Master's Thesis

Code Comprehension in LLMs: A Data flow perspective and empirical study on def-use chains, program slicing and output prediction

Ritika Basavaraj Hiremath

March 15, 2026

Advisor:

Anna-Maria Maurer	Chair of Software Engineering
Sebastian Böhm	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Vera Demberg	Chair of Computer Science and Computational Linguistics

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

To my mother,
for her unwavering support, encouragement, and belief in me throughout this journey.

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und zur Code-Vervollständigung und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision and code completion, and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional) Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Abstract

Code comprehension is a challenge for both developers and learners. It requires understanding syntax as well as the ability to trace how data flows and transforms throughout a program. With the rise of Large Language Models (LLMs), developers and learners increasingly rely on them to support code understanding. LLMs, however, are known to produce incorrect outputs when code complexity increases. This raises concerns about their reliability, especially in situations where developers must trace data, reason about program behavior, and identify where values change.

In this thesis, we investigated the extent to which LLMs can comprehend data flow in program snippets, focusing on def-use chains and related concepts, including Control Flow Graph (CFG), program output, and program slicing. To this end, we constructed a custom dataset of Kotlin code snippets, organized into levels of increasing data-flow complexity, each grounded in specific programming concepts. We evaluated the performance of the LLM across tasks ranging from identifying basic program elements such as defs, uses, function calls, basic blocks, block entry, and block exit, to constructing def-use chains and performing backward program slicing, under multiple prompting strategies including Zero-shot, One-shot, Chain of Thought (CoT), and One-shot CoT prompting.

We found that performance degraded at higher levels of complexity, with def-use chain construction proving particularly challenging across prompting strategies. This indicated that certain data flow tasks place demands on LLMs that prompting strategies alone cannot overcome. CoT and One-shot prompting exhibited a measurable, but task-dependent, influence, suggesting that no single strategy consistently improved performance. Lines Of Code (LOC) emerged as the complexity metric most consistently associated with performance degradation, which allowed us to identify it as a reliable predictor of where LLM performance breaks down across tasks, levels of code complexity and prompting strategies.

Together, these findings characterize the current limitations of LLM in data flow comprehension and demonstrate that code complexity metrics, particularly LOC and Cyclomatic complexity, served as meaningful indicators of where and how LLM performance degraded. This allowed us to conclude that complexity-aware evaluation is essential for understanding LLM reliability, and offered concrete insights that can inform future efforts to improve LLM support for complex code tasks.

Overall, this thesis provided insights into the limitations of LLMs in data flow comprehension and identified code complexity metrics as meaningful indicators of performance degradation even in data flow based complex code snippets.

Acknowledgments

I would like to express my sincere gratitude to my advisors, Anna-Maria Maurer and Sebastian Böhm, for dedicating additional time to supporting the development of my dataset, providing regular and constructive feedback, and accompanying me throughout this journey with consistent encouragement.

I owe particular thanks to Anna-Maria Maurer, who offered critical and thoughtful feedback even during her busiest periods. Your dedication in making time for me and your valuable guidance have significantly contributed to the quality of my writing.

I would also like to extend my gratitude to Prof. Dr. Sven Apel for providing me with the opportunity to write this thesis within the Chair of Software Engineering. It has been a privilege to work in such a stimulating research environment.

I would additionally like to thank Dr. Thomas Bock, under whose supervision I worked as a research assistant at the chair. This experience helped me develop a deeper understanding of and improve the code structures relevant to research.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout this journey.

Contents

List of Figures

List of Tables

Listings

Acronyms

AST Abstract Syntax Tree
ART Aligned Rank Transform
ART ANOVA Aligned Rank Transform Analysis of Variance
ANOVA Analysis of Variance
CoT Chain of Thought
CDG Control Dependency Graph
CFG Control Flow Graph
DDG Data Dependency Graph
DFA Data Flow Analysis
DFG Data Flow Graph
FP False Positive
FN False Negative
IDF Implicit Data Flow
IoU Intersection over Union
JSON JavaScript Object Notation
LLM Large Language Model
LOC Lines Of Code
OOP Object Oriented Programming
TP True Positive

1 Introduction

LLMs have become increasingly prominent in software engineering, with applications spanning code generation, bug detection, and code comprehension [22]. Since the introduction of ChatGPT in late 2022, research on LLMs has grown exponentially [34], with many studies evaluating their performance on code-related tasks such as code completion [53], bug detection [3, 45], and code generation [8, 10, 26, 34, 46]. While earlier models showed positive results on simpler programming assessments, they struggled with more complex ones [37] and despite significant progress, LLMs still cannot be fully relied upon to generate correct code [46]. Furthermore, the benchmark datasets commonly used to evaluate LLMs on code generation tasks have become outdated and unreliable, as they risk being absorbed into model training data [9, 20, 54]. This is particularly concerning given that developers and code learners increasingly rely on LLMs when they encounter unfamiliar code or struggle to understand why their program produces an unexpected result, situations that demand not just code generation, but the ability to trace data, reason about program behavior, and identify where values change.

Previously, researchers have explored various aspects of code to understand how LLMs comprehend it. Some focused on code semantic relationships [23, 31] while others focused on the ability of the LLM to detect errors [45]. Among the code representations used in such analyses, CFG has emerged as the most preferred [24, 31, 45], as it captures the different execution paths possible within a program. However, CFG alone does not reveal how data traverses through those paths, how values are defined, propagated, and consumed across a program. Furthermore, prior work has predominantly relied on code generation-based datasets, which test only the generation ability of LLMs and cannot provide complete insight into their code comprehension ability. This leaves a clear gap: whether LLMs can genuinely reason about data flow in a program remains largely unexplored.

Data flow analysis models the relationship of data in any program snippet [27], capturing how values are defined, propagated, and consumed across execution. This makes it a powerful technique not only for practical tasks such as code optimization, error detection, and model correctness checking, but also as a lens for evaluating code comprehension. Specifically, by observing how a variable is updated at different points in a program, one can assess whether an LLM has genuinely understood the behavior of a program. To do so the LLM is required to understand control flow, identify where variables are defined and used, determine which variables influence one another, and account for all factors that contribute to the final result. Data flow analysis thus provides a structured framework through which all of these aspects can be examined together. This potential has already been recognized in prior work: Huang et al. [23] applied data flow analysis concepts to Python to address

implicit data flow problems, ultimately drawing on def-use chains to design a solution that reduced such issues.

The goal of this thesis was to investigate the extent to which LLMs can comprehend data flow in program snippets. To this end, it relied on data flow analysis concepts to test the ability of LLMs to comprehend the intricacies of data flow within a program, using code snippets that were unlikely to be part of their training data.

To investigate data flow comprehension in LLMs, we formulated six research questions. The first three examined the ability of LLMs to perform data flow analysis tasks across various prompting strategies, while the last three explored how code complexity affected performance under those strategies. To answer these questions, we developed a dataset of Kotlin code snippets, organized into levels of increasing data flow complexity, and evaluated LLM performance across tasks and prompting strategies. Results showed that both prompting strategy and complexity affected performance, though their effects were task-dependent. Overall, LLM performance tended to decline at higher complexity levels, while certain prompting strategies improved results for specific tasks. Additionally, certain complexity metrics consistently correlated with performance degradation across all tasks. These findings characterize the current limitations of LLMs in data flow comprehension and offer practical insights for both developers and learners seeking to understand where LLM support is reliable. For researchers, this thesis highlights the data flow gap in current model capabilities that warrant further improvements.

2

Background

This chapter establishes the theoretical foundation of the thesis. It presents key concepts in data flow analysis, program comprehension, and code complexity metrics, and introduces the prompting strategies, task definitions, evaluation metrics, and statistical tests used in this study. Finally, it provides an overview of the large language model under evaluation.

2.1 Data Flow Analysis

Data Flow Analysis (DFA) models the relationships of data [27]. There are two primary approaches to data flow analysis: static and dynamic.

Static data flow analysis examines the program without executing it by inspecting the source code or intermediate representations to identify possible data paths. In contrast, dynamic data flow analysis tracks data behavior during program execution, often requiring instrumentation and runtime monitoring [6]. This technique computes possible values at various program points, enabling a deeper understanding of data dependencies and code behavior [28].

Static analysis techniques provide a scalable and efficient means to analyze programs, particularly useful in large-scale systems where execution is infeasible. Moreover, static analysis enables early detection of potential issues such as uninitialized variables, dead code, unreachable branches, and many more. It can be applied across diverse codebases without requiring test cases or execution environments, making it especially suitable for automated evaluation scenarios. Given these advantages, this thesis adopts static analysis as a foundational method for evaluating code comprehension abilities of LLMs. Static analysis is not only computationally efficient but also aligns well with the input-output nature of LLMs, where execution is not always possible or practical.

This section introduces essential definitions in data flow analysis, along with related concepts like control flow, program output, and program slicing, which underpin the analysis and comprehension of program behavior. Listing 2.1 illustrates these definitions with a simple code snippet that returns either the sum or the product of all digits in the variable (n).

Listing 2.1: Example Kotlin code snippet

```
1. sum_and_prod(var n){      // def:n;
2. var q = n%10              // def: q; use: n;
3. var r = n/10              // def: r; use: n;
```

```
4. var sum = 0                // def: sum;
5. var prod = 1               // def: prod;
6. var count = 0              // def: count;
7. while(n>0){                // use: n;
8.   sum += n%10              // def: sum; use: n,sum;
9.   prod *= n%10             // def: prod; use: n,prod;
10.  n = n/10                 // def: n; use: n;
11.  count++  }               // def: count; use: count;
12. if(count%2 ==0){          // use: count;
13.  return prod }            // use: prod;
14. else{
15.  return sum }}            // use: sum;
```

2.1.1 Def and Use

When a compiler runs a program, it writes every data assignment to an entity in memory, which we call a def. When the program reads this data from memory, it performs a use. Together, def-use pairs form the basis of data flow analysis, as they reveal how information propagates through the program. Entities are Variables (local variables, parameters), Functions (method definitions and calls), Classes (class definitions and usage), and Constructors (when creating objects), all created when data is written. Listing 2.1 shows defs and uses at each line of the example code.

Defs and uses definitions serve a critical purpose in data flow analysis. They are the foundations to trace dependencies between entities, identify which parts of the code affect others, and detect potential issues such as uninitialized entities, improper data usage, or unused entities.

This analysis provides valuable insights, such as the exact lines where an entity is defined and subsequently used. This information is essential for more advanced static analyses, such as code optimization, code refactoring, or bug detection. It serves as a foundation for understanding the overall data flow within a program.

2.1.2 Def-Use Chains

Table 2.1 presents the def-use chains for the code snippet shown in Figure 2.1. For every def of an entity in a program, we form a chain linking that def to all reachable uses without any intervening defs of the same entity. This chain is known as a def-use chain, connecting each def of an entity directly to all its corresponding uses. The format is: entity @ def_line $\rightarrow$ entity @ use_line. Variables with no uses are marked accordingly. (q @ 2), (r @ 3) and (sum_and_prod @ 1) do not have any uses of their definitions. Similarly, for every use of an entity, we form a use-def chain by linking that use to all possible definitions of the entity within the program.

Def@line_no → use@line_no
n @ 1 → n @ 2
n @ 1 → n @ 3
n @ 1 → n @ 7
n @ 10 → n @ 7
n @ 10 → n @ 8
n @ 10 → n @ 9
n @ 10 → n @ 10
sum @ 4 → sum @ 8
sum @ 8 → sum @ 8
sum @ 8 → sum @ 15
prod @ 5 → prod @ 9
prod @ 9 → prod @ 9
prod @ 9 → prod @ 13
count @ 6 → count @ 11
count @ 11 → count @ 11
count @ 11 → count @ 12

Table 2.1: Def-Use Chains Present in `sum_and_prod` Function Code

2.1.3 Program Slicing

In a program snippet *p*, and for an entity *e* defined or used at a particular line textitl, a *program slice* consists of all statements in textitp that affect the value of textite at line textitl, either directly or indirectly [28]. This set of relevant statements helps isolate the minimal subset of code that influences or is influenced by the entity in question, aiding in program comprehension, debugging, and maintenance. There are two main types of slicing: static slicing and dynamic slicing. Static slicing computes the slice based solely on the source code, without requiring program execution. Dynamic slicing requires a specific execution trace and produces slices dependent on actual inputs and runtime behavior [48].

Static slicing is preferred in this thesis as it allows for a general-purpose analysis without requiring execution, which aligns with the nature of static data flow analysis. Static slicing is also more scalable for analyzing many code snippets or large code bases, where runtime traces may be unavailable or infeasible to obtain.

Slicing is divided further into two categories: forward slicing and backward slicing. Forward slicing identifies all statements affected by the value at a given point (i.e., how data flows forward). Backward slicing identifies all statements that influence the value at that point (i.e., where the data came from) [43].

Listing 2.2: Slice of the variable sum at line 15

```
1. sum_and_prod(val n: Int){
2. var sum: Int = 0
3. while(n>0){
4. sum += n%10
5. n = n/10
```

A backward slice focuses on the parts of the program that influence the value of a particular variable at a given point. For instance, Listing 2.2 presents a backward slice for the variable (sum) at line 15. This slice consists of all the lines of code that affect the variable (sum). This example demonstrates how slicing narrows down the code to the relevant statements that affect the value of a variable. In the context of this thesis, backward slicing is more relevant because the goal is to trace how an entity's value is constructed or influenced before a given point. It offers insight into how large language models understand data dependencies and entity life cycles.

2.2 Control Flow Analysis

2.2.1 Basic Blocks

Basic blocks consist of sequences of code statements that execute together without any internal branching [27]. Each basic block has only one entry point and one exit point, meaning that control always enters at the beginning and leaves at the end without halting or branching in between. This structure allows the grouping of code lines that form an indivisible unit of execution.

Figure 2.1 presents the numbering of basic blocks, assigning each line of code to its respective block. For example, lines 1-6 constitute the first basic block. Similarly, line 7 forms the second basic block, lines 8-11 make up the third basic block, and so on.

Basic blocks play a foundational role in data flow analysis by serving as the structural units from which CFGs are constructed. Organizing code into basic blocks allows data flow analysis to efficiently track definitions and uses both within and across blocks, enabling accurate identification of variable lifetimes, unreachable code, and optimization opportunities. Since execution within a basic block is strictly sequential with a single entry and exit point, each block provides a well-defined scope for analyzing which definitions reach which uses. This makes basic blocks the atomic units of control flow and the natural building blocks for constructing def-use chains, as they provide clear boundaries that determine where a definition is made and where its corresponding uses can be reached across the program.

2.2.2 Control Flow Graph (CFG)

A CFG models the possible execution paths of a program [27]. The basic blocks are the nodes of the CFGs, and the edges connect different nodes in a particular direction, representing the direction of the Graph [2]. Every CFG has exactly one entry node and one exit node. For example, Figure 2.1 illustrates the control flow of the code snippet, Listing 2.1.

While CFGs themselves are not a part of data flow analysis, they are an essential input to it[1]. Data flow analysis operates over the structure defined by the CFG, tracking how data is defined, used, and propagated across all possible execution paths. Without this structural foundation, DFA would lack the control context necessary to reason precisely about program behavior at each point of execution.

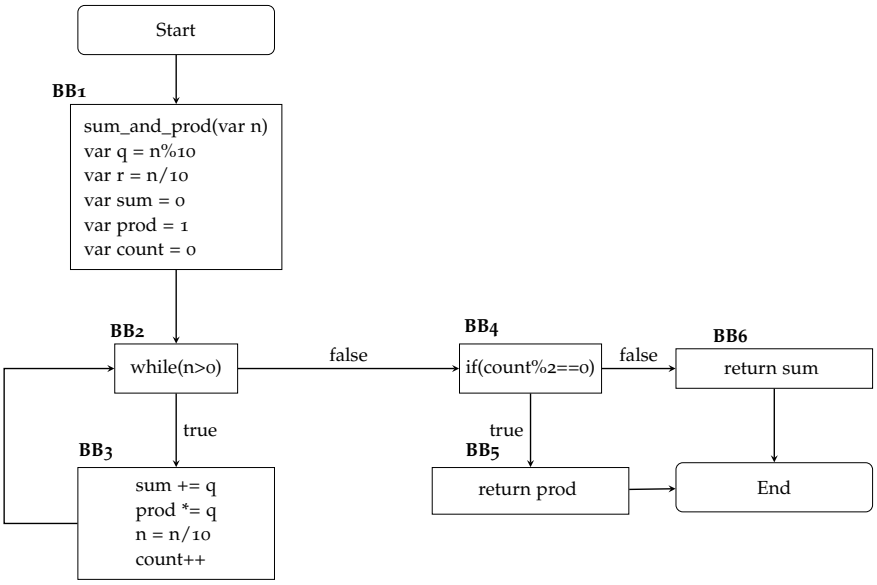

Figure 2.1: Control flow graph with basic blocks of Listing 2.1

2.3 Program Comprehension

Program Comprehension is the concept of gaining or regaining knowledge from a computer program [19]. When a developer attempts to understand a program, they first construct a mental representation of the code's structure, behavior, and purpose. This is known as a cognitive model [41]. This model is built incrementally by tracing how data flows through the program: following how variables are defined, modified, and used across execution paths helps a developer reason about what a piece of code actually does.

When evaluating whether an LLM understands code, we can draw a parallel with this human process. If a model truly comprehends a piece of code, it should implicitly reason about the same data dependencies a developer would trace. By measuring how well an LLM responds to tasks that require awareness of these dependencies, data flow analysis provides a structured and principled lens through which to evaluate code comprehension.

2.4 Code Complexity Metrics

The difficulty of understanding or maintaining a program is often estimated using code complexity metrics. These metrics aim to approximate how challenging a code base may be for humans to comprehend. However, none of them can fully capture the cognitive effort

involved in reading or analyzing code. Over the years, various metrics have been proposed to quantify code complexity, especially in the context of software engineering and program analysis [18, 32, 36].

Lines of Code (LOC) - the number of lines of source code or executable code present. A higher LOC generally indicates that a program performs more operations, manages more state, or handles more cases. All of these tasks demand greater comprehension effort from a developer. However, this is not always true. A snippet with a smaller LOC can encode highly intricate logic through dense expressions, recursion, or deeply nested control flow [32]. Nevertheless, LOC remains a useful baseline metric because it provides a direct measure of the volume of code that must be understood.

Cyclomatic Complexity - It is a quantitative measure of the number of linearly independent paths through a program's source code[32]. This metric is particularly useful when data is updated differently across different paths, as it directly reflects the number of distinct behavioral scenarios a reader must mentally account for to fully understand the program.

Peitek et al. [36] investigated how code complexity metrics reflect the difficulty of program comprehension with human participants. They conducted a study with 19 participants and 16 code snippets to contrast code complexity in 4 different complexity metrics, namely, code size (LOCs), vocabulary size ([18]), control flow (McCabe's [32]), and data flow (DepDegree [49]). Their results confirm that code complexity meaningfully impacts both researchers and practitioners, and that these metrics capture dimensions of comprehension effort that manifest in measurable ways. This thesis draws a direct parallel by treating an LLM as a participant in place of a human developer, investigating whether the same complexity metrics that predict human comprehension difficulty also influence the performance of an LLM on code comprehension tasks.

2.5 Kotlin

Kotlin is a prominently used programming language by Android developers. It is known for its concise syntax, null safety, interoperability with Java, and support for functional programming paradigms. In 2017, Google announced Kotlin to be the official programming language in Android development [14].

So far, the previous research has predominantly focused on programming languages like Python [12] [33] [23] and Java [24] [31]. In contrast, there has been limited attention given to Kotlin, despite its increasing adoption in the IT industry.

Beyond its popularity, Kotlin offers several unique features, including immutable collections, iterators, and data classes, which make it an attractive candidate for studying code comprehension in large language models. While some studies have included Kotlin among several languages [42], its distinct characteristics make it a suitable language for our analysis.

2.6 Evaluation Metrics

Precision and Recall quantify false positives and false negatives, while the Jaccard index measures the overall similarity between predicted results and ground truth. These metrics were selected because true negatives are not informative in this context; in LLM responses, true negatives can be infinite, as the model may correctly exclude an unbounded number of irrelevant elements. As Crall [11] demonstrated, metrics involving true negatives lose interpretability in highly imbalanced settings. In contrast, similarity-based metrics focus solely on the overlap between predicted and ground-truth sets, making them more appropriate when the evaluation goal is to assess how closely the model output matches the expected result. This property is well established in related domains such as object detection, where Intersection over Union (IoU), mathematically equivalent to the Jaccard index, is widely used as a primary evaluation metric [35]. Together, these three metrics provided a comprehensive and interpretable basis for evaluating LLM performance across tasks.

Precision Precision measures the proportion of correctly predicted positive instances among all predicted positives.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall Recall measures the proportion of correctly predicted positive instances among all actual positives.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Jaccard Index The Jaccard index measures the similarity between the predicted and ground-truth sets of values. Values closer to 1 indicate higher similarity between predictions and ground truth.

$$\text{Jaccard Index} = \frac{\text{TP}}{\text{TP} + \text{FN} + \text{FP}}$$

2.7 Statistical Tests

Statistical tests are used to determine whether observed differences or relationships in results are meaningful rather than due to chance. This section introduces the statistical methods applied in this thesis.

Friedman Test The Friedman test is a non-parametric alternative to the repeated measures ANOVA, used to detect differences across multiple conditions or groups [13]. It ranks the results within each observation and tests whether the distributions differ significantly across conditions. When a significant difference is found for this test, post-hoc analysis is required to identify the significantly different pairs.

Wilcoxon Signed-Rank Test The Wilcoxon signed-rank test is a non-parametric post-hoc test used to identify significant differences between pairs of conditions [50]. For each task, the best result per condition is selected as the representative attempt before applying the test. When multiple tests are performed, we utilize the Bonferroni correction to avoid an increase in false positives. The effect size for the Wilcoxon Signed-Rank test is interpreted as follows:

$$\begin{aligned} 0.1 \leq r < 0.3 &: \text{Small effect} \\ 0.3 \leq r < 0.5 &: \text{Moderate effect} \\ r \geq 0.5 &: \text{Large effect} \end{aligned}$$

where r is the effect size obtained from the Wilcoxon Signed-Rank test.

ART ANOVA The Aligned Rank Transform (ART) is a non-parametric method that enables the use of Analysis of Variance (ANOVA)-style analysis on data that does not meet the assumption of normality [51]. Standard ANOVA requires data to be normally distributed. However, when this assumption is violated, results may be unreliable. Aligned Rank Transform Analysis of Variance (ART ANOVA) addresses this by first aligning the data by removing the effects of all other factors from each observation and then ranking the aligned values.

Spearman's Rank Correlation Coefficient (ρ) Spearman's Rank Correlation Coefficient [40] is a non-parametric statistical test that compares the ranks of two variables and measures the dependence between them. Multiple tests using Spearman's rank correlation coefficient are performed on the same data, and the changes in false positives increase. To control for this, a False discovery rate correction using the Benjamini-Hochberg procedure [5] is applied. The Benjamini–Hochberg procedure controls the FDR at level α.

2.8 Large Language Model

LLMs are deep learning models trained on extensive and diverse datasets comprising natural language text, programming code, and other structured or unstructured data sources [44]. These models are typically based on transformer architectures and contain billions of parameters, enabling them to learn linguistic patterns, contextual relationships, and structural representations. LLMs train on vast and diverse datasets, enabling them to finetune for specific tasks, including code-related ones such as debugging, code generation, optimization, and more [10]. Once trained, LLMs can be used in a variety of ways, either through prompting (Zero-shot or Few-shot), finetuning on domain-specific data, or through tool integration in platforms such as **IDE!**s (**IDE!**s) or chat-based interfaces.

LLMs are increasingly becoming vital to a range of tasks in various fields, from education to industry [8]. Their usage has increased exponentially. They are used for text-generation [29], and multimodal generation like image, video, 3D, and audio content generation [21]. In code-related tasks, LLMs are used for code completion [53], bug detection [3, 45], and code generation [26]. However, they often rely heavily on surface-level patterns rather than a deep semantic understanding of code [46].

Several factors influence the performance of LLMs, including the quality and diversity of their training data, the model size, the task specificity, the prompting strategy, and whether developers have finetuned the model for a given domain. Prompting techniques such as CoT, In-context learning, and Few-shot examples can significantly impact the quality of their output, particularly in reasoning-intensive tasks like code comprehension or data flow analysis.

DeepSeek-Coder 6.7B Instruct DeepSeek-Coder 6.7B Instruct is an open-source code model from the DeepSeek-Coder series with 6.7B parameters. DeepSeek-Coders models outperform benchmark open-source models and closed-source models like Codex and GPT-3.5 on code-based datasets [16]. DeepSeek Coder Instruct models are meticulously finetuned using instructional data and achieve better results than the OpenAI GPT-3.5 Turbo model in code-related tasks [16]. The Methodology chapter outlines the justification for the choice of this model.

2.8.1 Prompting types

Prompting strategies define how instructions and context are provided to an LLM, and have a significant influence on the quality and nature of its outputs. This section introduces the four strategies used in this thesis.

Zero-shot prompting provides the model only with the task description and relevant definitions, offering no examples or demonstrations. This serves as a baseline, revealing the intrinsic ability of the LLM to reason about a problem based solely on its pre-trained knowledge.

Chain of Thought(CoT) prompting instructs the model to generate intermediate reasoning steps before arriving at a final answer [47]. This encourages more structured, step-by-step reasoning and has been shown to improve performance on tasks requiring logical inference.

One-shot prompting supplements the task description with a single worked example - a code snippet paired with its expected solution. This provides the model with a concrete demonstration of the expected output format and reasoning style.

One-shot CoT prompting combines both approaches, One-shot prompting and CoT prompting. A single example is provided alongside its step-by-step reasoning and solution. This encourages the model to both follow the demonstrated format and explicitly articulate its reasoning process.

The prompt formats for Zero-shot, CoT, One-shot, and One-shot CoT prompting are presented below, illustrating how each strategy is structured in practice. Each format contains placeholders for the task definition and expected output format, which are populated with the specific tasks described in the methodology.

Zero-shot prompting

[Task definitions]
Based on these definitions, answer the following questions in the format mentioned.
[Format for each task]

CoT prompting

[Task definitions]
Based on these definitions for each task, perform chain-of-thought prompting and then provide your answers in the provided JSON format.
[Format for each task]

One-shot prompting

[Task definitions]
Based on these definitions, answer the following questions in the format mentioned.
[Format for each task]
Here is an example code snippet and its solution in the expected format.
[Code Snippet and its solutions in the required format]

One-shot CoT prompting

[Task definitions]
Based on these definitions, perform chain-of-thought prompting and then provide your answers in the provided JSON format.
[Format for each task]
Here is an example code snippet and its solution in the expected format.
[Code Snippet and its solutions in the required format]

3

Methodology

This chapter describes the methodology used for the core evaluation of the thesis. It introduces research questions and gives insight into the operationalization and evaluation pipeline followed in this thesis.

As part of our preliminary investigation, we explored several foundational concepts in data flow analysis, including flow sensitivity, path sensitivity, and context sensitivity, along with their insensitive counterparts. We tested the performance of the LLM model Claude Sonnet 4 on these sensitivity-based concepts. From this preliminary testing, the model was observed to handle questions based on these concepts effectively. However, we observed more frequent and consistent errors in tasks regarding the identification of basic blocks and the generation of def-use chains. Hence, for this study, we selected def-use chains, basic block construction, program slicing, and related topics as the primary focus for further detailed evaluation.

3.1 Research Questions

This thesis is divided into six research questions that focus on program comprehension of LLMs. The first three questions investigate the precision of LLMs in data analysis concepts, while the last three examine the impact of code complexity metrics on LLMs.

We base all six research questions on four types of prompting strategies: Zero-shot prompting, One-shot prompting, CoT prompting, and One-shot CoT prompting. This helps us analyze the effects of the CoT-based and One-shot-based prompting approaches, both together and separately.

3.1.1 RQ: 1-3

RQ1: How precisely can the LLM identify all the defs, uses, function calls, and basic blocks with snippets with increasing complexity of data flow across various prompting methods?

RQ2: How precisely can the LLM create def-use chains with snippets with increasing complexity of data flow across various prompting methods?

RQ3: How precisely can the LLM identify the output of the code snippet and perform backward program slicing with snippets with increasing complexity of data flow across various prompting methods?

Motivation We focused on the above research questions and examined the performance of the LLM for Zero-shot, One-shot, CoT, and One-shot CoT prompting. **RQ1** focused on foundational tasks such as identifying definitions, uses, function calls, and basic blocks. These tasks are crucial for static analysis and compiler design, and represent the necessary building blocks before any higher-level data flow reasoning can be performed. Identifying definitions and uses is a prerequisite for forming def-use chains, while basic blocks provide insight into the CFG comprehension of the LLM and can help in the formation of Data Flow Graph (DFG). **RQ2** focused on the ability of the LLM to construct def-use chains rather than use-def chains, as def-use chains trace how a defined variable propagates forward through a program, which is more directly relevant to understanding data flow. Def-use chains offer a fundamental view into how data flows through a program snippet, and constructing them correctly requires the model to identify redefinitions, track usages, and account for ambiguous or overlapping data flows. **RQ3** investigated the ability of the LLM to accurately predict the final output of a program snippet and to perform program slicing, both of which require tracing data flow and simulating execution. These tasks tested whether the model could not only interpret code syntactically but also reason semantically about how values propagate and change throughout a program. Together, these tasks formed a structured progression from basic identification to complex reasoning, offering a comprehensive view of LLM data flow comprehension under various prompting strategies and levels of complexity.

Operationalization To perform the analysis, we examine the data distribution and assess significance for each prompting type, each level, and the combination of prompting type and level. If significant, we perform additional statistical tests to identify significant pairs and their effects. All the tests are performed separately for each evaluation metric.

3.1.2 RQ: 4-6

RQ4: How does the complexity of code measured by different metrics affect the precision of the LLM on the task of identification of defs, uses, function calls, and basic blocks with snippets with increasing complexity of data flow in various prompting?

RQ5: How does the complexity of code measured by different metrics affect the precision of the LLM on the task of def-use chains creation with snippets with increasing complexity of data flow in various prompting?

RQ6: How does the complexity of code measured by different metrics affect the precision of the LLM on snippet output identification and program slicing tasks with snippets with increasing complexity of data flow in various prompting?

Motivation The research questions **RQ4-6** focus on the same set of tasks as **RQ1-3**. These research questions specifically investigate how code complexity metrics correlate with the capability of the LLM to accurately perform these tasks. The objective is to assess whether well-established code complexity metrics evaluate the robustness of the LLM across various prompting strategies and at different levels of data flow complexity, examining whether they mitigate or exacerbate the effects of complexity.

We utilize LOC and Cyclomatic complexity. The snippets combine various data structures, object-oriented programming concepts, and general programming concepts such as recursion. Each snippet contains at least one independent path and entities that are both defined and used. These properties make metrics like Cyclomatic complexity reliable indicators of complexity across all snippets. LOC is not a very reliable metric for code complexity, as a program snippet with fewer lines of code can be just as, or even more, complex than one with more lines. However, it can still be used as a parameter to ensure the snippets are varied beyond just their data structure or Object Oriented Programming (OOP) properties.

Operationalization To answer this question, we observe the significance and effect of the two code complexity metrics on the global data. If found significant, we further examine the significance of each metric for each prompting type and level across all three evaluation metrics (Precision, Recall, and Jaccard Index). Doing so helps us obtain a detailed view of the overall effect of each complexity metric on TP, FN, and FP.

3.2 Operationalization

3.2.1 Code Snippets

The selection of the code snippets plays a vital role in the validity and generalizability of this study. LLMs are known to be trained on open-source code corpora; the exact datasets and repositories used during pretraining are often not publicly disclosed [15]. To mitigate the risk of testing with data the model may have already seen during training, we created a new dataset for this study.

A total of 88 Kotlin-based code snippets were finalized and developed. The concepts were selected to cover a broad range of programming paradigms, including general programming concepts (e.g., recursion, inheritance, arrays, trees, polymorphism), Kotlin-specific features (e.g., mutableListOf, mutableMapOf, iterables, var/val variables, and data class), and algorithmic concepts such as greedy, dynamic programming, and backtracking. These concepts were chosen because they represent realistic programming challenges that involve varying degrees of data flow complexity, and together they provide sufficient diversity to draw meaningful conclusions about LLM comprehension across different programming contexts. A few core programming concepts were identified and systematically combined to create pairwise interactions, thereby enhancing the conceptual diversity of the dataset. The dataset contains 22 concepts, including both singular concepts and pairwise combinations.

These 22 concepts were further divided into four levels of increasing data flow complexity. Four levels were chosen as they provided a sufficient granularity to observe meaningful differences in LLM performance across complexity without making the dataset unmanageably large. During initial development, levels were built incrementally on top of one another; however, this resulted in code snippets that were too long, causing the LLM to produce irregular responses, group lines together, or decline to answer. To address this, each level was redesigned to be independent, incorporating greater data flow depth through more deeply nested control structures or intricate interdependencies, without necessarily

increasing the number of lines of code. The order of increasing complexity in an ordinal format is,

$$[\text{Level } 1 < \text{Level } 2 < \text{Level } 3 < \text{Level } 4]$$

The levels are applied independently to each concept, meaning concepts cannot be compared in terms of complexity across concepts. For example, level 4 of the concept 'arrays' cannot be compared with level 4 of the concept 'trees'. Concept combinations were used to increase semantic and structural complexity, enabling us to examine how interactions among features affected model comprehension.

For example, for the concept "Sorting", These four snippets are based on the concept of Sorting. It explores various different existing sorting methods and asks the LLM to provide the result from mid way of the sorting. Level 1(Listing 3.1) is based on insertion sort. It is simple data flow, arr[i] is read into key, transformed through a shift operation back into arr, and rounds accumulates as a counter. There is a single, linear data dependency chain.

Listing 3.1: Sorting Level1 Snippet

```
fun sorting_array(arr: IntArray): IntArray {
    var rounds: Int = 0
    val half: Int = arr.size / 2
    for (i in 1 until arr.size) {
        val key: Int = arr[i]
        var j: Int = i - 1

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]
            j--
        }
        arr[j + 1] = key
        rounds++
        if (rounds >= half) {
            return arr
        }
    }
    return arr
}
```

Level 2 (Listing 3.2) introduces a branching data flow: the value key is first used to locate an insertion position via left/right/mid, and then that computed position feeds a separate shift loop. Data passes through two dependent transformation stages before being written back.

Listing 3.2: Sorting Level2 Snippet

```
fun sorting_array(arr: IntArray): IntArray {
    var rounds: Int = 0
    val half: Int = arr.size / 2
    for (i in 1 until arr.size) {
        val key: Int = arr[i]
```

```
        var left: Int = 0
        var right: Int = i

        while (left < right) {
            val mid: Int = (left + right) / 2
            if (key < arr[mid]) right = mid
            else left = mid + 1
        }

        for (j in i downTo left + 1) {
            arr[j] = arr[j - 1]
        }
        arr[left] = key

        rounds++
        if(rounds >= half ){
            return arr
        }
    }
    return arr
}
```

In Level 3 (Listing 3.3), the data flow is now gap-driven. The variable gap is computed externally and propagates its value into the inner loop as a read dependency, meaning the transformation of arr[j] depends not just on adjacent elements but on a dynamically changing stride value.

Listing 3.3: Sorting Level3 Snippet

```
fun sorting_array(arr: IntArray): IntArray {
    val n: Int = arr.size
    val half: Int = n / 2
    var rounds: Int = 0
    var gap = 1

    while (gap < n / 3) gap = 3 * gap + 1

    while (gap >= 1) {
        for (i in gap until n) {
            val key: Int = arr[i]
            var j: Int = i

            while (j >= gap && arr[j - gap] > key) {
                arr[j] = arr[j - gap]
                j -= gap
            }
            arr[j] = key

            rounds++
            if (rounds >= half) {
                return arr
            }
```

```
            }
            gap /= 3
        }
        return arr
}
```

Level 4 (Listing 3.4) is merge sort based snippet. This is the most complex data flow. The array is recursively partitioned into new data structures (left, right), each of which is independently transformed and then merged. Data flows across function boundaries and is reconstructed into a new structure (result), making it impossible to trace the transformation of a single element without following multiple call frames.

From these 4 levels of snippets, there is clear increase in data flow complexity of the code snippet. The snippet has a main data flow function/class, where the data flow complexity is clearly visible and a main calling function. All the concepts in this dataset follow this division of code. It may include additional supporting code if needed.

Listing 3.4: Sorting Level4 Snippet

```
fun optimized_sort(arr: IntArray, threshold: Int = 10): IntArray {
    if (arr.size <= 1) return arr
    if (arr.size <= threshold) {
        return sorting_array(arr)
    }

    val mid: Int = arr.size / 2
    val left: IntArray = arr.sliceArray(0 until mid)
    val right: IntArray = arr.sliceArray(mid until arr.size)

    return merge(optimized_sort(left, threshold), optimized_sort(right, threshold))
}

fun merge(left: IntArray, right: IntArray): IntArray {
    val result: MutableList<Int> = mutableListOf<Int>()
    var i: Int = 0
    var j: Int = 0

    while (i < left.size && j < right.size) {
        if (left[i] <= right[j]) {
            result.add(left[i])
            i++
        } else {
            result.add(right[j])
            j++
        }
    }

    while (i < left.size) {
        result.add(left[i])
        i++
    }
    while (j < right.size) {
```

```
        result.add(right[j])
        j++
    }
    return result.toIntArray()
}

fun sorting_array(arr: IntArray): IntArray {
    var rounds: Int = 0
    val half: Int = arr.size / 2
    for (i in 1 until arr.size) {
        val key: Int = arr[i]
        var j: Int = i - 1

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]
            j--
        }
        arr[j + 1] = key
        rounds++
        if (rounds >= half) {
            return arr
        }
    }
    return arr
}
```

Similarly, this increase in data flow complexity across the 4 levels of code snippets was applied to the remaining concepts.

3.2.2 Tasks Definition

For the def-use chain concepts, different authors proposed varying definitions over the years [1, 27]. Kennedy [27] described a variable as absolutely useful at a point **p** in the program if it was a parameter in a subroutine call at **p** or tested in a conditional branch at **p**. Allen and Cocke [1] described a use as an expression or part of an expression that referenced a data item without modifying it. These definitions, while foundational, introduced ambiguity when applied to modern code structures, particularly in the presence of functions, classes, literals, and built-in constructs.

To resolve this ambiguity, we tested variations in definitions for def, use, basic blocks, and def-use chains using LLM on Claude Sonnet 4. With the definitions from Kennedy [27], predefined functions were majorly misinterpreted as uses. With the definitions from Allen and Cocke [1], the use of the word "data" caused functions and classes to be missed in every snippet, and literals were misinterpreted as uses. We therefore adopted a more general approach by introducing the term "entity", where a def was when data was written to an entity in a program snippet, and a use was when an entity was read in that program snippet. This approach aligned with core data flow concepts while remaining flexible across different programming paradigms and control structures.

As long prompts can lead to decreased output quality in LLMs [46], the evaluation was structured into 5 distinct prompts for each prompting method to mitigate cognitive overload. Each task focused on a specific subset of program analysis objectives, enabling more targeted and manageable interactions with the LLM. Segmenting tasks in this manner prevented the model from being overwhelmed by too many interdependent tasks at once [39]. Although the tasks were presented in a sequential manner, from the fundamentals of data flow analysis to the final program output, the prompts were executed independently to avoid error propagation caused by the influence of previous tasks. Prior exploratory testing showed minimal differences between sequential and independent execution; however, independent execution was adopted to ensure robustness. The base prompts were defined as follows,

- **Prompt 1** targeted fundamental concepts, namely definitions (defs), usages (uses), and function calls (calls).
- **Prompt 2** focused on basic block identification, including block entry and exit points.
- **Prompt 3** addressed the construction of def-use chains.
- **Prompt 4** evaluated the ability of the model to predict program output.
- **Prompt 5** evaluated the ability of the model to perform backward program slicing.

3.2.3 Prompt Engineering

Prompt engineering plays a vital role in the results from the LLM. Small changes in the prompts can lead to major changes in the results. We observed this in our pilot study and during the thesis. To ensure the best results from the LLM, we tested various versions of the same definitions and the overall prompt before selecting the best.

Each prompt consisted of three main parts: the definitions of the relevant concepts, the task information, and the expected output format. The following examples illustrate the final zero-shot prompts for the previously defined set of tasks.

Prompt 1: Defs, Uses, Function calls

Entities are any variables, parameters, objects, or instances that can hold or store data.
Def - Data is written into an entity.
Use - Data is read into an entity.
For each line of code, based on the above definitions, identify the defs and uses of each entity and store them in the JSON format provided below. Along with the entities, also include any function calls that may take place at that line of code. Note that the given code is in the Kotlin programming language.
JSON format

Prompt 2: Basic Blocks

Basic blocks: A basic block is a linear sequence of lines of code having one entry point and one exit point.
Based on the definition, include each line of code in its basic block. Note that the given code is in the Kotlin programming language. Number and label each basic block, then indicate each code line under its corresponding block number. Put the final results in a JSON format:
JSON format

Prompt 3: Def Use Chains

Entities are any variables, parameters, objects, or instances that can hold or store data.
Def: Data is written into an entity.
Use: Data is read into an entity.
Def-use chains: A mapping from each def of an entity to all its subsequent uses that depend on that def.
Based on these definitions, identify the def-use chains of each entity in every line of code and put them in the format below. Note that the given code is in the Kotlin programming language. Ensure to include all the def-use chains for each entity in the JSON format below:
JSON format

Prompt 4: Program Output

Entities are any variables, parameters, objects, or instances that can hold or store data.
Calculate the output that will be produced when the function function is called. Do not describe the result and just provide the output.
The above tasks can be done without executing the code.

Prompt 5: Program Slice

Entities are any variables, parameters, objects, or instances that can hold or store data. A backward program slice shows all lines of code that contribute to the value of a specific entity at a certain point in the program.
Find the backward program slice for the entity named entity at the line code line inside the entity function function.
Include all lines of code that directly or indirectly influence how entity gets its value at this point. Trace dependencies backward through all relevant statements that define, update, or affect any entity used in this calculation. Exclude any code that does not impact the result of entity.
The above tasks can be done without executing the code. Provide your answer in the JSON format below:
JSON format

Similar prompt structures were applied for One-shot, CoT, and One-shot CoT prompting, the templates for which were introduced in the background chapter.

3.2.4 Prompting Technique

This study employs Zero-shot prompting, One-shot prompting, CoT prompting, and One-shot-CoT prompting to evaluate the performance of LLMs across various code comprehension and data flow analysis tasks.

During prior exploratory testing, the One-shot CoT prompting format yielded more precise responses than Zero-shot prompting. Earlier researchers have also shown that CoT prompting improves model performance, particularly on tasks requiring multi-step reasoning [30].

The decision to rely particularly on these prompting techniques rather than others is grounded in both practical and theoretical considerations. Schulhoff et al. [38] provides a systematic survey of prompt engineering. The authors demonstrate the effectiveness of the Few-shot CoT prompting technique relative to other techniques and observe that it provides the best results. However, Zero-shot prompting remains an essential benchmark for assessing the baseline capabilities of the model. At the same time, One-shot CoT strikes an effective balance by offering improved precision with minimal complexity and computational overhead.

The preliminary tests of this study also revealed minimal performance difference between One-shot and Two-shot prompting, further justifying the choice of the more straightforward and more efficient One-shot setup.

3.2.5 Large Language Model - Deepseek Coder 6.7B Instruct

To identify the best-performing LLM model on Kotlin code, we compared Qwen 2.5 Coder 7B Instruct [25], DeepSeek Coder 6.7B Instruct [17], and LLaMA 3.1 8B Instruct [15] using benchmark datasets such as HumanEval and MBPP, and by checking whether the models were trained with Kotlin data. Table 3.1 shows the performance of these models across the benchmarks and indicates whether they were trained in Kotlin.

	Qwen 2.5 Coder 7B Instruct [25]	**DeepSeek Coder 6.7B Instruct** [17]	**LLaMA 3.1 8B Instruct** [15]
HumanEval [10]	88.4%	66.1%	37.2%
MBPP [4]	83.5%	65.4%	47.6%
Kotlin training	Yes	Yes	No

Table 3.1: Comparison of the Large Language Models

HumanEval [10] and MBPP [4] are two famous code-based benchmark datasets. However, they are both Python-based datasets. The Qwen and Deepseek models perform quite well on these datasets, whereas LLaMA performs quite poorly. LLaMA is observed to be a general-purpose model trained to perform a wide range of tasks [15]. Hence, it is likely the reason for showing low precision with these benchmark datasets. Qwen 2.5 Coder 7B Instruct is reported to have good Kotlin performance on the McEval dataset [25]. The McEval dataset is a multilingual programming benchmark covering 40 languages (16K samples in total) that includes multilingual code generation, code explanation, and code completion tasks [7]. However, in the results Chai et al. [7] for Kotlin code, CodeQwen achieved 44% accuracy, while DeekSeek Coder Instruct achieved 66% accuracy.

DeepSeek Coder is trained on 3,121 Kotlin files (0.75% of total code corpus) [17]. In the research papers that have surveyed LLMs on such benchmark datasets, Deepseek Coder 6.7B Instruct performed quite well overall [9, 54]. The paper Zheng et al. [54] introduced DomainCodeBench, a benchmark dataset that includes Kotlin code. DeepSeek Coder 33B Instruct is observed to deliver the best results among other LLM models, while DeepSeek Coder 6.7B Instruct delivers competitive results. With clear information about DeekSeek Coder 6.7B Instruct training and performance on benchmark datasets that also include Kotlin data, it is a clear choice for LLM in this thesis. Additionally, Guo et al. [17] noted that the training data used filtered out files containing benchmark datasets such as HumanEval and MBPP, thereby making its results on these benchmarks more reliable.

3.2.6 Evaluation Pipeline

Iterative Refinement of Prompts In the initial design, each complexity level was built on top of the previous one, preserving prior complexity while introducing new elements. However, this resulted in code snippets that were too long, causing the LLM to produce irregular responses. Commonly observed issues included the LLM adding the line "continue the same for the remaining lines" despite explicit instructions, grouping lines of code together in its response, cutting responses short, or stating that the task was too complex to complete. To address these issues, the code snippets were significantly shortened and each level was redesigned independently. Additionally, the temperature was initially set to 0 to produce deterministic output, but this did not resolve the irregularities. The LLM was therefore prompted 5 times per snippet per prompting type, with temperature set to 0.05 and top_p set to 0.8, allowing for deterministic output with small variations.

Pass@5 Before assessing precision, we first evaluated the ability of the LLM to provide at least one response containing at least one correct element across the 5 runs. A run was considered a pass if the evaluated result contained Precision > 0, Recall > 0, or Jaccard Index > 0, and a failure otherwise. Missing data were filled in first, after which each unique combination of concept, level, and prompting type was assigned a value of 100 in case of a pass and 0 in case of a failure.

Evaluation Metrics The responses from the LLM were cleaned and the JavaScript Object Notation (JSON) outputs were extracted and compared against the ground truth to obtain Precision, Recall, and Jaccard Index. Precision and Recall quantify false positives and false negatives respectively, while the Jaccard Index measures the overall similarity between the predicted results and the ground truth.

Selection of Best Round Since each snippet was prompted 5 times, the round with the highest Jaccard Index was selected as the representative result for further analysis. The Jaccard Index was used as the selection criterion because it accounts for both the completeness of the ground truth in the LLM output and any additionally predicted elements, which Precision and Recall individually fail to capture. Precision does not check whether the entirety of the ground truth is present in the answer, while Recall does not account for

additionally predicted data. The Jaccard Index therefore provided the most comprehensive basis for selecting the best round.

Code Complexity Metrics Code complexity metrics were computed to capture different aspects of difficulty in a program. Cyclomatic complexity measured the number of independent paths through the code, where more paths indicated higher complexity. LOC measured the number of executable lines of code. These metrics were used in RQ4, RQ5, and RQ6 to examine how complexity related to LLM performance. Rather than treating complexity as a categorical variable aligned with the defined levels, complexity was treated as a continuous variable across the full dataset, as the number of samples per level was limited and the levels did not consistently align with computed complexity values.

Statistical tests To assess the distribution of the data, the Shapiro-Wilk test was applied to each evaluation metric separately for each prompting type and level. The null hypothesis stated that the data were normally distributed. In all cases, the results were statistically significant and the data were found to be not normally distributed, leading to the rejection of the null hypothesis. For RQ1, RQ2, and RQ3 focus on the patterns that emerge from the results of Friedmann's test and the Wilcoxon test to find the significant pairs. We analyze each task in depth using the Friedman test per prompting type, per level. Further, we utilize ART ANOVA for the combined effect of Prompting type and Level. For RQ4, RQ5, and RQ6, Spearman's Rank Correlation Coefficient is used with False discovery rate correction (using Benjamini-Hochberg). A p-value below 0.05 was considered statistically significant, and a p-value below 0.001 was considered highly significant.

4

Results

This chapter presents the results for the research questions described in the Methodology chapter. We first examine the ability of the LLM to provide a positive answer, as measured by the Pass@5 metric, then investigate each research question in detail.

4.1 Data Cleaning

The LLM generated outputs in JSON format; however, these outputs often contained additional explanatory text or formatting inconsistencies. In several cases, the JSON structure was incomplete or syntactically incorrect, for example, due to missing commas, misplaced brackets, or inconsistent quotation marks.

To ensure reliable evaluation, we applied a data-cleaning step that removed extraneous text and corrected minor formatting errors so that all outputs conformed to valid JSON syntax. Outputs that could not be automatically repaired were manually inspected or excluded from the analysis. This preprocessing step ensured consistent and accurate extraction of the required fields for subsequent evaluation.

4.2 Research Question 0

A total of 88 snippets were finalized. The results from the LLM on each of these snippets across the 9 tasks are cleaned and tabulated for each result metric. The subsections discuss the descriptive statistics of the results and the Pass@5 metric.

In total, there are 88x4 unique results finalized for each task, level, and prompting type. This result is obtained for each of the 9 tasks. These results were obtained from performing each prompt 5 times. Some prompts were combinations of tasks. So, for a single concept with 4 levels of code snippets, run 4 different prompting strategies for 5 prompts, 5 times. leading to 400 result files for each concept and in total 8,800 output files, from which the best results are chosen for each task are derived. For a lot of the output files, there were minor issues with the JSON format. In some cases, there was more than one JSON content that needed to be combined or chosen. For One-Shot and One-Shot CoT, we observed that the LLM tends to repeat back the answer for the example snippet. In those cases, we choose the second JSON file that was focused on the task. For all the tasks, the data ranges from 0 to 1 for all three metrics. A detailed description of the data distribution is present in Appendix a.

4.2.1 Pass@5

Pass@5 assesses the ability of the LLM to provide at least a low-positive answer for a task and a given snippet. The results were observed separately by prompting type and level.

Tasks	CoT	One-shot	One-shot CoT	Zero-shot
defs	60.227	65.909	64.773	59.091
uses	59.091	73.864	73.864	55.682
Function calls	22.727	36.364	29.545	25.000
blocks	79.545	86.364	88.636	78.409
entry	94.318	100.000	97.727	94.318
exit	100.000	100.000	97.727	98.864
Def-use chains	25.000	28.409	34.091	23.864
program output	68.182	73.864	72.727	70.455
program slice	97.727	97.727	95.455	95.455

Table 4.1: Average Pass@5 for All Categories of Tasks per Prompting Type

Each prompting technique was measured separately and listed in Table 4.1. This table shows the average percentage of tasks per prompting type that the LLM passed in Pass@5 metric across the entire dataset. We can see that the LLM fails extensively to provide any positive answer for tasks like Function calls and Def-use chains across all prompting types. The best average Pass percentage is observed for Program slice, Block entry, and Block exit.

The pass percentage at zero-shot on almost all the tasks is observed to be lower than that of the other prompting types. More specifically, in line with the literature, we observe a clear difference in the number of tasks answered between One-shot CoT and Zero-shot prompting, with One-shot CoT consistently higher.

Tasks	Level 1	Level 2	Level 3	Level 4
defs	64.77	65.91	60.23	59.09
uses	72.73	68.18	64.77	56.82
calls	34.09	35.23	23.86	20.46
blocks	90.91	79.55	84.09	78.41
entry	98.86	95.45	100.00	92.05
exit	100.00	100.00	100.00	96.59
chains	23.86	31.82	32.95	22.73
program output	76.14	70.45	70.45	68.18
program slice	97.73	98.86	97.73	92.05

Table 4.2: Average Pass@5 for All Categories of Tasks per Level

When observing the pass@5 metric level wise, there is an overall reduction of pass percentage from Level 1 to Level 4 for all the tasks in Table 4.2. However, in some cases, the intermediary levels have a better pass percentage than the level before. For example, for the task of defs identification, Level 2 has a pass percentage of 65.91%, and Level 1 has a pass percentage of 64.77%, resulting in an over 1% increase.

4.3 Research question 1

This section aims to answer the research question: *How precisely can the LLM identify all the defs, uses, function calls, and basic blocks with snippets with increasing complexity of data flow across various prompting methods?*. It focuses on the base tasks, namely identification of defs, uses, function calls, basic blocks, block entry, and block exit.

4.3.1 Defs

For the task of identification of defs, we observed the data to be highly significantly non-normal data for all three metrics ($p_{\text{Precision}} = 3.632\text{e}{-17}$, $p_{\text{Recall}} = 2.824\text{e}{-17}$, $p_{\text{Jaccard}} = 1.095\text{e}{-18}$) from the Shapiro-Wilk test. Hence, we reject the null hypothesis for the three metrics.

Table Table 4.3 presents the results of the Friedman test for detecting effects of prompting type and difficulty level, together with post-hoc pairwise Wilcoxon signed-rank tests (Bonferroni-corrected) for the three metrics.

Per Prompting Type For Precision, the Friedman test revealed a statistically significant effect of prompting type ($\chi^2 = 12.80$, $p = 5.100\text{e}{-03}$). Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified one significant pair: One-shot CoT significantly outperformed CoT ($W = 385$, $p = 1.200\text{e}{-02}$, $r = 0.419$). This shows that having One-shot method along with CoT makes the results more reliable. The remaining prompting type pairs did not differ significantly.

Per Level For the Jaccard Index, the Friedman test revealed a statistically significant effect of level ($p = 1.580\text{e}{-04}$). Post-hoc analysis using Wilcoxon signed-rank tests with Bonferroni correction identified two significant pairs. Level 1 with Level 2 ($W = 489.0$, $p = 2.726\text{e}{-02}$, $r = 0.373$). Level 1 with Level 3 ($W = 435.5$, $p = 8.924\text{e}{-04}$, $r = 0.482$). The remaining pairs did not differ significantly.

For Precision, the Friedman test revealed a statistically significant effect of level ($\chi^2 = 15.83$, $p = 1.229\text{e}{-03}$). Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified two significant pairs. Level 1 with Level 3 ($W = 473.5$, $p = 1.136\text{e}{-02}$, $r = 0.404$). Level 1 with Level 4 ($W = 551.5$, $p = 4.469\text{e}{-02}$, $r = 0.345$). The remaining pairs did not differ significantly.

For Recall, the Friedman test revealed a statistically significant effect of level ($\chi^2 = 22.63$, $p = 4.816\text{e}{-05}$). Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified two significant pairs. Level 1 with Level 3 ($W = 379.5$, $p = 8.136\text{e}{-04}$, $r = 0.497$). Level 1

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	7.04	7.050e−02	–
Recall	7.58	5.570e−02	–
Precision	12.80	5.100e−03	*CoT* vs *One-shot CoT* ($W = 385$, $p = 1.200\text{e}{-02}$, $r = 0.419$)
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	20.15	2.000e−04	L1 vs L2 ($W = 489.0$, $p = 2.700\text{e}{-02}$, $r = 0.373$)
			L1 vs L3 ($W = 435.5$, $p = 1.000\text{e}{-03}$, $r = 0.482$)
Recall	22.63	4.816e−05	L1 vs L3 ($W = 379.5$, $p = 8.136\text{e}{-04}$, $r = 0.497$)
			L1 vs L4 ($W = 364.5$, $p = 1.869\text{e}{-02}$, $r = 0.410$)
Precision	15.83	1.200e−03	L1 vs L3 ($W = 473.5$, $p = 1.136\text{e}{-02}$, $r = 0.404$)
			L1 vs L4 ($W = 551.5$, $p = 4.469\text{e}{-02}$, $r = 0.345$)

Table 4.3: Friedman Tests per Prompting Type and Metric for the Defs Dataset.

with Level 4 ($W = 364.5$, $p = 1.869\text{e}{-02}$, $r = 0.410$). The remaining pairs did not differ significantly.

Prompting Type and Level Now, to observe the interaction between prompting type and level, we perform an ART ANOVA. The results from this test are presented in Table a.11

Data Visualization To visualize the data for each metric, we display box plots in Figure 4.1. These plots show each metric across the prompting types and levels. From these plots, we can notice that the medial is always greater than zero. The metric Jaccard Index shows more outliers than the other plots. There is no consistent visible difference in results between the prompting types and levels, as proved by our statistical tests.

From the box plots, we know Level 1 performs better than Level 3 and Level 4 in all three evaluation metrics.

Summary In conclusion, the Friedman test on the prompting types showed a significant difference between CoT prompting and One-shot CoT prompting for defs identification, as measured by Precision. For a level-wise analysis, we observe that Level 1 shows a highly significant difference across the three metrics. Level 1 and Level 3 show consistent differences across all three metrics, and the box plots show that the median of Level 3 is always lower than that of Level 1. However, we do not observe a consistent reduction across all the levels. The combination of prompting type and levels does not show any statistical significance.

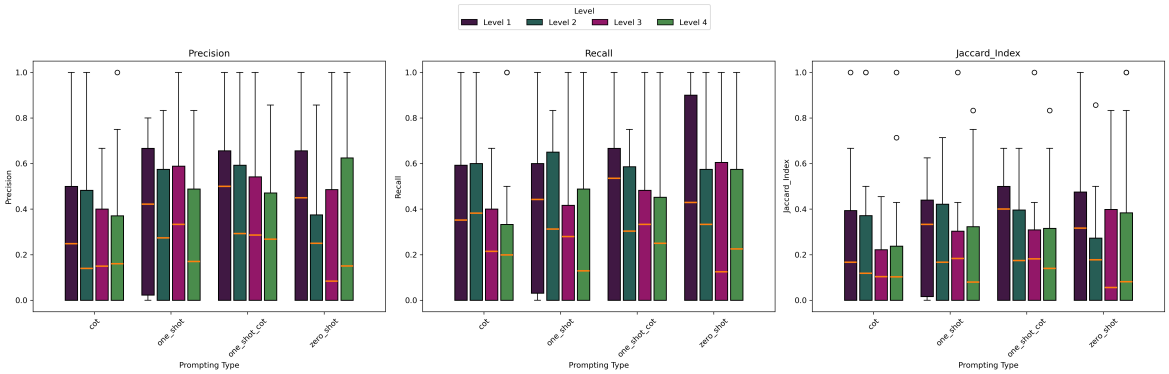

Figure 4.1: Box Plot for Defs Identification Data for All Three Metrics

4.3.2 Uses

Similar to the task of def identification, we observe the normality test results obtained for the three metrics ($p_{\text{Precision}} = 1.981\text{e}{-19}$, $p_{\text{Recall}} = 1.521\text{e}{-16}$, $p_{\text{Jaccard}} = 0.7612\text{e}{-22}$) to be highly significant.

Per Prompting Type We proceed with the Friedman test and find that a statistically significant difference exists in the data for Jaccard Index ($p = 3.390\text{e}02$) and Precision ($p = 1.900\text{e}02$). The Table 4.4 shows the results from these tests. Upon further inspection, we do not find any significant pairs for the Jaccard Index and Precision.

Per Level The Friedman test revealed statistically significant results across all three metrics: Jaccard Index ($\chi^2 = 17.45$, $p = 5.700\text{e}{-04}$), Precision ($\chi^2 = 15.58$, $p = 1.379\text{e}{-03}$), and Recall ($\chi^2 = 22.98$, $p = 4.071\text{e}{-05}$), as shown in Table 4.4. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified two significant pairs for the Jaccard Index and Precision: Level 1 with Level 3 and Level 1 with Level 4. For Recall, four significant pairs were found: Level 1 with Level 3, Level 1 with Level 4, Level 2 with Level 3, and Level 2 with Level 4. All effect sizes were medium (r ranging from 0.351 to 0.514). The remaining pairs did not differ significantly.

Prompting Type and Level The combination of these two independent variables did not show any significant difference. The statistical results obtained for each metric are presented in Table a.12.

Data Visualization The plot Figure 4.2 shows the distribution of each metric for each prompting type and level, using a Box plot, based on data from the task Uses identification. We can observe consistent differences in levels across all prompting types and metrics, as evidenced by the statistical tests. The median is always greater than zero, and a few outliers with values up to 1.0 are observed across all metrics.

Summary In Summary, there exists some statistical significance between the prompting types for the Jaccard Index Metric and Precision. However, it can not be observed for which

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	8.68	3.390e−02	–
Precision	9.95	1.900e−02	–
Recall	5.28	1.526e−01	–
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	17.45	5.700e−04	L1 vs L3 ($W = 734.5$, $p = 1.107\text{e}{-02}$, $r = 0.370$)
			L1 vs L4 ($W = 466.0$, $p = 1.573\text{e}{-04}$, $r = 0.514$)
Precision	15.58	1.379e−03	L1 vs L3 ($W = 699.5$, $p = 2.288\text{e}{-02}$, $r = 0.351$)
			L1 vs L4 ($W = 448.0$, $p = 2.689\text{e}{-04}$, $r = 0.506$)
Recall	22.98	4.071e−05	L1 vs L3 ($W = 529.5$, $p = 5.053\text{e}{-04}$, $r = 0.477$)
			L1 vs L4 ($W = 467.0$, $p = 2.714\text{e}{-04}$, $r = 0.502$)
			L2 vs L3 ($W = 572.0$, $p = 1.701\text{e}{-02}$, $r = 0.376$)
			L2 vs L4 ($W = 485.5$, $p = 4.042\text{e}{-02}$, $r = 0.359$)

Table 4.4: Friedman Tests per Prompting Type and Metric for the Uses Dataset.

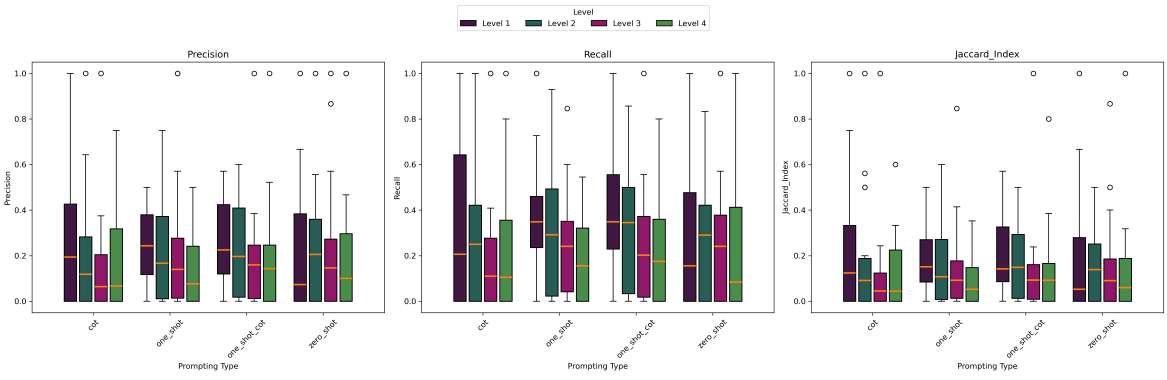

Figure 4.2: Box plot for Uses Identification Data for All Three Metrics

specific pairs of prompting types the significance is present. We again notice a significant difference between Level 1 and Level 3 as we did with task defs. Additionally, we also notice some more significant pairs. This shows that changes in data flow complexity in levels of snippets significantly affect the results of the LLM across all three metrics. More specifically, for Level 1, with Levels 3 and 4. However, the combination of prompting types and Levels did not yield a significant relationship.

4.3.3 Function Calls

For the task of Function calls, Shapiro-Wilk test for all three metrics ($p_{\text{Precision}} = 7.300\text{e}{-29}$, $p_{\text{Recall}} = 2.330\text{e}{-28}$, $p_{\text{Jaccard}} = 5.050\text{e}{-30}$) showed highly significant results.

The results per prompting type and level are present in table Table 4.5.

Per Prompting Type The Friedman test revealed no statistically significant effect of prompting type on any of the three metrics: Jaccard Index ($\chi^2 = 6.50$, $p = 8.960\text{e}{-02}$), Precision ($\chi^2 = 6.13$, $p = 1.056\text{e}{-01}$), and Recall ($\chi^2 = 5.51$, $p = 1.383\text{e}{-01}$). Hence, no post-hoc tests were conducted.

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	6.50	8.960e−02	–
Precision	6.13	1.056e−01	–
Recall	5.51	1.383e−01	–
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	11.70	8.485e−03	–
Precision	7.69	5.286e−02	–
Recall	20.51	1.328e−04	L1 vs L3 ($W = 114.5$, $p = 1.785\text{e}{-02}$, $r = 0.516$)
			L1 vs L4 ($W = 156.5$, $p = 1.920\text{e}{-02}$, $r = 0.484$)
			L2 vs L3 ($W = 78.0$, $p = 1.469\text{e}{-02}$, $r = 0.560$)
			L2 vs L4 ($W = 162.0$, $p = 1.415\text{e}{-02}$, $r = 0.491$)

Table 4.5: Friedman Tests per Prompting Type and Metric for the Function Calls Dataset.

Per Level The Friedman test revealed statistically significant results for Jaccard Index ($\chi^2 = 11.70$, $p = 8.485\text{e}{-03}$) and Recall ($\chi^2 = 20.51$, $p = 1.328\text{e}{-04}$), but not for Precision

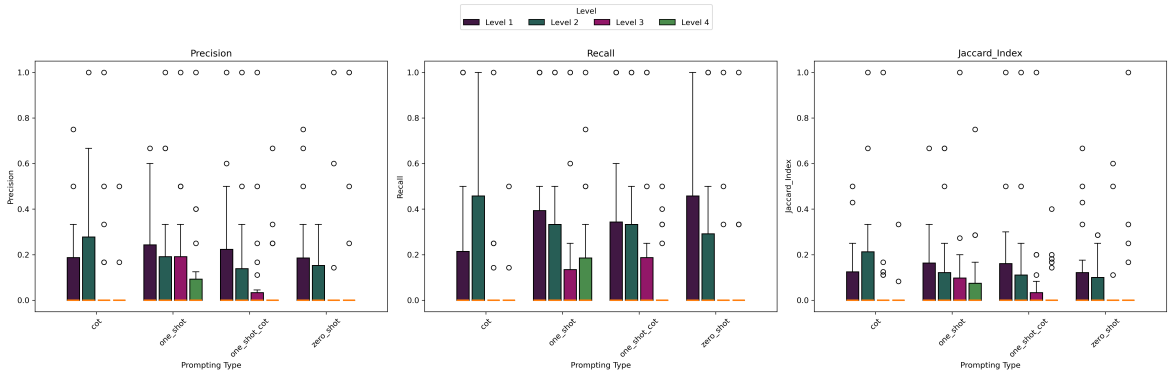

Figure 4.3: Box Plot for Function Calls Identification Data of All Three Metrics

($\chi^2 = 7.69$, $p = 5.286\text{e}{-02}$), as shown in Table 4.5. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) for Jaccard Index yielded no significant pairs despite the significant Friedman result. For Recall, four significant pairs were identified: Level 1 with Level 3 ($W = 114.5$, $p = 1.785\text{e}{-02}$, $r = 0.516$), Level 1 with Level 4 ($W = 156.5$, $p = 1.920\text{e}{-02}$, $r = 0.484$), Level 2 with Level 3 ($W = 78.0$, $p = 1.469\text{e}{-02}$, $r = 0.560$), and Level 2 with Level 4 ($W = 162.0$, $p = 1.415\text{e}{-02}$, $r = 0.491$). All effect sizes were medium to large (r ranging from 0.484 to 0.560). The remaining pairs did not differ significantly.

Prompting Type and Level We do not notice any statistically significant difference in the combination of prompting type and level. The statistical results for each metric are shown in Table a.13.

Data Visualization The Figure 4.3 shows the distribution of the function calls data of all three evaluation metrics. The median is 0. As data flow complexity increases, we observe a reduction in results across each metric from Level 1 to Level 4.

Summary In summary, we do not observe statistical significance for any prompting type or for the combination of prompting types and levels. However, we observe a significant difference across the multiple levels of the recall metric. It showed a significant difference between the lower levels (Level 1 and Level 2) and the higher levels of snippets (Level 3 and Level 4).

4.3.4 Basic Blocks

For the task of basic blocks, Shapiro-Wilk test on data for all three metrics($p_{\text{Precision}} = 4.465\text{e}{-09}$, $p_{\text{Recall}} = 1.064\text{e}{-17}$, $p_{\text{Jaccard}} = 2.71\text{e}{-20}$) showed highly significance.

The Table 4.6 shows the significant pairs for each prompting type and level. Additionally, for significant metrics, we observe significant pairs with Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected).

Per Prompting Type The Friedman test revealed highly significant effects of prompting type for Jaccard Index ($\chi^2 = 35.21$, $p < 1.000\text{e}{-04}$) and Recall ($\chi^2 = 35.12$, $p < 1.000\text{e}{-04}$),

and a significant effect for Precision ($\chi^2 = 18.04$, $p = 4.000\text{e}{-04}$), as shown in Table 4.6. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified four significant pairs for Jaccard Index and Recall, and two for Precision. One-shot with Zero-shot was significant across all three metrics (Jaccard Index: $W = 531.5$, $p = 1.427\text{e}{-05}$, $r = 0.545$; Precision: $W = 762.5$, $p = 1.179\text{e}{-02}$, $r = 0.365$; Recall: $W = 354.0$, $p = 2.686\text{e}{-05}$, $r = 0.573$). One-shot CoT with Zero-shot was also significant across all three metrics (Jaccard Index: $W = 715.0$, $p = 2.357\text{e}{-04}$, $r = 0.466$; Precision: $W = 795.5$, $p = 1.366\text{e}{-02}$, $r = 0.357$; Recall: $W = 367.0$, $p = 1.947\text{e}{-04}$, $r = 0.532$). For Jaccard Index and Recall, CoT with One-shot and CoT with One-shot CoT were additionally significant. The remaining pairs did not differ significantly. This shows that having One-shot approach with or without CoT almost consistently shows a significant difference with other prompting types.

Friedman Test – Prompting Type

Metric	χ^2	p	Significant pairs
Jaccard Index	35.21	$< 1.000\text{e}{-04}$	CoT vs One-shot ($W = 565.0$, $p = 4.406\text{e}{-04}$, $r = 0.474$)
			CoT vs One-shot CoT ($W = 812.0$, $p = 1.160\text{e}{-02}$, $r = 0.360$)
			One-shot vs Zero-shot ($W = 531.5$, $p = 1.427\text{e}{-05}$, $r = 0.545$)
			One-shot CoT vs Zero-shot ($W = 715.0$, $p = 2.357\text{e}{-04}$, $r = 0.466$)
Precision	18.04	$4.000\text{e}{-04}$	One-shot vs Zero-shot ($W = 762.5$, $p = 1.179\text{e}{-02}$, $r = 0.365$)
			One-shot CoT vs Zero-shot ($W = 795.5$, $p = 1.366\text{e}{-02}$, $r = 0.357$)
Recall	35.12	$< 1.000\text{e}{-04}$	CoT vs One-shot ($W = 303.0$, $p = 3.233\text{e}{-04}$, $r = 0.540$)
			CoT vs One-shot CoT ($W = 385.0$, $p = 2.709\text{e}{-03}$, $r = 0.465$)
			One-shot vs Zero-shot ($W = 354.0$, $p = 2.686\text{e}{-05}$, $r = 0.573$)
			One-shot CoT vs Zero-shot ($W = 367.0$, $p = 1.947\text{e}{-04}$, $r = 0.532$)

Friedman Test – Level

Metric	χ^2	p	Significant pairs
Jaccard Index	10.32	$1.601\text{e}{-02}$	L1 vs L4 ($W = 876.0$, $p = 4.961\text{e}{-04}$, $r = 0.432$)
Precision	7.53	$5.683\text{e}{-02}$	–
Recall	11.67	$8.588\text{e}{-03}$	L1 vs L3 ($W = 1036.0$, $p = 3.054\text{e}{-02}$, $r = 0.313$)
			L1 vs L4 ($W = 889.5$, $p = 1.044\text{e}{-03}$, $r = 0.415$)

Table 4.6: Friedman Tests per Prompting Type and Metric for the Basic Blocks Dataset.

Per Level The Friedman test revealed statistically significant results for Jaccard Index ($\chi^2 = 10.32$, $p = 1.601\text{e}{-02}$) and Recall ($\chi^2 = 11.67$, $p = 8.588\text{e}{-03}$), but not for Precision

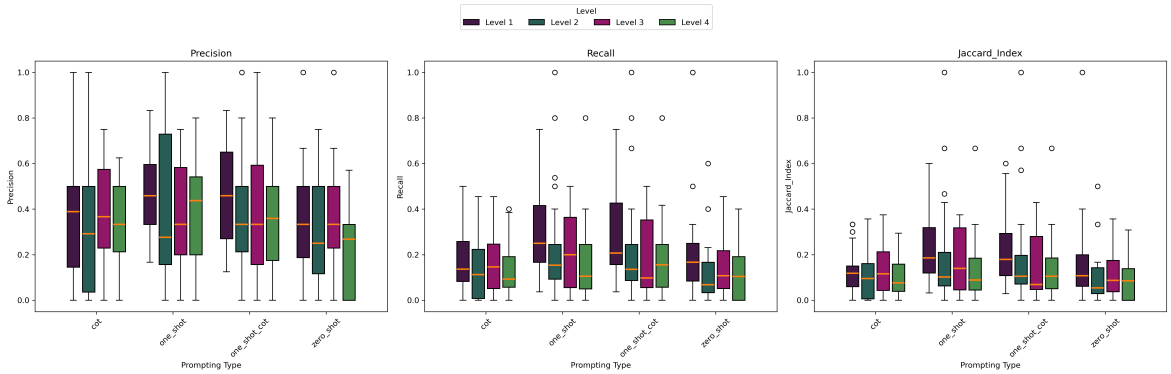

Figure 4.4: Box Plot for Basic Blocks Identification Data of All Three Metrics

($\chi^2 = 7.53$, $p = 5.683\text{e}{-02}$), as shown in Table 4.6. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified one significant pair for Jaccard Index: Level 1 with Level 4 ($W = 876.0$, $p = 4.961\text{e}{-04}$, $r = 0.432$). For Recall, two significant pairs were found: Level 1 with Level 3 ($W = 1036.0$, $p = 3.054\text{e}{-02}$, $r = 0.313$) and Level 1 with Level 4 ($W = 889.5$, $p = 1.044\text{e}{-03}$, $r = 0.415$). The remaining pairs did not differ significantly.

Prompting Type and Level The combination of these two independent variables did not show any significant difference. The statistical results obtained for each metric are presented in Table a.14. However, statistical differences for prompting types for all three metrics can also be observed here. Hence, confirming the previous Friedman test done per prompting type.

Data Visualization The Figure 4.4 shows the distribution of results across metrics, levels, and prompting types for basic blocks. The median is observed to be higher than zero in all cases. For precision, we observe only a few outliers. For recall and Jaccard Index, we observe many outliers.

Summary In summary, we observe statistical significance within prompting types and across levels individually. However, the combination of prompting type and levels does not show any statistical significance. The data distribution shows a median greater than zero. For prompting types, we observe a significant difference in one of the prompting types used with One-Shot prompting. For Levels, we mainly notice Level 1 with the higher levels. However, it is not observed to be consistent across the evaluation metrics.

4.3.5 Block Entry

For the task of basic blocks entry, Shapiro-Wilk test on data for all three metrics($p_{\text{Precision}} = 2.864\text{e}{-21}$, $p_{\text{Recall}} = 2.2\text{e}{-07}$, $p_{\text{Jaccard}} = 2.001\text{e}{-08}$) showed highly significance.

The Table 4.7 shows the significant pairs for each prompting type and level. Additionally, for significant metrics, we observe significant pairs with Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected).

Per Prompting Type The Friedman test revealed highly significant effects of prompting type for Jaccard Index ($\chi^2 = 42.67$, $p < 1.000e{-}04$) and Recall ($\chi^2 = 53.99$, $p < 1.000e{-}04$), but no significant effect for Precision ($\chi^2 = 3.27$, $p = 3.519e{-}01$), as shown in Table 4.7. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified four significant pairs for both Jaccard Index and Recall. One-shot with Zero-shot was the most significant pair across both metrics (Jaccard Index: $W = 419.0$, $p = 5.035e{-}08$, $r = 0.644$; Recall: $W = 238.5$, $p = 6.754e{-}08$, $r = 0.692$). CoT with One-shot was also highly significant for both metrics (Jaccard Index: $W = 373.5$, $p = 2.813e{-}07$, $r = 0.635$; Recall: $W = 310.0$, $p = 1.339e{-}06$, $r = 0.633$). CoT with One-shot CoT and One-shot CoT with Zero-shot were additionally significant for both metrics. The remaining pairs did not differ significantly.

Friedman Test – Prompting Type

Metric	χ^2	p	Significant pairs
Jaccard Index	42.67	$< 1.000e{-}04$	CoT vs One-shot ($W = 373.5$, $p = 2.813e{-}07$, $r = 0.635$)
			CoT vs One-shot CoT ($W = 802.0$, $p = 1.408e{-}03$, $r = 0.416$)
			One-shot vs Zero-shot ($W = 419.0$, $p = 5.035e{-}08$, $r = 0.644$)
			One-shot CoT vs Zero-shot ($W = 856.5$, $p = 5.620e{-}04$, $r = 0.431$)
Precision	3.27	$3.519e{-}01$	–
Recall	53.99	$< 1.000e{-}04$	CoT vs One-shot ($W = 310.0$, $p = 1.339e{-}06$, $r = 0.633$)
			CoT vs One-shot CoT ($W = 477.5$, $p = 1.283e{-}04$, $r = 0.515$)
			One-shot vs Zero-shot ($W = 238.5$, $p = 6.754e{-}08$, $r = 0.692$)
			One-shot CoT vs Zero-shot ($W = 546.0$, $p = 5.842e{-}05$, $r = 0.518$)

Friedman Test – Level

Metric	χ^2	p	Significant pairs
Jaccard Index	12.89	$4.878e{-}03$	L1 vs L2 ($W = 990.0$, $p = 6.026e{-}03$, $r = 0.363$)
			L1 vs L3 ($W = 1208.0$, $p = 1.082e{-}02$, $r = 0.333$)
			L1 vs L4 ($W = 1112.0$, $p = 4.124e{-}03$, $r = 0.364$)
Precision	3.36	$3.400e{-}01$	–
Recall	12.33	$6.337e{-}03$	L1 vs L2 ($W = 958.0$, $p = 5.644e{-}03$, $r = 0.368$)
			L1 vs L3 ($W = 1132.5$, $p = 3.557e{-}03$, $r = 0.366$)
			L1 vs L4 ($W = 1126.5$, $p = 5.149e{-}03$, $r = 0.357$)

Table 4.7: Friedman Tests per Prompting Type and Metric for the Basic Block Entry Dataset.

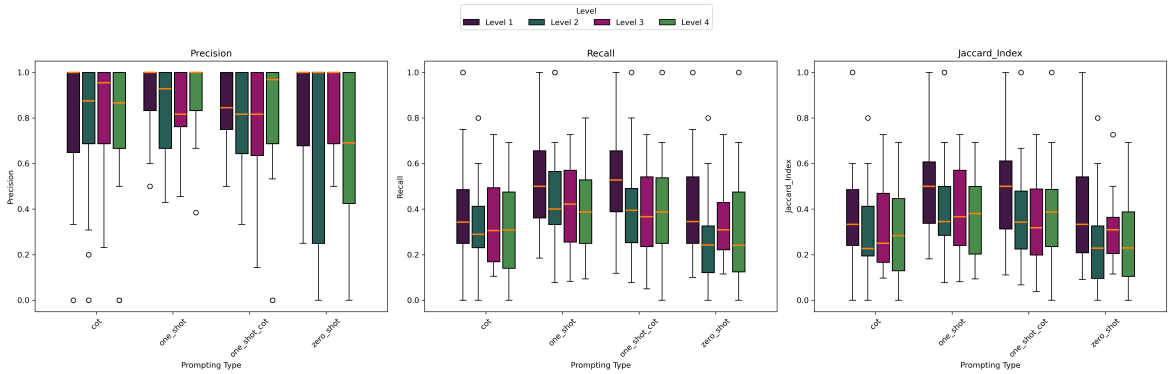

Figure 4.5: Box Plot for Basic Block Entry Identification Data of All Three Metrics

Per Level The Friedman test revealed statistically significant results for Jaccard Index ($\chi^2 = 12.89$, $p = 4.878\text{e}{-03}$) and Recall ($\chi^2 = 12.33$, $p = 6.337\text{e}{-03}$), but not for Precision ($\chi^2 = 3.36$, $p = 3.400\text{e}{-01}$), as shown in Table 4.7. Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified three significant pairs for both Jaccard Index and Recall metrics: Level 1 vs Level 2, Level 1 vs Level 3, and Level 1 vs Level 4. All effect sizes were medium (r ranging from 0.333 to 0.368). The remaining pairs did not differ significantly.

Prompting Type and Level The Table a.15 shows the results from the ART ANOVA Test. We do not notice any statistical difference in the combination of prompting types and level. However, the test confirms the statistically significant differences in the Jaccard Index and Recall metrics.

Data Visualization The Figure 4.5 shows the distribution of the data per metric, level, and prompting type. The median is observed to be higher than zero in all cases. We notice similar patterns in the Jaccard Index and Recall metrics. The majority of the data in the precision plot is observed to be closer to the value 1 than the value 0. Indicating that the LLM has fewer FP hallucinations.

Summary In Summary, prompting type and the level of data flow complexity individually affect the results of the LLM. However, the combination of these two independent variables does not show any statistical significance. Addition of one-shot based prompting shows a more significant difference with other prompting types without one-shot. Furthermore, Level 1 is observed to show a significant difference with all other levels. Both prompting and level-based resulting pairs are observed for the metric Jaccard Index and Recall.

4.3.6 Block Exit

For the task of basic blocks exit, Shapiro-Wilk test on data for all three metrics($p_{\text{Precision}} = 2.525\text{e}{-20}$, $p_{\text{Recall}} = 2.030\text{e}{-07}$, $p_{\text{Jaccard}} = 3.576\text{e}{-08}$) showed highly significance.

The Table 4.8 shows the significant pairs for each prompting type and level. Additionally, for significant metrics, we observe significant pairs with Post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected).

Per Prompting Type Table 4.8 shows a significant difference in the Jaccard Index and Recall metrics. Upon further inspection, we find that 4 of 6 prompting types are statistically highly significant. The significant pairs are observed with one-shot based prompting for metrics Jaccard Index and Recall. This indicates that the addition of one-shot creates a significant difference from other prompting types.

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	32.52	$<$ 1.000e−04	CoT vs One-shot ($W = 561.0$, $p = 2.391\text{e}{-04}$, $r = 0.488$)
			CoT vs One-shot CoT ($W = 974.5$, $p = 2.889\text{e}{-02}$, $r = 0.319$)
			One-shot vs Zero-shot ($W = 496.0$, $p = 4.462\text{e}{-05}$, $r = 0.532$)
			One-shot CoT vs Zero-shot ($W = 906.0$, $p = 5.927\text{e}{-03}$, $r = 0.371$)
Precision	1.08	7.819e−01	–
Recall	46.22	$<$ 1.000e−04	CoT vs One-shot ($W = 195.5$, $p = 5.327\text{e}{-06}$, $r = 0.657$)
			CoT vs One-shot CoT ($W = 404.5$, $p = 2.877\text{e}{-03}$, $r = 0.458$)
			One-shot vs Zero-shot ($W = 253.0$, $p = 6.564\text{e}{-06}$, $r = 0.629$)
			One-shot CoT vs Zero-shot ($W = 540.0$, $p = 6.575\text{e}{-04}$, $r = 0.469$)
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	9.09	2.807e−02	L1 vs L4 ($W = 1198.0$, $p = 1.465\text{e}{-02}$, $r = 0.325$)
Precision	1.46	6.921e−01	–
Recall	12.49	5.886e−03	L1 vs L3 ($W = 1233.0$, $p = 2.368\text{e}{-02}$, $r = 0.309$)
			L1 vs L4 ($W = 1107.5$, $p = 3.846\text{e}{-03}$, $r = 0.366$)

Table 4.8: Friedman Tests per Prompting Type and Metric for the Basic Block Exit Dataset.

Per Level As with the prompting type, we observe significant differences in the Jaccard Index and Recall metrics. Both have Level 1 with Level 4 as a significant difference in pairs. Recall also has Level 1 with Level 3 as significant pairs.

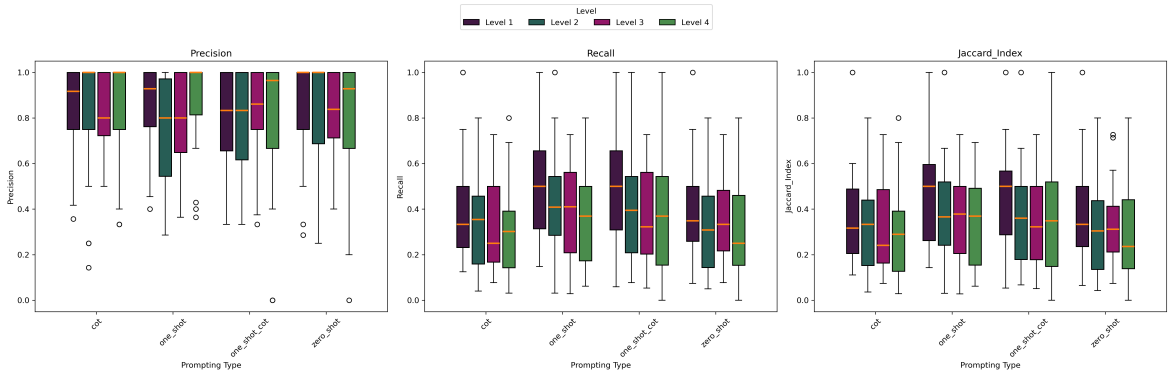

Figure 4.6: Box Plot for Basic Block Exit Identification Data of All Three Metrics

Prompting Type and Level The combination of these two independent variables does not show statistical significance. The Table a.16 shows the results from the ART ANOVA test.

Data Visualization The Figure 4.6 shows the distribution of the data across metrics, levels, and prompting types. The median is observed to be higher than zero in all cases. We notice similar patterns in the Jaccard Index and Recall metrics. For precision, the median is always greater than 0.6, indicating that the LLM provides more TP than FP.

Summary In Summary, prompting type and level individually have some significance in data. However, the combination of these two independent variables is not observed to be significant. The Figure 4.6 for precision shows that the LLM provides more TP than FP. We observe a similar pattern in results as basic block entry, with the exception that we observe fewer significant pairs than in basic block entry identification.

4.4 Research Question 2

This section aims to answer the research question: *How precisely can the LLM create def-use chains with snippets with increasing complexity of data flow across various prompting methods?*. To answer this question, we examine the data collected for the def-use chains task.

For the task of def-use chains, Shapiro-Wilk test on the data for all three metrics($p_{\text{Precision}} = 6.17\text{e}{-30}$, $p_{\text{Recall}} = 4.409\text{e}{-29}$, $p_{\text{Jaccard}} = 1.80\text{e}{-29}$) shows high statistical significance.

To properly analyze the results, the data are further analyzed by prompting type and code snippet level using the Friedman test (see Table 4.9).

Per Prompting Type The Friedman test revealed no statistically significant effect of prompting type for Jaccard Index ($\chi^2 = 6.23$, $p = 1.008\text{e}{-01}$) or Precision ($\chi^2 = 5.53$, $p = 1.370\text{e}{-01}$), as shown in Table 4.9. Although Recall yielded a significant Friedman result ($\chi^2 = 8.76$, $p = 3.270\text{e}{-02}$), post-hoc Wilcoxon signed-rank tests (Bonferroni-corrected) identified no significant pairs. No post-hoc tests were conducted for the Jaccard Index and Precision.

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	6.23	1.008e−01	–
Precision	5.53	1.370e−01	–
Recall	8.76	3.270e−02	–
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	5.21	1.570e−01	–
Precision	5.97	1.133e−01	–
Recall	4.87	1.816e−01	–

Table 4.9: Friedman Tests per Prompting Type and Metric for the Def-Use Chains Dataset.

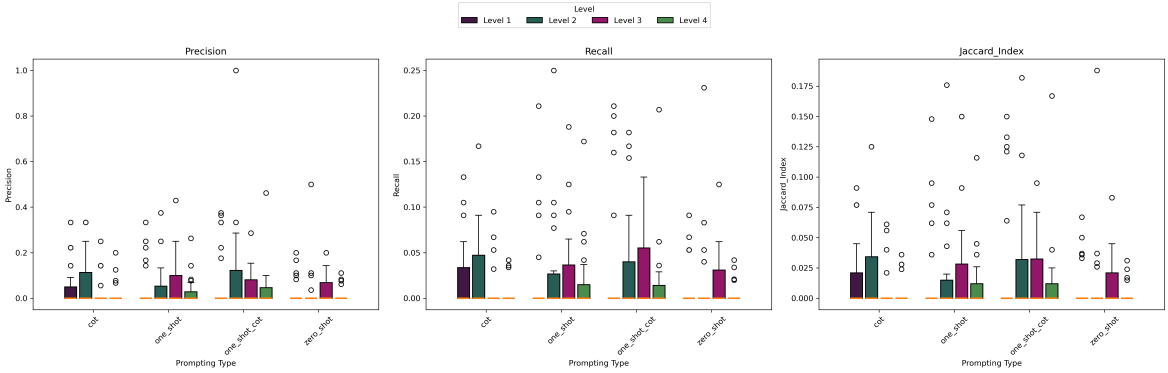

Figure 4.7: Box Plot for Def-Use Chains Identification Data of All Three Metrics

Per Level The Friedman test revealed no statistically significant effect of difficulty level on any of the three metrics: Jaccard Index ($\chi^2 = 5.21$, $p = 1.570\text{e}{-01}$), Precision ($\chi^2 = 5.97$, $p = 1.133\text{e}{-01}$), and Recall ($\chi^2 = 4.87$, $p = 1.816\text{e}{-01}$), as shown in Table 4.9. No post-hoc tests were conducted.

Prompting Type and Level The combination of prompting type and Levels did not prove to be significant from the ART ANOVA test, as observed in Table a.17.

Data Visualization The Figure 4.7 shows the distribution of the result across the metric. For all three metrics, we observe the median to be of value 0. For CoT Levels 3 and 4, we have only a few results that are not 0. Similarly, the LLM fails on most zero-shot tasks, except Level 3 tasks. Across Recall, Jaccard, and Precision, one-shot CoT prompting consistently outperforms all other prompting strategies. However, low medians indicate

that improvements are driven primarily by high-performing outliers rather than consistent gains.

Summary Overall, we do not see significant pairs to say the LLM can produce better def-use chains, though the prompting types. Nor does the level of data flow complexity significantly affect the ability of the LLM to identify def-use chains.

4.5 Research Question 3

This section aims to answer the research question: *How precisely can the LLM identify the output of the code snippet and perform backward program slicing with snippets with increasing complexity of data flow across various prompting methods?*.To answer this question, we analyze the data for the two tasks, Program output and Program slice.

4.5.1 Program Output

For the task of program output identification, Shapiro-Wilk test on the data for all three metrics($p_{\text{Precision}} = 2.706\text{e}{-17}$, $p_{\text{Recall}} = 2.591\text{e}{-17}$, $p_{\text{Jaccard}} = 3.057\text{e}{-16}$) shows high statistical significance.

Per Prompting Type The Friedman test revealed no statistically significant effect of prompting type on any of the three metrics: Jaccard Index ($\chi^2 = 3.22$, $p = 3.583\text{e}{-01}$), Precision ($\chi^2 = 5.34$, $p = 1.488\text{e}{-01}$), and Recall ($\chi^2 = 1.56$, $p = 6.694\text{e}{-01}$), as shown in Table 4.10. No post-hoc tests were conducted.

Per Level The Friedman test revealed statistically significant effects of difficulty level across all three metrics: Jaccard Index ($\chi^2 = 19.95$, $p = 1.737\text{e}{-04}$), Precision ($\chi^2 = 13.15$, $p = 4.318\text{e}{-03}$), and Recall ($\chi^2 = 16.22$, $p = 1.024\text{e}{-03}$), as shown in Table 4.10. For Jaccard Index, four significant pairs were identified: Level 1 with Level 2 ($W = 331.5$, $p = 1.429\text{e}{-03}$, $r = 0.495$), Level 1 with Level 3 ($W = 485.0$, $p = 3.991\text{e}{-02}$, $r = 0.359$), Level 1 with Level 4 ($W = 533.0$, $p = 1.177\text{e}{-04}$, $r = 0.507$), and Level 3 with Level 4 ($W = 525.5$, $p = 2.480\text{e}{-02}$, $r = 0.370$). For Precision, two significant pairs were found: Level 1 with Level 2 ($W = 360.0$, $p = 5.923\text{e}{-03}$, $r = 0.448$) and Level 1 with Level 4 ($W = 599.0$, $p = 2.707\text{e}{-03}$, $r = 0.425$). For Recall, three significant pairs were identified: Level 1 with Level 2 ($W = 393.5$, $p = 2.609\text{e}{-02}$, $r = 0.392$), Level 1 with Level 4 ($W = 539.0$, $p = 1.365\text{e}{-04}$, $r = 0.503$), and Level 3 with Level 4 ($W = 448.0$, $p = 1.576\text{e}{-02}$, $r = 0.398$). All effect sizes were medium (r ranging from 0.359 to 0.507). The remaining pairs did not differ significantly.

Prompting Type and Level The Table a.18 confirms the previously done Friedman test and shows that there is no statistically significant combined effect of Prompting types.

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	3.22	3.583e−01	–
Precision	5.34	1.488e−01	–
Recall	1.56	6.694e−01	–
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	19.95	1.737e−04	L1 vs L2 ($W = 331.5$, $p = 1.429\text{e}{-03}$, $r = 0.495$)
			L1 vs L3 ($W = 485.0$, $p = 3.991\text{e}{-02}$, $r = 0.359$)
			L1 vs L4 ($W = 533.0$, $p = 1.177\text{e}{-04}$, $r = 0.507$)
			L3 vs L4 ($W = 525.5$, $p = 2.480\text{e}{-02}$, $r = 0.370$)
Precision	13.15	4.318e−03	L1 vs L2 ($W = 360.0$, $p = 5.923\text{e}{-03}$, $r = 0.448$)
			L1 vs L4 ($W = 599.0$, $p = 2.707\text{e}{-03}$, $r = 0.425$)
Recall	16.22	1.024e−03	L1 vs L2 ($W = 393.5$, $p = 2.609\text{e}{-02}$, $r = 0.392$)
			L1 vs L4 ($W = 539.0$, $p = 1.365\text{e}{-04}$, $r = 0.503$)
			L3 vs L4 ($W = 448.0$, $p = 1.576\text{e}{-02}$, $r = 0.398$)

Table 4.10: Friedman Tests per Prompting Type and Metric for the Program Output Dataset

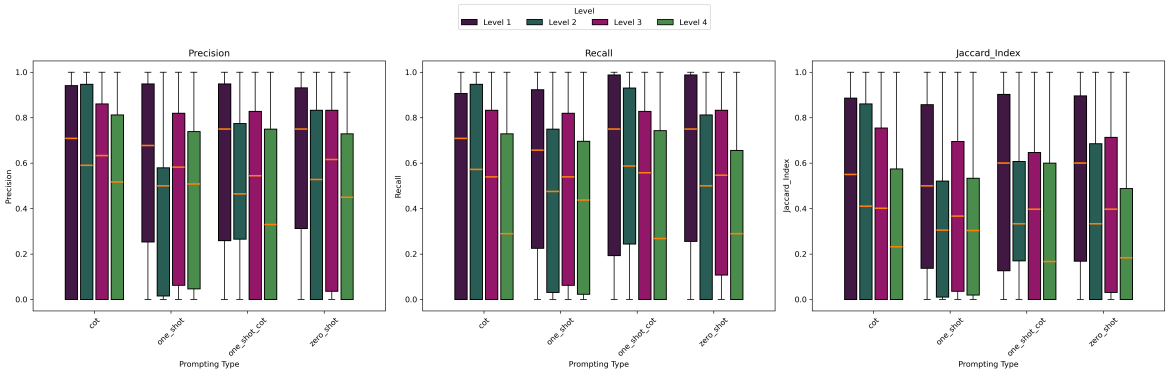

Figure 4.8: Box Plot for Program Output Identification Data of All Three Metrics

Data Visualization The Figure 4.8 shows that the data varies from 0 to 1. The entire range is covered by the output. This indicates that the results are quite varied. Higher median means that we often get results from the LLM on the task that are not entirely incorrect.

Summary In summary, the prompting type is not observed to have significant pairs. However, for level-wise, we observe many significant pairs. Level 1 is observed to almost always be significant with all the other Levels. The combination of prompting type and levels did not yield a statistically significant result.

4.5.2 Program Slice

For the task of program output identification, Shapiro-Wilk test on the data for all three metrics($p_{\text{Precision}} = 4.238\text{e}{-10}$, $p_{\text{Recall}} = 2.504\text{e}{-08}$, $p_{\text{Jaccard}} = 3.529\text{e}{-07}$) shows high statistical significance. The data is further observed per prompting type and per level separately. The results of these tests can be observed in Table 4.11

Per Prompting Type The Friedman test revealed no statistically significant effect of prompting type on any of the three metrics: Jaccard Index ($\chi^2 = 2.84$, $p = 4.162\text{e}{-01}$), Precision ($\chi^2 = 1.75$, $p = 6.269\text{e}{-01}$), and Recall ($\chi^2 = 6.81$, $p = 7.820\text{e}{-02}$), as shown in Table 4.11. No post-hoc tests were conducted.

Friedman Test – Prompting Type			
Metric	χ^2	p	Significant pairs
Jaccard Index	2.84	4.162e−01	–
Precision	1.75	6.269e−01	–
Recall	6.81	7.820e−02	–
Friedman Test – Level			
Metric	χ^2	p	Significant pairs
Jaccard Index	10.77	1.303e−02	L2 vs L3 ($W = 1251.5$, $p = 1.971\text{e}{-02}$, $r = 0.313$)
Precision	5.36	1.471e−01	–
Recall	10.57	1.432e−02	–

Table 4.11: Friedman Tests per Prompting Type and Metric for the Program Slice Dataset

Per Level The Friedman test revealed statistically significant results for Jaccard Index ($\chi^2 = 10.77$, $p = 1.303\text{e}{-02}$) and Recall ($\chi^2 = 10.57$, $p = 1.432\text{e}{-02}$), but not for Precision ($\chi^2 = 5.36$, $p = 1.471\text{e}{-01}$), as shown in Table 4.11. Post-hoc Wilcoxon signed-rank tests

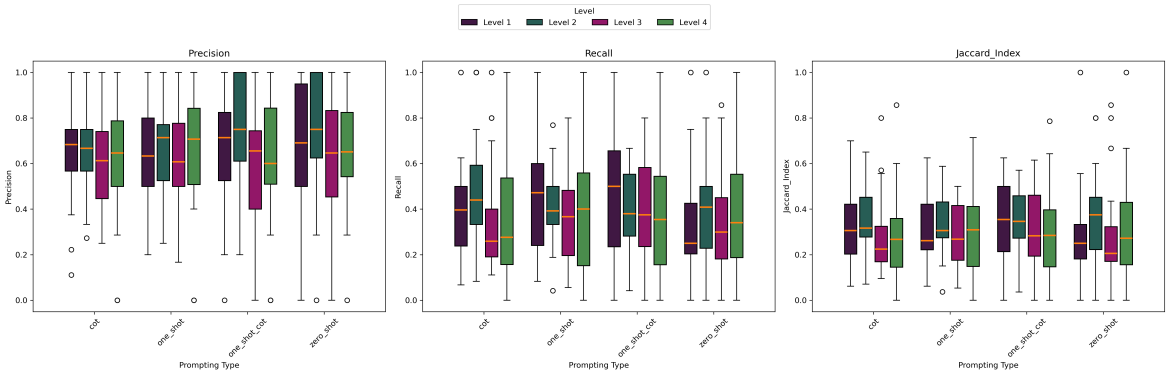

Figure 4.9: Box Plot for Program Slice Identification Data of All Three Metrics

(Bonferroni-corrected) for Jaccard Index identified one significant pair: Level 2 with Level 3 ($W = 1251.5$, $p = 1.971e{-}02$, $r = 0.313$). For Recall, no significant pairs were identified despite the significant Friedman result. The remaining pairs did not differ significantly.

Prompting Type and Level For the combined effect of prompting type and Level, no significant relationships were found. The Table a.19 shows the obtained results.

Data Visualization The Figure 4.9 shows the distribution of the data for all three metrics per prompting type and per level. The median is observed to be above 0 for all three metrics. For precision, we observe the data to be mainly distributed around the value 0.5 . This indicates that the LLM has more true positive answers than hallucinated positive responses. Level-wise, we do not observe a consistent reduction in results across the three metrics.

Summary In Summary, we observe only one significant pair(Level 2 with Level 3) per level-wise for the Jaccard Index metric. The data is observed to be distributed across the metric in the range [0,1]. We do not observe any statistically significant pairs per prompting type and for the combination of prompting type with levels.

4.6 Research Question 4

This section aims to answer the research question *How does the complexity of code measured by different metrics affect the precision of the LLM on the task of identification of defs, uses, function calls, and basic blocks with snippets with increasing complexity of data flow in various prompting?*. It analyzes each task separately. Basic blocks are further analyzed in three ways: the entirety of the basic block, the entry line of code to the basic block, and the exit line of code from the basic block.

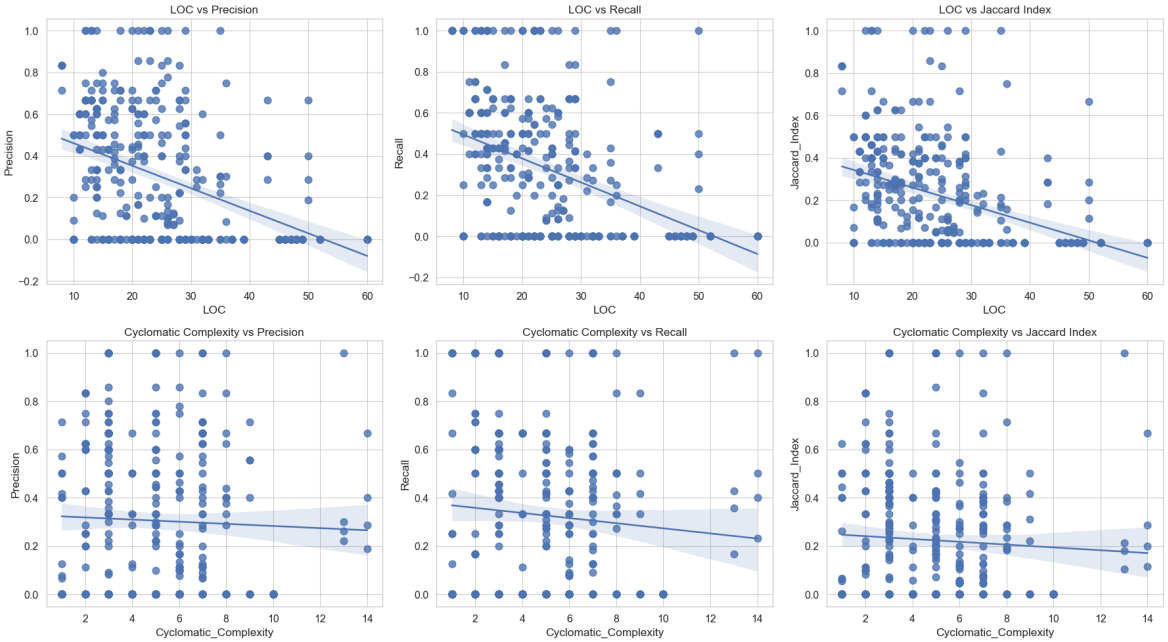

Figure 4.10: Defs Data Distribution on the Three Metrics against the Code Complexity Metrics

4.6.1 Defs

This subsection only analyzes the results for the task of defs identification. The Figure 4.10 shows the distribution per evaluation metric and code complexity metric. We observe a consistent negative slope for all the plots. The Table 4.12 shows the significant results for each prompting type and levels.

Global

LOC The global correlational analysis with the metric LOC revealed all evaluation metrics to be highly significant. Precision ($\rho = -0.41$, p-value = 3.058e−14), Recall ($\rho = -0.42$, p-value = 7.743e−15), and Jaccard Index ($\rho = -0.42$, p-value= 7.066e−15) show a consistent negative medium correlation. This indicates that with an increase in LOC, we see a consistent reduction in the performance of the LLM in all the evaluation metrics.

Cyclomatic Complexity The global correlation analysis with the metric, Cyclomatic complexity, on all of the evaluation metrics showed to be not significant throughout. This indicates that the Cyclomatic complexity does not have any effect on the results of the LLM.

Per Prompting Type

LOC For all 4 prompting strategies used, we observe highly significant results indicating the existence of some significant correlation. For CoT prompting ($\rho_{precision} = -0.43$, $\rho_{recall} = -0.43$, $\rho_{jaccard} = -0.43$) we observe a consistent negative moderate correlation. We also observe such similar negative moderate correlation for One-shot prompting ($\rho_{precision} =$

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	−0.407	−0.417	−0.422	3.058e−14	7.743e−15	7.066e−15
	Cyclomatic	−0.038	−0.091	−0.076	6.361e−01	1.558e−01	2.486e−01
			By Prompting Type (LOC only – Cyclomatic not significant)				
CoT	LOC	−0.427	−0.430	−0.426	1.559e−04	1.559e−04	1.559e−04
One-Shot	LOC	−0.401	−0.421	−0.429	3.362e−04	1.666e−04	1.559e−04
One-Shot CoT	LOC	−0.422	−0.456	−0.448	1.666e−04	7.113e−05	9.372e−05
Zero-Shot	LOC	−0.371	−0.373	−0.383	8.897e−04	8.447e−04	6.285e−04
			By Level (LOC only – Cyclomatic not significant)				
Level 1	LOC	−0.287	−0.317	−0.324	1.347e−02	5.493e−03	4.544e−03
Level 2	LOC	−0.413	−0.395	−0.417	2.117e−04	4.150e−04	1.953e−04
Level 3	LOC	−0.458	−0.440	−0.474	7.113e−05	1.222e−04	4.242e−05
Level 4	LOC	−0.392	−0.363	−0.380	4.536e−04	1.127e−03	6.667e−04

Table 4.12: Spearman Correlations between Complexity Metrics and Performance Metrics for the Defs Data

-0.40, $\rho_{recall} = -0.42$, $\rho_{jaccard} = -0.43$) and One-shot CoT ($\rho_{precision} = -0.42$, $\rho_{recall} = -0.46$, $\rho_{jaccard} = -0.45$). Zero-shot prompting shows low negative significance ($\rho_{precision} = -0.37$, $\rho_{recall} = -0.37$, $\rho_{jaccard} = -0.38$). These results indicate that for all the prompting types, we notice that LOC has a moderate effect and observe a reduction in results from LLM when LOC increases. Only for zero-shot, we observe a lower effect of LOC than the other prompting types.

Cyclomatic Complexity For all the prompting types, we do not observe any significant effect from the Cyclomatic complexity metric. This means Cyclomatic complexity has no effect on the results obtained from the LLM.

Per Level

LOC For all the 4 levels, we notice the existence of significant correlation. For Level-2 ($\rho_{precision} = -0.41$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.42$), Level-3 ($\rho_{precision} = -0.46$, $\rho_{recall} = -0.44$, $\rho_{jaccard} = -0.47$),and Level-4 ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.36$, $\rho_{jaccard} = -0.38$) we have high significant correlation. Level 2 and Level 3 have a negative moderate correlation. While Level 1 and Level 4 have a low negative correlation. This indicates that LOC always affects the results. Increase in LOC shows a reduction in results from LLM. The effect of LOC is higher in Level 2 and Level 3 than in Level 1 and Level 4.

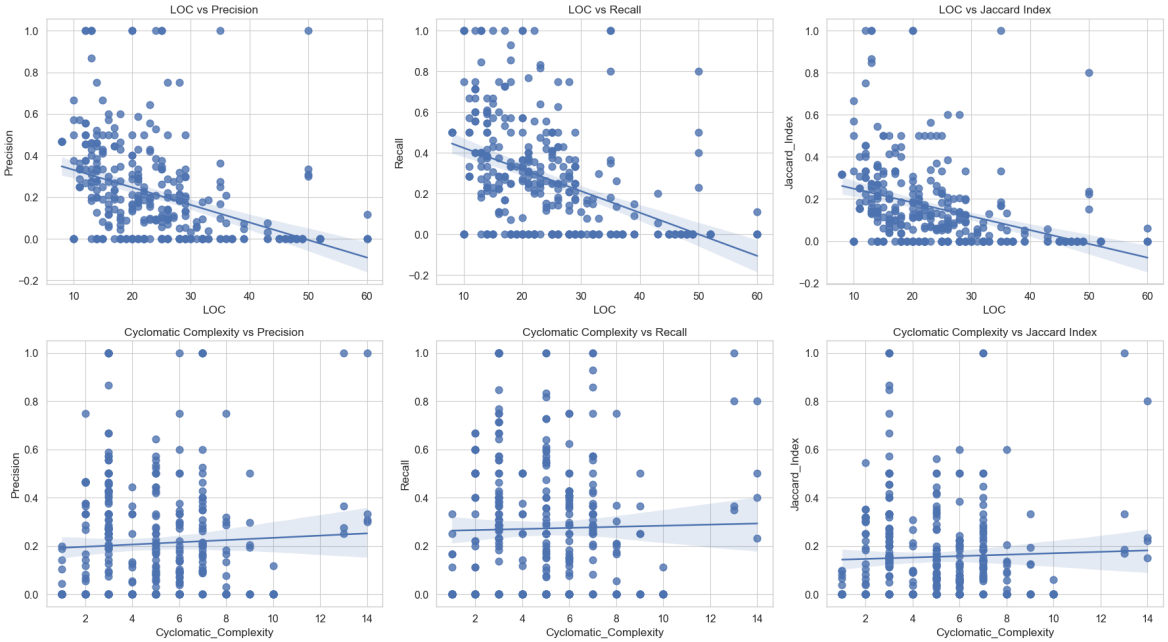

Figure 4.11: Uses Data Distribution on the Three Metrics against the Code Complexity Metrics

Cyclomatic Complexity Cyclomatic complexity does not show any significant correlation on any of the 4 levels of code complexity. This indicates that this metric does not influence the results from the LLM on the tasks of defs.

Summary Overall, LOC consistently shows a significant negative correlation for evaluation metrics. The effect of correlation stayed low to moderate. For cyclomatic complexity, we consistently observe no significant relation. Hence, we conclude for our research question for this task.

4.6.2 Uses

This subsection focuses on the task of use identification. The Figure 4.11 shows the distribution of all the evaluation metrics with code complexity metrics. For LOC we observe a consistent negative slope. While for cyclomatic complexity, the slope is slightly positive. The Table 4.13 shows the obtained significant results

Global

LOC The global correlational analysis with the metric LOC revealed all evaluation metrics to be highly significant. Precision ($\rho = -0.49$, p-value = 3.283e-21), Recall ($\rho = -0.47$, p-value = 1.111e-19), and Jaccard Index ($\rho = -0.49$, p-value= 3.283e-21) show a consistent negative medium correlation. This indicates that with an increase in LOC, we see a consistent reduction in the performance of the LLM in all the evaluation metrics.

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	−0.490	−0.472	−0.490	3.283e−21	1.111e−19	3.283e−21
	Cyclomatic		–			not significant	
			By Prompting Type (LOC only – Cyclomatic not significant)				
CoT	LOC	−0.481	−0.433	−0.477	7.535e−06	6.886e−05	9.084e−06
One-Shot	LOC	−0.573	−0.559	−0.579	3.551e−08	8.991e−08	2.741e−08
One-Shot CoT	LOC	−0.497	−0.506	−0.496	3.363e−06	2.603e−06	3.363e−06
Zero-Shot	LOC	−0.442	−0.429	−0.441	4.738e−05	7.687e−05	4.738e−05
			By Level				
Level 1	LOC	−0.369	−0.352	−0.363	8.837e−04	1.569e−03	1.061e−03
	Cyclomatic	+0.282	+0.299	+0.307	1.412e−02	8.585e−03	6.930e−03
Level 2	LOC	−0.501	−0.471	−0.499	3.226e−06	1.153e−05	3.342e−06
Level 3	LOC	−0.588	−0.594	−0.603	1.493e−08	1.126e−08	6.872e−09
Level 4	LOC	−0.385	−0.385	−0.389	4.860e−04	4.860e−04	4.444e−04

Table 4.13: Spearman Correlations between Complexity Metrics and Performance Metrics for the Uses Data

Cyclomatic Complexity The global correlation analysis with the metric, Cyclomatic complexity, on all of the evaluation metrics showed to be not significant throughout. This indicates that the Cyclomatic complexity does not have any effect on the results of the LLM.

Per Prompting Type

LOC For all 4 prompting strategies used, we observe a highly significant correlation. CoT prompting ($\rho_{precision} = -0.48$, $\rho_{recall} = -0.43$, $\rho_{jaccard} = -0.48$), One-shot (Precision: $\rho = -0.57$, $\rho_{recall} = -0.56$, $\rho_{jaccard} = -0.58$), One-shot CoT ($\rho_{precision} = -0.50$, $\rho_{recall} = -0.51$, $\rho_{jaccard} = -0.50$) and Zero-shot ($\rho_{precision} = -0.44$,$\rho_{recall} = -0.43$, $\rho_{jaccard} = -0.44$) shows negative moderate correlation. When LOC increases for all the prompting strategies, we observe a moderate effect on the reduction in the results of the LLM.

Cyclomatic Complexity For all the prompting types, we do not observe any significant effect from the cyclomatic complexity metric. This means cyclomatic complexity has no effect on the results obtained from the LLM.

Per Level

LOC For all the 4 levels, we observe statistically significant. Level 1 ($\rho_{precision} = -0.37$, $\rho_{recall} = -0.35$, $\rho_{jaccard} = -0.36$) showed high statistical significance for the metric precision with negative moderate correlation for all evaluation metrics. Level 2 ($\rho_{precision} = -0.50$, $\rho_{recall} = -0.47$, $\rho_{jaccard} = -0.50$) , Level 3 ($\rho_{precision} = -0.59$, $\rho_{recall} = -0.59$, $\rho_{jaccard} = -0.60$) and Level 4 ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.38$, $\rho_{jaccard} = -0.39$) all show negative moderate correlation for all evaluation metrics. This means an increase in LOC shows a reduction in results from LLM.

Cyclomatic Complexity Cyclomatic complexity showed only significant results for Level 1 ($\rho_{precision} = 0.28$, $\rho_{recall} = 0.30$, $\rho_{jaccard} = 0.31$) with positive low correlation. Indicating that an increase in cyclomatic complexity for Level 1 snippets shows an increase in the results obtained from the LLM on all three evaluation metrics. However, the same could not be observed for other Levels. Level 2 , Level 3, and Level 4 did not show any significant correlation.

Summary Overall, LOC consistently shows a significant negative correlation for evaluation metrics. The effect of correlation stayed low to moderate. For cyclomatic complexity, we consistently observe no significant relation except for one case in Level 1 cyclomatic complexity. Hence, we conclude for our research question for this task.

4.6.3 Function Calls

This subsection focuses on the task of functional calls identification. The Figure 4.12 shows the distribution of all the results for the task of function calls on all three evaluation metrics with LOC and cyclomatic complexity. In all the plots, we observe a negative slope. The Table 4.14 shows significant results obtained per prompting type and level for both the code complexity metrics.

Global

LOC For all three metrics, Precision ($\rho = -0.28$, p-value= 3.470e$-$06), Recall($\rho = -0.26$, p-value=9.225e$-$06), and Jaccard Index ($\rho = -0.28$, p-value= 3.470e$-$06), we found the results to be highly significant. This shows that the LOC has a negative low correlation on the three metrics. Increase in LOC, we see a reduction in results from the LLM.

Cyclomatic Complexity For all three metrics, Precision ($\rho = -0.16$, p-value = 1.004e$-$02), Recall($\rho = -0.16$, p-value = 1.004e$-$02), and Jaccard Index ($\rho = -0.17$, p-value = 9.229e$-$03), we found the results to be highly significant. This shows that the cyclomatic

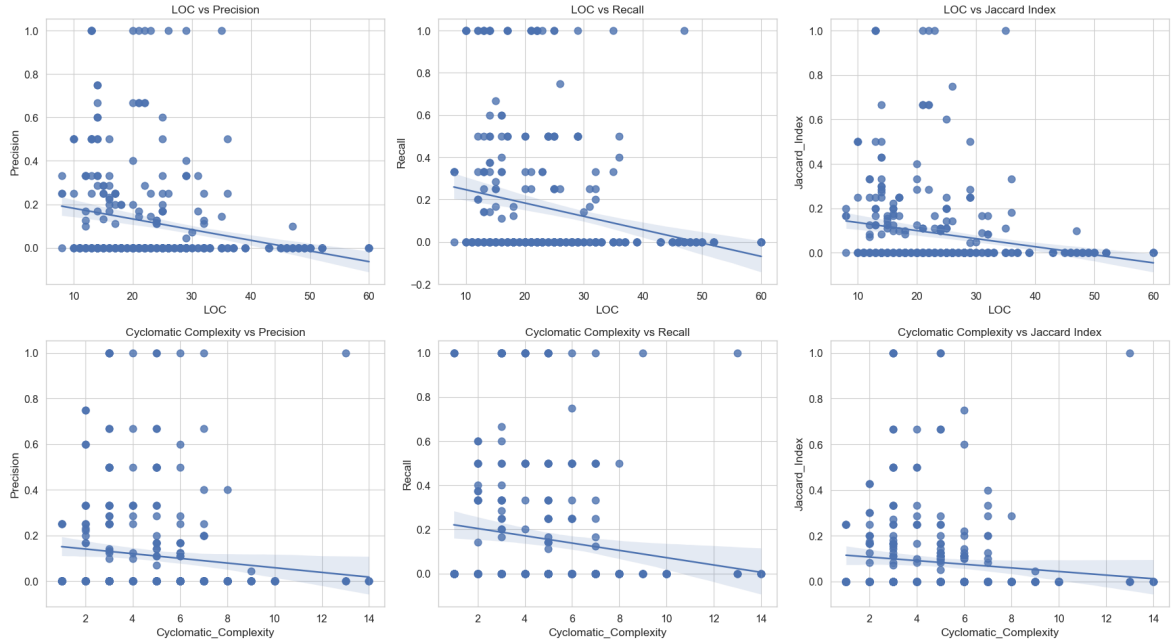

Figure 4.12: Function Calls Data Distribution on the Three Metrics against the Code Complexity Metrics

complexity has a low negative correlation with the three metrics. An increase in cyclomatic complexity results in a reduction in results from the LLM.

Per Prompting Type

LOC The effect of LOC on each of the prompting types observed. CoT and One-shot CoT did not show any significance. One-shot ($\rho_{precision} = -0.35$, $\rho_{recall} = -0.33$, $\rho_{jaccard} = -0.35$) prompting and Zero-shot prompting ($\rho_{precision} = -0.32$, $\rho_{recall} = -0.30$, $\rho_{jaccard} = -0.31$) showed significant effect with negative low correlation.

Cyclomatic Complexity The effect of Cyclomatic complexity with the 4 prompting types was found to be not significant. The only exception was for CoT ($\rho_{precision} = -0.25$, $\rho_{recall} = -0.24$, $\rho_{jaccard} = -0.25$) for the metric Jaccard Index, a significant negative low correlation was observed. This indicates that only for the Jaccard Index metric in CoT prompting, an increase in Cyclomatic complexity of a snippet indicated a low reduction in result for Jaccard Index for this task.

Per Level

LOC At Level 1 ($\rho_{precision} = -0.41$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.41$), the results indicate a highly significant moderate negative correlation across all evaluated metrics, suggesting a consistent inverse relationship at this level. At Level 3 ($\rho_{precision} = -0.27$, $\rho_{recall} = -0.23$, $\rho_{jaccard} = -0.26$), a significantly low negative correlation is observed for Precision and the Jaccard Index, while Recall does not show a significant relationship. In contrast, Level 2 and

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
				Global			
	LOC	−0.280	−0.264	−0.277	3.470e−06	9.225e−06	3.470e−06
	Cyclomatic	−0.162	−0.163	−0.168	1.004e−02	1.004e−02	9.229e−03
				By Prompting Type			
CoT	Cyclomatic	–	–	−0.249	–	–	4.992e−02
One-Shot	LOC	−0.351	−0.328	−0.349	5.782e−03	9.785e−03	5.782e−03
One-Shot CoT	*no significant correlations*						
Zero-Shot	LOC	−0.315	−0.303	−0.311	1.148e−02	1.472e−02	1.226e−02
				By Level			
Level 1	LOC	−0.412	−0.410	−0.413	6.655e−04	6.655e−04	6.655e−04
	Cyclomatic	−0.269	−0.267	−0.273	3.366e−02	3.395e−02	3.326e−02
Level 2	*no significant correlations*						
Level 3	LOC	−0.272	–	−0.263	3.326e−02	–	3.570e−02
Level 4	*no significant correlations*						

Table 4.14: Spearman Correlations between Complexity Metrics and Performance Metrics for the Function Calls Data

Level 4 do not demonstrate any statistically significant correlations. Overall, meaningful effects are primarily concentrated at Level 1, with partial significance at Level 3.

Cyclomatic Complexity The effect of Cyclomatic complexity with the 4 levels was found to be not significant, with the exception of Level 1 ($\rho_{precision} = -0.27$, $\rho_{recall} = -0.27$, $\rho_{jaccard} = -0.27$). For all the metrics, a significant negative low correlation was observed. This indicates that only for Level 1, an increase in Cyclomatic complexity of a snippet in Level 1 indicated a low reduction in result for all the evaluation metrics for this task.

Summary In Summary, the effect of LOC and Cyclomatic complexity are observed to be individually significant on the entirety of the data. However, the effect on the prompting types and the level of code complexity is not consistent across. LOC still shows more significant correlation than cyclomatic complexity on the three metrics.

4.6.4 Basic blocks

This subsection focuses on the results obtained on the task for basic block identification. The Figure 4.13 shows the distribution of the data from the three evaluation metrics on the metric LOC and Cyclomatic complexity. The slope on all the plots is observed to be negative. The Table 4.15 shows the statistically significant results for this task.

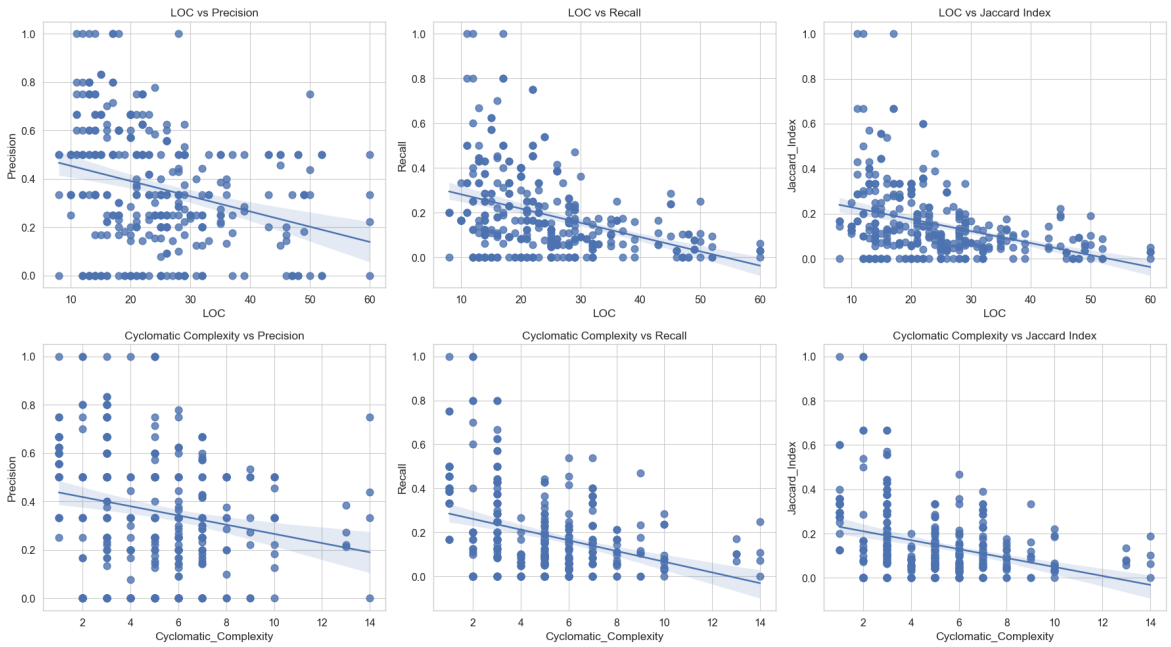

Figure 4.13: Basic Block Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC The global correlational analysis with the metric LOC revealed all evaluation metrics to be highly significant. Precision ($\rho = -0.31$, p-value = 6.929e−08), Recall ($\rho = -0.38$, p-value = 2.239e−12), and Jaccard Index ($\rho = -0.39$, p-value= 1.690e−12) show a consistent negative low correlation. This indicates that with an increase in LOC, we see a consistent reduction in the performance of the LLM in all the evaluation metrics.

Cyclomatic Complexity The global correlational analysis with the metric cyclomatic complexity revealed all evaluation metrics to be highly significant. Precision ($\rho = -0.22$, p-value = 1.685e−04), Recall ($\rho = -0.30$, p-value = 1.763e−07), and Jaccard Index ($\rho = -0.30$, p-value= 1.763e−07) show a consistent negative medium correlation. This indicates that with an increase in cyclomatic complexity, we see a consistent reduction in the performance of the LLM in all the evaluation metrics.

Per Prompting Type

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	−0.307	−0.384	−0.390	6.929e−08	2.239e−12	1.690e−12
	Cyclomatic	−0.221	−0.297	−0.295	1.685e−04	1.763e−07	1.763e−07
			By Prompting Type				
CoT	LOC	−0.240	−0.309	−0.311	3.016e−02	6.083e−03	5.854e−03
	Cyclomatic	–	−0.270	−0.289	–	1.557e−02	1.066e−02
One-Shot	LOC	−0.371	−0.399	−0.423	1.021e−03	4.010e−04	1.982e−04
	Cyclomatic	–	−0.251	−0.261	–	2.389e−02	1.959e−02
One-Shot CoT	LOC	−0.386	−0.527	−0.518	6.404e−04	1.195e−06	1.868e−06
	Cyclomatic	−0.283	−0.401	−0.376	1.197e−02	4.010e−04	8.603e−04
Zero-Shot	LOC	−0.253	−0.318	−0.324	2.363e−02	4.813e−03	4.091e−03
	Cyclomatic	−0.229	−0.279	−0.285	3.842e−02	1.263e−02	1.149e−02
			By Level				
Level 1	LOC	−0.246	–	–	2.684e−02	–	–
	Cyclomatic	−0.276	−0.378	−0.369	1.351e−02	8.597e−04	1.021e−03
Level 2	LOC	−0.304	−0.363	−0.368	6.941e−03	1.092e−03	1.021e−03
	Cyclomatic	−0.282	–	–	1.197e−02	–	–
Level 3	LOC	−0.366	−0.399	−0.406	1.063e−03	4.010e−04	3.608e−04
	Cyclomatic	−0.418	−0.425	−0.445	2.254e−04	1.960e−04	9.589e−05
Level 4	LOC	−0.233	−0.364	−0.360	3.518e−02	1.092e−03	1.186e−03
	Cyclomatic		–		not significant		

Table 4.15: Spearman Correlations between Complexity Metrics and Performance Metrics for the Basic Block Data

LOC The results show that the One-shot ($\rho_{precision} = -0.37$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.42$) and One-shot CoT ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.53$, $\rho_{jaccard} = -0.52$) for the prompting strategies, LOC exhibits highly significant negative low to moderate correlation across all evaluated metrics, indicating a strong and consistent inverse relationship under these

approaches. The CoT ($\rho_{precision} = -0.24$, $\rho_{recall} = -0.31$, $\rho_{jaccard} = -0.31$) and Zero-shot ($\rho_{precision} = -0.25$, $\rho_{recall} = -0.32$, $\rho_{jaccard} = -0.32$) strategies, LOC demonstrate significant negative low correlation, though comparatively weaker than the one-shot variants. Overall, prompting strategies incorporating examples, particularly when combined with CoT reasoning, show the strongest and most consistent effects.

Cyclomatic Complexity The CoT prompting ($\rho_{precision} = -0.21$, $\rho_{recall} = -0.27$, $\rho_{jaccard} = -0.29$) strategy and One-shot ($\rho_{precision} = -0.18$, $\rho_{recall} = -0.25$, $\rho_{jaccard} = -0.26$) strategy demonstrate a significant negative low correlation overall, although Precision does not reach statistical significance. In contrast, the One-shot CoT ($\rho_{precision} = -0.28$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.38$) strategy exhibits a highly significant low negative correlation overall, with Precision also reaching statistical significance. The Zero-Shot ($\rho_{precision} = -0.23$, $\rho_{recall} = -0.28$, $\rho_{jaccard} = -0.29$) strategy shows a significant negative low correlation across the evaluated metrics. This indicates that for all prompting strategies, an increase in Cyclomatic complexity indicates a low reduction in Recall and Jaccard Index consistently across all the metrics. For One-shot CoT, we observe a slightly higher recall value.

Per Level

LOC At Level 1 ($\rho_{precision} = -0.25$, $\rho_{recall} = -0.16$, $\rho_{jaccard} = -0.20$) , no overall statistical significance is observed, although Precision shows a significant negative low correlation. Level 2 ($\rho_{precision} = -0.30$, $\rho_{recall} = -0.36$, $\rho_{jaccard} = -0.37$), Level 3 ($\rho_{precision} = -0.37$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.41$), and Level 4 ($\rho_{precision} = -0.23$, $\rho_{recall} = -0.36$, $\rho_{jaccard} = -0.36$) exhibit highly significant negative low correlation across the evaluated metrics, with Precision also reaching significance at Levels 2 and 4. Level 3 exhibits a moderate negative correlation for the metrics Recall and Jaccard Index. Overall, we observe consistent negative correlation for all significant relationships. It indicates that an increase in LOC for any of the significant levels and metrics, a reduction in the result from the LLM on that metric can be observed.

Cyclomatic Complexity For Level 1 ($\rho_{precision} = -0.28$, $\rho_{recall} = -0.38$, $\rho_{jaccard} = -0.37$), a highly significant negative low correlation is observed overall, with precision also reaching significance. At Level 2 ($\rho_{precision} = -0.28$, $\rho_{recall} = -0.17$, $\rho_{jaccard} = -0.21$), no overall significant correlation is found, except that precision shows a significantly low negative correlation. Level 3 ($\rho_{precision} = -0.42$, $\rho_{recall} = -0.43$, $\rho_{jaccard} = -0.44$) demonstrates a highly significant negative moderate correlation across the metrics. In contrast, Level 4 does not show any statistically significant correlation.

Summary In summary, although LOC and Cyclomatic complexity exhibit significant correlations when considering the overall dataset, these relationships are not consistently maintained across individual levels and prompting strategies. The significance and strength of the correlations vary depending on the specific level and prompting type, indicating that the observed effects are context-dependent rather than uniformly robust.

4.6.5 Basic Block Entry

This section focuses on the task of Basic block entry data. The Figure 4.14 shows the distribution of the data for all combinations of the three evaluation metrics with code complexity metrics. The slope in all cases is observed to be negative. The Table 4.16 contains all the statistically significant results observed for this task.

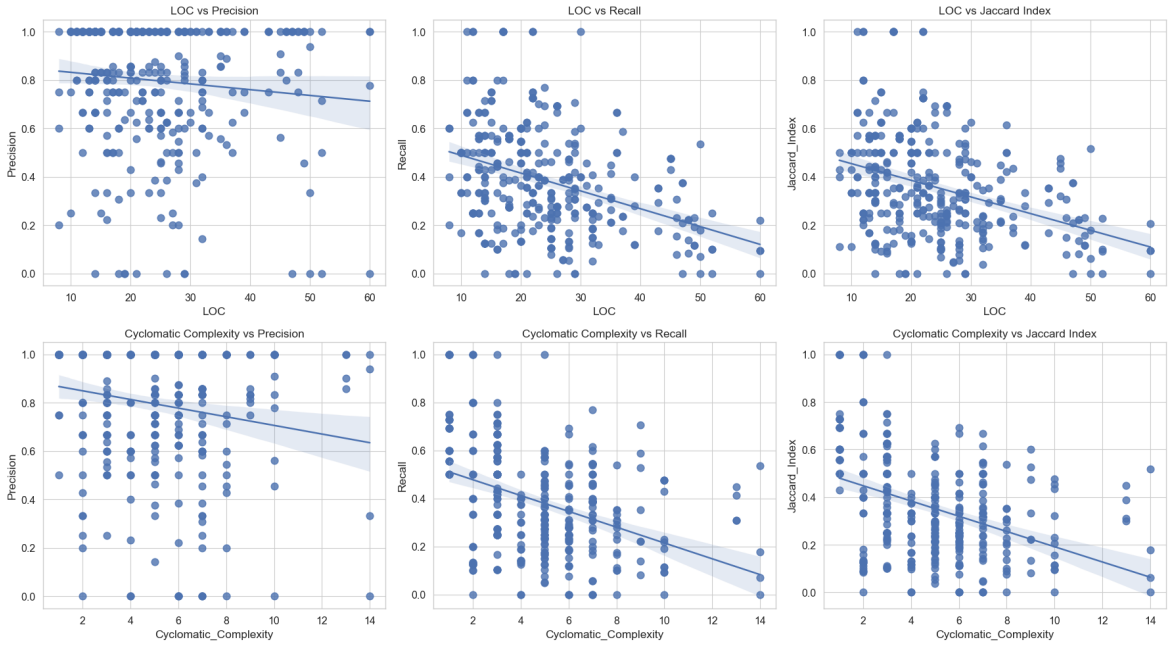

Figure 4.14: Basic Block Entry Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC The LOC complexity metric showed high significance for the metrics: Jaccard Index($\rho = -0.35$, p-value= 1.4791e-10) and Recall($\rho = -0.36$, p-value= 7.174e$-$11). For precision($\rho = -0.13$, p-value = 2.603e$-$02), LOC shows statistical significance with low negative correlation.

Cyclomatic Complexity The Cyclomatic complexity showed high significance on all three metrics, Precision ($\rho = -0.28$, p-value= 3.470e$-$06), Recall($\rho = -0.26$, p-value=9.225e$-$06), and Jaccard Index ($\rho = -0.28$, p-value= 3.470e$-$06)

Per Prompting Type

LOC The Spearman correlation results indicate that the CoT prompting ($\rho_{precision} = -0.20$, $\rho_{recall} = -0.29$, $\rho_{jaccard} = -0.31$) strategy shows a significant moderate negative correlation overall, although Precision does not reach statistical significance. The one-shot ($\rho_{precision} = -0.11$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.41$) approach demonstrates a highly

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	−0.127	−0.359	−0.352	2.603e−02	7.175e−11	1.479e−10
	Cyclomatic	−0.223	−0.380	−0.387	8.646e−05	4.138e−12	2.928e−12
			By Prompting Type				
CoT	LOC	–	−0.291	−0.305	–	1.014e−02	6.916e−03
	Cyclomatic	−0.327	−0.367	−0.416	3.708e−03	9.182e−04	1.715e−04
One-Shot	LOC	–	−0.402	−0.410	–	2.416e−04	1.891e−04
	Cyclomatic	–	−0.408	−0.413	–	1.979e−04	1.715e−04
One-Shot CoT	LOC	−0.234	−0.516	−0.502	3.950e−02	2.426e−06	4.942e−06
	Cyclomatic	–	−0.473	−0.438	–	1.907e−05	7.526e−05
Zero-Shot	LOC	–	−0.276	−0.225	–	1.493e−02	4.749e−02
	Cyclomatic	−0.242	−0.324	−0.320	3.405e−02	4.063e−03	4.389e−03
			By Level				
Level 1	LOC		–			not significant	
	Cyclomatic	−0.254	−0.458	−0.457	2.603e−02	3.808e−05	3.808e−05
Level 2	LOC	–	−0.413	−0.422	–	1.715e−04	1.423e−04
	Cyclomatic	−0.532	−0.414	−0.477	1.031e−06	1.715e−04	1.756e−05
Level 3	LOC	–	−0.445	−0.449	–	5.701e−05	5.157e−05
	Cyclomatic	–	−0.376	−0.372	–	6.751e−04	7.825e−04
Level 4	LOC	–	−0.303	−0.258	–	7.143e−03	2.446e−02
	Cyclomatic	–	−0.235	–	–	3.929e−02	–

Table 4.16: Spearman Correlations between Complexity Metrics and Performance Metrics for the Basic Block Entry Data

significant negative correlation, with Precision remaining non-significant. Similarly, the zero-shot ($\rho_{precision} = 0.02$, $\rho_{recall} = -0.28$, $\rho_{jaccard} = -0.22$) strategy exhibits a highly significant negative low correlation overall, while Precision does not show significance. In contrast, the one-shot CoT ($\rho_{precision} = -0.23$, $\rho_{recall} = -0.52$, $\rho_{jaccard} = -0.50$) strategy yields a highly

significant negative correlation across the metrics, with Precision also reaching statistical significance. The metrics Recall and Jaccard Index show moderate correlation in both One-shot and One-shot CoT prompting. Overall, stronger and more consistent effects are observed when example-based prompting is combined with CoT reasoning.

Cyclomatic Complexity The correlation analysis shows that the CoT prompting strategy ($\rho_{precision} = -0.33$, $\rho_{recall} = -0.37$, $\rho_{jaccard} = -0.42$) exhibits a highly significant negative low to moderate correlation across the evaluated metrics. Both the One-shot ($\rho_{precision} = -0.15$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.41$) and One-shot CoT ($\rho_{precision} = -0.15$, $\rho_{recall} = -0.47$, $\rho_{jaccard} = -0.44$) approaches also demonstrate highly significant with low negative correlation on the metrics, Jaccard Index and Recall. However, in these two strategies, Precision does not reach statistical significance. The zero-shot strategy shows a significant negative correlation across the metrics. Overall, cyclomatic complexity consistently presents strong inverse relationships for prompting types, particularly under CoT based prompting.

Per Level

LOC At Level 1, no statistically significant correlation is observed. Level 2 ($\rho_{precision} = -0.16$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.42$) and Level 3 ($\rho_{precision} = -0.21$, $\rho_{recall} = -0.45$, $\rho_{jaccard} = -0.45$) exhibit highly significant negative moderate correlation for the metrics Jaccard Index and Recall. However, Precision does not reach statistical significance at either level. Level 4 ($\rho_{precision} = 0.06$, $\rho_{recall} = -0.30$, $\rho_{jaccard} = -0.26$) shows a significant negative correlation, again with precision remaining non-significant. Overall, stronger effects emerge at intermediate levels, primarily driven by Recall and the Jaccard Index rather than Precision.

Cyclomatic Complexity For Cyclomatic complexity, Level 1 ($\rho_{precision} = -0.25$, $\rho_{recall} = -0.46$, $\rho_{jaccard} = -0.46$) demonstrates a significant negative low to moderate correlation across the metrics. Level 2 ($\rho_{precision} = -0.53$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.48$) shows a highly significant negative moderate correlation, indicating the strongest and most consistent effect across levels. Level 3 ($\rho_{precision} = -0.06$, $\rho_{recall} = -0.38$, $\rho_{jaccard} = -0.37$) also presents a highly significant negative low correlation overall, although precision does not reach statistical significance. At Level 4 ($\rho_{precision} = 0.02$, $\rho_{recall} = -0.23$, $\rho_{jaccard} = -0.18$), only Recall shows a significantly low negative correlation, while the other metrics do not demonstrate statistical significance. Overall, the results suggest more consistent and stronger effects at lower and intermediate levels compared to Level 4.

Summary In summary, although LOC and Cyclomatic complexity exhibit significant correlations when considering the overall dataset, these relationships are not consistently maintained across individual levels and prompting strategies. The significance and strength of the correlations vary depending on the specific level and prompting type, indicating that the observed effects are context-dependent rather than uniformly robust.

4.6.6 Basic Block Exit

This section focuses on the task of basic block exit identification. The Figure 4.15 shows the distribution of the data for the three metrics with LOC and Cyclomatic complexity. The plots show a negative slope across all the evaluation metrics. The Table 4.17 shows the significant results obtained per prompting type and Level.

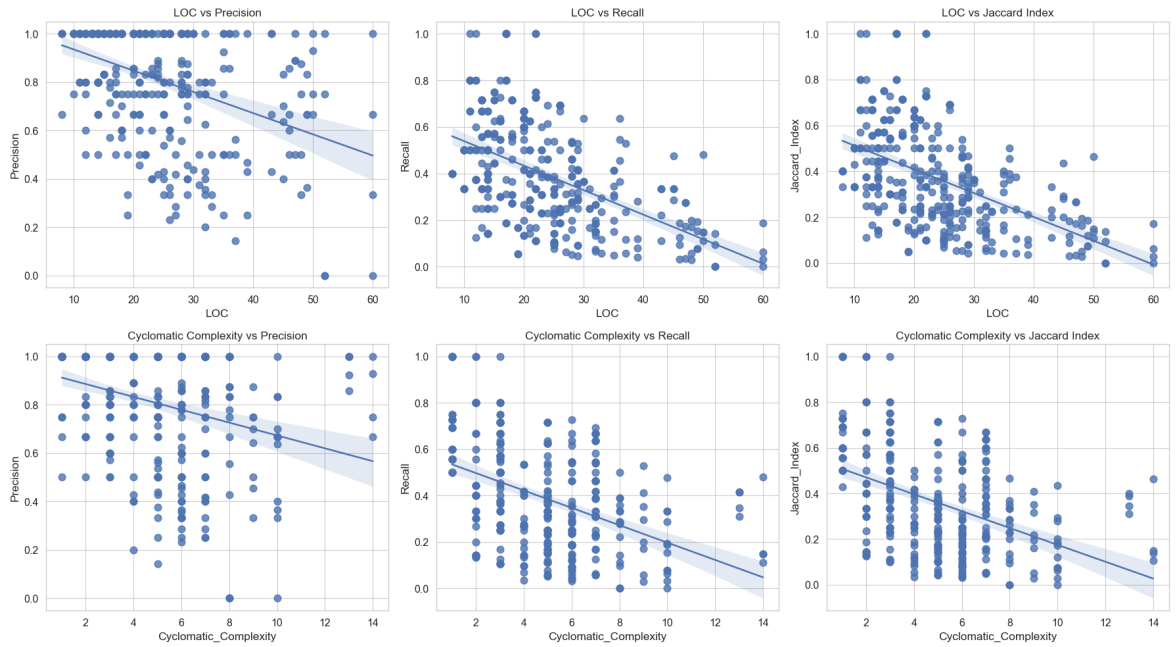

Figure 4.15: Basic Block Exit Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC For all three metrics, Precision ($\rho = -0.41$, p-value= 7.229e−15), Recall($\rho = -0.52$, p-value= 4.057e−24), and Jaccard Index ($\rho = -0.54$, p-value= 2.707e−26), we found the results to be highly significant. This shows that the LOC has a negative medium correlation on the three metrics. Increase in LOC, we see a reduction in results from the LLM.

Cyclomatic Complexity For all three metrics, Precision ($\rho = -0.34$, p-value= 2.371e−10), Recall($\rho = -0.43$, p-value= 2.592e−16), and Jaccard Index ($\rho = -0.45$, p-value= 8.160e−18), we found the results to be highly significant. This shows that the cyclomatic complexity has a low negative correlation with the three metrics. An increase in cyclomatic complexity results in a reduction in results from the LLM.

Per Prompting Type

LOC For all 4 promoting types, we observe highly significant Spearman correlations. CoT ($\rho_{precision} = -0.42$, Recall: $\rho = -0.49$, Jaccard Index: $\rho = -0.51$) and One-shot ($\rho_{precision} =$

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
				Global			
	LOC	−0.413	−0.518	−0.540	7.229e−15	4.057e−24	2.707e−26
	Cyclomatic	−0.343	−0.432	−0.451	2.371e−10	2.592e−16	8.160e−18
				By Prompting Type			
CoT	LOC	−0.416	−0.494	−0.507	8.253e−05	2.085e−06	1.171e−06
	Cyclomatic	−0.394	−0.385	−0.406	1.847e−04	2.558e−04	1.188e−04
One-Shot	LOC	−0.350	−0.488	−0.517	9.653e−04	2.811e−06	7.004e−07
	Cyclomatic	−0.272	−0.403	−0.434	1.118e−02	1.321e−04	3.749e−05
One-Shot CoT	LOC	−0.376	−0.571	−0.571	3.733e−04	2.581e−08	2.581e−08
	Cyclomatic	–	−0.485	−0.460	–	3.205e−06	1.091e−05
Zero-Shot	LOC	−0.487	−0.539	−0.574	2.811e−06	1.925e−07	2.543e−08
	Cyclomatic	−0.501	−0.474	−0.506	1.409e−06	5.248e−06	1.171e−06
				By Level			
Level 1	LOC	−0.412	−0.299	−0.343	9.717e−05	5.104e−03	1.240e−03
	Cyclomatic	−0.390	−0.546	−0.569	2.133e−04	1.358e−07	2.802e−08
Level 2	LOC	−0.514	−0.625	−0.654	7.923e−07	4.651e−10	4.519e−11
	Cyclomatic	−0.484	−0.452	−0.504	3.253e−06	1.621e−05	1.236e−06
Level 3	LOC	−0.543	−0.598	−0.634	1.540e−07	3.971e−09	2.371e−10
	Cyclomatic	−0.398	−0.506	−0.528	1.634e−04	1.171e−06	3.828e−07
Level 4	LOC	−0.317	−0.410	−0.424	2.977e−03	1.039e−04	5.846e−05
	Cyclomatic		–			not significant	

Table 4.17: Spearman Correlations between Complexity Metrics and Performance Metrics for the Basic Block Exit Data

-0.35, Recall: $\rho = -0.49$, Jaccard Index: $\rho = -0.52$) show low negative correlation form Precision and Recall. Negative Moderate effect can be observed for the metric Jaccard Index. This indicates LOC has a low to moderate effect on the results from the LLM on this task.

For One-shot CoT ($\rho_{precision} = -0.38$, $\rho_{recall} = -0.57$, $\rho_{jaccard} = -0.57$) and Zero-shot ($\rho_{precision} = -0.49$, $\rho_{recall} = -0.54$, $\rho_{jaccard} = -0.57$) shows moderate negative correlation except for the metric Precision in One-shot CoT, there we observe low negative correlation.

Cyclomatic Complexity Cyclomatic complexity on all prompting types is highly significant. For metric precision, in One-shot ($\rho_{precision} = -0.27$, $\rho_{recall} = -0.40$, $\rho_{jaccard} = -0.43$) and One-shot CoT ($\rho_{precision} = -0.19$, $\rho_{recall} = -0.48$, $\rho_{jaccard} = -0.46$) shows only significant results for precision and we observe low negative effect for this metric. For the metrics, the Jaccard Index and Recall have a moderate negative effect. CoT ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.39$, $\rho_{jaccard} = -0.41$) showed low negative effect on Precision and Recall. With the Jaccard Index at, we observe a moderate effect. For zero-shot ($\rho_{precision} = -0.50$, $\rho_{recall} = -0.47$, $\rho_{jaccard} = -0.51$), we observe a negative moderate effect on all three metrics. These results indicate that an increase in cyclomatic complexity indicates a reduction in results from the LLM on this task.

Per Level

LOC LOC shows highly significant results across all four levels. For Level 1 ($\rho_{precision} = -0.41$,$\rho_{recall} = -0.30$, $\rho_{jaccard} = -0.34$), we observe a low to moderate negative effect across the metrics. Level 2 ($\rho_{precision} = -0.51$, $\rho_{recall} = -0.62$, $\rho_{jaccard} = -0.65$) and Level 3 ($\rho_{precision} = -0.54$, $\rho_{recall} = -0.60$, $\rho_{jaccard} = -0.63$) show moderate negative effects for all three metrics, with Recall and Jaccard Index demonstrating stronger correlations. For Level 4 ($\rho_{precision} = -0.32$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.42$), the results remain highly significant with a negative moderate effect overall, while Precision shows only significant results and indicates a low negative effect. These findings suggest that an increase in LOC is associated with a decrease in the performance of the LLM across Precision, Recall, and Jaccard Index.

Cyclomatic Complexity Cyclomatic complexity shows highly significant results for Levels 1, 2, and 3, while for Level 3, precision is only significant. Level 4 does not show significant results. For Level 1 ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.55$, $\rho_{jaccard} = -0.57$), we observe a low negative effect on Precision and a moderate negative effect on Recall and Jaccard Index. Level 2 ($\rho_{precision} = -0.48$, $\rho_{recall} = -0.45$, $\rho_{jaccard} = -0.50$) demonstrates moderate negative effects across all three metrics. For Level 3 ($\rho_{precision} = -0.40$, $\rho_{recall} = -0.51$, $\rho_{jaccard} = -0.53$), Precision shows a low negative effect, while Recall and Jaccard Index exhibit moderate negative effects. Level 4 shows no statistically significant results. These findings indicate that higher cyclomatic complexity is generally associated with reduced LLM performance, particularly for Recall and Jaccard Index. Except for Level 4

Summary In summary, both Cyclomatic complexity and LOC are observed to have a highly significant effect on the results for the three metrics. This effect is consistently observed for all the levels and prompting types. The effect on the metrics has varied from a low to a medium negative effect. This indicates that as both Cyclomatic complexity and LOC increase individually, there is a reduction in results from the LLM on all three metrics, with the only exception of Cyclomatic complexity on Level 4 snippets.

4.7 Research Question 5

This section focuses on the research question: *How does the complexity of code measured by different metrics affect the precision of the LLM on the task of def-use chains creation with snippets with increasing complexity of data flow in various prompting?*. To do so, we observe the data for the creation of task-def-use chains. The Figure 4.16 shows the distribution of the data in a scatter plot with a negative slope. For all the plots, we observe a negative slope. The Table 4.18 contains the statistically significant results obtained for this task.

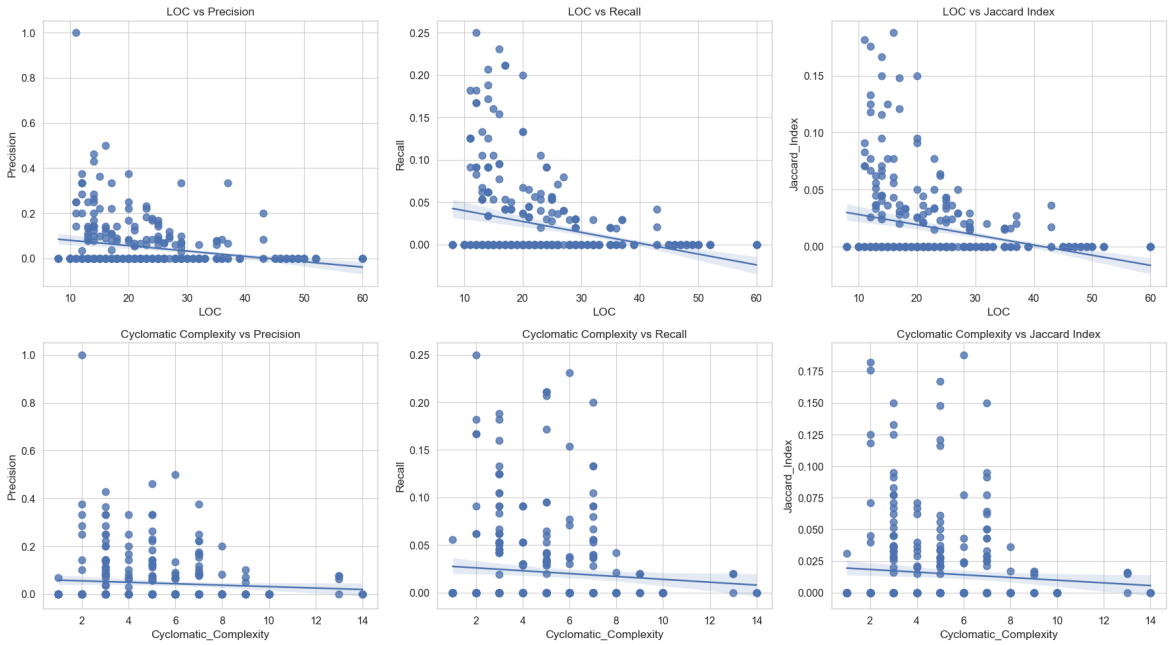

Figure 4.16: Def-Use Chains Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC The global correlation analysis revealed weak but statistically significant negative monotonic relationships between LOC and all evaluated performance metrics. Specifically, LOC exhibited a correlation of $\rho = -0.26$ with Precision (p-value= 1.269e−05), $\rho = -0.28$ with Recall(p-value= 2.456e−06), and $\rho = -0.28$ with the Jaccard index (p-value= 2.456e−06). These results indicate that as LOC increases, the performance of the LLM decreases across all evaluation measures. Although the effect size is weak, the consistency across metrics suggests that code length contributes to performance degradation.

Cyclomatic Complexity The global correlation analysis between cyclomatic complexity and each of the result metrics across the entire data set showed extremely weak, non-significant relationships ($\rho_{precision} = -0.013$, $\rho_{recall} = -0.025$, $\rho_{jaccard} = -0.023$). This informs us that on a global scale, cyclomatic complexity does not have an effect on the performance of the LLM on the task of def-use chains identification.

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	−0.261	−0.283	−0.280	1.269e−05	2.457e−06	2.457e−06
	Cyclomatic		–			not significant	
			By Prompting Type (LOC only – Cyclomatic not significant)				
CoT	LOC	−0.264	−0.282	−0.278	2.657e−02	1.989e−02	2.070e−02
One-Shot	LOC	−0.288	−0.295	−0.295	1.844e−02	1.760e−02	1.760e−02
One-Shot CoT	LOC	−0.274	−0.320	−0.317	2.221e−02	1.077e−02	1.077e−02
Zero-Shot	LOC	–	−0.246	−0.241	–	4.043e−02	4.376e−02
			By Level				
Level 1	LOC		–			not significant	
	Cyclomatic	+0.372	+0.381	+0.377	2.147e−03	1.917e−03	2.009e−03
Level 2	LOC	−0.281	−0.336	−0.326	1.989e−02	7.360e−03	9.603e−03
	Cyclomatic		–			not significant	
Level 3	LOC	−0.386	−0.415	−0.414	1.868e−03	6.614e−04	6.614e−04
	Cyclomatic	−0.254	−0.292	−0.286	3.387e−02	1.760e−02	1.844e−02
Level 4	LOC	−0.266	−0.292	−0.291	2.657e−02	1.760e−02	1.760e−02
	Cyclomatic		–			not significant	

Table 4.18: Spearman Correlations between Complexity Metrics and Performance Metrics for the Def-Use Chains Data

Per Prompting Type

LOC The effect of the prompting strategy is observed across different types. For CoT ($\rho_{precision} = -0.26$, $\rho_{recall} = -0.28$, $\rho_{jaccard} = -0.28$), One-shot ($\rho_{precision} = -0.29$, $\rho_{recall} = -0.30$, $\rho_{jaccard} = -0.30$), One-shot CoT ($\rho_{precision} = -0.27$, $\rho_{recall} = -0.32$, $\rho_{jaccard} = -0.32$), and Zero-shot ($\rho_{precision} = -0.23$, $\rho_{recall} = -0.25$, $\rho_{jaccard} = -0.24$), we observe a significant weak negative correlation across most metrics. The exception is Precision in the zero-shot setting, which shows slightly weaker correlation but still trends negative.

Overall, we can conclude that a weak negative correlation exists between prompting strategies, suggesting that changes in model prompting strategy tend to be associated with

small decreases in Precision, Recall, and Jaccard Index. CoT and One-shot strategies exhibit slightly stronger negative correlations than Zero-shot, suggesting that more structured prompts may further amplify this trend.

Cyclomatic Complexity The effect of the prompting strategy is observed across different types. For CoT, One-shot, One-shot CoT, and Zero-shot, no statistically significant correlations were observed for any metric. Hence, we can conclude that there is no clear relationship between the prompting strategy and the evaluation metrics in this case. Both positive and negative trends are very weak, indicating that changes in prompting style do not significantly affect Precision, Recall, or Jaccard Index for this dataset.

Per Level

LOC The effect of LOC is observed per level. For Level 1 ($\rho_{precision} = -0.06$, $\rho_{recall} = -0.05$, $\rho_{jaccard} = -0.06$), it is found that there does not exist a significant relation for this task. For Level 2($\rho_{precision} = -0.28$, $\rho_{recall} = -0.34$, $\rho_{jaccard} = -0.33$) and Level 4($\rho_{precision} = -0.27$, $\rho_{recall} = -0.29$, $\rho_{jaccard} = -0.29$) we observe weak negative correlation with statistically significance. Level 3 ($\rho_{precision} = -0.39$, $\rho_{recall} = -0.41$, $\rho_{jaccard} = -0.41$) was observed to have moderate correlation and high statistical significance.

Overall, we can conclude that there is a negative correlation as the data flow complexity increases across the levels. For Level 1, the simpler data flow complexity shows no significance, and for Level 3, we observe a peak in negative correlation.

Cyclomatic Complexity Cyclomatic complexity does not show a consistent effect on the Levels of data flow complexity. For Level 1 ($\rho_{precision} = 0.37$, $\rho_{recall} = 0.38$, $\rho_{jaccard} = 0.38$), we observe high statistical significance and a positive low-moderate relationship. This tells us that, for Level 1, results improve across all metrics with an increase in lines of code. For Level 2 ($\rho_{precision} = -0.16$, $\rho_{recall} = -0.17$, $\rho_{jaccard} = -0.16$) and Level 4 ($\rho_{precision} = -0.04$, $\rho_{recall} = -0.06$, $\rho_{jaccard} = -0.06$) we do not observe any statistically significant relationship. For Level 3, we observe a weak, statistically significant negative correlation. This tells us that for Level 3 ($\rho_{precision} = -0.25$, $\rho_{recall} = -0.29$, $\rho_{jaccard} = -0.29$), the longer the code performance of the LLM reduces.

Summary Overall, code complexity affects LLM performance in def-use chain identification in a subtle way. LOC consistently shows negative correlations with precision, recall, and Jaccard index, especially at higher complexity levels, with the strongest effect at Level 3. Globally and across prompting strategies, longer code tends to reduce performance, indicating that code length is a reliable predictor of decreased LLM effectiveness. Cyclomatic complexity, on the other hand, shows weaker and less consistent effects. Globally and for most prompting strategies, correlations are non-significant, though Level 1 shows slightly positive correlations and Level 3 exhibits weak negative correlations. Prompting strategy modulates these trends slightly, with structured prompts (CoT, One-shot) showing marginally stronger negative correlations. Overall, the results suggest that while longer code generally impairs performance, the impact of control flow complexity is smaller and context-dependent.

4.8 Research Question 6

This section aims to answer the research question *How does the complexity of code measured by different metrics affect the precision of the LLM on snippet output identification and program slicing tasks with snippets with increasing complexity of data flow in various prompting?*. It focuses on the two tasks, Program output and Program slice.

4.8.1 Program Output

The below results are specific to the task of program output. The conclusions are drawn specifically from the data obtained from the LLM on the three evaluation metrics(Jaccard, index, Precision and Recall). The Figure 4.17 shows the evaluation metrics with code complexity metrics. We observe a positive slope for LOC and a negative slope for Cyclomatic complexity. The Table 4.19 contains the significant results obtained for this metric.

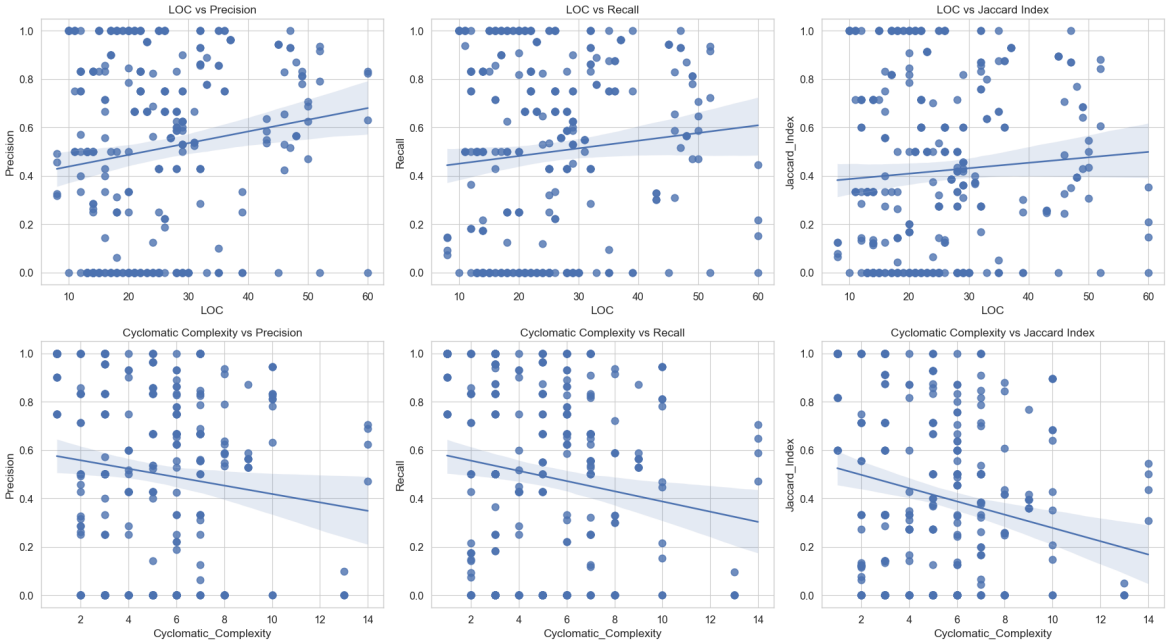

Figure 4.17: Program Output Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC The LOC metric showed no statistical significant correlation on Precision ($\rho = 0.09$), Recall ($\rho = 0.05$) and Jaccard Index ($\rho = 0.06$). This indicates that, on the global data the code complexity metric, LOC has no affect on the results obtained from the LLM for this task.

Cyclomatic Complexity Cyclomatic complexity shows statistically significant correlation for two of the three evaluation metrics, Recall($\rho = -0.17$, p = 4.165e−02) and Jaccard

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC		–			not significant	
	Cyclomatic	–	−0.168	−0.181	–	4.165e−02	3.581e−02
			By Prompting Type & Level – no significant correlations				

Table 4.19: Spearman Correlations between Complexity Metrics and Performance Metrics for the Program Output Data

Index($\rho = -0.18$, p =3.581e−02). We observe low negative correlation on the complete data. The metric, Precision ($\rho = -0.15$, p = 9.570e−02) shows no statistical significance. This indicates that, on the global data without specifying snippet level or prompting type, the results in the metrics Recall and Jaccard Index decrease slightly with increasing cyclomatic complexity, reflecting a low negative correlation.

Per Prompting type Neither of the metrics, LOC and Cyclomatic complexity show any significant correlation with any of the prompting types. This signifies that there is no effect of LOC and Cyclomatic complexity on the results from LLM per prompting type on this task.

Per Level Neither of the metrics, LOC and Cyclomatic complexity show any significant correlation with any of the levels of defined data-flow complexity. This signifies that there is no effect of LOC and Cyclomatic complexity on the results from LLM per level of data-flow complexity on this task.

Summary In summary for the task of Program output, the two code complexity metrics show low to no correlation with the results from the LLM. Low negative correlation is observed for Jaccard Index and Recall for the complexity metric, Cyclomatic complexity. Across prompting types and complexity levels, no significant effects were observed, indicating that LLM performance on program output identification is largely unaffected by the two code-complexity metrics.

4.8.2 Program Slice

The below results are specific to the task of program slice. The conclusions are drawn specifically from the data obtained from the LLM. The Figure 4.18 shows the distribution of the data for this task in the three evaluation metrics and code complexity metrics. We observe negative slope all plots except for precision with Cyclomatic complexity. The Table 4.20 contains the significant results obtained for this task.

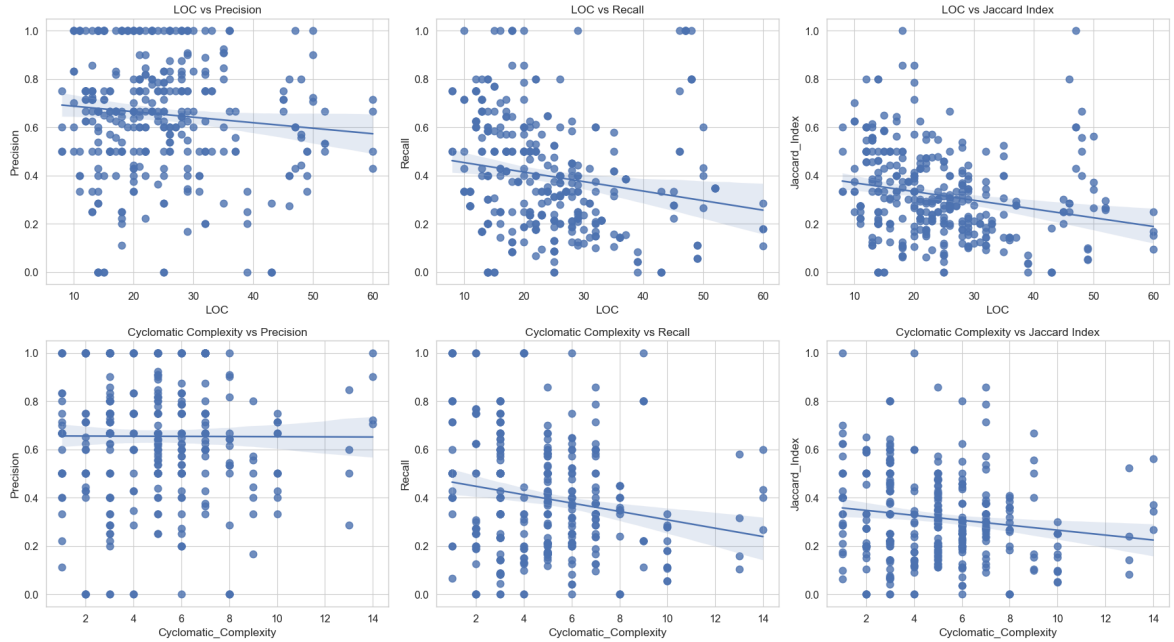

Figure 4.18: Program Slice Data Distribution on the Three Metrics against the Code Complexity Metrics

Global

LOC For LOC, we observe for Jaccard Index ($\rho = -0.29$, p = 5.343e−07) and Recall ($\rho = -0.29$, p = 5.343e−07) negative high significant correlation. However for the metric, Precision, we do not observe any significant relationship

Cyclomatic Complexity For Cyclomatic complexity, we observe similar results as LOC. For Jaccard Index ($\rho = -0.14$, p-value = 3.167e−02) and Recall ($\rho = -0.17$, p-value = 3.167e−02) negative high significant correlation. However for the metric, Precision, we do not observe any significant relationship.

Per Prompting Type

LOC Significant negative correlations with LOC are observed for Recall and Jaccard Index across CoT ($\rho_{recall} = -0.308$, $\rho_{jaccard} = -0.338$), One-shot ($\rho_{recall} = -0.308$, $\rho_{jaccard} = -0.293$), and One-shot CoT ($\rho_{recall} = -0.330$, $\rho_{jaccard} = -0.338$). Zero-shot yields no significant correlations for any metric. Precision is non-significant across all prompting types. The consistency of the Recall and Jaccard effects across three of four prompting strategies suggests that code length has a reliable, if modest, negative impact on these metrics regardless of prompting approach.

Cyclomatic Complexity No significant correlations between Cyclomatic complexity and any performance metric were observed for any prompting type. This is consistent with

		ρ			p_{adj}		
Group	**Complexity**	**Precision**	**Recall**	**Jaccard**	**Precision**	**Recall**	**Jaccard**
			Global				
	LOC	–	−0.295	−0.294	–	5.343e−07	5.343e−07
	Cyclomatic	–	−0.172	−0.139	–	1.003e−02	3.167e−02
			By Prompting Type (LOC only – Cyclomatic not significant)				
CoT	LOC	–	−0.308	−0.338	–	1.909e−02	1.003e−02
One-Shot	LOC	–	−0.308	−0.293	–	1.909e−02	2.307e−02
One-Shot CoT	LOC	–	−0.330	−0.338	–	1.140e−02	1.003e−02
Zero-Shot	*no significant correlations*						
			By Level (LOC only – Cyclomatic not significant)				
Level 1	LOC	–	−0.405	−0.411	–	1.222e−03	1.222e−03
Level 2	LOC	−0.300	–	−0.286	2.245e−02	–	2.703e−02
Level 3	LOC	–	−0.294	–	–	2.307e−02	–
Level 4	*no significant correlations*						

Table 4.20: Spearman Correlations between Complexity Metrics and Performance Metrics for the Program Slice Data

the global finding that Cyclomatic complexity has a limited and inconsistent effect on LLM performance for this task.

Per Level

LOC The effect of LOC varies across levels. Level 1 shows a significant negative correlation for Recall ($\rho = -0.405$, $p = 1.222\text{e}{-03}$) and Jaccard Index ($\rho = -0.411$, $p = 1.222\text{e}{-03}$), representing the strongest level-wise effect observed. Level 2 shows a partial effect, with significance for Precision ($\rho = -0.300$) and Jaccard Index ($\rho = -0.286$) but not Recall. Level 3 shows significance only for Recall ($\rho = -0.294$). Level 4 yields no significant correlations.

Cyclomatic Complexity No significant correlations between Cyclomatic complexity and any performance metric were observed at any level. This further confirms that control flow complexity has no meaningful effect on LLM performance for the program slicing task.

Summary Overall, code complexity has a limited effect on LLM performance for the task of program slicing. LOC shows weak but consistent negative correlations globally and across most prompting types for Recall and Jaccard Index, with the strongest effect observed at Level 1. Precision is largely unaffected by LOC across all prompting types and levels. Cyclomatic complexity shows marginal global significance for Recall and Jaccard Index but yields no significant effects at the prompting type or level subgroup level, suggesting its global effect is too diffuse to be captured in finer-grained analyses.

5 Discussion

5.1 RQ1: Identification of Foundation Elements for Def-Use Chains

5.1.1 Defs, Uses and Function Calls

For defs, uses, and function call identification, the prompting strategy had no influence overall. A notable exception was Precision for defs identification, where One-shot CoT significantly outperformed CoT with a moderate effect size, suggesting that combining an example with CoT reasoning provided a meaningful advantage for this specific task.

Regarding complexity levels, lower levels often outperformed higher ones across the three tasks, though the pairs were not consistent across tasks, indicating that the effect of complexity was task-dependent rather than systematic.

These results suggest that for foundational identification tasks, neither prompting strategy nor complexity level acted as a consistent performance driver. This is likely attributable to the nature of the tasks themselves; since defs, uses, and function calls are identified per line of code, the LLM may be implicitly reasoning about the broader data flow of the entire snippet even when only asked to perform a local identification. In a practical scenario, consider a developer using an LLM to identify all variable definitions in a complex snippet; the model may correctly identify definitions on simpler lines while conflating or missing others where data flow interactions across the snippet introduce ambiguity. This implies that even for seemingly straightforward identification tasks, the surrounding data flow context of a snippet influences model behavior in ways that neither prompting strategy nor complexity level alone can reliably control.

For developers and code learners, One-shot CoT prompting is the most reliable choice for defs identification, where precision is important, while for uses and function calls, the prompting strategy has little practical impact on performance. Across all three tasks, the LLM should not be fully trusted to identify these elements in higher complexity snippets, as increasing data flow complexity consistently leads to more false negatives and reduced recall. For researchers, the consistent degradation across levels confirms that data flow complexity meaningfully impacts LLM performance on these tasks, and future work should investigate whether model-level improvements or targeted training on higher complexity snippets can reduce false negative rates.

5.1.2 Basic Blocks, Block Entry and Block Exit

For basic block identification, One-shot and One-shot CoT consistently outperformed Zero-shot and CoT across all three metrics, with effect sizes ranging from low to moderate. Block entry showed the strongest level effect, with Level 1 differing significantly from all other levels on Jaccard Index and Recall, while no significant differences were observed among Levels 2, 3, and 4. For block entry and exit, Precision did not show significant pairs across prompting types, likely reflecting high TP and fewer FP across all strategies.

The consistent advantage of One-shot-based strategies for basic blocks suggests that the LLM benefits from a concrete example when identifying structural boundaries in code, as recognizing basic blocks requires understanding the overall control flow structure of a snippet rather than reasoning about individual lines in isolation. The fact that Level 1 differed significantly from all higher levels, while Levels 2, 3, and 4 performed similarly, suggests that even a modest increase in complexity was sufficient to degrade LLM performance on block entry identification, beyond which further complexity had little additional impact. The conservative prediction behavior for block entry and exit further suggests that the model becomes increasingly uncertain as complexity grows, defaulting to under-prediction rather than over-prediction.

For developers and code learners, one-shot-based prompting strategies should be preferred for basic block identification, as they consistently outperform Zero-shot and CoT across all three metrics. The results also suggest that the LLM is generally conservative in its predictions for block entry and exit, generating few false positives but frequently missing true positives, and this miss rate worsens with increasing data flow complexity. Developers should therefore exercise caution when relying on LLMs to identify control flow elements in higher complexity code. For researchers, the strong level effect observed for block entry, where Level 1 differs significantly from all other levels, warrants further investigation into whether the difficulty of identifying block entry points scales differently with complexity compared to basic blocks and block exit.

5.2 RQ2: Def-Use Chains

As shown in Chapter 4, for the task of def-use chains, no significant pairs were observed per prompting type or per level. Across all levels and prompting types, results were consistently low with many outliers, with a maximum Recall of 0.25 and a maximum Jaccard Index of 0.19, as seen in Table 2.1.

The high rate of missed TPs and substantial FPs throughout suggests that def-use chain construction placed considerable demand on the LLM, exceeding its ability to reliably track variable definitions and their propagation across a snippet. The near absence of results at Level 4 further indicates that the model largely failed to engage with higher complexity data flow, rather than producing partially correct outputs. While One-shot and One-shot CoT strategies appeared to yield some results at higher complexity levels, the lack of statistical significance suggests that the presence of an example provided limited but insufficient scaffolding for the model to reason about complex data dependencies.

These results suggest a fundamental gap in the ability of the LLM to reason about data flow. The model neither consistently focuses on a given entity to produce a complete def-use chain, nor does it reliably produce a correct but partial chain. Developers and code learners should therefore exercise caution when relying on LLMs to understand or trace data flow in code. For researchers, these findings highlight the need for models to place greater emphasis on data flow reasoning during training or architectural design.

To better understand the effect of data flow complexity and prompting strategies, future studies should consider using snippets with even lower complexity levels and constraining the LLM to one or two entities when constructing def-use chains. This would help isolate whether the LLM is capable of producing def-use chains under simplified conditions. At a deeper level, improving model architecture with a stronger focus on data flow graph construction for code could meaningfully improve performance on such tasks.

5.3 RQ3: Program Output and Program Slice

From the results in RQ3, we observed that prompting types do not yield any significant pairs for either the task of program output or program slice. For program output, significant differences are observed across multiple levels, with Level 1 consistently showing significance against higher levels across all three metrics, although the specific significant pairs and effect sizes vary. For the program slice, only one significant pair was observed: Level 2 (M = 0, $\mu = 0.19$) with Level 3 (M = 0, $\mu = 0.15$) on the Jaccard Index metric, with a low effect size.

For both tasks, prompting strategies did not demonstrate any meaningful improvement in LLM performance, suggesting that, regardless of the prompting approach, the LLM performs at a consistent baseline. The multiple significant pairs observed across levels for the program output task are particularly meaningful, as they indicate that increasing data flow complexity genuinely affects LLM performance: code snippets with higher data flow complexity are more likely to challenge the LLM in correctly identifying the output. For the task of program slicing, only one significant pair was observed, which is likely attributable to the selection of the backward slice point. The line of code chosen for performing the backward program slice was selected randomly and did not follow an increasing data flow complexity pattern, even though the snippets themselves did. This inconsistency likely dampened the effect of complexity on the results, explaining the limited significance observed.

These results suggest that neither developers nor code learners can fully rely on LLMs to correctly predict program outputs or identify the relevant lines affecting an entity at a given point in code. For researchers, the program slicing task warrants further investigation with a controlled increase in data flow complexity at the slice point to better understand LLM behavior. Improving LLM performance on these tasks would likely require incorporating program output and program slicing examples into model training or model architecture design.

5.4 RQ4: Complexity Metrics with Foundational Tasks

This section focuses on results obtained for research question 4. The results are again grouped into two subsections. The results of defs, uses, and function calls are discussed together. Similarly, for the tasks of basic block entry and basic block exit. as they were prompted together for the LLM.

5.4.1 Defs, Uses and Function Calls

LOC showed a consistent negative effect across defs, uses, and function call identification, indicating that longer code snippets systematically reduced model performance on these tasks. Cyclomatic Complexity, by contrast, showed little to no significant effect across the three tasks, with only isolated exceptions at specific levels and prompting types, likely attributable to dataset variability rather than a genuine relationship.

The dominance of LOC over Cyclomatic Complexity as a performance predictor suggests that the LLM is more sensitive to code length as a surface-level feature than to the structural complexity of the underlying data flow. This implies that the model tends to process these tasks locally, at the line level, rather than developing a holistic understanding of how data is defined and used across the entire snippet. As code grows longer, the model appears to lose track of relevant entities, not because the control flow becomes more complex, but simply because more lines of code increase the cognitive demand on the model.

The inconsistent results for function calls across levels and prompting types are likely attributable to the uneven distribution of function calls across snippets, which introduces noise that obscures the effect of complexity. For developers and code learners, CoT based prompting appears to be a more stable choice for function call identification in longer code. For researchers, these findings highlight that LOC may be a more reliable predictor of performance degradation in foundational tasks than structural complexity metrics, which warrants further investigation into how models represent and process data flow at different levels of code length.

5.4.2 Basic Blocks, Block Entry and Block Exit

LOC showed a consistent low negative correlation across all three tasks globally, with its effect varying in strength across levels and prompting types. Cyclomatic Complexity showed a more selective effect, appearing most consistently for block exit while remaining largely absent for basic blocks and block entry.

For basic blocks and block entry, the effect of LOC was primarily reflected in Recall and Jaccard Index rather than Precision, suggesting that longer code led to missed predictions rather than incorrect ones. Block exit was the most complexity-sensitive of the three tasks, with both LOC and Cyclomatic Complexity showing consistent negative correlations globally and across most prompting types and levels, indicating that longer and more structurally complex code caused the model to both miss correct exit points and generate incorrect ones.

The consistent significance of both LOC and Cyclomatic Complexity for block exit, but not for basic blocks and block entry, highlights that the nature of the specific control flow element being identified plays an important role in how complexity affects LLM performance. This suggests that the LLM does not process all control flow elements uniformly; exit points, being inherently tied to branching and control flow transitions, appear to place greater demand on the model as both code length and structural complexity grow. More broadly, these findings indicate that the ability of the LLM to comprehend control flow structure degrades in a task-specific manner, implying that a general measure of code complexity is insufficient to predict where and how LLM performance will break down. For developers and code learners, One-shot and One-shot CoT prompting strategies yield better overall performance for block exit identification, though their results are also more sensitive to increases in code length.

5.5 RQ5: Complexity Metrics with Def-Use Chains Creation

For the task of def-use chain creation in comparison with code complexity metrics, LOC shows a significant negative effect globally and consistently across all prompting types and levels, indicating that longer code snippets reduce the correctness of the model's output regardless of the prompting strategy used. The effect varies in strength across levels and prompting types, with Level 3 showing the strongest association and Zero-shot being the least sensitive strategy to increases in code length. This suggests that for longer snippets, Zero-shot prompting may be a preferable choice for developers and code learners.

Cyclomatic complexity shows an inconsistent pattern across levels, with a small positive effect at Level 1, no significant effect at Levels 2 and 4, and a weak negative effect at Level 3, and no significant relationship across prompting types. The overall low cyclomatic complexity of the dataset (M = 5) is likely a contributing factor to these erratic results, as there is insufficient variation in control flow complexity to produce stable and meaningful correlations.

The consistent negative effect of LOC suggests that developers and code learners should be cautious when asking LLMs to construct def-use chains for longer code snippets, with Zero-shot prompting being a marginally more reliable option in such cases. For researchers, future studies should incorporate snippets with higher cyclomatic complexity alongside increased data flow complexity, as the current dataset's low control flow variation limits the ability to draw strong conclusions about the role of structural complexity in this task.

5.6 RQ6: Complexity Metrics with Program Output and Program Slice Tasks

For the task of program output prediction, code complexity showed minimal impact on LLM performance. LOC shows no significant correlation with any evaluation metric, and

while cyclomatic complexity shows a small negative correlation with Recall and Jaccard Index globally, this effect is weak and does not persist across prompting types or levels. Although both LOC and cyclomatic complexity increase from Level 1 to Level 4, this does not correspond to any consistent change in performance.

For program slicing, code complexity showed a limited but slightly more notable effect. LOC shows weak negative correlations with Recall and Jaccard Index globally and across most prompting types, suggesting that longer code moderately impairs the ability of the model to identify correct slices, though Precision remains unaffected. Cyclomatic complexity shows no significant relationship with any metric across prompting types or levels, reinforcing that control flow structure plays no meaningful role in this task, which is expected given that program slicing is fundamentally a data dependency problem. The non-monotonic performance pattern across levels, where Level 2 outperforms both Level 1 and higher levels, further suggests that the defined difficulty levels do not correspond to a linear increase in challenge for this task.

Across both tasks, structural complexity metrics show limited explanatory power over LLM performance. Developers and code learners should be aware that while longer snippets may slightly reduce the reliability of LLM-generated slices, complexity alone is not a strong predictor of failure. For researchers, future work should explore whether semantic complexity measures, such as the number of variable interactions or the depth of nested expressions, better capture the factors that challenge LLMs on these tasks.

5.7 Model Implications

The results across all research questions reflect the capabilities and limitations of the LLM, DeepSeek Coder 6.7B Instruct. The consistently low performance on tasks such as def-use chain construction and function call identification suggests that the model lacks sufficient capacity to reason deeply about data flow relationships in code. The strong and consistent negative effect of LOC across most tasks indicates that the ability of the model to process and reason over longer code snippets is limited, which is expected given the parameter size of the model. The absence of any meaningful prompting effect for most tasks suggests that the model has reached a performance ceiling that cannot be overcome through prompt engineering alone, and that improvements would require either a larger model with greater reasoning capacity or task-specific fine-tuning on data flow related tasks such as def-use chains, program slicing and output prediction or even a better improved model able to process Abstract Syntax Tree (AST) and DFG . The relatively better performance on structurally simpler tasks, such as block entry and exit identification, compared to semantically complex tasks, such as def-use chain creation, further suggests that the model is better equipped for pattern-based identification than for deeper semantic reasoning over data dependencies. Future work should evaluate larger or more recent code models on the same benchmark to determine whether the observed limitations are specific to DeepSeek Coder 6.7B Instruct or reflect a broader challenge for LLMs on data flow reasoning tasks.

6 Threats to Validity

This chapter focuses on the threats to the validity of this thesis. These threats are divided into internal, construct, and external validity.

6.1 Internal Validity

Internal validity concerns the internal factors that account for the effects observed in LLM results. They play a vital role in the validity of this thesis.

The selection of code snippets poses a significant threat to internal validity. Open-source datasets do not fully cover the concepts required for this thesis, and as Grattafiori et al. [15] notes, models such as LLaMA 3 have already been trained on many available open-source datasets, raising the risk that evaluation snippets may have been seen during training. To mitigate this threat, self-created snippets are used throughout this thesis. These snippets are designed to be as diverse as possible, covering different data structures, OOP properties, and other key concepts such as backtracking and recursion in varied combinations. Predefined algorithms are also included with self-created variations to reduce the likelihood that any snippet is part of the model's training data.

The definition of tasks within prompts poses a further threat to internal validity. Ambiguous or imprecise definitions of data analysis terms can cause the LLM to produce outputs that reflect prompt misinterpretation rather than a genuine failure of code comprehension, thereby conflating prompt quality with model capability. To address this, multiple alternative definitions were systematically tested, and those that minimized prompt-related errors were selected, ensuring that observed errors more accurately reflect the comprehension ability of the LLM.

Presenting all tasks within a single prompt is a threat to internal validity. Liu et al. [30] demonstrated that LLMs perform worse on longer prompts when relevant context appears neither at the beginning nor at the end, meaning that aggregating tasks risks degrading model performance independently of model capability. This effect was observed during initial testing. To mitigate this, the tasks are divided into five sequentially presented prompts, each focused on a related subset of the analysis, reducing cognitive overload.

6.2 Construct Validity

Construct validity refers to the degree to which the operationalization of the measures in the study that are used to represent the constructs in the real world [52]. To analyze LLM

code comprehension, it is possible to approach it from the perspectives of control flow graphs, error detection, code completion, and others. However, these perspectives do not dive into whether the LLM can comprehend the data and how it flows through the program snippets. To address this, data flow analysis concepts are adopted as the primary evaluation framework, enabling targeted assessment of the ability of the LLM to reason about data and its movement through program snippets.

The configuration of the temperature and top-p parameters poses a threat to construct validity. These hyperparameters directly influence the stochasticity of model outputs, so suboptimal settings can introduce variability that obscures the model's true deterministic capability. To obtain the most consistent and reproducible results, temperature is set to 0.05 and top-p to 0.8, allowing only minimal variation around the most probable output while still avoiding complete determinism.

The construction of ground-truth labels poses a threat to construct validity. Existing AST-based tools and program-analysis frameworks were evaluated, but did not reliably support all required tasks. Configuring multiple tools for the full dataset would have required substantial setup time. As a result, relying solely on automated tooling was not feasible, meaning that the ground truth may not fully capture all edge cases, and manual refinement introduces the risk of subjective interpretation. To mitigate this, ground truth labels for definitions, uses, function calls, and def–use chains were generated using custom scripts and subsequently verified through careful manual correction, combining the consistency of automated extraction with human oversight.

The post-processing of LLM outputs introduces a further threat to construct validity. Because LLM responses frequently contained malformed JSON or extraneous explanatory text between JSON contents, a cleaning step was necessary prior to evaluation. However, this step risks introducing interpretation bias when the intended output is ambiguous. This could affect how accurately the cleaned data represents the original LLM responses.

6.3 External Validity

The use of a single LLM threatens external validity, as findings derived from one model may not generalize to others with different architectures, training data, or parameter scales. To address this, the choice of model was based on a systematic comparison of performance on code-related benchmarks among available open-source models. DeepSeek Coder 6.7B Instruct was selected because it demonstrated superior accuracy on complex code tasks relative to comparable open-source alternatives at the time of the study. Nevertheless, these findings should be interpreted with caution, as newer models may exhibit substantially different comprehension capabilities, and replication across multiple models is recommended for future work.

The exclusive use of the Kotlin programming language is a threat to external validity, as language-specific constructs may influence LLM behavior in ways that do not generalize to other languages. For this study, we seek a language that supports a variety of data structures and OOP properties. However, specific constructs of the Kotlin language might affect the results from the LLM.

7

Related Work

LLMs are increasingly being used for code-related tasks. To understand how far LLMs can actually comprehend code to perform such code-related tasks, researchers have used program analysis concepts with different programming languages. They analyzed code semantic relationships like CFGs [24, 31, 45], and AST [23, 31], data flow analysis [23], and error detection analysis [45].

Ma et al. [31] designed their study into two semantic probing tasks, namely, semantic relationship predictor and semantic propagation prediction, while also focusing on the attention distribution in learning the code semantics. Semantic relationship predictor aims to recover significant semantic structures like CFGs, Control Dependency Graphs (CDGs), and Data Dependency Graphs (DDGs) from the representational space. Semantic propagation prediction observes long dependency relationships in the vector space. For example, a variable is defined initially but used at the end. From their experiments, the authors found that pre-trained models and LLMs can express code syntax quite well and capture code semantics to some extent. The study generalizes these results across three types of transformer architectures: encoder-only, decoder-only, and encoder-decoder. This paper provided insight into the code understandability of different model architectures. The code semantic structures, such as DDGs, CDGs, CFGs, and ASTs, are very relevant to this thesis. The paper showed how they extracted their relevant information from these graphs for the assessment of code understandability in LLMs. This work is directly relevant to this thesis as it demonstrated that semantic structures such as DDGs and CFGs are meaningful probes for assessing code understanding in LLMs, and established that data flow related semantics remain a weak point across model architectures. This motivated the focus of this thesis on data flow comprehension as an underexplored dimension of LLM code understanding.

Huang et al. [24] focused on correct CFG generation in error-prone data with their approach by asking relevant research questions. They used the CoT approach with four different steps. Namely, structure hierarchy extraction, nested code block extraction, and CFG generation of Nested code blocks' CFGs to access error-tolerant and understanding ability of pre-trained LLMs. This paper explores CFG to analyze LLMs' code understandability of code flow, but does not further investigate whether the LLM comprehends how the data flows in the program.

Huang et al. [23] addresses the Implicit Data Flow (IDF) problems, such as object sharing, callable object side effects, and the implicit data flows in comprehension observed in Python programming. They detected that LLMs tend to hallucinate, causing errors and uncertainty in their accurate IDF predictions. To address this, the authors designed a CoT-based five-step process (namely, Import statements completion, Comprehension transformation, Program slicing, Def-use extraction, and Def-use flow fusion) and adopted the principle of single

responsibility. The paper utilizes the def-use chains concept defined by Ken Kennedy. The evaluation metrics, including def-coverage, use-coverage, and DFG accuracy, are quite interesting for this thesis. This paper also highlights the importance of designing prompts and the overall framework of DFG-chain.

Wang et al. [45] is a novel data flow analysis framework to address the hallucinations of the LLM with bug detection. Interestingly, instead of the def-use chains concept, the authors use the source/sink extraction approach and then focus on CFGs and extracting data facts and path conditions from the programs. The authors relied on CFGs to examine how program value propagates via dataflow facts. LLMDFA evaluates four different LLMs and tests them against synthetic benchmarks as well as real-world programs. From these evaluations, the results demonstrate that LLMDFA outperforms standard benchmarks and shows sufficient evidence of its potential. The paper demonstrated that it is possible to define a different extraction formulation (source/sink) instead of the def-use chains approach for data flow analysis. Error detection can be a good assessment for the code understanding ability of the LLMs.

In summary, the reviewed literature explores how LLMs interact with different aspects of code understanding, from control flow to data flow and human cognition. They lacked in-depth exploration of programming concepts, such as OOPs and data structures, which can make code complex, and instead focused on predefined standard datasets. However, this raises the concern that researchers might have already trained LLMs on these readily available datasets.

8 Concluding Remarks

8.1 Conclusion

Comprehending code is a routine task for both developers and code learners. As the code base grows in complexity, developers rely on LLM tools like Copilot and OpenAI Codex to help them understand code. However, the precise limitations of code comprehensibility of the LLMs with regard to how data flows through the program remain unclear. This study investigated these limitations by evaluating the ability of an LLM to comprehend code, particularly in data flow analysis. The ability of the LLM is systematically evaluated on its performance on a range of tasks, like defs, uses, function calls, basic blocks, basic block entry and basic block exit identification, def-use chains creation, program slicing, and finally the program output.

To mitigate the risk of data contamination, the study employed self-created Kotlin code snippets of varying complexity rather than open-source datasets that may have been seen during model training. Research questions were formulated to evaluate LLM performance on these tasks under different prompting strategies and complexity levels, enabling a deeper examination of which prompting approaches are most effective and to what degree of data flow complexity an LLM can reliably reason.

The study further examined the relationship between traditional code complexity metrics and LLM precision, assessing whether statistically significant correlations exist across prompting strategies and complexity levels. Understanding this relationship is valuable not only for characterizing the current capabilities pf the LLM but also for guiding the design of more effective prompting strategies, curriculum learning approaches, and targeted model improvements. By identifying the complexity thresholds at which LLM performance begins to degrade, this analysis offers actionable insights for both model developers and tool designers seeking to enhance reliability in code comprehension tasks.

The results reveal that LLM performance varies considerably across tasks, prompting strategies, and complexity levels. For def-use chain construction, performance was consistently low across all evaluation metrics, prompting types, and complexity levels, indicating that this task presents a fundamental challenge for the model regardless of how the prompt is framed. For program output prediction, performance differed significantly across complexity levels but not across prompting strategies, suggesting that prompt variation alone has little effect on this task. A similar pattern emerged for program slicing, with no significant differences across prompting strategies, though one significant pair was identified across complexity levels.

With regard to code complexity metrics, LOC consistently negatively affected LLM performance across most tasks, suggesting that longer code places a systematic burden on the model regardless of task type. Cyclomatic complexity, by contrast, showed a non-linear and context-sensitive relationship with performance, with its effect varying across tasks, levels, and prompting types. This indicates that structural complexity alone does not reliably predict LLM performance degradation, and that the nature of the task mediates how complexity influences the model. Together, these findings suggest that LOC serves as a more consistent indicator of performance degradation, while cyclomatic complexity reflects the nuanced interplay between code structure and task demands.

8.2 Future Work

Several directions remain open for future investigation. First, the dataset could be expanded by incorporating additional code snippets that cover both the concepts already studied and those not yet addressed, such as predefined algorithms like searching, a broader range of sorting variants, and string-based snippets. This would increase the statistical power and generalizability of the findings. Once a sufficiently large dataset has been established, the methodology could be extended to other programming languages, enabling cross-language comparisons of LLM performance on equivalent data flow tasks.

As the dataset grows, future work could also explore more advanced prompting techniques, such as Few-shot, Multi-shot, or Context-aware CoT prompting, to assess whether these approaches yield measurable improvements in accuracy across varying levels of code complexity. Additionally, comparisons between multiple LLMs, and between LLMs and human participants would provide a richer picture of current model capabilities and their practical limitations.

Finally, the limitations identified in this study can directly inform improvements to LLM architectures and training procedures, particularly in areas where data flow reasoning and code execution proved most challenging. As models are refined through such targeted interventions, developers and code learners alike will be better positioned to rely on these tools with greater confidence for complex code understanding tasks.

a

Appendix

a.1 Descriptive statistics

a.1.1 Code complexity metrics

This subsection provides detailed descriptive statistics for the two code complexity metrics and for the results obtained for each task. The Table a.1 shows the level-wise and entity-wise data. The Overall statistics for both metrics are applied to each prompting type.

LOC grows from a mean of 18.86 at Level 1 to 30.32 at Level 4, while Cyclomatic Complexity rises from 3.59 to 6.00 over the same range. The overall means of 24.34 and 4.89 for the two metrics indicate that LOC and Cyclomatic complexity remain quite low across the code snippets.

	LOC					Cyclomatic Complexity				
Level	μ	max	min	M	σ	μ	max	min	M	σ
1	18.86	33	10	18.0	6.04	3.59	7	1	3.0	2.06
2	22.95	46	11	21.0	9.63	4.59	8	1	5.0	1.99
3	25.23	49	11	25.0	10.94	5.36	10	1	5.0	2.38
4	30.32	60	8	28.5	13.36	6.00	14	1	6.0	3.37
Overall	24.34	60	8	22.0	10.98	4.89	14	1	5.0	2.63

Table a.1: Descriptive Statistics of LOC and Cyclomatic Complexity per Level and Overall

The tables: Table a.2, Table a.3, Table a.4, Table a.5, Table a.6, Table a.7, Table a.8, Table a.9, and Table a.10 provide descriptive statistics for each task by prompting type and level. This data is used in Chapter 5 to understand what the results mean.

	Recall					**Precision**					**Jaccard Index**					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.30	1.00	0	0.29	0.32	0.26	1.00	0	0.20	0.29	0.20	1.00	0	0.12	0.25	88
one_shot	0.32	1.00	0	0.29	0.30	0.33	1.00	0	0.27	0.31	0.22	1.00	0	0.17	0.23	88
one_shot_cot	0.33	1.00	0	0.33	0.31	0.33	1.00	0	0.29	0.31	0.23	1.00	0	0.18	0.23	88
zero_shot	0.36	1.00	0	0.30	0.38	0.31	1.00	0	0.21	0.33	0.25	1.00	0	0.14	0.29	88
Level																
1	0.40	1.00	0	0.43	0.36	0.36	1.00	0	0.42	0.31	0.27	1.00	0	0.29	0.25	88
2	0.34	1.00	0	0.33	0.31	0.30	1.00	0	0.25	0.30	0.22	1.00	0	0.17	0.24	88
3	0.28	1.00	0	0.28	0.29	0.28	1.00	0	0.24	0.30	0.19	1.00	0	0.16	0.23	88
4	0.30	1.00	0	0.22	0.33	0.28	1.00	0	0.20	0.32	0.22	1.00	0	0.11	0.28	88

Table a.2: Descriptive Statistics per Prompting Type and per Level for the Task of Defs Identification

	Recall					**Precision**					**Jaccard Index**					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.27	1.00	0	0.21	0.32	0.20	1.00	0	0.11	0.26	0.16	1.00	0	0.09	0.23	88
one_shot	0.28	1.00	0	0.25	0.24	0.21	1.00	0	0.16	0.21	0.15	0.85	0	0.10	0.16	88
one_shot_cot	0.29	1.00	0	0.29	0.26	0.22	1.00	0	0.18	0.22	0.16	1.00	0	0.13	0.18	88
zero_shot	0.25	1.00	0	0.23	0.28	0.21	1.00	0	0.14	0.27	0.16	1.00	0	0.09	0.22	88
Level																
1	0.33	1.00	0	0.29	0.30	0.25	1.00	0	0.19	0.25	0.19	1.00	0	0.13	0.22	88
2	0.31	1.00	0	0.29	0.29	0.23	1.00	0	0.16	0.23	0.17	1.00	0	0.12	0.19	88
3	0.23	1.00	0	0.22	0.24	0.19	1.00	0	0.11	0.26	0.14	1.00	0	0.08	0.20	88
4	0.22	1.00	0	0.16	0.25	0.17	1.00	0	0.11	0.22	0.12	1.00	0	0.06	0.17	88

Table a.3: Descriptive Statistics per Prompting Type and per Level for the Task of Uses Identification

	Recall					**Precision**					**Jaccard Index**					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.12	1.00	0	0.00	0.28	0.10	1.00	0	0.00	0.22	0.07	1.00	0	0.00	0.19	88
one_shot	0.19	1.00	0	0.00	0.32	0.14	1.00	0	0.00	0.24	0.10	1.00	0	0.00	0.19	88
one_shot_cot	0.16	1.00	0	0.00	0.30	0.11	1.00	0	0.00	0.23	0.09	1.00	0	0.00	0.20	88
zero_shot	0.15	1.00	0	0.00	0.30	0.10	1.00	0	0.00	0.23	0.08	1.00	0	0.00	0.18	88
Level																
1	0.22	1.00	0	0.00	0.36	0.13	1.00	0	0.00	0.22	0.10	1.00	0	0.00	0.19	88
2	0.20	1.00	0	0.00	0.33	0.13	1.00	0	0.00	0.23	0.11	1.00	0	0.00	0.21	88
3	0.12	1.00	0	0.00	0.27	0.10	1.00	0	0.00	0.24	0.07	1.00	0	0.00	0.20	88
4	0.09	1.00	0	0.00	0.21	0.09	1.00	0	0.00	0.22	0.06	1.00	0	0.00	0.16	88

Table a.4: Descriptive Statistics per Prompting Type and per Level for the Task of Function Calls Identification

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.16	0.50	0	0.13	0.14	0.35	1.00	0	0.33	0.25	0.12	0.38	0	0.10	0.11	88
one_shot	0.23	1.00	0	0.17	0.21	0.40	1.00	0	0.37	0.26	0.18	1.00	0	0.13	0.18	88
one_shot_cot	0.23	1.00	0	0.17	0.22	0.39	1.00	0	0.33	0.25	0.18	1.00	0	0.13	0.18	88
zero_shot	0.15	1.00	0	0.11	0.16	0.32	1.00	0	0.33	0.24	0.12	1.00	0	0.09	0.14	88
Level																
1	0.24	1.00	0	0.17	0.20	0.41	1.00	0	0.41	0.24	0.19	1.00	0	0.14	0.17	88
2	0.19	1.00	0	0.12	0.22	0.35	1.00	0	0.30	0.28	0.15	1.00	0	0.10	0.20	88
3	0.18	0.50	0	0.11	0.16	0.36	1.00	0	0.33	0.25	0.14	0.43	0	0.10	0.13	88
4	0.16	0.80	0	0.11	0.16	0.33	0.80	0	0.33	0.23	0.13	0.67	0	0.08	0.13	88

Table a.5: Descriptive Statistics per Prompting Type and per Level for the Task of Basic Blocks Identification

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.34	1.00	0	0.31	0.20	0.78	1.00	0	0.96	0.29	0.31	1.00	0	0.26	0.19	88
one_shot	0.44	1.00	0	0.41	0.22	0.86	1.00	0	0.95	0.18	0.42	1.00	0	0.38	0.21	88
one_shot_cot	0.43	1.00	0	0.42	0.23	0.80	1.00	0	0.83	0.23	0.39	1.00	0	0.38	0.22	88
zero_shot	0.32	1.00	0	0.28	0.21	0.76	1.00	0	0.86	0.31	0.30	1.00	0	0.28	0.20	88
Level																
1	0.46	1.00	0	0.42	0.23	0.84	1.00	0	1.00	0.23	0.43	1.00	0	0.39	0.23	88
2	0.36	1.00	0	0.33	0.22	0.77	1.00	0	0.83	0.29	0.33	1.00	0	0.30	0.22	88
3	0.36	0.73	0	0.33	0.18	0.82	1.00	0	0.83	0.20	0.34	0.73	0	0.32	0.18	88
4	0.35	1.00	0	0.31	0.23	0.77	1.00	0	0.86	0.30	0.32	1.00	0	0.30	0.21	88

Table a.6: Descriptive Statistics per Prompting Type and per Level for the Task of Basic Block Entry Identification

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.35	1.00	0	0.33	0.22	0.82	1.00	0	1.00	0.23	0.34	1.00	0	0.30	0.21	88
one_shot	0.43	1.00	0	0.40	0.22	0.81	1.00	0	0.85	0.21	0.40	1.00	0	0.38	0.21	88
one_shot_cot	0.42	1.00	0	0.40	0.25	0.79	1.00	0	0.85	0.25	0.38	1.00	0	0.37	0.23	88
zero_shot	0.36	1.00	0	0.33	0.21	0.81	1.00	0	1.00	0.26	0.34	1.00	0	0.31	0.21	88
Level																
1	0.45	1.00	0	0.41	0.23	0.83	1.00	0	1.00	0.21	0.42	1.00	0	0.39	0.22	88
2	0.38	1.00	0	0.38	0.23	0.78	1.00	0	0.83	0.25	0.36	1.00	0	0.34	0.22	88
3	0.37	0.73	0	0.33	0.21	0.81	1.00	0	0.80	0.20	0.35	0.73	0	0.33	0.21	88
4	0.35	1.00	0	0.33	0.23	0.81	1.00	0	1.00	0.27	0.33	1.00	0	0.30	0.22	88

Table a.7: Descriptive Statistics per Prompting Type and per Level for the Task of Basic Block Exit Identification

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.02	0.17	0	0.00	0.03	0.04	0.33	0	0.00	0.09	0.01	0.13	0	0.00	0.03	88
one_shot	0.03	0.25	0	0.00	0.05	0.05	0.43	0	0.00	0.09	0.02	0.18	0	0.00	0.04	88
one_shot_cot	0.03	0.21	0	0.00	0.06	0.07	1.00	0	0.00	0.14	0.02	0.18	0	0.00	0.04	88
zero_shot	0.01	0.23	0	0.00	0.03	0.03	0.50	0	0.00	0.07	0.01	0.19	0	0.00	0.03	88
Level																
1	0.03	0.21	0	0.00	0.05	0.05	0.38	0	0.00	0.10	0.02	0.15	0	0.00	0.04	88
2	0.03	0.25	0	0.00	0.05	0.07	1.00	0	0.00	0.15	0.02	0.19	0	0.00	0.04	88
3	0.02	0.19	0	0.00	0.04	0.04	0.43	0	0.00	0.08	0.02	0.15	0	0.00	0.03	88
4	0.01	0.21	0	0.00	0.03	0.03	0.46	0	0.00	0.07	0.01	0.17	0	0.00	0.02	88

Table a.8: Descriptive Statistics per Prompting Type and per Level for the Task of Def-Use Chains Creation

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.50	1.00	0	0.54	0.40	0.52	1.00	0	0.61	0.40	0.44	1.00	0	0.38	0.39	88
one_shot	0.48	1.00	0	0.50	0.36	0.50	1.00	0	0.53	0.37	0.39	1.00	0	0.33	0.34	88
one_shot_cot	0.51	1.00	0	0.59	0.39	0.50	1.00	0	0.54	0.38	0.42	1.00	0	0.38	0.37	88
zero_shot	0.50	1.00	0	0.50	0.39	0.51	1.00	0	0.56	0.39	0.43	1.00	0	0.37	0.38	88
Level																
1	0.58	1.00	0	0.75	0.40	0.58	1.00	0	0.75	0.40	0.52	1.00	0	0.60	0.39	88
2	0.50	1.00	0	0.50	0.39	0.48	1.00	0	0.50	0.38	0.40	1.00	0	0.33	0.36	88
3	0.51	1.00	0	0.55	0.38	0.52	1.00	0	0.58	0.38	0.42	1.00	0	0.39	0.35	88
4	0.39	1.00	0	0.32	0.36	0.45	1.00	0	0.48	0.37	0.33	1.00	0	0.23	0.34	88

Table a.9: Descriptive Statistics per Prompting Type and per Level for the Task of Backward Program Output

	Recall					Precision					Jaccard Index					
	μ	max	min	M	σ	μ	max	min	M	σ	μ	max	min	M	σ	n
Prompting Type																
cot	0.40	1.00	0	0.33	0.26	0.64	1.00	0	0.67	0.23	0.31	0.86	0	0.29	0.18	88
one_shot	0.40	1.00	0	0.40	0.22	0.65	1.00	0	0.67	0.24	0.32	0.71	0	0.30	0.16	88
one_shot_cot	0.41	1.00	0	0.38	0.24	0.65	1.00	0	0.67	0.26	0.33	0.79	0	0.33	0.18	88
zero_shot	0.37	1.00	0	0.33	0.24	0.67	1.00	0	0.67	0.26	0.32	1.00	0	0.27	0.22	88
Level																
1	0.42	1.00	0	0.35	0.24	0.67	1.00	0	0.68	0.26	0.33	1.00	0	0.30	0.18	88
2	0.43	1.00	0	0.40	0.22	0.70	1.00	0	0.71	0.22	0.35	0.80	0	0.33	0.16	88
3	0.37	1.00	0	0.33	0.23	0.61	1.00	0	0.63	0.24	0.29	0.86	0	0.25	0.18	88
4	0.37	1.00	0	0.35	0.26	0.64	1.00	0	0.65	0.27	0.31	1.00	0	0.28	0.21	88

Table a.10: Descriptive Statistics per Prompting Type and per Level for the Task of Backward Program Slice.

a.2 Results additional plots - ANOVA

This section contains the additional results obtained from the statistical test of ART ANOVA. The tables: Table a.11, Table a.12, Table a.13, Table a.14 Table a.15, Table a.16, Table a.17, Table a.18, Table a.19 show the results from ART ANOVA for each task. For all the cases the observed significant results are marked in bold. These tests were performed to check the effect of the combination of prompting types and levels of snippets. However, this pair was not found significant for any of the evaluation metrics and tasks.

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	2.0261	1.192e-01	1.380e-01	0.0074
	Level	1.2627	2.948e-01	2.908e-01	0.0095
	Prompting Type * Level	1.1962	2.996e-01	3.184e-01	0.0062
Recall	Prompting Type	1.3925	2.533e-01	2.596e-01	0.0050
	Level	2.2965	8.621e-02	1.137e-01	0.0215
	Prompting Type * Level	0.9628	4.723e-01	4.167e-01	0.0060
Jaccard Index	Prompting Type	1.0658	3.700e-01	3.588e-01	0.0050
	Level	1.3909	2.538e-01	2.602e-01	0.0132
	Prompting Type * Level	1.4361	1.752e-01	2.418e-01	0.0103

Table a.11: Repeated Measures ANOVA Results for Defs Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	0.1175	9.495e-01	8.656e-01	0.0005
	Level	2.2239	9.406e-02	1.111e-01	0.0180
	Prompting Type * Level	1.3674	2.055e-01	2.586e-01	0.0102
Recall	Prompting Type	0.8834	4.546e-01	4.348e-01	0.0035
	Level	3.5008	2.045e-02	2.429e-02	0.0346
	Prompting Type * Level	1.2081	2.921e-01	3.135e-01	0.0093
Jaccard Index	Prompting Type	0.1839	9.070e-01	8.440e-01	0.0008
	Level	1.8811	1.418e-01	1.532e-01	0.0213
	Prompting Type * Level	0.8045	6.126e-01	4.898e-01	0.0058

Table a.12: Repeated Measures ANOVA Results for Uses Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	1.0466	3.783e-01	3.632e-01	0.0044
	Level	0.4791	6.980e-01	6.648e-01	0.0061
	Prompting Type * Level	0.9408	4.910e-01	4.327e-01	0.0104
Recall	Prompting Type	1.7170	1.725e-01	1.879e-01	0.0073
	Level	2.9628	3.879e-02	4.938e-02	0.0337
	Prompting Type * Level	0.5867	8.071e-01	6.612e-01	0.0065
Jaccard Index	Prompting Type	0.7116	5.487e-01	4.904e-01	0.0028
	Level	1.0858	3.617e-01	3.591e-01	0.0121
	Prompting Type * Level	0.8021	6.148e-01	4.966e-01	0.0091

Table a.13: Repeated Measures ANOVA Results for Function Calls Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	η^2_G
Precision	Prompting type	3.8209	**1.401e-02**	**1.914e-02**	0.0194
	Level	2.2964	8.622e-02	1.033e-01	0.0159
	Prompting type × Level	0.6138	7.844e-01	6.734e-01	0.0075
Recall	Prompting type	11.2973	**5.032e-06**	**5.964e-05**	0.0434
	Level	2.7273	5.141e-02	6.857e-02	0.0287
	Prompting type × Level	1.2225	2.833e-01	3.087e-01	0.0108
Jaccard Index	Prompting type	10.0949	**1.607e-05**	**1.035e-04**	0.0380
	Level	2.3220	8.361e-02	1.052e-01	0.0227
	Prompting type × Level	0.9634	4.718e-01	4.114e-01	0.0104

Table a.14: Repeated Measures ANOVA Results for Basic Blocks Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	2.5859	6.091e-02	8.740e-02	0.0233
	Level	1.5742	2.044e-01	2.129e-01	0.0138
	Prompting Type * Level	1.4358	1.753e-01	2.331e-01	0.0259
Recall	Prompting Type	20.4945	**2.205e-09**	**1.705e-07**	0.0610
	Level	2.9191	**4.086e-02**	**4.874e-02**	0.0429
	Prompting Type * Level	1.5783	1.242e-01	1.960e-01	0.0132
Jaccard Index	Prompting Type	20.5902	**2.053e-09**	**4.377e-08**	0.0551
	Level	3.0493	**3.498e-02**	**4.192e-02**	0.0414
	Prompting Type * Level	1.3235	2.270e-01	2.712e-01	0.0116

Table a.15: Repeated Measures ANOVA Results for Basic Blocks Entry Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	0.4549	7.148e-01	6.927e-01	0.0021
	Level	0.5666	6.391e-01	6.135e-01	0.0055
	Prompting Type * Level	0.7966	6.198e-01	5.354e-01	0.0120
Recall	Prompting Type	19.2865	**5.511e-09**	**1.977e-07**	0.0256
	Level	2.1625	1.013e-01	1.090e-01	0.0301
	Prompting Type * Level	1.4885	1.546e-01	2.290e-01	0.0067
Jaccard Index	Prompting Type	13.4196	**7.126e-07**	**4.298e-05**	0.0155
	Level	1.6981	1.765e-01	1.817e-01	0.0230
	Prompting Type * Level	0.7480	6.646e-01	5.203e-01	0.0037

Table a.16: Repeated Measures ANOVA Results for Basic Blocks Exit Identification Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	2.8251	**4.573e-02**	5.778e-02	0.0182
	Level	2.3484	8.100e-02	1.042e-01	0.0188
	Prompting Type * Level	0.6349	7.664e-01	5.717e-01	0.0120
Recall	Prompting Type	3.4312	**2.221e-02**	**4.151e-02**	0.0208
	Level	1.6909	1.780e-01	1.883e-01	0.0173
	Prompting Type * Level	0.3121	9.703e-01	8.263e-01	0.0046
Jaccard Index	Prompting Type	3.0542	**3.477e-02**	5.551e-02	0.0188
	Level	1.8853	1.411e-01	1.549e-01	0.0168
	Prompting Type * Level	0.3122	9.703e-01	8.180e-01	0.0049

Table a.17: Repeated Measures ANOVA Results for Def-Use Chains Creation Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	0.3657	7.780e-01	7.484e-01	0.0008
	Level	1.7802	1.600e-01	1.701e-01	0.0179
	Prompting Type * Level	0.7976	6.189e-01	5.093e-01	0.0047
Recall	Prompting Type	0.3465	7.917e-01	7.816e-01	0.0010
	Level	3.2445	**2.772e-02**	**4.765e-02**	0.0305
	Prompting Type * Level	1.1108	3.569e-01	3.514e-01	0.0060
Jaccard Index	Prompting Type	0.8060	4.952e-01	4.894e-01	0.0022
	Level	3.8346	**1.379e-02**	**2.114e-02**	0.0335
	Prompting Type * Level	0.9984	4.428e-01	4.042e-01	0.0052

Table a.18: Repeated-Measures ANOVA Results for Program Output Dataset

Metric	**Factor**	**F**	$\mathbf{p}_{unc}$	$\mathbf{p}_{GG}$	**Partial** η^2
Precision	Prompting Type	0.6032	6.153e-01	5.723e-01	0.0026
	Level	0.9400	4.268e-01	4.139e-01	0.0183
	Prompting Type * Level	0.9203	5.087e-01	4.426e-01	0.0090
Recall	Prompting Type	1.7567	1.646e-01	1.725e-01	0.0044
	Level	0.9037	4.445e-01	4.096e-01	0.0129
	Prompting Type * Level	1.7203	8.680e-02	1.825e-01	0.0146
Jaccard Index	Prompting Type	0.5053	6.800e-01	5.776e-01	0.0019
	Level	0.9173	4.377e-01	4.045e-01	0.0149
	Prompting Type * Level	0.5570	8.309e-01	6.264e-01	0.0054

Table a.19: Repeated Measures ANOVA Results for Backward Program Slicing Dataset

Statement on the Usage of Generative Digital Assistants

In the preparation of this thesis, the freely available ChatGPT (GPT-4o) and Claude AI (Claude Sonnet 4.6) were selectively employed for text paraphrasing, reformatting and improving LaTeX tables, and identifying transitional gaps between sentences. These interactions were conducted via incognito or temporary chat sessions, and no additional study-specific context was provided to either tool. All suggestions were critically reviewed, and only content that was factually accurate, contextually relevant, and aligned with the research objectives was incorporated.

For scripting tasks, DeepSeek V3.2 and ChatGPT were used to resolve issues in existing scripts and complete missing code segments. In isolated instances, for specific code errors encountered during data processing were examined with ChatGPT, strictly without the surrounding context of the broader script, and again under the temporary chat option. In all cases, the tools were provided only with the minimal content necessary for the task at hand. All generated or corrected code was thoroughly tested and adopted only where it demonstrably satisfied the functional requirements and edge cases.

We are fully aware of the limitations and potential risks of generative AI tools, including hallucinated facts or superficial suggestions. Therefore, the generated content was not used blindly, but rather served as a supporting aid with this thesis. This approach helped ensure the reliability, coherence, and academic integrity of the content.

Bibliography

[1] F. E. Allen and J. Cocke. "A program data flow analysis procedure." In: *Commun. ACM* 19.3 (Mar. 1976), p. 137. ISSN: 0001-0782. DOI: 10.1145/360018.360025. URL: https://doi.org/10.1145/360018.360025.

[2] Frances E. Allen. "Control flow analysis." In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[3] Kamel Alrashedy and Ahmed Binjahlan. *Language Models are Better Bug Detector Through Code-Pair Classification*. 2024. arXiv: 2311.07957 [cs.SE]. URL: https://arxiv.org/abs/2311.07957.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732.

[5] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing." In: *Journal of the Royal Statistical Society: Series B (Methodological)* 57.1 (Dec. 2018), pp. 289–300. ISSN: 0035-9246. DOI: 10.1111/j.2517-6161.1995.tb02031.x. eprint: https://academic.oup.com/jrsssb/article-pdf/57/1/289/49173396/jrsssb_57_1_289.pdf. URL: https://doi.org/10.1111/j.2517-6161.1995.tb02031.x.

[6] A. Cain, T. Y. Chen, D. D. Grant, F.-C. Kuo, and J.-G. Schneider. "An Object Oriented Approach towards Dynamic Data Flow Analysis (Short Paper)." In: *2008 The Eighth International Conference on Quality Software*. 2008, pp. 163–168. DOI: 10.1109/QSIC.2008.18.

[7] Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, Zekun Wang, Boyang Wang, Xianjie Wu, Bing Wang, Tongliang Li, Liqun Yang, Sufeng Duan, and Zhoujun Li. *McEval: Massively Multilingual Code Evaluation*. 2024. arXiv: 2406.07436 [cs.PL]. URL: https://arxiv.org/abs/2406.07436.

[8] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. *A Survey on Evaluation of Large Language Models*. 2023. arXiv: 2307.03109 [cs.CL]. URL: https://arxiv.org/abs/2307.03109.

[9] Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. *A Survey on Evaluating Large Language Models in Code Generation Tasks*. 2025. arXiv: 2408.16498 [cs.SE]. URL: https://arxiv.org/abs/2408.16498.

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

[11] Jon Crall. *The MCC approaches the geometric mean of precision and recall as true negatives approach infinity*. 2023. arXiv: 2305.00594 [cs.CV]. URL: https://arxiv.org/abs/2305.00594.

[12] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. *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.

[13] Milton Friedman. "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance." In: *Journal of the American Statistical Association* 32.200 (1937), pp. 675–701. ISSN: 01621459, 1537274X. URL: http://www.jstor.org/stable/2279372 (visited on 03/10/2026).

[14] Bruno Gois Mateus and Matias Martinez. "An empirical study on quality of Android applications written in Kotlin language." In: *Track "ICSE 2020 Journal First", The 42nd International Conference on Software Engineering*. Vol. 24. Empirical Software Engineering 6. Seoul, South Korea, July 2020, pp. 3356–3393. DOI: 10.1007/s10664-019-09727-4. URL: https://uphf.hal.science/hal-03388917.

[15] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[16] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[18] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[19] Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. "A systematic mapping study of clone visualization." In: *Computer Science Review* 37 (2020), p. 100266. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2020.100266. URL: https://www.sciencedirect.com/science/article/pii/S1574013719302679.

[20] Sabaat Haroon, Ahmad Faraz Khan, Ahmad Humayun, Waris Gill, Abdul Haddi Amjad, Ali R. Butt, Mohammad Taha Khan, and Muhammad Ali Gulzar. *How Accurately Do Large Language Models Understand Code?* 2025. arXiv: 2504.04372 [cs.SE]. URL: https://arxiv.org/abs/2504.04372.

[21] Yingqing He, Zhaoyang Liu, Jingye Chen, Zeyue Tian, Hongyu Liu, Xiaowei Chi, Runtao Liu, Ruibin Yuan, Yazhou Xing, Wenhai Wang, Jifeng Dai, Yong Zhang, Wei Xue, Qifeng Liu, Yike Guo, and Qifeng Chen. *LLMs Meet Multimodal Generation and Editing: A Survey*. 2024. arXiv: 2405.19334 [cs.AI]. URL: https://arxiv.org/abs/2405.19334.

[22] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. *Large Language Models for Software Engineering: A Systematic Literature Review*. 2024. arXiv: 2308.10620 [cs.SE]. URL: https://arxiv.org/abs/2308.10620.

[23] Qing Huang, Zhiwen Luo, Zhenchang Xing, Jinshan Zeng, Jieshan Chen, Xiwei Xu, and Yong Chen. "Revealing the Unseen: AI Chain on LLMs for Predicting Implicit Dataflows to Generate Dataflow Graphs in Dynamically Typed Code." In: 33.7 (Sept. 2024). ISSN: 1049-331X. DOI: 10.1145/3672458. URL: https://doi.org/10.1145/3672458.

[24] Qing Huang, Zhou Zou, Zhenchang Xing, Zhenkang Zuo, Xiwei Xu, and Qinghua Lu. *AI Chain on Large Language Model for Unsupervised Control Flow Graph Generation for Statically-Typed Partial Code*. 2023. arXiv: 2306.00757 [cs.SE]. URL: https://arxiv.org/abs/2306.00757.

[25] Binyuan Hui et al. *Qwen2.5-Coder Technical Report*. 2024. arXiv: 2409.12186 [cs.CL]. URL: https://arxiv.org/abs/2409.12186.

[26] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL]. URL: https://arxiv.org/abs/2406.00515.

[27] Ken Kennedy. "Use-definition chains with applications." In: *Comput. Lang.* 3.3 (Jan. 1978), 163–179. ISSN: 0096-0551. DOI: 10.1016/0096-0551(78)90009-7. URL: https://doi.org/10.1016/0096-0551(78)90009-7.

[28] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. "Static analysis of android apps: A systematic literature review." In: *Information and Software Technology* 88 (2017), pp. 67–95. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0950584917302987.

[29] Xun Liang, Hanyu Wang, Yezhaohui Wang, Shichao Song, Jiawei Yang, Simin Niu, Jie Hu, Dan Liu, Shunyu Yao, Feiyu Xiong, and Zhiyu Li. *Controllable Text Generation for Large Language Models: A Survey*. 2024. arXiv: 2408.12599 [cs.CL]. URL: https://arxiv.org/abs/2408.12599.

[30] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL]. URL: https://arxiv.org/abs/2307.03172.

[31] Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhan Wang, Qiang Hu, Jie Zhang, and Yang Liu. *Unveiling Code Pre-Trained Models: Investigating Syntax and Semantics Capacities*. 2024. arXiv: 2212.10017 [cs.SE]. URL: https://arxiv.org/abs/2212.10017.

[32] T.J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[33] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. *Using an LLM to Help With Code Understanding*. 2024. arXiv: 2307.08177 [cs.SE]. URL: https://arxiv.org/abs/2307.08177.

[34] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[35] Rafael Padilla, Wesley L. Passos, Thadeu L. B. Dias, Sergio L. Netto, and Eduardo A. B. da Silva. "A Comparative Analysis of Object Detection Metrics with a Companion Open-Source Toolkit." In: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030279. URL: https://www.mdpi.com/2079-9292/10/3/279.

[36] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. "Program Comprehension and Code Complexity Metrics: An fMRI Study." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 524–536. DOI: 10.1109/ICSE43902.2021.00056.

[37] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. "Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?" In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2023. ACM, June 2023, 117–123. DOI: 10.1145/3587102.3588792. URL: http://dx.doi.org/10.1145/3587102.3588792.

[38] Sander Schulhoff et al. *The Prompt Report: A Systematic Survey of Prompt Engineering Techniques*. 2025. arXiv: 2406.06608 [cs.CL]. URL: https://arxiv.org/abs/2406.06608.

[39] HaoYang Shang, Xuan Liu, Zi Liang, Jie Zhang, Haibo Hu, and Song Guo. *United Minds or Isolated Agents? Exploring Coordination of LLMs under Cognitive Load Theory*. 2025. arXiv: 2506.06843 [cs.AI]. URL: https://arxiv.org/abs/2506.06843.

[40] C. Spearman. "The Proof and Measurement of Association between Two Things." In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101. ISSN: 00029556. URL: http://www.jstor.org/stable/1412159 (visited on 07/02/2025).

[41] M.-A.D Storey, F.D Fracchia, and H.A Müller. "Cognitive design elements to support the construction of a mental model during software exploration." In: *Journal of Systems and Software* 44.3 (1999), pp. 171–185. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(98)10055-9. URL: https://www.sciencedirect.com/science/article/pii/S0164121298100559.

[42] Wannita Takerngsaksiri, Micheal Fu, Chakkrit Tantithamthavorn, Jirat Pasuksmit, Kun Chen, and Ming Wu. *Code Readability in the Age of Large Language Models: An Industrial Case Study from Atlassian*. 2025. arXiv: 2501.11264 [cs.SE]. URL: https://arxiv.org/abs/2501.11264.

[43] Frank Tip. "A survey of program slicing techniques." In: *Journal of Programming Languages* 3.3 (1995), pp. 121–189. URL: https://onlinelibrary.wiley.com/doi/10.1002/swf.41.

[44] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[45] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. "LLMDFA: Analyzing Dataflow in Code with Large Language Models." In: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang. Vol. 37. Curran Associates, Inc., 2024, pp. 131545–131574. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/ed9dcde1eb9c597f68c1d375bbecf3fc-Paper-Conference.pdf.

[46] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. *Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models*. 2025. arXiv: 2406.08731 [cs.SE]. URL: https://arxiv.org/abs/2406.08731.

[47] 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.

[48] Mark Weiser. "Program slicing." In: *Proceedings of the 5th International Conference on Software Engineering (ICSE)*. IEEE, 1981, pp. 439–449. URL: https://dl.acm.org/doi/pdf/10.5555/800078.802557.

[49] E.J. Weyuker. "Evaluating software complexity measures." In: *IEEE Transactions on Software Engineering* 14.9 (1988), pp. 1357–1365. DOI: 10.1109/32.6178.

[50] Frank Wilcoxon. "Individual Comparisons by Ranking Methods." In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 00994987. URL: http://www.jstor.org/stable/3001968 (visited on 03/10/2026).

[51] Jacob O. Wobbrock, Leah Findlater, Darren Gergle, and James J. Higgins. "The aligned rank transform for nonparametric factorial analyses using only anova procedures." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 143–146. ISBN: 9781450302289. DOI: 10.1145/1978942.1978963. URL: https://doi.org/10.1145/1978942.1978963.

[52] Marvin Wyrich, Justus Bogner, and Stefan Wagner. "40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study." In: *ACM Computing Surveys* 56.4 (Nov. 2023), 1–42. ISSN: 1557-7341. DOI: 10.1145/3626522. URL: http://dx.doi.org/10.1145/3626522.

[53] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. *RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation*. 2023. arXiv: 2303.12570 [cs.CL]. URL: https://arxiv.org/abs/2303.12570.

[54] Dewu Zheng, Yanlin Wang, Ensheng Shi, Xilin Liu, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. *Top General Performance = Top Domain Performance? DomainCodeBench: A Multi-domain Code Generation Benchmark*. 2025. arXiv: 2412.18573 [cs.SE]. URL: https://arxiv.org/abs/2412.18573.