National Conference on Behavioural Economics and Intelligent Decision Systems for Climate Change and Sustainable Development
Int. J. Adv. Res. in Artificial Intelligence and Machine Learning Reviews
Volume 2, Issue 1 - 2026

Review Article

Exploring the Role of Programming and Cognitive Skills in Code Comprehension and Workload Measurement

Divjot Singh[1], Ashutosh Mishra[2], Ashutosh Aggarwal[3]

[1,2,3]Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala, Punjab, India
DOI: https://doi.org/10.24321/3117.4809.202605

I N F O

Corresponding Author:
Divjot Singh, Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala, Punjab, India
E-mail Id:
dsingh phd21@thapar.edu
How to cite this article:
Singh D, Mishra A, Aggarwal A. Exploring the Role of Programming and Cognitive Skills in Code Comprehension and Workload Measurement. *Int J Adv Res Artif Intell Mach Learn Rev* 2026; 2(1): 188-193.

Date of Submission: 2025-11-07
Date of Acceptance: 2025-11-18

A B S T R A C T

This study explores how programming and cognitive skills contribute to software com- prehension and how programmers' cognitive workload is measured. The review shows that code reading, tracing, and debugging are the most frequently studied programming skills, supported by cognitive abilities such as working memory, reasoning, and problem-solving. Coding tasks and comprehension tests are the most common evaluation methods, while advanced tools such as fMRI, EEG, and eye-tracking provide deeper insights into mental effort. Key parameters used across studies include task accuracy, completion time, and brain activity. However, research in this area still faces challenges such as small sample sizes, self-report bias, high sensor costs, and difficulty replicating real programming condi- tions. Overall, the findings highlight the need for practical and scalable methods to better understand how programmers think and manage cognitive load while working with complex code.

Keywords: Code comprehension, Cognitive skills, Coding tasks

Introduction

Software maintenance is a critical and enduring phase of the software development lifecycle, often consuming the majority of time, cost, and effort compared to initial development.[1] Code comprehension serves as the backbone of these maintenance tasks, enabling programmers to interpret complex logic, trace dependencies, and anticipate the potential impact of changes within large and often poorly documented systems.[2] Since maintenance engineers frequently deal with legacy code written by others, the process demands not only technical expertise but also the ability to reconstruct the original design intent and reasoning behind the implementation.[3] Inefficient comprehension can lead to errors, redundant work, and extended debugging cycles, which in turn increase project costs and reduce software reliability. Thus, understanding how developers read, reason about, and mentally model code is essential for improving maintenance productivity and minimising technical debt in long-term software evolution.

Code comprehension is not merely a mechanical process of reading code; it is a cognitively demanding activity that engages multiple higher-order mental functions.[4] Cognitive skills such as attention, working memory, abstraction, reasoning, and problem-solving play pivotal roles in how

programmers make sense of unfamiliar code structures. Effective comprehension requires the ability to mentally simulate code execution, integrate new information with prior knowledge, and shift between different levels of abstraction—from line-by-line syntax to overall program logic.[5] These cognitive operations help developers identify bugs, infer program behaviour, and predict outcomes of code modifications. Empirical studies have shown that programmers with stronger cognitive flexibility and memory retention exhibit faster and more accurate compre- hension performance, especially when dealing with complex or nested code segments. Moreover, cognitive overload can impair understanding and decision-making, leading to lower efficiency and increased error rates. Recognising the cognitive dimension of code comprehension not only advances our understanding of programmer behaviour but also informs the design of better tools, interfaces, and training programmes aimed at reducing mental strain and enhancing comprehension effectiveness during software maintenance.

Literature review and research questions

Recent research on code comprehension demonstrates an increasing convergence of educational, cognitive, and computational perspectives. In programming pedagogy, Denny et al.[6] and Smith et al.[7] advanced the Explain-in-Plain-English (EiPE) framework, enabling students to express code functionality in natural language while automated pipelines and large language models (LLMs) generated equivalent code for verification. These studies showed that such tasks enhance conceptual understanding and emerging prompt-crafting skills, though challenges of subjectivity and large-scale deployment persist. Oli et al.[8] further emphasised guided self- explanation through an intelligent tutoring system, revealing significant learning gains for low- prior-knowledge learners, while Stankov et al.[9] introduced CodeCPP to automate fair code- tracing assessments, though it highlighted concerns about metric validity and bias in question design.

Cognitive and neuroimaging research complements these pedagogical findings. Using fMRI, Peitek et al.[10] showed that program comprehension engages working memory, attention, and language-processing regions, indicating high cognitive load, while Peitek et al.[11] linked code complexity and vocabulary size to increased mental effort and reduced accuracy. Extending this perspective, Feitelson[12] critiqued conventional software complexity metrics, arguing for human-centred validation to better represent actual comprehension difficulty.

Recent studies also examined how human cognition aligns with computational models. Li et al.[13] found minimal correlation between human eye-tracking data and LLM attention patterns during code summarisation, suggesting fundamental differences in reasoning processes. Zhu et al.[14] demonstrated that imperceptible Unicode perturbations significantly reduce LLMs' com- prehension accuracy, exposing model fragility. In industrial contexts, Bexell et al.[15] explored how professional developers comprehend unfamiliar codebases, identifying strategies such as re- producing, localising, and simulating bugs, often influenced by cognitive load and emotional re- sponses. Collectively, these studies depict a cohesive progression in code comprehension research that bridges educational assessment, cognitive mechanisms, and the growing role of intelligent systems.

Recent studies in programming and learning sciences reveal how cognitive, emotional, and metacognitive processes jointly shape how programmers develop understanding and problem- solving skills. Lapierre et al.[16] explored the emotional and cognitive contrasts between novices and beginners, showing that fear and cognitive overload negatively affect programming performance, while greater expertise moderates mental effort more efficiently. Similarly, Paludo and Montresor (2024)[17] emphasised that integrating AI-based feedback in programming ed- ucation can foster reflective and metacognitive learning. Their Reflective AI Programming Lab encourages students to articulate reasoning to AI agents, promoting deeper problem comprehen- sion and critical reflection. Expanding on the role of metacognition, Shekh-Abed[18] analysed high-school project-based learning, finding that students' self-assessed cognitive abilities often overestimate actual performance, underscoring the need for direct and reflective assessments of metacognitive self-knowledge.

On the cognitive front, Vettori et al. (2024)[19] conducted a systematic review identifying key cognitive factors—such as working memory, attention control, and metacognitive aware- ness—that predict reading and code comprehension in complex tasks. Peitek[20] contributed a neuro-cognitive perspective, combining fMRI and eye-tracking to model top-down and bottom-up comprehension strategies and to visualise brain activation during code understanding. Similarly,

Yeh et al.[21] employed EEG to detect differences in alpha and theta brainwave patterns during program comprehension, demonstrating that higher theta and alpha magnitudes correspond to increased cognitive load. At a software-engineering level, Chentouf[22] proposed a cognitive model-based learning tool for teaching maintenance project staffing decisions through problem- based and competition-based learning, blending technical and non-technical skill development.

Further, Arasteh et al.[23] approached comprehension computationally, using hybrid grey wolf and genetic algorithms to cluster software modules, facilitating structural un-

ISSN: 2395-3802
DOI: https://doi.org/10.24321/3117.4809.202605

derstanding in maintenance contexts. These algorithmic methods reduce coupling and enhance cohesion be- tween modules, ultimately improving code comprehensibility. Collectively, these works bridge metacognition, cognitive measurement, and computational modelling, illustrating how program- ming comprehension involves not just code reasoning but also awareness, reflection, and emo- tional regulation. Together, they point toward an interdisciplinary paradigm in programming education—one that aligns cognitive science, AI, and engineering to better understand how programmers learn, think, and evolve.

In conclusion, the reviewed studies collectively emphasise that effective code comprehension relies on a balanced interplay between programming proficiency and cognitive skill development. Programming skills such as code tracing, debugging, and problem-solving provide the structural foundation for understanding software, while cognitive abilities govern how effectively program- mers interpret and reason about code. Research across educational, neuro-cognitive, and com- putational domains consistently shows that stronger cognitive control and reflective awareness enable programmers to manage complexity, reduce cognitive overload, and adapt strategies to unfamiliar problems. Thus, fostering cognitive skills alongside technical instruction is crucial for nurturing deeper understanding rather than surface-level code familiarity. As programming increasingly integrates with intelligent systems and collaborative tools, the cultivation of higher- order cognitive abilities will remain central to developing adaptive, analytical, and self-regulated programmers capable of thriving in complex software environments.

Based on the conducted LR, some research questions have been defined below:

1. RQ1. Which programming and cognitive skills are predominantly examined in contemporary research on software comprehension?
2. RQ2. What different measurement methods have been adopted to measure the cognitive work- load?
3. RQ3. What salient parameters are commonly observed in the assessment of programmers' cog- nitive workload?
4. RQ4. What challenges do researchers encounter in measuring the cognitive workload of pro- grammers?

Results and Discussion

The answers to posed research questions have been discussed below.

1. RQ1. Which programming and cognitive skills are predominantly examined in contemporary research on software comprehension? The analysis of the reviewed studies

Table 1.Programming and cognitive skills widely used by researchers

Skills	Count
Code reading/tracing	15
Code explanation	6
Code debugging	4
Working memory	6
Critical thinking	2
Reasoning	4
Problem solving	3
Decision making	1

shows that code reading/tracing are the most frequently examined programming skills in software comprehension research as shown in Table 1. Most studies focus on how programmers interpret and mentally follow the logic of given code segments to understand their functionality. This is expected since code reading forms the foundation of comprehension and further programming activities such as debugging and maintenance. Code explanation and debugging are also widely studied, as they help in understanding not only what the code does but also how programmers reason about it or fix logical errors.

From the cognitive perspective, working memory is the most commonly discussed factor influencing code comprehension. It helps programmers retain and manipulate different parts of code while reasoning about control flow and logic. Reasoning and problem-solving are also essential cognitive processes, as they support the ability to link abstract program structures with functional outcomes. A smaller number of studies address critical thinking and decision-making, particularly in real-world or project-based programming contexts. Overall, the findings suggest that effective code comprehension depends not only on programming proficiency but also on cognitive abilities such as reasoning, memory, abstraction, and reflection.

2. RQ2. What different measurement methods have been adopted to measure the cognitive workload?

The results indicate that coding tests and comprehension tasks are the most commonly used methods to measure programmers' cognitive workload as shown in Figure 1. These are practical, performance-based tools that allow researchers to assess understanding through accuracy, time taken, and error rate. Surveys and self-report questionnaires are also widely used to capture subjective perceptions of mental effort and task difficulty.

ISSN: 2395-3802
DOI: https://doi.org/10.24321/3117.4809.202605

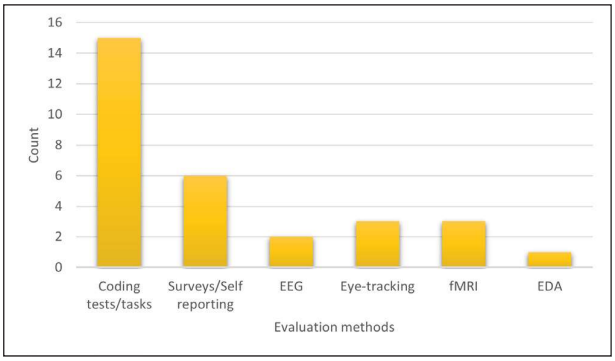

Figure 1.Distribution of evaluation methods used to measure programmers' cognitive workload

In addition to behavioural and self-reported measures, several studies have used physiological tools to record cognitive workload more objectively. Functional magnetic resonance imaging (fMRI), electroencephalography (EEG), and eye-tracking have been applied to measure brain activity and attention while participants read or debug code. A smaller number of studies use electrodermal activity (EDA) to capture emotional or stress responses during programming tasks. Together, these findings show that researchers are combining both traditional behavioral assessments and advanced physiological methods to gain a deeper understanding of mental effort in programming activities.

3. RQ3. What salient parameters are commonly observed in the assessment of programmers' cognitive workload?

The most frequently used parameters across studies are test accuracy and time taken to com- plete programming tasks Figure 2. These two behavioural measures provide simple yet powerful indicators of comprehension performance and efficiency. Accuracy reflects how well a partici- pant understands or debugs a piece of code, while task completion time gives an idea of how much cognitive effort was required to reach that understanding. More advanced studies have

introduced physiological and neural parameters such as brain activation levels, pupil dilation, and eye fixation patterns. These indicators help in identifying variations in attention, focus, and mental workload. Some studies also measure skin response as an indicator of emotional or stress- related reactions during programming. Overall, the parameters observed across studies combine performance, physiological, and emotional aspects, providing a comprehensive understanding of cognitive workload in programming contexts.

4. RQ4. What challenges do researchers encounter in measuring the cognitive work- load of programmers?

Although different tools and techniques have been developed to assess cognitive workload, researchers still face several challenges in this area. One of the most common issues is the small sample size in experimental studies, especially those using sensors such as fMRI or EEG. These tools are expensive and time-consuming, which limits the number of participants that can be studied and reduces the generalisability of results.

Another challenge is the bias that often occurs in self-reported data. Participants may over- estimate or underestimate their perceived mental effort, which affects the accuracy of subjective workload measures. The high cost and technical complexity of using advanced sensors also cre- ate practical difficulties, particularly in classroom or natural programming settings. Finally, programming tasks themselves add complexity to the measurement process, as experimental code snippets are usually simplified and may not represent real-world programming challenges. These limitations highlight the need for improved, affordable, and context-aware methods that can accurately measure cognitive workload in realistic programming environments.

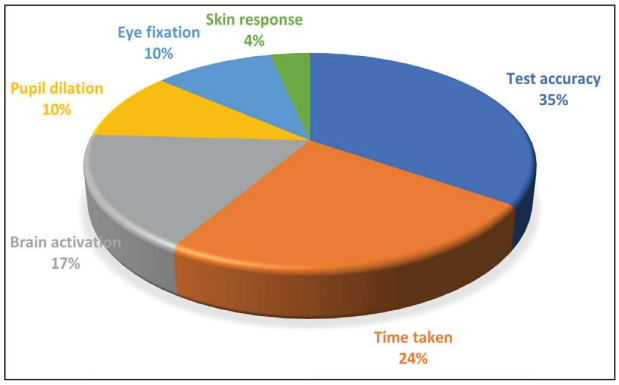

Figure 2.Distribution of evaluation parameters

Conclusion

In summary, the analysis shows that most studies on software comprehension focus on code reading, tracing, and debugging as the main programming skills, supported by cognitive abilities such as working memory, reasoning, and problem-solving. Coding tasks remain the most common approach to assess comprehension, while advanced tools such as fMRI, EEG, and eye-tracking are increasingly being used to capture deeper insights into mental effort. Accuracy and task completion time are the most frequently used parameters for evaluating cognitive workload. However, challenges such as small sample sizes, bias in self-reporting, high sensor costs, and the difficulty of simulating real-world programming tasks continue to limit research outcomes. Overall, the findings suggest that both programming and cognitive skills play a vital role in code comprehension, and future studies should aim to develop more realistic and scalable methods for measuring programmers' cognitive workload effectively.

ISSN: 2395-3802
DOI: https://doi.org/10.24321/3117.4809.202605

References

1. Norman F. Schneidewind. The state of software maintenance. IEEE Transactions on Software Engineering, (3):303–310, 2006.
2. Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 175–186. IEEE, 2017.
3. Norman Peitek, Sven Apel, Chris Parnin, Andr´e Brechmann, and Janet Siegmund. Pro- gram comprehension and code complexity metrics: A replication package of an fmri study. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 168–169. IEEE, 2021.
4. Noa Ragonis and Ronit Shmallo. The application of higher-order cognitive thinking skills to promote students' understanding of the use of static in object-oriented programming. Informatics in Education, 21(2):331–352, 2022.
5. Wei Li, Ji-Yi Huang, Cheng-Ye Liu, Judy CR Tseng, and Shu-Pan Wang. A study on the relationship between student'learning engagements and higher-order thinking skills in programming learning. Thinking Skills and Creativity, 49:101369, 2023.
6. Paul Denny, David H. Smith IV, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. Explaining code with a purpose: An integrated approach for developing code comprehension and prompting skills. In Proceedings of the 2024 ACM Conference on In- novation and Technology in Computer Science Education (ITiCSE 2024), pages 283–289, Milan, Italy, July 2024. ACM.
7. David H. Smith IV, Paul Denny, and Max Fowler. Prompting for comprehension: Exploring the intersection of explain in plain english questions and prompt writing. In Proceedings of the Eleventh ACM Conference on Learning at Scale (L@S 2024), pages 1–12, Atlanta, GA, USA, July 2024. ACM.
8. Priti Oli, Rabin Banjade, Arun Balajiee Lekshmi Narayanan, Peter Brusilovsky, and Vasile Rus. Exploring the effectiveness of reading vs. tutoring for enhancing code comprehension for novices. In Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing (SAC 2024), pages 1–10, Avila, Spain, April 2024. ACM.
9. Emil Stankov, Mile Jovanov, and Ana Madevska Bogdanova. Smart generation of code tracing questions for assessment in introductory programming. Computer Applications in Engineering Education, 31(1):5–25, 2023.
10. Norman Peitek, Janet Siegmund, Sven Apel, Christian K¨astner, Chris Parnin, Anja Beth- mann, Thomas Leich, Gunter Saake, and Andr´e Brechmann. A look into programmers' heads. IEEE Transactions on Software Engineering, 46(4):442–462, 2020.
11. Norman Peitek, Sven Apel, Chris Parnin, Andr´e Brechmann, and Janet Siegmund. Pro- gram comprehension and code complexity metrics: An fmri study. In Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE 2021), pages 1–13. IEEE, 2021.
12. Dror G. Feitelson. From code complexity metrics to program comprehension. Communica- tions of the ACM, 66(5):52–59, 2023.
13. Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In Proceedings of the 32nd IEEE/ACM International Con- ference on Program Comprehension (ICPC 2024), pages 1–5, Lisbon, Portugal, April 2024. ACM.
14. Bangshuo Zhu, Jiawen Wen, and Huaming Chen. What you see is not always what you get: An empirical study of code comprehension by large language models. arXiv preprint, 2025.
15. Andreas Bexell, Emma S¨oderberg, Christofer Rydenf¨alt, and Sigrid Eldh. How do developers approach their first bug in an unfamiliar code base? an exploratory study of large program comprehension. In Proceedings of the Psychology of Programming Interest Group Workshop (PPIG 2024), pages 174–185, Lund, Sweden, 2024. PPIG. Supported by Ericsson AB, the Swedish Foundation for Strategic Research (grant FFL18-0231), and the Wallenberg AI, Autonomous Systems and Software Program (WASP).
16. Hugo G. Lapierre, Patrick Charland, and Pierre-Majorique L´eger. Looking "under the hood" of learning computer programming: the emotional and cognitive differences between novices and beginners. Computer Science Education, 34(3):331–352, 2024.
17. Giulia Paludo and Alberto Montresor. Fostering metacognitive skills in programming: Leveraging ai to reflect on code. In Proceedings of the 2nd International Workshop on Artificial Intelligence Systems in Education, Bolzano, Italy, November 2024. University of Bolzano. © 2024 The Authors. Licensed under CC BY 4.0.
18. Aziz Shekh-Abed. Metacognitive self-knowledge and cognitive skills in project-based learn- ing of high school electronics students. European Journal of Engineering Education, 50(1):214–229, 2025.
19. Giulia Vettori, Laura Casado Ledesma, Sara Tesone, and Chiara Tarchi. Key language, cognitive and higher-order skills for l2 reading comprehension of expository texts in english as foreign language students: a systematic review. Reading and Writing, 37:2481–2519, 2024.
20. Norman Peitek. A neuro-cognitive perspective of

ISSN: 2395-3802
DOI: https://doi.org/10.24321/3117.4809.202605

program comprehension. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18 Companion), page 4, Gothenburg, Sweden, 2018. ACM.

21. Martin K.-C. Yeh, Yu Yan, Dan Gopstein, and Yanyan Zhuang. Detecting and comparing brain activity in short program comprehension using eeg. In Proceedings of the IEEE Fron- tiers in Education Conference (FIE). IEEE, 2017. Supported by NSF Grant No. 1444827.
22. Zohair Chentouf. A cognitive system to teach software maintenance project staffing. TEM Journal, 6(4):699–706, 2017.
23. Bahman Arasteh, Mohammad Abdi, and Asgarali Bouyer. Program source code comprehen- sion by module clustering using combination of discretized gray wolf and genetic algorithms. Advances in Engineering Software, 173:103252, 2022.

ISSN: 2395-3802
DOI: https://doi.org/10.24321/3117.4809.202605