SmartDoc: A Context-Aware Agentic Method Comment Generation Plugin

Vahid Etemadi
Independent Researcher
Shiraz, Iran
vetemadi87@gmail.com

Gregorio Robles
Universidad Rey Juan Carlos
Madrid, Spain
grex@gsyc.urjc.es

Abstract

Context: The software maintenance phase involves many activities such as code refactoring, bug fixing, code review or testing. Program comprehension is key to all these activities, as it demands developers to grasp the knowledge (e.g., implementation details) required to modify the codebase. Methods as main building blocks in a program can offer developers this knowledge source for code comprehension. However, reading entire method statements can be challenging, which necessitates precise and up-to-date comments. **Objective**: We propose a solution as an IntelliJ IDEA plugin, named SmartDoc, that assists developers in generating context-aware method comments. **Method**: This plugin acts as an Artificial Intelligence (AI) agent that has its own memory and is augmented by target methods' context. When a request is initiated by the end-user, the method content and all its nested method calls are used in the comment generation. At the beginning, these nested methods are visited and a call graph is generated. This graph is then traversed using depth-first search (DFS), enabling the provision of full-context to enrich Large Language Model (LLM) prompts. **Result**: The product is a software, as a plugin, developed for Java codebase and installable on IntelliJ IDEA. This plugin can serve concurrently for methods whose comments are being updated , and it shares memory across all flows to avoid redundant calls. o measure the accuracy of this solution, a dedicated test case is run to record SmartDoc generated comments and their corresponding ground truth. For each collected result-set, three metrics are computed, BERTScore, BLEU and ROUGE-1. These metrics will determine how accurate the generated comments are in comparison to the ground truth. **Result**: The obtained accuracy, in terms of the precision, recall and F1, is promising, and lies in the range of **0.80** to **0.90** for BERTScore. In addition, a feedback mechanism is developed to enable receiving users' opinion as satisfaction grade and their text.

Keywords

Program comprehension, AI agents, Context awareness, Large language models, Method comments

1 Introduction

Code comprehension is a prerequisite for making software changes, as it influences the developer's perception of the code to be changed. This perception is necessary for developers to make their changes in a way to avoid breaking the functionality (especially in a low test-coverage rate). These changes could be refactoring the code, adding new features, or fixing a bug. Occasionally, code reviewers must understand and read the code before they can offer their advice, accept or reject a pull request. The level of understanding of the code to be changed also affects developer productivity, reducing the likelihood of missed deadlines.

There is a process behind understanding the code, in which code summarization and code comments can play a key role. For example, at the method level, the comment provided would help software developers to facilitate this understanding process, especially if there are nested calls in the body, or it is unapproachably long [5]. Given this complexity, these methods are usually left commentless or sometimes become outdated when a change is applied to the method but not to its comment [4]. Over time, this process can become detrimental to the project if developers who wrote that code are no longer in the team [3]. In all these cases, a developer who touches that block of code (i.e., the method) for the first time will often need more time to make the necessary changes. This is because incomplete understanding of the code may lead to introducing code smells or transitive technical debt [11].

In this paper, we present SmartDoc, an IntelliJ IDEA plugin designed to help developers who need to quickly grasp a method's functionality from its comment. This tool consumes a remote/local LLM server by providing appropriate context (Retrieval Augmented Generation (RAG)), and offers an AI-generated comment for the given method. This process is done by an implemented agent that handles each user request. After initiating the request as an action, the handler which is linked to agent core receives the request carrying target method metadata. It then goes through several key processes to deliver eventual generated structurally valid comment. Internally, the agent traverses nested method calls to enrich its prompt that is finally submitted to the LLM service. In this way, with minimal effort and by harnessing the power of LLMs, a developer can save time and effort devoted to making the intended changes.

The rest of the paper is organized as follows: in Section 2, related works are investigated. In Section 3, the conceptual design of SmartDoc is presented, and the architectural details of the created tool are delivered. In Section 4, we provide the evaluations applied to the generated comments by SmartDoc, and Section 5 discusses the tool and concludes this paper.

2 Related works

Program comprehension refers to the ability to understand code at any level of granularity –whether for maintenance, debugging, review, or development– thereby enabling effective reasoning and software modification [9]. Understanding the code may become challenging as developers have to spend a lot of time if they do not have access to the relevant resources. The situation can become even more complex when developers work with a tightly coupled codebase. According to the Developer Ecosystem Survey

conducted by JetBrains, 42% of developers spend less than 1 hour each day on manual code reviews. Other 45% spend 1-2 hours per day. Much of this effort likely is devoted to understanding and comprehending the code. What if this time could be saved through the automatic generation of method comments designed to convey the functionality of each method?

Generative AI, and specifically LLMs are invented to help developers to become more productive and precise during software development [13]. Code summarisation is the task of interpreting a given block of code into an understandable and easy to follow text, usually more human readable text developed close to natural language format [1]. It serves various purposes, including as an approach for describing a method's functionality. While code comments are not always a natural interpretation of the code, they can convey the method intent with less effort. One of the key features of recently appeared LLMs is their fundamental design feature that enable an effective summaraization based on the user prompt [14]. These types of transformer-based models are capable of generating desired output given an engineered prompt. LLMs can be instructed to produce desired output, however, they are featured as being non-deterministic. Non-deterministic behavior means that in each try with the same input directed to a specific single model, different and still correct responses are expected. In addition, when it is related to producing documents, responses should follow specific structure, opening a new topic about structured output [8].

LLMs are resource intensive when trained in terms of the required infrastructure. However, they can be fine-tuned for specific use case, e.g, a dedicated codebase [7]. But a fine-tuned model that is expected to deal with many types of prompts needs a high volume of resources, making the deployment more costly. In this case, some techniques can be effective, such as:

- Using a caching system, a local, structurally-organized memory that avoids re-submitting prompts and keeps former responses for possible hits.
- Include limitations for number of tokens submitted for a given prompt or an engineered prompt to avoid irrelevant extra response tokens.
- A nested selection mechanism, meaning an agent AI is responsible for making decision to select right model [12].

LLMs are also supported by a few techniques that help with reasoning power, enabling task breakdown (impact on caching etc.) and structured output preparation. Chain-of-Thought is designed to allow sequential reasoning power by analyzing responses received and cached so far to mimic how an intelligent creature decide [8]. This can be impactful when a chain of calls is made to a remote LLM for sub-context members that must contribute to the final output. Few-shot learning is a technique that extends an input prompt with a few examples to guide a remote model toward a structured response [16]. This structured output preparation can be achieved with other approaches as well. For instance, injecting some training parameters, resulting in a customized model, or simply using a retry-based attempt with a predefined try-count to eventually receive desired structure.

There are similar tools that offer code understanding using LLMs. Authors in [10] study the procedure that developers take to send their prompt to LLM. The IDE-plugin, GILT (Generation-based Information-support with LLM Technology), they have created is capable of injecting contextual information into prompt prior to submission to LLM provider. Their works generally for all type of questions a developer may want to ask LLM, including implementation tasks. A few key metrics included when the context added to the prompt such as developer interaction history. In our work, we suggest a deep, precise context, designed specifically for method comment generation that goes beyond shallow prompt preparation.

In an another similar effort, authors in [2] recommend enriching prompts for understanding code comment shared in Stack Overflow. They suggest using context provided in questions to help converging toward better generated comment by LLM. It is similar to our work as we both intend to provide more precise and relevant context for more accurate responses.

JetBrains AI Assistant is another example of an AI-powered IDE solution, positioning itself as a pioneer in integrating generative AI into modern development environments[1]. This comprehensive extension provides developers with a wide range of AI features — from code completion and AI chat to intelligent edit suggestions. SmartDoc, in particular, focuses on delivering an engineered approach for generating comments for given methods. While we cannot compete with JetBrains AI Assistant as a general-purpose tool that supports various development activities, our work can be seen as an effort to provide an activity-specific solution focused on code comment generation for end users.

3 SmartDoc

SmartDoc, as its name suggests, is a tool that assists developers in comment generation, primarily at the method level[2]. This tool is delivered as an IntelliJ IDEA plugin and can be considered an AI agent. This agent is able to produce comments that can equalize with the human-generated text (see Section 4, BERTScore). It leverages a recursive traversal of all callees rooted in the method body to create a temporary memory, enabling taking advantage of RAG principles. Via following this approach, the eventual request submitted for receiving the comment carries all relevant key contexts combined. In RAG-powered prompt generation, a contextual data source plays a key role in assisting remote AI to provide accurate response. Via these settings included in SmartDoc, it can be labeled as a context-aware system.

3.1 Architecture

Interaction between SmartDoc and the end-user in the abstract level, as a workflow, is shown in Figure 1. This is a specific flow taken place when a developer interacts with the plugin. Each request is particularity associated with a method that has a specific signature and input parameters. Each user's request for a method initiates this workflow that is listened by the doc generator component and ends with a structurally valid JavaDoc style comment (editable by developer as well). Within this workflow, the doc generator component is responsible for coordinating context provisioning and communication with the remote/local LLM provider.

At the core, each comment generation request for a given method travels steps 1 to 9 (shown in Figure 2) to generate a comment.

[1] https://www.jetbrains.com/help/idea/ai-assistant-in-jetbrains-ides.html
[2] Available from: https://github.com/vahidetemadi/smartdoc/tree/develop

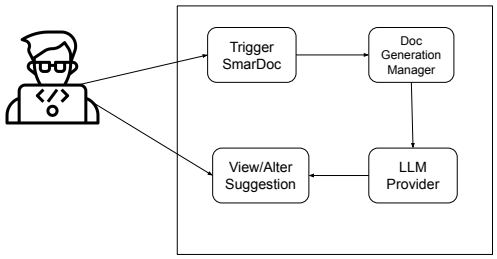

Figure 1: SmartDoc workflow representing interaction of user with the system and internal data flow

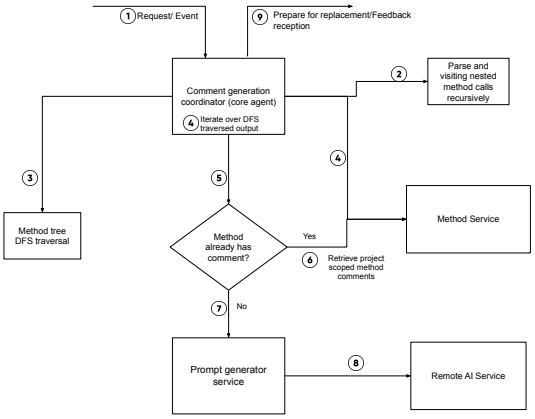

Figure 2: The workflow of the comment generation request

This detailed flow describes the core logic behind processing each request, including all dependencies between services to handle that request.

SmartDoc, at its core, is assisted by an AI agent (see Figure 3). This agent involves a caching memory (suitable for context-preserving and optimized resource usage), that enable a chain-of-thoughts mechanism. When the agent makes a request to the local/remote LLM[3] and receives a response that contains the JavaDoc-style match, it will consider it for next context inclusion.

Figure 4 represents an example of classes that are dependent due to method calls. These method calls then create a call graph shown in Figure 5. We assume it as a tree, meaning circular method calls are not allowed as best practices in software development. This graph is then traversed DFS to enable context-preparation prior to making the call for parent methods. In this way, for a given method, when requested to generate a comment for it, all its descendants have an explanation already. This is a key process that enables offering a RAG-powered solution.

[3]In this paper, when we call remote or local LLM, it refers to the type of deployment of that model. We anticipated this and use appropriate type of API to make the calls.

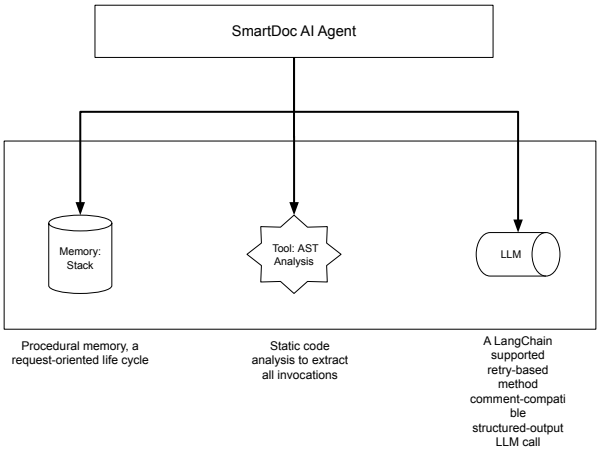

Figure 3: SmartDoc agent component. It comprises the system core, acting as key runner for producing precise comment.

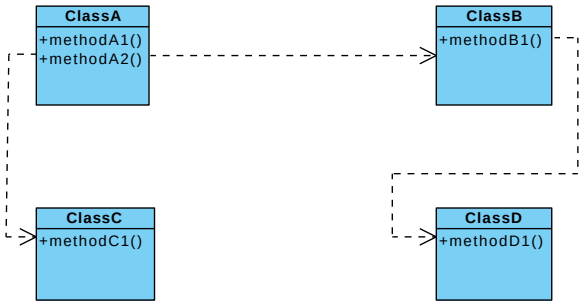

Figure 4: An example of classes that are coupled via method calls. Assume methodA1 in ClassA has a call to methodA2

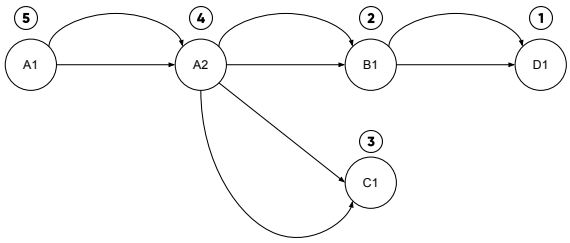

Figure 5: Call graph and the order of visiting each method, according to the class diagram shown in Figure 4

SmartDoc is working for all the application-level methods definition. We assume for all public third-party APIs, target LLM understand its explanation, as it has been trained over those publicly available data. This is a powerful feature of LLMs as they have been

trained over millions of data records and input texts, covering a super wide range of publicly available programming languages, SDKs and third-party libraries. Therefore, no matter how many and in what way third party methods or the programming language core are used, the remote AI can understand and provide interpretation. At the end, it is only project-internal method calls that are requested to generate new explanation for completing the context.

3.2 Dealing with structured output

To maximize LLM efficiency in generating responses, it is required to have structured output compatible to the initial requirements. The final polished output should be a JavaDoc style compatible text, enabling a valid (re)placement for current method comment. A simple and straightforward solution is instructing through prompts for the required structured output. LLMs can be provided with an engineered prompt to generate these compatible comments (i.e., instructing via system and user messages in request submission).

During the implementation, we had tow approaches to follow: (I) Using a retry-based method: defining a max retry count and initiate next request in case of invalid response. (II) Using a few-shot learning, via injecting specific structure to fine-tune the LLM for only responding valid comments.

For this version, a simple retry-based approach was selected. It works fine when the model owns remarkable number of parameters and is generalized for a wide range of use cases. For instance, the DeepSeek remote model responds remarkably accurately before reaching the max-try-count. In addition, to avoid any inconsistencies, if the response partially contains the desired output, we apply a predefined regex on the response to ensure valid comment replacement.

4 Results and evaluations

We developed an application, delivering it as a IntelliJ IDEA plugin, dedicated to Java codebase. Results in our system means every comment generated for a given method. This comment text is completely compatible with JavaDoc style standard. The method comment is applicable when a developer wants to grasp required knowledge to make a change to the project. Its effectiveness can be recognized when we can evaluate its effect in several ways, ensuring its true impact on the development process.

The current system has a non-deterministic, variable nature, stemming from the AI model which generates comments for the input method. It means, each time a request is sent a valid and still different response is received, making deterministic test case design difficult. This enforces traditional testing useless and needs endeavors for a new test and evaluation design. All this leads us to focus on different metrics which measures response quality (e.g., relevance, coherence, etc.), as well as deterministic-metrics, such as precision, accuracy and F1. These metrics can be available using open source implementations that curated for such these purposes.

4.1 Implementing the evaluation set-up

To conduct the evaluations, given the current IDE plugin, SmartDoc, we organize the evaluation process in terms of several test cases. A parameterized test case that is developed to run against different sub modules of the Eclipse Ditto project[4]. We have selected this project since it has a great maintenance and involves implementation with correct, humani integrated method comments. This test case is in addition to other test cases developed during the development of SmartDoc, as we used various techniques to ensure a maintainable software.

The evaluation of the final output is intended to be done at two levels: First, to compute how accurate generated comment are in terms of three key metrics. Second, using a feedback system that receive users' opinion for each generated comment. Although, this feedback data is expected to be available after first beta release of the plugin.

4.1.1 Hallucinations (invalid responses). Hallucinations refer to invalid or factually incorrect responses generated by an LLM for a given request. To measure this effect, we can use three commonly adopted community metrics: BERTScore, ROUGE-1, and BLEU [6]. Although these metrics are not specifically designed to evaluate hallucinations, they provide indicative values that can help assess this phenomenon. These metrics compute the degree of similarity between the generated output (referred to as the actual) and the human-produced reference output (the expected), based on semantic, exact, and partial token matches.

BERTScore: BERTScore measures semantic similarity between actual and expected outputs, for each given method, using comparison-pair contextual embedding. Figure 6 compares the computed metrics for precision, recall and F1 categories for 5 packages of Eclipse Ditto open source project (applied to minimum of 10 methods per package).

BLEU: This metric is often used for machine translation use cases. However, since our comparison goal has many features in common with those models, we decided to report for this metric as well. Figure 7 represents the final result for those five packages.

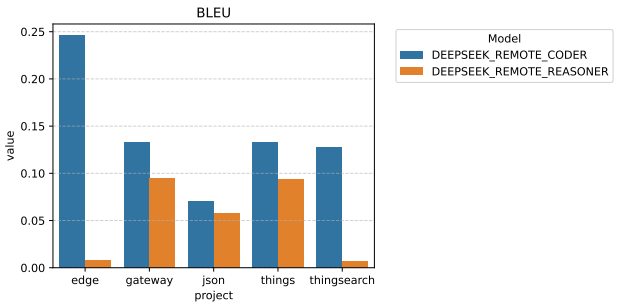

Figure 7: BLUE metric obtained from comparing SmartDoc with human wrote comments using *sacrebleu package*

ROUGE-1 score: This metric computes the ratio of uni-grams of actual text (from generated comment) that appear in the expected (from ground truth). This metric specifically counts number of tokens that is common between actual and expected outputs. Figure 8 lists all comparisons in terms of precision, recall and F1, again for those five packages with minimum 10 methods.

[4]https://eclipse.dev/ditto/

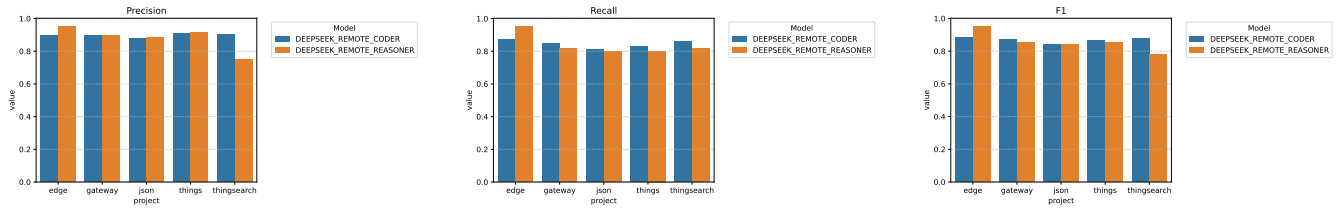

Figure 6: BERTScore metric comparing SmartDoc output with the ground truth

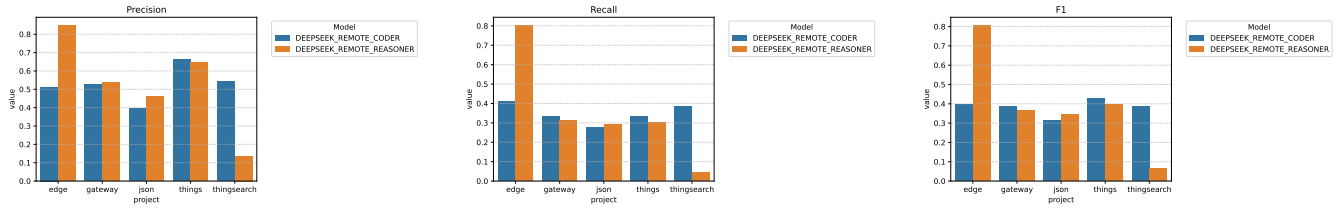

Figure 8: ROUGE-1 metric comparing SmartDoc output with the ground truth

4.1.2 Feedback collection (satisfied end-user). We required to devise an approach to enable user feedback collection. For this purpose, once a user receives his generated comment (upon the request initiated), the plugin asks for a rate and it is quite optional for the user to send the feedback. In case of feedback submission, none of user's private information–at any level–is collected, and only the rate and any text provided saved. This key data can help us to conduct a qualitative study and extend the evaluation. This analysis actually takes place after a collection-period done, and can enable us to measure how satisfied the end-users are in general.

5 Discussion and conclusion

We can observe promising results regarding similarity metrics. BERTScore report is suggesting a range of 0.80 to 0.90 in terms of precision, recall and F1. ROUGE-1 score, as a partial n-gram comparison, has an average 0.40 to 0.50 in terms of similar metrics and BLEU, as an exact token based similarity measure offers a value of maximum of 0.25. This drop in accuracy when moving from BERTScore to BLEU is normal, as it transitions from a semantic similarity measure to exact token matches.

In addition to what evaluated so far, there is a direct relationship between the accuracy of the remote LLM model used and the eventual comments generated. It means, using a more comprehensive model as the comment generator, we expect better results, and more satisfaction rate. This is extra to the impact of useful context provided when a call to LLM takes place, meaning a more relevant context can increase the final response relevance.

This plugin is evolvable, meaning there are still new features that can be added while preserving its current reliable functionality. For instance, support for other languages, plus adding new models to offer the end-users more options. This plugin enables end-users concurrent comment generation, allowing them to submit multiple requests in a specific time window. It has minimum interference with text editor events and the users can continue their programming tasks while awaiting responses for comment provision.

We will invest on chain-of-thoughts [15] in more details in the new updates. In this way, the agent breaks down the engineered prompt preparation given all nested method calls. In every single step, it can reason and select best explanation for the callees. It also can be supported by a reasoner to offer its best for context preparation and eventual comment generation.

As mentioned, we enabled a feedback system, allowing users to anonymously submit us their feedback. This feedback by now allows a five stars rates along with the *LLM model* user selected for his request. We may decide and offer an agreement asking user to collect the method metadata as well in the newer updates. It can be effective for analyzing how our method acts in different situation (e.g., number of nested method calls, method length, duration of comment generation, resource utilization)

In this plugin, we focus on static analysis of codebase, only visiting the current snapshot of the codebase. This can cover methods overloaded and explicit calls. However, the method visiting to create call graph becomes more challenging when programmers use specific patterns or design principles that require method overriding or runtime method dispatching. We plan to cover this feature in the upcoming updates.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[2] Suborno Deb Bappon, Saikat Mondal, and Banani Roy. 2024. Autogenics: Automated generation of context-aware inline comments for code snippets on programming q&a sites using llm. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 24–35.

[3] Vahid Etemadi, Omid Bushehrian, and Gregorio Robles. 2022. Task assignment to counter the effect of developer turnover in software maintenance: A knowledge diffusion model. *Information and software technology* 143 (2022), 106786.

[4] Yuan Huang, Yinan Chen, Xiangping Chen, and Xiaocong Zhou. 2025. Are your comments outdated? Toward automatically detecting code-comment consistency. *Journal of Software: Evolution and Process* 37, 1 (2025), e2718.

[5] Yuan Huang, Hanyang Guo, Xi Ding, Junhuai Shu, Xiangping Chen, Xiapu Luo, Zibin Zheng, and Xiaocong Zhou. 2023. A comparative study on method comment and inline comment. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–26.

[6] Abhishek Kumar, Sandhya Sankar, Partha Pratim Das, and Partha Pratim Chakrabarti. 2025. Using Large Language Models for multi-level commit message generation for large diffs. *Information and Software Technology* 187 (2025), 107831.

[7] Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient Fine-tuning for LLM-based Recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*. 365–374.

[8] Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. 2024. " we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–9.

[9] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (2014), 1–37.

[10] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[11] Tahira Niazi, Teerath Das, Ghufran Ahmed, Syed Muhammad Waqas, Sumra Khan, Suleman Khan, Ahmed Abdelaziz Abdelatif, and Shaukat Wasi. 2023. Investigating novice developers' code commenting trends using machine learning techniques. *Algorithms* 16, 1 (2023), 53.

[12] Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. 2025. Agentic large language models, a survey. *arXiv preprint arXiv:2503.23037* (2025).

[13] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. 2024. Future of software development with generative AI. *Automated Software Engineering* 31, 1 (2024), 26.

[14] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959* (2024).

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[16] Derek Xu, Tong Xie, Botao Xia, Haoyu Li, Yunsheng Bai, Yizhou Sun, and Wei Wang. 2024. Does Few-Shot Learning Help LLM Performance in Code Synthesis? *arXiv preprint arXiv:2412.02906* (2024).