

Exploring Large Language Models for Analyzing and Improving Method Names in Scientific Code

Gunnar Larsen
University of Hawai'i at Mānoa
Hawai'i, USA
gunnarl@hawaii.edu

Carol Wong
University of Hawai'i at Mānoa
Hawai'i, USA
carolw8@hawaii.edu

Anthony Peruma
University of Hawai'i at Mānoa
Hawai'i, USA
peruma@hawaii.edu

Abstract—Research scientists increasingly rely on implementing software to support their research. While previous research has examined the impact of identifier names on program comprehension in traditional programming environments, limited work has explored this area in scientific software, especially regarding the quality of method names in the code. The recent advances in Large Language Models (LLMs) present new opportunities for automating code analysis tasks, such as identifier name appraisals and recommendations. Our study evaluates four popular LLMs on their ability to analyze grammatical patterns and suggest improvements to 496 method names extracted from Python-based Jupyter Notebooks. Our findings show that the LLMs are somewhat effective in analyzing these method names and generally follow good naming practices, like starting method names with verbs. However, their inconsistent handling of domain-specific terminology and only moderate agreement with human annotations indicate that automated suggestions require human evaluation. This work provides foundational insights for improving the quality of scientific code through AI automation.

Index Terms—Program Comprehension, Method Names, Jupyter Notebooks, Grammar Patterns, Part-of-Speech

I. INTRODUCTION

As Large Language Models (LLMs) grow in popularity, their application in software engineering is becoming increasingly significant [1]. These models assist developers with various tasks, such as code generation, debugging, code review assistance, and documentation [2]–[5]. Additionally, through the use of IDE plugins like GitHub Copilot¹ and specialized AI-integrated IDEs such as Cursor² and Replit³, developers can easily leverage LLMs to work faster and more efficiently.

Despite these advancements, there are also drawbacks associated with using AI-powered tools in software engineering [6]. Code quality issues, including security vulnerabilities, potential biases in the generated code, and hallucinations, lead to developers losing trust in AI-generated code, as they often need to double-check and correct the output [7], [8]. Additionally, crafting prompts can be time-consuming, and latency or interruptions may disrupt workflow. Moreover, there are also concerns regarding the ethical and legal implications of AI-generated content [9]–[11].

While software engineers have the necessary education and experience to effectively utilize these LLM tools and

evaluate their output, individuals without a software engineering background may inadvertently use code generated by LLMs without fully understanding how it could affect the overall quality of the codebase and future maintenance efforts. This includes research scientists, who often need to develop software programs to support their research. Typically, scientists working in non-computer science research domains learn programming through on-the-job training or from informal resources like online articles and blogs, which can lead to a lack of understanding regarding the principles of writing readable and maintainable code [12].

Prior research into the code typically written by research scientists reveals issues such as PEP8 violations, stylistic inconsistencies, high coupling, and significant challenges in reproducibility [13]–[16]. Furthermore, Wong et al. [17] examined the method names in such code and found many instances where names deviated from standard naming practices, such as starting with a verb. Instead, these names tend to be based on the output of the methods, rather than their actions. The authors also noted the presence of ambiguous single-term names, unconventional word order, and methods that include abbreviations or acronyms, many of which require domain-specific knowledge for interpretation. They also observed instances where the names incorrectly ended with a verb.

A. Goal and Research Questions

In this context, our exploratory study aims to investigate the ability of LLMs to evaluate method names in scientific code and to generate alternative naming suggestions when appropriate. The naming conventions observed in the code written by research scientists frequently diverge from standard practices, leading to reduced clarity and maintainability. Method names, in particular, play an important part in conveying the purpose and functionality of code, and poorly chosen names can result in confusion and hinder collaboration among researchers [18]–[20]. By utilizing the capabilities of LLMs, we seek to determine how effectively these models can understand context and propose method names that adhere to established conventions and best practices. We envision our findings advancing the knowledge of identifier naming in scientific code and the effectiveness and drawbacks of LLMs in generating improved naming suggestions. Additionally, it offers practical guidelines for non-computer scientists when

¹<https://github.com/features/copilot>

²<https://cursor.com/>

³<https://replit.com/>

using LLMs for code generation and evaluation. We address the following research questions (RQs):

RQ1: How effectively can LLMs identify and classify grammatical patterns in method names? Method names in scientific code typically do not follow standard software engineering naming conventions. This RQ aims to explore the effectiveness of LLMs in generating part-of-speech tags for these method names. This initial understanding will help us determine the reliability of LLMs as tools for analyzing the grammatical structure of method names in scientific code.

RQ2: To what extent are LLM-suggested method name corrections aligned with software engineering best practices? In this RQ, we aim to evaluate the quality and appropriateness of method name improvements suggested by LLMs for scientific code, including how they handle abbreviations and acronyms contained in the name. Insights gained from this exploration will assist us in developing better tools and techniques to improve code quality in notebook environments.

B. Contributions

The main contributions of this work are:

- **Evaluation of LLMs:** Assesses the ability of LLMs to analyze grammatical patterns in method names within scientific programs and suggest corrections aligned with software engineering best practices.
- **Advancing Code Quality:** Contributes to improving the quality of scientific programs, thereby enhancing their maintainability and reproducibility.

II. METHODOLOGY

In this section, we describe in detail our approach to answering our RQs. Figure 1 depicts an overview of the key activities in our methodology, which we elaborate below.

A. Source Dataset

In this study, we utilize the dataset made available by Wong et al. [17]. In their study, the authors manually reviewed 691 method names extracted from 384 Jupyter Notebooks, which were randomly sampled from a larger dataset of 847,881 Python-based Jupyter Notebooks, which were collected from public GitHub repositories by Grovov et al. [14]. The authors (i.e., Wong et al.) annotated these method names for grammatical patterns, using part-of-speech tags, defined by Newman et al. [21], specifically tailored for analyzing source code identifiers. This dataset is the only one that has part-of-speech annotated method names from scientific code.

B. Part-of-Speech Tags

These part-of-speech tags utilized by Wong et al. include nouns (N), noun modifiers (NM) for adjectives and noun adjuncts, verbs (V), verb modifiers (VM), prepositions (P), determiners (DT), conjunctions (CJ), pronouns (PR), digits (D), and preambles (PRE). Details about the tags are available in the study by Newman et al. [21] and online at [22].

C. Method Extraction

For each of the 691 method names in the source dataset, we extracted the complete method definition from its respective notebook file. We utilized the `nbformat` Python package to parse the notebook and extract the method code.

D. Large Language Model Analysis

For this study, we utilized four popular Large Language Models (LLMs). Each of these LLMs has been shown by its respective authors to demonstrate significant capabilities in understanding and generating code-related tasks. The models we selected are:

- Google Gemini 2.0 Flash⁴
- Alibaba Qwen 2.5 Coder 32B⁵
- Meta LLaMa 3.3 70B⁶
- DeepSeek-R1 70B⁷

We built custom scripts to interact with the Gemini model using Google’s Gemini API. For the remaining three models, we utilized the Ollama⁸ platform to run the LLMs locally.

E. Prompt Design

The effectiveness of LLM-based analysis depends on carefully crafted prompts, as their phrasing, structure, and context influence output quality and consistency [23]. Hence, we evaluated multiple variations of the prompt to determine the most effective. Our evaluation included prompts with different roles, levels of detail, various examples, and alternative formatting approaches. We found that prompts with clear structural guidelines and explicit output formatting requirements produced the most consistent results across different LLM architectures. Our final prompt design includes the following elements:

- A role specification for the LLM as “an expert software engineer specializing in Python programming”
- A detailed explanation of the part-of-speech tagging process, including a comprehensive table of tags (N, V, NM, VM, P, etc.) with examples
- Instructions for splitting method names into individual terms based on camelCase, PascalCase, or snake_case
- Guidelines for handling acronyms, abbreviations, and numerical elements in method names
- Explicit directions for evaluating the quality of method names against software engineering best practices
- A structured JSON response format with fields for:
 - Current method name and its grammar pattern
 - Corrected method name (if needed)
 - Corrected grammar pattern (if needed)
- An example of the output structure
- The code associated with the method

We observed the need to provide explicit instructions regarding the part-of-speech tagging, as without such explicit

⁴<https://ai.google.dev/gemini-api/docs/models#gemini-2.0-flash>

⁵<https://ollama.com/library/qwen2.5-coder>

⁶<https://ollama.com/library/llama3.3>

⁷<https://ollama.com/library/deepseek-r1>

⁸<https://ollama.com/>

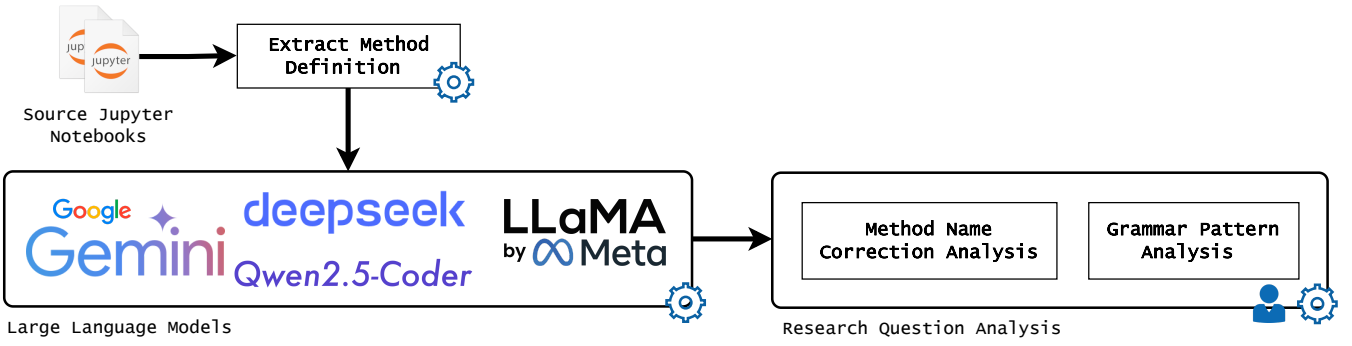


Fig. 1. KEY ACTIVITIES IN OUR METHODOLOGY.

TABLE I
ACCURACY AND AGREEMENT RATES BETWEEN EACH LLM AND THE HUMAN ANNOTATION (GROUND TRUTH)

Model	Accuracy	Cohen's Kappa
Gemini 2.0 Flash	0.611	0.577
Qwen 2.5 Coder 32B	0.454	0.412
DeepSeek-R1 70B	0.393	0.344
LLaMa 3.3 70B	0.357	0.295

guidance, the LLMs default to standard natural language part-of-speech tags, which are inadequate for analyzing identifier names in code. This approach ensures that the LLMs use a consistent tagging scheme and facilitates ease of comparison against the human-annotated tags in the source dataset.

F. Research Question Analysis

The output generated from the LLMs was saved to a SQLite database in a table whose fields correspond to the output structure specified in the prompt.

For our analysis, we employed standard statistical measures to effectively summarize the data. We also wrote SQL queries and custom code to retrieve information from the database to address our RQs. Additionally, we manually searched the database to include representative examples that would enhance our analysis. To evaluate the level of agreement among the LLMs and between the original annotations, we utilized Fleiss' Kappa and Cohen's Kappa. Both of these measures assess inter-rater reliability, but they differ in their application: Cohen's Kappa is appropriate for situations involving two raters, while Fleiss' Kappa is suitable for three or more raters. Artifacts for this study are available at [24].

III. RESULTS

In this section, we answer our RQs by analyzing the output of the LLMs. It is important to note that although the prompts used for the LLMs were identical, not all models produced the desired output. In some cases, certain LLMs either did not adhere to the instructions and instead returned lengthy

TABLE II
THE TOP THREE MOST COMMON GRAMMAR PATTERN MISCLASSIFICATIONS FOR EACH LLM.

Human Annotated Grammar Pattern	Model Generated Grammar Pattern	Count	Percentage
Model: <i>Gemini 2.0 Flash</i>			
N	PRE	16	8.29%
VM	V	11	5.70%
V,NPL	V,N	9	4.66%
Model: <i>Qwen 2.5 Coder 32B</i>			
V,N	V,NM	42	15.50%
V,NPL	V,NM	15	5.54%
v	N	13	4.80%
Model: <i>DeepSeek-R1 70B</i>			
V,NPL	V,N	18	5.64%
N	V	13	4.08%
NM,N	N,N	13	4.08%
Model: <i>LLaMa 3.3 70B</i>			
N	V	15	4.98%
V,N	V	14	4.65%
V,N	V,NM	12	3.99%

paragraphs or followed the required output format but failed to provide values for all fields in the JSON structure. Additionally, there were also instances of the model hallucinating, generating output for methods that did not exist. Out of a total of 691 input methods, the Gemini model successfully returned complete outputs for 690 of them. In comparison, the LLaMA model produced details for 615 methods, the Qwen model for 595 methods, and DeepSeek for 526 methods. However, as mentioned above, some outputs from these models are hallucinations. After examining the outputs from each of the LLMs, we identified a total of 496 methods that contained valid outputs common to all four models. This number reflects our decision to exclude methods that were only returned by

some models, ensuring consistency in the results we analyzed. Henceforth, our analysis is based only on these 496 methods.

RQ1: *How effectively can LLMs identify and classify grammatical patterns in method names?*

This RQ examines the part-of-speech tags generated by the LLMs for the method names and compares these tags against the human-annotated tags in the original dataset. First, when assessing the level of agreement among the LLMs, we find that all four models agree in 18.8% of cases, resulting in a Fleiss’ Kappa of 0.333. This indicates a fair level of agreement among the models. However, comparing the output solely across LLMs serves only to shed light on AI behavior, not to validate correctness. Therefore, we also compared the output of each LLM against the human-annotated tags in the original dataset. In Table I, we show the accuracy and agreement of the part-of-speech tags for each of the four LLMs. Even though Gemini had the highest accuracy, 61.1%, the Cohen’s Kappa of 0.577 is indicative of a moderate agreement.

Our next analysis focuses on the common grammar patterns that each LLM misclassified. Table II shows the top three most common grammar patterns each LLM misclassified. From this, we observe that Gemini struggles with single-term names that are acronyms. The $N \rightarrow PRE$ misclassification affecting methods having names like “SVD” and “MSE” reveals that Gemini incorrectly categorizes mathematical and technical abbreviations as preambles.

Qwen tends to misclassify nouns as noun modifiers ($V,N \rightarrow V,NM$ and $V,NPL \rightarrow V,NM$). Unlike a noun, a noun modifier adds specificity to the noun by providing more detail and clarifying the particular type of object to which the action relates. Hence, it is interesting that even though method names like “load_image” are composed of only a verb and a noun, Qwen often classifies the noun as a noun modifier. This misclassification occurs despite the fact that in this context, “image” acts as a direct object of “load,” not a modifier.

The analysis also reveals challenges with words that can function as both nouns and verbs. Qwen classifies single-word method names “clean” and “animate” as nouns, while the human-annotated dataset has them listed as nouns. In contrast, Deepseek and LLaMA incorrectly classify certain human-annotated nouns, such as “sigmoid” and “answer”, as verbs. These dual-function words require understanding of both method naming conventions and contextual behavior, not just linguistic rules, for accurate classification.

An interesting observation regarding certain models is how they classify plural words. For example, human annotations for method names like “build_data” and “process_features” categorize them as V,NPL , meaning a verb followed by a plural noun. However, models such as Gemini, Qwen, and Deepseek classify the plural term as singular (i.e., V,N).

Summary. RQ1 examines how effectively LLMs classify grammatical patterns in method names from scientific code. Among the four models, Gemini achieved the highest accuracy and moderate agreement with human annotations. LLaMa

TABLE III
NUMBER OF TIMES EACH LLM RETAINED THE ORIGINAL METHOD NAME.

Model	Preserved Original Name	
	Count	Percentage
Gemini 2.0 Flash	324	65.3%
Qwen 2.5 Coder 32B	160	32.3%
DeepSeek-R1 70B	136	27.4%
LLaMa 3.3 70B	35	7.1%

TABLE IV
LENGTH OF METHOD NAMES GENERATED BY THE LLMs.
THE VALUES WITHIN PARENTHESES REPRESENT THE GROWTH RELATIVE TO THE ORIGINAL METHOD NAME.

Model	Average		Average	
	Words Per Method Name		Characters Per Method Name	
	Original Method	Corrected Method	Original Method	Corrected Method
Gemini 2.0 Flash	1.78	2.77 (+55.62%)	10.86	17.97 (+65.47%)
DeepSeek-R1 70B	1.94	2.96 (+52.58%)	11.29	19.96 (+76.79%)
Qwen 2.5 Coder 32B	1.87	2.99 (+59.89%)	10.88	19.77 (+81.71%)
LLaMa 3.3 70B	1.98	3.29 (+66.16%)	11.71	22.62 (+93.17%)

has the lowest accuracy and agreement. Common misclassifications included acronyms, plural nouns, and dual-function words, highlighting challenges with contextual and domain-specific details.

RQ2: *To what extent are LLM-suggested method name corrections aligned with software engineering best practices?*

While the prior RQ focused on the grammar patterns generated by LLMs for the original practitioner-crafted method name, this RQ focuses on the extent to which LLMs evaluate the quality of the original name and suggest a correction.

First, we analyze the agreement between the four LLMs by comparing the corrected names provided by each LLM. We observe that only 78 out of 496 corrected names (about 15.7%) are identical across all four LLMs. Further, the level of agreement among the four LLMs is measured by a Fleiss’ Kappa value of 0.389, indicating a fair level of consensus.

Next, we examined the extent to which each LLM agreed with the original method name (i.e., the suggested corrected name was the same as the original name). As shown in Table III, Gemini showed the strongest preference for keeping original names unchanged, preserving about two-thirds (65.3%) of the names. In contrast, LLaMA was the most aggressive in proposing alternative names, with approximately 93% of its suggestions being different from the original.

Moving on, we next examine the structural characteristics of the suggested method names. We perform this analysis exclusively on the method names that were changed by each

LLM. Table IV shows the findings of this analysis. Since the number of method names corrected varied across each LLM, the values in the column labeled *Original Method* represent the average counts for only the original method names specific to each LLM. We observe that all four models tend to lengthen the method name. LLaMA produced the longest method names, with approximately 66% more words and 93% more characters than the original names on average. In contrast, Gemini shows the least growth of the corrected method name. Deepseek and Qwen contribute method names that are typically around 20 characters long and comprise approximately three words for their suggested renames.

Our next set of analyses explored the semantic characteristics of the names. We first start by investigating the common terms that each LLM adds to and removes from the original name when it suggests an alternate name. Table V shows the top two frequently added and removed terms for each LLM⁹. Two observations we notice in all four LLMs are as follows: (1) the term “calculate” dominates all four LLMs as the most added term, highlighting the unique characteristic of scientific code, and (2) the expansion of the acronym “mse”. Another interesting observation is the replacement of the term “get” with a synonym like “extract” and “retrieve” by most LLMs.

Examining the part-of-speech tags generated by the LLMs for the corrected names, we found that the majority of grammar patterns produced by all four LLMs begin with a verb. Specifically, 97.98% of the grammar patterns from LLaMA, 96.37% from Qwen, 95.56% from DeepSeek, and 85.69% from Gemini start with a verb. In contrast, approximately 55% of the patterns in the original dataset begin with a verb.

Further, as part of the corrected name grammar pattern analysis, we observed a discrepancy between the number of words in a method name and the number of part-of-speech tags in the grammar pattern. From Table VI, we observe that Gemini shows the highest consistency in matching part-of-speech tags to words, with a rate of 95.77%. In contrast, LLaMA has the lowest matching rate (69.15%) and generates fewer part-of-speech tags than there are words in the method name. For instance, LLaMA suggests renaming “pcr_noise” to “apply_pcr_noise_model,” which contains four words, but it generates only two tags (V,NM) for the corrected name. On the other hand, DeepSeek tends to assign more part-of-speech tags than words. For example, while the model suggests renaming “update” to “train_model,” the new name is matched with a grammar pattern of three tags (V,N,N) rather than two.

Our final analysis examines how the LLMs handle method names containing abbreviations/acronyms. The original dataset of 496 method names contains 140 instances of such terms.

First, we examine how many of these 140 methods were not corrected (i.e., the LLM did not propose an alternative). We observe that Gemini preserved the original name in 53 instances (approximately 37.9%). In comparison, Qwen, DeepSeek, and LLaMA preserved the original name 8 times (5.7%), 6 times (4.3%), and 1 time (0.7%), respectively.

TABLE V
THE TWO TERMS MOST FREQUENTLY ADDED AND REMOVED BY EACH LLM IN THE CORRECTED METHOD NAME.

Action	Term	Count	Example
Model: <i>Gemini 2.0 Flash</i>			
Added	calculate	51	variance → calculate_variance
Added	image	12	pdiguide_imgRead → read_pdiguide_image
Removed	mse	8	MSE → calculate_mean_squared_error
Removed	im	7	im_convert → convert_image
Model: <i>Qwen 2.5 Coder 32B</i>			
Added	calculate	86	square → calculate_square
Added	get	21	author_url → get_author_url
Removed	mse	15	MSE → calculate_mean_squared_error
Removed	get	12	get_params → initialize_parameters
Model: <i>DeepSeek-R1 70B</i>			
Added	calculate	55	pcr_noise → calculate_pcr_noise
Added	compute	20	_hash → compute_hash
Removed	get	21	get_dataset → retrieve_dataset
Removed	mse	15	MSE → mean_squared_error
Model: <i>LLaMa 3.3 70B</i>			
Added	calculate	91	square → calculate_square
Added	perform	27	preprocess → perform_preprocessing
Removed	get	37	get_year → extract_year_from_name
Removed	mse	15	MSE → calculate_mean_squared_error

TABLE VI
CONSISTENCY OF WORD COUNTS AND PART-OF-SPEECH TAGS FROM LLMs FOR THE CORRECTED METHOD NAME.

Model	Equal	More	Fewer
	Tags and Words	Tags than Words	Tags than Words
Gemini 2.0 Flash	475 (95.77%)	19 (3.83%)	2 (0.40%)
DeepSeek-R1 70B	384 (77.42%)	30 (6.05%)	82 (16.53%)
Qwen 2.5 Coder 32B	442 (89.11%)	22 (4.44%)	32 (6.45%)
LLaMa 3.3 70B	343 (69.15%)	20 (4.03%)	133 (26.81%)

TABLE VII
COUNT OF ABBREVIATIONS AND ACRONYMS EXPANDED BY EACH LLM.

Model	Abbreviations & Acronyms	
	Not Expanded	Expanded
Gemini 2.0 Flash	88 (62.86%)	52 (37.14%)
Qwen 2.5 Coder 32B	58 (41.43%)	82 (58.57%)
DeepSeek-R1 70B	42 (30%)	98 (70%)
LLaMa 3.3 70B	38 (27.14%)	102 (72.86%)

⁹Due to space constraints, we limited the table to two terms for each action.

Next, from Table VII, we note that Gemini is more conservative and expands only about 37.14% of the abbreviations and acronyms. In contrast, LLaMA is more aggressive, expanding 72.86%. Some observations we notice are that certain computing abbreviations, such as ‘CSV’, ‘JSON’, ‘ASCII’, and ‘PNG’ are not expanded by any of the LLMs. Similarly, we find that some domain-specific abbreviations, like ‘PCR’ (Polymerase Chain Reaction) ‘FES’ (Filter Encoding Standard), and ‘WISDM’ (Wireless Sensor Data Mining), are not expanded by all four LLMs. In contrast, abbreviations like ‘MSE’ (Mean Squared Error) and acronyms like ‘ACC’ (Accuracy) are expanded by most of the LLMs.

Summary. RQ2 assesses LLMs’ ability to suggest method name corrections that adhere to best practices. Gemini retained 65.3% of original names, while LLaMA changed 93%. Corrected names were longer and often started with verbs, with common additions included terms like “calculate.” LLaMA is the most aggressive among the LLMs in expanding abbreviations and acronyms. However, computing and domain-specific abbreviations, such as “CSV” and “PCR” were often not expanded in the suggested name.

IV. DISCUSSION

As an exploratory study, our findings show that while LLMs can provide valuable guidance for improving scientific code readability, their suggestions require careful human evaluation, particularly in domain-specific contexts.

A clear observation from our RQ results is the unique behavior of the individual LLMs in our evaluation. Specifically, there are significant differences in how Gemini and LLaMA generate grammatical patterns and assess the quality of names. This shows that different architectural approaches and training methodologies can lead to fundamentally different code analysis philosophies. Moreover, the challenges with the LLMs in handling domain-specific terminology show that the assumptions built into current LLM architectures regarding code quality may not align with the needs of scientific computing. This highlights the need for explicit domain-aware training approaches instead of relying on general-purpose language models for specialized code analysis tasks.

The results we achieved are a result of the prompt we created, which was developed through multiple iterations and an in-depth understanding of software engineering principles and the behavior patterns of LLMs. It is important to acknowledge that research scientists who lack a background in software engineering are unlikely to replicate our level of success. While they might apply basic prompts like “improve this method name” or “make this code better,” these simplistic approaches may yield poorer results than our findings. This potential gap between our controlled experimental conditions and real-world usage scenarios is a concern. Hence, there is a need to make LLMs more accessible and effective for users across varying levels of technical expertise.

Key Implications For SE Research Community. Our work advances the field of AI for SE by providing an initial evaluation of LLMs for improving program comprehension

in scientific code. Our early results highlight several opportunities for further research in this area, such as domain-specific fine-tuning approaches for LLMs tailored to specific scientific disciplines and conducting more human-centric studies to understand how research scientists utilize LLMs.

Key Implications For Research Scientists. While non-computer science researchers may be inclined to rely entirely on LLMs to write code that supports their research, our findings provide guidance on when to trust LLM suggestions and when human judgment is essential. Thus, helping them make more informed decisions regarding adopting AI coding assistants. Scientists can utilize these insights to better evaluate code improvements generated by LLMs, especially as domain-specific terminology often needs human review for accuracy.

Key Implications For Educators. When teaching programming concepts to students from non-traditional computer science backgrounds, educators should emphasize both the advantages and limitations of using AI tools for software engineering. Helping students understand when and why LLMs may fail in code analysis tasks is essential for developing their critical evaluation skills in AI-assisted development.

V. THREATS TO VALIDITY

The prompt’s quality can significantly impact LLM output, potentially affecting our findings. Additionally, since we applied the same prompt across all four LLMs, there is a risk that this prompt may not have been ideal for every model.

As LLMs continue to evolve, the results observed in our study may not reflect the capabilities of future models. Our findings should therefore be considered as a snapshot of current LLM capabilities rather than an assessment of their potential for code analysis tasks. Additionally, due to the non-deterministic nature of LLM outputs, there is a threat to reproducing our results. Furthermore, the selected LLMs may not represent all available models, which could also limit the applicability of our results. However, since this is an exploratory study, our goal is to identify preliminary trends and patterns that can guide future research.

Our dataset is limited to a subset of methods found in Python-based Jupyter Notebooks, which restricts the generalizability of our findings. We also acknowledge that method names were analyzed in isolation, without additional context such as class names, file names, or usage patterns. While this simplifies the analysis and reduces prompt complexity, it may limit the relevance or precision of the suggested names, especially in cases where meaning is derived from surrounding structures. Finally, we did not include human evaluation of method name corrections generated by LLMs, limiting our assessment of semantic appropriateness. Expert feedback could enhance the evaluation’s real-world applicability.

VI. CONCLUSION & FUTURE WORK

This exploratory study examines the effectiveness of LLMs in analyzing and improving method names in scientific code. Our evaluation of four popular LLMs on 496 method names

shows notable differences in LLM performance for grammatical pattern recognition and method name improvements. Gemini achieved the highest accuracy in recognizing grammatical patterns, but a moderate agreement with human annotations. For method name corrections, Gemini was conservative and preserved most of the original names, while LLaMA made more aggressive changes. All models showed a bias toward names that begin with a verb, aligning with software engineering practices. As an initial investigation into the use of LLMs for improving program comprehension in scientific code, our findings establish a foundation for future research in this area. Our future work will include other identifier types, such as variables, parameters, and classes, for a comprehensive understanding of identifier quality in scientific code.

VII. ACKNOWLEDGMENTS

The technical support and advanced computing resources from University of Hawaii Information Technology Services – Research Cyberinfrastructure, funded in part by the National Science Foundation CC* awards # 2201428 and # 2232862 are gratefully acknowledged. Additionally, this work was partially supported by the Undergraduate Research Opportunities Program in the Office of the Vice Provost for Research and Scholarship at the University of Hawai‘i at Mānoa.

REFERENCES

- [1] C. Ebert and P. Louridas, “Generative ai for software practitioners,” *IEEE Software*, 2023. 1
- [2] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, p. 1–79, Nov. 2024. 1
- [3] J. T. Liang, C. Yang, and B. A. Myers, “A large-scale survey on the usability of ai programming assistants: Successes and challenges,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE ’24, (New York, NY, USA), Association for Computing Machinery, 2024. 1
- [4] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekk, and D. Doermann, “Future of software development with generative ai,” *Automated Software Engineering*, vol. 31, Mar 2024. 1
- [5] L. Belzner, T. Gabor, and M. Wirsing, *Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study*, p. 355–374. Springer Nature Switzerland, Dec. 2023. 1
- [6] C. Gao, X. Hu, S. Gao, X. Xia, and Z. Jin, “The current challenges of software engineering in the era of large language models,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, p. 1–30, May 2025. 1
- [7] H. Mozannar, G. Bansal, A. Fournay, and E. Horvitz, “Reading between the lines: Modeling user behavior and costs in ai-assisted programming,” in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI ’24, p. 1–16, ACM, May 2024. 1
- [8] S. Lertbanjongngam, B. Chinthanet, T. Ishio, R. G. Kula, P. Leelaprute, B. Manaskasemsak, A. Rungsawang, and K. Matsumoto, “An empirical evaluation of competitive programming ai: A case study of alphacode,” in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, p. 10–15, IEEE, Oct. 2022. 1
- [9] C. Bull and A. Kharrufa, “Generative artificial intelligence assistants in software development education: A vision for integrating generative artificial intelligence into educational practice, not instinctively defending against it,” *IEEE Software*, vol. 41, no. 2, 2024. 1
- [10] D. Russo, “Navigating the complexity of generative ai adoption in software engineering,” *ACM Trans. Softw. Eng. Methodol.*, 2024. 1
- [11] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp. 31–53, 2023. 1
- [12] A. Chen, C. Wong, B. Sharif, and A. Peruma, “Exploring code comprehension in scientific programming: Preliminary insights from research scientists,” in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, ICPC ’25, (New York, NY, USA), Association for Computing Machinery, 2025. 1
- [13] J. Wang, L. Li, and A. Zeller, “Better code, better sharing: on the need of analyzing jupyter notebooks,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER ’20, (New York, NY, USA), p. 53–56, Association for Computing Machinery, 2020. 1
- [14] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, “A large-scale comparison of Python code in Jupyter notebooks and scripts,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR ’22, (New York, NY, USA), p. 353–364, Association for Computing Machinery, 2022. 1, 2
- [15] K. Adams, A. Vilkomir, and M. Hills, “A Comparison of Machine Learning Code Quality in Python Scripts and Jupyter Notebooks,” *J. Comput. Sci. Coll.*, vol. 39, p. 96–108, Nov. 2023. 1
- [16] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, “A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 507–517, 2019. 1
- [17] C. Wong, G. Larsen, R. Huang, B. Sharif, and A. Peruma, “Method Names in Jupyter Notebooks: An Exploratory Study,” in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, ICPC ’25, (New York, NY, USA), Association for Computing Machinery, 2025. 1, 2
- [18] E. W. Høst and B. M. Østvold, *Debugging Method Names*, p. 294–317. Springer Berlin Heidelberg, 2009. 1
- [19] E. W. Host and B. M. Ostvold, “The programmer’s lexicon, volume i: The verbs,” in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, p. 193–202, IEEE, Sept. 2007. 1
- [20] R. Alsuhailani, C. Newman, M. Decker, M. Collard, and J. Maletic, “On the naming of methods: A survey of professional developers,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, p. 587–599, IEEE, May 2021. 1
- [21] C. D. Newman, R. S. AlSuhaibani, M. J. Decker, A. Peruma, D. Kaushik, M. W. Mkaouer, and E. Hill, “On the generation, structure, and semantics of grammar patterns in source code identifiers,” *Journal of Systems and Software*, vol. 170, p. 110740, 2020. 2
- [22] SCANL, “Identifier naming structure catalogue,” https://github.com/SCANL/identifier_name_structure_catalogue. 2
- [23] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A systematic survey of prompt engineering in large language models: Techniques and applications,” *arXiv preprint arXiv:2402.07927*, 2024. 2
- [24] “Artifact package.” url = <https://doi.org/10.5281/zenodo.16310985>. 3