Completing Code using Large Language Models

Sarthak Siddhant Bharadwaj
sarthak7@colostate.edu
Computer Science
Colorado State University

Sri Ram Duvvuri
sriram17@colostate.edu
Computer Science
Colorado State University

***Abstract*— The main goal of our project is to improve software development's issue identification and code completion by utilizing Large Language Models (LLMs)[1]. Our goal is to investigate how LLMs can be used to detect software defects[2] and increase the accuracy of code production, which would enhance the caliber and productivity of software projects[3]. Furthermore, we suggest looking into how system 2 thinking might be included into LLMs to provide AI cognizance a more sophisticated level of analytical reasoning[4]. It is anticipated that this investigation will produce novel approaches and perspectives, greatly advancing AI-supported software engineering[5]. Our emphasis on these fields should contribute to the creation of complex, high-caliber software solutions, which will be a significant advancement in the nexus between software development methodologies and artificial intelligence.**

Keywords: LLM, System-2 thinking, AI cognizance, Code completion, Bug fixing

I. INTRODUCTION

The recent rise in the advancement of Large Language Models (LLMs) specialized for generating code has attracted considerable interest. Transformer-based models like Codex and Llama-2, which are trained on extensive collections of open-source code repositories, have shown effectiveness in generating code across multiple programming languages, including Python, Java, and C. These models view code generation as a process of transformation, where they convert natural language descriptions (prompts) into executable programming language statements.

Positioned as potential AI collaborators in software projects, Large Language Model (LLM)-based code generators are anticipated to have a significant impact on the overall quality of software projects. However, akin to human-written code, code generated by LLMs is susceptible to errors. Nevertheless, recent research by Asare et al., which analyzed vulnerabilities present in LLM-generated code (utilizing Copilot) compared to those in human-written code, revealed that LLM models do not produce identical vulnerabilities as humans[2], [6]. Concurrently, various studies have indicated that LLMs and human developers may not concentrate on the same aspects of a prompt to solve a coding task. This raises a fundamental question: Do LLMs generate faults similar

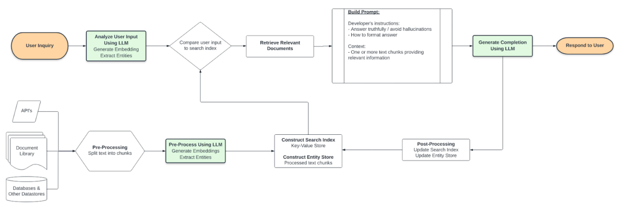


Fig. 1. The above figure is the basic working architecture of a Large Language Model

to those produced by human developers? Addressing this question is critical, as the efficacy of widely used quality assurance techniques such as mutation testing relies on an accurate characterization of faults present in the code being evaluated[4].

II. MOTIVATION

This section provides an overview into the realm of Large Language Models (LLMs) and their utilization within natural language processing (NLP) which undergo initial training on extensive datasets to develop broad proficiency before they are further trained on specialized datasets such as code. This initial pretraining phase[6] enables the model to acquire general representations that can subsequently be refined through additional training specifically focused on programming data.

Notable examples of LLMs include the GPT series developed by OpenAI and Google's BERT, which have gained widespread recognition for their proficiency in producing cohesive and contextually fitting text. Within the domain of code completion, scholars have delved into diverse methodologies, ranging from rule-based frameworks to statistical models and machine learning algorithms. Despite showcasing potential effectiveness in specific contexts, these methodologies frequently encounter challenges in reconciling the intricacies and variations[7] present in both natural language and programming languages simultaneously.

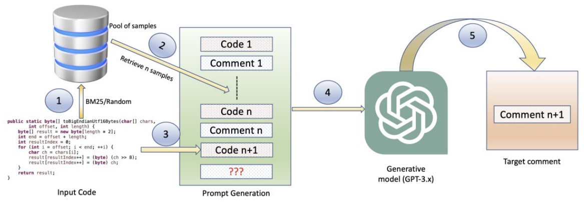
III. Background

A. Large Pre Trained Language Model

Large Pre-Trained Language Models (LLMs) have become prevalent in natural language processing (NLP), demonstrating impressive capabilities across various tasks such as machine translation , text summarization , and classification . These models are built upon the Transformer architecture , comprising an encoder to capture input representation and a decoder to generate output tokens. Initially, LLMs undergo unsupervised pre-training on vast amounts of text data and are subsequently fine-tuned for specific tasks. However, some tasks may lack sufficient fine-tuning data. To address this, researchers have explored the potential of LLMs[17] to perform on downstream tasks without fine-tuning. This is accomplished through prompt engineering , wherein the model is provided with natural language descriptions and task demonstrations before presenting the target input. This approach capitalizes on the broad applicability of LLMs, as their unsupervised pre-training dataset already covers diverse domains and tasks. Leveraging this concept alongside the exponential growth in LLM size, remarkable performance in numerous tasks can be achieved even without fine-tuning.

Various large pre-trained language models (LLMs) have been introduced for automating code generation. One particularly powerful LLM is OpenAI's Codex, extensively utilized across diverse code-related SE tasks. Codex, a closed-source auto-regressive LLM based on GPT, ranges in size from 13 million to 14 billion parameters and is fine-tuned on a dataset comprising 63 million public repositories from GitHub. Notably, [20]Codex powers GitHub Copilot[7], an in-IDE developer coding assistant capable of generating code based on provided context, with a maximum input length of 4,124 tokens. Recently, GitHub Copilot introduced "Copilot Chat," refined with human feedback for dialog applications. Copilot Chat proves adaptable for various code-related tasks, excelling in generating code fragments, describing code snippets in natural language, generating unit tests, and rectifying faulty code, all tailored to the specific context of the task at hand.

LLMs can be categorized into three main types based on their architectures: encoder-only, decoder-only, and encoder-decoder models. Encoder-only models, exemplified by **BERT** , comprise solely the encoder component of a Transformer. They are primarily designed to learn data representations and are trained using the Masked Language Modeling (MLM) objective. In Masked Language Modelling, a small percentage (e.g., 15) of tokens in the training data are replaced by masked tokens, and the models are trained to predict the original values of these masked tokens based on



```
public static byte[] toBigEndianUtf16Bytes(char[] chars,
        int offset, int length) {
    byte[] result = new byte[length * 2];
    int end = offset + length;
    int resultIndex = 0;
    for (int i = offset; i < end; ++i) {
        char ch = chars[i];
        result[resultIndex++] = (byte) (ch >> 8);
        result[resultIndex++] = (byte) ch;
    }
    return result;
}
```

Fig. 2. The above figure is how a Large Language Models work

bidirectional contexts. In contrast, decoder-only models, such as GPT-3 and GPT-Neo, are large generative models that employ the decoder to predict the next token output given all previous tokens (i.e., left context or prefix only). Encoder-decoder models, like **T5** and **BART[10]**, combine the usage of both encoder and decoder components, particularly for sequence-to-sequence tasks. In such tasks, the training objective aims to recover the correct output sequence[16] given the original input (e.g., from corrupted to uncorrupted). One training objective for encoder-decoder models is span prediction tasks, where random spans (consisting of multiple tokens) are replaced with artificial span tokens, and the model is tasked with recovering the original tokens. During inferencing, encoder-decoder models can be utilized to infill text by incorporating artificial span tokens in place. Recently, researchers have also merged MLM with generative models to facilitate both bidirectional and autoregressive text generation or infilling. In the context of our APR scenario, all types of LLMs hold potential for generative or infilling-style APR, and we have selected nine state-of-the-art LLMs for our study.

B. Open Coding

Open coding, also referred to as post-formed code, is a qualitative research technique utilized for analyzing textual data such as survey responses and interview transcripts. In open coding, researchers adopt a relatively unstructured approach, examining the data line by line and creating codes or labels for themes and concepts identified within the data. These codes are subject to dynamic adjustments, allowing for additions, removals, or mergers to refine the final code book. This flexibility permits a more adaptive and nuanced analysis compared to the rigid structure of preformed codes.

In the open coding process, two human reviewers independently commence coding on the same set of samples. Throughout this phase, reviewers explore patterns, similarities, and categories within the samples that are pertinent to the quantitative variables, such as characteristics or root causes of bugs in our study. They assign discrete categories to each sample. Subsequently, the two reviewers engage in a discussion session to exchange and deliberate on their identified categories, thereby establishing the initial code book.

IV. OBJECTIVE

We began by reviewing the studies conducted by Al-rimy et al. — and Eze et al. — to gain insights into the historical landscape of the developments of large language models code generation. Subsequently, we examined recent research surveys —- and to comprehend the current problems posed by large language models in the modern era. Our research has identified various shortcomings in existing large language models code generation.

A. *By Developing and Evaluating LLM based approach for bug fixing*

This objective centers on developing and assessing methods that utilize large pre-trained language models (LLMs) to automatically detect and rectify bugs in software code.

B. *To Search about the Effectiveness of LLM's in Understanding Code Bugs*

This objective seeks to evaluate the proficiency of LLMs in precisely recognizing and understanding a wide array of code bugs, encompassing syntax errors, logical inconsistencies, and performance bottlenecks.

C. *Integration of LLM's into Software Development Projects for Bug Identification*

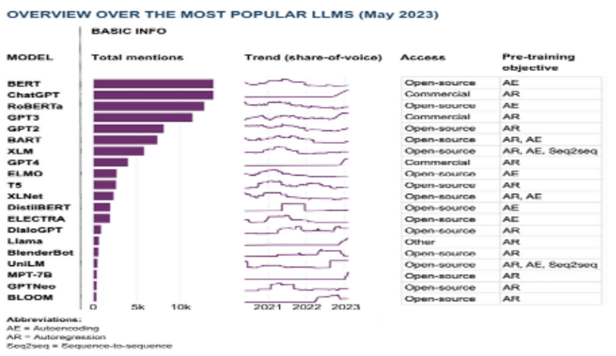
This objective entails exploring the integration of LLMs into established software development workflows to aid developers in effectively identifying and addressing code bugs.

D. *Assess the Validation of LLM based Bug Approaches*

This objective aims to assess how well bug-fixing techniques based on LLMs perform across a range of programming languages, project scales, and software development contexts in terms of their robustness and applicability.

V. RECENT DEVELOPMENTS

In recent research, there has been a growing exploration into employing LLMs for code completion endeavors, capitalizing on their adeptness in comprehending and producing text that mirrors human language. For instance, Codex, an LLM crafted by OpenAI, has showcased remarkable proficiency in generating code snippets prompted by natural language cues[8]. Addition- ally, various studies have concentrated on refining preexisting LLMs through training on datasets pertinent to coding, aiming to enhance their efficacy in code completion tasks. Nevertheless, despite these strides, challenges persist in tailoring LLMs to suit the intricate nuances



MODEL	Access	Pre-training objective
BERT	Open-source	AE
ChatGPT	Commercial	AR
RoBERTa	Open-source	AE
GPT3	Commercial	AR
GPT2	Open-source	AR
BART	Open-source	AR, AE
XLM	Open-source	AR, AE, Seq2seq
GPT4	Commercial	AR
ELMO	Open-source	AE
T5	Open-source	AR
XLNet	Open-source	AR, AE
DistilBERT	Open-source	AE
ELECTRA	Open-source	AE
DialoGPT	Open-source	AR
Llama	Other	AR
BlenderBot	Open-source	AR
UniLM	Open-source	AR, AE, Seq2seq
MPT-7B	Open-source	AR
GPTNeo	Open-source	AR
BLOOM	Open-source	AR

Fig. 3. Recent usage of Large Language Models by number of users

inherent in programming languages and in optimizing their utility for practical implementation within software development ecosystems[9]. Some recent advancements which have been the main focus have centered on enhancing the efficiency of Large Language Models, aiming to minimize both their computational requirements and environmental footprint. Strategies like model distillation, sparse attention mechanisms, and parameter pruning have been investigated to decrease the computational burden of training and inference while maintaining performance levels[5].

VI. RESEARCH QUESTION

In this vast field we are curious about:

RQ1. What traits do defects have that appear in code that LLMs create for tasks that are part of actual projects?

RQ2. How important are the found bug patterns in LLM-generated code for researchers and software developers that work with LLMs?

RQ3. What can be plausible effects of System 2 Thinking on LLMs , How can we merge System 2 thinkingto empower the LLMs?

VII. APPROACH

In this study, motivated by previous research findings, we thoroughly investigated different approaches to enhance the precision and accuracy of Large Language Models (LLMs) in tasks including bug identification and code completion. We evaluated techniques such domain-specific knowledge base integration, model modification, and data augmentation to find the best ways to improve LLM performance in coding contexts. Finding these strategies was crucial because it helped us better customize LLMs to the particular needs of software development, producing a more potent and advanced model. This research was critical to laying the groundwork for future innovations by identifying strategies that greatly improve LLMs' comprehension and production of code[1].

Dataset	#Bugs	#SF	#SH	#SL	Source	Language
Defects4J 1.2	391	255	154	80	real-world	Java
Defects4J 2.0	438	228	159	78	real-world	Java
QuixBugs-Java	40	40	37	36	coding problems	Java
QuixBugs-Python	40	40	40	40	coding problems	Python
ManyBugs	185	39	23	12	real-world	C
Total	1094	572	413	246		

Fig. 4. Evaluation of Dataset Statistics

A theoretical yet possible approach to enhance Large Language Models' (LLMs') performance in software engineering tasks like code completion and debugging was to include System 2 thinking into LLMs. Theoretically, LLMs had a better understanding of the logic and semantics of code thanks to System 2 thinking, which is characterized by analytical and purposeful processing. This enabled them to identify errors in the code as well as more accurately comprehend the aim of the programmer[3][7]. We looked into how this combination might lead to LLMs that, in theory, understand complex problem-solving circumstances better and provide more precise solutions. We looked into ways to significantly improve LLM performance while taking on challenging datasets from different programming languages and projects related to the programming languages.

VIII. Initial Findings

A. Dataset Model

As a thorough assessment tool, we rely on Defects4J version 2.0, a carefully curated collection of real-world bugs from 17 Java projects. Each bug in Defects4J is connected to a corresponding bug report, making it ideal for evaluating performance. Despite encountering 58 bugs with incorrect pairings and six bugs with differing directory structures, we were able to evaluate 750 bugs out of the total 814 bugs with paired bug reports.

Among the bugs in the Defects4J benchmark, 60 bugs are also found in commonly used research datasets. We use this subset for comparing reproduction techniques[13].

Given that Codex, the LLM utilized in our research, was trained with data obtained up until July 2022, worries about data leakage are still important for the Defects4J dataset. To tackle this issue, we collected 581 Pull Requests (PRs) from 17 GitHub repositories, making sure that this specific dataset was not used in the training of Codex. We then refined these PRs to only keep the ones that added tests to the project, resulting in 435 PRs. By further eliminating PRs that were not merged into the main branch or associated with multiple issues, we were left with 84 PRs[13]..

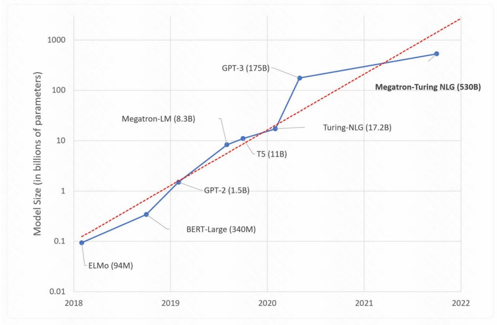


Fig. 5. Model Size comparison of various Large Language Models

In order to confirm the existence of bugs in 84 pull requests, we made sure that the tests added by developers in the PRs failed before merging without any errors and passed after merging. We only considered bug pairs for our final list if all tests were successfully replicated. After this thorough verification process, we identified 31 bugs that could be reproduced along with their bug reports, creating the GitHub Recent Bugs dataset. This dataset helps confirm that the patterns seen in Defects4J are not affected by any leakage of data[11].

B. Metrics and Evaluation

We decided to use a more cautious approach with an agreement threshold of 1 in order to retain as many reproduced bugs as possible. Out of the 640 bugs identified with a Fully Implemented Bug (FIB), we selected 450 bugs to further analyze. Among these chosen bugs, 339 were successfully reproduced, resulting in a precision rate of 0.75 (calculated by dividing 339 by 640), while the recall rate (the proportion of selected reproduced bugs among all reproduced bugs) was 0.93 (calculated by dividing 339 by 361). On the other hand, 197 bugs that were not reproduced were filtered out, while only a few successfully reproduced bugs were eliminated. It is important to note that by using a more aggressive threshold of 10, we could achieve a higher precision rate of 0.89, although this would come at the cost of a lower recall rate of 0.45[8].

The main challenge in enhancing Large Language Models (LLMs) for software development is the limitation of time and computing resources. The process of training and refining these models demands a considerable amount of processing power and time, especially as they become more complex and advanced. This task often exceeds the capabilities of individual researchers or smaller organizations, hindering their progress in this field. Moreover, there is a growing demand for more efficient models that can deliver high performance without the environmental impact caused by extensive computational resources[10]. This obstacle significantly

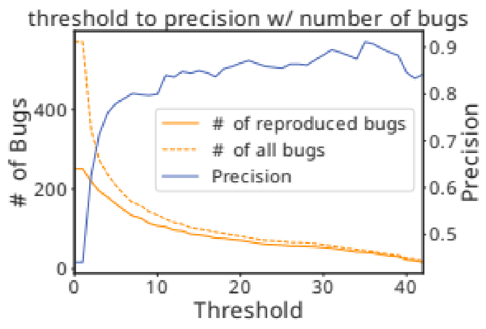


Fig. 6. The above figure is representation of bugs selected and it's precision

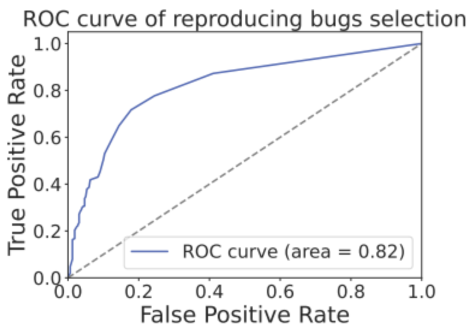


Fig. 7. The ROC Curve for Reproduction of Bugs

hampers the rapid advancement and accessibility of state-of-the-art AI tools for software development.

AI and machine learning face a challenge in integrating advanced reasoning similar to human-like thinking into Large Language Models (LLMs). This limitation hinders LLMs' effectiveness in complex software engineering tasks like debugging and code interpretation. The existing technological gap underscores the urgent necessity for innovative advancements in computational optimization, training methods, and model structure. Overcoming these barriers is crucial to fully harnessing LLMs in software development, empowering developers globally with more precise, reliable, and advanced assistance. Closing this gap not only fulfills technical needs but also transforms the coding landscape., Closing this gap not only fulfills technical needs but also transforms the coding landscape issues are tackled and resolved in the digital age.

IX. Results and Findings

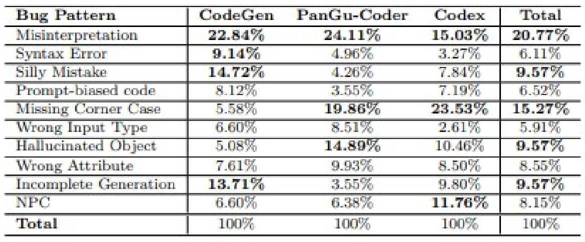
In this section, we explore the bug patterns derived from our analysis, We investigate the characteristics, occurrence rates, and distribution of these bugs across the three distinct LLMs.

Across the three Large Language Models analyzed, various characteristics were observed. Among these, misinterpretation stands out, occurring when an LLM struggles to interpret a given prompt accurately. Additionally, syntax errors are prevalent, often arising from issues such as incorrect indentation or missing semicolons. Similarly, generating incorrect inputs is notable, as instances were found where the LLM produced erroneous function names despite the provided data. Here are some of the results we obtained during our analysis. Here is the taxonomy for the bug patterns which are reproduced.

A. Findings

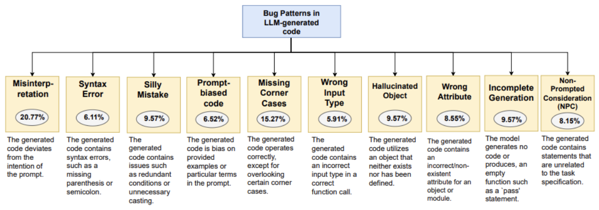
As we can see in the reults that the frequency of occurring more bugs is **Misinterpretation** and followed by **Missing Cases** which generates incomplete code and another point to note here is that these bugs are frequently occuring in the code generated by the LLM's and not by humans.

Another important finding was about the cost implications of integrating Large Language Models (LLMs) into organizational workflows are multifaceted, encompassing various elements. Initially, substantial upfront investments are necessary, as LLMs often demand significant computational resources and specialized hardware for both training and inference tasks. Additionally, licensing fees and subscription costs from LLM vendors can constitute a considerable portion of the overall expenditure. These initial costs are compounded by ongoing expenses such as maintenance, updates, and support services. Moreover, scalability introduces another layer of cost complexity, with larger models or



Bug Pattern	CodeGen	PanGu-Coder	Codex	Total
Misinterpretation	**22.84%**	**24.11%**	**15.03%**	**20.77%**
Syntax Error	**9.14%**	4.96%	3.27%	6.11%
Silly Mistake	**14.72%**	4.26%	7.84%	**9.57%**
Prompt-biased code	8.12%	3.55%	7.19%	6.52%
Missing Corner Case	5.58%	**19.86%**	**23.53%**	**15.27%**
Wrong Input Type	6.60%	8.51%	2.61%	5.91%
Hallucinated Object	5.08%	**14.89%**	10.46%	**9.57%**
Wrong Attribute	7.61%	9.93%	8.50%	8.55%
Incomplete Generation	**13.71%**	3.55%	9.80%	**9.57%**
NPC	6.60%	6.38%	**11.76%**	8.15%
Total	100%	100%	100%	100%

Fig. 8. Results of our analysis



The generated code deviates from the intention of the prompt.

The generated code contains syntax errors, such as a missing parenthesis or semicolon.

The generated code contains issues such as redundant conditions or unnecessary casting.

The generated code is bias on provided examples or particular terms in the prompt.

The generated code operates correctly, except for overlooking certain corner cases.

The generated code contains an incorrect input type in a correct function call.

The generated code utilizes an object that neither exists nor has been defined.

The generated code contains an incorrect/non-existent attribute for an object or module.

The model generates no code or produces, an empty function such as a `pass' statement.

The generated code contains statements that are unrelated to the task specification.

Fig. 9. The taxonomy of bug patterns

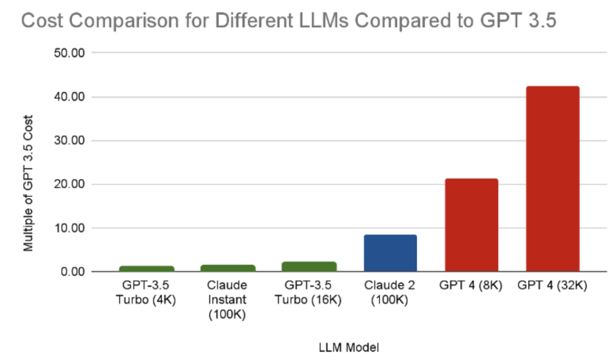


Fig. 10. Cost Comparison for Different LLM's

increased computational resources potentially required to accommodate growing datasets and complex tasks. Furthermore, indirect expenses associated with integrating LLMs into existing infrastructure, training personnel, and adapting workflows to leverage these models must also be considered. While the capabilities of LLMs offer considerable potential across various applications, organizations must conduct thorough assessments of the total cost implications and carefully balance them against anticipated benefits prior to implementation.

B. Suggestions for Improvements

To address the identified areas for improvement, several potential avenues can be explored. Firstly, fine-tuning strategies should be refined to enhance LLMs' capability in detecting subtle bugs. This could involve optimizing domain-specific fine-tuning methods or incorporating transfer learning from related tasks. Secondly, efforts should be directed towards optimizing LLM architectures and parameters to minimize computational overhead while ensuring accuracy is maintained. Additionally, the integration of LLM-based bug detection techniques into developer tools and Integrated Development Environments (IDEs) could streamline the bug resolution process, offering developers real-time feedback. Lastly, collaborative learning approaches, wherein LLMs assimilate knowledge from multiple developers, hold promise in enhancing detection accuracy and efficiency by tapping into a wider context of code understanding.

X. THREATS TO VALIDITY

A. Internal Validity

Internal Validity pertains to whether our experiments establish causality. In our study, two issues pose a threat to internal validity: the unpredictability of tests and the variability in LLM query. Although we do encounter a small proportion of generated tests that exhibit the unpredictability, their occurrence is notably minor (less than '6' percent) compared to the total number of tests generated by the Large Language Models. Therefore, we contend that their presence does not substantially impact our findings.[16]

B. External Validity

The issue of **External validity** pertains to the extent to which the findings presented can be applied to other projects in java or projects in different programming languages. It is challenging to determine whether the results demonstrated here would apply universally across various projects or extend to projects developed in other programming languages.[15]

XI. CONCLUSION

How does code generation by LLM[14] affect human lives. In this paper, we initially highlight the significance of the report-to-test issue through an examination of pertinent literature and an analysis of 300 open-source repositories, even though they are superior at code debugging? What potential advantages or consequences may there be on a broad socioeconomic level if we combined human cognigance with LLM? The choice of what data to feed the model is far more critical than how much data we need to feed it. Our findings and that of other professors working on the same topic suggest that further time and developments in the field will be necessary before we can draw any firm conclusions. It is encouraging to see the noteworthy effort in this area by P. Sadayappan.

XII. CONTRIBUTION

Sarthak Bharadwaj : I completed a thorough literature review with great engagement as one of my contributions to the research paper, and I also attended several conferences that deepened our conversations on ”Completing Code using Large Language Models.”

During the literature review, I focused on gathering and evaluating significant scholarly articles, journals, and prior research findings that investigate the use of large language models in code completion tools. My objective was to understand the current state, limitations, and prospective applications of these technologies.

Sriram Duvvuri : The model and methodology were my proposals. I mainly focused on the evaluation techniques to evaluate the performance of the Large Language Models. But, due to tight deadlines, scarce resources, and the early stages of development in this particular field, we were unable to satisfactorily answer the System 2 thinking question.”

XIII. FUTURE WORKS

Improving the way we gather data by collecting a wider variety of datasets from different sources could make our evaluation process more reliable and comprehensive. Using advanced methods like automated testing or machine learning to confirm bug issues and their reproducibility could simplify the verification process and lessen the need for manual work[7]. Creating and testing new assessment criteria that go beyond just precision and recall, like F1 score, information retrieval metrics, or performance measures specific to a certain field, can give us a better understanding of how well bug reproduction techniques work and where their constraints lie. This could lead to more in-depth comparisons and analyses.

Large Language Models (LLMs) provide intriguing paths for future study to improve software development, especially in the areas of debugging and code completion. Further research should focus on improving the way that System 2 thinking is included into LLMs in order to increase the models' comprehension of intricate logic and programming tasks. This research may result in the creation of increasingly complex AI tools that are better able to understand the subtleties of various coding languages and developer intentions. Not only would these tools increase the accuracy of code creation, but they would also provide customized recommendations and debugging support, which might completely change the way developers interact with coding environments and greatly increase productivity and code quality[14].

Furthermore, there is a serious concern about the environmental impact of large-scale LLM deployments, which calls for more research. The goal of future research should be to maximize the energy efficiency of these models without compromising their functionality. Through the utilization of sophisticated methods such as model pruning and investigation of new, lightweight architectures, scientists can create development tools powered by AI that are long-lasting. By lowering the necessary computational resources, these efforts will not only increase the accessibility of LLMs but also contribute to the larger objective of developing ecologically friendly technologies[15]. Such advancement is essential to guaranteeing that software engineering can broadly benefit from artificial intelligence (AI) while leaving the least possible environmental impact.

REFERENCES

[1] K. Jesse, T. Ahmed, P. T. Devanbu and E. Morgan, ”Large Language Models and Simple, Stupid Bugs,” 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), Melbourne, Australia, 2023, pp. 563-575, doi: 10.1109/MSR59073.2023.00082.

[2] ”Decoding Logic Errors: A Comparative Study on Bug Detection by Students and Large Language Models”ACE '24: Proceedings of the 26th Australasian Computing Education ConferenceJanuary 2024Pages 11–18

[3] H. Pearce, B. Tan, B. Ahmad, R. Karri and B. Dolan-Gavitt, ”Examining Zero-Shot Vulnerability Repair with Large Language Models,” 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2023, pp. 2339-2356, doi: 10.1109/SP46215.2023.10179324.

[4] ”Using Large Language Models to Enhance Programming Error Messages” SIGCSE 2023: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1March 2023Pages 563–569https://doi.org/10.1145/3545945.3569770

[5] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu and B. Myers, ”Using an LLM to Help With Code Understanding,” in 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 2024 pp. 881-881.

[6] Marvin, G., Hellen, N., Jjingo, D., Nakatumba-Nabende, J. (2024). Prompt Engineering in Large Language Models. In: Jacob, I.J., Piramuthu, S., Falkowski-Gilski, P. (eds) Data Intelligence and Cognitive Informatics. ICDICI 2023. Algorithms for Intelligent Systems. Springer, Singapore.

[7] S. Kang, J. Yoon and S. Yoo, ”Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction,” 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 2312-2323, doi: 10.1109/ICSE48619.2023.00194.

[8] ”LLM-Based Code Generation Method for Golang Compiler Testing”ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software EngineeringNovember 2023Pages 2201–2203https://doi.org/10.1145/3611643.3617850

[9] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, “How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies,” in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1281–1291. [Online]. Available: https://doi.org/10.1145/3468264.3473922

[10] R. Premraj, T. Zimmermann, S. Kim, and N. Bettenburg, “Extracting structural information from bug reports,” in Proceedings of the 2008 international workshop on Mining software repositories - MSR '08, ACM Press. New York, New York, USA: ACM Press, 05/2008 2008, pp. 27–30.

[11] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” in Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems, 2021, pp. 1–7.

[12] M. Soltani, P. Derakhshanfar, X. Devroey, and A. Van Deursen, “A benchmark-based evaluation of search-based crash reproduction,” Empirical Software Engineering, vol. 25, no. 1, pp. 96–138, 2020.

[13] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Martin, “Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset,” Empirical Software Engineering, vol. 22, pp. 1936– 1964, 2016

[14] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862.

[15] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 79, 1–13. https://doi.org/10.1145/3597503.3639138

[16] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair Is Nearly Generation: Multilingual Program Repair with LLMs”, AAAI, vol. 37, no. 4, pp. 5131-5140, Jun. 2023.

[17] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).

Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[18] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs.nbsp;Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665

[19] Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '23). Association for Computing Machinery, New York, NY, USA, 1146–1155. https://doi.org/10.1145/3583131.3590481

[20] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 126, 1–13. https://doi.org/10.1145/3597503.3639121

[21] XIV. Sources of Information

A. Journals

[22] Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. ACM Trans. Knowl. Discov. Data Just Accepted (February 2024). https://doi.org/10.1145/3649506

[23] LLM-GEm: Large Language Model-Guided Prediction of People's Empathy Levels towards Newspaper Article" Md Rakibul Hasan, Md Zakir Hossain, Tom Gedeon, and Shafin Rahman. 2024. LLM-GEm: Large Language Model-Guided Prediction of People's Empathy Levels towards Newspaper Article. In Findings of the Association for Computational Linguistics: EACL 2024, pages 2215–2231, St. Julian's, Malta. Association for Computational Linguistics.

[24] Hardware Security Bug Code Fixes By Prompting Large Language Models B. Ahmad, S. Thakur, B. Tan, R. Karri and H. Pearce, "On Hardware Security Bug Code Fixes By Prompting Large Language Models," in IEEE Transactions on Information Forensics and Security, doi: 10.1109/TIFS.2024.3374558. keywords: Maintenance engineering;Computer bugs;Codes;Hardware;Security;Software;Registers;Hardware Security;Large Language Models;Bug Repair

B. Conferences

[25] https://2023.emnlp.org/program/ EMNLP stands out as a prominent conference in the field of natural language processing, addressing a diverse array of subjects such as Large Language Model's and beyond.

[26] https://2023.aclweb.org/ The 61st Annual meeting of the Association form Computa- tional Linguistic has been One of the top-tier conferences in computational linguistics, it showcases research on a multitude of language processing aspects, encompassing Large Language Models, language generation, and comprehension among others.

[27] https://devday.openai.com/ OpenAI's first developer conference was mainly dedicated on Large Language Model's.

C. Research groups

[28] OpenAI - OpenAI has gained recognition for its innovative efforts in creating expansive large language models like GPT.

[29] Google Research - The AI Language team within Google's research division is actively engaged in substantial research on large language model comprehension and generation.

[30] Microsoft Research - Microsoft's research division investigates various AI-related subjects, encompassing Natural Language Processing and Large Language Models among them. Their research frequently centers on creating models and algorithms aimed at enhancing language comprehension and generation capabilities.

[31] Stanford University - Conducting research in AI and machine learning has been a top priority and their primary emphasis lies in reinforcement learning, yet they also contribute to the exploration of Large Language Model's and their practical implementations.

D. Industrial publications

[32] https://lingming.cs.illinois.edu/publications/icse2023a.pdf

[33] https://www.microsoft.com/en-us/research/blog/partnering-people-with-large-language-models-to-find-and-fix-bugs-in-nlp-systems/

[34] https://www.ibm.com/blog/open-source-large-language-models-benefits-risks-and-types/

E. News

[35] https://about.fb.com/news/2023/08/code-llama-ai-for-coding/

[36] https://stackoverflow.blog/2023/12/28/self-healing-code-is-the-future-of-software-development/