TECHNIUM

Technium Vol. 31, pp.179-188 (2026)
ISSN: 2668-778X
www.techniumscience.com

Prompting Patterns for Code Understanding: A Controlled Experiment with Professional Developers

Hasanain Hazim Azeez
Computer Science and IT Faculty, Wasit University, Al-kut, Iraq.
hbashagha@uowasit.edu.iq

Ahmed Alaa Mohsin
Computer Science and IT Faculty, Wasit University, Al-kut, Iraq.
ahmed.alaa@uowasit.edu.iq

Hind Ali Abdul-Hassan
Computer Science and IT Faculty, Wasit University, Al-kut, Iraq.
hind.ali@uowasit.edu.iq

Abstract.

Even though program comprehension tasks occupy a sizable fraction of a developer's working hours, most developers still have a hard time comprehend poorly documented or unfamiliar code. While natural-language interfaces to explore source code built upon recent advances in large-language models (LLMs) are available, the quality of the conversation is heavily dependent upon the way questions are structured. In this paper, we inquire whether initial-highlighter prompt patterns provide benefit to developers as they understand code via LLM interaction. For a controlled experiment, we recruited thirty professional developers that tried to decode Python functions they had never seen before. As part of a within-subjects design they first responded to comprehension questions without help, then queried an LLM with either an unstructured natural-language query (the baseline) or one of two structured prompt patterns. Comprehension was assessed in terms of accuracy, time and subjective confidence. The results demonstrate that structured patterns significantly improved comprehension accuracy, and it also took less time to complete compared with the baseline. We provide a set of guidelines to use to leverage effective prompt designs, and indicate how prompt patterns can supplement existing documentation and code understanding tools. The findings highlight the potential of prompt engineering to augment developers' comprehension while cautioning against over-reliance on automated suggestions.

Keywords: Code comprehension; prompt engineering; large-language models; controlled experiment; software engineering; professional developers.

1 Introduction

Reason For Visit: Understanding source code is at the center of software engineering. The proportion of time developers spend reading and understanding code is consistently reported

between 30% and 70% across surveys and empirical studies[1]. Poor or antiquated documentation[2] adds to the expense of understanding. Extensive mining of mailing lists, issue trackers and pull requests has also demonstrated that the creation and maintenance of documentation is rarely a priority, leading to the presence of misleading or outdated information [3]. Even though practitioners view documentation important for understanding software artifacts, they also suggest better support for documentation[4]. Work on professional developers show that they will read less, but build a mental model in their heads by taking the point of view of end users and relying on experience rather than formal documentation [5]

. This gap between research prototypes and industrial practice motivates the search for tools that improve program comprehension.

The emergence of generative AI systems, such as ChatGPT, Copilot and CodeWhisperer, has created new opportunities to assist software developers. Generative systems can summarise code, suggest fixes or generate tests[6], and early evaluations of GitHub Copilot reported that it produces syntactically valid code in 91.5 % of cases, although only 28.7 % of its outputs are entirely correct[7]. Adoption has been rapid: ChatGPT reached more than 100 million users within two months of its release, and generative AI tools are now integrated into popular development environments[8]. The remarkable performance of GPT-4 across domains, including programming, suggests the emergence of general reasoning abilities[9]. However, these models are opaque, sensitive to the phrasing of prompts and can hallucinate plausible but incorrect answers. Guidelines for prompting LLMs have therefore begun to appear. Midolo et al. derive ten prompt optimisation guidelines specific to code generation, such as clearly specifying inputs and outputs, including examples and pre-conditions, and decomposing complex tasks[10]. Their evaluation shows that such structured prompts significantly improve output quality[11]. Bistarelli et al.'s review of 66 papers on code generation notes that most experiments focus on Python and that ethical and security constraints are under-explored[12]. Kruse et al. conducted a controlled experiment comparing predefined few-shot prompts with ad-hoc natural-language queries for code documentation; participants preferred structured prompts and achieved higher quality documentation[13]. While these studies imply that the quality of prompt can vary depending on the prompt design used for the image generation of code or documentation, little work has focused on identifying patterns in prompts for code comprehension tasks.

Overall, previous work demonstrates better code generation and documentation through structured prompts, as well as benefits from annotations and visual aids when comprehending programs; meanwhile, existing experiments often have low ecological validity. What we are unaware of is covered multiple times by the prompt patterns to understand real code when professional developers use LLMs, but no controlled study has tackled it. In this paper, the following contributions are made.

Prompt Pattern Creation for Understanding Code We specified two structured prompt patterns—one is an explicit question pattern (asking for function signature, purpose and key variables) and the other is a chain-of-thought pattern to guide the LLM to think in a step-by-step manner—and compare them against unstructured queries.

Controlled experiment with professional developers. We collect data from thirty intermediate-level developers evaluating their understanding of novel Python functions across three prompting conditions. Overall, the study design prioritizes ecological validity by employing real-world code and directly interacting with the LLM as it would be in practice.

Empirical evaluation of prompt patterns. We describe how each pattern had an impact on the accuracy, time and perceived confidence in comprehension, and statistically analyse the results. We infer principles for designing prompts that enhance understanding of the code.

Discussion of implications. Based on empirical findings and theoretical insights from the literature, we discuss how prompt patterns complement the two existing documentation practices and code comprehension models.

The remainder of the paper is organised as follows. Section 3 details the experimental methodology, Section 4 presents the results and discussion, Section 5 provides the conclusion, and the references follow thereafter.

2 Related Work

Program comprehension research is an area of program understanding that achieves techniques that improve developers understanding of legacy code. Algorithm labels in source code facilitate understanding by about 23% with no added time to completion[14], and linear representation of API code examples reduce reaction time for developers[15]. CodeMap enhances the human–AI collaborative experience by combining LLM text with visualizations; CodeMap users find it easier to use and have to read less text from LLM outputs[16]. However, the design of code comprehension experiments are inconsistent. In a systematic mapping study of 95 experiments over a period of forty years, the variety of study designs was revealed, and the need for more rigorous methods was noted[1]. In their extended argument, Chin and Holmes contend that much of our experimentation is of limited ecological validity in that it uses artificial code snippets, small methods, and student subjects[17].

Introduction Prompt engineering is a new discipline to steer LLM towards useful outputs. The "pre-train, prompt and predict" paradigm changes the definition of tasks into textual prompts to support few-shot and zero-shot learning[18], while instruction for code generation is to specify input, output and examples[10]. Nonetheless, prompting strategies for code understanding are still underexplored. Surprisal-based experiments demonstrate that human subjects find less predictable code fragments (from the viewpoint of a large language model)

more difficult to process than predictable fragments[19]. LLM tools assist developers in finding relevant code and familiarising them with low-level understanding and field studies show that this can be done without expertise[20]. For the task of vulnerability repair, including control-flow graphs and other context in prompts markedly increases success rates[21]. Cognitive factors including intelligence and conscientiousness can be confounding variables in comprehension studies [22], and reviewers typically adopt opportunistic strategies to construct mental models from changes to code [23]. Our discoveries indicate that considering cognitive load and developer strategies should be part of prompt design.

3 Methodology

We take a controlled within-subject experimental approach to determine if professional developers benefit from structured prompting patterns to better understand new-to-them source code when using a large language model (LLM). Consistent with best practices in empirical software engineering, the experimental design was deliberately designed to enhance internal validity, ecological validity and statistical power, respectively.

There are three primary research questions guiding the investigation. We first study whether exerting structural prompt patterns results in better accuracy in understanding code than natural-language queries, which are in an unstructured form. Next, we investigate whether these types of prompt patterns minimize the time spent in code comprehension effort. Third, we investigate if structured prompting increases developers subjective confidence in their understanding. We therefore hypothesize that given structured prompts, which are informed by prior work on prompt engineering, accuracy, task completion time and confidence will increase compared to unstructured prompts (for structured prompts) based on findings in program comprehension research.

In order to test these hypotheses, we used a within-subject experimental design. In three different prompting conditions, comprehension tasks were given to each participant. Participants interacted with the LLM in the baseline condition using a free-form natural-language query of their choice. In one of the most structured conditions (Pattern A), participants used a standard template which clearly prompts for the function purpose, input parameters, output values, and important variables. In the second structure (Pattern B), we had participants employ a chain–of–thought prompt that tells the LLM to reason step by step and to specify how the individual code statements contribute to the overall working of the code. We used a Latin square design to counterbalance the order of these conditions so as to reduce learning and the order effect.

The experiment featured thirty professionals engaged in software engineering. Every one of them was a Python programmer who used people whose industry career had already begun to become familiar with today's lifestyle. Each one of these developers had at least five years 'experience in the field.

Participants were recruited through industrial networks or friendly relations (i.e., in order to get a more realistic, ecological momentary experience; the practice, as modern software development is). Students were intentionally noctured out.

Two realistic Python functions were selected as the basis for the exercises. They reflect a typical code maintenance situation and contain the comprehensibility challenge of understanding.

The first essentially converts configuration data in schema-ed files and multi-level dictionaries to the nested data structures, and the second deduples records so that dissimilar individual entities become homogeneous. We deliberately tuned the code length and complexity of all tasks not to take any potential bias into account. All functions had nontrivial control flow, and were 20·30 lines long.

Table 1: Three prompt conditions used in the experiment.

Table 1 – Prompt conditions used in the experiment.

Condition	Description	Example prompt
Baseline	Participant writes a free-form natural-language question to ask the LLM anything about the code.	"Can you explain what this function does?"
Pattern A: explicit questions	A template asks for the function's purpose, inputs, outputs and key variables.	"Summarise the purpose of the function, list its input parameters and return values, and describe the main variables used."
Pattern B: chain-of-thought	A template instructs the LLM to reason step-by-step and explain each line's contribution.	"Describe the function's behaviour step by step, explaining how each line contributes to the final result."

The procedure for each experimental session was consistent and structured. First, participants completed a brief pre-test intended to verify that they already possessed a baseline familiarity with Python syntax and constructs. They then completed the three comprehension tasks, one under each condition of prompting. For each of the tasks, participants were shown the source code, then notified that they were only able to interact with the LLM once with the prompt pattern for that task. Once the LLM responded, the participants answered 4 comprehension questions focused on the function: its intended purpose, input–output behavior, and important implementation details. Time to complete the task was automatically recorded. After every task, participants rated their confidence in their answers and in the helpfulness of the LLM's response with a five-point Likert scale. Participants filled a brief post-questionnaire at the end

of the session that involved qualitative feedback and prompt preferences. Figure 1 shows the general workflow of the experiments.

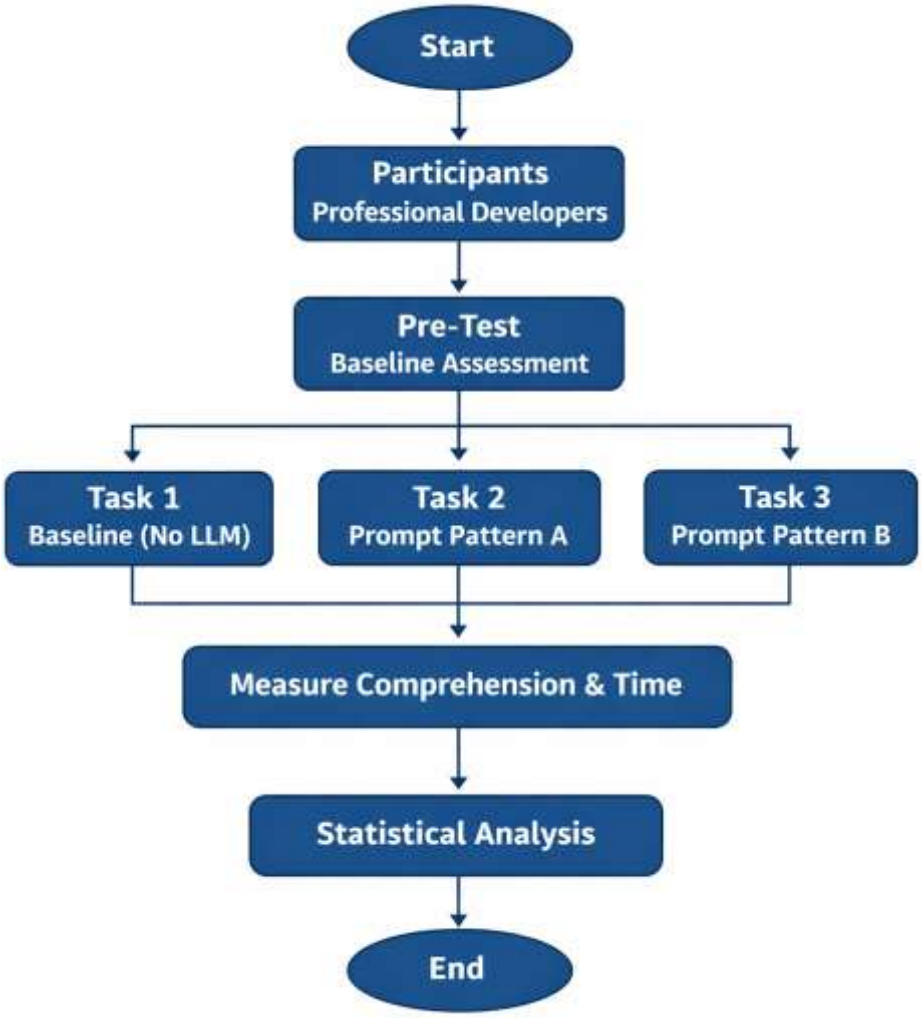

Figure 1 – Flowchart of the controlled experiment methodology.

Three dependent variables were collected during the experiment. Comprehension accuracy was computed as the proportion of correct answers across the four comprehension questions:

$$\text{Accuracy} = \frac{\sum_{i=1}^{4} \text{correct}_i}{4}.$$

In addition, completion time was measured in minutes from the moment the code was presented until the participant submitted their final answers. Subjective confidence was calculated as the mean of the Likert-scale ratings associated with each task.

We used repeated−measures analysis of variance (ANOVA) with prompt condition as the between-subjects factor for statistical analysis. Assumptions of normality and sphericity were assessed prior to analysis and Greenhouse–Geisser corrections were used when appropriate. Bonferroni-adjusted paired t-tests were used for post-hoc comparisons between prompting conditionstesting. Omnibus tests were characterized by partial eta squared (ηp 2), while Cohen's d was used to characterize pairwise comparisons. Statistical significance was defined at $p < 0.05$. Table 2 summarises participant demographics.

Table 2 – Participant demographics ($n = 30$). Participants could report multiple languages.

Characteristic	Mean ± SD
Age (years)	32.1 ± 4.8
Professional experience (years)	8.5 ± 3.2
Primary programming languages	Python (28), Java (10), JavaScript (7)
Prior LLM usage	Yes (23), No (7)

4 Results and discussion

Results show the main quantitative results of the experiment (i.e., comprehension accuracy and confidence, both on proportion scale (0–1), and completion time (mean ± SD) in minutes. On all measures this gives a clear and consistent advantage to structured prompt patterns versus unstructured, natural-language queries.

Results A repeated-measures ANOVA revealed a significant effect of prompt condition on accuracy of comprehension, ($F(2,58)=12.6$, $p<0.001$), and completion time for the task, ($F(2,58)=8.1$, $p<0.01$). The trend was similar for the confidence ratings, with higher confidence given in the structured compared to unstructured prompts. These findings can significantly impact code comprehension effectiveness (and efficiency) among developers and highlight the importance of prompt engineering with LLMs.

Table 3. Average comprehension accuracy, completion time, and confidence by prompt condition (mean ± SD). Higher values indicate better performance, except for completion time.

Condition	**Accuracy**	**Completion time (min)**	**Confidence**
Baseline	0.72 ± 0.13	9.4 ± 2.1	0.64 ± 0.15
Pattern A	0.85 ± 0.10	7.6 ± 1.8	0.78 ± 0.12
Pattern B	0.88 ± 0.09	7.2 ± 1.6	0.82 ± 0.11

Pairwise comparisons post-hoc established that all effects corresponding to structured prompting strategies being significant for improvement in accuracy and completion time with respect to the baseline condition. Effect sizes Cohen's d between Patterns A and the baseline were medium for accuracy (d=0.56) and time (d=0.68), while these effects were larger between Patterns B (d=0.71 for accuracy | d=0.77 for time). The difference between Pattern A and Pattern B were also statistically significant for accuracy (p=0.045), indicating that in this case requesting the inputs, outputs, and prominent variables of the model and requesting it to reason step by step served the model more than simply requesting the model to reason step by step.

According to these quantitative results, structured prompt patterns resulted in higher quality mental models of the code. As structured prompts, they seem to be making the model reduce cognitive friction when synthesizing programs by focusing on functional intent all the time and control flow most, effectively guiding developers to pay attention what are semantically important things for the program which needs its own unique philosophically clear name. In fact, the qualitative feedback from participants almost unanimously suggested that structured prompts would conduct their logical reasoning and reduce the burden to read unfamiliar code. With the chain-of-thought pattern, participants could follow the control flow in a more structured and systematic way, and with the explicit question pattern, they found it easier to identify which variables to choose and what actions to take to consider that data. In contrast, baseline queries that were not well-defined tended to return summaries that omitted important implementation details.

These results, more broadly, coincide with previous studies suggesting that mechanisms of guidance (program-level tags, structured documentation, or visual cues) can significantly improve the understanding of programs. The goal of our results is to augment this line of work, as we demonstrate that prompt engineering, in isolation, can serve as a lenient form of cognitive scaffolding when embedded in LLM-assisted development workflows. Compared to studies that focus solely on code generation or documentation, this paper provides controlled evidence about the relevance of prompt patterns also in comprehension tasks when large language models are interacted with by professional developers.

This has massive implications on tool design. By injecting template filled prompts or optimally recommended prompts into integrated development environments, developers could receive more relevant, actionable explanations from LLMs. Specifically, asking them to use reasoning step by step appears to be an effective way to make the outputs from these models more interpretable and actionable. At the same time, results also act as a caution for excessive use of unstructured queries, which can easily lead to less confident but shallow explanations. These results suggest that prompt engineering should be regarded not as a haphazard part of user skill, but rather a first-class design element of LLM-based developer assistance tools.

5 Conclusion

In this paper, we described a controlled experiment that studied how structured prompt patterns affect professional developers understanding of code in interacting with a large-language model. We contrast unstructured queries with two structured patterns (a template request such as purpose, inputs, outputs and variables, and a chain-of-thought prompt requiring step-by-step reasoning). Results indicate structured prompts facilitate comprehension accuracy, efficiency of prompt completion, and developer confidence. Our findings indicate prompt engineering can provide meaningful support for program comprehension and that developer assistance tools should incorporate it. Our work contributes to the emerging literature on prompt engineering for software engineering tasks by providing clear guidelines on how to design high-quality prompts and paves the way for future work into the use of human–AI collaboration on code understanding.

6 References

[1] A. Yetistiren, I. Ozsoy, and S. Demirel, "Assessing the quality of GitHub Copilot's code generation," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2022, pp. 1–5.
[2] S. Bubeck et al., "Sparks of artificial general intelligence: Early experiments with GPT-4," *arXiv preprint arXiv:2303.12712*, 2023.
[3] J. Zhang et al., "Practices and challenges of using GitHub Copilot: An empirical study," *IEEE Softw.*, vol. 40, no. 5, pp. 34–42, Sep.–Oct. 2023.
[4] C. Ebert and P. Louridas, "Generative AI for software practitioners," *IEEE Softw.*, vol. 40, no. 3, pp. 102–108, May–Jun. 2023.
[5] P. Liu et al., "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 1–35, Jan. 2023.
[6] T. Roehm, R. Tiarks, C. Koschke, and W. Maalej, "How do professional developers comprehend software?" *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1037–1053, Sep. 2012.
[7] E. Aghajani et al., "Software documentation issues unveiled," in *Proc. 26th IEEE Int. Conf. Softw. Anal., Evol. and Reeng. (SANER)*, 2019, pp. 119–130.
[8] M. Wyrich, J. Bogner, and S. Wagner, "A systematic mapping study on code comprehension experiments," *ACM Comput. Surv.*, vol. 56, no. 4, pp. 1–34, Apr. 2024.
[9] Y. F. Liu, J. Kim, C. Wilson, and M. Bedny, "The cortical network for computing code comprehension," *eLife*, vol. 9, p. e59340, Dec. 2020.
[10] S. Wagner and M. Wyrich, "A study of intelligence and personality in code comprehension," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2574–2589, Apr. 2023.
[11] F. Zampetti, M. D. Penta, and M. Linares-Vásquez, "Self-admitted technical debt practices: A comparison between industry and open source," *J. Syst. Softw.*, vol. 179, p. 110991, Sep. 2021.
[12] I. Stanik, C. König, and A. Maalej, "A simple NLP-based approach to support onboarding and retention in open-source communities," in *Proc. 30th Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, 2018, pp. 517–522.
[13] C. Casalnuovo, P. Devanbu, and E. Morgan, "Does surprisal predict code comprehension difficulty?" in *Proc. 42nd Annu. Meeting Cognit. Sci. Soc.*, 2020, pp. 564–570.

[14] J. Dominic, B. Tubre, D. Kunkel, and P. Rodeghero, "The human experience of comprehending source code in virtual reality," *Empirical Softw. Eng.*, vol. 27, no. 7, p. 173, Nov. 2022.
[15] J. Cui et al., "Code comprehension: A review and large language models exploration," in *Proc. 4th IEEE Int. Conf. Softw. Eng. Artif. Intell. (SEAI)*, 2024, pp. 1–8.
[16] B. Vasilescu et al., "Social media in hands-on software engineering education," in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 310–320.
[17] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. 35th IEEE Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 712–721.
[18] L. Mou et al., "Convolutional neural networks over tree structures for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.
[19] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, Sep. 2018.
[20] V. J. Hellendoorn and P. Devanbu, "Are code sequences predictable?" in *Proc. 25th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2017, pp. 157–167.
[21] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 9th ed. New York, NY, USA: McGraw-Hill Education, 2020.
[22] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1995.
[23] I. Sommerville, *Software Engineering*, 10th ed. Harlow, UK: Pearson Education, 2016.