Cognitive Modeling of Code Readability: Working Memory and Semantic Recoverability

Tairan Wang

MEng Mathematical Computation
Department of Computer Science
University College London

Supervisor: Prof. Earl T. Barr

Submission Date: 24th April 2026

Abstract

We study code readability from a cognitive perspective and propose two complementary metrics. CognaScore models readability as short-term working memory load, based on lexeme diversity and semantic abstraction. PreFixScore measures readability through prefix-based semantic recoverability, using predictability as a proxy for long-term memory.

We evaluate both methods on two datasets with different granularity: the JetBrains dataset (function-level) and the Schnappinger dataset (file-level). CognaScore shows stable alignment with human judgments across both settings and retains predictive signal beyond code length. In contrast, PreFixScore performs well on short snippets but fails to generalize to larger programs, where it exhibits weak or negative correlation.

These results suggest that code readability depends on both local cognitive load and global semantic structure, and cannot be captured by a single factor.

Contents

1 Introduction

Code readability refers to the degree to which source code can be read, understood, and reasoned about by human developers.

The modern software engineering lifecycle consists of two major phases: the software development phase and the software maintenance phase. Software systems that lack subsequent updates rapidly become obsolete [30]; therefore, in modern software engineering, the maintenance phase can account for approximately 70% to 90% [21] of the entire software lifecycle.

A large body of empirical studies has shown that, in both the development and maintenance phases, developers spend only about 5% of their effective working time writing code, whereas approximately 50% to 70% [46, 70] of their time is devoted to understanding existing code.

With the rapid improvement of code-related capabilities in large language models in recent years, LLMs have been systematically integrated into the software development process, particularly to assist human developers with code implementation and testing [7]. This shift implies that developers increasingly spend more time reading and reviewing code generated by LLMs, rather than writing code entirely by themselves.

Existing studies have shown that code produced by LLMs still has room for improvement in terms of readability, maintainability, and complexity [57, 37, 36, 47]. One possible reason is that existing evaluation practices for LLM-generated code primarily focus on functional correctness, while non-functional aspects such as readability receive comparatively less attention [32, 54]. Therefore, a reliable and interpretable code readability metric is particularly important.

1.1 Tacit Knowledge and Subjective Judgments of Code Readability

Tacit knowledge refers to the form of knowledge that is rooted in personal experience, intuition, and know-how, and cannot be easily articulated, codified or transferred through formal language or documentation [31].

Commonly cited examples of tacit knowledge include skills such as swimming, recognizing familiar individuals through facial perception, and making intuitive judgments. Some studies argue that code readability is inherently subjective [9, 56, 59], and that inter-rater agreement among human developers is low [62], rendering readability annotations unreliable or difficult to generalize [8, 62]. From this perspective, code readability can be viewed as a form of tacit knowledge, relying on human intuition and experiential judgment.

However, other studies have shown that, at least within the same organization, developers with similar background knowledge tend to produce consistent and reliable readability assessments [61]. Moreover, there exist widely recognized structural factors that influence code readability, such as variable and identifier naming [10, 20, 58, 11], the use of magic numbers [20], nesting depth and number of branching [53, 66].

This study does not deny the presence of subjective factors in code readability. However, we argue that code readability also involves stable and interpretable cognitive and structural factors, which play a decisive role in code comprehension and can be modeled and quantified to a large extent.

1.2 Toward Operational Readability Scores

Prior research has explored a range of automated approaches to modeling code readability. Early work primarily relied on structural complexity metrics (Section 2.2), while later studies formulated readability as a supervised learning problem using diverse code features (Section 2.3). More recently, human-centered research has attempted to ground readability in cognitive processes involved in code comprehension (Section 2.5). Despite these efforts, existing approaches remain fragmented and context-dependent, and none has yet yielded a general and transferable model of code readability [58, 62].

To address these limitations, this study adopts a perspective grounded in human cognition and memory and investigates how code readability can be modeled in terms of cognitive load. We propose two complementary models grounded in working memory that are motivated by distinct memory mechanisms involved in code comprehension and provide a generalizable assessment of code readability aligned with human perception.

CognaScore models code readability in terms of the working memory demands experienced during code comprehension. Grounded in working memory theory, it accounts for both the amount of information that developers must actively maintain while reading code and the abstraction of semantically related elements into unified mental representations through chunking. By explicitly modeling

these mechanisms within a single working memory framework, CognaScore provides a generalizable assessment of code readability that is aligned with human perception.

PreFixScore, in contrast, adopts a top-down perspective on code comprehension. Instead of modeling local cognitive load, it evaluates readability based on the extent to which the full program can be semantically recovered from a partial prefix. This formulation captures the role of prior knowledge and long-term memory, reflecting how developers form and validate hypotheses during code reading.

Together, these two models provide complementary views of readability: CognaScore focuses on local working memory demands during incremental processing, while PreFixScore captures global semantic predictability. By combining these perspectives, we aim to better approximate the cognitive processes underlying human code comprehension.

2 Background and Related Work

This section reviews prior work on code readability and traces the development of the field from different modeling perspectives. We first introduce the concept of code readability and clarify its relationship with related notions. We then review automated modeling approaches based on structural complexity and machine learning, and discuss their limitations. Building on these observations, we further examine research that approaches code readability from a human-centered perspective, with a focus on cognitive load and working memory mechanisms, which provide the theoretical foundation for the models proposed in this work.

2.1 Code Readability: Concepts and Definitions

In a broad sense, for any text-based content, readability refers to the ease with which a text can be understood by its readers. Readability is a multidimensional concept that involves multiple factors within the reading material and their relationships, which together affect how effectively readers can use the content [19]. Therefore, readability is not limited to the difficulty of vocabulary or syntax, but is also related to information presentation, visual design, structure, and organization [50, 18]. In addition, readability is influenced by the reader's comprehension ability and background knowledge [18, 19].

In the context of code readability, this general notion of readability has been extended to source code. Code readability concerns how easily humans can understand a piece of code, including its purpose, control flow, and runtime behavior. In many empirical studies, readability is assessed through human judgments, where developers are asked to assign subjective scores to code samples based on their perceived ease of understanding [9, 24, 62]. Existing readability models are commonly trained and/or evaluated using such human-annotated readability scores. [9, 59, 56, 43]

Two related but potentially confusing concepts are software maintainability and code understandability. In software engineering, maintainability is commonly defined as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [1]. Under this definition, maintainability encompasses code readability, but also includes additional aspects such as extensibility, flexibility, and the availability and quality of documentation.

Understandability typically refers to the degree to which developers can construct a semantic model of a program and comprehend its behavior under a given context and task. Its influencing factors extend beyond the readability of source code itself, and include program documentation, domain knowledge, and established development conventions [39, 58]. At the source code level, understandability and readability are highly related and partially overlapping concepts. In empirical studies, some work employs comprehension-related behavioral measures—such as comprehension time, accuracy, or modification quality—to validate or approximate code readability [55, 61]. In this work, we view readability as a code property that supports understanding, and model it quantitatively within a cognitive load framework.

2.2 Structural Complexity as a Proxy for Readability

Code complexity metrics are a class of rule-based, automatically computable structural measures that are widely used as proxies for code readability and understandability.

Cyclomatic Complexity (McCabe Cyclomatic Complexity) represents a program as a control flow graph G and is defined as:

$$\text{Cyclomatic Complexity} = E - N + 2P,$$

where E denotes the number of edges in the control flow graph, N denotes the number of nodes, and P denotes the number of connected components [40].

Halstead metrics (Halstead Complexity) models a program as a symbolic system composed of operators and operands [29]. Let n_1 and n_2 denote the numbers of distinct operators and operands, and let N_1 and N_2 denote the total numbers of operator and operand occurrences, respectively. Based on these quantities, Halstead defines:

$$\begin{aligned} \text{Program Vocabulary:} \quad & n = n_1 + n_2, \\ \text{Program Length:} \quad & N = N_1 + N_2, \\ \text{Volume:} \quad & V = N \cdot \log_2(n), \\ \text{Difficulty:} \quad & D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}, \\ \text{Effort:} \quad & E_H = D \cdot V. \end{aligned}$$

Another widely used complexity metric is Cognitive Complexity, introduced by SonarSource in 2016 to measure code understandability [12]. Although it is a proprietary commercial metric, it has gained considerable influence in the developer community and is often adopted as an alternative to Cyclomatic Complexity [13]. Its design is guided by several principles, including that breaks in linear flow increase cognitive load, each additional nesting level increases cognitive load, and method calls are considered free.

However, existing empirical studies have shown that structural complexity metrics typically exhibit only limited and unstable correlations with human-perceived readability and understandability.

Cyclomatic complexity primarily reflects the number of execution paths in a program, and in many cases, path count alone does not sufficiently explain the cognitive effort required for code comprehension [63]. As a result, modern studies generally treat cyclomatic complexity as a structural metric rather than a direct measure of code readability.

A limitation of Halstead complexity lies in the definition of operators and operands, which can be inconsistent across modern programming languages and may lead to unstable measurements [3]. More importantly, both cyclomatic complexity and Halstead complexity are strongly correlated with code length [38], and have been shown to be insufficient as stable proxies for code readability [4] or understandability [34, 26].

Some studies have shown that Cognitive Complexity performs comparably to Cyclomatic Complexity [13, 48]; however, it still cannot fully serve as a proxy for code understandability [48, 26, 34] or readability [20].

2.3 Machine-Learning-Based Readability Models

Other lines of work view code readability as a complex construct that involves subjectivity and inconsistency in human judgment. In the absence of a widely accepted and formal definition of readability, manually modeling readability based on individual code features is inherently challenging. As a result, recent studies have commonly formulated code readability as a supervised learning problem, approximating human readability perception by incorporating a diverse set of code features into machine learning models.

In 2009, Buse and Weimer proposed a binary classification model based on 25 features, covering multiple aspects such as structural, syntactic, lexical, and layout-related properties of code, and reported that the model achieved approximately 80% effectiveness in predicting readability judgments, performing on par with, and on average better than, individual human annotators [9].

In 2011, Posnett's model, developed through forward stepwise refinement, uses only three features—lines of code, Halstead volume V, and character entropy—yet outperforms Buse's model on Buse's dataset [56]. The model is defined as:

$$\text{readability} = \frac{1}{1 + e^{-z}}, \quad z = 8.87 - 0.033V + 0.40\text{Lines} - 1.5\text{Entropy}$$

Posnett's model exhibits relatively high interpretability and highlights the importance of code size in readability modeling.

In 2018, Scalabrino et al. proposed a more comprehensive readability prediction model, highlighting the importance of textual aspects of source code, including dictionary-based lexical analysis, the presence of abbreviations, and the consistency between identifiers and comments [59]. Building upon the

features used in earlier models by Buse, Posnett, and Dorn [24], the model incorporates a richer set of textual features and combines them with structural and visual/layout features into a unified framework. Empirically, this model outperforms all prior readability models.

In 2018, Mi et al. proposed a deep learning-based model that avoids manual feature engineering by directly transforming source code into numerical matrices and applying a CNN-based architecture under a supervised learning setting [43]. Their approach outperforms prior methods, including the model by Scalabrino et al. In 2022, Mi et al. further introduced a more complex hybrid neural network that combines CNN, BERT, and BiLSTM components, achieving additional performance improvements over previous approaches [42].

However, existing studies indicate that no current readability model is able to accurately and reliably assess code readability or understandability in a manner that is consistently aligned with human perception [58, 62]. Moreover, prior work has shown that when human developers perform modifications intended to improve code readability, these improvements are often not reflected in the outputs of existing models [52]. In other words, readability improvements perceived by developers do not necessarily manifest as measurable gains under current readability metrics.

2.4 LLM-based Readability Assessment

In recent years, the capabilities of large language models (LLMs) have improved substantially, achieving strong performance in software engineering tasks such as code generation, program repair, and automated testing. Recent studies have explored the use of LLMs to assess code readability, either by directly predicting readability judgments or by deriving proxy signals from the model's internal behavior.

Dillmann et al. (2024) quantify code predictability by computing the weighted average token-level cross-entropy of a language model, measuring the model's surprise when processing code[23]. The signal correlates with code readability and understandability, but its standalone predictive power is weak, with logical lines of code (LLOC) acting as a confounding variable. After incorporating LLOC, cross-entropy provides only minimal additional predictive information, and performance improves only marginally compared to a model using LLOC alone.

Wendkûuni C. Ouédraogo et al. (2025) conduct a large-scale study showing that prompt-based LLM readability evaluation achieves moderate alignment with human judgments and outperforms the Scalabrino model in certain aspects [51]. However, systematic bias and significant variability remain. The authors further demonstrate that prompt design has a substantial impact on performance, improving both alignment and explanation quality. Nevertheless, a clear trade-off exists between accuracy and stability, as improvements in alignment are accompanied by increased variability in the results.

2.5 Code Readability from a Cognitive Perspective

In contrast to machine-learning-based readability models that place code features at the center of analysis, another line of work emphasizes a human-centered perspective. These approaches seek to explain code readability by analyzing and modeling human cognition, interpreting readability in terms of cognitive load, in order to identify more stable and transferable underlying mechanisms.

Experimental studies based on electroencephalography (EEG) and eye-tracking have shown that, as code complexity increases, features related to Theta, Alpha, and Beta brain waves exhibit the highest discriminative power [41]. Such increases in complexity are also associated with pronounced pupil dilation and frequent back-and-forth eye movements during code reading [2].

Siegmund et al. argue that program comprehension is a cognitive process that is difficult to measure directly. In 2014, they conducted an fMRI study to observe brain activation patterns during natural code reading [64]. The results show that the activated regions are primarily associated with working memory, attention, and language processing.

Norman Peitek et al.'s fMRI study in 2021 found that brain activation during code reading correlates with comprehension accuracy and, to a lesser extent, with response time [55]. Overall, neural activation patterns show stronger correlations with subjective complexity ratings than with objective metrics such as Halstead complexity and McCabe cyclomatic complexity. These subjective ratings also correlate with behavioral performance, suggesting that developers' perceived difficulty often reflects their actual comprehension.

These studies indicate that code comprehension involves stable and observable cognitive load, which is consistently reflected in human readability judgments. However, existing structural complexity metrics fail to adequately model such cognitive load, and current machine-learning-based readability models

likewise lack an explicit representation of the cognitive demands experienced by humans during code understanding. Although some previously used features, such as dictionary matching or abbreviation detection, implicitly proxy certain aspects of human cognitive load, there is currently no general code readability model that is explicitly grounded in human cognitive load.

2.6 Working Memory Mechanisms

Building on prior research in cognitive science, we adopt working memory as a simulatable cognitive proxy for modeling the cognitive load involved in code reading. Previous studies have explored code readability from a working-memory perspective; however, these approaches often overlook fundamental properties of working memory. In particular, semantically similar information tends to occupy less working memory capacity, and experts can rely on long-term knowledge in familiar domains without incurring substantial working memory load.

Working memory refers to the system or systems that are assumed to be necessary in order to keep things in mind while performing complex tasks such as reasoning, comprehension and learning [5]. Early memory models dichotomized memory into long-term memory and short-term memory; however, subsequent research has characterized working memory as a functional system that is task-oriented in nature [6]. Within this framework, short-term memory constitutes the storage component of working memory, whereas working memory as a whole supports higher-level processes such as comprehension, reasoning, and decision-making. Accordingly, we adopt the notion of working memory to model code readability as a process of understanding.

Gilboy and Thapen, in their blog, refer to the concept of working memory, and point out that human working memory has clear capacity limits [27, 68]. They propose a working memory algorithm based on variables, methods, and their scopes, which estimates the number of elements that need to be remembered to correctly understand the current line of code. This approach relies entirely on counting and does not explicitly model the internal mechanisms of working memory.

The basic units of information maintained in working memory are commonly referred to as chunks. Two key mechanisms of working memory are chunking and long-tern working memory. Chunking refers to the process by which multiple similar chunks are grouped into a single chunk, rather than being maintained separately, thereby reducing the demand on working memory capacity [67]. This process can be viewed as a form of cognitive recoding, in which related, regular, or similar information is abstracted into a higher-level representation [67, 33].

Long-term working memory describes the ability of domain experts to retrieve familiar patterns or templates from long-term memory with little or no working memory cost [25, 28]. For example, when developers encounter familiar algorithms or code structures, they do not incur the same cognitive load as when processing unfamiliar ones.

3 CognaScore

CognaScore is a cognitive model designed to estimate code readability by approximating the working memory demands imposed during code comprehension. Grounded in established findings from cognitive psychology, the model draws on principles of chunking and abstraction to characterize how semantic information is organized and maintained during code reading. The remainder of this section introduces the core terminology, presents an overview of the CognaScore procedure, and elaborates on the mechanisms for lexeme extraction and semantic abstraction.

3.1 Terminology

To model working memory in a computational setting, we introduce the following terminology to describe the representational units and memory structures used in our formulation.

A *lexeme* is a lexical element of source code that conveys semantic information and is likely to be mentally maintained during code comprehension, such as an identifier, a function name, or other meaningful tokens.

A *vector embedding* is a continuous numerical representation associated with a lexeme, derived from a pretrained embedding model, which encodes its semantic content and enables quantitative comparison of semantic relatedness.

A *chunk* is defined as the basic unit of working memory. It consists of a lexeme together with its corresponding vector embedding, representing both its symbolic form and semantic content.

Algorithm 1: CognaScore: Cognitive load estimation via semantic chunk abstraction.

Input: Source code P
Output: Readability score $\hat{y} \in [1, 5]$
$C \leftarrow \mathit{extractChunks}(P)$;
$E \leftarrow \mathit{embed}(C)$;
$\mathcal{B} \leftarrow \mathit{DBSCAN}(E; \text{minPts}{=}2)$; `// initial clusters`
$K \leftarrow |\{b \in \mathcal{B}\}|$;
$(B, R, J) \leftarrow \mathit{countTypes}(C)$; `// BITWISE, REGEX, JUNK counts`
$\hat{y} \leftarrow 5.797698 - 0.068358\,K - 0.068358\,B - 0.068358\,R - 0.176813\,J$
return $\hat{y}$

A *bundle* is a finite multiset of chunks. In cognitive psychology, chunking is understood as an abstraction process in which information is compressed, while the resulting representation is still considered a single chunk[67, 33]. To avoid ambiguity arising from this reuse of terminology, we use the term bundle to denote the outcome of chunking in our computational model.

Working memory is modeled as a temporary and manipulable storage of information during code comprehension. In our formulation, it is represented as a set of bundles. While no fixed capacity limit is imposed, larger working memory representations correspond to higher cognitive load.

3.2 CognaScore Model

CognaScore aims to estimate the cognitive load involved in code comprehension by modeling the demands placed on working memory. The core idea is that code readability decreases when developers must simultaneously maintain a large number of unrelated chunks, whereas consistent and semantically meaningful naming allows related chunks to be abstracted into a single bundle, thereby reducing cognitive load.

Given a source program, CognaScore operates in four main steps.

1. First, lexical and structural elements that are likely to occupy working memory are extracted from the code.
2. Second, these elements are mapped into a semantic space using vector embeddings.
3. Third, semantically similar elements are grouped through abstraction, forming bundles that approximate the effects of chunking in working memory.
4. Finally, the resulting bundle structure is used to derive an estimate of working memory load.

Algorithm 1 presents a concrete instantiation of the proposed approach. In this implementation, lexemes are first extracted from the source code and grouped into bundles via DBSCAN clustering in embedding space. Chunks associated with constructs that are known to be difficult for humans to comprehend, such as bitwise operations, regular expressions, and semantically uninformative variable names, are explicitly taken into account. The resulting measures are finally combined through a linear regression model to produce a readability score on a five-point scale.

3.3 Lexeme Extraction

Based on prior research[20, 59], we consider the following code constructs to constitute the minimal cognitive units involved in human code comprehension: identifiers, literals, expressions, function calls, control structures, comments, and import statements. For each construct type, we define a lexeme as the combination of its type and name, so as to capture both its syntactic role and semantic identity.

For variables, an additional prefix def is introduced for their first occurrence to distinguish definition from subsequent uses. For functions, the lexeme includes the function name together with its formal parameters or actual arguments, reflecting the semantic role of parameter passing in comprehension. Control structures are treated as built-in control functions and are handled uniformly within the same lexeme framework.

For constructs that may be nested, such as expressions and function calls, lexemes are extracted recursively to account for their hierarchical structure. We further distinguish between used and unused

Source code	Lexeme(s)
`import java.util.List;`	`import_stdlib_util_List`
`import com.foo.Bar;`	`import_external_foo_Bar`
`import com.foo.Bar;` (*unused*)	`unused_import_external_foo_Bar`
`42`	`literal_42`
`"hello"`	`literal_hello`
`int x = y;`	`x, int_x, def_x, def_x_y`
`int x = 42;`	`x, int_x, def_x, def_x_42, literal_42`
`"[a-z]+"`	`regex_[a-z], regex_+`
`a + b`	`a+b`
`foo(a, b)`	`foo_a_b`
`obj.foo(a)`	`obj.foo_a`
`void foo(int p, int q) {...}`	`def_foo_p_q, foo, int_p, int_q, p, q`
`if (cond) {...}`	`if_cond, L(cond)`
`if (c1) {...} else if (c2) {...}`	`if_c1, elif_c2, L(c1), L(c2)`
`if (c1) {...} else if (c2) {...} else {...}`	`if_c1, elif_c2, L(c1), L(c2), else_not_c1_not_c2`
`while (cond) {...}`	`while_cond`
`do {...} while (cond);`	`do-while_cond`
`for (i = 0; i < n; i++) {...}`	`for_n` (*trivial start/step*)
`for (int i = 2; i < n; i += 3) {...}`	`for_n_i=2_i+=3` (*non-trivial start/step*)
`for (T x : xs) {...}`	`for_x_xs`
`switch (expr) {...}`	`switch_expr`
`x += y`	`x+=y`

Table 1: Illustrative examples of lexeme construction under the proposed cognitive modeling approach. Each source code fragment is mapped to the set of lexemes assumed to occupy working memory during code comprehension.

import statements, as unused imports are assumed to impose a lower cognitive load during code comprehension. Trivial for loops, whose initialization starts at zero and whose step size is one, are not fully encoded, reflecting their limited cognitive impact.

Regular expression literals are further decomposed into smaller lexemes to capture their internal minimal cognitive units. In addition, lexemes consisting of a single alphabetic character are duplicated four times in the working memory representation, to compensate for the strong frequency bias associated with single-letter tokens, which would otherwise dominate semantic similarity over meaningful distinctions. The complete set of extraction rules is summarized in Table 1.

3.4 Abstraction via Embedding Similarity

Chunking is a widely accepted mechanism in cognitive psychology, whereby semantically similar information does not occupy independent slots in working memory[67]. This principle also applies to code comprehension. During code reading, developers continuously organize and abstract information until a coherent understanding of the program is formed. Intuitively, consistent and meaningful naming and structure facilitate abstraction, resulting in a smaller working memory footprint and, consequently, higher code readability. In contrast, inconsistent or semantically meaningless naming is difficult to abstract, leading to a larger working memory footprint and reduced readability.

Defining semantic similarity explicitly is challenging, as rules based solely on string matching or handcrafted lexical heuristics are insufficient to capture the full range of semantic relatedness. Vector embeddings provide a practical means of measuring semantic similarity by mapping lexemes into a continuous semantic space[14]. Under this representation, the distance between the embeddings of two lexemes serves as a proxy for their semantic similarity.

Within this framework, semantically related chunks are naturally grouped into bundles, each of which represents an abstracted unit in working memory. A smaller number of bundles corresponds to lower cognitive load, which in turn indicates higher code readability.

In the computational realization of this abstraction process, semantic similarity between chunks is measured using cosine distance in embedding space, which captures the relative orientation of semantic representations independent of their magnitude. We employ a density-based clustering strategy with a minimal cluster size of two, reflecting the cognitive assumption that chunking does not require a fixed lower bound on group size and can occur as soon as two semantically related elements are identified. Under this setting, chunks that are sufficiently similar are grouped into the same bundle, approximating the abstraction effects observed in human working memory.

4 PreFixScore

While CognaScore models code readability from a bottom-up perspective grounded in working memory constraints, it primarily captures the local cognitive load incurred during incremental code comprehension. However, human understanding of code is not solely driven by such step-by-step processing. In many cases, developers are able to rapidly infer the overall intent and structure of a program from partial observations, relying on previously acquired knowledge and familiar patterns.

Motivated by this observation, we propose PreFixScore, a complementary readability model that adopts a top-down perspective. Instead of modeling local cognitive load, PreFixScore estimates readability based on the extent to which the full program can be semantically recovered from its partial prefix. Intuitively, code that exposes its intent early is easier to understand, whereas code that requires extensive exposure before its behavior becomes clear imposes a higher cognitive burden.

4.1 Cognitive Motivation

Human code comprehension does not always require a complete, line-by-line analysis of all program details. In many cases, developers are able to infer the intent and implementation of a program from partial observations, relying on knowledge stored in long-term memory. Once a sufficient match is established between the observed code and previously encountered patterns, the remaining parts of the program can often be processed with reduced cognitive effort[28].

During this process, developers form hypotheses about the program's behavior based on partial observations, and subsequently validate these hypotheses as more code is read. When the predicted structure aligns with the remaining code, comprehension proceeds rapidly. Otherwise, the initial hypothesis is revised, and further reading is required until a coherent understanding is achieved.

This iterative process can be interpreted as a form of predictive processing, in which comprehension proceeds in a hierarchical manner. High-level expectations generate predictions about lower-level details, while incoming information is compared against these predictions. Mismatches lead to updates of the higher-level representation, and the process repeats until a coherent interpretation is achieved [16].

4.2 Prefix-based Recovery Procedure

Motivated by the predictive process described above, we define a procedure that evaluates how readily a program can be reconstructed from partial observations. The method operationalizes the cycle of prediction and validation by progressively revealing a program, generating candidate completions, and assessing their consistency with the original implementation (Algorithm 2).

Given a program P, we first define its effective logical code as the sequence of non-whitespace characters. The program is then partitioned into $K-1$ exposure stages, corresponding to revealing approximately $\frac{1}{K}, \frac{2}{K}, \ldots, \frac{K-1}{K}$ of its effective logical code. The partitioning is performed directly on the code sequence without aligning to syntactic or comment boundaries.

At each stage, the visible prefix is provided to a completion model (LLM A), which is tasked with reconstructing the full program. The generated completion is then compared against the original program using a second model (LLM B), which evaluates their semantic consistency.

To account for the stochastic nature of LLM outputs, the above process is repeated three times independently at each stage. If all three runs produce valid completions, the program is assigned the score corresponding to that stage. If two out of three runs succeed, a small penalty (0.5) is applied to the stage score to break ties between fully and partially consistent recoveries. Otherwise, the procedure proceeds to the next stage.

The stage score is defined as a function of the exposure ratio s, given by $score(s) = K \cdot (1 - s) + 1$, ensuring that earlier successful recovery leads to a higher readability score. If no stage satisfies the acceptance criteria, the final score is set to 1.

Algorithm 2: PreFixScore: Readability via prefix-based semantic recovery.

Input: Source code P, number of stages K
Output: Readability score $\hat{y} \in [1, K]$
$\mathcal{S} \leftarrow \left\{ \frac{1}{K}, \frac{2}{K}, \ldots, \frac{K-1}{K} \right\}$; // exposure stages
for $s \in \mathcal{S}$ **do**
  $P_s \leftarrow partial(P, s)$
  $r \leftarrow Evaluate(P, P_s)$; // multi-run recovery and consistency check
  if r *is successful* **then**
    return $\hat{y} \leftarrow score(s, r)$
return $\hat{y} \leftarrow 1$

```
You are assigned the role of a software developer.  Your task is to produce the complete
program, including the given program prefix, by following the requirements specified in the
comments.  Below is the program you need to complete:
+ "prompt"
```

Figure 1: Standard prompt for code completion.

5 Methodology

We evaluate the proposed readability models under a unified experimental framework that combines controlled analysis and empirical evaluation. This section introduces the datasets used in the study, describes the configuration of the underlying models, and presents the evaluation protocols applied under different labeling schemes.

5.1 Datasets

We employ three datasets in this study: a controlled synthetic dataset for model analysis, and two external datasets for empirical evaluation, namely the JetBrains readability dataset and the Schnappinger et al. maintainability dataset [62, 60].

This study constructs a small-scale, controlled, synthetic dataset for model debugging and behavioral analysis. The dataset is based on MBJP programming tasks and consists of method-level Java code generated by a large language model under explicitly specified constraints. During generation, we separately enforce high-readability and low-readability coding instructions, ensuring functional equivalence across solutions while introducing systematic differences solely in code expression.

The dataset contains 17 representative tasks and is primarily used to examine the reliability of the proposed readability model, as well as to identify early-stage deficiencies and modeling omissions. It is important to emphasize that this dataset is not used for final performance evaluation. Instead, it serves as a controlled experimental setting to support qualitative and quantitative analysis of the model's underlying mechanisms and assumptions.

Specifically, we use the GPT-4o model to generate method-level Java implementations for 17 tasks selected from the MBJP dataset, under two distinct prompting conditions: a standard prompt (Figure 1) and a prompt that explicitly encourages low readability (Figure 2). To promote diversity in the generated code, we do not employ few-shot examples. Instead, we rely on the test suites provided by MBJP to ensure functional correctness, retaining only solutions that pass all tests under both prompting conditions.

We subsequently assessed the readability of the generated code using a simple web-based interface, employing a five-point Likert scale. Given the high level of inter-rater agreement, which falls into the almost perfect agreement range according to the quadratic weighted Cohen's kappa (0.81), we use the mean of the assigned scores as the readability label for this small-scale, controlled synthetic dataset in subsequent experiments. Detailed statistics are reported in Appendix A.

5.2 Model Configuration

The configuration of underlying models plays a central role in both CognaScore and PreFixScore, as their behavior depends on the quality of semantic representations and generation capabilities. To ensure

```
 You are assigned the role of a software developer.  Your task is to produce the complete
program, including the given program prefix, by following the requirements specified in
the comments.  While ensuring functional correctness, you should deliberately reduce the
readability of the code.  Below is the program you need to complete:
+ "prompt"
```

Figure 2: Low-readability prompt for code completion.

Model (LLM A + LLM B)	K=3	K=4	K=5	K=7	K=10
GPT-4.1-nano + GPT-4.1-nano	0.507	0.208	0.347	0.283	0.085
Qwen3-32B + GPT-5-nano	0.537	0.576	0.512	0.445	0.398
GPT-4.1 + GPT-5-nano	0.445	0.465	0.447	0.254	0.321

Table 2: Effect of the number of exposure stages K on Spearman correlation under a fixed LLM configuration.

that the proposed methods are not sensitive to arbitrary design choices, we examine these configurations through controlled experiments and justify the final selections based on empirical evidence.

5.2.1 Embedding Model for CognaScore

CognaScore relies on vector embeddings to capture semantic relatedness between lexemes during abstraction. As abstraction is performed through clustering in embedding space, the validity of the model depends on whether the embedding representation preserves meaningful semantic structure, particularly for short identifiers and phrases.

To evaluate this assumption, we conduct a set of controlled experiments that analyze embedding behavior with respect to semantic relatedness and sensitivity to word frequency under conditions relevant to the abstraction mechanism in CognaScore. Specifically, we examine whether cosine distance consistently reflects semantic relationships across different lexical settings.

We evaluate three representative embedding models [22, 69, 49]. For each model, we analyze cosine distances for synonym–antonym pairs and randomly sampled word pairs, and further verify that cosine distance exhibits gradual decay as word pairs become more distant in the synonym–antonym network. Detailed experimental procedures and additional analyses are reported in Appendix B.

Experimental results indicate that cosine distance is an effective metric when applied to static word embeddings produced by more recent, contrastively trained models. It consistently distinguishes semantically related word pairs from randomly paired words and exhibits gradual semantic degradation along lexical relation chains. In contrast, the `bert-base-uncased` model [22] exhibits a strong bias toward word frequency, which diminishes the effectiveness of cosine distance in capturing semantic relationships, consistent with prior findings [35].

5.2.2 LLM Configuration for PreFixScore

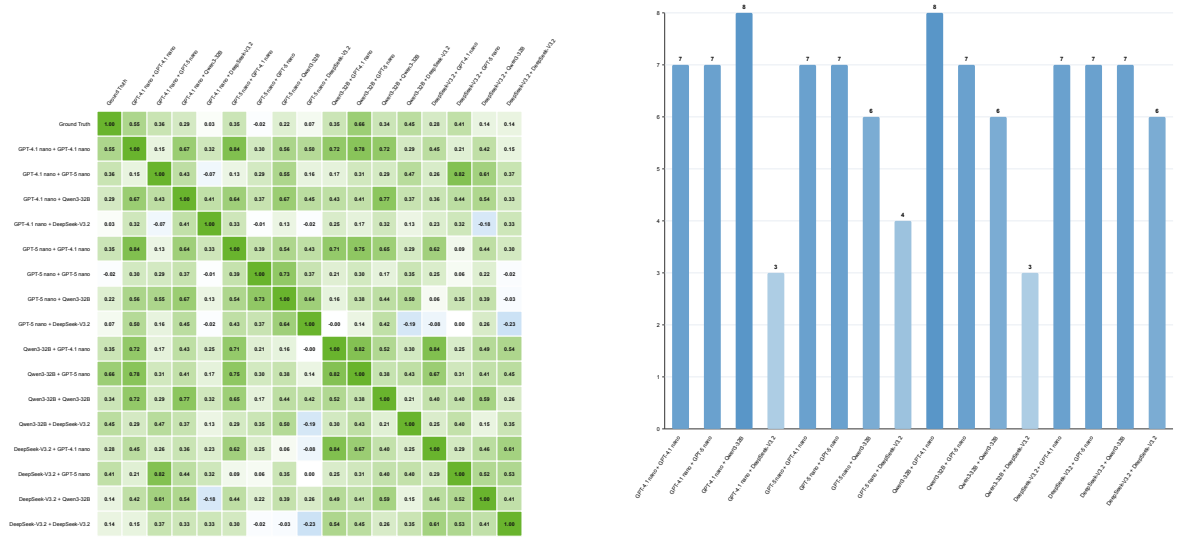
PreFixScore depends on several hyperparameters, including the number of exposure stages K and the choice of underlying LLMs. To guide the selection of these parameters, we conduct controlled experiments on the MBJP dataset.

We first study the effect of the number of exposure stages. Specifically, we evaluate $K \in \{3, 4, 5, 7, 10\}$, and measure the alignment with ground-truth readability using Spearman's rank correlation coefficient. The results are shown in Table 2.

The results show a consistent degradation in performance for larger values of K, indicating that overly fine-grained partitioning introduces instability in the evaluation process. At the same time, very small values of K provide limited resolution, as many samples collapse into a small number of distinct scores.

Based on these observations, we adopt $K = 5$ as a representative configuration, as it provides a balanced trade-off between resolution and stability.

In addition, we observe that some model configurations produce insufficiently diverse outputs, which affects the discriminative power of the resulting scores. This aspect will be further examined in subsequent



(a) Correlation matrix

(b) Prediction diversity

Figure 3: PreFixScore behavior under different LLM configurations. Left: correlation with ground truth. Right: diversity of predicted scores.

experiments.

PreFixScore also relies on large language models to simulate the predictive process in code comprehension. Specifically, an LLM is used in two roles: (1) program recovery from partial observations (LLM A), and (2) semantic consistency evaluation between the recovered and original programs (LLM B).

Prior work has shown that prompt design has a substantial impact on LLM performance, particularly for tasks involving code understanding and semantic comparison [51]. Following this line of work, we adopt a structured prompting strategy that explicitly requires the model to assess consistency along three dimensions: (1) intent consistency, (2) implementation consistency, and (3) absence of side effects or redundant code. The full prompt templates are provided in Appendix C.

To evaluate the sensitivity of PreFixScore to model choice, we conduct a controlled study over multiple LLM configurations. We consider four representative models (DeepSeek-V3.2, GPT-4.1-nano, GPT-5-nano, and Qwen3-32B) and evaluate all pairwise combinations for LLM A and LLM B. For each configuration, we measure the correlation between the resulting PreFixScore and the ground-truth readability scores.

The results indicate that model selection has a noticeable impact on performance. While most configurations exhibit moderate positive correlation with the ground truth, certain combinations yield significantly weaker alignment. Based on these observations, we use GPT-4.1-nano for both LLM A and LLM B in all subsequent experiments. This configuration provides a strong and stable alignment with ground-truth readability while maintaining consistency across runs. Detailed results of the configuration study are shown in Figure 3.

5.3 Evaluation

We evaluate CognaScore, the LLM-based readability approach, and PreFixScore on two complementary datasets: the JetBrains code readability dataset [62] and the Schnappinger et al. maintainability dataset [60].

The JetBrains dataset provides binary readability labels, and prior work has reported results of various machine-learning-based readability models on this dataset under a unified evaluation protocol. As it does not provide continuous readability scores, we follow prior work and report classification performance using the Matthews Correlation Coefficient (MCC) [15], which is well-suited for imbalanced binary classification.

To complement this evaluation, we additionally consider the Schnappinger dataset, which provides probabilistic annotations over a four-level Likert scale. This enables a more fine-grained evaluation

Readability Model	MCC
Posnett et al.	-0.036
Dorn	-0.038
Mi et al.	0.033
Scalabrino et al.	0.325
GPT5.1-nano	0.103
CognaScore	0.245
PreFixScore	0.221

Table 3: Alignment with Human Readability Judgments on the JetBrains Dataset

through rank-based correlation. We convert the probability distributions into continuous readability scores via expectation and scale inversion, such that higher values correspond to better readability. On this dataset, we report Spearman's rank correlation coefficient to measure alignment with human judgments [65].

All parameters of CognaScore were calibrated on the controlled synthetic dataset described in Section 5.1 and fixed prior to evaluation. Similarly, the readability threshold used in PreFixScore was determined on the same controlled dataset. No parameter tuning was performed on either evaluation dataset, ensuring that the reported results reflect generalization rather than dataset-specific optimization.

For each code snippet, CognaScore computes a readability score following the procedure described in Section 3. The LLM-based approach directly predicts readability scores via prompting, while PreFixScore produces stage-based scores based on semantic recoverability from partial observations (Section 4). All methods are evaluated under the corresponding protocols of each dataset.

For the JetBrains dataset, we follow prior work and report results for representative readability models, including Posnett et al., Dorn, Mi et al., and Scalabrino et al. In addition, we include a direct LLM-based readability assessment as an additional baseline.

For the Schnappinger dataset, we report results for Posnett et al. and Scalabrino et al., and further include simple structural indicators such as effective lines of code (LOC), as well as the same direct LLM-based scoring approach.

For the LLM-based baseline, we adopt one of the best-performing prompt strategies proposed by Ouédraogo et al. [51], specifically the zero-shot (ZSL) prompting scheme. The full prompt template is provided in Appendix D.

5.4 Research Questions

We investigate the following research questions:

RQ1 How well do CognaScore and PreFixScore align with human judgments of code readability?

RQ2 How do CognaScore and PreFixScore compare to existing readability models and direct LLM-based scoring?

RQ3 How do structural characteristics of code, such as length and functional granularity, influence the performance of CognaScore and PreFixScore?

RQ4 To what extent can prefix-based semantic recovery serve as a reliable proxy for code readability?

6 Results

The empirical evaluation shows a clear contrast between the two proposed approaches. Across both binary and fine-grained settings, CognaScore maintains stable alignment with human readability judgments. In contrast, PreFixScore provides useful signal on short snippets, but its effectiveness decreases substantially on larger, real-world code units.

The remainder of this section reports the detailed results on the JetBrains and Schnappinger datasets, followed by analyses of structural effects and failure modes.

Method	Spearman ρ	Pearson r
Posnett et al.	0.186	0.466
Scalabrino et al.	0.458	0.414
LOC	-0.618	-0.610
GPT5.1-nano	0.056	0.123
CognaScore	0.582	0.600
PreFixScore	-0.186	-0.236

Table 4: Correlation with Human Readability Scores on the Schnappinger Dataset

Variable	Pearson r	Spearman ρ
LOC	0.371	0.316
Number of functions	0.418	0.380

Table 5: Correlation between PreFixScore predictions and structural size measures on the Schnappinger dataset.

6.1 Results on the JetBrains Dataset

We evaluate both CognaScore and PreFixScore on the JetBrains code readability dataset [62], where readability is formulated as a binary classification problem. Following prior work, we adopt a threshold of 3 on the five-point Likert scale to distinguish between readable (≥ 3) and non-readable (< 3) code snippets.

All parameters of CognaScore were calibrated solely on the controlled synthetic dataset described in Section 5.1 and were fixed prior to evaluation. No parameter tuning was performed on either evaluation dataset.

Under this setting, CognaScore achieves a Matthews Correlation Coefficient (MCC) of 0.245, as shown in Table 3. This performance is second only to the model of Scalabrino et al. [59], and substantially outperforms earlier models such as Posnett et al. [56] and Dorn [24].

For PreFixScore, since it produces discrete stage-based scores rather than a linear readability scale, its outputs must be mapped to binary labels. The decision threshold is selected on a separate MBJP validation set and set to 2.0, such that samples with scores greater than or equal to 2.0 are classified as readable, and those below 2.0 as non-readable. Under this setting, PreFixScore achieves an MCC of 0.221, as reported in Table 3, outperforming the compared baseline methods but remaining below CognaScore.

A closer inspection of the error patterns shows that CognaScore correctly identifies most readable samples, but is less effective on unreadable ones, indicating a bias toward predicting code as readable. In contrast, PreFixScore produces more false negatives, suggesting that short prefixes often do not expose enough semantic structure for reliable recovery.

Length effects are limited on this dataset. Although CognaScore exhibits moderate negative correlation with LOC, the ground-truth labels themselves show only weak dependence on LOC, and the correlation between CognaScore and human judgments decreases only slightly after controlling for LOC. Similarly, PreFixScore shows only weak correlation with LOC, and its correlation with human judgments slightly increases after controlling for length. These results suggest that, on JetBrains, neither model derives its predictive signal primarily from code length.

6.2 Results on the Schnappinger Dataset

On the Schnappinger dataset, CognaScore achieves a Spearman correlation of 0.582 and a Pearson correlation of 0.600, as shown in Table 4. Under the same evaluation setting, it outperforms all compared readability baselines, including the model of Scalabrino et al., with relative improvements of approximately 27% in Spearman and 45% in Pearson correlation.

However, CognaScore remains slightly below simple structural indicators such as effective lines of code (LOC), which achieve even higher correlation with human judgments (Table 4). This indicates that structural properties provide strong predictive signals in this dataset. At the same time, after controlling for LOC, CognaScore still retains positive partial correlation with human judgments, as

Method	Partial Spearman ρ	Partial Pearson r
CognaScore	0.178	0.155
PreFixScore	-0.008	-0.012

Table 6: Partial correlation with human readability scores after controlling for effective lines of code (LOC).

shown in Table 6, suggesting that it captures readability signals beyond simple size effects.

In contrast, PreFixScore achieves a Spearman correlation of -0.186 and a Pearson correlation of -0.236, as reported in Table 4, indicating that it does not provide meaningful alignment with human judgments in this setting. A key difference between the two datasets lies in structural granularity: whereas JetBrains consists of short snippets, the Schnappinger dataset contains substantially larger class-level code units.

Further analysis shows that PreFixScore is positively correlated with both effective lines of code and the number of functions, indicating a tendency to assign higher scores to longer and more function-heavy files, as shown in Table 5. This behavior is particularly problematic for real-world class-level code containing repetitive or template-like structure. For example, generated DOM binding files may be highly regular and easy for the recovery model to complete from an early prefix, while still being judged by humans as difficult to read.

After controlling for LOC, PreFixScore exhibits near-zero partial correlation with human judgments (Table 6). This suggests that its predictions on this dataset are largely explained by structural size rather than independent readability signals. In addition, the comparison prompts used in PreFixScore were designed primarily in a single-function setting, and appear overly permissive when applied to large multi-function programs.

6.3 Failure Analysis

CognaScore. The main failure mode of CognaScore is length calibration. On long files, the model tends to assign overly low raw scores, leading to systematic underestimation. This behavior follows directly from the model design: as code length increases, the number of distinct lexemes also increases, which raises the estimated working-memory load. On shorter snippets, the opposite problem may occur, with short but semantically or structurally poor code receiving scores that are too high.

PreFixScore. The dominant failure mode of PreFixScore depends on code granularity. On short snippets, it tends to produce false negatives because the visible prefix often does not provide enough information for reliable recovery. On large multi-function files, the opposite problem occurs: the model tends to overestimate code that is structurally regular or easy to continue, even when it is not perceived as readable by humans. This limitation is reinforced by the current comparison prompts, which were tuned in a single-function setting and become overly permissive on larger file-level programs.

6.4 Answering the Research Questions

RQ1. CognaScore aligns well with human readability judgments across both evaluation settings. PreFixScore shows useful signal on short snippets, but does not generalize reliably to larger real-world code units.

RQ2. CognaScore is competitive with existing readability models and clearly stronger than direct LLM-based scoring. PreFixScore outperforms several baselines on JetBrains, but is not competitive overall.

RQ3. Structural characteristics such as code length and functional granularity substantially affect both models. CognaScore is influenced by length but still retains independent predictive signal, whereas PreFixScore becomes increasingly biased as structural scale grows.

RQ4. Prefix-based semantic recovery can provide a useful proxy for readability in constrained, function-level settings, but does not remain reliable on large, multi-function programs.

7 Discussion

Like other simple readability metrics, CognaScore can be misled by certain code patterns. In particular, a program may exhibit strong lexical consistency and semantic clustering, while still being logically unclear or poorly structured.

For example, consider the following fragment adapted from a standard prefix-sum computation:

```
int temp1 = 0;
for (int i = 0; i < n; i++) {
    temp1 = temp1 + arr[i];
    temp2[i] = temp1;
}
```

Although the code follows a simple and regular structure, the use of non-descriptive variables such as `temp1` and `temp2` obscures its intent. In contrast, a more readable version makes the underlying computation explicit:

```
int prefixSum = 0;
for (int i = 0; i < n; i++) {
    prefixSum += arr[i];
    result[i] = prefixSum;
}
```

Even when the control flow and operations are identical, the semantic role of each variable is clearly conveyed in the latter version. However, the model may still assign a relatively high score to the former.

This limitation stems in part from the behavior of current embedding models. In particular, lexical similarity in the embedding space does not always reflect how developers interpret variable names. For example, identifiers such as `a`, `b`, and `c` often exhibit high cosine similarity, despite carrying little semantic meaning in most non-mathematical contexts. As a result, code with weak or non-descriptive naming may still appear structurally consistent from the model's perspective.

In addition, although control structures are incorporated into CognaScore, prior studies have shown that different types of control flow and deep nesting can significantly increase comprehension difficulty. Such effects are difficult to capture solely through low-level working-memory abstractions, which primarily operate at the level of lexemes. This limitation also motivates the introduction of PreFixScore, which provides a complementary perspective based on global semantic recoverability.

Ideally, CognaScore and PreFixScore should be unified into a single framework. Intuitively, code segments that can be reliably predicted from context may not require explicit working-memory modeling, while less predictable regions could be evaluated through cumulative cognitive load. Exploring such a hybrid approach remains an important direction for future work.

A key limitation of PreFixScore lies in the interaction between prefix segmentation and the use of LLM-based consistency evaluation. The current design relies on a fixed number of prefix stages together with an LLM-as-judge mechanism. While this combination performs well on the MBJP dataset used during development, it becomes less stable when applied to longer programs, where the evaluation results exhibit higher variance and reduced reliability.

One contributing factor is that the prefix segmentation scheme was tuned on function-level data with relatively uniform length. When applied to file-level programs with substantially larger variation in size, a fixed number of segments may fail to provide meaningful exposure granularity. This suggests that the segmentation process should be adapted dynamically based on program length or structural complexity.

In addition, our error analysis indicates that the LLM-as-judge mechanism is not consistently reliable in this setting, particularly for long and structurally complex code. A possible direction for improvement is to replace or complement the LLM-based comparison with more stable similarity measures, such as static or embedding-based semantic similarity, which may provide more consistent behavior across different scales.

8 Conclusion

In this work, we propose two cognitively motivated metrics for code readability. CognaScore models readability from the perspective of short-term working memory, capturing the combined effect of lexeme diversity and abstraction. PreFixScore, in contrast, measures readability based on prefix-based semantic recoverability, using predictability as a proxy for long-term memory and prior knowledge.

We evaluate both methods on two complementary datasets: the JetBrains dataset at the function level and the Schnappinger dataset at the file level. CognaScore demonstrates stable alignment with human judgments across different evaluation settings, outperforming traditional readability models in fine-grained evaluation. Importantly, it retains meaningful predictive power even after controlling for code length, indicating that it captures cognitive signals beyond simple structural factors.

In contrast, PreFixScore performs well on function-level code, but fails to generalize to file-level programs, where it exhibits weak or even negative correlation with human judgments. This suggests that simple prefix-based methods relying on a fixed number of segmentation stages are not sufficiently robust, and that adapting the exposure process to program length or structure may be necessary.

Overall, the results indicate that neither abstraction-based modeling of short-term memory nor prefix-based semantic predictability alone is sufficient to capture code readability. Readability inherently depends on both local cognitive load and global semantic structure, and should be understood as a multi-factor property.

A promising direction for future work is to unify these two perspectives into a single framework. Intuitively, code regions that can be reliably predicted from context may not require explicit modeling of cognitive load, while less predictable parts could be evaluated through cumulative working-memory cost. Such a hybrid approach may better reflect the actual process of human code comprehension.

References

[1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[2] Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In *Proceedings of the 30th IEEE/ACM international conference on program comprehension*, pages 111–121, 2022.

[3] Rafa E Al Qutaish and Alain Abran. An analysis of the design and definitions of halstead's metrics. In *15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag*, pages 337–352, 2005.

[4] Duaa Alawad, Manisha Panta, Minhaz Zibran, and Md Rakibul Islam. An empirical study of the relationships between code readability and software complexity. *arXiv preprint arXiv:1909.01760*, 2019.

[5] Alan Baddeley. Working memory. *Current biology*, 20(4):R136–R140, 2010.

[6] Alan David Baddeley. Working memory. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 302(1110):311–324, 1983.

[7] Jasdeep Singh Bhalla and Mansimar Kaur Jodhka. Leveraging large language models in the software development lifecycle: Opportunities and challenges. *International Journal of Advanced Computer Science & Applications*, 16(11), 2025.

[8] Neil CC Brown, Marcus Messer, and Jennifer Ikin. Failures in reliably assessing program code readability. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research*, pages 1–7, 2025.

[9] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

[10] Simon Butler. The effect of identifier naming on source code readability and quality. In *Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium*, pages 33–34, 2009.

[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[12] A Campbell. Cognitive complexity: A new way of measuring understandability. sonarsource (2016).

[13] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

[14] Dhivya Chandrasekaran and Vijay Mago. Evolution of semantic similarity—a survey. *Acm Computing Surveys (Csur)*, 54(2):1–37, 2021.

[15] Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21(1):6, 2020.

[16] Andy Clark. Whatever next? predictive brains, situated agents, and the future of cognitive science. *Behavioral and brain sciences*, 36(3):181–204, 2013.

[17] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[18] Scott Crossley, Aron Heintz, Joon Suh Choi, Jordan Batchelor, Mehrnoush Karimi, and Agnes Malatinszky. A large-scaled corpus for assessing text readability. *Behavior Research Methods*, 55(2):491–507, 2023.

[19] Edgar Dale and Jeanne S Chall. The concept of readability. *Elementary English*, 26(1):19–26, 1949.

[20] Carlos Eduardo C Dantas, Adriano M Rocha, and Marcelo A Maia. How do developers improve code readability? an empirical study of pull requests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 110–122. IEEE, 2023.

[21] Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi. Which factors affect software projects maintenance cost more? *Acta Informatica Medica*, 21(1):63, 2013.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[23] Marc Dillmann, Julien Siebert, and Adam Trendowicz. Evaluation of large language models for assessing code maintainability. *arXiv preprint arXiv:2401.12714*, 2024.

[24] Jonathan Dorn. A general software readability model. *MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf)*, 5:11–14, 2012.

[25] K Anders Ericsson and Walter Kintsch. Long-term working memory. *Psychological review*, 102(2):211, 1995.

[26] Matteo Esposito, Andrea Janes, Terhi Kilamo, and Valentina Lenarduzzi. Early career developers' perceptions of code understandability. a study of complexity metrics. *IEEE Access*, 2025.

[27] Tim Gilboy. How much information is too much information?, February 2022. Accessed: 2025-03-23.

[28] Fernand Gobet. Expert memory: A comparison of four theories. *Cognition*, 66(2):115–152, 1998.

[29] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[30] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):1–28, 2013.

[31] Jeremy Howells. Tacit knowledge. *Technology analysis & strategic management*, 8(2):91–106, 1996.

[32] Rohit Khankhoje. Quality challenges and imperatives in smart ai software. *Computer Science & Information Technology (CS & IT)*, pages 143–152, 2023.

[33] Benjamin Kowialiewski, Benoît Lemaire, and Sophie Portrat. Similarity-based compression in working memory: Implications for decay and refreshing models. *Computational Brain & Behavior*, 7(1):163–180, 2024.

[34] Luigi Lavazza, Sandro Morasca, and Marco Gatto. An empirical study on software understandability and its dependence on code characteristics. *Empirical Software Engineering*, 28(6):155, 2023.

[35] Bohan Li, Hao Zhou, Junxian He, Mingxuan Wang, Yiming Yang, and Lei Li. On the sentence embeddings from pre-trained language models. *arXiv preprint arXiv:2011.05864*, 2020.

[36] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26, 2024.

[37] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024.

[38] Md Abdullah Al Mamun, Christian Berger, and Jörgen Hansson. Effects of measurements on correlations of software code metrics. *Empirical Software Engineering*, 24(4):2764–2818, 2019.

[39] Umme Ayda Mannan, Iftekhar Ahmed, and Anita Sarma. Towards understanding code readability and its impact on design quality. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, pages 18–21, 2018.

[40] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[41] Júlio Medeiros, Ricardo Couceiro, Gonçalo Duarte, João Durães, João Castelhano, Catarina Duarte, Miguel Castelo-Branco, Henrique Madeira, Paulo De Carvalho, and César Teixeira. Can eeg be adopted as a neuroscience reference for assessing software programmers' cognitive load? *Sensors*, 21(7):2338, 2021.

[42] Qing Mi, Yiqun Hao, Liwei Ou, and Wei Ma. Towards using visual, semantic and structural features to improve code readability classification. *Journal of Systems and Software*, 193:111454, 2022.

[43] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104:60–71, 2018.

[44] Pu Miao, Zeyao Du, and Junlin Zhang. Debcse: rethinking unsupervised contrastive sentence embedding learning in the debiasing perspective. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 1847–1856, 2023.

[45] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[46] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd international conference on program comprehension*, pages 25–35. IEEE, 2015.

[47] Sergio Moreschini, Elvira-Maria Arvanitou, Elisavet-Persefoni Kanidou, Nikolaos Nikolaidis, Ruoyu Su, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Valentina Lenarduzzi. The evolution of technical debt from devops to generative ai: A multivocal literature review. *Journal of Systems and Software*, 231:112599, 2026.

[48] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 1–12, 2020.

[49] Zach Nussbaum, John X Morris, Brandon Duderstadt, and Andriy Mulyar. Nomic embed: Training a reproducible long context text embedder. *arXiv preprint arXiv:2402.01613*, 2024.

[50] Pawan Kumar Ojha, Abid Ismail, and Kuppusamy Kundumani Srinivasan. Perusal of readability with focus on web content understandability. *Journal of King Saud University-Computer and Information Sciences*, 33(1):1–10, 2021.

[51] Wendkûuni C Ouédraogo, Yinghua Li, Xueqi Dang, Pawel Borsukiewicz, Xin Zhou, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. Human-aligned code readability assessment with large language models. *arXiv preprint arXiv:2510.16579*, 2025.

[52] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[53] Kang-il Park, Jack Johnson, Cole S Peterson, Nishitha Yedla, Isaac Baysinger, Jairo Aponte, and Bonita Sharif. An eye tracking study assessing source code readability rules for program comprehension. *Empirical Software Engineering*, 29(6):160, 2024.

[54] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Benchmarks and metrics for evaluations of code generation: A critical review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 87–94. IEEE, 2024.

[55] 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)*, pages 524–536. IEEE, 2021.

[56] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82, 2011.

[57] Alfred Santa Molison, Marcia Moraes, Glaucia Melo, Fabio Santos, and Wesley KG Assuncao. Is llm-generated code more maintainable & reliable than human-written code? In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 151–162. IEEE, 2025.

[58] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE, 2017.

[59] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[60] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Defining a software maintainability dataset: collecting, aggregating and analysing expert evaluations of software maintainability. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 278–289. IEEE, 2020.

[61] Agnia Sergeyuk, Olga Lvova, Sergey Titov, Anastasiia Serova, Farid Bagirov, and Timofey Bryksin. Assessing consensus of developers' views on code readability. *arXiv preprint arXiv:2407.03790*, 2024.

[62] Agnia Sergeyuk, Olga Lvova, Sergey Titov, Anastasiia Serova, Farid Bagirov, Evgeniia Kirillova, and Timofey Bryksin. Reassessing java code readability models with a human-centered approach. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 225–235, 2024.

[63] Martin Shepperd and Darrel C Ince. A critique of three metrics. *Journal of systems and software*, 26(3):197–210, 1994.

[64] 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*, pages 378–389, 2014.

[65] Charles Spearman. The proof and measurement of association between two things. In J. J. Jenkins and D. G. Paterson, editors, *Studies in Individual Differences: The Search for Intelligence*, pages 45–58. Appleton-Century-Crofts, 1961.

[66] Yahya Tashtoush, Noor Abu-El-Rub, Omar Darwish, Shorouq Al-Eidi, Dirar Darweesh, and Ola Karajeh. A notional understanding of the relationship between code readability and software complexity. *Information*, 14(2):81, 2023.

[67] Mirko Thalmann, Alessandra S Souza, and Klaus Oberauer. How does chunking help working memory? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 45(1):37, 2019.

[68] Nick Thapen. Can you fit all of this code in your head?, 10 2020. Accessed: 2025-06-10.

[69] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*, 2022.

[70] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

A Likert-scale Readability Ratings

Task ID	Rater 1 Score	Rater 2 Score	Mean Score
MBJP/1	3	3	3.0
MBJP/10	5	4	4.5
MBJP/11	4	4	4.0
MBJP/12	1	3	2.0
MBJP/14	4	5	4.5
MBJP/17	2	2	2.0
MBJP/18	4	5	4.5
MBJP/19	5	5	5.0
MBJP/2	4	5	4.5
MBJP/21	3	2	2.5
MBJP/22	5	5	5.0
MBJP/3	5	5	5.0
MBJP/4	3	3	3.0
MBJP/6	3	2	2.5
MBJP/7	4	3	3.5
MBJP/8	1	1	1.0
MBJP/9	1	2	1.5

Table 7: Raw human readability ratings and mean scores on a five-point Likert scale for the controlled synthetic dataset.

B Embedding Experiments

To guide model selection, we evaluate three representative embedding models—`bert-base-uncased`[22], `e5-base-v2`[69], and `nomic-embed-text-v1.5`[49]. These models are open-source and were released in 2019, 2022, and 2024 respectively. Each achieved strong performance at the time of release and is representative among embedding models.

For words that consist of multiple subtokens, we follow the common practice of using the mean of all subtoken embeddings as the final word representation.

B.1 Embedding Database Construction

To support our experiments, we construct three word-level datasets without filtering words based on length. The datasets are derived from word frequency statistics provided by the `wordfreq` library and synonym/antonym relations in `WordNet` [45].

1. **Synonym-Antonym Dataset**: We collect all English words of the specified lengths that appear in the `wordfreq` vocabulary and have at least one synonym or antonym listed in `WordNet`. Multi-word expressions are excluded to ensure token-level alignment.

2. **Pseudoword Dataset**: For each 4-10 word length, we generate 10,000 pseudowords using alternating vowel-consonant patterns. To ensure non-lexicality, all generated tokens are filtered against the `english-words.95` list, a size-95 subset of the SCOWL word lists that includes nearly all valid English words across dialects and rare forms.

3. **Fixed Levenshtein Distance Dataset**: For each target word from `wordfreq`, we identify other words of the same length whose Levenshtein distance is strictly less than the word length. Candidates are sorted by frequency, and up to 10 pairs are retained per distance value.

B.2 Synonym-Antonym Distance Analysis

To investigate the effect of word frequency on embedding similarity, we analyze cosine distances for synonym-antonym pairs drawn from two frequency bands: the top 10,000 words (26,493 pairs) and words ranked 10,000 to 50,000 (29,806 pairs). Pairs with only one word in the specified range are excluded. For comparison, we also compute cosine distances for 25,000 randomly sampled word pairs from each band.

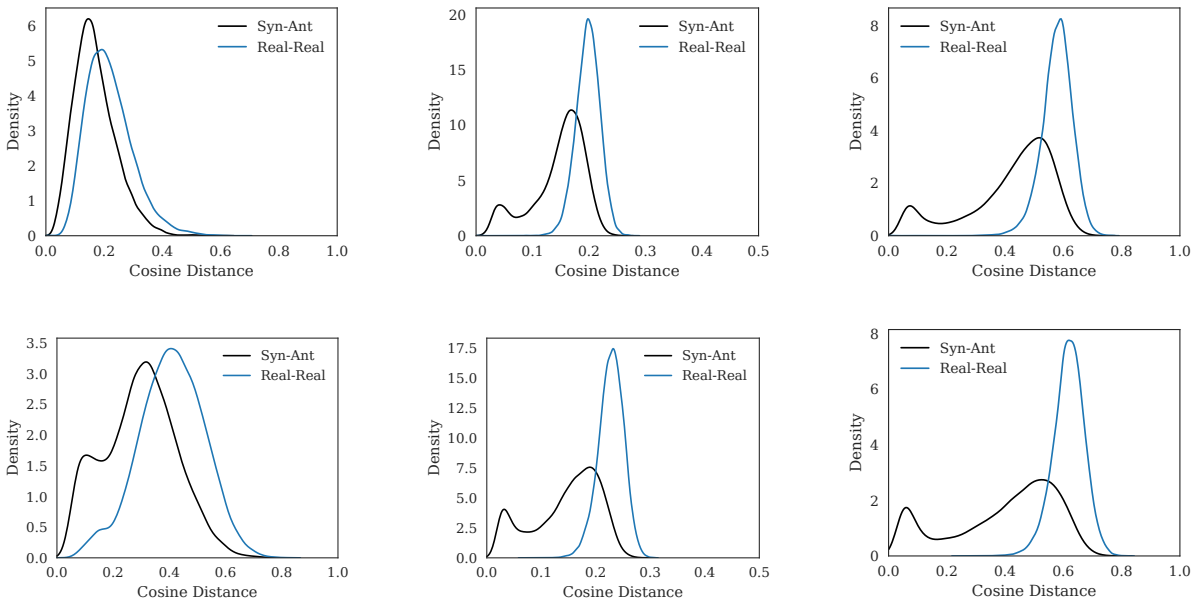


Figure 4: Cosine distance distributions between synonym-antonym pairs and randomly sampled real-real word pairs across different frequency ranges and embedding models. The top row shows results for word pairs from the top 10,000 most frequent words, while the bottom row shows results for words ranked between 10,000 and 50,000. From left to right, each column corresponds to results from BERT, E5, and Nomic embeddings, respectively. In each subfigure, the x-axis represents the cosine distance between word embeddings, and the y-axis denotes the estimated probability density.

Our results show that the `bert-base-uncased` model is highly sensitive to word frequency, with this factor having a much greater impact on embedding distances than the underlying semantic relationship between synonyms and antonyms. This sensitivity is likely a consequence of its pretraining objective, and has also been noted in prior studies[44, 35]. In contrast, models trained with contrastive learning exhibit a reduced frequency effect. To further validate this observation, we compute Cliff's Delta for the cosine distance distributions of synonym-antonym pairs across different frequency bands[17]. Among the three models, only `bert-base-uncased` exhibits a large Cliff's Delta, while both `e5-base-v2` and `nomic-embed-text-v1.5` show negligible effect sizes. This indicates that, for the latter two models, frequency-related variation is minimal compared to the impact of genuine semantic differences such as synonymy and antonymy.

Model	Cliff's Delta (0–10k vs. 10k–50k)
`bert-base-uncased`	-0.5781 (large)
`e5-base-v2`	-0.0356 (negligible)
`nomic-embed-v1.5`	0.0410 (negligible)

Table 8: Cliff's Delta values comparing cosine distance distributions between the 0–10k and 10k–50k frequency bands for each model.

B.3 Analyzing Semantic Degradation in Synonym-Antonym Networks

To ensure that the selected model remains robust across local variations in the embedding space, we further examine whether semantic drift occurs within networks of lexical relations. In particular, we design a semantic drift experiment based on WordNet, incorporating both synonymy and antonymy.

Specifically, we extract word pairs from synonym-antonym chains of length 2 and 5 steps in WordNet, where each pair is connected via intermediate relations—either synonyms or antonyms (e.g., $a \to b \to v \to d$, where each edge is a synonym or antonym link). Chains containing duplicate words are removed to avoid degenerate loops.

To account for frequency effects, we conduct the experiment separately for two frequency bands: the top 10,000 most frequent words, and words ranked 10,000–50,000. All intermediate words in the chain must fall within the specified frequency band. For each resulting (a, d) pair, we compute the cosine distance between their embeddings.

We do not consider longer chains, as the remained WordNet lexical graph approaches a complete graph at length 5, where most word pairs become connected via some relational path. Further extension therefore offers limited structural discrimination.

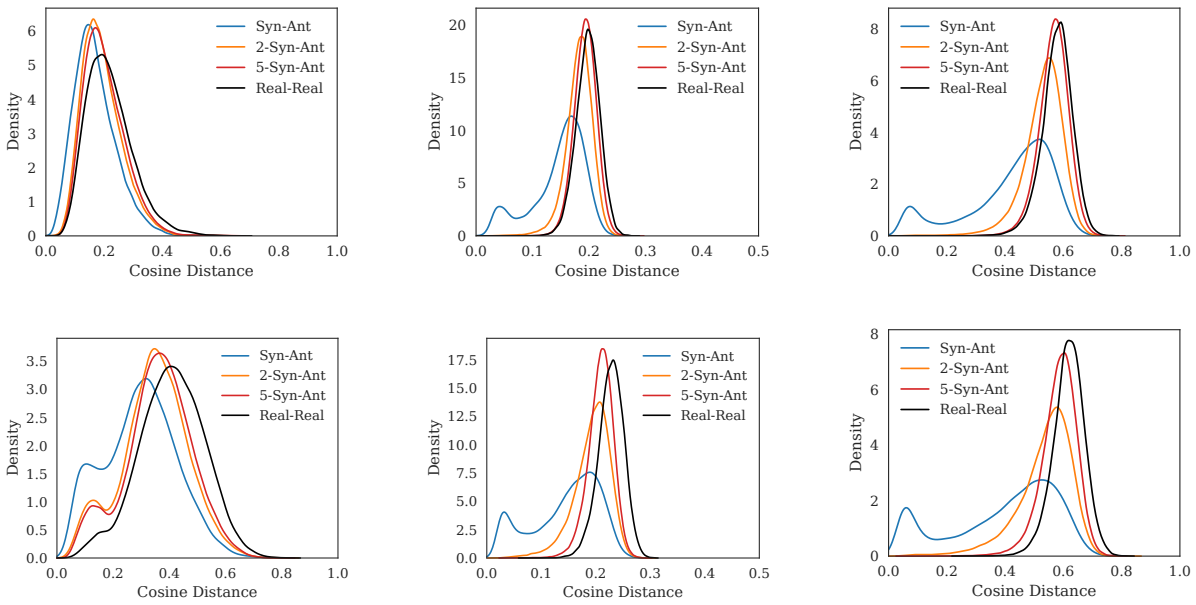


Figure 5: Cosine distance distributions between directly linked words, 2-hop lexical relation chains, 5-hop lexical relation chains, and randomly sampled word pairs, across different frequency ranges and embedding models. The top row shows results for word pairs from the top 10,000 most frequent words, while the bottom row shows results for words ranked between 10,000 and 50,000. From left to right, each column corresponds to results from BERT, E5, and Nomic embeddings, respectively. In each subfigure, the x-axis represents the cosine distance between word embeddings, and the y-axis denotes the estimated probability density.

B.4 Assessing Embedding Sensitivity to Non-Semantic Perturbations

To further evaluate whether embedding models organize word representations based on semantic meaning rather than surface-level features, we conduct a set of experiments involving real words paired with non-semantic counterparts. Specifically, we test two types of word pairs:

- **Real–Pseudoword**: each real word is paired with a pseudoword that follows English phonotactics but has no semantic meaning;
- **Real–Levenshtein1**: each real word is paired with another word of the same length and at a Levenshtein distance of 1, excluding any known synonym or antonym relation.

For both pair types, we conduct experiments separately for two frequency bands: the top 10,000 most frequent words, and words ranked between 10,000 and 50,000. For the Real-Pseudoword experiment, we randomly sample 25,000 word pairs from each frequency band. For the Real–Levenshtein1 experiment, each word is allowed up to 10 Levenshtein1 neighbors, yielding 4,847 pairs in the top-10k range and 19,706 pairs in the 10k–50k range.

Synonym and antonym pairs are included as semantic references for comparison.

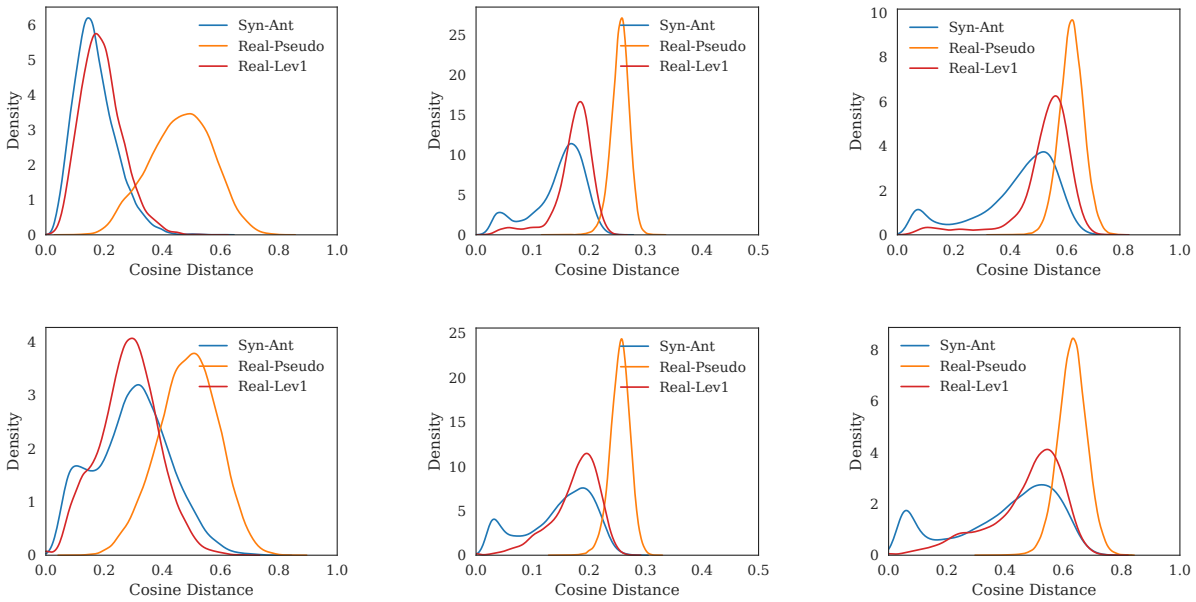


Figure 6: Cosine distance distributions between semantic word pairs (including both synonyms and antonyms), real-pseudoword pairs, and real-Levenshtein-1 pairs across different frequency ranges and embedding models. The top row shows results for word pairs from the top 10,000 most frequent words, while the bottom row shows results for words ranked between 10,000 and 50,000. From left to right, each column corresponds to results from BERT, E5, and Nomic embeddings, respectively. In each subfigure, the x-axis represents the cosine distance between word embeddings, and the y-axis denotes the estimated probability density.

C Prompts for Program Recovery and Consistency Evaluation

PreFixScore uses two separate prompts: a *program recovery prompt*, used to generate a full program from a partially revealed program, and a *consistency evaluation prompt*, used to compare the recovered program against the original one. For reproducibility, we report both prompts below. In all experiments, the placeholder {`visible_code`} denotes the revealed portion of the source program, {`original_code`} denotes the ground-truth full program, and {`predicted_code`} denotes the program generated by the recovery model.

Program Recovery Prompt. This prompt is given to the recovery model (LLM A). Its purpose is to reconstruct the full Java program from a partially revealed program prefix while preserving the revealed content.

```
You are a software engineer. Complete the full Java program from the
revealed prefix.

Requirements:
- Do not explain or analyze.
- Output a fenced code block containing the full program, not only the suffix.
- Preserve the revealed prefix and do not change its semantics.
- If some details are uncertain, still produce the most reasonable complete
  program.

Program prefix:
```java
{visible_code}
```
```

Consistency Evaluation Prompt. This prompt is given to the comparison model (LLM B). Its purpose is to judge whether the reconstructed program is semantically consistent with the original program under the criteria defined in Section 4.2.

```
You are a software engineer. Compare the two Java programs directly and
judge whether they are consistent.

Judge only by these 3 rules:

1. The code intent must match.
2. The implementation approach must match.
3. There must be no obvious side effects.

If the task is the same but the implementation is different, mark them as
inconsistent.
If the program contains dead code, repeated logic, irrelevant branches, or
unrelated helper logic that is not part of the core implementation, mark
them as inconsistent.

Output only one JSON object in exactly this format:
{
"task_match": true,
"method_match": true,
"io_match": true,
"side_effects_match": true,
"redundancy_ok": true,
"method_evidence": [
"evidence 1"
],
"detail_conflicts": [
"conflict 1"
],
"detail_alignment": [
"alignment 1"
],
"equivalent": true,
"reason": "one-sentence reason",
"confidence": 0.0
}

Original program:

‘‘‘java
{original_code}
‘‘‘

Completed program:

‘‘‘java
{predicted_code}
‘‘‘
```

Use in PreFixScore. At each exposure stage, the recovery prompt is first applied to the partially revealed program to obtain a completed candidate. The consistency evaluation prompt is then applied to the original and recovered programs. A recovery is treated as successful only when the comparison output indicates semantic equivalence under the predefined matching criteria.

D Prompts for LLM-based Readability Evaluation

This appendix provides the prompt template used for the direct LLM-based readability baseline. The prompt follows a zero-shot (ZSL) strategy based on the approach proposed by Ouédraogo et al. [51], and is designed to elicit structured readability assessments from the model.

````
You are a software engineer evaluating code readability.

Evaluate the following code on a scale from 0 (unreadable) to 20 (highly readable),
explicitly using these criteria:

1. Code Structure - logical organization vs tangled code
2. Nesting - flat vs deeply nested
3. Clarity of intent - clear purpose vs ambiguous
4. Code length - concise vs unnecessarily long
5. Action granularity - one action per line vs multiple
6. Reading flow - natural vs disorganized

Return ONLY JSON:
{
  "score": "0-20",
  "reasoning": "concise explanation (max 200 words)"
}

Code:
```java
{code}
```
````