Bachelor's Thesis

Relating Data-Flow Metrics to Human Code Comprehension

Moritz Schaefer

April 30, 2026

Advisors: Sebastian Böhm
Anna-Maria Maurer

Reviewers: Prof. Dr. Sven Apel
Prof. Dr. Andreas Zeller

Chair of Software Engineering ◇ Saarland Informatics Campus ◇ Saarland University

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, 30.04.2025 

(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, 30.04.2025 

(Datum Date) (Unterschrift Signature)

Abstract

Program comprehension is a fundamental activity in software engineering and is influenced by both cognitive processes and structural properties of the source code. Numerous approaches have proposed code metrics to approximate code understandability. However, their relationship to actual human code comprehension performance is not yet fully understood.

This thesis investigates how structural code properties relate to program comprehension by analyzing individual performance and variability across participants. Comprehension is operationalized using task correctness and response time. Based on this operationalization, multiple classes of structural metrics are examined, including size-based, control-flow-based and data-flow-based representations.

The results show that structural metrics have measurable but only limited associations with code comprehension outcomes. Size-related measures demonstrate consistent relationships with performance, while data-flow characteristics provide additional explanatory power by capturing dependencies between variables that contribute to code comprehension difficulty. Furthermore, the analysis of variability shows that structural properties influence not only performance but also the consistency with which code is understood across participants.

Overall, the findings indicate that human code comprehension cannot be fully explained by individual metrics alone, but benefits from considering multiple structural perspectives. In particular, incorporating data-flow-related information contributes to a more comprehensive understanding of how code structure affects human comprehension performance.

Contents

List of Figures

List of Tables

Listings

Acronyms

PCA Principal Component Analysis
CFG Control-Flow Graph
UDG Use-Def Graph
AST Abstract Syntax Tree
LOC Lines of code
REC recursion count
VO variable overwrites
VC variable count
GI gini coefficient
DD dependency degree
MFAS minimum feedback arc set
CC Cyclomatic complexity
LMM Linear mixed-effects model
GLMM Generalized linear-mixed-effects model
R^2 coefficient of determination
RMSE root mean squared error
ROC-AUC area under the curve
LogLoss logarithmic loss

1 Introduction

Understanding source code is a core activity in software development, as developers must repeatedly interpret existing programs in tasks such as debugging, reviewing and maintaining code [40]. Code comprehension difficulty depends on a variety of factors, including familiarity with programming constructs, the quality of identifiers and the overall structure of the code [9, 35]. In particular, the way a program organizes its control-flow and data-flow is crucial, since it governs both the execution order of operations and the dependencies between values [33]. Although these structural metrics have been extensively studied from a program analysis perspective [15, 30], their precise impact on how humans comprehend code remains insufficiently understood [40].

With this thesis, we contribute to closing this gap by investigating how structural properties of code relate to human code comprehension. Rather than conducting a new experiment, we build on two existing controlled user studies [4, 32], in which participants' understanding of multiple code snippets was empirically measured. Using these results, we analyze in this thesis which flow-based properties best explain differences in human code comprehension performance.

1.1 Goal of this Thesis

The primary goal of this thesis is to identify which structural properties of code, particularly those related to data-flow and control-flow, have the strongest influence on human code comprehension. To achieve this, in this thesis, we examine a range of established program-structure metrics such as cyclomatic complexity [28] and dependency degree [5] and evaluate how strongly they relate to comprehension outcomes derived from the user studies.

More specifically, we construct in this thesis control-flow graphs and use-def graphs for each code snippet in the dataset and compute a set of structural metrics that capture flow complexity and data dependencies. We then analyze the relationships among these metrics and evaluate their explanatory power for human code comprehension. For this purpose, we use statistical models, especially mixed-effects models, to investigate how structural metrics relate to the observed comprehension performance, while taking the variability of participants and code snippets into account.

1.2 Overview

We structure this thesis into several chapters. In Chapter 2 we introduce the theoretical foundations, including cognitive perspectives on program comprehension and graph-based representations of source code, such as control-flow and data-flow graphs.

In Chapter 3 we present the empirical approach of this thesis. There we describe the construction of control-flow and data-flow representations, the extraction of structural metrics and the statistical methods to analyze their relationship with comprehension performance.

In Chapter 4 we report and interpret the findings of the empirical analysis, focusing on how different structural metrics relate to human code comprehension.

In Chapter 5 we review existing research on code comprehension and software metrics, with a particular focus on studies that investigate structural properties of code and their relationship to human code comprehension.

Finally, in Chapter 6 we summarize the main contributions of this thesis and outline directions for future work.

2

Background

In this chapter, we present the theoretical foundations of this thesis by connecting cognitive perspectives on code comprehension with formal representations of program structures. We discuss how developers understand source code and introduce control-flow and data-flow as key abstractions for capturing structural properties relevant to program behavior.

Building on these representations, we introduce a set of structural metrics derived from control-flow and use-def graphs. We also discuss Principal Component Analysis (PCA) as a general technique for handling correlated variables. Finally, we introduce linear and generalized linear mixed-effects models as the main statistical framework for analyzing the relationship between structural properties and comprehension performance.

2.1 Code Comprehension and Cognitive Load

Code comprehension refers to the cognitive process by which developers understand the structure, behavior and intent of source code [8]. It is a fundamental activity in software development and plays a crucial role in tasks such as debugging, maintenance and code review [38]. Successful comprehension involves constructing mental representations of how a program executes, including both its control-flow and the dependencies between data elements [33].

A wide range of factors influence code comprehension. These include a developer's programming experience, familiarity with language constructs and readability, such as naming conventions, layout and formatting [37]. Beyond these factors, the structural complexity of a program has a substantial impact on the cognitive effort required to understand it [24].

In particular, complex control-flow structures, such as deep nesting or numerous branching paths and data dependencies, increase cognitive load by forcing developers to simultaneously track multiple execution paths and variable states [33]. As this cognitive load increases, comprehension becomes more difficult and error-prone [36].

In this thesis, we focus on capturing and quantifying such structural aspects of source code using established control-flow and data-flow metrics and on relating these quantified properties to empirically measured code comprehension performance.

2.2 Control-Flow and Control-Flow Graphs

Control-flow representations describe the order in which program statements may execute. They allow us to reason about branching behavior, loops and reachability. In software metrics, control-flow graphs form the basis for classical measures such as cyclomatic complexity [28]

We use control-flow analysis to model the possible execution paths of a program. This technique plays a central role in static program analysis and compiler construction, where it supports tasks such as optimization, test generation and detection of unreachable code [26].

A widely used abstraction for representing control-flow is the Control-Flow Graph (CFG) [1].

Before constructing the CFG, we parse the source code into an Abstract Syntax Tree (AST) [1]. The AST represents the syntactic structure of a program as a tree, where nodes correspond to language constructs such as expressions, statements or declarations. This representation abstracts away formatting details and provides a structured basis for further analysis.

Formally [1], let a program P be a finite sequence of statements

$$P = \langle s_1, s_2, \ldots, s_n \rangle$$

obtained after parsing the source code into an AST.

Nodes. We model each statement s_i as a node in the control-flow graph. Additionally, we introduce distinguished entry and exit nodes. Based on this, we define a graph-based representation that captures how control may flow between them.

Control-Flow Graph. The *Control-Flow Graph* of program P is a directed graph

$$\mathrm{CFG}(P) = (V, E, v_{\text{entry}}, v_{\text{exit}})$$

where:

- $V = \{s_1, \ldots, s_n\} \cup \{v_{\text{entry}}, v_{\text{exit}}\}$ is the set of nodes,
- $E \subseteq V \times V$ is a set of directed edges,
- v_{entry} and v_{exit} are distinguished entry and exit nodes.

An edge $(s_i, s_j) \in E$ exists if and only if control may be transferred from a statement to another statement during execution.

Edge Construction. Edges in the control-flow graph are introduced according to the following rules:

- **Sequential flow:** If statement s_j immediately flows statement s_i, then $(s_i, s_j) \in E$.
- **Conditional branches:** If the statement s_i is a conditional with successor s_t and s_f, then

$$(s_i, s_t) \in E \quad \text{and} \quad (s_i, s_f) \in E.$$

- **Loops:** If execution may return from statement s_j to an earlier statement s_i, then $(s_j, s_i) \in E$.
- **Entry and exit:** An edge $(v_{\text{entry}}, s_1) \in E$ exists for the initial statement s_1 and an edge $(s_k, v_{\text{exit}}) \in E$ exists for each statement s_k that may terminate execution.

Control-flow graphs form the basis for many classical and modern complexity metrics, including cyclomatic complexity [28].

2.3 Data-Flow and Use-Def Graphs

We use data-flow analysis [1] to model how values propagate through a program. Unlike control-flow analysis, which focuses on execution order, data-flow analysis captures dependencies between computations. Understanding these dependencies is essential for code comprehension, since developers must track how values are produced and used across the program.

A key challenge when modeling data-flow lies in choosing an appropriate level of abstraction. Control-flow graphs operate at the level of statements, which is sufficient for reasoning about execution paths. However, this level of abstraction is too coarse for accurately capturing data dependencies. Dependencies often arise within expressions inside a single statement, for example through arithmetic operations or chained assignments. A statement-level representation would collapse these internal relationships and lose important information about how values propagate.

To address this limitation, we represent data dependencies at the level of program elements derived from the abstract syntax tree. This finer granularity allows us to model definitions and uses at the level where values are actually created and consumed, including intermediate computations within expressions. As a result, the representation more accurately reflects both the semantics of value propagation and the way developers reason about data dependencies during comprehension.

Definitions and Uses. Let $\mathcal{V}$ denote the set of variables in program P.

For each variable $x \in \mathcal{V}$:

- $\text{Def}(x)$ is the set of program points where x is defined,
- $\text{Use}(x)$ is the set of program points where x is used.

These notations provide the foundation for identifying how values are introduced and subsequently consumed in the program. They are essential for constructing data dependencies, as they define the potential sources and targets of value propagation.

Reaching Definition. A definition Def reaches a use Use, denoted by

$$d \rightarrow^* u,$$

if there exists a path along which the value assigned at Def can propagate to Use without being overwritten by another definition of the same variable.

This concept formalizes when a definition can influence a particular use and thus determines which dependencies are semantically relevant. It is a key criterion for deciding whether a data-flow edge should be introduced.

Dependency Graph. Based on these notations, we define the use-def graph of program P as a directed graph

$$\mathrm{DG}(P) = (N, D)$$

where:

- N is the set of program elements that correspond to definitions and uses,
- $D \subseteq N \times N$ is the set of directed edges representing data dependencies.

An edge $(d, u) \in D$ exists if the value defined at d may influence the computation at u.

This representation captures data dependencies at a finer granularity than control-flow graphs. While control-flow models describe the order of execution at the level of statements, the data-flow graph explicitly represents how values propagate through individual computations. This distinction is essential for accurately modeling dependency complexity and for reflecting the cognitive effort required to track variable interactions.

2.4 Flow-Based Structural Code Metrics

Based on control-flow and data-flow representations, numerous metrics have been proposed to quantify structural properties of programs [15]. This thesis focuses on metrics that reflect execution complexity, data dependency complexity and graph structural characteristics.

Control-Flow Metrics. Cyclomatic complexity (CC) [28] measures the number of independent paths through a program's CFG. It is defined as:

$$V(G) = E - N + 2P \tag{2.1}$$

where E denotes the number of edges in the control-flow graph, N the number of nodes and P the number of connected components. For a single-entry, single-exit program, $P = 1$.

Higher cyclomatic complexity indicates increased branching and a larger number of possible execution paths, which generally leads to higher cognitive load during code comprehension.

Minimum Feedback Arc Set. The minimum feedback arc set (MFAS) [41] of a directed graph quantifies cyclic structures by identifying the smallest set of edges whose removal makes the graph acyclic. For a control-flow graph $G = (V, E)$, the MFAS is defined as

$$\mathrm{MFAS}(G) = \min_{F \subseteq E} \{|F| \mid (V, E \setminus F) \text{ is acyclic}\}. \tag{2.2}$$

A non-empty feedback arc set indicates the presence of cycles in the control-flow, typically corresponding to loops. Larger MFAS values reflect more complex cyclic structures and increased control-flow complexity.

Data-Flow Metrics. The dependency degree (DD) [5] metric captures the complexity of data dependencies within a program. Derived from the use-def graph, it counts the number of incoming dependency edges for each use, indicating how many variable definitions may influence a given operation.

Additional data-flow related metrics considered in this thesis include the number of variables (variable count (VC)), the number of variable overwrites (VO) and recursion occurrences (recursion count (REC)). All of them contribute to the cognitive effort required to track the program state.

2.5 Graph-Structural Properties

To characterize structural properties of control-flow and data-flow graphs, we require quantitative measures that capture how connections are distributed across nodes. In particular, we are interested in the extent to which dependencies are evenly distributed or concentrated on a small number of nodes, as this can affect program comprehension.

Gini Coefficient. To quantify the structural centralization of control-flow and data-flow graphs, we employ the gini coefficient (GI) for graphs as proposed by Domicolo et al. [12].

The gini coefficient originates from economics, where it measures inequality in income distributions. Domicolo et al. adapt this concept to graph theory by defining a degree-based gini index that captures the inequality of node degree distributions in a graph. The degree of a node is measured by counting in- and outgoing edges of the node. For a graph with node degrees $d1, \ldots, d_n$, the gini coefficient is defined as:

$$G = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |d_i - d_j|}{2n \sum_{i=1}^{n} d_i}$$

The value of G lies in the interval $[0, 1]$. A value of $G = 0$ indicates a perfectly uniform degree distribution, whereas larger values correspond to stronger degree inequality and thus greater structural centralization of the graph.

In the context of control-flow and use-def graphs, a high gini coefficient indicates the presence of structurally dominant nodes (e.g., highly connected hubs), which may increase cognitive load during comprehension due to concentrated dependencies.

Compared to power-law fitting approaches, the gini coefficient provides a robust and well-defined measure of structural inequality even for relatively small graphs.

2.6 Size Metric

Lines of code (LOC) is one of the most widely used software metrics and serves as a basic measure of program size [15]. It is typically defined as the number of non-empty, non-comment lines in source code.

Unlike structural metrics derived from control-flow or data-flow representations, LOC does not capture structural or semantic properties of a program. Instead, it provides a coarse-grained estimate of code size.

LOC is particularly important because many structural metrics scale with program size. Larger programs tend to exhibit higher complexity values simply due to their size [25]. Prior work has shown that size can act as a confounding factor when analyzing relationships between code metrics and external outcomes such as maintainability or comprehension [25].

For this reason, LOC is commonly included as a baseline metric to distinguish whether observed effects are due to structural complexity or merely reflect differences in program size.

2.7 Summary of Metrics

Table 2.1 below summarizes all structural metrics used in this thesis, together with their definitions.

Table 2.1: Overview of structural code metrics

Metric	Representation	Description	Abbreviation
Cyclomatic Complexity	CFG	Number of independent paths	CC
Minimum Feedback Arc Set	CFG	Measure of cyclic structure	MFAS
Dependency Degree	UDG	Incoming data dependencies	DD
Gini coefficient	CFG/UDG	Centralization of flow	GI
Variable Count	UDG	Number of variables	VC
Variable Overwrites	UDG	Reaching variable overwrites	VO
Recursion	AST	Structural self-reference	REC
Lines of Code	Code	Lines within a snippet	LOC

2.8 Principal Component Analysis

Principal Component Analysis reduces dimensionality by transforming correlated variables into orthogonal components [17]. These components are linear combinations of the original variables and are ordered such that the first component captures the largest proportion of variance in the data. This is achieved by computing the eigenvectors of the covariance matrix, which define the directions of maximum variance, while the corresponding eigenvalues indicate the amount of the explained variance.

We include PCA as a potential technique to address multicollinearity and reduce dimensionality. However, we do not rely on PCA in the final modeling approach, as we instead analyze the metrics directly after preprocessing and normalization.

2.9 Linear Mixed-Effects Models

Empirical studies of code comprehension commonly involve hierarchically structured data, where multiple observations are nested within higher-level entities [40]. In the context

of this thesis, comprehension performance is measured across multiple code snippets and participants, leading to repeated observations per participant and per task. Such dependencies violate the independence assumptions of standard linear regression models [3].

Linear mixed-effects models (LMMs) [3] address this issue by combining fixed effects, which represent population-level relationships of interest, with random effects, which capture variability associated with grouping factors such as participants or tasks.

Formally, a LMM can be expressed as:

$$y = X\beta + Z\gamma + \epsilon, \tag{2.3}$$

where y is the response vector (e.g. correctness, time), X is the design matrix for fixed effects, β is the vector of fixed effect coefficients, Z is the design matrix for random effects, γ represents random-effects coefficients and ϵ is the residual error term.

We specify fixed effects using standard formula notation. An additive model is written as

$$y \sim A + B$$

where each predictor contributes independently and linearly to the response. In this formulation, the effect of A does not depend on B and vice versa.

To model dependencies between predictors, we use interaction terms. A pure interaction is written as

$$y \sim A : B,$$

which captures how the effect of A changes as B varies, without including their individual main effects.

In practice, interactions are typically modeled together with their main effects using the shorthand notation

$$y \sim A * B.$$

This expression expands to

$$y \sim A + B + A : B,$$

thereby capturing both the individual contributions of A and B as well as their combined effects.

LMMs offer several advantages for analyzing code comprehension data. They allow the model to account for unobserved individual differences between participants and variability in task difficulty through random effects and they yield more reliable estimates of fixed effects compared to models that ignore the hierarchical structure.

In this thesis, LMMs are applied to assess the relationship between structural code metrics and comprehension performance while accounting for participant and task-level variability.

2.10 Generalized Linear Mixed-Effects Models

Generalized linear-mixed-effects models (GLMMs) [3] extend LMMs to non-normal response variables by introducing a link function and a distribution from the exponential family. For example, logistic GLMMs model binary outcomes such as correctness. GLMMs allow us to

model comprehension performance more appropriately when the response variable is not continuous.

In this thesis, we use GLMMs when the response variable is categorical (e.g., correctness) and LMMs when it is continuous (e.g., response time), allowing us to model comprehension performance appropriately across different measurement types.

2.11 Model Performance Indicators

The R^2 measures the proportion of variance in the dependent variable that is explained by the model. In mixed-effects models, marginal R^2 reflects the contribution of the fixed effects (i.e, the structural metrics), allowing us to assess how much explanatory power these predictors provide [23].

The RMSE quantifies the average magnitude of prediction errors. It indicates how far predicted values deviate from observed values on average and is expressed in the same unit as the dependent variable. Lower RMSE values correspond to better predictive accuracy [20].

The area under the curve (ROC-AUC) measures the model's ability to distinguish between correct and incorrect responses across different classification thresholds. A value of 0.5 corresponds to random guessing, while higher values indicate better performance [20].

We evaluate the quality of probabilistic predictions using logarithmic loss (LogLoss), by measuring how well predicted probabilities align with actual outcomes. It penalizes confident but incorrect predictions more strongly, making it sensitive to prediction uncertainty. Lower values indicate better performance [6].

Accuracy measures the proportion of correctly classified observations. While it provides an intuitive measure of performance, it can be misleading in imbalanced datasets and is therefore interpreted alongside other metrics such as ROC-AUC and LogLoss [20].

3

Methodology

In this chapter, we describe the methodological approach used to analyze the relationship between structural properties of source code and human comprehension performance.

The analysis builds on an existing dataset, obtained from a controlled user study [32], in which participants performed comprehension tasks on a set of code snippets. Rather than conducting a new empirical experiment, we focus on extracting structural code metrics from the given snippets and analyze their relationship with empirically observed comprehension outcomes.

To this end, program representations capturing both control-flow and data-flow are constructed for each snippet. Based on these representations, a set of structural metrics is extracted and subsequently related to measures of comprehension performance, including task correctness and response time.

The methodology combines correlation analysis and mixed-effects modeling to provide both an initial exploratory assessment and a more robust statistical analysis that accounts for variability across participants and tasks.

3.1 Research Questions

We investigate in this thesis how structural properties of source code relate to human comprehension performance. In particular, we focus on the role of data-flow information in comparison to traditional control-flow based complexity measures.

> ***RQ₁:*** *How do data-flow metrics correlate with empirically measured code comprehension, relative to established code complexity metrics?*

To address this question, we extract structural metrics from Control-Flow Graphs (CFGs) and Use-Def Graphs (UDGs) for each snippet. These metrics include classical control-flow measures such as cyclomatic complexity and minimum feedback arc set, as well as data-flow-related measures such as dependency degree, variable usage and recursion.

We then perform a correlation analysis between each structural metric and comprehension outcomes, in particular task correctness and response time. This analysis provides an initial assessment of which structural properties are most strongly associated with comprehension and whether data-flow metrics offer additional explanatory power beyond traditional control-flow measures.

While the first research question focuses on individual relationships, we investigate with the second question how multiple metrics jointly explain comprehension performance:

> **RQ_2:** *How can combinations of data-flow and traditional code metrics be used to identify the most important predictors of code complexity?*

To answer this question, we use mixed-effects models, allowing us to simultaneous analyze multiple structural metrics while accounting for variability across participants and code snippets. We include structural metrics as fixed effects, while we model participant and task-specific variability as random effects.

We explore different modeling strategies, including dimensionality reduction using Principal Component Analysis and direct use of normalized metrics.

While the second research question accounts for variability across participants within the model framework, we shift the focus of the third research question on variability in comprehension performance as the primary outcome:

> **RQ_3:** *How do structural code metrics relate to the variability of comprehension performance across participants?*

In contrast to RQ_2, where response time and task correctness are modeled directly, this question considers how consistently different participants understand the same code snippet. To capture this, variability is operationalized based on the dispersion of participant responses per snippet.

Rather than modeling raw performance, the dependent variables are defined as deviations from typical (snippet-level) performance. This allows the analysis to focus on whether certain structural properties are associated with more consistent or more heterogeneous comprehension behavior.

As in RQ2, we employ linear mixed-effects models with structural metrics as fixed effects and participants and snippets as random effects. This ensures that the hierarchical structure of the data is preserved while enabling the analysis of factors influencing variability across participants.

3.2 Operationalization

In this section, we describe how the theoretical concepts and metrics introduced in the background are operationalized for empirical analysis. It details how we compute structural code metrics from program representations, how these metrics are prepared for statistical analysis and how they are integrated with comprehension data to evaluate the research questions.

3.2.1 Graph Construction

To compute structural metrics related to control-flow and data dependencies, two types of program graphs are constructed for each code snippet: a Control-Flow Graph (CFG) and a Use-Def Graph (UDG).

Control-Flow Graph Construction We generate CFGs using a custom static analysis tool implemented in Python using the JAVALANG parser[1]. We construct the CFG at the *statement level*, which is a common abstraction choice in static analysis research [28]. In the context of program comprehension, such representations align with cognitive mental models primarily around executable statements and control structures rather than low-level syntactic tokens [33] Examples include variable declarations, assignments, conditional predicates, loop conditions and return statements.

Edges in the CFG represent one possible execution order between statements. For instance, conditional branches introduce multiple outgoing edges, while loop constructs create back-edges representing repeated execution. Method calls, as well as multiple calls of the same function are represented as dedicated call nodes and if the called method is available in the analyzed snippet, its CFG is inlined into the graph (see 3.1). After graph construction, we remove unreachable nodes to simplify the final structure. We do not inline recursive calls, instead we add a single call and return node (see figure 3.1) as this would lead to infinite expansion of the graph. Therefore, recursive calls are not inlined, instead we add a single call and return node to represent this call. To account for this limitation, we include a separate metric that counts recursive calls.

Listing 3.1: Java code snippet which we use for example 3.1.

```
1   public int update(int n){
2       if(n < 0){
3           n = n-1;
4           return update(n-1);
5       }
6       return add(n);
7   }
8   private int add(int n){
9       int result = n+1;
10      return result;
11  }
```

The resulting CFG therefore models the execution paths through the method at the granularity of individual program statements.

Data-Flow-Graph Construction In contrast to the CFG, we construct the UDG to represent data-flow relationships between program elements. We generate the graph using the CODE PROPERTY GRAPH library[2], which we use as an analysis backend to extract these relationships.

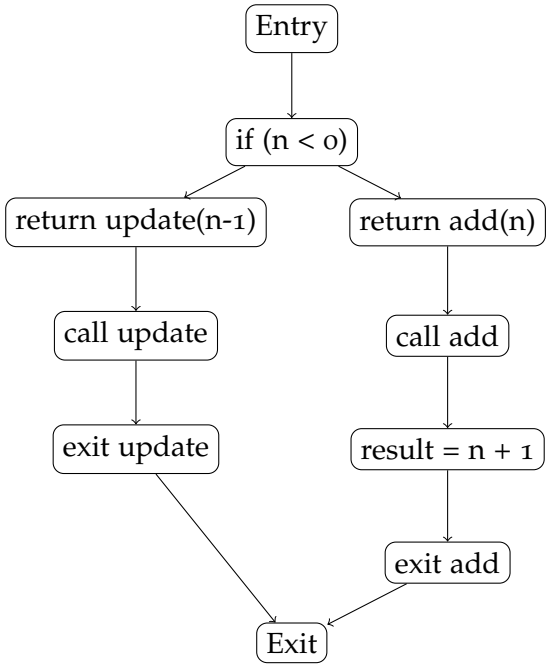
In the resulting graph, nodes correspond to AST-level program elements, such as expressions, operators, variable references and intermediate computation nodes. Edges represent data dependencies, i.e., an edge is added whenever data flows from one node to another.

Because the graph is derived from the abstract syntax tree, it is significantly more fine-grained than the CFG. A single source-code statement may correspond to multiple nodes in the UDG.

This difference in granularity has important consequences for the subsequent analysis. First, UDGs typically contain substantially more nodes and edges than their corresponding

1 https://github.com/c2nes/javalang
2 https://github.com/Fraunhofer-AISEC/cpg

Figure 3.1: CFG construction example, illustrating function calls.



CFGs, due to their finer-grained representation of data dependencies [5] Second, structural metrics derived from these graphs are therefore not directly comparable, as their magnitude is influenced by the underlying representations. In particular, metrics computed on UDGs may appear larger simply due to the higher number of nodes rather than reflecting increased structural complexity.

To address this issue, we apply a dedicated extraction and preprocessing pipeline, which ensures that metrics derived from different representations can be compared meaningfully.

3.2.2 Metric Extraction

We extract structural metrics from the constructed program representations using a combination of Python-based analysis and a Kotlin-based data-flow analysis pipeline.

Overview of the Extraction Pipeline The extraction process follows a two-stage approach. First, we compute the control-flow based metrics directly in Python using the constructed CFGs. After that we compute the data-flow based metrics using an external analysis implemented in Kotlin, based on the Code Property Graph framework, which we extend to include custom modifications such as computation of the gini coefficient as well as the dependency degree size normalization.

Control-Flow Metrics We compute control-flow related metrics using custom Python scripts operating on CFGs represented directed graphs. The graphs are constructed as described in the previous section and processed using the networkx library.

We compute cyclomatic complexity, minimum feedback arc set and gini coefficient using the CFG. We compute cyclomatic complexity based on the number of nodes and edges in the graph, capturing the number of independent execution paths. We approximate Minimum Feedback Arc Set by iteratively removing edges participating in cycles until the graph becomes acyclic. The number of removed edges reflects the degree of cyclic control-flow complexity. And lastly we compute the gini coefficient based on the degree distribution of nodes in the control-flow graph, capturing structural centralization.

The individual metrics and their computation are described in more detail in the following sections.

Data-Flow Metrics We compute data-flow related metrics using a Kotlin-based analysis pipeline built on the Code Property Graph framework. This analysis extracts fine grained data dependencies between program elements. The Python pipeline invokes the Kotlin analysis via a subprocess call, which processes each code snippet and outputs the computed metrics in JSON format.

Dependency degree and the gini coefficient for the same graph used in the dependency degree are computed.

AST-Based Structural Metrics In addition to graph-based metrics, several structural data-flow properties are extracted directly from the abstract syntax tree (AST) using the JAVALANG parser. These metrics include recursion count, variable count and variable overwrites.

Moreover, the size of each code snippet is captured using Lines of Code (LOC). LOC is one of the most widely used software metrics and serves as a baseline measure of program size [15]. Prior work has shown that many structural metrics correlate with program size and that size itself can influence comprehension performance [25]. Including LOC in the analysis therefore allows distinguishing whether observed effects are due to structural complexity or simply reflect differences in code size.

3.2.3 Data Aggregation

We combine all extracted metrics into a unified dataset, where each row corresponds to a single code snippet and each column represents a specific metric.

The resulting dataset is stored in a CSV file.

3.2.4 Data Preprocessing and Normalization

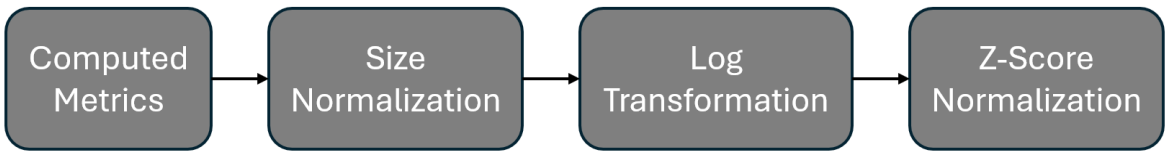


Figure 3.2: In this figure we show the preprocessing pipeline for the metric values.

We illustrate the preprocessing workflow applied in this study in figure 3.2. The pipeline consists of three main steps: size normalization, distribution adjustment and Standardization. Each step is described in detail below.

A key challenge for comparability across metrics arises from the fact that the CFG and UDG are constructed at different levels of abstraction. While CFG nodes represent entire statements, UDG nodes represent individual AST elements within those statements. As a result, the UDG typically contains substantially more nodes than the CFG for the same code snippet.

The use of different abstraction levels is intentional and reflects the different characteristics of control-flow and data-flow reasoning. Control-flow analysis traditionally operates on basic blocks or statements, as its primary goal is to model the order of execution paths through a program [2]. Such representations are sufficient for reasoning about branching structure and reachability.

In contrast, data-flow analysis requires a more fine-grained representation of program behavior. Data dependencies often occur inside expressions and between intermediate computations within a single statement. Use-def analyses therefore operate on program points corresponding to definitions and uses, often derived from AST-level representations [5]. This fine granularity is essential for accurately tracking how values propagate through computations.

From a cognitive perspective, program comprehension involves constructing both control-flow and data-dependency mental models [33]. These models differ in granularity, further motivating the use of distinct abstraction levels.

Because of this difference in granularity, raw metric values derived from the UDG would be biased by the larger number of nodes compared to the CFG. To ensure comparability across metrics, we normalize and preprocess the metrics before performing statistical analysis.

Size normalization. Many structural metrics scale with program size. Prior work has shown that the size effects can dominate metric behavior and distort comparison across programs [25]. More generally, software metrics are known to be sensitive to size unless explicitly normalized [15].

Therefore, for graph-based metrics derived from CFGs and UDGs are normalized with respect to the number of nodes in the corresponding graph. This ensures that differences in metric values reflect structural characteristics rather than trivial differences in graph size.

For a metric value M computed on a graph with $|V|$ nodes, the normalized value is defined as

$$M_{norm} = \frac{M}{|V|}$$

where $|V|$ denotes the number of nodes in the graph from which the metric is derived. This step reduces the influence of graph size and allows metrics extracted from CFG and UDG representations to be compared more meaningfully.

Distribution adjustment. Software metrics frequently exhibit skewed distributions, often caused by a property to be not present within multiple snippets. To dectect such effects, the distribution of each normalized metric is inspected. To account for skewed distributions, we compute the skewness of each metric using the sample skewness coefficient:

$$f = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2\right)^{3/2}}$$

where x_i denotes the observed values, μ the sample mean and n the number of observations. This measure quantifies the asymmetry of the distribution.

We use a threshold of $|f| > 0.5$ to identify moderate to strong skewness. For postitively skewed distributions ($|f| > 0.5$), we apply a logarithmic transformation.

For negatively skewed distributions ($f < -0.5$), we apply a reflection prior to transformation, defined as $x' = max(x) - x$, followed by a logarithmic transformation $log(1 + x')$.

The transformations reduce skewness and stabilize variance, which improves the robustness of subsequent statistical analyses. Log-transformations are commonly used for handling skewed data in empirical studies [18].

Standardization. Finally we standardize all metrics using z-score normalization [21]:

$$z = \frac{x - \mu}{\sigma}$$

where x is the metric value, μ is the mean and σ is the standard deviation of the metric across all code snippets. Standardization ensures that all variables have zero mean and unit variance, preventing metrics with larger numerical ranges from dominating the results of subsequent analyses such as principal component analysis.

This preprocessing pipeline follows standard practices in empirical software engineering and statistical modeling to ensure comparability, robustness and interpretability of results [15, 18, 21, 25].

3.2.5 Correlation Analysis for RQ1

To investigate the relationship between structural code metrics and human comprehension performance (RQ_1), we conduct a correlation analysis between the extracted metrics and the observed comprehension outcomes.

Choice of Correlation Measure We use Spearman's rank correlation coefficient, as it is well-suited for analyzing monotonic relationships without requiring linearity or normally distributed data. This makes it particularly appropriate for empirical software engineering datasets, which often exhibit skewed distributions and outliers [16].

Spearman correlation operates on ranked values rather than raw data, making it more robust to extreme observations and distributional irregularities. In addition, it captures monotonic relationships, allowing the identification of consistent increasing or decreasing trends even when these are not strictly linear [21].

This choice is supported by the characteristics of the data used in this study. Several structural metrics (e.g recursion count and variable overwrites) exhibit skewed distributions and contain lots of zero values, since these properties are often not present. Furthermore, the relationship between structural complexity and comprehension performance is not necessarily linear, but is expected to follow a monotonic trend (e.g. increasing complexity tends to reduce correctness without following a strictly linear pattern).

Overall Spearman correlation provides a robust and flexible measure that aligns well with the distributional properties of the data and the expected form of relationships.

Data Preparation We conduct the correlation analysis on the extracted metric values normalized (only) for graph size as described in section 3.2.4.

Comprehension performance is operationalized as *task correctness* and *response time*

Computation of Correlations We compute pairwise Spearman correlations between each structural metric and comprehension outcomes (correctness and response time) as well as between the structural metrics among themselves to assess interdependencies.

In addition to the correlation coefficients, we compute a corresponding significance matrix (p-values). This matrix indicates whether the observed correlations are statistically significant under the null hypothesis of no associations.

Evaluation of the Correlation Matrix The evaluation of the correlation matrix follows standard practices in empirical research, where both effect size and statistical significance are considered jointly. Effect sizes are interpreted using established conventions for correlation strength, as proposed by Cohen [11], which provide a guideline for distinguishing negligible, weak, moderate and strong relationships as shown in tables 3.1.

We use statistical significance (p-values) to assess whether observed relationships are unlikely to have occurred by chance. However, significance alone is not sufficient, as even small effects can become statistically significant. Therefore, only correlations that are both *statistically significant and practically meaningful* are considered robust.

This combined evaluation approach is widely used in empirical software engineering [22, 27] to ensure that reported relationships are both reliable and substantively relevant.

Table 3.1: In this table we show the correlation magnitudes.

range	relationship
$\|\rho\| < 0.1$	negligible
$0.1 <= \|\rho\| < 0.3$	weak
$0.3 <= \|\rho\| < 0.5$	moderate
$\|\rho\| >= 0.5$	strong

Particular attention is given to metrics that show moderate to strong correlations with task correctness, as these indicate meaningful associations with comprehension performance.

Moreover, the direction of the effects is taken into account, the sign of the correlation coefficient indicates the direction of the relationship. If the sign is negative, then higher metric values are associated with lower task correctness, indicating increased comprehension difficulty. If the sign is positive, higher values of structural metrics are associated with higher task correctness, indicating that the corresponding structural property may facilitate or at least not hinder comprehension.

Third, we use the statistical significance to assess the reliability of observed correlations 3.2

Table 3.2: In this table we show the level of significance we consider as significant.

range	significant
$p < 0.05$	statistically
$p < 0.01$	highly

Only correlations that are both, substantively meaningful (effect size) and statistically significant are considered robust findings.

Interpretation for RQ1 The correlation analysis provides initial, descriptive assessment of how individual structural metrics relate to human comprehension performance. Metrics that show consistent and significant associations with correctness or response time are considered potential indicators of cognitive difficulty and are further examined in subsequent analyses (RQ_2 and RQ_3)

We conduct the analysis on the metric values described in the previous section. Spearman's correlation coefficient is then computed between each metric and the correctness scores of the participants.

3.2.6 Modeling Approaches for RQ2

To address RQ_2, we analyze the relationship between combinations of structural code metrics and comprehension performance using (generalized) linear mixed-effects modeling. The goal is to identify the most relevant predictors of code complexity while accounting for variability across participants and code snippets.

Mixed-effects models are widely used for analyzing hierarchical data structures, particularly in empirical software engineering and behavioral studies [3]. They allow separating population-level effects from variability associated with participants and tasks, leading to more reliable and generalizable estimates. Model selection using information criteria such as AIC is a standard approach for balancing model fit and complexity [16]

Modeling Framework Two types of models are employed depending on the response variable. We model *response time* using Linear mixed-effects models (LMMs) and *task correctness* using Generalized linear-mixed-effects models (GLMMs) with a binomial link function.

In both cases, the models include *fixed effects* to model structural code metrics and *random effects* for participants to account for individual ability differences and code snippets to account for task specific difficulty.

This structure accounts for the hierarchical nature of the data and enables generalizable conclusions about the relationship between code structure and comprehension performance.

Predictor Representation We explored several strategies for representing structural metrics as predictors.

Initially, we apply PCA to the normalized metrics to reduce dimensionality and address multicollinearity. We use the resulting principal components as predictors in mixed-effects models.

However, we do not retain this approach for the final analysis, as it does not lead to improved model performance and significantly reduces interpretability. In particular, principal components obscure the contribution of individual structural metrics, making it difficult to identify concrete predictors of code complexity.

In the final modeling approach we use the original normalized structural metrics directly as predictors. This preserves interpretability and allows identifying specific structural properties that influence comprehension performance.

Handling of Recursion Exploratory analysis reveals that recursion-related metrics exhibit a disproportionately strong effect on comprehension performance. At the same time, only a small subset of code snippets contains recursion.

To assess the robustness of the results and mitigate potential bias introduced by recursive constructs, we analyze two dataset variants. First the *full dataset*, this includes all code snippets and second *filtered data* this excludes snippets containing recursion.

This separation allows us to distinguish general structural effects from those driven primarily by recursion.

Model Construction Strategies To identify relevant predictors, two complementary model construction strategies are applied.

Stepwise Model Construction

We use an iterative forward-selection procedure to construct models incrementally. Starting from an empty model, predictors are added one at a time. At each step, candidate models are evaluated based on the Akaike Information Criterion (AIC) and the statistical significance of the added predictor. A predictor is retained only if it improves model fit (lower AIC) and is statistically significant.

This approach yields parsimonious models that balance explanatory power and complexity.

Full Additive Models

In addition to the stepwise model selection, we estimate models including all predictors in an additive form:

$$Reponse \sim X_1 + X_2 + ... + X_n$$

This approach provides a baseline for comparison and allows assessing the overall explanatory power of the full metric set without feature selection.

As an initial approach, we apply PCA to the normalized metrics to reduce dimensionality and mitigate multicollinearity.

We use the resulting principal component scores as fixed-effect predictors. This approach provides a compact representation of structural properties while ensuring that predictors are uncorrelated.

Handling Sparse and Zero-Inflated Metrics Some structural metrics exhibit highly skewed distributions with a large number of zero values (e.g recursion count). To address this, we explore alternative representations. First we excluded sparse metrics from the predictor set and second we made a binary transformation. If the property is absent, the value is 0 and if the property is there, the value is 1. The transformations allow evaluating whether the presence of a structural property is more informative than its magnitude.

Model Evaluation To evaluate the predictive performance of the mixed-effects models, we employ a set of complementary metrics tailored to the type of response variable. For continuous outcomes (response time), we use R^2-values and RMSE, while for binary outcomes (task correctness), we rely on ROC-AUC, LogLoss and classification accuracy. Using multiple evaluation criteria provides a more comprehensive assessment of model quality, as each metric captures different aspects of predictive performance [16].

Summary of Final Approach Figure 3.3 shows the final analysis modeling approach. The analysis proceeds from preprocessed structural metrics to mixed-effects modeling and comparative model evaluation.

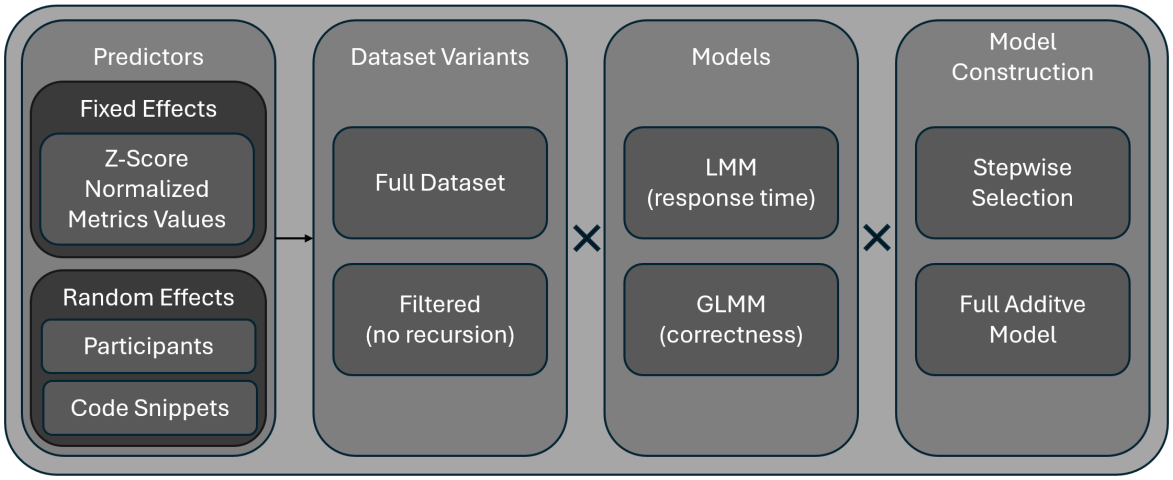


Figure 3.3: In this figure we illustrate the final analysis pipeline for RQ_2.

The analysis is based on normalized structrual metrics to preserve interpretability and identify concrete predictors of comprehension performance, rather than using dimensionality reduction techniques such as PCA.

To ensure robustness, all analyses are conducted on both the full dataset and a filtered dataset excluding recursive snippets.

Model construction is performed using both stepwise selection and full additive models for comparison.

3.2.7 Operationalization for RQ3

To address RQ_3, variability in comprehension performance is operationalized at the level of individual participant responses relative to the typical behavior observed for each code snippet.

For each snippet, the mean correctness and mean response time across all participants are computed. Based on these reference values, deviation measures are defined for each participant. Specifically, we compute the absolute deviation of each observation from the corresponding snippet mean.

Formally, for a participant i and snippet s, the deviation is defined as:

$$D_{i,s} = |x_{i,s} - \bar{x}_s|$$

where $x_{i,s}$ denotes the observed value (task correctness or response time) and $\bar{x}_s$ represents the mean value across all participants for snippet s.

This results in two deviation-based outcome variables. *behavioral deviation* and *duration deviation* capturing the deviation of task correctness and response time respectively from the snippet mean.

These deviation measures quantify how strongly individual responses differ from the typical performance on a given snippet. In contrast to aggregate variability measures (e.g., standard deviation per snippet), this formulation preserves the participant-level structure of the data and allows the use of mixed-effects modeling.

Modeling Approach The deviation measures replace the original dependent variables used in RQ_2. Since both deviation variables are continuous, we use linear mixed-effects models (LMMs) for both analyses.

Model Construction In contrast to RQ_2, the analysis for RQ_3 focuses exclusively on additive models. Preliminary experiments including interaction terms and stepwise model construction frequently resulted in convergence issues and unstable parameter estimates.

This behavior is likely due to the reduced variance and increased noise in the deviation-based outcome variables, which limits the reliable estimation of more complex model structures. Additive models therefore provide a more stable and interpretable basis for analysis.

Limitations of Behavioral Variability A methodological limitation arises from the binary nature of task correctness. Since correctness is encoded as a binary variable, the corresponding deviation measure can take only a limited number of distinct values per snippet.

As a result, behavioral deviation exhibits restricted variability, which limits its explanatory power in the modeling framework. Consequently, findings related to correctness variability should be interpreted with caution.

Overall, this operationalization enables the analysis of variability in comprehension performance while maintaining consistency with the modeling framework used for RQ_2.

4

Evaluation

We present in this chapter the empirical evaluation and address the research questions by analyzing the relationship between structural code metrics and comprehension performance using correlation analysis and mixed-effects modeling. Before presenting the results, we first provide an overview of the dataset and the metric representations used in the analysis.

Data Overview. The evaluation is based on behavioral and structural data collected from two empirical studies [4, 32]. Across both studies, the dataset comprises 56 participants and 55 code snippets, of which 12 contain recursion. For each snippet, structural metrics are extracted as described in Section 3.2 and combined with comprehension outcomes in terms of response time and task correctness.

Metric Representations Used in the Analysis. Different stages of the analysis use different representations of the structural metrics. For the correlation analysis (RQ1), we use metrics that are normalized only with respect to graph size (see Section 3.2.4). This normalization is applied exclusively to graph-based metrics, as these are directly influenced by the number of nodes in the underlying CFG or UDG representation. Non-graph-based metrics (e.g. variable overwrites or recursion count) do not scale proportionally with graph size and are therefore retained in their original form. This approach ensures comparability between control-flow and data-flow metrics while preserving the original distribution and relative differences between snippets, which is important for interpreting monotonic relationships. Figure 4.1 shows the distribution of these graph-size normalized metrics, which form the basis for the correlation analysis. The distributions reveal that several metrics are highly skewed and contain many zero values (e.g. recursion-related metrics).

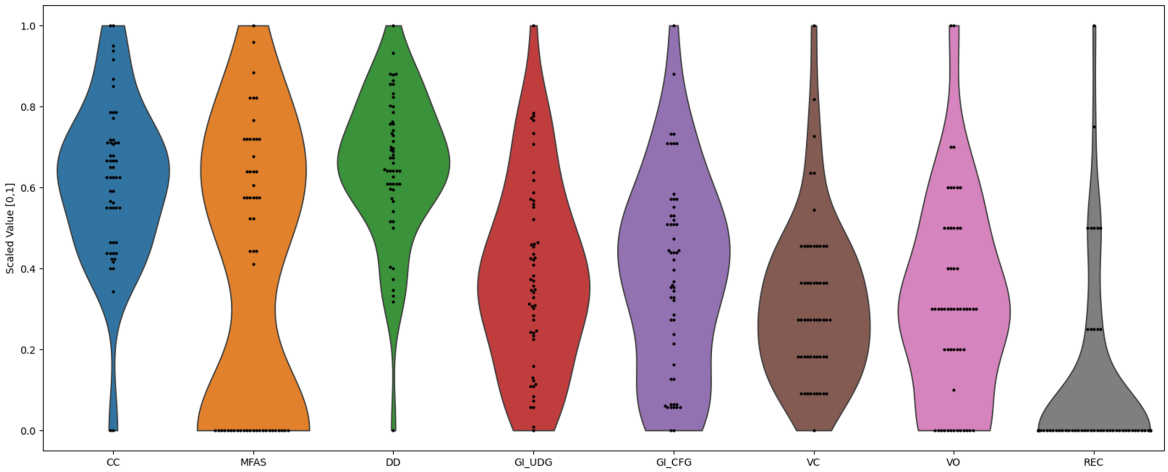


Figure 4.1: Distribution of Structural Metrics (scaled for visualization).

In contrast, the multivariate modeling (RQ2) uses fully normalized and standardized metrics, including distribution adjustment and z-score normalization, to ensure comparability across predictors. Figure 4.2 illustrates the distribution of these fully normalized metrics using violin plots, showing more balanced distributions with reduced skewness. This additional preprocessing step improves model stability and prevents individual metrics from dominating the analysis due to differences in scale.

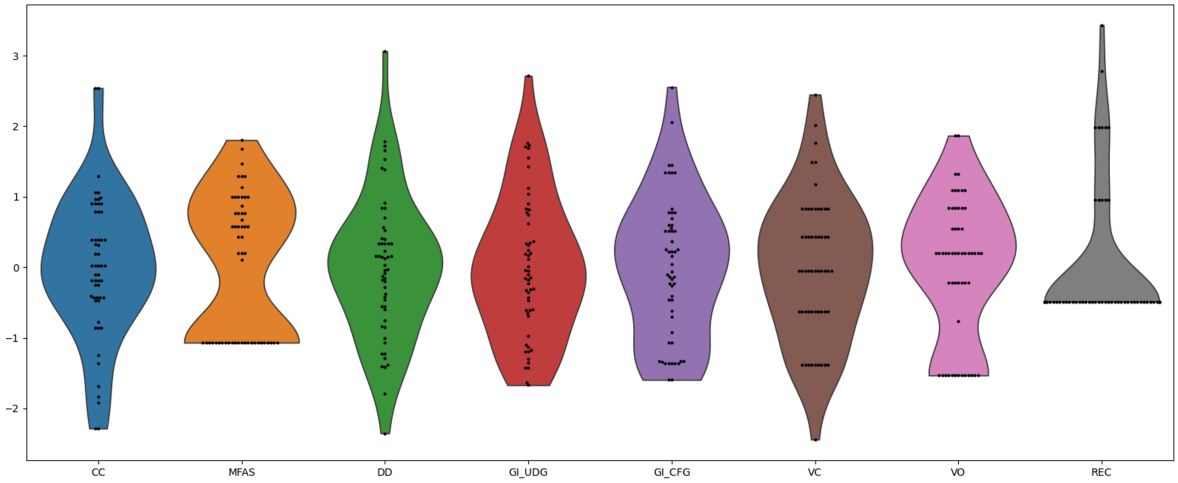


Figure 4.2: Distribution of Normalized and Standardized Structural Metrics.

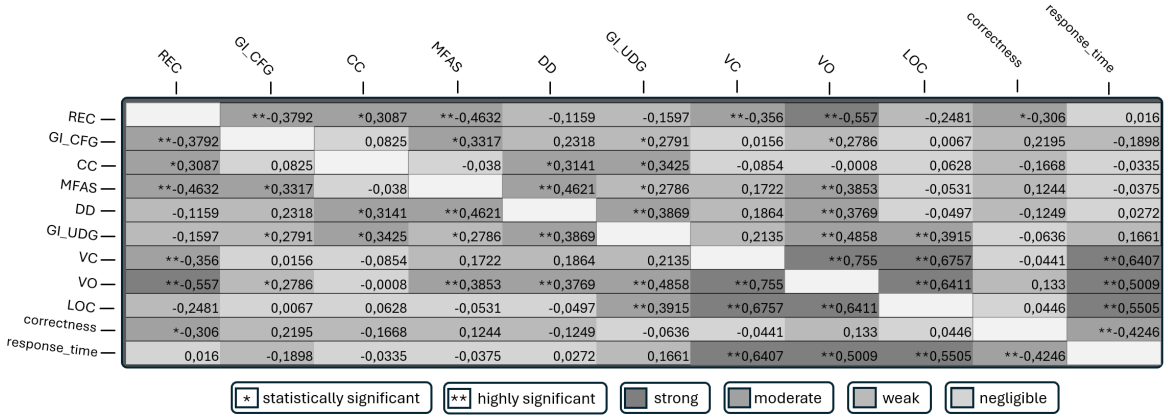
4.1 Results

4.1.1 Results for RQ1

To investigate the relationship between structural metrics and comprehension performance, we compute Spearman's rank correlation coefficient (ρ) between all metrics, task correctness

and response time. In addition, corresponding p-values were calculated to assess statistical significance.

Figure 4.3 shows the resulting correlation matrix. Correlation coefficients are displayed together with significance levels, allowing the identification of statistically robust relationships.



	REC	GI_CFG	CC	MFAS	DD	GI_UDG	VC	VO	LOC	correctness	response_time
REC		**-0,3792	*0,3087	**-0,4632	-0,1159	-0,1597	**-0,356	**-0,557	-0,2481	*-0,306	0,016
GI_CFG	**-0,3792		0,0825	*0,3317	0,2318	*0,2791	0,0156	*0,2786	0,0067	0,2195	-0,1898
CC	*0,3087	0,0825		-0,038	*0,3141	*0,3425	-0,0854	-0,0008	0,0628	-0,1668	-0,0335
MFAS	**-0,4632	*0,3317	-0,038		**0,4621	*0,2786	0,1722	**0,3853	-0,0531	0,1244	-0,0375
DD	-0,1159	0,2318	*0,3141	**0,4621		**0,3869	0,1864	**0,3769	-0,0497	-0,1249	0,0272
GI_UDG	-0,1597	*0,2791	*0,3425	*0,2786	**0,3869		0,2135	**0,4858	**0,3915	-0,0636	0,1661
VC	**-0,356	0,0156	-0,0854	0,1722	0,1864	0,2135		**0,755	**0,6757	-0,0441	**0,6407
VO	**-0,557	*0,2786	-0,0008	**0,3853	**0,3769	**0,4858	**0,755		**0,6411	0,133	**0,5009
LOC	-0,2481	0,0067	0,0628	-0,0531	-0,0497	**0,3915	**0,6757	**0,6411		0,0446	**0,5505
correctness	*-0,306	0,2195	-0,1668	0,1244	-0,1249	-0,0636	-0,0441	0,133	0,0446		**-0,4246
response_time	0,016	-0,1898	-0,0335	-0,0375	0,0272	0,1661	**0,6407	**0,5009	**0,5505	**-0,4246	

Figure 4.3: In this figure we show the results of the correlation analysis, using a correlation matrix. We highlight significant results with stars.

Overall Correlation Structure. The correlation matrix reveals several strong and statistically significant relationships among structural metrics.

In particular, *variable-related and size-related metrics* (variable count, variable overwrites and LOC) form a cluster of strongly positively correlated measures. These relationships are consistently significant and this indicates that these metrics capture closely related aspects of program structure.

Additionally, *dependency degree* shows moderate positive correlations with both control-flow and data-flow metrics, suggesting that it reflects a structural property that spans both domains.

Structural Metrics and Response Time. Several structural metrics exhibit *strong and statistically significant positive correlations with response time.*

The strongest relationships are observed for variable count, variable overwrites and LOC (see Figure 4.3). For example, higher variable counts and larger code size are consistently associated with increased response time. This indicates that snippets with more variables and more complex variable interactions require substantially more time to comprehend.

In contrast, traditional control-flow metrics such as cyclomatic complexity show only weak or non-significant correlations with response time, suggesting that they capture less relevant aspects of cognitive effort in this dataset.

Structural Metrics and Task Correctness. In contrast to response time, task correctness shows almost no significant relationships with structural metrics.

The only exception is *recursion count*, which exhibits a moderate negative and statistically significant correlation with correctness. This indicates that snippets involving recursion are more likely to be answered incorrectly.

However, this interpretation requires careful consideration. Recursive snippets differ fundamentally from non-recursive ones. They introduce implicit control-flow and require reasoning about repeated function calls [33]. These aspects are not fully captured by the structural metrics derived from control-flow and data-flow graphs. Thus, the observed correlation may not reflect a general structural effect, but rather the presence of a qualitatively different type of problem.

To better understand whether the absence of correlations between other structural metrics and correctness is genuine or potentially masked by the distinct nature of recursive snippets, we perform an additional analysis excluding recursive cases.

Correlation Structure Without Recursive Snippets. To assess whether recursion masks other relationships, we repeat the analysis excluding recursive snippets.

The overall correlation structure among structural metrics remains largely unchanged. In particular, variable-related and size-related metrics continue to show strong and statistically significant positive correlations with response time (see figure 4.4), confirming that data-flow related complexity is robustly associated with increased cognitive effort.

However, no structural metrics exhibit a statistically significant correlation with task correctness after removing recursion. This reinforces the observation that correctness is largely not explained by the structural metrics considered in this study.

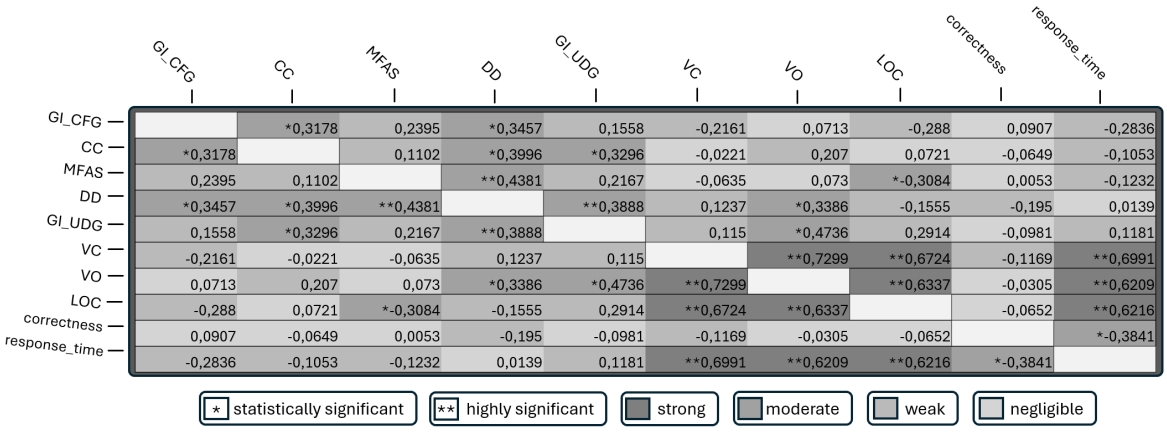


Figure 4.4: This figure shows the results of the correlation analysis without recursion snippets in the dataset.

Summary of Findings. Across both analyses, two consistent observations emerge. First, structural metrics, particularly those related to variable usage and program size, show strong and robust associations with *response time*, indicating their relevance for cognitive effort. Second, task correctness shows almost no systematic relationship with these structural metrics. The only observed is driven by recursion, which represents a special case not fully captured by the metric set.

These findings suggest that while data-flow metrics provide meaningful insights into the effort required for comprehension, they do not directly explain whether a task is solved correctly.

4.1.2 Results for RQ2

To answer RQ_2, which investigates how combinations of structural metrics jointly explain comprehension performance, we construct mixed-effects models for both response time and task correctness. The goal is to identify which metrics or combinations provide the most relevant predictive signal for code comprehension. For both dependent variables, full additive models and stepwise (build-up) models were evaluated, with and without recursive snippets in the dataset. Based on a comparison of predictive performance and model interpretability, we select a subset as final models for further analysis and reporting.

Table 4.1 provides an overview of the models used for response time and their performance metrics and Table 4.2 shows the same for task correctness.

Table 4.1: Overview of the final LMMs for response time, including additive and stepwise model constructions with and without recursion. The table reports the selected fixed effects for each model and their predictive performance in terms of marginal R^2 and RMSE.

Model	Recursion	Final Model	Performance	
			R^2	RMSE
Additive	No	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC	0.296	32.16
Stepwise	No	LOC + VC + CC	0.282	32.49
Additive	Yes	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC + REC	0.263	32.12
Stepwise	Yes	VC + (GI_CFG * LOC)	0.233	32.76

Table 4.2: Overview of the final GLMMs for task correctness, including additive and stepwise model constructions with and without recursion. The table shows the selected predictors and their classification performance measured by ROC-AUC, LogLoss and accuracy.

Model	Recursion	Final Model	Performance		
			ROC-AUC	LogLoss	Accuracy
Additive	No	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC	0.586	0.542	0.78
Stepwise	No	(DD * LOC) * (CC + GI_DFG)	0.737	0.530	0.78
Additive	Yes	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC + REC	0.6647	0.566	0.74
Stepwise	Yes	((((REC + DD) * CC) + GI_DFG) + VO) * (VC + LOC)	0.754	0.547	0.74

The tables 4.1 and 4.2 highlight that stepwise models tend to produce more compact representations for response time while generating highly complex interaction structures for task correctness.

To assess the improvement of the models by the structural metrics, we compare all models against baseline models that include only an intercept and random effects (participant and snippet), i.e., models without any structural predictors. For response time, the baseline model yields an R^2-value close to zero and RMSE values between 37 and 38 indicating that, without structural information, the model effectively predicts the global mean response time. In contrast, the inclusion of structural metrics leads to substantial improvements. For example the additive model without recursion (Table 4.1) achieves an R^2-value of 0.296 and an RMSE of 32.16. Similarly, even the more compact stepwise model (LOC + VC + CC) reaches an R^2-value of 0.282. These results illustrate that metrics such as lines of code, variable count and cyclomatic complexity contribute meaningfully to explaining response time, reducing prediction error by approximately 5-6 seconds. However, a considerable portion of the variance remains unexplained.

For task correctness, the baseline model achieves a ROC-AUC of 0.5, corresponding to random classification, while accuracy remains relatively high due to class imbalance (e.g., approximately 70% correct vs 30% incorrect responses). Compared to this baseline, all reported models show improved discriminative performance. For instance, the stepwise model including recursion (Table 4.2) achieves the highest ROC-AUC of 0.754. This model includes complex interaction terms such as combinations of recursion count, dependency degree, cyclomatic complexity and variable-related metrics, indicating that correctness is influenced by interactions between multiple structural properties rather than single predictors. However, improvements in LogLoss are more moderate (e.g. 0.547) suggesting that, while the models can distinguish between correct and incorrect responses, their probabilistic predictions remain uncertain. Overall, these results indicate that structural metrics provide useful predictive signal, but are not sufficient to fully explain comprehension outcomes. In particular, the stepwise model including recursion achieves the highest ROC-AUC of 0.754, indicating a substantial improvement over chance-level prediction.

4.1.3 Results for RQ3

To address RQ_3, where we investigate how structural code metrics relate to the variability of comprehension performance across participants, we model deviation-based outcome variables using linear-mixed-effects models. Specifically, we analyze *duration deviation* and *correctness deviation*, representing deviations from snippet-level average response time and correctness, respectively.

As in RQ_2, we compare baseline models (without structural predictors) to additive models including all structural metrics and evaluate performance with and without recursive snippets.

Tables 4.3 and 4.4 provide an overview of the models and their predictive performance.

Table 4.3: Overview of the LMMs for duration-deviation outcomes, including additive models with and without recursion. The table reports the fixed effects and their predictive performance in terms of R^2 and RMSE.

Recursion	Model	Performance	
		R^2	RMSE
No	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC	0.169	17.18
Yes	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC + REC	0.142	17.50

Table 4.4: Overview of the LMMs for correctness-deviation outcomes, including additive models with and without recursion. The table reports the fixed effects and their predictive performance in terms of R^2 and RMSE.

Recursion	Model	Performance	
		R^2	RMSE
No	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC	0.080	0.240
Yes	GI_CFG + CC + MFAS + DD + GI_DFG + VC + VO + LOC + REC	0.076	0.239

The table 4.3 show that additive models provide moderate predictive performance for duration deviation, while performance for correctness deviation remains limited.

To assess the improvement of the models by structural metrics, we again compare them against baseline models without structural predictors. For duration deviation, baseline models achieve R^2 values close to zero and RMSE values around 18.9, indicating that without structural information, the model effectively predicts the average deviation.

The inclusion of structural metrics leads to noticeable improvements. For example, the additive model without recursion (Table 4.3) achieves an R^2-value of 0.169 and reduces RMSE

to 17.18. Similarly, the model including recursion reaches an R^2-value of 0.142 with an RMSE of 17.50. These results indicate that structural metrics provide useful predictive signal for variability in response time, reducing prediction error by approximately 1.5 to 2 seconds. However, a large portion of the variance remains unexplained.

For correctness deviation, baseline models again show no explanatory power ($R^2 \approx 0$). Compared to this baseline, the inclusion of structural metrics leads only to small improvements. The best-performing model (without recursion, Table 4.4) achieves an R^2-value of 0.080, with only a marginal reduction in RMSE.

Overall, these results indicate that structural metrics provide limited explanatory power for variability in comprehension performance. While improvements are more pronounced for duration deviation, correctness deviation remains largely unexplained by the considered structural properties.

4.2 Discussion

4.2.1 Discussion for RQ1

The correlation analysis provides several insights into how structural properties of source code relate to human comprehension performance.

A central observation is that, across the full dataset, only recursion count shows a significant moderate negative correlation with task correctness. This initially suggests that recursion plays a key role in determining whether a task is solved correctly.

However, this interpretation must be treated with caution. Recursive snippets differ from non-recursive ones, as they require reasoning about repeated function calls, implicit control-flow and stack behavior [33]. As a result, recursion represents a form of *conceptual complexity* that is only partially reflected in the extracted structural properties. This suggests that recursion introduces aqualitatively different type of difficulty rather than simply increasing structural complexity along the same dimension as other metrics. In this sense, recursion may act as a *confounding factor* that dominates the relationship with correctness and masks more subtle effects of other structural properties.

To address this, we repeat the analysis without recursive snippets. The results show that once recursion is removed, *no structural metric exhibits a significant correlation with correctness*. This strengthens the interpretation that correctness is largely influenced by factors beyond the structural properties captured in this study. One plausible explanation is that correctness depends more strongly on *non-structural aspects*, such as semantic clarity, familiarity with programming constructs, or individual problem-solving strategies. While structural complexity may increase the difficulty of a task, participants may still arrive at the correct solution by investing additional time or applying compensatory strategies. This interpretation is supported by the consistent *negative correlation between response time and correctness*, indicating that tasks requiring more time are associated with lower accuracy. This indicates that increased effort does not necessarily translate into successful comprehension.

In contrast to correctness, response time shows strong and consistent relationships with several structural metrics, particularly those related to variable usage and program size. For example, metrics such as variable count and LOC exhibit clear positive correlations

with response time, indicating that larger and more data-flow intensive programs require more cognitive effort. From a cognitive perspective, this is intuitive. Understanding how variables are defined, modified and propagated through a program requires maintaining and updating a mental representation of program state, which increases cognitive load.

Interestingly, traditional control-flow metrics such as cyclomatic complexity and minimum feedback arc set show only weak and non-significant relationships with both correctness and response time. This indicates that control-flow complexity alone does not adequately capture the aspects of program structure that influence comprehension difficulty in the analyzed dataset.

Overall, the results suggest a clear distinction between two dimensions of comprehension. First, effort (response time) is strongly influenced by structural properties, particularly data-flow complexity. Second, success (correctness), is not well explained by these metrics and appears to depend on additional factors, with recursion representing a special case of conceptual difficulty.

These findings partically align with prior work by [31], who report that commonly used structural metrics such as lines of code and dependency degree exhibit small to medium correlations with comprehension performance, while cyclomatic complexity does not show significant relationships. In our results, we replicate the absence of significant correlations for cyclomatic complexity, reinforcing the conclusion that traditional control-flow metrics are limited in their ability to capture cognitively relevant aspect of program comprehension. However, in contrast to [31], we do not observe meaningful correlations for dependency degree, which shows only weak and non-significant relationships with comprehension performance in our dataset.

Furthermore, our analysis extends the prior work by incorporating a broader set of data-flow based metrics. While dependency degree alone does not exhibit strong effects, other data-flow related measures, particularly variable-related metrics, show consistent and strong relationships with response time. This suggests that data-flow complexity is indeed relevant for comprehension, but may not be adequately captured by a single metric such as dependency degree.

Therefore, these findings provide an answer to RQ_1. Data-flow related metrics show stronger and more consistent relationships with comprehension effort that traditional control-flow metrics, but neither type of metric reliably explains task correctness.

4.2.2 Discussion for RQ2

The results of RQ_2 show that structural code metrics provide only limited explanatory power for human program comprehension.

For response time, models explain up to approximately 30% of the variance. While this indicates that structural properties are relevant, a large portion of variability remains unexplained. For task correctness, additive models perform close to random classification, while stepwise models improve performance through complex interaction terms. This suggests that correctness depends on interactions between structural properties rather than on individual metrics alone. However, this improvement comes at a cost of interpretability and potential overfitting [16].

A key insight concerns the role of random effects. Baseline models show that random effects dominate response time but contribute only marginally to correctness. This indicates that correctness is primarily driven by fixed effects, whereas response time is influenced by both structural properties and random effects associated with participants and code snippets.

Given these limitations, alternative modeling strategies may be more appropriate. In particular, we conducted exploratory analyses in which comprehension outcomes were aggregated at the snippet level (e.g. by averaging response time and correctness across participants) and modeled using linear models without random effects. For example, a stepwise model for aggregated response time achieved an R^2-value of approximately 0.51 with substantially lower predictive error (RMSE $\approx$ 17.5), compared to the mixed-effects models on the full data. This suggests that a substantial portion of the variability at the individual level may be dominated by noise, such as differences in participant ability, attention fluctuations and other uncontrolled factors influencing response time and correctness. Aggregation reduces this variability by averaging over participants, thereby helping to reveal more stable underlying relationships between structural metrics and comprehension.

These findings are in line with the results reported by [24], who show that even combinations of structural code metrics provide only limited predictive accuracy for code comprehension. Notably, their study focuses primarily on traditional code metrics such as lines of code and cyclomatic complexity, without incorporating detailed data-flow based measures.

Similarly, in our study, combining multiple metrics improves model performance compared to individual predictors, but the overall explanatory power remains moderate. In particular, while multivariate models capture a large portion of variance in response time, a substantial amount of variability remains unexplained and predictive performance for task correctness is only improved through complex interaction structures.

Together, these results reinforce the conclusion that structural metrics alone are insufficient to fully capture the factors that determine human code comprehension.

4.2.3 Discussion for RQ3

The results of RQ_3 show that structural metrics provide only limited explanatory power for variability in comprehension performance across participants.

For duration deviation, the models achieve R^2-values of up to 0.169 (Table 4.3), indicating that structural metrics capture part of the variability in response times. In particular, variable count and LOC, as well as recursion, contribute to explaining variability in response time. This suggests that larger and structurally more complex snippets not only increase response time on average (as shown in RQ2), but also lead to less consistent behavior across participants.

However, a substantial portion of the variance remains unexplained. Even the best-performing models explain less than 20% of the total variance, indicating that additional factors beyond structural properties influence how consistently participants understand code.

For correctness derivation, the explanatory power of the models is considerably lower. The best-performing model achieves an R^2-value of 0.08 representing only a small improvement over the baseline. This indicates that structural metrics provide only weak predictive signal for variability in correctness.

Compared to RQ2, this highlights an important difference. While structural metrics show moderate explanatory power for performance, particular for response time, they are less effective in explaining variability across participants. This suggests that the factors influencing how difficult a code snippet is, are not identical to those determining how consistently it is understood.

Overall, these findings indicate that structural metrics contribute to explaing variability in comprehension performance, but their explanatory power remains limited. A large portion of the variance is not captured by the considered metrics, suggesting that additional factors such as semantic properties or individual differences, play an important role.

4.3 Summary of the Findings

Across all three research questions, the results show a consistent but nuanced picture of the role of structural code metrics in explaining code comprehension. In RQ1, individual metrics exhibit only weak to moderate correlations with comprehension outcomes, indicating that no single structural property is sufficient to capture code comprehension. RQ2 demonstrates that combining multiple metrics in mixed-effects models substantially improvements predictive performance, particularly for response time, suggesting that comprehension difficulty arises from the interaction of several structural factors. However, even the best-performing models explain only a portion of the variance and predictive performance for task correctness remains limited.

RQ3 extends these findings by showing that structural metrics are even less effective in explaining variability across participants. While some improvement is observed for response time deviation, the overall explanatory power remains modest and variability in correctness is largely unexplained. Taken together, these results indicate that structural metrics capture important aspects of code complexity, especially when combined, but they do not fully account for comprehension performance or its variability. This suggests that additional factors, such as semantic properties of code, readability and individual differences in cognitive processing, plays a significant role and should be considered in future work.

4.4 Threats to Validity

When evaluating the relationship between structural code metrics and human comprehension performance, several threats to validity must be considered. Following established guidelines in empirical software engineering [39], we distinguish between construct validity, internal validity, conclusion validity and external validity.

Construct Validity. Construct validity concerns whether the chosen metrics and measurements accuractely capture the underlying concepts of interest. We operationalize in this thesis code comprehension using task correctness and response time derived from

existing user studies [4, 32]. While these measures are widely used [24, 32, 36], they only approximate the complexity of code comprehension, which also includes aspects such as mental effort, understanding depth and strategy use [8, 33]. As a result, comprehension may not be fully captured by these observable outcomes.

Similarly, the structural metrics we use in this study represent only a subset of possible code properties [30]. While control-flow and data-flow metrics capture important aspects of program structure, they omit other factors known to influence comprehension, such as naming quality, code layout, semantic clarity and commenting [7]. This may lead to an incomplete representation of the true drivers of comprehension difficulty.

Furthermore, the abstraction levels we use for control-flow graphs (statement-level) and data-flow graphs (AST-level) differ. Although we apply normalization, this mismatch may still influence how well the metrics reflect cognitively relevant structures.

Internal Validity. Internal validity concerns whether the observed relationships can be attributed to the investigated factors rather than confounding variables. A key threat arises from the use of existing datasets, where factors such as participant experience, familiarity with programming constructs and individual problem-solving strategies cannot be fully controlled [40]. Although mixed-effects models account for participant and task variability, unobserved confounders may still bias the results.

Another potential issue is the presence of recursion in a subset of snippets. Recursive code introduces a qualitatively different form of complexity that is not fully captured by the structural metrics. As observed in the analysis, recursion can dominate certain relationships, particularly with task correctness and my therefore distort the interpretation of more general structural effects.

Additionally, preprocessing steps such as normalization, transformation and standardization may influence the statistical relationship between variables [18]. These techniques implicitly assume certain data distributions (e.g., symmetry or log-linearity) and can alter the strength and form of observed relationships, potentially affecting the interpretation of results.

Conclusion Validity. Conclusion validity refers to the reliability of the statistical analysis and the extent to which correct inferences are drawn from the data. One threat arises from the relatively small dataset size, which limits statistical power and may reduce the robustness of estimated effects, particularly in more complex models with multiple predictors and interaction terms.

Moreover, several metrics exhibit skewed and zero-inflated distributions, which affect correlation and regression analyses despite the applied transformations. While Spearman correlation and mixed-effects models are chosen to mitigate these issues, residual distributional effects may remain.

Another concern is the risk overfitting [16], especially in stepwise model construction with higher-order interaction terms. Such models may capture dataset-specific patterns rather than generalizabile relationships, reducing the reliability of conclusions.

External Validity. External validity concerns the generalizability of the results beyond the studied context. A primary limitation is that the analysis is based on relatively small code

snippets. Comprehension of such snippets may differ substantially from understanding larger, real-world software systems that involve additional complexities such as architectural strcture, long-range dependencies and tool support [25].

Furthermore, the dataset originates from a controlled user study with a specific participant population and programming language (Java). As a result, the findings may not generalize to other developer populations, levels of expertise or programming languages.

Finally, the controlled experimental setting differs from real-world development environments, where factors such as time pressure, collaboration and tooling influence comprehension [39]. These contextual differences further limit the generalizability of the results

Overall, while the study provides meaningful insights into the relationship between structural code metrics and comprehension performance, these threats should be considered when interpreting the results and their applicability to broader contexts.

5

Related Work

Related Work

Research on program comprehension and software understandability spans software engineering, empirical software engineering, human-computer interaction and cognitive science. This section summarizes prior work on (1) cognitive foundations of program comprehension, (2) structural and graph-based code metrics and (3) empirical models of readability and understandability.

Program Comprehension. Code comprehension is a central activity in software development [8]. Prior work describes comprehension as the construction of mental representations of program structure and behavior [33]. These representations include both control-flow abstractions and data-flow relationships, highlighting that understanding code involves integrating multiple structural aspects. Subsequent work has highlighted the importance of tools, strategies and developer behavior in comprehension tasks [38]. Since then, numerous empirical studies have investigated how developers understand code and which factors influence performance [36]. A recent systematic mapping study covering four decades of research highlights the diversity of experimental designs, measurement methods and results in program comprehension research [40].

Behavioral studies using eye-tracking reveal systematic differences between novices and experts in reading strategies and attention patterns [10]. EEG-based studies further show that cognitive load varies significantly between programmers during comprehension tasks [37]. Peitek et al. [31] extended this line of work by combining brain and behavioral measurements. In addition to collecting response time and correctness, they analyzed correlations between structural code metrics and comprehension performance. Their results indicate that metrics such as lines of code, Halstead complexity and dependency degree exhibit small to medium correlations with performance, while cyclomatic complexity shows no significant correlation in their setting.

Structural Code Metrics. Structural code metrics aim to quantify properties of source code related to complexity and maintainability [15]. One of the earliest and most widely used measures is cyclomatic complexity, introduced by McCabe [28], which captures the number of independent control-flow paths in a program.

Aho et al. [1] introduce Control-Flow Graphs (CFGs), which provide a formal representation of program execution structure and form the basis for many complexity measures. Extensions of CFG-based analysis include metrics such as the minimum feedback arc set [41], which captures cyclic dependencies in control-flow.

Data-flow representations complement control-flow analysis by modeling dependencies between variable definitions and uses. Metrics such as dependency degree quantify the complexity of these relationships and have been proposed as indicators of low-level structural complexity [5].

However, the interpretation of code metrics is not trivial. Mamun et al. [27] shows that correlations between metrics can depend on the measurement choices, highlighting the need for careful statistical analysis when relating metrics to external outcomes.

Readability and Understandability. Numerous studies have investigated whether source code metrics can predict readability or understandability. Buse et al. [9] proposed readability models based on structural code features and human ratings. Subsequent work simplified these models and emphasized the strong influence of code size on prediction performance [34].

Later approaches incorporated additional feature types. Dorn [13] introduced visual and spatial features, while Scalabrino et al. [35] combined structural and textual features, showing improved predictive performance.

Lavazza et al. [24] investigated the relationship between code metrics and understandability measured via maintenance task performance. Their results show that even combinations of metrics provide only limited predictive accuracy, suggesting that structural metrics alone are insufficient to fully explain code understandability.

Additional studies confirm these mixed findings. Muñoz Barón et al. [29] report moderate correlations between Cognitive Complexity and understandability measures, while other work highlights the limited ability of existing metrics to capture changes in readability over time [14].

Summary. Previous studies have shown that program comprehension is influenced by both cognitive processes and the structural properties of the source code. Although numerous metrics and models have been proposed, the empirical findings regarding their relationship to human comprehension performance is inconsistent. Correlation based studies suggest that while structural properties are relevant, they are not sufficient on their own to fully explain code comprehension performance. This motivates further research into how different structural representations relate to empirically measured comprehension outcomes.

6 Concluding Remarks

In this chapter, we summarize the key findings of this thesis and discuss directions for future research.

6.1 Conclusion

In this thesis, we examine how structural code metrics relate to program comprehension, considering both individual performance and differences between participants. Motivated by prior studies that report mixed results regarding the explanatory power of such metrics [19, 24, 40], we analyze how structural properties are associated with task correctness and response time.

The results show that structural metrics are related to comprehension performance, but their explanatory power remains limited. Consistent with prior studies [31], size-related measures show comparatively stable relationships with performance outcomes. Among structural measures, data-flow related metrics such as variable count and variable overwrites also exhibit meaningful associations. In contrast, traditional control-flow metrics tend to provide weaker or less consistent explanations.

The analysis of variability suggests that structural properties also influence how consistently code is understood across participants. This implies that code characteristics affect not only overall performance levels but also the degree to which comprehension outcomes vary between individuals.

Taken together, these findings suggest that program comprehension arises from a combination of structural aspects and additional factors, including commenting practices, identifier naming, code layout and semantic clarity. While structural metrics capture important elements of code comprehension, they do not fully explain the observed performance differences. In particular, the observed variability highlights limitations of approaches that rely exclusively on mean-based analyses.

6.2 Future Work

The results of this work suggest several directions for future research. First, the limited explanatory power of individual metrics suggests that future work should consider combinations of structural metrics, possibly complemented by cognitive or behavioral measures, in order to better capture the multi-dimensionality of code comprehension.

Second, future work should extend the set of considered metrics beyond purely structural properties. Including measures related to code comments, identifier naming quality and

formatting or layout may provide a more comprehensive representation of code quality. Such features are known to influence readability and comprehension [35] and may improve the predictive performance of statistical models by capturing factors that are not reflected by structural metrics alone.

Third, integrating additional data sources, such as eye-tracking, could help to better understand the underlying processes that drive differences in comprehension and provide a closer link between comprehension performance and structural code properties.

Finally, future work should investigate the role of contextual factors, such as developer experience, task characteristics and code context. Extending the experimental setup to more realistic programming scenarios and larger code bases would improve the external validity of the findings and contribute to a more comprehensive understanding of program comprehension.

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistant has been used: ChatGPT (GPT-5).

The tool was used in a supportive manner to clarify and identify correct LaTeX commands and syntax, for example for tables, formulas and references, as well as to polish and improve the linguistic quality of text. This included improving phrasing, sentence flow, the use of connective words and restructuring or merging sentences to enhance readability.

In practice, I drafted the text independently, then submitted selected passages to the model for language improvements. The revised outputs were subsequently reviewed and carefully adapted to ensure accuracy, coherence and consistency with my original intended meaning and style.

Bibliography

[1] Aho Alfred, S Monica, Sethi Ravi, Ullman Jeffrey D, and Others. *Compilers Principles, Techniques*. Chapters 2.8.2 and 8.4. Pearson, 2007.

[2] Frances E Allen. "Control Flow Analysis." In: *ACM Sigplan Notices* 5.7 (1970), pp. 1–19.

[3] R Harald Baayen, Douglas J Davidson, and Douglas M Bates. "Mixed-effects Modeling with Crossed Random Effects for Subjects and Items." In: *Journal of memory and language* 59.4 (2008), pp. 390–412.

[4] Annabelle Bergum, Norman Peitek, Maurice Rekrut, Janet Siegmund, and Sven Apel. "On the Influence of the Baseline in Neuroimaging Experiments on Program Comprehension." In: *ACM Transactions on Software Engineering and Methodology* 35.3 (2026), pp. 1–27.

[5] Dirk Beyer and Ashgan Fararooy. "A Simple and Effective Measure for Complex Low-Level Dependencies." In: *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010, pp. 80–83.

[6] Christopher M Bishop and Nasser M Nasrabadi. *Pattern Recognition and Machine Learning*. Vol. 4. 4. Chapter 1.5.5. Springer, 2006.

[7] Jürgen Börstler and Barbara Paech. "The Role of Method Chains and Comments in Software Readability and Comprehension—an Experiment." In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 886–898.

[8] Ruven Brooks. "Towards a Theory of the Comprehension of Computer Programs." In: *International journal of man-machine studies* 16 (1983), p. 1065366.

[9] Raymond PL Buse and Westley R Weimer. "Learning a Metric for Code Readability." In: *IEEE Transactions on software engineering* 36.4 (2009), pp. 546–558.

[10] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. "Eye Movements in Code Reading: Relaxing the Linear Order." In: *2015 ieee 23rd international conference on program comprehension*. IEEE. 2015.

[11] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. routledge, 2013.

[12] Carly Domicolo and Hosam Mahmoud. "Degree-based Gini Index for Graphs." In: *Probability in the Engineering and Informational Sciences* 34.2 (2020), pp. 157–171.

[13] Jonathan Dorn. "A General Software Readability Model." In: *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)* 5 (2012), pp. 11–14.

[14] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. "Improving Source Code Readability: Theory and Practice." In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 2–12.

[15] Norman Fenton and James Bieman. *Software Metrics: a Rigorous and Practical Approach*. CRC press, 2014.

[16] Andrew Gelman and Jennifer Hill. *Data Analysis using Regression and Multilevel/Hierarchical Models*. Cambridge university press, 2007.

[17] Felipe L Gewers, Gustavo R Ferreira, Henrique F De Arruda, Filipi N Silva, Cesar H Comin, Diego R Amancio, and Luciano da F Costa. "Principal Component Analysis: A Natural Approach to Data Exploration." In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–34.

[18] Hanan M Hammouri, Roy T Sabo, Rasha Alsaadawi, and Khalid A Kheirallah. "Handling skewed data: A Comparison of Two Popular Methods." In: *Applied Sciences* 10.18 (2020), p. 6247.

[19] Gao Hao, Haytham Hijazi, João Durães, Júlio Medeiros, Ricardo Couceiro, Chan Tong Lam, César Teixeira, João Castelhano, Miguel Castelo Branco, Paulo Carvalho, et al. "On the Accuracy of Code Complexity Metrics: A Neuroscience-Based Guideline for Improvement." In: *Frontiers in Neuroscience* (2023).

[20] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, et al. *An Introduction to Statistical Learning: with Applications in R*. Vol. 103. Chapters 3.1.3, 4.4.3, 6.7. Springer, 2013.

[21] Ian T Jolliffe and Jorge Cadima. "Principal Component Analysis: A Review and Recent Developments." In: *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences* 374.2065 (2016).

[22] Barbara Kitchenham, Lech Madeyski, and Pearl Brereton. "Meta-analysis for Families of Experiments in Software Engineering: A Systematic Review and Reproducibility and Validity Assessment." In: *Empirical Software Engineering* 25.1 (2020), pp. 353–401.

[23] Michael H Kutner, Christopher J Nachtsheim, and John Neter. *Applied Linear Statistical Models*. Chapter 19, pp. 836–839. McGraw-Hill, 1984.

[24] Luigi Lavazza, Sandro Morasca, and Marco Gatto. "An Empirical Study on Software Understandability and its Dependence on Code Characteristics." In: *Empirical Software Engineering* 28.6 (2023), p. 155.

[25] Cristina V Lopes and Joel Ossher. "How Scale Affects Structure in Java Programs." In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 675–694.

[26] Edward S Lowry and Cleburne W Medlock. "Object Code Optimization." In: *Communications of the ACM* 12.1 (1969), pp. 13–22.

[27] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. "Effects of Measurements on Correlations of Software Code Metrics." In: *Empirical Software Engineering* 24.4 (2019), pp. 2764–2818.

[28] Thomas J McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320.

[29] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. "An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability." In: (2020), pp. 1–12.

[30] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. "Source Code Metrics: A Systematic Mapping Study." In: *Journal of Systems and Software* 128 (2017), pp. 164–197.

[31] 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)*. IEEE. 2021, pp. 524–536.

[32] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. "Correlates of Programmer Efficacy and Their Link to Experience: A Combined EEG and Eye-Tracking Study." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 120–131.

[33] Nancy Pennington. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs." In: *Cognitive psychology* 19.3 (1987), pp. 295–341.

[34] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. "A Simpler Model of Software Readability." In: *Proceedings of the 8th working conference on mining software repositories*. 2011, pp. 73–82.

[35] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. "A Comprehensive Model for Code Readability." In: *Journal of Software: Evolution and Process* 30.6 (2018), e1958.

[36] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging." In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 378–389.

[37] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. "Measuring Neural Efficiency of Program Comprehension." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 140–150.

[38] M-A Storey. "Theories, Methods and Tools in Program Comprehension: Past, Present and Future." In: *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE. 2005, pp. 181–191.

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in Software Engineering*. Vol. 236. Chapter 2.7 pp. 23ff, Chapter 9.8, p. 131 ff. Springer, 2012.

[40] Marvin Wyrich, Justus Bogner, and Stefan Wagner. "40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study." In: *ACM Computing Surveys* 56.4 (2023), pp. 1–42.

[41] D Younger. "Minimum Feedback Arc Sets for a Directed Graph." In: *IEEE Transactions on Circuit Theory* 10.2 (1963), pp. 238–245.