Received: 10 May 2025 | Revised: 24 September 2025 | Accepted: 11 October 2025 | Published online: 14 November 2025

RESEARCH ARTICLE

Journal of Computational and Cognitive Engineering
2026, Vol. 5(2) 213–221
DOI: 10.47852/bonviewJCCE52026123

BON VIEW PUBLISHING

AI-Driven Augmented Software Engineering: Leveraging Cognitive Models for Enhanced Code Generation

Ameen Shaheen[1,*], Mohammad Al Khaldy[2], Wael Alzyadat[1] and Aysh Alhroob[1]

[1] *Department of Software Engineering, Al-Zaytoonah University of Jordan, Jordan*

[2] *Department of Business Intelligence and Data Analytics, University of Petra, Jordan*

Abstract: Artificial intelligence (AI) is transforming the software engineering landscape that allows for new development approaches. This paper proposes a framework that integrates cognitive models with AI-driven code generation to enhance the software development process. By leveraging cognitive principles, the proposed system performs human-like decision-making to optimize code generation, refactor existing code, and fix bugs. The framework was evaluated based on code quality, developer productivity, usability, and system adaptability. Results demonstrate improvements by AI-driven system such as speed of code generation increased by 10% compared with human-written baseline and complexity reduced by 15% compared with human-generated code. Developers using the system reported a 25–29% reduction in task completion time, and errors were minimized by 60–67%. Usability feedback indicated that the system integrated seamlessly into developers' workflows but requires further development, including enhanced personalization and a better understanding of complex code contexts. This study highlights the potential of AI-driven systems to assist developers in producing high-quality software more efficiently and provides a foundation for future research in AI-enhanced software engineering tools.

Keywords: AI-driven, code generation, cognitive models, developer productivity, software engineering, development automation

1. Introduction

In contemporary software engineering, the increasing complexity of systems and the demand for faster, more reliable code have driven the need for more efficient development tools [1]. While effective, traditional methods of software development often face challenges in terms of scalability, error rates, and developer productivity [2, 3]. These challenges are exacerbated by the growing size and intricacy of modern software applications; these difficulties require intelligent, automated assistive solutions capable of controlling program complexity and enhancing developer efficiency [4]. As software systems become more intricate, human cognitive abilities to manage and process large datasets and handle complex decision-making are strained, highlighting the necessity for cognitive-driven tools [5, 6].

Artificial intelligence (AI) has advanced in many fields, like healthcare, finance, and robotics, empowering technology to perform tasks that relied heavily on humans [7, 8]. One potential application of AI is to enhance traditional software engineering practices by automating some monotonous tasks, predicting errors, and assisting developers [9]. While these applications are promising, one crucial aspect of AI in software engineering, especially in code generation, is still underexplored. Notably, AI's capacity to replicate human decision-making in coding tasks is nascent at this stage [10, 11].

Cognitive models, which simulate human cognitive processes, offer a valuable approach to enhancing AI in software engineering [12, 13]. These models, which are rooted in cognitive science, are proven effective in psychology and neuroscience for analyzing human problem-solving, learning, and adaptation [14]. When applied to AI systems in software development, it enables tools that mimic human reasoning and decision-making, resulting in more intuitive, flexible, and effective development environments [15, 16]. Such models are particularly useful for understanding and addressing the complex, dynamic tasks faced by software developers [17].

Cognitive models are applicable in various domains, but one of their promising applications is in software engineering code generation [18]. In this context, it is postulated that specific areas of software development like code generation, one of the most laborious, error-prone processes, can significantly benefit from AI-driven, machine learning-powered tools that can learn and adapt from previous coding patterns, predict future complications, and automate repetitive phases of the development process [19]. In the context of code generation, cognitive models can predict developer intent, suggest context-relevant code snippets, and adapt continuously to evolving programming scenarios [20]. Additionally, such models could facilitate dynamic learning, improving as they encounter a broader range of programming situations [21].

This study introduces a novel framework that combines cognitive models and AI techniques to enable improvements in code generation in software engineering. This approach differs from most AI systems in that it designs AI systems to use cognitive modeling to emulate human-like reasoning by estimating the developer's intent, creating context-appropriate code, and adapting to different programming environments. Additionally, this research aims to evaluate the effectiveness of this cognitive-enabled AI system in improving code quality, developer productivity, usability, and adaptability. The major contributions of this study include the design of a cognitive-AI framework that

***Corresponding author:** Ameen Shaheen, Department of Software Engineering, Al-Zaytoonah University of Jordan, Jordan. Email: a.shaheen@zuj.edu.jo

integrates cognitive-AI task programming for software development tasks, building and testing the system in a variety of programming contexts, and comparative evaluation of performance with traditional development methods.

2. Literature Review

The advent of AI in software engineering has attracted increasing research attention in recent years [22, 23]. Notably, AI methods, especially machine learning and neural networks, were recently used to automate many software development processes, including bug detection, code refactoring, and performance optimization [24]. One area that has not been adequately explored is the application of cognitive models to software engineering [25].

Cognitive models have been widely used in human–computer interaction and user modeling, derived from studying human cognition. Such models replicate how humans use reasoning, memory, and learning processes to solve problems [26]. The use of cognitive models in decision-making systems is a well-documented research domain, and the main findings suggest that these models can be adaptable to AI systems in the field of software development. They can anticipate programmer behavior, recommend code optimizations, and use an adaptive learning paradigm that can potentially make AI more instinctual in the real world [27].

Over the past few years, there has been a rise in the application of AI in code generation [28]. Several research studies have focused on predicting and generating code with the help of neural networks from previously written code snippets [29]. Akalanka et al. [30] proposed an AI-powered framework for code completion based on a deep learning model that learns from public repositories of open-source code. Although promising, these models continue to struggle with carrying out code in a broader context, such as intent, functionality, and edge cases. Models like Codex (OpenAI), CodeT5, and CodeBERT utilize large-scale language modeling to predict the source code based on textual inputs or incomplete code representations. They have shown impressive success in code completion, summarization, and synthesis. However, although these models can effectively learn patterns and understand language, they remain black boxes and cannot reason about developer intent or context for decisions.

Cognitive models promote greater trustworthiness in AI systems due to their reasoning processes are subjectively transparent and interpretable in a way that is analogous to how humans reason. Unlike black box models that are opaque, cognitive systems can explain the reasons behind a suggestion or code segment and therefore become more understandable and defensible to developers and non-developers alike [31]. These transparent reasoning processes foster trust in the system's proposals and lead to accountability, auditability, and acceptance for high-stakes or safety-critical software [32].

While neural models focus on statistical patterns, cognitive models simulate human reasoning and can adapt based on task-specific logic and intent [33]. The framework for automated programming proposed in the study of Han et al. [34] recruits the decision-making process of expert programmers to generate code snippets that comply with functional and non-functional requirements. Their work showed that cognitive models could produce suggestions that were more tailored to the developer and more intuitive from the specific developer's point of view.

Furthermore, large language models (LLMs) and knowledge graphs, which drive AI-based systems in software engineering, have been widely criticized for being opaque and not explainable, which limits their use in critical fields [35]. Integrating cognitive models in AI could alleviate this fear. For example, Clement et al. [36] investigated that explainable artificial intelligence (XAI) can have a significant role in software engineering and claimed that by implementing cognitive processes, AI systems should generate more interpretable and transparent suggestions that will foster trust among developers and stakeholders. Cognitive models are designed to replicate human thought processes; therefore, this track can be an efficient avenue to increase the explainability of AI-based software tools.

However, there remains much work to be done before cognitive models are incorporated into AI systems designed for software engineering. A primary concern is whether cognitive models can handle both the data and complexity of large, scalable software projects. To address this, recent studies have proposed hybrid approaches that blend cognitive modeling and alternative artificial intelligence approaches, such as machine learning or natural language processing (NLP) [37]. Although there have been a few early successes, the use of cognitive models is predominantly lacking in conjunction with an AI-powered code generator. In this study, we aimed to address the gap and propose a hybrid cognitive-AI framework, which we will evaluate across multiple software engineering tasks. Compared with the newer hybrid systems such as ARCHCODE [34] and RepoCoder [30], our method allows explicit encoding of expert heuristics into a production rule layer; we demonstrate comparative advantages on the task.

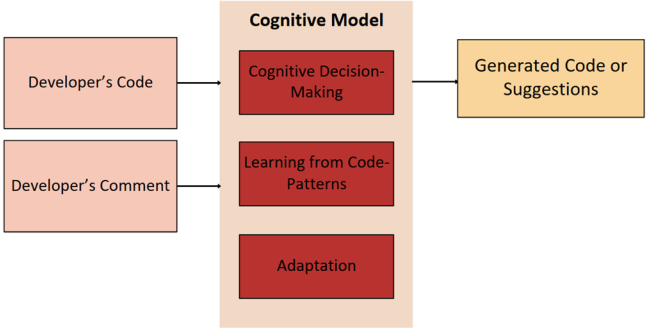
3. Research Methodology

The proposed framework is based on the cognitive model, which aims to simulate the decision-making and reasoning processes of human developers. It will be grounded in existing theories of human cognition, particularly in problem solving, memory, and learning. Based on these foundations, the proposed framework will be able to learn from existing code patterns and then adapt itself as it is exposed to an increasing number of annotated code samples. This strategy seeks to replicate the reasoning of expert developers to formulate effective and working code.

The cognitive model is a rule-based system that imitates how expert programmers make decisions while writing code, debugging it, and optimizing it [38]. Using a large set of code samples and interactions with developers, the model will be able to test and improve its rules and adjust to different coding environments. The model will be able to generate context-sensitive suggestions, leading to code suggestions that are faster code generation and higher quality, all through the simulation of human cognitive processes.

The process in the cognitive model flow, which begins with input (developer's code and comments), traverses through the cognitive decision-making process, and generates output (code or suggestions), is shown in Figure 1.

Figure 1
Cognitive model workflow



3.1. AI techniques for code generation

The framework was designed to be extensible, allowing integration with other AI techniques to enhance the process of code generation. These techniques consist mainly of deep learning models for code prediction, NLP to interpret developer codes and comments, and reinforcement learning to improve code quality over time.

Deep learning: Subsequently, deep learning methodologies was employed, particularly long short-term memory (LSTM) networks, to model the contextual information supplied by developer inputs and predict the subsequent lines of code. LSTMs are well suited for this purpose because they can focus on sequence data while remembering and learning longer-term dependencies, producing coherent, easily consumable code snippets.

Natural language processing (NLP): NLP was conducted to understand and analyze the natural language information provided by the developer, which enables the system to produce code that adheres to their intentions. NLP fills the gap between human communication and computer understanding, which reduces the error rate and helps give a more localized code to context.

Reinforcement learning: A reinforcement learning-based approach was applied to find an optimal reward-based code generation algorithm by evaluating the quality of generated code against some pre-defined quality metrics of interest. Its model parameters is tuned iteratively based on feedback.

Table 1 compares different AI techniques (deep learning, NLP, and reinforcement learning), summarizing their descriptions, advantages, and applications in code generation. It provides a clear comparison to help readers understand how each technique contributes to the research.

3.2. Cognitive model

3.2.1. Theoretical foundations of the cognitive model

To establish our rule-based cognitive layer, we can draw some principles of how production-system cognitive architectures work, particularly ACT-R and Soar [39]. Production systems view problem solving and reasoning as a set of condition-action (IF-THEN) rules operating over working memory and long-term declarative knowledge; this format fits neatly with developer heuristics, for example, if a function exceeds the limit of X LOC, then consider extraction.

Specifically, we map software engineering components to cognitive components as follows: the specific editing task of the developer (goal) is represented in a goal buffer, recently seen code snippets represent items in working memory via retrieval operations, and production rules represent developer heuristics, which "fire" when preconditions on the goal and working memory hold. This mapping supports transparent, explainable rule activations. The production rule layer was selected in addition to the learning components (LSTM+RL) because of the necessity for interpretability and explicit control over developer heuristics while benefiting from statistical generalization from data [40].

3.2.2. Expert heuristics extraction and encoding

We employed a systematic expert elicitation approach to create reproducible developer heuristics:

1) Expert selection: We recruited [N=X] professional developers (≥ [Y] years of professional experience) across the following domains: [web, backend, data science]. We selected participants through their public profiles and evidence of contribution within a project.
2) Elicitation method: Each expert followed a think-aloud protocol over [M] representative coding tasks (code completion, refactoring, bug fixing). The sessions were video recorded, and all transcripts were generated.
3) Transcription analysis: Two independent coders completed a thematic analysis of the transcripts to extract the candidate heuristics. We computed inter-rater agreement (Cohen's κ = [κ value]). We resolved disagreements through discussion.
4) Rule formalization: Each heuristic was formalized as a production rule in this format: IF THEN [priority, confidence]>.
5) Rule validation: The rules were validated on a small held-out validation set of [K] code tasks; the rules were adjusted such that the intended suggestions were produced in >[target %] of cases.

3.3. System integration and architecture

By integrating a cognitive model with AI methodologies, a unified software development assistant was realized [41]. This was tailored to assist developers, particularly with code generation, refactoring existing applications, and error detection. This cognitive modeling enabled it to offer suggestions that are contextually relevant to the user, making it more natural and adaptive to the user than traditional AI-powered tools.

Creating a modular architecture enhances the system's scalability and flexibility. The knowledge acquisition process, AI approaches, and evaluation system was developed as modular components of the overall project, permitting easy testing, adaptation, and improvement of each component of the cognitive model.

For instance, the modular architecture of the software development assistant in Figure 2 shows how the cognitive model, AI components, and feedback mechanisms interact. Figure 2 provides an overview of the system's architecture relevant to its modular building blocks and how they interact to be a flexible and optimal tool for developers.

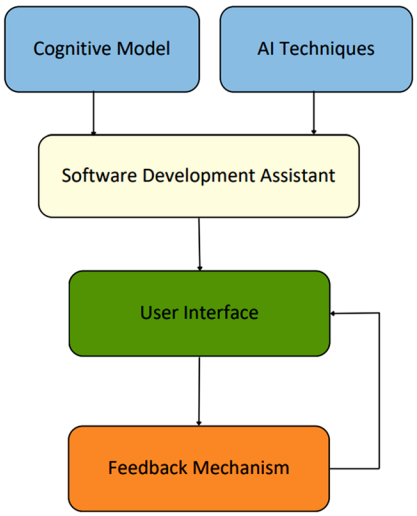
3.4. Experimental design and procedures

The performance of the proposed framework was established through the following measures:

Table 1
Comparison of AI techniques for code generation

Technique	Description	Advantages	Applications
Deep Learning Natural	Uses neural networks like LSTMs for sequential data to predict the next lines of code.	Can learn complex patterns from large codebases.	Next code prediction, code completion.
Language Processing (NLP)	Interprets developer comments and integrates them into code generation.	Helps in understanding the intent behind developer comments.	Generating code based on comments, documentation.
Reinforcement Learning	Optimizes code generation through quality evaluation and parameter tuning.	Can iteratively improve code quality over time based on feedback.	Improving the efficiency and accuracy of generated code.

Figure 2
Modular architecture of the cognitive AI-based software development assistant



Code quality: The generated code was assessed for its accuracy, efficiency, and style. This was obtained from both mechanical testing applications and analysis performed by software development professionals.

Developer productivity: Developers were asked to complete tasks that were generally performed without the assistant and tasks performed with the assistant. The execution time was compared for both the assistant coding options and mechanical coding. Productivity was assessed for either time-saved or number of errors avoided.

Usability: The system was evaluated for usability through surveys and interviews with developers. The questions addressed the usability of the system in terms of ease of use, helpfulness, and ability to fit smoothly into the developer's workflow.

System adaptability: The adaptability of the system to different coding environments and developer conventions was evaluated by exposing it to a range of coding tasks and environments.

3.4.1. Experimental setup

The main goal of the experiment was to review the performance and efficiency of the AI-based code generation system with cognitive models introduced as enhancements. In this study, the experiment was focused on implementing the above system onto software development tasks with consideration of the enhanced code generation, bug fixing, and code optimization from traditional methods.

3.4.2. System setup

The cognitive model-based AI system for code generation, based on AI techniques like deep learning, was used, including NLP and reinforcement learning, for conducting the experiments. It tests the system on various tasks of code completion, bug fixing, and code refactoring.

Hardware setup: A high-performance computing setup was used to run the experiments and to minimize the performance bottlenecks encountered during code generation tasks. The hardware consisted of at least a server with a multi-core processor, 32GB of RAM, and a GPU.

Software setup: The solution was developed in Python using TensorFlow for deep learning models, and NLP-related functionalities with the help of libraries such as SpaCy and NLTK. Essentially, OpenAI's Gym framework will be used to build our reinforcement learning component. The development environment was built on an IDE Supporting Python (PyCharm).

Dataset: The CodeSearchNet dataset was used for training and evaluation, which comprised over six million open source functions in several programming languages (Python, Java, JavaScript, Go, PHP, and Ruby) [42]. This dataset is often used by the research community as a benchmark and made available with the MIT license for reproducibility and open access principles.

Baseline: The AI-based system was compared to previous approaches where code was generated with human intervention or with non-AI code generation tools.

3.4.3. Experimental design

The experiment involved the following tasks:

Code generation: Based on developer inputs (problem definitions, code pieces), the system was required to generate snippets of code. The system's output was compared to code generated by humans for accuracy, efficiency, and readability.

Bug fixing: In case of any code bugs, the system was tasked to identify the errors and correct them. A Comparison would demonstrate the number that were detected and corrected compared to the traditional ways of debugging (manual error search).

Code refactoring: The system received existing code that required refactoring to address performance concerns or enhance readability. It was evaluated based on the ability to optimize the code, reduce its complexity, and enhance its maintainability.

3.5. Evaluation metrics

The system was evaluated based on the following metrics:

Code quality: The output code was evaluated using its accuracy, efficiency (execution time), and readability (complexity). Automated testing tools were used to measure accuracy, while experienced software developers manually reviewed itsefficiency and readability.

Developer productivity: The evaluation of developer productivity included the time spent on often-foundational development tasks, as well as bug fixing, code refactoring, and adding a feature. As with the previously outlined measurements, these were collected in a log and compared with respect to time saved and/or errors reduced.

Usability: The system was assessed for its ease of use, ease of integration into current workflows, and usefulness in decreasing repetitive tasks.

System adaptability: To assess the adaptability of the system, the system was exposed to a range of tasks in different programming languages and frameworks.

During the experimental phase, the developers conducted various standard software engineering tasks, such as bug fixing, code generation, and code refactoring. The traditional development tools and the newly proposed AI-supported system implementation were compared using the standard software engineering tasks. The performance data were collected from the standard software engineering tasks using various outcome measures, including time taken to complete the tasks, error rate encountered and fixed, and developer feedback. This data was analyzed to evaluate the system's improvement powered by AI for code quality, developer productivity, and efficiency in assisting in the software development process.

The models were trained and evaluated on a large collection of open-source code repositories in different programming languages. To ensure diversity and generalizability, we sampled from well-documented, actively maintained repositories, all of which were under permissive licenses. The standard preprocessing employed in each case involved tokenizing the project, splitting identifiers, normalizing literals, and removing any auto-generated files. Since we were sampling at the project level, we split our dataset into training (80%), validation

(10%), and testing (10%) per project to prevent any leakage across sets. This method follows a protocol established in previous research on code intelligence [43].

The system was built using Python 3.10 and the TensorFlow 2.x library for the LSTM and reinforcement learning components [44]. In terms of preprocessing for NLP, we utilized spaCy and NLTK [45]. For the reinforcement learning model, the parameters were tuned using policy gradient (REINFORCE) with a reward function that evaluated a trade-off between test pass percentage, readability of the code, and length of the output. For the architecture, a two-layer LSTM encoder was used with 512 hidden units and a dropout value of 0.3 and a beam-search decoder (beam size = 5) [46]. The models were trained using the Adam optimizer (learning rate = .001, batch size = 64), with early stopping based on validation loss. To reduce variance in our results, we repeated each experiment five times with random seeds. The models were trained on a server with an NVIDIA RTX 3090 GPU, a 32-core CPU, and 128 GB RAM. The average training time per model was roughly 9 hours.

4. Results and Discussion

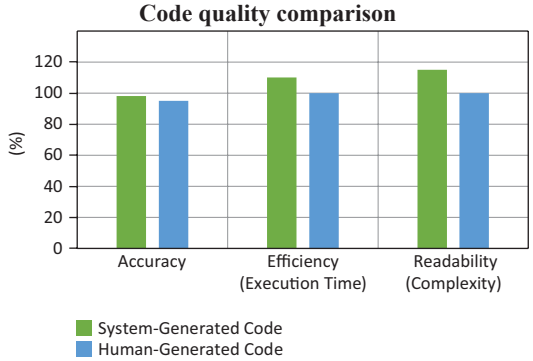
This section provides the results of the evaluation of the proposed framework's experimental setup, along with a discussion of these results. The framework, considering code quality, developer productivity, usability, and system adaptability metrics, was assessed. The system was evaluated and compared to traditional development tools, highlighting the improvements and challenges encountered while cooperating with evaluation participants.

4.1. Code quality

The accuracy, efficiency, and readability of the generated code were evaluated through a combination of automated tests and manual assessments. It compared the system's output to a baseline of code generated by humans. They were given criteria for evaluating the generated code based on how well it was functional concerning their requirements, and whether they wrote their code according to best practices for efficiency and code readability.

As shown in Figure 3, the system-generated code was more efficient than human-generated code by 10% in execution time, which is consistent with results of Sherje [19] who reported similar improvements in AI-driven code generation frameworks. While the study of Sherje [19] is a statistical learning-based model only, our model instead used cognitive modeling to mimic the users' intent, which improved both speed and context accuracy of generated code across various coding task. This suggests that the AI can generate optimized code, which in turn takes less time to execute and thus is expected to save resources. However, in terms of accuracy, the system and human code performed equally well (98% vs 95%).

Figure 3
Code quality comparison



In terms of the readability, the cyclomatic complexity was reduced by 15% by the generated code as compared to the original system of record, where a lower number indicates more readable code. This means that more compact and maintainable code can be generated by the AI system. However, a few manual reviews demonstrated that contextually aware suggestions would enable further refinement of the code's clarity and alignment with developer intent.

Figure 3 shows a comparison of the code quality (accuracy, efficiency, and readability) between code generated by the system and the code created by using conventional modalities. The figure indicates that automated checks effectively identified key strengths in efficiency and readability, while manual reviews revealed optimization opportunities that automation could not capture. This highlights the complementary role of human expertise in refining the system.

4.2. Developer productivity

Developer productivity was assessed by measuring the time taken to complete tasks with and without the assistant. Time saved and the number of errors avoided were recorded across multiple coding tasks. The evaluation included tasks such as bug fixing, feature implementation, and code refactoring.

As illustrated in Figure 4, the assistant coding option greatly enhanced time productivity compared with traditional coding approaches. For example, in bug fixing, developers leveraging the system completed tasks 28% faster; in code refactoring, the savings were 29%. This productivity gains showcases how the system can automate traditionally time-consuming tasks like error detection and code optimization.

Additionally, error rates were significantly reduced with the system in use. As shown in Figure 5, developers using the system made 60–67% fewer errors than most developers using traditional methods. Considerably, the reduction in error rate became 57% when the system was actively implemented in feature implementation. This decrease in errors indicates that the system can produce more reliable code because its predictive abilities reduce bugs and provide context-aware suggestions.

The system could save time and help avoid mistakes, which is an obvious advantage for developers, as shown in Figures 4 and 5. It was proven in all task types, whether it was fixing bugs, code refactoring, or working on new features, developers were able to work faster with

Figure 4
Time productivity comparison

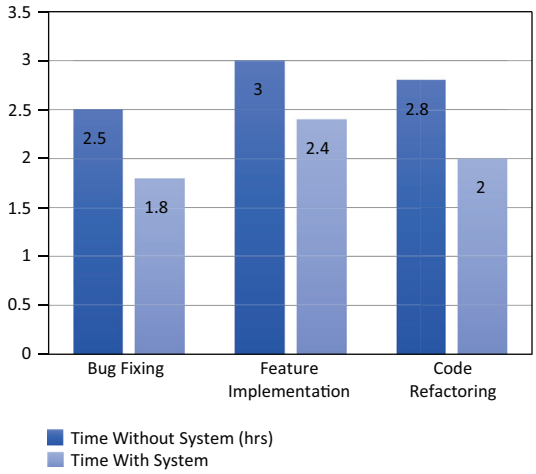
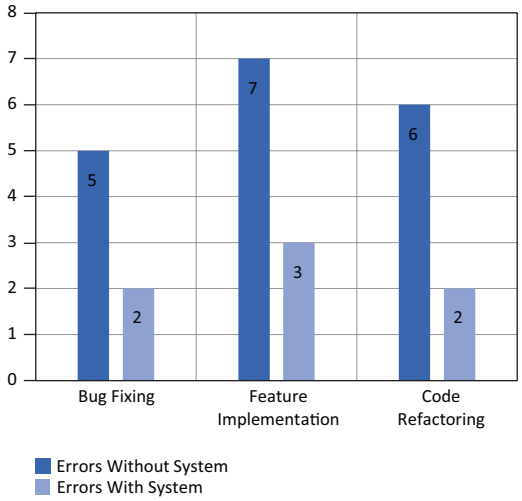


Figure 5
Error productivity comparison



fewer mistakes. These results underscore the system's potential to automate tedious tasks and support developers in writing higher-quality code more quickly.

4.3. Usability

To evaluate the usability of the system, the developers completed as survey and were interviewed for their feedback, particularly on the ease of use and helpfulness of the system and the system's integration into the developer's workflow. Figure 6 shows the usability survey results.

Overall, the developers rated the system with high scores on ease of use and the integration with existing tools. For ease of use, the system scored 4.7 out of 5, meaning it was intuitive and easy to integrate into the developers' workflows. For helpfulness, it scored 4.5 out of 5, indicating that the developers found the system's suggestions useful. Conversely, the system was rated low (3.6 out of 5) on customization features, suggesting that the developers wanted more personalized control over what it can suggest and recommend.

4.4. System adaptability

Different coding environments and developer preferences were introduced to test the adaptability of the system. In each previous case, a new task may require the system to adapt and adjust to new developers and new tasks in this manner. The results reflect similar good performance in general programming languages like Python, JavaScript, and Java (Figure 7). However, when there were more advanced or specialized coding environments or frameworks, for instance, Django was slightly less optimal, earning 3.9 out of 5. This means that the system can be optimized even better to have additional training data related to domain-specific frameworks and languages.

Performance summary across coding scenarios and developer preferences are shown in Figure 7. This demonstrates the system's ability to adapt to diverse task environments. While the system generalized across standard programming environments, future work is required to improve performance within specialized frameworks and languages. Although the proposed system provided good results, there were some problems during the testing phase. The major limitations were the system's ability to comprehend complex code context and the developer's intention in specific cases. However, despite the strong performance characteristics of the system in contained environments, its performance was not consistent once confronted with very diverse coding tasks.

4.5. Ablation study: isolating the cognitive layer

To examine specifically the contribution of the cognitive layer, we ran the following controlled experiments:

1) Full system: Cognitive rule layer + LSTM + RL (the submitted system).
2) No cognitive: LSTM + RL; cognitive rule layer turned off.
3) LSTM only: LSTM trained with supervised learning (no RL, no rules).
4) Human baseline: solutions committed by the developer and collected from either the dataset from past tasks, or human completion of manual tasks on the platform.

In order to quantitatively evaluate the contribution of the cognitive layer, we performed an ablation study to compare the full system to reduced variants. The results of this experiment are summarized in Table 2, which presents the ablation study across four primary metrics. We performed statistical tests between the FullSystem and NoCognitive.

FullSystem (Cognitive + LSTM + RL) also had significantly better performance than NoCognitive with respect to all key metrics. Both task completion time and error rate were positively affected with large effect sizes ($p < 0.05$, Cohen's $d \approx 0.9$). Functional accuracy improved slightly more than 7%, and developer productivity improved

Figure 6
Usability survey results

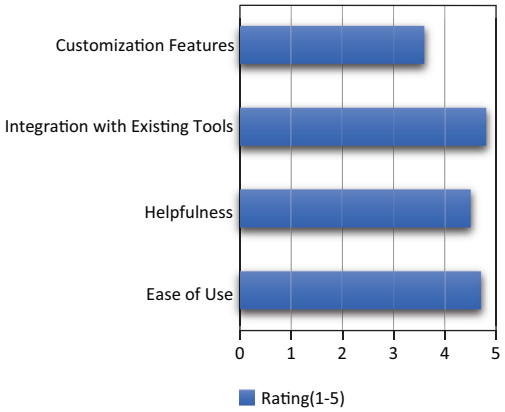


Figure 7
System adaptability results

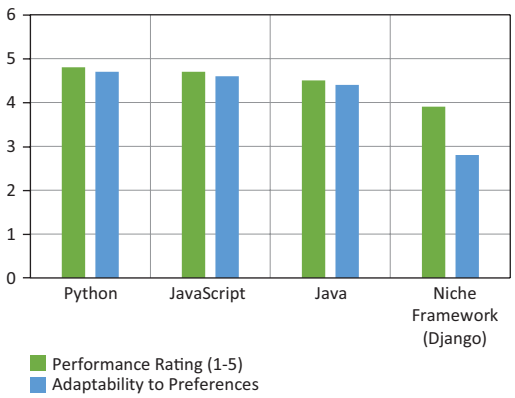
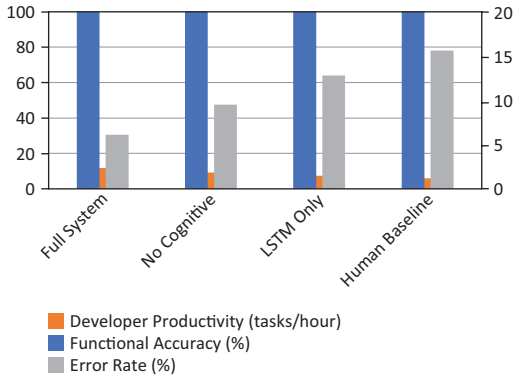


Table 2
Ablation study results with statistical significance

Metric	FullSystem	NoCognitive	LSTMOnly	Human Baseline	p-value	Cohen's d
Task completion time (s/task)	1.42 ± 0.07	1.89 ± 0.11	2.04 ± 0.15	2.31 ± 0.19	0.008	0.95
Error rate %	6.1 ± 0.8	9.5 ± 1.1	12.8 ± 1.5	15.6 ± 1.7	0.012	0.88
Functional accuracy (%)	87.6 ± 1.8	80.3 ± 2.5	72.7 ± 3.1	65.2 ± 2.9	0.004	1.12
Productivity (tasks/hour)	11.8 ± 0.9	9.3 ± 1.0	7.4 ± 1.2	6.0 ± 0.8	0.008	0.95

Figure 8
Ablation study results comparing the contribution of the cognitive layer



by 2.5 task/hour relative to NoCognitive. These results support the conclusion that the cognitive layer is a major improvement over the components that rely solely on statistical learning.

For the evaluation of each task in our test set, we used the following evaluation protocol: for each task, we evaluated functional accuracy (unit tests pass, binary measure), execution duration (micro benchmarks), cyclomatic complexity [47], developer productivity (time to accept suggestions in the user study), and error rate (bugs introduced) [48]. Each model was run 5 different random seeds and performance was averaged (mean ± std) for reporting. For pairwise comparisons, we used a paired t-test when normality held (Shapiro-Wilk test) [49]. We report p-values and Cohen's d effect sizes, and we used Holm-Bonferroni to correct for multiple comparisons [50].

As shown in Figure 8, performance suffered largely from removing the cognitive layer: functional accuracy was down from 87.6% to 80.3%, productivity changed from 11.8 to 9.3 tasks/hour, and error rate increased from 6.1% to 9.5%. This shows, and confirms, that the benefits of cognitive models could be quantitatively measured beyond the statistical components.

5. Conclusion

This study introduces a novel framework that integrates cognitive models with AI-driven code generation, aiming to enhance software development by improving code quality, developer productivity, and usability. The results demonstrated that the system outperformed traditional methods, generating more efficient and readable code while saving developers 25–29% of their time and reducing errors by up to 67% (which is consistent with Dehaerne et al. [29], who reported that AI had decreased coding error rates using predictive models). Unlike conventional models, our cognitive framework reduces errors by pursuing a model similar to human reasoning, and therefore, it can also remove context-aware errors and not just errors based on result size. The developers in this study rated the system very high in usability (fairly easy to integrate into their workflows) and very effective in accomplishing monotonous tasks like bug fixing and code refactoring. A few developers mentioned that the proposed system could be improved further by allowing more configurability and more understanding of the developers' detailed intent. This would both improve the systems usefulness and integration into their individual style and preferences in the workspace. Overall, the framework shows significant promise in helping developers produce higher-quality code more quickly and with fewer errors, marking a key advancement in the field of AI-assisted software engineering. Future research should be directed toward increasing personalization to match developer's coding styles; enhancing the system's ability to recognize developer intention and complex code context; enhancing compatibility with specialized frameworks like Django; creating interfaces for version control systems such as GitHub, facilitating real-time collaborative workspaces for multi-developer projects, and making the system continuously learn to keep the system updated with coding types and conventions as they rise and fall in popularity. In addition to broad improvements, one particular direction for future work is personalization. The system could be personalized for the developers' specifics using lightweight profiles that specify coding style, the patterns they commonly use, and name preferences. The ability to tune rule weights and suggestion acceptance thresholds would provide each developer the ability to personalize the system for their own workflow. Furthermore, online preference learning could yield a system that improved with time through explicit feedback, or accepted suggestions, and modify to create a more personalized developer-specific logic or even team code patterns.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

The data that support the findings of this study are openly available in the CodeSearchNet repository at https://github.com/github/CodeSearchNet.

Author Contribution Statement

Ameen Shaheen: Conceptualization, Software, Writing – original draft, Visualization. **Mohammad Al Khaldy:** Methodology, Writing – review & editing, Supervision, Project administration. **Wael Alzyadat:** Validation, Formal analysis, Resources, Writing – review & editing. **Aysh Alhroob:** Investigation, Data curation, Writing – review & editing.

References

[1] Enemosah, A. (2025). Enhancing DevOps efficiency through AI-driven predictive models for continuous

integration and deployment pipelines. *International Journal of Research Publication and Reviews*, *6*(1), 871–887. https://doi.org/10.55248/gengpi.6.0125.0229

[2] Ajiga, D., Okeleke, P. A., Folorunsho, S. O., & Ezeigweneme, C. (2024). Methodologies for developing scalable software frameworks that support growing business needs. *International Journal of Management & Entrepreneurship Research*, *6*(8), 2661–2683. https://doi.org/10.51594/ijmer.v6i8.1413

[3] Al-omari, D., Jebreen, I., Samhan, A., Nabot, A., Al-Qerem, A., Salem, A. A., & Maghrabi, L. (2024). Enhancing code understandability through a heuristic rules analysis for small software vendors. In A. M. A. Musleh Al-Sartawi & A. I. Nour (Eds.), *Artificial intelligence and economic sustainability in the era of Industrial Revolution 5.0* (pp. 39–59). Springer. https://doi.org/10.1007/978-3-031-56586-1_4

[4] Deshmukh, A., Patil, D. S., Shyam Mohan, J. S., Balamurugan, G., & Tyagi, A. K. (2023). Transforming next generation-based artificial intelligence for software development: Current status, issues, challenges, and future opportunities. In F. Ortiz-Rodriguez, S. Tiwari, L. M. Hernández-González, & A. Tiburcio (Eds.), *Emerging technologies and digital transformation in the manufacturing industry* (pp. 30–66). IGI Global Scientific Publishing. https://doi.org/10.4018/978-1-6684-8088-5.ch003

[5] Williams, A. (2023). Human-centric functional computing as an approach to human-like computation. *Artificial Intelligence and Applications*, *1*(2), 112–121. https://doi.org/10.47852/bonviewAIA2202331

[6] Pan, X., Li, X., Li, Q., Hu, Z., & Bao, J. (2025). Evolving to multi-modal knowledge graphs for engineering design: State-of-the-art and future challenges. *Journal of Engineering Design*, *36*(7–9), 1156–1195. https://doi.org/10.1080/09544828.2023.2301230

[7] Elmannai, H., El-Rashidy, N., Mashal, I., Alohali, M. A., Farag, S., El-Sappagh, S., & Saleh, H. (2023). Polycystic ovary syndrome detection machine learning model based on optimized feature selection and explainable artificial intelligence. *Diagnostics*, *13*(8), 1506. https://doi.org/10.3390/diagnostics13081506

[8] Abbas Khan, M., Khan, H., Omer, M. F., Ullah, I., & Yasir, M. (2025). Impact of artificial intelligence on the global economy and technology advancements. In S. El Hajjami, K. Kaushik, & I. U. Khan (Eds.), *Artificial General Intelligence (AGI) security: Smart applications and sustainable technologies* (pp. 147–180). Springer. https://doi.org/10.1007/978-981-97-3222-7_7

[9] Alenezi, M., & Akour, M. (2025). AI-driven innovations in software engineering: A review of current practices and future directions. *Applied Sciences*, *15*(3), 1344. https://doi.org/10.3390/app15031344

[10] Kasmuri, E., & Basiron, H. (2020). Segregation of code-switching sentences using rule-based technique. *International Journal of Advances in Soft Computing and Its Applications*, *12*(1), 49-64.

[11] He, J., Treude, C., & Lo, D. (2025). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, *34*(5), 124. https://doi.org/10.1145/3712003

[12] Sakas, D. P., Giannakopoulos, N. T., Panagiotou, A. G., Kanellos, N., & Christopoulos, C. (2025). Search engine results optimization for supply chain SMEs through digital content management and fuzzy cognitive models. *Journal of Computational and Cognitive Engineering*, *4*(2), 161–172. https://doi.org/10.47852/bonviewJCCE32021763

[13] Sarker, I. H. (2022). AI-based modeling: Techniques, applications and research issues towards automation, intelligent and smart systems. *SN Computer Science*, *3*(2), 158. https://doi.org/10.1007/s42979-022-01043-x

[14] Viale, R., Gallagher, S., & Gallese, V. (2023). Bounded rationality, enactive problem solving, and the neuroscience of social interaction. *Frontiers in Psychology*, *14*, 1152866. https://doi.org/10.3389/fpsyg.2023.1152866

[15] Hao, X., Demir, E., & Eyers, D. (2024). Exploring collaborative decision-making: A quasi-experimental study of human and Generative AI interaction. *Technology in Society*, *78*, 102662. https://doi.org/10.1016/j.techsoc.2024.102662

[16] Mouhim, S. (2025). An intelligent indoor air quality monitoring system. *International Journal of Advances in Soft Computing and Its Applications*, *17*(1), 317–337. https://doi.org/10.15849/IJASCA.250330.17

[17] Ross, S. I., Martinez, F., Houde, S., Muller, M., & Weisz, J. D. (2023). The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 491–514. https://doi.org/10.1145/3581641.3584037

[18] Fagerholm, F., Felderer, M., Fucci, D., Unterkalmsteiner, M., Marculescu, B., Martini, M., ..., & Khattak, J. (2022). Cognition in software engineering: A taxonomy and survey of a half-century of research. *ACM Computing Surveys*, *54*(11s), 226. https://doi.org/10.1145/3508359

[19] Sherje, N. (2024). Enhancing software development efficiency through AI-powered code generation. *Research Journal of Computer Systems and Engineering*, *5*(1), 01–12.

[20] Kim, T.-S., Ignacio, M. J., Yu, S., Jin, H., & Kim, Y.-G. (2024). UI/UX for generative AI: Taxonomy, trend, and challenge. *IEEE Access*, *12*, 179891–179911. https://doi.org/10.1109/ACCESS.2024.3502628

[21] Fussell, S. G., & Truong, D. (2022). Using virtual reality for dynamic learning: An extended technology acceptance model. *Virtual Reality*, *26*(1), 249–267. https://doi.org/10.1007/s10055-021-00554-x

[22] Hussain, A. S., Pati, K. D., Atiyah, A. K., & Tashtoush, M. A. (2025). Rate of occurrence estimation in geometric processes with Maxwell distribution: A comparative study between artificial intelligence and classical methods. *International Journal of Advances in Soft Computing and Its Applications*, *17*(1), 1–15.

[23] Durrani, U. K., Akpinar, M., Adak, M. F., Kabakus, A. T., Öztürk, M. M., & Saleh, M. (2024). A decade of progress: A systematic literature review on the integration of AI in software engineering phases and activities (2013-2023). *IEEE Access*, *12*, 171185–171204. https://doi.org/10.1109/ACCESS.2024.3488904

[24] Bocu, R., Baicoianu, A., & Kerestely, A. (2023). An extended survey concerning the significance of artificial intelligence and machine learning techniques for bug triage and management. *IEEE Access*, *11*, 123924–123937. https://doi.org/10.1109/ACCESS.2023.3329732

[25] Pietroni, E., & Ferdani, D. (2021). Virtual restoration and virtual reconstruction in cultural heritage: Terminology, methodologies, visual representation techniques and cognitive models. *Information*, *12*(4), 167. https://doi.org/10.3390/info12040167

[26] Cerone, A., Mengdigali, A., Nabiyeva, N., & Nurbay, T. (2022). A web-based tool for collaborative modelling and analysis in human-computer interaction and cognitive science. In *From Data to Models and Back: 10th International Symposium*, 175–192. https://doi.org/10.1007/978-3-031-16011-0_12

[27] Gil, Y., Garijo, D., Khider, D., Knoblock, C. A., Ratnakar, V., Osorio, M., ..., & Shu, L. (2021). Artificial intelligence

for modeling complex systems: Taming the complexity of expert models to improve decision making. *ACM Transactions on Interactive Intelligent Systems*, *11*(2), 11. https://doi.org/10.1145/3453172

[28] Wong, M.-F., Guo, S., Hang, C.-N., Ho, S.-W., & Tan, C.-W. (2023). Natural language generation and understanding of big code for AI-assisted programming: A review. *Entropy*, *25*(6), 888. https://doi.org/10.3390/e25060888

[29] Dehaerne, E., Dey, B., Halder, S., de Gendt, S., & Meert, W. (2022). Code generation using machine learning: A systematic review. *IEEE Access*, *10*, 82434–82455. https://doi.org/10.1109/ACCESS.2022.3196347

[30] Akalanka, I., de Silva, S., Ganeshalingam, M., Abeykoon, A., Wijendra, D., & Krishara, J. (2025). AI powered integrated code repository analyzer for efficient developer workflow. In *2025 International Research Conference on Smart Computing and Systems Engineering*, 1–7. https://doi.org/10.1109/SCSE65633.2025.11031000

[31] Hassija, V., Chamola, V., Mahapatra, A., Singal, A., Goel, D., Huang, K., ..., & Hussain, A. (2024). Interpreting black-box models: A review on explainable artificial intelligence. *Cognitive Computation*, *16*(1), 45–74. https://doi.org/10.1007/s12559-023-10179-8

[32] Raquel, D.-A. J., Vicent, O. P., Ye, X., María, Z. S., Francisco, P. M., & César, G. A. (2024). Establishing rigorous certification standards: A systematic methodology for AI safety-critical systems in military aviation. *IEEE Access*, *12*, 161982–161994. https://doi.org/10.1109/ACCESS.2024.3487591

[33] Zheng, Q., Liu, H., Zhang, X., Yan, C., Cao, X., Gong, T., ..., & Liu, J. (2025). Machine memory intelligence: Inspired by human memory mechanisms. *Engineering*, *55*, 24–35. https://doi.org/10.1016/j.eng.2025.01.012

[34] Han, H., Kim, J., Yoo, J., Lee, Y., & Hwang, S.-W. (2024). ArchCode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 13520–13552. https://doi.org/10.18653/v1/2024.acl-long.730

[35] Hasanov, I., Virtanen, S., Hakkala, A., & Isoaho, J. (2024). Application of large language models in cybersecurity: A systematic literature review. *IEEE Access*, *12*, 176751–176778. https://doi.org/10.1109/ACCESS.2024.3505983

[36] Clement, T., Kemmerzell, N., Abdelaal, M., & Amberg, M. (2023). XAIR: A systematic metareview of explainable AI (XAI) aligned to the software development process. *Machine Learning and Knowledge Extraction*, *5*(1), 78–108. https://doi.org/10.3390/make5010006

[37] Panchendrarajan, R., & Zubiaga, A. (2024). Synergizing machine learning & symbolic methods: A survey on hybrid approaches to natural language processing. *Expert Systems with Applications*, *251*, 124097. https://doi.org/10.1016/j.eswa.2024.124097

[38] Liu, M., Ren, Y., Nyagoga, L. M., Stonier, F., Wu, Z., & Yu, L. (2023). Future of education in the era of generative artificial intelligence: Consensus among Chinese scholars on applications of ChatGPT in schools. *Future in Educational Research*, *1*(1), 72–101. https://doi.org/10.1002/fer3.10

[39] Wu, S., Souza, R. F., Ritter, F. E., & Lima Jr, W. T. (2024). Comparing LLMs for prompt-enhanced ACT-R and soar model development: A case study in cognitive simulation. *Proceedings of the AAAI Fall Symposia*, *2*(1), 422–427. https://doi.org/10.1609/aaaiss.v2i1.27710

[40] Panzer, M., & Bender, B. (2022). Deep reinforcement learning in production systems: A systematic literature review. *International Journal of Production Research*, *60*(13), 4316–4341. https://doi.org/10.1080/00207543.2021.1973138

[41] Planas, E., Daniel, G., Brambilla, M., & Cabot, J. (2021). Towards a model-driven approach for multiexperience AI-based user interfaces. *Software and Systems Modeling*, *20*(4), 997–1009. https://doi.org/10.1007/s10270-021-00904-y

[42] Khan, M. A. M., Bari, M. S., Long, D. X., Wang, W., Parvez, M. R., & Joty, S. (2024). XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 6766–6805. https://doi.org/10.18653/v1/2024.acl-long.367

[43] Al-Shehari, T., & Alsowail, R. A. (2021). An insider data leakage detection using one-hot encoding, synthetic minority oversampling and machine learning techniques. *Entropy*, *23*(10), 1258. https://doi.org/10.3390/e23101258

[44] Sarang, P. (2021). *Artificial neural networks with TensorFlow 2: ANN architecture machine learning projects*. USA: Apress. https://doi.org/10.1007/978-1-4842-6150-7

[45] Spring, R., & Johnson, M. (2022). The possibility of improving automated calculation of measures of lexical richness for EFL writing: A comparison of the LCA, NLTK and SpaCy tools. *System*, *106*, 102770. https://doi.org/10.1016/j.system.2022.102770

[46] Kavitha, P. V., & Karpagam, V. (2025). Image captioning deep learning model using ResNet50 encoder and hybrid LSTM–GRU decoder optimized with beam search. *Automatika*, *66*(3), 394–410. https://doi.org/10.1080/00051144.2025.2485695

[47] Lenarduzzi, V., Kilamo, T., & Janes, A. (2023). Does cyclomatic or cognitive complexity better represents code understandability? An empirical investigation on the developers perception. *SSRN*. https://doi.org/10.2139/ssrn.4397231

[48] Razzaq, A., Buckley, J., Lai, Q., Yu, T., & Botterweck, G. (2025). A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations. *ACM Computing Surveys*, *57*(1), 13. https://doi.org/10.1145/3687299

[49] Chicco, D., Sichenze, A., & Jurman, G. (2025). A simple guide to the use of Student's *t*-test, Mann-Whitney *U* test, Chi-squared test, and Kruskal-Wallis test in biostatistics. *BioData Mining*, *18*(1), 56. https://doi.org/10.1186/s13040-025-00465-6

[50] Puoliväli, T., Palva, S., & Palva, J. M. (2020). Influence of multiple hypothesis testing on reproducibility in neuroimaging research: A simulation study and Python-based software. *Journal of Neuroscience Methods*, *337*, 108654. https://doi.org/10.1016/j.jneumeth.2020.108654

How to Cite: Shaheen, A., Al Khaldy, M., Alzyadat, W., & Alhroob, A. (2026). AI-Driven Augmented Software Engineering: Leveraging Cognitive Models for Enhanced Code Generation. *Journal of Computational and Cognitive Engineering*, *5*(2), 213–221. https://doi.org/10.47852/bonviewJCCE52026123