table A.10: Project: **Lang**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	**7.29% ± 0.42**	41.58% ± 2.61	12.40% ± 0.71
P-FS	7.10% ± 0.09	**53.56% ± 2.36**	**12.54% ± 0.19**
FS-LE	7.26% ± 0.43	34.85% ± 2.20	12.01% ± 0.72

Table A.11: Project: **JacksonDatabind**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	**6.23% ± 1.43**	47.46% ± 7.31	**10.91% ± 2.20**
P-FS	5.91% ± 1.62	**58.82% ± 6.07**	10.65% ± 2.65
FS-LE	6.16% ± 1.38	40.19% ± 6.44	10.56% ± 1.99

Table A.12: Weighted Averages; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

A.2.2 Partition 3

Approach	Precision	Recall	F1 Score
FS	**7.57% ± 1.73**	52.00% ± 13.60	13.22% ± 3.07
P-FS	7.46% ± 2.82	**64.00% ± 20.78**	**13.35% ± 4.97**
FS-LE	5.84% ± 2.40	52.00% ± 22.21	10.49% ± 4.33

Table A.13: Project: **JacksonXml**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	6.38% ± 0.70	50.26% ± 6.21	11.32% ± 1.25
P-FS	6.86% ± 0.92	58.97% ± 8.12	12.29% ± 1.62
FS-LE	**6.97% ± 0.77**	**62.56% ± 5.33**	**12.54% ± 1.35**

Table A.14: Project: **Chart**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	4.66% ± 0.80	34.12% ± 5.86	8.20% ± 1.39
P-FS	**4.76% ± 0.61**	**38.43% ± 4.42**	**8.47% ± 1.07**
FS-LE	4.75% ± 0.52	38.04% ± 3.69	8.44% ± 0.89

Table A.15: Project: **Time**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	11.71% ± 2.83	45.38% ± 9.93	18.60% ± 4.38
P-FS	9.82% ± 0.99	48.46% ± 6.18	16.32% ± 1.71
FS-LE	**12.65% ± 0.94**	**55.77% ± 5.60**	**20.61% ± 1.58**

Table A.16: Project: **Cli**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	**6.46% ± 0.54**	44.53% ± 3.96	11.28% ± 0.94
P-FS	6.10% ± 0.37	**50.94% ± 3.12**	10.90% ± 0.67
FS-LE	6.37% ± 0.28	50.57% ± 4.83	**11.31% ± 0.54**

Table A.17: Project: **Jsoup**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	8.14% ± 0.24	44.83% ± 2.46	13.78% ± 0.45
P-FS	8.20% ± 0.53	49.40% ± 3.71	14.06% ± 0.90
FS-LE	**8.44% ± 0.80**	**50.20% ± 4.62**	**14.45% ± 1.37**

Table A.18: Project: **Math**; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

Approach	Precision	Recall	F1 Score
FS	6.99% ± 1.78	43.92% ± 4.88	12.00% ± 2.64
P-FS	6.83% ± 1.47	49.66% ± 6.17	11.96% ± 2.30
FS-LE	**7.17% ± 2.00**	**50.49% ± 7.06**	**12.49% ± 3.08**

Table A.19: Weighted Averages; Model: **LM70**; Format: value ± confidence interval; Best results in **bold**

A.3 Ablation study: position of the reasoning field in Chain-of-Thought approaches

Project	Precision	Recall	F1 Score
Csv	**8.78% ± 3.04**	27.06% ± 9.80	**13.23% ± 4.53**
	7.73% ± 1.29	**34.12% ± 6.11**	12.59% ± 2.07
JxPath	5.14% ± 1.64	24.55% ± 8.32	8.50% ± 2.74
	6.55% ± 0.97	**43.18% ± 6.91**	**11.37% ± 1.69**
JCore	**4.36% ± 1.30**	22.86% ± 6.73	**7.33% ± 2.17**
	3.85% ± 0.75	**35.36% ± 7.08**	6.95% ± 1.35
Compress	**6.48% ± 1.00**	22.97% ± 4.11	10.10% ± 1.60
	6.32% ± 0.42	**31.62% ± 3.48**	**10.54% ± 0.77**
JXml	3.54% ± 4.76	24.00% ± 32.38	6.16% ± 8.30
	4.34% ± 2.50	**36.00% ± 20.78**	**7.74% ± 4.46**
Cli	**10.04% ± 2.38**	36.54% ± 8.10	15.75% ± 3.68
	9.44% ± 0.55	**53.08% ± 4.34**	**16.03% ± 0.99**
W. Avg.	**6.14% ± 2.15**	25.45% ± 4.95	9.82% ± 3.11
	5.97% ± 2.03	**37.98% ± 7.49**	**10.24% ± 3.24**

Table A.20: Weighted Averages; Top: `reasoning` field **before** the `has_bug` field; Bottom: `reasoning` field **after** the `has_bug` field; Format: value ± confidence interval; Best results in **bold**

Bibliography

[1] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. «How Long Will It Take to Fix This Bug?» In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 2007, pp. 1–1. DOI: 10.1109/MSR.2007.13.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. «Who should fix this bug?» In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 361–370. ISBN: 1595933751. DOI: 10.1145/1134285.1134336. URL: https://doi.org/10.1145/1134285.1134336.

[3] Rafael-Michael Karampatsis and Charles Sutton. «How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset». In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. ACM, June 2020, pp. 573–577. DOI: 10.1145/3379597.3387491. URL: http://dx.doi.org/10.1145/3379597.3387491.

[4] Hussam Hourani, Ahmad Hammad, and Mohammad Lafi. «The Impact of Artificial Intelligence on Software Testing». In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 2019, pp. 565–570. DOI: 10.1109/JEEIT.2019.8717439.

[5] Vahit Bayrı and Ece Demirel. «AI-Powered Software Testing: The Impact of Large Language Models on Testing Methodologies». In: *2023 4th International Informatics and Software Engineering Conference (IISEC)*. 2023, pp. 1–4. DOI: 10.1109/IISEC59749.2023.10391027.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL]. URL: https://arxiv.org/abs/1409.3215.

[7] Yang Liu. *Fine-tune BERT for Extractive Summarization*. 2019. arXiv: 1903.10318 [cs.CL]. URL: https://arxiv.org/abs/1903.10318.

[8] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2020. arXiv: 1906.08237 [cs.CL]. URL: https://arxiv.org/abs/1906.08237.

[9] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. *SkCoder: A Sketch-based Approach for Automatic Code Generation*. 2023. arXiv: 2302.06144 [cs.SE]. URL: https://arxiv.org/abs/2302.06144.

[10] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé, and Xiaoguang Mao. «Natural Language to Code: How Far Are We?» In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 375–387. ISBN: 9798400703270. DOI: 10.1145/3611643.3616323. URL: https://doi.org/10.1145/3611643.3616323.

[11] Shangwen Wang, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Yan Lei, and Xiaoguang Mao. «Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network». In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622815. URL: https://doi.org/10.1145/3622815.

[12] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. *One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization*. 2023. arXiv: 2303.15822 [cs.SE]. URL: https://arxiv.org/abs/2303.15822.

[13] Chia-Yi Su and Collin McMillan. *Distilled GPT for Source Code Summarization*. 2024. arXiv: 2308.14731 [cs.SE]. URL: https://arxiv.org/abs/2308.14731.

[14] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. 2023. arXiv: 2302.06527 [cs.SE]. URL: https://arxiv.org/abs/2302.06527.

[15] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. *A3Test: Assertion-Augmented Automated Test Case Generation*. 2023. arXiv: 2302.10352 [cs.SE]. URL: https://arxiv.org/abs/2302.10352.

[16] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[17] Tim Menzies, Jeremy Greenwald, and Art Frank. «Data Mining Static Code Attributes to Learn Defect Predictors». In: *IEEE Transactions on Software Engineering* 33.1 (2007), pp. 2–13. DOI: 10.1109/TSE.2007.256941.

[18] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. «On the Accuracy of Spectrum-based Fault Localization». In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 2007, pp. 89–98. DOI: 10.1109/TAIC.PART.2007.13.

[19] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2025. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[20] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. «Software Testing With Large Language Models: Survey, Landscape, and Vision». In: *IEEE Trans. Softw. Eng.* 50.4 (Apr. 2024), pp. 911–936. ISSN: 0098-5589. DOI: 10.1109/TSE.2024.3368208. URL: https://doi.org/10.1109/TSE.2024.3368208.

[21] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536.

[22] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[23] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. «A neural probabilistic language model». In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[24] Andrew M. Dai and Quoc V. Le. *Semi-supervised Sequence Learning*. 2015. arXiv: 1511.01432 [cs.LG]. URL: https://arxiv.org/abs/1511.01432.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[26] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. *A Systematic Literature Review on Large Language Models for Automated Program Repair*. 2024. arXiv: 2405.01466 [cs.SE]. URL: https://arxiv.org/abs/2405.01466.

[27] Laria Reynolds and Kyle McDonell. *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm*. 2021. arXiv: 2102.07350 [cs.CL]. URL: https://arxiv.org/abs/2102.07350.

[28] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. «Interleaving Static Analysis and LLM Prompting». In: *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. SOAP 2024. Copenhagen, Denmark: Association for Computing Machinery, 2024, pp. 9–17. ISBN: 9798400706219. DOI: 10.1145/3652588.3663317. URL: https://doi.org/10.1145/3652588.3663317.

[29] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[30] OpenAI. *ChatGPT*. 2022. URL: https://openai.com/index/chatgpt/.

[31] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[32] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[33] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[35] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. «QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge». In: *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 55–56. ISBN: 9781450355148. DOI: 10.1145/3135932.3135941. URL: https://doi.org/10.1145/3135932.3135941.

[36] Kaibo Liu, Yudong Han, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Federica Sarro, Gang Huang, and Yun Ma. «TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs». In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 113–117. ISBN: 9798400705878. DOI: 10.1145/3643991.3644870. URL: https://doi.org/10.1145/3643991.3644870.

[37] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. «Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques». In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 464–468. DOI: 10.1145/3524842.3528482.

[38] René Just, Darioush Jalali, and Michael D. Ernst. «Defects4J: a database of existing faults to enable controlled testing studies for Java programs». In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

[39] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. «Dissection of a bug dataset: Anatomy of 395 patches from Defects4J». In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2018. DOI: 10.1109/saner.2018.8330203. URL: http://dx.doi.org/10.1109/SANER.2018.8330203.

[40] Just, René and Jalali, Darioush and Ernst, Michael D. *Defects4J, fault localization data*. 2017. URL: https://bitbucket.org/rjust/fault-localization-data/src/master/.

[41] Just, René and Jalali, Darioush and Ernst, Michael D. *Defects4J*. 2014. URL: https://github.com/rjust/defects4j.

[42] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. *How Far Have We Gone in Vulnerability Detection Using Large Language Models*. 2023. arXiv: 2311.12420 [cs.AI]. URL: https://arxiv.org/abs/2311.12420.

[43] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[44] URL: https://ollama.com/search.

[45] Qingxiu Dong et al. *A Survey on In-context Learning*. 2024. arXiv: 2301.00234 [cs.CL]. URL: https://arxiv.org/abs/2301.00234.

[46] Lei Huang et al. «A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions». In: *ACM Transactions on Information Systems* 43.2 (Jan. 2025), pp. 1–55. ISSN: 1558-2868. DOI: 10.1145/3703155. URL: http://dx.doi.org/10.1145/3703155.

[47] URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-8000-us-nvidia-946977-r1-web.pdf.

[48] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Mar. 2018. URL: https://program-repair.org/defects4j-dissection/#!/.

[49] URL: https://soot-oss.github.io/soot/.

[50] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. *AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts*. 2020. arXiv: 2010.15980 [cs.CL]. URL: https://arxiv.org/abs/2010.15980.

[51] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. *A Deep Dive into Large Language Models for Automated Bug Localization and Repair*. 2024. arXiv: 2404.11595 [cs.SE]. URL: https://arxiv.org/abs/2404.11595.

[52] Shuai Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664 [cs.SE]. URL: https://arxiv.org/abs/2102.04664.

[53] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. «Automated Program Repair in the Era of Large Pre-trained Language Models». In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 1482–1494. DOI: 10.1109/ICSE48619.2023.00129.

[54] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. *Automated Repair of Programs from Large Language Models*. 2023. arXiv: 2205.10583 [cs.SE]. URL: https://arxiv.org/abs/2205.10583.

[55] Julian Aron Prenner, Hlib Babii, and Romain Robbes. «Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs». In: *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 2022, pp. 69–75. DOI: 10.1145/3524459.3527351.

[56] Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. «Patch Generation with Language Models: Feasibility and Scaling Behavior». In: *Deep Learning for Code Workshop*. 2022. URL: https://openreview.net/forum?id=rHlzJh_b1-5.

[57] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. *Examining Zero-Shot Vulnerability Repair with Large Language Models*. 2022. arXiv: 2112.02125 [cs.CR]. URL: https://arxiv.org/abs/2112.02125.

[58] Jie Lin and David Mohaisen. «From Large to Mammoth: A Comparative Evaluation of Large Language Models in Zero-Shot Vulnerability Detection». In: Jan. 2025. DOI: 10.14722/ndss.2025.241491.

[59] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. «A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries». In: *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 2020, pp. 508–512. DOI: 10.1145/3379597.3387501.

[60] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. «A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection». In: *CoRR* abs/2403.17218 (2024). URL: https://doi.org/10.48550/arXiv.2403.17218.

[61] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Hengbo Tong, Swarna Das, Earl T. Barr, and Wei Le. *To Err is Machine: Vulnerability Detection Challenges LLM Reasoning*. 2025. arXiv: 2403.17218 [cs.SE]. URL: https://arxiv.org/abs/2403.17218.

[62] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. *Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities*. 2024. arXiv: 2311.16169 [cs.CR]. URL: https://arxiv.org/abs/2311.16169.

[63] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. *LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks*. 2024. arXiv: 2312.12575 [cs.CR]. URL: https://arxiv.org/abs/2312.12575.

[64] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. «Software Vulnerability Detection using Large Language Models». In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2023, pp. 112–119. DOI: 10.1109/ISSREW60843.2023.00058.

[65] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. «VulDeePecker: A Deep Learning-Based System for Vulnerability Detection». In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. DOI: 10.14722/ndss.2018.23158. URL: http://dx.doi.org/10.14722/ndss.2018.23158.

[66] Guru Bhandari, Amara Naseer, and Leon Moonen. «CVEfixes: automated collection of vulnerabilities and their fixes from open-source software». In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE '21. ACM, Aug. 2021, pp. 30–39. DOI: 10.1145/3475960.3475985. URL: http://dx.doi.org/10.1145/3475960.3475985.

[67] Chongzhou Fang et al. *Large Language Models for Code Analysis: Do LLMs Really Do Their Job?* 2024. arXiv: 2310.12357 [cs.SE]. URL: https://arxiv.org/abs/2310.12357.

[68] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. «Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach». In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649828. URL: https://doi.org/10.1145/3649828.

[69] Wei Ma et al. *LMs: Understanding Code Syntax and Semantics for Code Analysis*. 2024. arXiv: 2305.12138 [cs.SE]. URL: https://arxiv.org/abs/2305.12138.