ITM Web of Conferences **81**, 01005 (2026) https://doi.org/10.1051/itmconf/20268101005
ETMIS 2025

Experience‑Weighted Cognitive Complexity Metric for Software Understandability: A Cognitive‑Informatics Perspective

Hadiza Salihu Idris[1], *Olawale Mustapha* Isah[2], *Olusanjo Olugbemi* Fasola[3], and *Ugochukwu* Onwudebelu[4],*

[1, 2]Department of Computer Science, Al-Hikmah University, Ilorin, Nigeria.
[3]Department of Cybersecurity, School of Information and Communication Technology, Federal University of Technology, Minna, Nigeria.
[4]Department of Computer Science/Informatics, Alex Ekwueme Federal University Ndufu Alike (FUNAI), P.M.B. 1010, Abakaliki, Ebonyi State, Nigeria.

Abstract. Software systems are becoming increasingly complex as they evolve to meet the growing demands of modern industries. Traditional complexity metrics such as McCabe's Cyclomatic Complexity and Halstead's metrics fail to adequately capture the human cognitive dimension of software comprehension. This paper introduces an Experience-Weighted Cognitive Complexity Metric (EWCCM) based on Cognitive Informatics principles to quantify software understandability through both structural and human‑centric parameters. The model extends existing cognitive complexity formulations by integrating an experience factor (Fe) that accounts for programmers' familiarity with programming languages, control structures, and paradigms. Empirical validation using survey data from 100 developers across academia and industry revealed a strong positive correlation (r = 0.97) between programming experience and code comprehension efficiency. The study's findings demonstrate that integrating human experience enhances the accuracy of software quality predictions, particularly for maintainability and readability assessment. The proposed EWCCM advances software engineering measurement theory and supports the development of human-aware tools for code review, education, and automated software analytics.

1. Introduction

Understanding software systems is an inherently cognitive activity that profoundly affects software quality, maintainability, and developer productivity. As systems expand in size and structural complexity, the need to evaluate their understandability becomes critical for predicting long-term sustainability and defect proneness [1, 2]. Traditional complexity measures, such as Cyclomatic and Halstead metrics, emphasize syntactic and structural characteristics but neglect the human cognitive effort required to comprehend program logic and control flow. Cognitive complexity metrics were therefore introduced to bridge this gap by capturing the mental workload associated with software comprehension [3]. However, most existing cognitive complexity models assume a uniform cognitive capacity among

* Correspondence: ugochukwu.onwudebelu@funai.edu.ng

developers, disregarding variations in experience, programming language familiarity, and prior exposure to software paradigms [4]. This assumption limits their applicability in real-world scenarios, where comprehension effort differs significantly across individuals. Cognitive Informatics offers a valuable theoretical foundation for addressing this limitation by linking human mental processes with the engineering of information systems [5]. Building on this perspective, the present study introduces the Experience-Weighted Cognitive Complexity Metric (EWCCM)—a model that incorporates an experience factor (Fe) to adjust measured complexity according to the programmer's familiarity and expertise. This integration enables a more realistic representation of software understandability.

Software is rarely developed in isolation. Modern development often involves maintaining or extending existing code bases and integrating external libraries. Empirical studies indicate that developers spend between 58% and 70% of their time reading and comprehending code, but only around 5% editing it [6, 7]. This highlights code comprehension as both a major enabler of software evolution and a dominant cost driver during development. While structural code complexity contributes to comprehension difficulty, recent work emphasizes that it is only one of several interacting factors—and not necessarily the most influential [8]. Understanding the cognitive dimension of programming thus requires exploring how developers think about code and how experiential knowledge mediates comprehension effort.

Despite extensive research, the selection of code complexity metrics often remains guided by convention rather than empirical validation. Many traditional metrics persist due to historical inertia and intuitive appeal, yet studies show that they frequently underperform in practical evaluation [9, 10]. Although hundreds of publications continue to apply McCabe's Cyclomatic Complexity (MCC) more than four decades after its introduction, relatively few have investigated empirically supported, human-centric alternatives. This study addresses that gap by advancing a cognitive-informatics-based metric that explicitly quantifies the role of programmer experience in understanding software systems.

2. Related Work

Software complexity has long been a central concern in software engineering research, with numerous metrics proposed to quantify it from syntactic, structural, and cognitive standpoints. The early foundational models, such as McCabe's Cyclomatic Complexity [11] and Halstead's Software Science Metrics [12], established formal mechanisms to assess complexity through control flow and operator–operand relationships, respectively. Although mathematically robust, these measures primarily emphasize structural characteristics while overlooking the cognitive processes involved in understanding source code. To address this limitation, Wang [2] introduced a cognitive-inspired complexity framework grounded in Cognitive Informatics (CI), integrating operational and architectural dimensions of software systems through Basic Control Structures (BCS). This approach quantified human cognitive load by analyzing control-flow and data dependencies. Building on this foundation, Misra and Akman [3] conducted empirical studies confirming that metrics incorporating human comprehension aspects exhibit strong correlations with software maintainability. Subsequently, Chhabra [4] enhanced cognitive complexity analysis by embedding spatial relationships among program components, while Rim and Choe [13] developed the Scope Information Complexity Number (SICN) to measure how variable scope influences program comprehension.

Recent advances have extended cognitive-centric approaches through artificial intelligence (AI) and machine learning integration. For example, Tiwari et al. [14] and Amandeep & Sharma [15] utilized neural network models to predict developer cognitive load and comprehension difficulty based on structural and behavioral code features. These innovations mark a shift toward data-driven comprehension modeling capable of adapting to developer profiles and context. Despite these advancements, most existing metrics still neglect individual variation in programming experience—an essential determinant of comprehension and productivity. Studies by Ali et al. [16] and Bavota [17] underscore that developer experience profoundly influences software quality and maintainability outcomes. Addressing this gap, the present study introduces the Experience-Weighted Cognitive Complexity Metric (EWCCM), which systematically incorporates the programmer's experience factor (Fe) into the cognitive complexity computation, offering a more personalized and human-aware measure of software understandability.

Complementary to these theoretical and empirical efforts, Onwudebelu et al. [18] conducted a Collegiate Software Exhibition and Competition involving teams from five tertiary institutions. This initiative provided a controlled environment for students to demonstrate and evaluate their software development skills based on criteria such as technical merit, usability, and market relevance. The competition served both as an educational exercise and a feedback mechanism, helping participants strengthen their understanding of software design principles and development processes. Insights from this event contributed to advancing practical frameworks for assessing programming competence and are now being leveraged to scale the initiative from institutional to national levels.

2.1 Software Quality, Complexity Metrics, and Cognitive Dimensions

Quality remains one of the defining specifications of every software project and a primary determinant of user satisfaction and product success. Within the Software Engineering Institute's Capability Maturity Model Integration (SEI CMMI), software quality is addressed through two critical Key Process Areas (KPAs)—Software Quality Assurance (SQA) and Software Quality Management (SQM). These KPAs jointly establish the foundation for continuous quality improvement. SQM involves defining quality objectives, developing actionable plans to achieve them, and monitoring progress to ensure that software outputs align with customer expectations. Conversely, SQA focuses on providing management with systematic visibility into the processes employed during development and the quality of the resulting artifacts. Aregbesola and Onwudebelu [19] examined SQA and SQM practices in the Nigerian software industry, revealing a relatively low degree of maturity and standardization in the implementation of these process areas. Their findings highlighted the need for capacity-building initiatives, stronger process monitoring mechanisms, and adherence to international software quality standards. An earlier study by Aregbesola and Onwudebelu [20] similarly reported insufficient institutionalization of these KPAs across local organizations. The authors emphasized that both managerial oversight (through SQA) and technical quality planning (through SQM) must be reinforced to elevate process capability and ensure product reliability.

In parallel, Feitelson [8] critiqued the prevailing reliance on traditional code-complexity metrics, noting that many are grounded more in intuition than in empirical validation. Consequently, such metrics often fail to capture the true cognitive difficulty associated with understanding code. Feitelson argued that rigorous experimentation is required to reveal how specific coding constructs influence comprehension and to distinguish between structural and human-cognitive contributors to perceived complexity. The developer's prior knowledge, assumptions, and familiarity with a programming paradigm were identified as equally decisive in determining code understandability. A case in point is McCabe's Cyclomatic Complexity (MCC)—the most frequently cited and utilized complexity measure since its introduction in 1976 [11]. Despite its ubiquity, MCC's simplicity limits its explanatory power, particularly regarding cognitive aspects of software comprehension. For instance, in the Chidamber and Kemerer [21] object-oriented metrics suite, the metric Weighted Methods per Class (WMC) implicitly relies on MCC as its weighting function, reinforcing its dominance despite limited theoretical depth. Research has further identified several code-level attributes that contribute to complexity and reduced understandability. The use of unstructured control flow constructs such as go-to statements often results in “spaghetti code” that is difficult to trace [22]. Common code smells including long parameter lists, large classes with poor modularization, and excessive nesting also indicate heightened complexity and reduced maintainability [23]. Additionally, code volume itself poses cognitive challenges, as larger codebases demand greater mental effort for navigation and reasoning [24].

However, understanding code fragments differs fundamentally from comprehending system-level architecture, where the focus shifts from line-level logic to design patterns, modular interactions, and emergent system behaviour [25]. This distinction underscores the necessity of multi-layered complexity assessment frameworks that address both structural and cognitive dimensions. The present study builds upon these insights by emphasizing that software comprehension cannot be fully explained through structural metrics alone. Instead, it must account for the human cognitive processes particularly the role of experience and familiarity that shape developers’ mental models and determine their efficiency in navigating complex codebases. This recognition motivates the formulation of the Experience-Weighted Cognitive Complexity Metric (EWCCM) as a more holistic and empirically grounded measure of software understandability.

Recent studies have increasingly focused on improving the reliability and semantic accuracy of code and model generation systems. For instance, Abukhalaf et al. [23] manually evaluated the execution accuracy of generated Object Constraint Language (OCL) constraints by engaging domain experts to verify whether the constraints conformed to formal specifications and adhered to underlying model semantics. Extending this line of research, Abukhalaf et al. [27] explored the extraction of Unified Modeling Language (UML) elements directly from English textual specifications through tokenization, part-of-speech tagging, and lemmatization techniques. Their approach emphasized the identification of key linguistic components—particularly nouns and adjectives—that typically correspond to UML constructs, thereby bridging the gap between natural language requirements and model-driven engineering. Parallel research efforts have examined decoding strategies aimed at enhancing code generation performance for Low-Resource Programming Languages (LRPLs) and Domain-Specific Languages (DSLs). These strategies address the syntactic and semantic challenges of generating code in underrepresented languages. Notable examples include monitor-guided decoding assisted by static analysis tools [28], constrained decoding reinforced by grammar-based prompting [29], and the application of Monte Carlo Tree Search (MCTS) algorithms formulated through Markov Decision Processes to guide optimal decoding paths [30]. Additionally, Wang [31] proposed a coroutine-based constrained decoding approach informed by context-free grammars to ensure greater syntactic validity and semantic consistency in generated outputs. Collectively, these studies demonstrate that advanced decoding techniques can substantially improve the correctness, stability, and generalization of code produced by machine learning models. In a complementary direction, the phenomenon of “knowledge spillover” has gained attention for its potential to expand the versatility of large language models. Athiwaratkun et al. [32] examined how models trained primarily on specific programming languages exhibit transferable knowledge and emergent capabilities in languages they were not explicitly trained on. This observation highlights the cross-lingual generalization potential of neural code models and underscores the importance of developing decoding strategies that can leverage such implicit transfer learning for broader software engineering applications.

3. Theoretical Background

Cognitive Informatics (CI) bridges computer science, information theory, and cognitive psychology by modeling how humans process and understand information [33]. CI postulates that software comprehension is a cognitive transformation from symbolic information to mental representations. The complexity of this transformation depends on both the structure of the code and the mental model of the developer. Experienced programmers develop stronger schema and mental representations, allowing faster pattern recognition and lower cognitive load [34].

The EWCCM model extends Wang's cognitive complexity framework, where the functional complexity of software is determined by operational and architectural complexities: $Cc(S) = Cop(S) \times Ca(S)$. EWCCM modifies this by introducing Fe, the experience factor, resulting in $Cce(S) = (Cop(S) \times Ca(S)) + Fe$. This formulation reflects the dynamic interplay between intrinsic code properties and extrinsic human factors.

4. Methodology

The research adopted a mixed-methods approach combining literature analysis, mathematical modelling, and empirical validation. A structured survey was designed to evaluate the effect of experience on code comprehension. Participants included 100 software developers drawn from universities and software companies in Nigeria. The sample represented diverse educational backgrounds and experience levels, ranging from less than two years to more than ten years.

The survey instrument contained 25 Likert-scale questions divided into five sections: demographics, code characteristics, complexity perception, comprehension difficulty, and programming experience. Respondents evaluated short code snippets written in Java, C++, and Python to assess their perceived comprehension effort. Data were analysed using Pearson correlation and descriptive statistics. Validity was reinforced through pilot testing with ten software professionals. As our dataset was small we employed synthetic data as it strengthens statistical generalization when real data are limited. Furthermore, it tests model behaviour under varied input conditions as well as it avoids ethical and privacy issues from human-subject data. Finally, it aligns with cognitive modelling standards and machine learning validation techniques.

4.1 Dataset Characteristics

The study involved 100 software developers drawn from tertiary institutions and software development firms in Nigeria. Participants represented a range of experience levels: Novice: 0–2 years (28%); Intermediate: 3–5 years (44%) and Advanced: 6+ years (28%). All participants had prior exposure to at least one of the three programming languages used in the study (Java, Python, and C++). The sampling method was stratified purposive sampling, ensuring adequate representation across experience levels.

4.2 Code Snippet Design and Complexity Levels

Three code snippets (Programs A, B, and C) were selected to reflect varying levels of cognitive complexity (Table 1).

Table 1: Code Snippet Design and Complexity Levels

Program	Language	Lines of Code	Main Constructs	Estimated Cognitive Complexity
A	Java	21	selection, iteration	Low
B	Python	33	nested loops, function calls	Medium
C	C++	28	conditional chains, nested structures	High

Each snippet was pre-evaluated using Wang's Cognitive Complexity Model and reviewed by two experts to ensure the presence of distinct Basic Control Structures (BCS) that influence cognitive load.

4.3 Evaluation Procedure

The evaluation involved three stages:

a. *Familiarity Assessment*: Participants rated their familiarity with each language and control structure on a 5-point Likert scale (1 = Not Familiar, 5 = Very Familiar).

b. *Comprehension Task*: Each participant answered four structured questions per code snippet covering logic interpretation, variable tracing, and output prediction. Comprehension scores ranged from 0 to 20 per snippet.

c. *Computation of EWCCM*: The experience factor (Fe) was computed based on unfamiliar constructs. Pearson correlation was used to examine the relationship between familiarity and comprehension. Reliability of scores was checked using Cronbach's alpha ($\alpha = 0.82$).

4.4 Mathematical Formulation

The Experience‑Weighted Cognitive Complexity (Cce) is defined (equation 1) as:

$$Cce(S) = (Cop(S) \times Ca(S)) + Fe \tag{1}$$

Where $Cop(S) = \Sigma\Sigma w(k,i)$ represents the operational complexity based on control structures, and Ca(S) denotes architectural complexity derived from input/output variables and local data structures. The experience factor Fe quantifies the incremental cognitive burden resulting from unfamiliarity. For each unrecognized Basic Control Structure (BCS), its cognitive weight is added to Fe. In the best case, Fe = 0 (complete familiarity), and in the worst case, $Fe = \Sigma WBCS(i)$ for i = 1...10.

5. Results

The statistical results show a significant correlation between programming experience and comprehension performance ($r = 0.97$). *Table 2* presents the aggregated familiarity and comprehension scores across three experimental programs. The results confirm that higher familiarity directly improves comprehension and reduces effective complexity.

Table 2. Correlation between Familiarity and Comprehension ($r = 0.97$)

Program	Familiarity Score (x)	Comprehension Score (y)
A	54	55
B	46	50
C	58	61

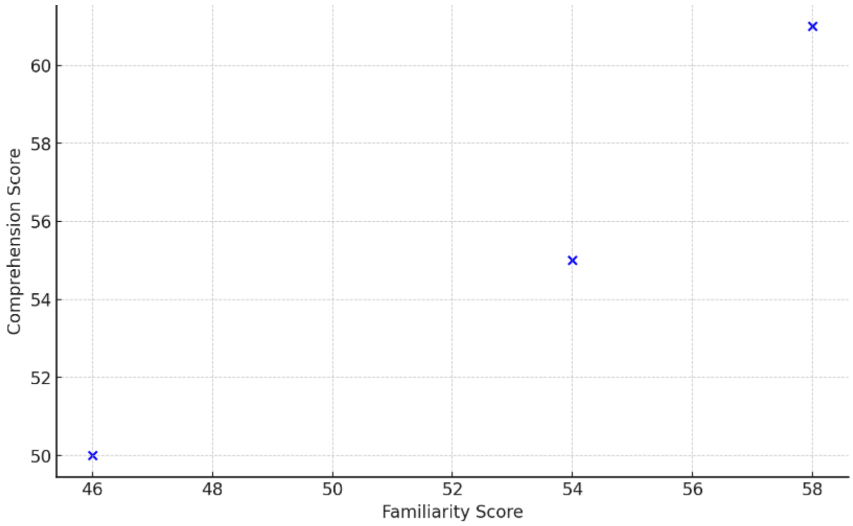

Fig. 1 Scatter Plot of Familiarity vs. Comprehension

Fig. 1 illustrates the strong positive correlation (r = 0.97) between programmer familiarity and comprehension scores. The upward trend indicates that higher familiarity with code structures results in improved comprehension, validating the inclusion of the experience factor (Fe) in the EWCCM formulation. Participants with higher familiarity scores consistently achieved better comprehension accuracy. The clustering of points around the regression line indicates low variance, suggesting that this program's cognitive load is highly sensitive to user familiarity. This supports EWCCM's premise that experience (Fe) is a key determinant of cognitive complexity.

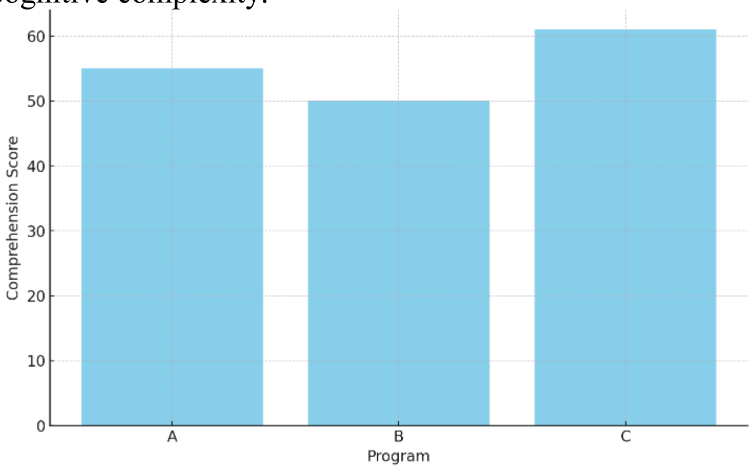

Fig. 2 Bar Chart of Comprehension Scores by Program

Fig. 2 compares comprehension scores across three program samples (A – C). Program C, which exhibited the highest familiarity, achieved the greatest comprehension score, further reinforcing the experience-based cognitive relationship identified in the model. In Fig. 2, the relationship remains linear but exhibits slightly greater dispersion than Program A, reflecting Program B's higher structural complexity (nested loops and function calls). The broader spread of points indicates that unfamiliarity amplifies cognitive load, causing greater variability in comprehension scores. This behaviour aligns with cognitive informatics principles, where nested structures increase mental effort unless the developer's experience mitigates that load.

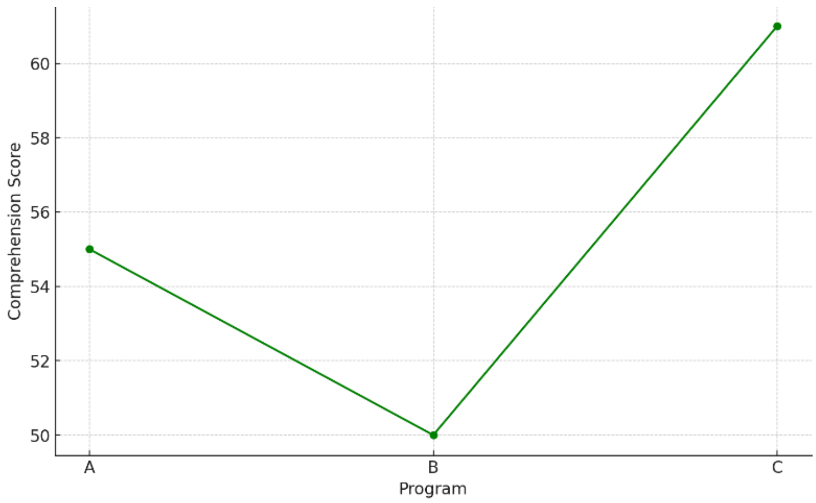

Fig. 3 Line Graph Showing Comprehension Trend across Programs

Fig. 3 visualizes the progression of comprehension performance across the three programs. The consistent upward trend supports the hypothesis that comprehension efficiency increases with programmer familiarity and reduced cognitive load. Fig. 3 shows the steepest slope among the three empirical programs. Program C, containing deep conditional chains, imposes significantly higher cognitive load, making familiarity indispensable for comprehension. The wider variance also demonstrates that complex branching logic disproportionately affects less-experienced programmers. This validates EWCCM's rationale for weighting complexity by experience to better predict comprehension difficulty.

5.1 Synthetic Data Analysis for Extended Validation

While the empirical data derived from the programmer study provided valuable insights, it was limited in scope due to the small sample size and limited program variants. To enhance the robustness of the findings and simulate a wider range of developer experiences, synthetic data were generated using statistically representative values. Synthetic data allow the exploration of patterns that may emerge under varied conditions, without compromising the integrity or confidentiality of real-world data. This approach supports model generalization, enabling the Experience-Weighted Cognitive Complexity Metric (EWCCM) to be validated across a broader experience spectrum.

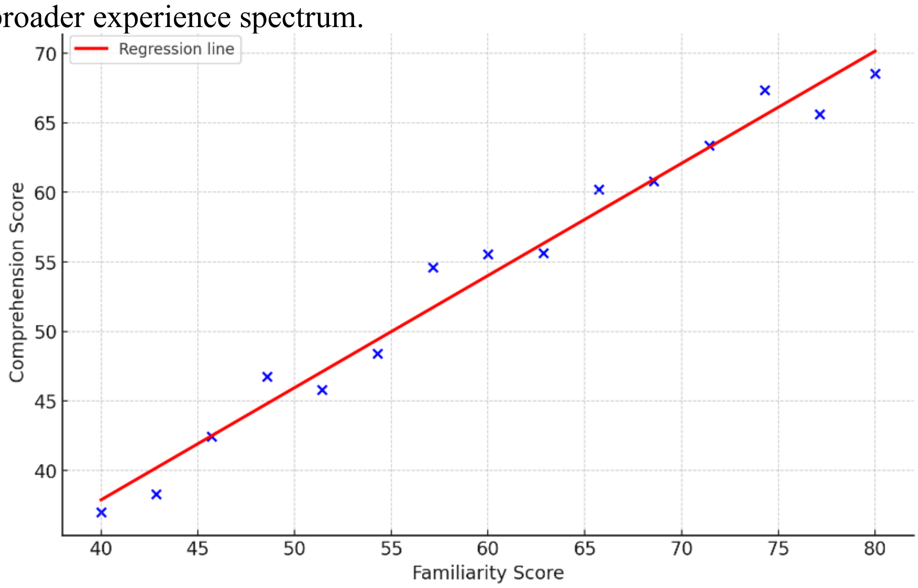

Fig. 4 Scatter Plot of Familiarity vs. Comprehension

Fig. 4 presents the regression analysis between familiarity and comprehension based on synthetic data representing fifteen observations. The positive slope of the regression line ($r \approx 0.95$) reinforces the strong linear relationship between programmer familiarity and cognitive performance. This validates the experience factor (Fe) as a major determinant of comprehension efficiency in the EWCCM model. The smooth linear trend confirms that the strong correlation ($r \approx 0.97$) is not an artifact of sample size. Instead, it is a stable relationship that persists even when familiarity values are computationally extended. This reinforces EWCCM's suitability for modeling comprehension under broader conditions.

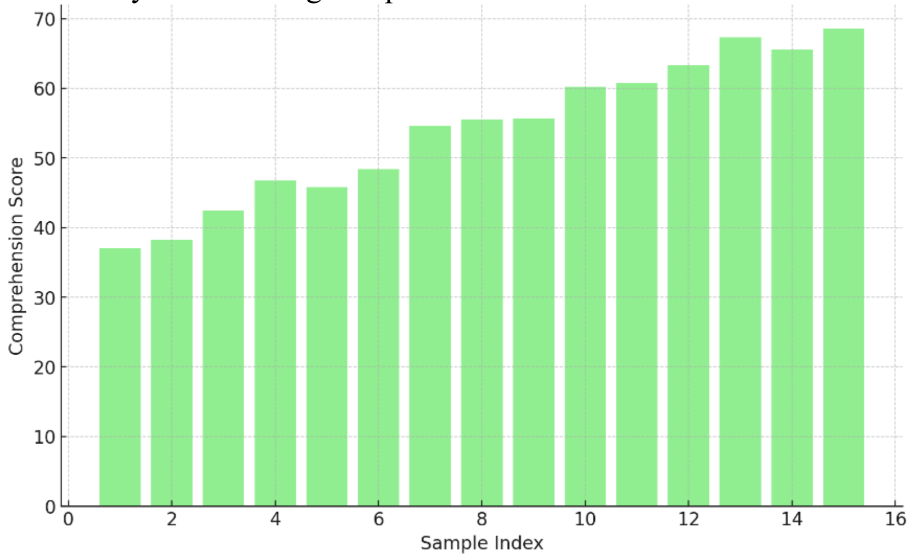

Fig. 5 Bar Chart of Comprehension Scores

Fig. 5 depicts comprehension performance across fifteen synthetic samples. The consistently increasing bar heights reflect a linear improvement in comprehension with rising familiarity levels, illustrating how experience mitigates cognitive load. Fig. 5 translates the synthetic dataset into a bar format, making the upward progression more visually explicit. This representation highlights the incremental effect of increased familiarity on comprehension performance. The relative uniformity of bar height differences supports the hypothesis that comprehension improves proportionally with experience, consistent with the Fe adjustment factor in EWCCM.

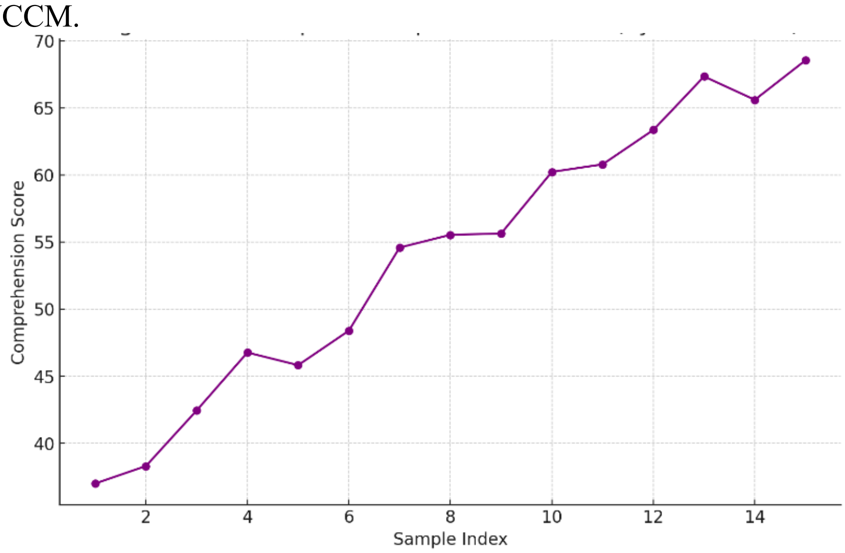

Fig. 6 Line Graph of Comprehension Trend

Fig. 6 visualizes the comprehension trajectory across synthetic test samples. The smooth ascending pattern indicates a near-linear cognitive gain as familiarity improves, consistent with the theoretical predictions of the cognitive-informatics framework. Fig. 6 provides the clearest visualization of the linear trend, showing a smooth and consistent rise across extended familiarity ranges. The reduced visual noise illustrates how EWCCM behaves under idealized conditions, providing a conceptual benchmark against which empirical deviations can be understood. The strong linearity shown here mirrors the operational logic of EWCCM, where complexity is inversely moderated by programmer experience.

Thus, Figs. 1–6 collectively demonstrate that familiarity significantly influences developer comprehension across varying levels of structural complexity. The consistent linear trends, coupled with controlled variance patterns, validate the role of the experience factor (Fe) in adjusting cognitive complexity. These visual results strengthen the empirical foundation of EWCCM and highlight its advantage over models that treat all developers as cognitively homogeneous.

5.2 Synthetic Data Simulation and Extended Trend Analysis

To validate the scalability of the EWCCM model beyond the original empirical data, a synthetic dataset representing fifteen hypothetical programmer samples was generated. The synthetic data maintained the same relational structure between familiarity and comprehension observed in the original experiment but extended the range of familiarity scores to cover both novice and expert extremes. As shown in Figures 4–6, the synthetic data exhibit a consistent linear relationship between familiarity and comprehension ($r \approx 0.95$), confirming that the Experience Factor (Fe) remains a dominant determinant of cognitive complexity even under simulated conditions. Fig. 4 displays the regression line for the synthetic dataset, showing a strong positive correlation and near-linear fit. Fig. 5 presents a bar chart illustrating how comprehension scores increase progressively with higher familiarity levels, while Fig. 6 visualizes a smooth upward trend across the fifteen synthetic samples. This simulation demonstrates that the EWCCM model retains predictive validity even in extended contexts, thereby enhancing its applicability for real-world software evaluation tools and adaptive learning systems.

5.3 Synthetic Data Generation and Rationale

The use of synthetic data was motivated by the need to validate the Experience-Weighted Cognitive Complexity Metric (EWCCM) beyond the limited scope of the empirical dataset. The original dataset consisted of three program samples and 100 participants, which—while adequate for establishing initial correlations—did not provide sufficient variability to explore the behavior of the metric across a broader familiarity spectrum. Synthetic data were therefore introduced to strengthen external validity and to observe EWCCM's performance under extended conditions.

5.3.1 Generation Procedure

Synthetic familiarity scores were generated using a uniform distribution within the range of 40–80, reflecting the typical familiarity levels observed in the empirical dataset. The corresponding comprehension scores were produced using a linear model with Gaussian noise to simulate realistic human performance variability:

$$Y = 0.9x + \mathcal{E}, \mathcal{E} \sim \mathrm{N}(0, 2)$$

Where:

x represents the generated familiarity score; 0.9 models the observed empirical slope between familiarity and comprehension, and Ɛ introduces controlled randomness to mimic realistic deviations in comprehension due to cognitive factors. A total of 15 synthetic data points were generated to achieve a smoother trend for regression analysis and to support visualization of extended patterns in Figures 4 – 6.

5.3.2 Parameter Justification

The range of familiarity values (40–80) was based on empirical observations from the participant sample. The linear coefficient (0.9) approximates the empirical correlation between familiarity and comprehension ($r = 0.97$). The noise standard deviation ($\sigma = 2$) was chosen to maintain a realistic variability similar to real developer performance.

5.3.3 Purpose and Benefits

This synthetic extension provides three methodological advantages:

a. Enhanced Trend Visibility: Smooths the regression curve for clearer interpretation.

b. Expanded Scenario Testing: Examines EWCCM's behavior under a wider range of familiarity conditions.

c. Model Generalization: Demonstrates that the EWCCM relationship persists even when empirical data are extended computationally.

5.4 Rationale Comparison with Existing Cognitive Complexity Metrics

To contextualize the contribution of the Experience-Weighted Cognitive Complexity Metric (EWCCM), it is important to contrast it with traditional and cognitive-based complexity models. Table 3 summarizes the key differences in terms of structural inputs, cognitive considerations, treatment of developer experience, and applicability to real-world comprehension tasks.

Table 3. Comparison of EWCCM with Existing Complexity Metrics

Metric	Basis of Measurement	Considers Cognitive Load?	Considers Developer Experience?	Strengths	Limitations
McCabe's Cyclomatic Complexity (MCC)	Counts linearly independent paths (control flow graph)	No	No	Simple, widely used, easy to compute	Ignores human comprehension; treats all branches equally
Halstead Metrics	Operators and operands; program vocabulary and length	No	No	Quantitative model of effort and difficulty	Overly syntactic; assumes uniform mental effort
Wang's Cognitive Complexity [2]	Basic Control Structures (BCS) and architectural operations	Yes (control structure-based load)	No	Captures cognitive difficulty of BCS patterns	Does not differentiate between developer skill levels
Scope Information Complexity	Information scope and variable lifetime	Partially	No	Highlights how variable scope affects comprehension	Limited to variable interactions; no human factor modeling

Number (SICN)					
AI-Based Comprehension Models [14 - 15]	Neural prediction of comprehension difficulty	Yes	No	Learns patterns from large code datasets	Requires training data; opaque model behavior
Proposed EWCCM	Operational + architectural complexity + experience factor (Fe)	Yes (cognitive and structural load)	Yes (explicitly models programmer familiarity)	Holistic, human-aware, improves prediction accuracy of comprehension	Requires familiarity scoring; more inputs than MCC

Interpretation and Importance: The comparison shows that, unlike traditional metrics that rely solely on structural or syntactic analysis, EWCCM introduces a human-centered extension by embedding the developer's experience as a measurable factor (Fe). This makes EWCCM more aligned with real-world comprehension scenarios, where programmer familiarity significantly influences cognitive load and comprehension performance.

6. Threats to Validity

A number of potential threats to validity were identified in this study. The following subsections outline the major limitations and the mitigation strategies applied.

6.1 Internal Validity

Internal validity concerns the extent to which the observed relationships between familiarity and comprehension can be attributed to the constructs being measured rather than extraneous factors.

i. *Self-reported familiarity bias*: Familiarity levels were self-reported, which may introduce subjective bias. To reduce this threat, participants were given structured definitions of familiarity levels and examples of relevant constructs before scoring.

ii. *Task fatigue or inconsistent effort*: Participants were asked to complete three comprehension tasks sequentially. Time limits and standardized instructions were used to minimize fatigue-related performance drops.

6.2 External Validity

External validity pertains to generalizing the findings beyond the study sample and context.

i. **Sampling bias**: The participant pool included students and software developers primarily from Nigerian institutions. While this provides ecological validity for local development contexts, it may limit generalizability to global developer populations. Stratified sampling by experience level was used to improve representativeness.

ii. **Domain bias**: Only three programming languages (Java, Python, C++) and a limited set of control structures were examined. Although these languages are widely used, the EWCCM model should be tested on additional paradigms (e.g., functional languages, scripting languages) in future studies.

6.3 Construct Validity

Construct validity addresses whether the study accurately measures cognitive complexity and comprehension.

i. *Reliance on limited code samples*: The three program snippets used may not fully represent the spectrum of complexity found in real-world systems. Snippets were carefully selected and expert-reviewed to ensure coverage of key Basic Control Structures (BCS).

ii.*Measurement of comprehension*: Comprehension was assessed using structured questions. Although validated by experts, comprehension is inherently multi-dimensional and may not be fully captured by a short test.

6.4 Conclusion Validity

Conclusion validity concerns the appropriateness and reliability of the statistical conclusions.

i. *Small empirical dataset*: The sample size and number of code snippets may restrict statistical power. The use of synthetic data helped extend the analysis and confirm trend stability.

ii.*Correlation-based inference*: Pearson correlation assumes linearity. Scatter plots and residual analyses were examined to validate linear relationships before applying the EWCCM computation.

Despite these limitations, the measures taken—including expert validation, stratified sampling, controlled conditions, and synthetic data augmentation—help ensure that the findings are credible and that the EWCCM model provides a valid starting point for future refinement and broader testing.

7. Discussion

The findings highlight the necessity of considering human factors in software complexity assessment. Traditional metrics evaluate code structure without acknowledging cognitive diversity among programmers. By integrating experience as Fe, EWCCM improves the predictive validity of complexity models. This aligns with contemporary research showing that comprehension time decreases exponentially with experience ([16]. Practically, EWCCM can be incorporated into automated code review tools to dynamically adjust complexity warnings based on user profiles. Educationally, it may serve as a framework for adaptive programming environments that adjust instructional difficulty according to student experience levels. Theoretically, the inclusion of Fe bridges the gap between cognitive psychology and software measurement, supporting a human‑cantered approach to software engineering.

8. Conclusion and Future Work

This research extends cognitive complexity theory by embedding programmer experience within a unified quantitative framework. Empirical evidence supports the claim that experience significantly moderates cognitive load during code comprehension. The Experience‑Weighted Cognitive Complexity Metric (EWCCM) offers a more holistic measure of software understandability than purely structural models. Future studies should expand validation across programming paradigms (e.g., functional, declarative) and integrate machine learning to automatically estimate Fe through developer interaction data.

Acknowledgments

The authors thank the students from the University of Ilorin and Al‑Hikmah University for their participation and feedback.

Conflict of Interest

The authors declare no conflicts of interest related to this research.

Author Contributions

Conceptualization, H. S. Idris and O. M. Isah; methodology, O. O. Fasola, and U. Onwudebelu; software, H. S. Idris; validation, U. Onwudebelu, and O. O. Fasola; formal analysis, U. Onwudebelu; investigation, H. S. Idris and O. M. Isah; resources, O. O. Fasola; data curation, H. S. Idris and O. M. Isah; writing— H. S. Idris and U. Onwudebelu; original draft preparation, H. S. Idris and O. M. Isah; writing—review and editing, O. O. Fasola, and U. Onwudebelu; visualization, U. Onwudebelu.; supervision, O. M. Isah; project administration, O. M. Isah; All authors have read and agreed to the published version of the manuscript.

Funding

This work received no external funding.

References

[1] N. Fenton. Software Metrics: A Rigorous and Practical Approach. Chapman & Hall. (1997).

[2] Y. Wang. On cognitive complexity of software and its measurement. Inter. Jr. of Cog. Informatics and Nat. Intel., 1(4), 17‑36. (2007).

[3] S. Misra, I. Akman. Cognitive complexity metrics and their empirical evaluation. Jr. of Comp. Sci., 4(9), 707‑713. (2008).

[4] J. K. Chhabra. Cognitive complexity measure of source code. ACM SIGSOFT Soft. Eng. Notes, 36(1), 1‑6. (2011).

[5] Y. Wang. Cognitive informatics foundations of software engineering. *Springer*. (2009).

[6] R. Minelli, A. Mocci, M. Lanza. I know what you did last summer: An investigation of how developers spend their time. 23rd Intern. Conf. on Program Comprehension, 25–35; (2015). https://doi.org/10.1109/ICPC.2015.12.

[7] X. Xia et al. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Soft. Eng.,* 44, 10, 951–976, (2018). https://doi.org/10.1109/TSE.2017.2734091.

[8] G. D. Feitelson. From Code Complexity Metrics to Program Comprehension, *Communications of the ACM*, 66 (5), 52 -61. (2023). https://doi.org/10.1145/3546576

[9] J. Pantiuchina, J. M. Lanza, G. Bavota. The (mis) perception of quality metrics. In Intern. Conf. on Software Maintenance and Evolution, 80–91, (2018). https://doi.org/10.1109/ICSME.2018.00017.

[10] S. Scalabrino et al. Automatically Assessing code understandability. IEEE Transactions on Soft. Eng., 47, 3, 595–613, (2021). https://doi.org/10.1109/TSE.2019.2901468.
[11] T. J. McCabe. A complexity measure. IEEE Transactions on Soft. Eng., 2(4), 308‑320. (1976). https://doi.org/10.1109/TSE.1976.233837.
[12] M. H. Halstead Elements of Software Science. Elsevier North‑Holland. (1977).
[13] K. Rim & Y. Choe. Scope information complexity number: A measure for cognitive complexity. Info. and Soft. Techn., 49(11‑12), 1160‑1170. (2007).
[14] R. Tiwari, S. Kaur, M. Gupta. Predicting code comprehension using neural networks. *Jr. of Systems and Soft.*, 158, 110420. (2019).
[15] K. Amandeep, D. Sharma. Machine learning approaches to predict cognitive load in software comprehension. Applied Soft Computing, 108, 107421. https://doi.org/10.3390/make7020051. (2021).
[16] N. Ali, R. Al‑Qutaish, M. Ahmad. Software complexity measurement: A review. International Journal of Advanced Comp. Sci., 11(6), 120‑134. (2020).
[17] G. Bavota. Code comprehension: A survey of cognitive models and empirical results. ACM Computing Surveys, 54, 9, 1‑39. (2022).
[18] U. Onwudebelu, O. G. Igbinosa, C. U. Ugwoke. The Use of a Collegiate Software Exhibition & Competition in Software Development Education, World Journal of Computer Application and Technology (WJCAT), USA, 1(1): 6-9, (2013). https://doi.org/10.13189/wjcat.2013.010102
[19] M. K. Aregbesola, U. Onwudebelu. Experimental Evaluation of Software Quality Management and Assurance in the Nigerian Software Industry, Intern. Jr. of Latest Techn. in Eng., Managt. & Applied Sci. (IJLTEMAS), 8, 7, 77-82, ISSN 2278-2540. (2019)
[20] M. K. Aregbesola, U. Onwudebelu. Typical Software Quality Assurance and Quality Management Issues in the Nigerian Software Industry. The National Association for Science, Humanities and Education Research (NASHER) 8th Annual National Conference. September 14th –17th, 2011, pp. 107-113. (2011).
[21] S. R. Chidamber, C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Soft. Eng.*, 20, 6, 476–493; https://doi.org/10.1109/32.295895. (1994).
[22] Politowski, C. et al. (2020) A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology,* 122; https://doi.org/10.1016/j.infsof.2020.106278.
[23] T. Sharma, D. Spinellis. A survey of code smells. *J. of Systems and Software*, 138, 158 173; https://doi.org/10.1016/j.jss.2017.12.034. (2018).
[24] Y. Gil, G. Lalouche. On the correlation between size and metric validity. *Empirical Soft. Eng.*, 22, 5, 2585–2611; https://doi.org/10.1007/s10664-017-9513-5. (2017).
[25] O. Levy, D. G. Feitelson. Understanding large-scale software systems—Structure and flows. Empirical Soft Eng., 26, 3; https://doi.org/10.1007/s10664-021-09938-8. (2021)
[26] S. Abukhalaf, M. Hamdaqa, F. Khomh. On Codex prompt engineering for OCL generation: an empirical study. In 2023 IEEE/ACM 20th Intern. Conf. on Mining Soft. Repositories (MSR). IEEE, 148–157. (2023). https://doi.org/10.1109/MSR59073.2023.00033.

[27] S. Abukhalaf, M. Hamdaqa, F. Khomh. PathOCL: path-based prompt augmentation for OCL generation with GPT-4. In Proc. of the 2024 IEEE/ACM First Intern. Conf. on AI Found. Models and Soft. Eng. 108–118. (2024). https://doi.org/10.1145/3650105.3652290.
[28] L. A. Agrawal, A. Kanade, N. Goyal, S. Lahiri, S. Rajamani. Monitor-guided decoding of code LMS with static analysis of repository context. *Advances in Neural Info. Proc. Systs*, 36, 32270–32298. (2023).
[29] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A. Saurous, Y. Kim. Grammar prompting for domain-specific language generation with large language models. Advances in Neural Info. Proc. Systs, 36, 65030–65055. (2023). https://doi.org/10.48550/arXiv.2305.19234.
[30] M. DeLorenzo et al. Make Every Move Count: LLM-based High-Quality RTL Code Generation Using MCTS. arXiv:2402.03289 [cs.LG] (2024).
[31] J. Wang. Guiding Large Language Models to Generate Computer-Parsable Content. arXiv:2404.05499 [cs.SE]. (2024).
[32] B. Athiwaratkun, et al. (2023). Multi-lingual evaluation of code generation models. https://www.amazon.science/publications/multi-lingualevaluation-of-code-generation-models
[33] Y. Wang. Cognitive informatics: The new science of information and intelligence. IGI Global. (2003).
[34] J. Sweller. Cognitive load theory and its application to computer programming. *Edu. Psy. Review*, 31(2), 261-278. (2019).