Faculty of Natural and Applied Sciences Journal of Computing and Applications
Print ISSN: 3026-8133
www.fnasjournals.com
Volume 3; Issue 1; March 2026; Page No. 61-73.
DOI: https://doi.org/10.63561/jca.v3i1.1204

A Framework for Evaluating Prompt Engineering Techniques for Code Generation in PHP Using Open-Source Large Language Models

***Alemoh, R.B., & Muhammad-Bello, B.L.**
Faculty of Computing - Nile University of Nigeria - Nigeria

[*]**Corresponding author email**: 246370012@nileuniversity.edu.ng

Abstract
As generative Artificial Intelligence (AI) systems become increasingly integrated into software development workflows in software engineering, there is a need for rigorous and reproducible evaluation of how prompt engineering techniques influence the quality, reliability, and efficiency of code generated by open-source large language models (LLMs). This paper presents a structured empirical evaluation framework to assess the effects of prompt techniques (zero-shot, few-shot, and chain-of-thought) on PHP code generation using three commonly adopted open-source models (CodeLLaMA, Mistral, and StarCoder2). A reproducible experimental pipeline was developed to execute controlled prompt–model templates across multiple coding tasks (easy, medium and difficult) in software engineering. The framework automatically measures multiple performance metrics, including functional accuracy (pass@1 and pass@10), execution time, memory usage, lines of code, cyclomatic complexity, and coding standard violations. The results indicated that the most effective prompt strategy was the use of few-shot prompting on all models and then zero-shot that demonstrated improvement with repeated sampling (pass@10). Tasks with reasoning requirements benefited from chain-of-thought prompting in some aspects, which made the code more complex and lengthier. In the analysis of the effect of model variability for the consideration of this study, two-way Analysis of Variance (ANOVA) indicated that the underlying model architecture is still a significant factor in determining the performance differences. The execution time and memory usage were similar for all the strategies in prompting, while the code quality measures indicated structural differences in the code based on the prompt design. These results indicate that prompt engineering influences observable performance trends while adhering to the model's architecture and task-level limitations, hence the need to consider the combination of prompt techniques, model capacity, task types, and evaluation methodology. THe assessment framework contributes to the development of standardized benchmarking tools in the PHP ecosystem, aside the contribution to the body of empirical research. This framework is a foundation for potential future extension to multiple metrics, reproducible assessment of open-source LLMs across programming languages, and task domains.

Keywords: Prompt Engineering, Code Generation, Large Language Models, Open-Source LLMs, Software Engineering

Introduction

Large language models (LLMs) have sparked a paradigm change from code generation support to coding agents, opening up a new development approach called "Vibe Coding" in which developers verify AI-generated implementations by looking at the results rather than line-by-line code understanding (Ge et al., 2025). Quality code can be developed from these models when prompted, which can also automate documentation, facilitate debugging, and manage tasks such as refactoring and test creation to support software engineers. Effectively using these models relies majorly on prompt engineering techniques, which involves carefully crafting prompts (inputs) to produce accurate, relevant, and high-quality results.

While basic prompting techniques are now common, the impact of advanced approaches such as few-shot learning, Chain-of-Thought prompting, role-based instructions, and contextual code injection on the quality of code generated in software engineering tasks remains underexplored (Khojah et al., 2024). Although code-generating LLMs have evolved, little empirical research has been done on how certain prompt engineering techniques work in real-world

software engineering tasks when applied explicitly to open-source LLMs. The software engineering community is fascinated by their ability to change the way software is developed, and there are many discussions about their impact (Haque, 2025). Li et al. (2024) mentioned that code generation is a potent method in modern software development, enhancing efficiency, minimizing errors, and promoting standardization and consistency. ChatGPT has shown tremendous promise in autonomous code generation lately (Li et al., 2024). Existing research focuses mainly on GPT models performance, with limited empirical evidence on how different prompting techniques influence correctness, efficiency, and maintainability of generated code across several models. Moreso, relying only on common GPT models that are around can limit perspective on costs, transparency, accuracy, memory usage and accessibility of large language models even though it still needs to be explored. This paper addresses this research gap by developing an evaluation framework for the assessment of prompt engineering approaches in code generation using open-source LLMs. The focus is the development of a structured evaluation framework for the evaluation of the effect of prompt engineering techniques (zero-shot, few-shot, and chain-of-thought) for code generation using large language models in the field of software engineering. Building on insights from existing literature, the study evaluates three open-source large language models (Code LLaMA, StarCoder, and Mistral) to assess their performance under three different prompt techniques. Each model will be tested using the selected prompt engineering techniques, applied to generate PHP code from structured textual instructions. Several evaluation metrics will be carried out to establish evidence-based best practices for prompt engineering in software engineering use cases. This paper contributes to the understanding of code generation using large language models by developing a structured evaluation framework for assessing prompt engineering techniques in software engineering. The framework is based on text-to-code generation using the PHP programming language and involves the evaluation of zero-shot, few-shot, and chain-of-thought techniques based on the chosen open-source model. It establishes evidence-based standards and best practices by assessing generated code using metrics such as accuracy, quality, execution efficiency, and memory usage. The findings aim to support software engineers and researchers in developing software that is more reliable, maintainable, and efficient through the effective use of prompt engineering techniques on open-source LLMs.

Related Works

Recent advancements have been observed in the utilization of Large Language Models (LLMs), especially within software engineering, a field experiencing extensive LLM integration. Mayer et al. (2024) conducted a comparative analysis to assess the proficiency level of five LLMs (Bard, BingChat, ChatGPT, Llama 2, and Code Llama) in generating code based on text-to-code generation tasks. The authors created problem descriptions from LeetCode to develop Python code. Later, the testing tools of LeetCode were integrated to conduct the evaluation. Based on the results, it has been identified that the models did not work in the same manner. Among the LLMs, ChatGPT surpassed the general-purpose and code-specific models like CodeLlama. In this research, the accuracy, speed, and memory usage of the LLMs were compared with human efforts.

However, the results revealed that there were consistent errors in the code, including indentation, structural, and increased error rates in response to longer prompts (Mayer et al., 2024). Prompt-driven programming has been introduced due to the increased adoption of large language models (LLMs) among software developers. In this approach, developers generate code from natural language (NL) inputs. Although, there are concerns regarding the safety and reliability of these generated codes. Tony et al. (2025) conducted a study to examine the impact of several prompting methods on the safety of code generated by LLMs. The research began with a review of existing prompting methods in relation to code generation challenges. The authors used a dataset of 150 NL security-related prompts to test different prompting techniques and their usefulness in addressing security challenges. The results demonstrated that standard prompting techniques can improve the security of generated code. Notable improvements were observed when employing the Recursive Criticism and Improvement (RCI) method. (Tony et al., 2025). Shin et al. (2023) performed an empirical assessment of two dominant two main methods, prompt engineering and fine-tuning, to automate code-related tasks in software engineering. Prompt engineering involves creating specific instructions to guide large language models (LLMs), like ChatGPT. Fine-tuning, on the other hand, means retraining models that already exist for example CodeBERT by using datasets that are specific to various fields. Three different prompting techniques were used on GPT-4 and tested with 17 fine-tuned models on tasks (code summarisation, creation, and translation). The results indicated that GPT-4, despite its flexibility, did not consistently outperform fine-tuned models. For instance, a 28.3% performance discrepancy was observed in code generation on the MBPP dataset. Subsequent research through a user study including 27 graduate students and 10 industry experts showed that GPT-4's performance significantly improved with the incorporation of human input in conversational prompting. The result showed the effectiveness of interactive prompting while presenting the limitations of fully automated prompting. Over the past

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

several years, large language models have been widely used in a variety of "Automated Software Engineering (ASE) domains, including code completion" (Wei et al., 2023). code generation (Shin et al., 2023), test case generation (Shin et al.,2024), "the generation of test oracles "(Shin, et al., 2024), code translation (Sun et al., 2022), and software correction that is automated (Li et al., 2022). Small language models (SLMs) are a more sustainable alternative as they use less computational resources while remaining effective for fundamental programming tasks. Ashraf et al. (2025) investigated the capacity of prompt engineering to enhance the energy efficiency of SLMs in code development. Four open source SLMs (StableCode-Instruct-3B, Qwen2.5-Coder-3B-Instruct, CodeLlama-7B-Instruct, and Phi-3-Mini-4K-Instruct) using 150 Python tasks obtained from LeetCode, categorized as easy, medium, and hard were evaluated. Also, four distinct prompting techniques (ole prompting, zero-shot, few-shot, and chain-of-thought (CoT)) were used to evaluate each model. With a custom baseline the runtime, memory usage, energy expenditure of each generated solution were analyzed. The research indicates that CoT prompting consistently produced Qwen2.5-Coder and StableCode-3B exhibit lower energy consumption, although CodeLlama-7B and Phi-3-Mini-4K consistently underperformed relative to the baseline across all prompting techniques. These findings indicate the model sensitivity of prompt engineering benefits and show that properly designed prompts may enhance the energy efficiency and environmental sustainability of software development processes (Ashraf et al., 2025). Tony et al. (2024) compare the four prompting techniques (zero-shot, zero-shot CoT, RCI, and personamemetic proxy) in terms of their effectiveness in generating secure code with LLMs and finds RCI the most promising method to address security issues in the generated code.

Tony et al. (2025) analyzes diverse prompting strategies for secure code creation employing LLMs, evaluating their effectiveness on models including GPT-3, GPT-3.5, and GPT-4. However, it does not address the principles of prompt engineering within the broader context of software engineering. Prompt engineering offers a beneficial alternative to fine-tuning, enabling the adjustment of pre-trained language models without requiring a supervised dataset. Gao et al. (2023) undertook an empirical examination of the impact of in-context demonstration construction on in-context learning performance across code summarization, bug repair, and program synthesis tasks. The research systematically investigated three key elements: selection, order, and the quantity of illustrative examples. These factors significantly influence model performance when utilizing well-designed prompts, thereby surpassing conventional development methodologies. The optimized examples increased BLEU-4 scores by 9.90% in code summarization, exact match (EM) scores by 175.96% in bug correction, and EM scores by 50.81% in program synthesis. Practical guidelines for creating effective in-context examples were provided, highlighting the essential role of demonstration quality in improving the effectiveness of large language models for code intelligence tasks (Gao et al., 2023). Almeida (2025) comparative study assessed the effects of zero-shot and multi-shot prompting methodologies on task performance in large language models (LLMs), employing metrics like accuracy, precision, and recall. The findings indicated that multi-shot prompting, which incorporates contextual examples within the prompt, frequently improves model performance across various natural language processing (NLP) tasks relative to zero-shot prompting. Furthermore, the study emphasized the significance of adapting prompt techniques to align with the complexity and specific context of the tasks under consideration (Almeida, 2025).

Materials and Methods

The structure of the research was developed based on the factorial design, which can be applied to examine how particular prompt engineering methods (P) affect the code production in PHP, through open-source large language models (M), on a selected coding task set (T). A factorial design is able to explore multiple factors by assessing all the combinations of the various levels of the respective factors, thereby enabling the estimation of both main and interaction effects (Montgomery, 2017). All combinations of the models, prompt techniques and the category of tasks were employed. This factorial design informed the definition of the experimental variables, creation of the PHP task set, prompt templates design, and the creation of an execution environment with results logging, as shown in Figure 1.

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. ***FNAS Journal of Computing and Applications, 3*(1), 61-73.** https://doi.org/10.63561/jca.v3i1.1204

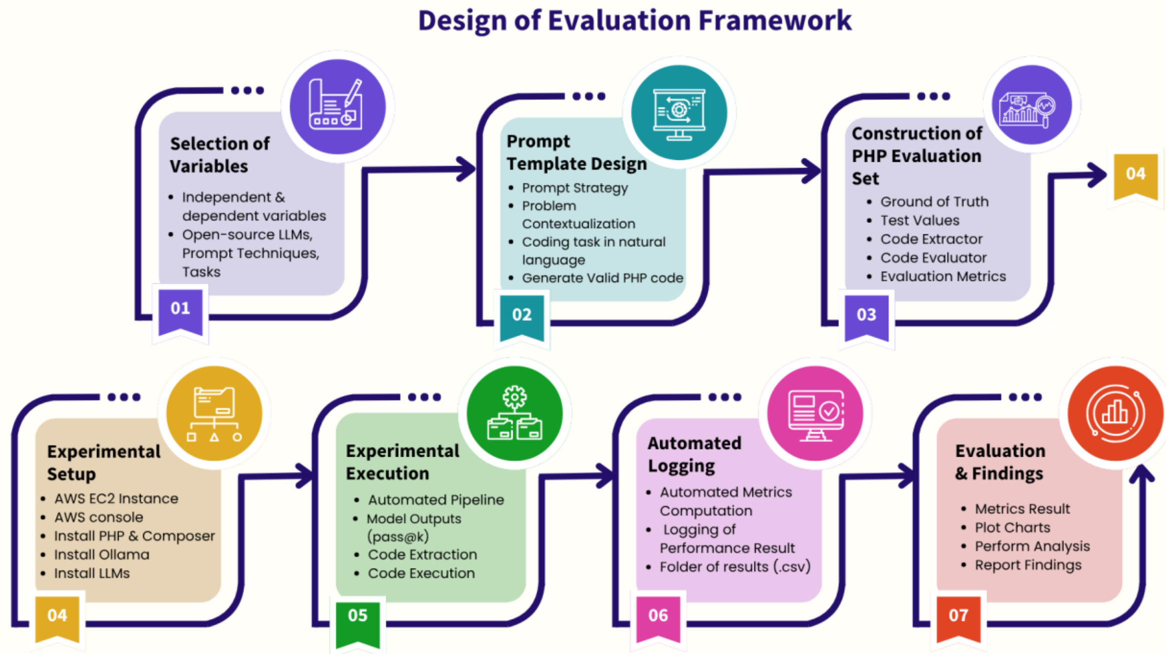

Figure 1: Design of evaluation Framework

Selection of Variables

Table 1 lists the variables that were used to form the factorial design. These variables are used to ensure that the experimental framework includes factors that are significant, such as the type of model, open-source LLMs prompt methods, and programming language.

Table 1: Selection of Experimental Variables

Variable Type	Factor	Selected Models | Techniques | Language	Justification
Model (M)	LLM	Code LLaMA, StarCoder, Mistral	Open-source accessibility and established state-of-the-art performance in code generation benchmarks.
Prompt (P)	Techniques	Zero-Shot, Few-Shot, Chain-of-Thought (CoT)	Representation of fundamental, example-based, and reasoning-based prompting patterns.
Language (L)	Domain	PHP	Focused on code generation in the industry-relevant PHP ecosystem.

Large Language Models (LLMs)

The selected open-source models (Code LLaMA, StarCoder, and Mistral) act as an important independent variable (M) to test the effect of underlying architectures and data on the performance of generated codes.

Prompt Engineering Techniques

The three techniques of prompting (Zero-Shot (ZS), Few-Shot (FS), and Chain-of-Thought (CoT)) form the primary independent variable (P). The empirical examination is on the effectiveness of the techniques of prompting on the output of LLMs.

Programming Language and Scope

PHP is the only domain programming language used for software engineering code generation in this study.

Construction of the PHP Task Set

The six(6) set of tasks (dataset) evenly distributed across three difficulty levels (easy, medium, and difficult) were retrieved from LeetCode. All the coding problems were not related to any framework in order to test the model ability to generate solutions in core PHP code. The coding problems tests for function implementation, logical reasoning, edge-case handling, and algorithmic efficiency by the models.

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Ground Truth and Test Harness
Each of these tasks has two main components to be considered. Firstly, a ground-truth solution, which is a well-written implementation in PHP that follows coding standards such as PSR-12 and has low complexity. This is used to determine the quality and efficiency of the solution. Secondly, a PHPUnit test suite, which contains unit tests to test the accuracy of the solution.

Prompt Template Design
Standardized templates were also prepared to ensure that the only difference between the templates is the application of each prompt engineering technique, so that no differences occur with regard to the phrasing of the task. The common basic structure of the templates is as follows:
1. Description of the tasks using natural language.
2. Instruction to generate the code in PHP.
3. Restriction of the output of only valid PHP code.

Experimental Environment
All code generation and evaluation tests were conducted on a single Amazon EC2 instance for a consistent and reproducible experiment. The instance was created with the "Deep Learning Base AMI with Single CUDA (Ubuntu 22.04) + t3.xlarge" configuration, which includes 4 vCPUs and 120 GiB of storage. All commands were executed through the AWS EC2 Console. These open-source model variants (codellama:7b, starcoder2:3b, and mistral:7b) were installed and executed using Ollama. This setup enables a stable and controlled environment for generating and evaluating PHP code, while avoiding issues such as local memory limitations or incomplete outputs.

Experimental and Code Execution
To balance determinism, a temperature setting of 0.8 was used, with ten samples generated per task.
All outputs were executed independently to obtain performance metrics. The generated and extracted PHP codes were executed on the same AWS EC2 instance console to ensure that there were no hardware-related factors to influence the experiment's outcome. The experiment used PHP 8.1. Each code experiment was conducted in isolation to ensure an accurate measure of the performance metrics. In addition, the execution will be performed using pass@1 and pass@10. This will ensure that the results show the reliability of the model, where the first represents the reliability of the model, while the latter represents the probability that at least one of the generated samples will satisfy all the test conditions.

Code Evaluation Metrics
The generated code was evaluated using both functional and structural metrics:

- **Functional Correctness and Accuracy**

An automated PHPUnit test harness is utilized to assess the functional accuracy of generated PHP code by executing it against predefined test values reflective of expected behavior and test cases. Two key accuracy metrics are applied: Pass@1, measuring the percentage of tasks where the first generated code sample passes all test cases, and Pass@K (K=10), which evaluates the likelihood that at least one of the ten generated outputs is correct. This accuracy metric was originally proposed by Chen et al. (2021) and is given by Equation 3.1. This accuracy metric is considered to be more robust where the output of the model is considered to be variable.
Eq 3.1 Unbiased Estimator

$$Pass@K = \frac{1}{Tasks} \sum_{t=1}^{Tasks} \left[1 - \frac{C(N - Ct, K)}{C(n, k)}\right]$$

Where: N=10 is the total number of generated samples per task, Ct is the number of correct samples for task t, and |T| is the total number of tasks evaluated.

- **Code Quality**:
    - PSR-12 compliance (measured using PHP CodeSniffer)
    - Cyclomatic complexity (measured using PHP Mess Detector)
- **Efficiency Metrics**:
    - Execution time (measured using hrtime(true))
    - Memory usage (measured using memory_get_peak_usage(true))

Evaluation results were automatically recorded in CSV format for interpretation and further analysis.

Data Analysis and Statistical Methods
A Two-way ANOVA test is used to check whether the observed differences are significant or not. The variables of interest in this experiment are Prompt Technique (P) and Large Language Model (M), and the dependent variables of

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

interest are a set of performance metrics such as accuracy, code quality score, runtime efficiency, and memory usage. The analysis of ANOVA framework is on the:

- main effect of prompt techniques on the code generation based on evaluation metrics.
- main effect of model choice.
- interaction effect between prompt technique and model, showing if prompt technique is dependent on models.

Results

Table1: Impact of Prompting Techniques Across Tasks (Pass@1)

Coding Problems	Model	Zero-shot	Few-shot	CoT
CP-01	codellama	100	100	100
	mistral	0	100	0
	starcoder2	0	0	0
CP-02	codellama	0	100	100
	mistral	0	100	33.33
	starcoder2	0	100	100
CP-03	codellama	50	100	0
	mistral	100	33.33	50
	starcoder2	0	0	0
CP-04	codellama	100	66.67	66.67
	mistral	66.67	66.67	66.67
	starcoder2	0	0	33.33
CP-05	codellama	0	0	33.33
	mistral	0	0	0
	starcoder2	0	0	0
CP-06	codellama	33.33	33.33	33.33
	mistral	33.33	33.33	33.33
	starcoder2	0	0	0

From the table above, there is a comparison of pass@1 for six coding tasks, and this shows that with an increase in difficulty, there is a decline in performance for all the prompting techniques. This highlights the impact of each model. For CP-01, which is the easiest, it is clear that there is a variation in accuracy for all models. CodeLLaMA scores 100%, StarCoder2 does not achieve any accuracy, and Mistral shows some improvement in accuracy under few-shot prompting. It is clear that few-shot is the best method, as all models have correct results for tasks CP-02 and CP-03, unlike the other prompt techniques. Zero-shot prompting is used successfully for specific models, such as Mistral for CP-03 and CodeLLaMA for CP-04. In tasks CP-05 and CP-06, none of the techniques yielded correct results. Overall, the ranking of prompting methods from best to worst is few-shot, chain of thought, and then zero-shot. CodeLLaMA with few-shot prompting, followed by Mistral and StarCoder2, ranks highest in robustness. Success diminishes as task difficulty increases, indicating that pass@1 may not accurately reflect the models' overall capabilities concerning algorithm understanding.

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Figure2 : Pass@1 Performance Across Tasks

Table 2 : Impact of Prompting Techniques Across Tasks (Pass@10)

Coding Problem	Model	Zero-shot	Few-shot	CoT
CP-01	codellama	71.11	64	76
CP-01	mistral	100	100	0
CP-01	starcoder2	0	66.67	0
CP-02	codellama	66.67	72.22	100
CP-02	mistral	40	33.33	27.78
CP-02	starcoder2	100	66.67	50
CP-03	codellama	79.63	85.42	45
CP-03	mistral	70	66.67	75
CP-03	starcoder2	33.34	50	10
CP-04	codellama	76.67	66.67	53.33
CP-04	mistral	63.34	53.34	66.67
CP-04	starcoder2	16.66	55.55	33.33
CP-05	codellama	13.33	20.83	33.33
CP-05	mistral	40	23.33	3.33
CP-05	starcoder2	0	0	0

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

CP-06	codellama	33.33	33.33	33.33
CP-06	mistral	30	26.66	33.33
CP-06	starcoder2	0	16.66	16.66

Table 2 shows the overall pass@10 metrics on six coding tasks. Note that unlike pass@1, pass@10 gives each model ten chances, hence measuring the probability that at least one answer among those attempts will be correct. This, in turn, reveals the hidden potential, but the performance degrades as the complexity of the tasks increases. For the straightforward tasks, such as CP-01 and CP-02, most models show high performance for pass@10. As the complexity increases especially for CP-05 and CP-06, the performance of all models decreases. This suggests that multiple attempts improve the results but do not completely mitigate the effects of complexity. As for the prompting strategies, few-shot prompting is the most consistent and stable in its results for pass@10 especially for tasks CP-01 to CP-04 over other models. Although zero-shot prompting performs well for the easier tasks, it degrades for the more complex tasks. Chain-of-thought prompting has shown promise in certain situations, such as CP-02 for CodeLLaMA and CP-03 for Mistral and inconsistencies when dealing with more complex scenarios. Therefore, CodeLLaMA with few-shot prompting performs the best, followed by CodeLLaMA with zero-shot prompting. Next in line is Mistral with zero-shot prompting, showing significant improvement in pass@10 compared to pass@1, which indicates that multiple attempts help the model recover from early underperformance. StarCoder2 even under multiple attempts performs the worst among all the prompting methods. Thus, in brief, pass@10 is shown to improve results over pass@1, but does not guarantee any level of accuracy in the results other than probability of success. When it has to do with a well defined algorithmic structure task, few-shot prompting is the most certain technique. Chain of thought is not shown to improve code generation, and in fact, can even worsen stability in the code. Prompt engineering is merely an enhancement to pre-existing knowledge; this is because there are limitations when the models are sampled with increasing levels of complex tasks. The heatmap of accuracy is represented in figure 3.

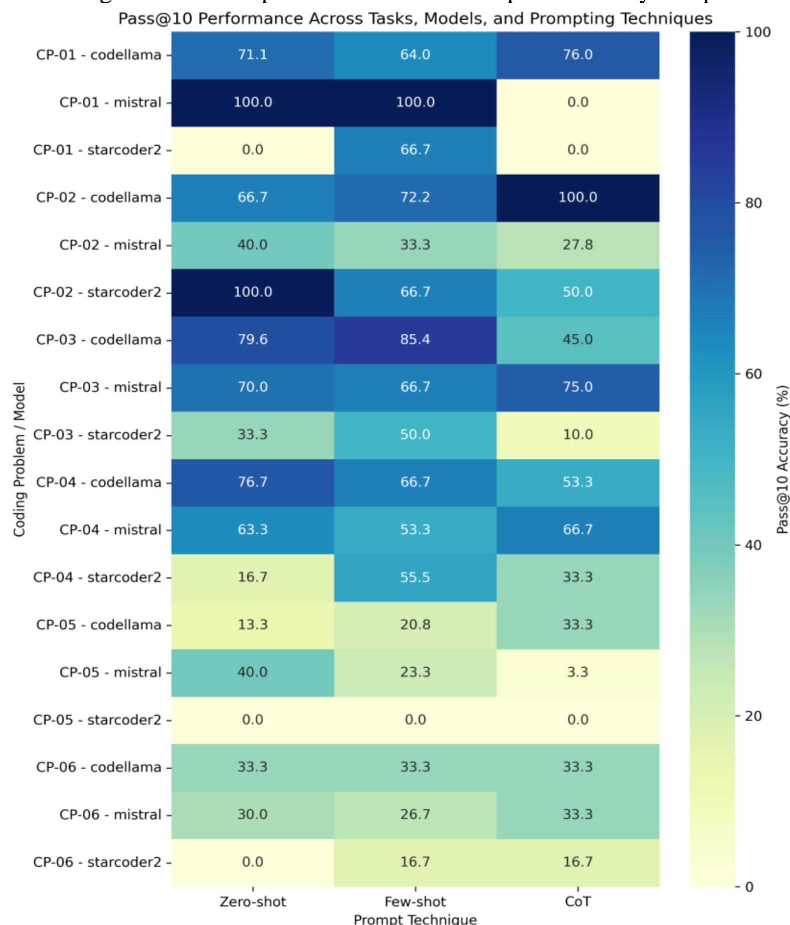

Figure 3: Pass@10 Performance Across Tasks

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Table 3:Model-Level Performance Comparison

Model	Zero-shot Mean	Few-shot Mean	CoT Mean
codellama	55.555	66.66666667	47.22166667
mistral	30.555	55.555	33.33333333
starcoder2	22.22166667	16.66666667	0

The table above shows the mean pass@1 accuracy of all coding tasks for each of the models and prompting techniques, showing the variation in baseline single attempt capabilities. The results show that there are variations in the capabilities of each of the models, with Code LLaMA showing the highest accuracy, followed by Mistral, while StarCoder2 showed the lowest accuracy. Among the prompting techniques, the results show that the highest mean accuracy of all the models was achieved by the few-shot prompting technique. For Code LLaMA, it showed the highest mean accuracy of all the prompting techniques, at 66.67%, compared to zero-shot at 55.56% and chain of thought at 47.22%. For the second-best model, Mistral, it showed improved accuracy at 55.56% compared to other prompting techniques. However, it showed lower accuracy compared to Code LLaMA. StarCoder2 showed the worst accuracy of all the models, with negligible accuracy at chain of thought prompting. The results indicate that prompting techniques enhance model capabilities, but are insufficient alone; model capabilities are also crucial for overall performance.

Table 4: Model-Level Performance Comparison (Pass@10)

Model	Zero-shot Mean	Few-shot Mean	CoT Mean
codellama	56.83283333	57.078	56.78931667
mistral	34.3515	50.55491667	57.22183333
starcoder2	18.33283333	42.59191667	25

The pass@10 mean accuracy, as depicted in Table 4, has a unique trend from pass@1. The reason for this is that the model is prompted ten times for each code output. The performance of CodeLLaMA is consistent with all prompting strategies because it can output the right solution irrespective of the prompt structure. The effectiveness of Mistral improves significantly with pass@10, with Chain-of-Thought prompting having the highest mean accuracy. This implies that with more prompts, the model can perform better, regardless of its initial performance. The outcome of StarCoder2 even with more attempts improves with few-shot prompting but has the lowest performance overall. The pass@10 mean accuracy results depicts the probability of attaining a right answer with more prompts.

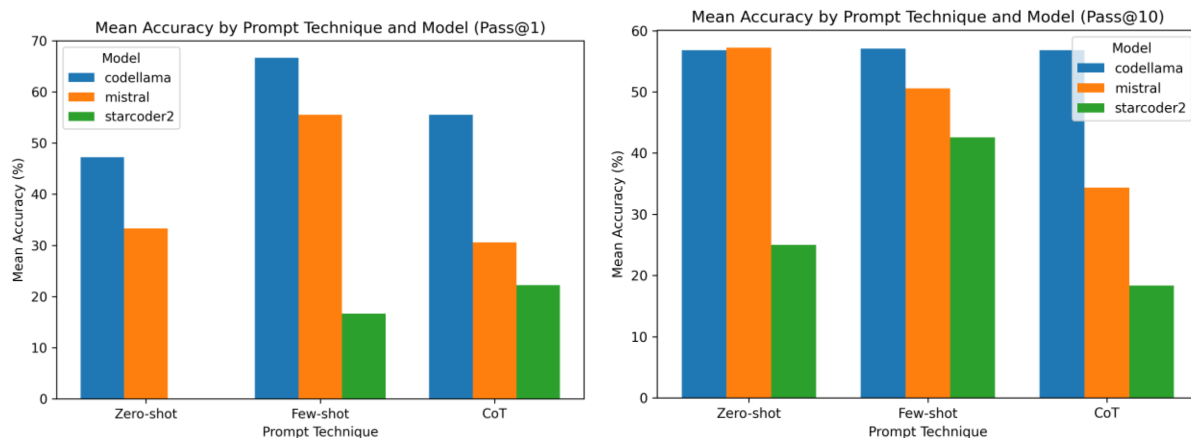

Figure 4: Mean Accuracy by Prompt Technique and Model

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Table5: Descriptive Statistics for Pass@1 Accuracy

prompt	model	N	Mean	SD
cot	codellama	6	55.555	40.36958893
cot	mistral	6	30.555	26.70212932
cot	starcoder2	6	22.22166667	40.36848792
few_shot	codellama	6	66.66666667	42.16422923
few_shot	mistral	6	55.555	40.36958893
few_shot	starcoder2	6	16.66666667	40.82482905
zero	codellama	6	47.22166667	45.23621134
zero	mistral	6	33.33333333	42.16422923
zero	starcoder2	6	0	0

As shown in the table above, the results show significant differences in performance across the LLMs and prompting techniques. For Code LLaMA, the few-shot prompting approach yielded the highest average accuracy, reaching 66.67%. Chain-of-thought prompting achieved an accuracy of 55.56%, while zero-shot prompting resulted in 47.22%. In the case of Mistral, 55.56%, was attained through few-shot prompting, succeeded by zero-shot prompting at 33.33%, and chain-of-thought prompting at 30.56%. Conversely, StarCoder2 demonstrated suboptimal performance across all prompting methodologies. Zero-shot prompting was the worst at 0%, followed by significant improvements in performance when using the few-shot and chain-of-thought prompting techniques. The high standard deviation values show significant differences in performance across different tasks. This justifies the use of two-way ANOVA to evaluate the statistical significance of the effects of model types, prompting techniques, and their interaction on code generation accuracy.

Table 6: Descriptive Statistics for Pass@10 Accuracy

prompt	model	N	Mean	SD
cot	codellama	6	56.83283333	26.41086018
cot	mistral	6	34.3515	31.25166876
cot	starcoder2	6	18.33283333	19.86045206
few_shot	codellama	6	57.078	24.69802585
few_shot	mistral	6	50.55491667	29.39569196
few_shot	starcoder2	6	42.59191667	27.81486755
zero	codellama	6	56.78931667	27.04890993
zero	mistral	6	57.22183333	25.94255056
zero	starcoder2	6	25	39.08694015

The results in Table 6 indicate an increase in mean accuracy for all models compared to pass@1 outcomes, which demonstrates the influence of repeated sampling on code generation performance. Code LLaMA displayed accuracy across all prompting techniques. It exhibited only minor variations, specifically, zero-shot prompting achieved a mean accuracy of 56.79%, few-shot prompting 57.08%, and chain of thought prompting 56.83%. Therefore, the model's performance seems to be largely unaffected by the prompting method used. Under pass@10 for Mistral, Zero-shot prompting yielded a performance of 57.22%, while few-shot prompting achieved 50.55%. This suggests that repeated sampling improves the model's capacity to produce accurate outputs, even when initial outputs fail. StarCoder2 has also shown an improvement with few-shot prompting reaching a score of 42.59% mean accuracy even though it is low when compared to the other models. While the standard deviations remain moderate to high, the results indicate that these remain low as compared to pass@1, which shows the relative stability of the results obtained by repeated

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

sampling. For this pass@10 results, a two-way ANOVA would be an effective tool to determine the effect of model type, prompt techniques, as well as the interaction effect, on code accuracy.

For pass@1, the results showed that prompt technique had little significant impact, $F(2,45) = 1.19$, $p = .313$, this means that the differences between zero-shot, few-shot, and chain-of-thought were not statistically reliable, even though few-shot appeared better for the models. The model type had an effect, $F(2,45) = 6.07$, $p = .005$, indicating its impact in the differences of accuracy. There was no significant interaction between prompt and model, $F(4,45) = 0.28$, $p = .890$, showing that no specific prompt consistently improved one model more than others. In general, the findings suggest that model architecture plays an important role when combined with prompt technique in determining pass@1 accuracy. Similarly for pass@10 results, there was no significant statistical difference observed on the effect of prompt technique, $F(2,45) = 1.10$, $p = .342$. The descriptive results revealed several findings. Generally, the few-shot prompting had the highest average accuracy among the models, while the zero-shot prompting had the highest accuracy for the Mistral model. However, the type of prompt did not have a statistically significant effect under pass@10. On the contrary, the model had a significant impact on the results, which revealed that the inter-model differences had a substantial impact. There was no significant interaction between the prompt type and the model, which revealed that the prompting effect was similar across the models. Overall, the findings suggest that prompt engineering has a practical impact on model performance, as observed in descriptive trends, but model architecture remains the dominant factor, with prompting effects varying across tasks and models rather than exhibiting a uniform statistically significant pattern.

Discussion

The prompting techniques influenced the evaluation metrics (accuracy, quality, speed and memory usage), but the underlying effect of each model ability is also a limitation. This reflects the broader view of large language models discussed by (Haque, 2025), where improvements are significant but still dependent on model capability. Few-shot prompting was consistent across models with the highest accuracy, particularly under pass@1. This supports the role of code generation in the improvement of efficiency and reliability as highlighted by Li et al. (2024). Under pass@10, Zero-shot prompting showed improvements in the possibility of generating a correct code solution. This aligns with the idea that LLMs can generate multiple possible solutions in software development contexts (Li et al., 2024). Chain-of-thought effect is seen in the addition of more structure to responses and sometimes increased code length even though it did not lead to higher accuracy. This can be related to the growing capabilities of LLMs to produce structured outputs as noted by (Haque, 2025), even when correctness is not guaranteed. The structure of the task is significant in this case. The simpler tasks resulted in high accuracy in all settings, while complex ones with deeper reasoning, did not produce fully functional solutions even after multiple attempts. This further support concerns in existing discussions about the limitations of LLMs in handling complex software engineering problems (Haque, 2025). The Execution time and memory usage were constant in all models and prompting techniques. This suggests that, despite improvements in code generation efficiency, prompt variations may not significantly affect system performance. This indicates that prompt design mainly affected correctness rather than efficiency. Code quality measures revealed minor differences in terms of length, complexity, and formatting, but accuracy was not necessarily increased as a consequence. This is consistent with the idea that standardization and consistency in generated code do not always imply correctness. Additionally, under pass@1 and pass@10, the model architecture had a statistically significant effect on accuracy across. This supports the need to explore beyond commonly used models as identified in the problem statement. The best average accuracy across coding tasks and prompting strategies is CodeLLama, followed by Mistral with moderate performance under pass@10, and StarCoder2 with the lowest performance. This comparison aligns with the study's objective of evaluating multiple open-source models rather than focusing only on GPT-based systems.

In single-run evaluations, differences in model architecture are factors that also shaped the results, while repeated sampling increased overall success rates and reduced sensitivity to prompt design. This reinforces the importance of comprehensive evaluation frameworks as proposed in this study. Taken together, the findings suggest that prompt engineering can influence outcomes, but it cannot replace the core ability of the model. This directly addresses the research gap identified in the problem statement regarding the need for empirical evaluation of prompting techniques across models. Its impact should therefore be assessed using statistical testing, multiple evaluation metrics, and a range of task types to provide a balanced understanding of performance in software engineering contexts.

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Conclusion

This work demonstrates that the assessment of prompt engineering techniques for code generation from open-source language models can be achieved in a clear and repeatable way using a structured framework. The evaluation can be done while taking into consideration various factors, such as the correctness, time taken, memory usage, and quality of the software (such as the number of lines of code, cyclomatic complexity, and PSR-12 conformance). The results in this study proves that the prompt methods impact the way the model generates the code especially for few-shot prompting, though the limitations of the model cannot be avoided.
Putting into consideration the code generation format such as (adding text explanation, incomplete code, empty function) from these selected models when prompted, there's a need for manual review by software engineers and security auditors to improve the code and make it contextual.

Recommendations

1. Researchers, software engineers, and practitioners should adopt clear and reproducible evaluation processes for prompt techniques and model interaction as AI becomes more integrated into digital tools and human activities.
2. Contextual evaluation should be prioritized by using consistent task sets, prompt formats, and automated correctness tests, as this provides more reliable results than isolated benchmarks.
3. Large language models (LLMs) should be selected based on their capabilities in specific areas of software engineering to ensure quality and secure software development.
4. The use of illustrative examples can improve the stability of model outputs (context engineering), while reasoning-based prompting may influence performance but can also introduce complexity; evaluation factors should therefore be carefully considered.
5. Model performance depends on both inherent capabilities and evaluation setup: single-response scenarios rely on the model's baseline strength, whereas multiple attempts increase the likelihood of correct outcomes and reduce dependence on prompt design; adaptive systems that adjust prompting strategies may further improve results.

References

Ashraf, H., Danish, S. M., Rahman, S., & Sattar, Z. (2025). *Toward Green Code: Prompting small language models for Energy-Efficient Code generation*. https://arxiv.org/abs/2509.09947

Almeida, J. (2025). Prompt Engineering: A comparative study of prompting techniques in AI language models. *Prompt Engineering: A Comparative Study of Prompting Techniques in AI Language Models*, 1–4. https://doi.org/10.1109/isec64801.2025.11147384

Chen, M., Tworek, J., Jun, H., Yuan, Q., De Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., . . . Zaremba, W. (2021, July 7). *Evaluating large language models trained on code*. https://arxiv.org/abs/2107.03374

Ge, Y., Mei, L., Duan, Z., Li, T., Zheng, Y., Wang, Y., Wang, L., Yao, J., Liu, T., Cai, Y., Bi, B., Guo, F., Guo, J., Liu, S., & Cheng, X. (2025). *A Survey of Vibe Coding with Large Language Models*. https://arxiv.org/abs/2510.12399

Gao, S., Wen, X., Gao, C., Wang, W., Zhang, H., & Lyu, M. R. (2023). What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs? *ASE '23: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 761–773. https://doi.org/10.1109/ase56229.2023.00109

Haque, M. A. (2025). LLMs: A game-changer for software engineers? *BenchCouncil Transactions on Benchmarks Standards and Evaluations*, *5*(1), 100204. https://doi.org/10.1016/j.tbench.2025.10020

Li, Y., Shi, J., & Zhang, Z. (2024). *An approach for rapid source code development based on ChatGPT and prompt engineering*. Purple Mountain Laboratories; State Key Laboratory of Mathematical Engineering and Advanced Computing.

Khojah, R., De Oliveira Neto, F. G., Mohamad, M., & Leitner, P. (2024, December 29). *The impact of prompt programming on Function-Level code Generation*. arXiv.org. https://arxiv.org/abs/2412.20545

Montgomery, D. C. (2017). *Design and analysis of experiments* (9th ed.). Wiley.

Mayer, L., Heumann, C., & Aßenmacher, M. (2024, September 6). *Can OpenSource beat ChatGPT? -- A Comparative Study of Large Language Models for Text-to-Code Generation*. arXiv.org. https://arxiv.org/abs/2409.04164

Shin, J., Tang, C., Mohati, T., Nayebi, M., Wang, S., & Hemmati, H. (2023, October 11). *Prompt Engineering or*

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204

Fine-Tuning: An Empirical Assessment of LLMs for code. arXiv.org. https://arxiv.org/abs/2310.10508

Shin, J., Wei, M., Wang, J., Shi, L., & Wang, S. (2023). The good, the bad, and the missing: Neural code generation for machine learning tasks. *ACM Transactions on Software Engineering and Methodology*, *33*(2), 1–24. https://doi.org/10.1145/3630009

Tony, C., Ferreyra, N. E. D., Mutas, M., Dhif, S., & Scandariato, R. (2025). Prompting Techniques for secure code Generation: A Systematic investigation. *ACM Transactions on Software Engineering and Methodology*, *34*(8), 1–53. https://doi.org/10.1145/3722108

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). *Chain-of-Thought prompting elicits reasoning in large language models*. https://arxiv.org/abs/2201.11903

Cite this article as:
Alemoh, R. B., & Muhammad-Bello, B. L. (2026). A framework for evaluating prompt engineering techniques for code generation in PHP using open-source large language models. *FNAS Journal of Computing and Applications, 3*(1), 61-73. https://doi.org/10.63561/jca.v3i1.1204