How Developers Evaluated LLM-Generated Code and How They Debug the LLM-Generated Code

MD ROBIUL ISLAM, Department of Computer Science, William & Mary, USA

Large Language Models (LLMs) are increasingly used in software development tasks such as code generation and debugging. This study investigates how developers utilize LLM-generated code and the methods they adopt to address errors within it. Based on a survey of professional developers, we find that while LLMs are commonly used for generating small code snippets with explanations, they often produce incorrect logic, syntax errors, inefficient code, and security vulnerabilities. Developers rely on a mix of traditional debugging techniques, such as print statements and interactive debuggers, and LLMs themselves to correct these issues. Our findings reveal both the potential and the limitations of current LLM tools, highlighting the need for more reliable outputs and better debugging support. This study contributes to the broader understanding of how LLMs are integrated into development workflows and provides direction for future improvements in tool design and developer support systems.

Additional Key Words and Phrases: LLMs, Code Generation, Developers

ACM Reference Format:
Md Robiul Islam. 2025. How Developers Evaluated LLM-Generated Code and How They Debug the LLM-Generated Code. 1, 1 (May 2025), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The programming process has been greatly impacted by recent developments in LLMs, which are being incorporated into the workflow more and more. One such example is GitHub Copilot [10], a code-generation tool that uses OpenAI's GPT [22] model to produce lines or subroutines of code from natural language comments or existing code. The main purpose of these code generation tools is to increase productivity by providing features like autocomplete, speedier task completion, and simpler syntax recall. In a controlled study, the Copilot-using group finished programming tasks 55.8% quicker than the non-Copilot-using group [25]. Although code generation is made easier by LLM-powered tools, the resulting code's quality is not assured. According to a recent study [30], LLM-generated code frequently has problems with code quality, including runtime and compilation errors, inaccurate outputs, and concerns with maintainability and style. Developers must, therefore, invest time in assessing the code's accuracy, resolving any issues, and incorporating the code into the current codebase after utilizing these tools to generate a portion of code. According to Hussen et al., while using Copilot for programming, developers spend a considerable amount of time (37.99% overall) considering and validating ideas, debugging and testing, and changing written code [19]. As demonstrated in various contexts such as conversational agents, qualitative coding, and natural language data queries, error discovery and repair procedures are essential for effective human-AI

Author's Contact Information: Md Robiul Islam, miislam@wm.edu, Department of Computer Science, William & Mary, Williamsburg, Virginia, USA.

ACM XXXX-XXXX/2025/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

interaction, directing the creation of underlying AI models and user interfaces. The study found that developers using Copilot completed the task 55.8% faster than those in the control group. The results highlight the potential of AI pair programmers to enhance coding efficiency and suggest that these tools could play a significant role in supporting individuals transitioning into software development careers[25]. Even though LLMs have a lot of potential for code production, it is becoming more and more important to formally and thoroughly investigate the dependability and caliber of code produced by LLMs. This is because new programmers and others without any prior coding knowledge are now using LLMs in addition to experienced developers. If LLM-generated code quality problems are not appropriately found and fixed, they could negatively impact code comprehension, introduce errors, or expose users' projects to security risks. Thus, a reduction in the general caliber of software systems may result from the extensive use of LLMs for code creation. Thus, it is essential to look into and fix frequent problems with code quality that can occur while using code created by LLMs.

In this work, motivated by previous problems of using LLMs such as ChatGPT, we surveyed 30 different software developers with different experience (0 to 6+ years) of using LLMs for code-related tasks such as code generation[17] , code summarization [33], etc,. Our findings reveal the methods developers use to assess functional correctness, performance, and edge case handling in AI-generated code. Additionally, we examine the debugging techniques developers rely on—ranging from traditional methods like print statements and IDE debuggers to prompting the LLM itself for self-repair. This study contributes to the broader understanding of human-AI collaboration in software engineering and highlights areas where current AI coding assistants fall short, pointing to opportunities for tool improvement. This leads us to the following research questions:

- **RQ1:** How do developers approach the validation of LLM-generated code?
- **RQ2:** What common patterns appear in errors made by LLM-generated code?

To answer these questions, we surveyed 30 developers from Bangladesh. We didn't consider any students as developer-like, or in our study, we only considered developers as developers. Ningzhi Tang et al. conducted a lab study with 28 participants to know how developers validate and repair LLM-generated code using eye tracking and IDE Actions [27]. They conducted the survey in a closed lab with the door and window closed, which is not similar to a real developer environment. Also, they surveyed students with three projects that were not similar to the real-time projects of clients.

2 Background & Related Work

Recent advances in LLMs such as Codex [23], ChatGPT [2], and CodeLlama [18] have significantly impacted software engineering by automating code generation, documentation, and even basic testing tasks [6]. These models are trained on vast corpora of code and natural language [16], enabling them to generate plausible and often functional code snippets in response to natural language prompts. As a result, developers are increasingly incorporating LLMs into their workflows to improve productivity and explore alternative implementations [32].

Despite their utility, concerns remain about the correctness, security, and maintainability of LLM-generated code. Studies have shown that these models often produce code with logical errors, insufficient error handling, or inefficient algorithms [26]. Pearce et al. [24] found that GitHub Copilot could generate insecure code, particularly in cases where prompt clarity was low. Additionally, LLMs are generally unaware of project-specific contexts, such as business rules or dependencies, which can further reduce the reliability of their outputs [7].

In terms of code validation and debugging, prior work has mostly focused on automatic evaluation using test cases or static analysis tools [13]. However, less is known about how developers themselves assess and repair LLM-generated code during real-world tasks. Niu et al. [21] conducted an empirical study of developers' prompt engineering practices and noted that many users manually validate outputs using domain knowledge and debugging tools. Yet, there is a research gap in understanding the cognitive strategies and practical techniques developers use to validate and correct AI-generated code.

Our work builds on this foundation by examining how developers evaluate the quality and correctness of LLM-generated code and how they approach debugging when errors are encountered. By analyzing survey data from software professionals, we aim to uncover patterns in human judgment, validation strategies, and tool usage that have not been extensively studied in prior work.

2.1 A Study of Code Generation Tools Using Large Language Models

In this era of transformer-based models [29], code generation is one of the best options for developers. Several AI assistants such as GitHub Copilot [9], ChatGPT [2], Gemini [28] or newly introduced DeepSeek [11] is uses by the developers of code generation [5], code summarization [1], bug fixing [3], assert statement generation [31] and unit test case generation [8]. Recent studies explored how developers interact with LLM code generation tools. A study conducted by Barke et al. [4] found two forms of developer interactions: acceleration, in which developers use Copilot to code faster, and exploration, in which they utilize it to investigate future directions. Another research by Mzannar et al. [19] discovered a taxonomy of typical developer tasks and used Copilot to classify 21 developers' coding sessions retroactively. They discovered that developers devote over half of their task time to Copilot-related tasks and a significant portion to verifying and revising Copilot's recommendations. Our study adds to the existing literature by examining how developers validate and repair code generated by LLMs.

2.2 Techniques Developers Use for Debugging

Debugging is a multifaceted activity that includes understanding the code, identifying the source of errors, and implementing fixes. It has been a persistent area of focus in software engineering research for many years [27]. Researchers studying professional developers working on large-scale systems have noted that typical debugging practices often follow two main phases: locating the issue and resolving it. Additionally, several studies have explored how usable and effective current debugging tools are in supporting these activities [15]. In contrast to code authored by humans, code produced by large language models (LLMs) exhibits distinct characteristics. Prior research indicates that such code tends to be more verbose [20], often missing necessary contextual information [14], and generally does not match the quality standards of developer-written code [12]. When developers debug code they wrote themselves, they usually have a clear understanding of its structure and purpose. This makes the process different from reviewing and fixing code generated by LLMs, which they did not create. Our study focuses specifically on how developers handle issues in LLM-generated code, rather than traditional, human-written code. By doing this, we aim to better understand how LLMs are changing software development and to identify ways to build tools that can support developers in dealing with the unique challenges these models introduce.

3 Method

Our original plan included both a survey and interviews, but due to time constraints, we concentrated solely on the survey. Below, we describe our participants demographic, survey design and our analysis.

3.1 How participants were recruited and who they were

We designed a survey that included several questions and shared it with developers through a LinkedIn post. We also sent the Google Form link directly to developers and asked them to share it with their colleagues. In addition, we distributed the survey link via email and social media apps like Messenger. Table 1 provides an overview of our participants.

Table 1. Perticipants information

Demographic	Response option	Number of participants (N=29)	Percentage
Gender	Man	23	79.31%
	Woman	6	20.69%
Programming language	Python	6	20.69%
	Java	1	3.45%
	C++	1	3.45%
	PHP	10	34.48%
	JavaScript	8	27.59%
	Dart	1	3.45%
	ABAP	2	6.90%
Role	Junior Software Engineer	8	27.59%
	Associate Software Engineer	5	17.24%
	Software Engineer	10	34.48%
	Application Developer	1	3.45%
	Senior Software Engineer	4	13.79%
	SQA Engineer	1	3.45%
Experience	Beginner (0-2)	12	41.38%
	Intermediate (3-5)	15	51.72%
	Advance (6+)	2	6.90%

3.2 Survey design

To collect data for our study, we designed a survey using Google Forms and distributed it through various channels, including LinkedIn, Messenger, WhatsApp, and other social media platforms. In the first two weeks, we received only five responses. To increase participation, we directly shared the form with developers within our professional networks, who further disseminated it among their contacts. In total, we collected 29 responses from software developers across various companies and backgrounds. Among the respondents, five were female developers.

The survey consisted of 32 questions, combining multiple-choice and open-text formats. Multiple-choice questions (MCQs) included both single-select and multi-select options. We also incorporated optional text fields alongside some MCQs, allowing participants to elaborate on their selections. The initial questions captured demographic and professional background information, such as the respondent's name, company affiliation, and years of experience. Most questions were multiple-choice, while five were strictly open-ended text responses. Among all participants, 33.3% reported PHP as their primary programming language, and 20% identified their role as Junior Software Engineer

Biography of a participant

Name: Saiful Islam
Company: Brain Station Limited
Role: Senior Software Engineer
Experience: 3-5 years

3.3 Prior experience with LLMs

To understand how familiar participants were with large language models (LLMs), we included questions about their prior usage and the tools they commonly engage with. Most respondents reported using LLM-based tools such as ChatGPT, GitHub Copilot, and Gemini for various development tasks, including code generation, debugging, and documentation. Their experience levels varied, with some using these tools regularly in daily workflows, while others had only experimented with them occasionally. This background information helped contextualize their responses and provided insights into how experience with LLMs may influence their evaluation and debugging practices.

3.4 Demographic and background information

We collected demographic information from participants, including their company name, job role, years of experience, and preferred programming languages. To better understand their development background, we also asked about the nature and level of the projects they typically work on. Company affiliation was inferred from the account used to submit the survey response.

3.5 Open-ended question

The survey included a number of open-ended questions designed to collect detailed and personalized responses from participants. Unlike multiple-choice items, these questions allowed respondents to express their thoughts freely, provide justifications, and elaborate on their experiences with LLM-generated code. This approach helped us capture nuanced perspectives that would not be possible through predefined answer choices, offering valuable qualitative insights into how developers evaluate and debug AI-generated outputs.

3.6 Improving developer trust in AI-generated code

To understand how trust in AI-generated code could be improved, we asked participants what features or changes would increase their confidence in using such code. Many respondents emphasized the need for greater transparency in how the code is generated, including explanations of the underlying logic and decision-making process. Others suggested that trust would improve if the code passed standard validation checks, included clear documentation, and adhered to industry best practices. Several participants noted that consistency, contextual awareness, and alignment with project-specific constraints would also contribute to greater trust. These insights point to a demand for more explainable, verifiable, and context-aware AI coding tools.

3.7 Instances where AI-generated code caused issues

Several participants reported specific situations where AI-generated code led to problems during development. In many cases, the code appeared syntactically correct but contained logical flaws that went unnoticed until runtime, such as incorrect condition handling or improper loop structures. Some developers shared that the generated code lacked proper input validation, which introduced vulnerabilities or unexpected behavior when deployed. Others noted that the AI often failed to account for project-specific dependencies or architectural constraints, resulting in integration failures or inefficient implementations. These examples highlight the risks of relying solely on AI-generated code without human oversight and underscore the importance of thorough validation, especially when using LLMs in production environments.

4 Survey Result

In this section, we report on our two RQs.

4.1 RQ1. How do developers approach the validation of LLM-generated code?

In our survey research, we asked 32 questions of the responders. All of the responders are from Bangladesh, and all of them are working in a software company there. In our question, we asked how frequently they use LLM-generated code in the production code or their project. Almost 54.2% of respondents responded that they use LLM-generated code frequently. Surprisingly, 8.3% of respondents responded that they have never used LLM-generated to in the projects. In the Table **??**, information about the developers is presented, where most of the developers' company size is below 100 employees. Among all the 24 responses, 21 responders validated LLM-generated code by manual process, and the percentage is 87.5%. Some of them also run the test cases, and a few responders compare the generated code with the existing implementation from other projects. Regarding debugging methods, 17 developers use print statements or logging systems to fix the generated code. Some of them also use interactive debugging tools such as pdb or VS Code debugger. Some people also give the LLM-generated code to the LLMs to fix the bugs they produce inside the code. LLMs made the most faults in functional correctness, according to 19 engineers.

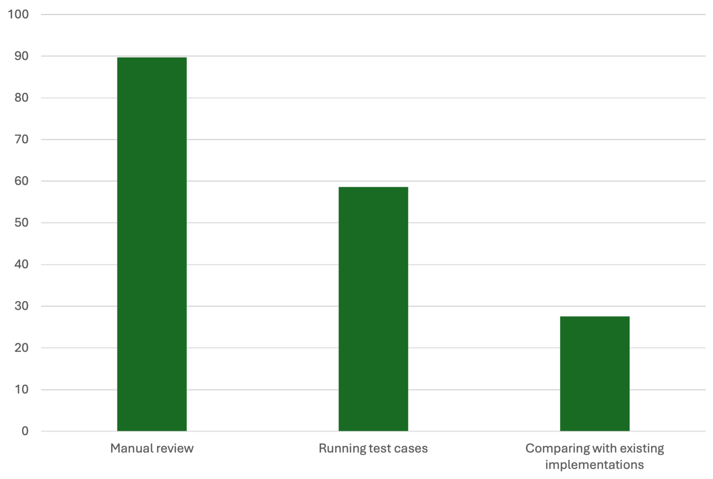


Fig. 1. Methods developers use to evaluate AI-generated code

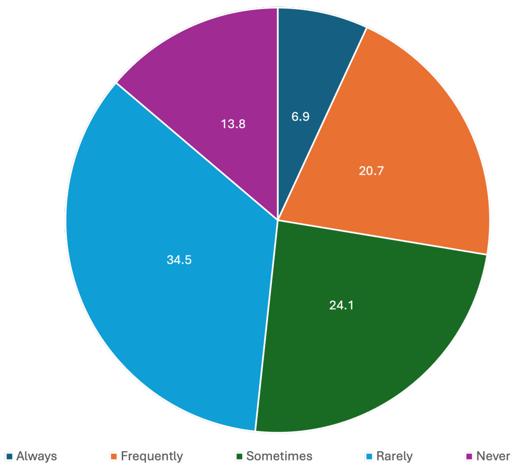


Fig. 2. Frequency of developers accepting AI-generated code without modifications

Figure 1 presents the methods developers use to evaluate AI-generated code. The most common approach is manual review, selected by approximately 90% of respondents. This indicates a strong preference for human oversight when assessing the quality and correctness of code produced by LLMs. The second most frequently used method is running test cases, reported by nearly 60% of participants. This suggests that many developers rely on automated testing to validate the functional behavior of AI-generated code. In contrast, comparing with existing implementations was the least used strategy, chosen by about 30% of respondents. This implies that while benchmarking against known solutions can be helpful, it is not the primary method developers rely on during evaluation.

These results highlight that despite the growing presence of LLMs in software development, developers still prioritize traditional validation techniques to ensure reliability and correctness.

Figure 3 presents the key aspects that developers prioritize when evaluating AI-generated code. The data highlights five major evaluation criteria: functional correctness, performance/efficiency, security vulnerabilities, readability/maintainability, and the potential for research and development (R&D).Among these, functional correctness emerged as the most critical factor, with approximately 77% of participants indicating it as a top priority. This emphasizes the importance developers place on whether the AI-generated code performs the intended task accurately. Readability and maintainability were also highly prioritized, selected by nearly 73% of respondents. This indicates that developers consider not only what the code does but also how understandable and maintainable it is in real-world development settings—an important consideration for collaborative and long-term projects. Performance and efficiency followed closely, with 70% of participants marking it as a key concern. Developers are clearly attentive to whether the code generated by AI performs optimally and utilizes system resources effectively. Security vulnerabilities were selected by about 63% of respondents, demonstrating that while security is a significant concern, it may not be as immediate a priority as correctness or performance when first evaluating generated code. In contrast, finding new approaches and contributions to R&D was the least selected option, with only a small percentage of respondents indicating it as a priority. This suggests that developers primarily view AI tools as functional aids rather than sources of novel research breakthroughs.

These results indicate that developers evaluate AI-generated code through a practical lens, focusing on correctness, usability, and performance, rather than innovation or exploration.

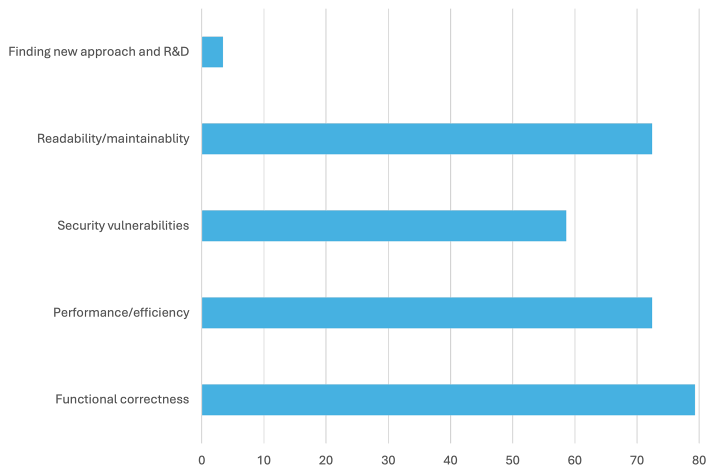


Fig. 3. Key aspects developers prioritize when evaluating AI-generated code

Summary of RQ$_1$: Developers validate LLM-generated code primarily through manual review and traditional debugging techniques such as print statements and logging. Many also use interactive tools like IDE debuggers to inspect code behavior step-by-step. Some developers prompt the LLM itself to revise or explain the generated code. Key validation criteria include functional correctness, readability, and edge case coverage. Overall, developers rely on their own judgment and experience to assess whether the code is reliable and suitable for use.

4.2 RQ2. What common patterns show up in errors made by LLM-generated code?

To address RQ2, we analyzed the types of errors that developers frequently encounter in LLM-generated code. Understanding these recurring issues is essential for evaluating the reliability and practicality of using LLMs in real-world software development tasks. Our findings suggest that while LLMs are capable of producing syntactically correct and readable code, they often exhibit consistent patterns of failure across different scenarios. These patterns include logical inconsistencies, inefficient implementations, unhandled edge cases, and occasional syntactic mistakes. By identifying and categorizing these common errors, we gain deeper insights into the limitations of current LLMs and the areas where future improvements are most needed.

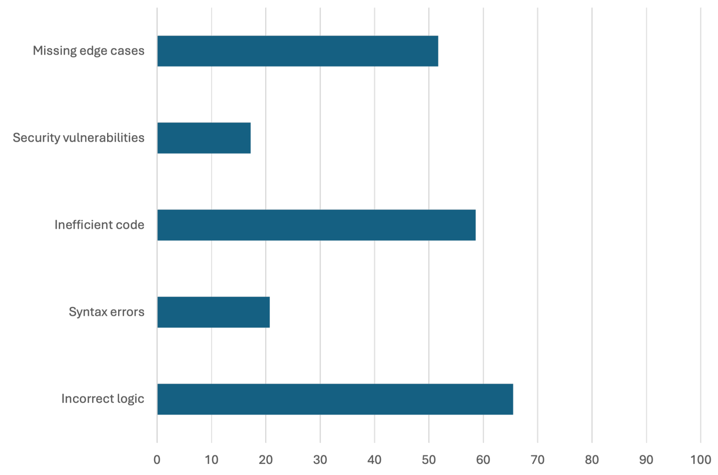


Fig. 4. Common errors produced by LLMs

Figure 4 addresses RQ2 by illustrating the most common error patterns developers identified in LLM-generated code. These responses highlight several recurring issues that can hinder the practical use of LLMs in software development workflows.

The most frequently reported error is incorrect logic, selected by more than 65% of participants. This suggests that, while LLMs can produce syntactically valid code, they often misinterpret problem requirements or apply flawed reasoning, resulting in code that does not behave as intended. This logical inconsistency represents a fundamental limitation in the model's ability to understand task semantics.

Inefficient code was also reported by a significant portion of respondents (approximately 58%), pointing to a pattern where LLMs generate functionally correct but suboptimal solutions. These may include unnecessary computations, verbose structures, or failure to use best practices for performance issues that become critical in production environments.

Another common error pattern involves missing edge cases, cited by over 50% of participants. This indicates that LLMs often fail to anticipate and handle atypical or boundary inputs, a key requirement for robust and dependable code.

Less frequently observed errors include syntax errors and security vulnerabilities, reported by fewer than 25% of participants. This suggests that while LLMs are largely successful at generating grammatically correct code, they struggle more with high-level reasoning than with surface-level syntax. Security issues, though less prevalent, remain a concern in contexts where code correctness alone is insufficient.

Overall, these findings reveal consistent and predictable patterns in LLM-generated code errors, reinforcing the need for careful human oversight and suggesting specific areas where LLM development and evaluation should focus to improve reliability and usability.

Summary of RQ$_2$: Developers observed several recurring error patterns in LLM-generated code. The most common issues included incorrect logic, inefficient implementations, and failure to handle edge cases. While syntax errors were less frequent, they still appeared occasionally, especially in complex prompts. Some developers also noted that the code often lacked consideration of project-specific context or dependencies. Overall, the errors reflected a gap in deep reasoning and contextual understanding, even when the code appeared syntactically correct.

4.3 Can humans differentiate AI generated code?

To explore whether developers can distinguish between AI-generated and human-written code, we asked participants: `"Do you think this code is AI-generated?"` Among the 29 responses, 72.4% answered "no," indicating they believed the code was written by a human, 13.8% answered "yes," and the remaining participants selected "don't know." Figure 5 represents the code we have given to the participant.

To better understand the reasoning behind their choices, we followed up by asking participants to justify their responses. Several respondents reflected on their own programming experiences when evaluating the origin of the code. For instance, one participant noted that the coding style resembled how they wrote C programs early in their career. They interpreted this familiarity as a sign that the code was likely human-written, suggesting that AI-generated code would typically follow a more polished or standardized structure:

> "When I started programming in the beginning days, I used C programming. I also write code like that. If it is done by any AI tools, its standard might be different." (P29)

Another participant believed that AI-generated code would include more detailed explanations or comments for each step, which were missing in the provided snippet. This absence of descriptive guidance led them to believe the code was written by a human:

> "If it was AI generated, it would contain more explanation for every step." (P21)

Other response response with the variable name is rubel, also there is no comment, usually AI generated code contains comments for every line. Furthermore, the variable was similar to human written.

We asked another question: `Do you think this code is human written?`, Almost 19 out of 29 participants respond with yes, 24.1% with no, and 10.3% with don't know. We also did the same for this question to justify their answer.

Several participants justified their belief that the code was human-written based on stylistic details they associated with human authorship. For example, one participant focused on the formatting style, particularly the presence of additional spaces, which they felt was atypical for AI-generated code:

> "Because in AI, it normally does not provide a large space like that in code." (P17)

Another participant associated the absence of explanatory comments with human-written code, suggesting that AI-generated code typically includes more descriptive elements or annotations:

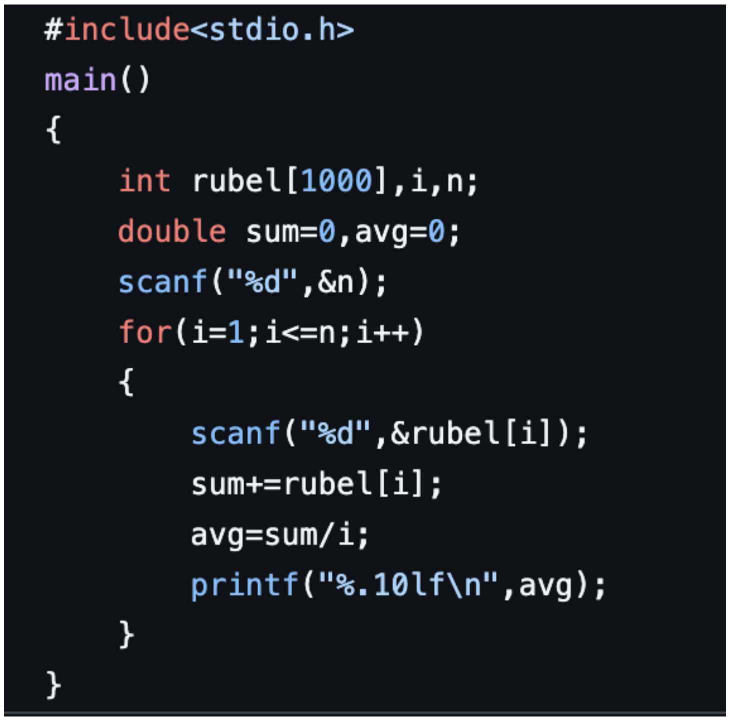
> "It looks like human written code as it does not contain any explanation." (P24)

Personal coding experience also influenced some participants' judgments. One respondent noted that the style of the code closely resembled how they typically write code, which led them to assume a human wrote it:

> "Because I used to write code like this." (P27)

Finally, the general structure and flow of the code were cited as a key indicator of human authorship. A participant highlighted the overall organization as a factor in their decision:

> "Overall organization of the code." (P27)



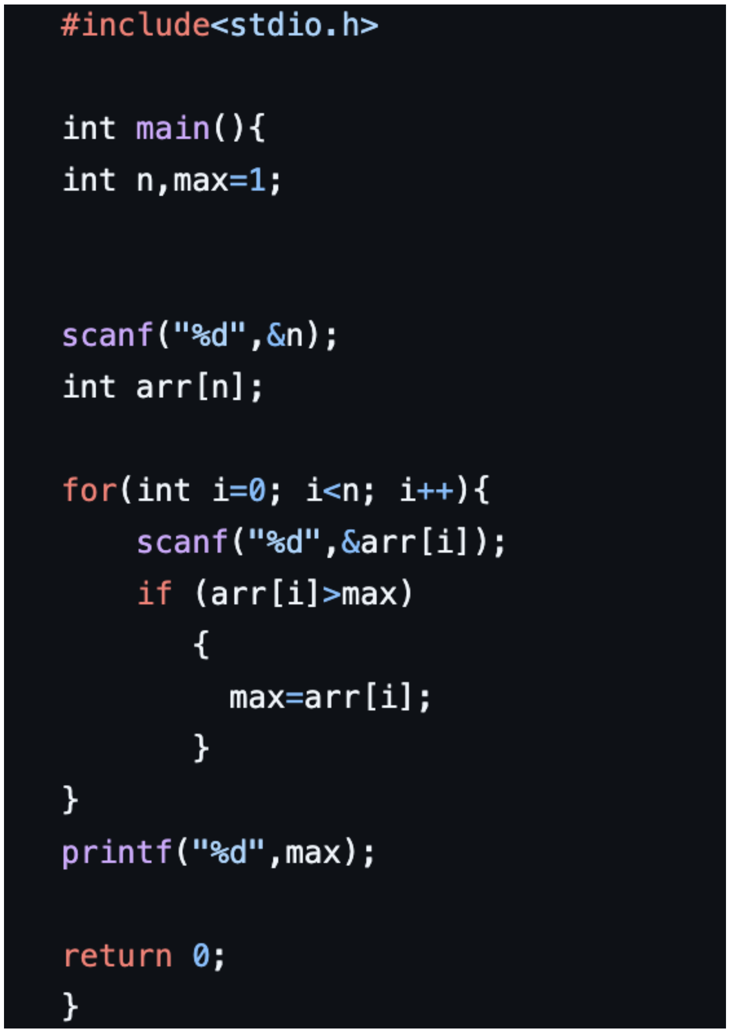
```c
#include<stdio.h>
main()
{
    int rubel[1000],i,n;
    double sum=0,avg=0;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        scanf("%d",&rubel[i]);
        sum+=rubel[i];
        avg=sum/i;
        printf("%.10lf\n",avg);
    }
}
```

Fig. 5. A code snippet written in C that calculates a running average of user-input integers.

Takeaways: Participants' justifications for evaluating whether code was AI-generated often reflected their own programming experiences and expectations of code style. Many associated the lack of comments and formatting irregularities with human-written code, believing that AI-generated code tends to be more standardized or overly explained. Some noted that the structure resembled how they wrote code early in their careers, leading them to infer human authorship. Others emphasized that AI-generated code typically includes more step-by-step explanations, which were missing in the given example. These reflections suggest that perceptions of "humanness" in code are shaped by developers' habits, expectations, and familiarity with both AI tools and real-world coding practices.

4.4 Improvements they want to see

Participants provided valuable feedback on how current tools for evaluating and debugging AI-generated code could be improved. Several developers suggested that tools should explain the generated code more clearly, especially highlighting the purpose behind specific logic or structure. Others emphasized the importance of built-in error detection features that can point out logical flaws, inefficiencies, or missing edge cases. A few participants also recommended integrating side-by-side comparisons with human-written code to assess code quality better. Overall, there was a shared expectation that future tools should offer



```
#include<stdio.h>

int main(){
int n,max=1;


scanf("%d",&n);
int arr[n];

for(int i=0; i<n; i++){
    scanf("%d",&arr[i]);
    if (arr[i]>max)
       {
         max=arr[i];
       }
}
printf("%d",max);

return 0;
}
```

Fig. 6. A code snippet written in C that finds the maximum value from an array

more guidance, context, and interactive support to help developers better understand and verify the AI-generated outputs.

Several participants pointed out the need for AI tools to better handle basic issues such as syntax and functionality. One participant briefly highlighted common problem areas:

> "Syntax errors, functional development and diversity. " (P12)

Others provided more detailed feedback on specific limitations of current LLMs. One participant emphasized that AI tools often fail to understand project-specific contexts, such as business logic and architectural constraints. They recommended using project documentation and historical data to enhance code relevance:

> "I've noticed that AI often lacks awareness of project-specific constraints, such as business logic and external dependencies. It would be helpful if AI could learn from project documentation, past commits, and architecture diagrams to generate more relevant and accurate code. " (P9)

Memory and continuity across conversations or prompts were also a recurring concern. One respondent expressed frustration that the AI did not recall previously provided information, suggesting the need for more persistent context awareness:

> "It should have more memory than I already talked about, my problem based on that it should give the solution. " (14)

Several participants also emphasized the value of explanations and examples to accompany code outputs. One noted that understanding why certain coding decisions were made would help them trust and learn from the tool more effectively:

> "It would be good when if AI generated text gives the code with a basic example and explanation of why it used this way. " (P19)

A few participants offered high-level suggestions for future improvements. One developer outlined several areas where current tools fall short and how improvements could make them more practical and trustworthy:

> "Enhancing AI-generated code evaluation and debugging tools requires improvements in contextual understanding, dynamic code analysis, seamless integration with development environments, and robust security measures. These advancements will lead to more accurate code suggestions, efficient debugging, and secure, maintainable software. " (P2)

Finally, the need for stronger capabilities in context awareness, error detection, and security analysis was reiterated by another respondent, who envisioned smarter tools that support both reliability and safety:

> "I would like to see AI-generated code evaluation and debugging tools with better context awareness, advanced error detection, and enhanced security vulnerability analysis to ensure more reliable and secure code. " (P21)

5 Limitation and Future Work

This study provides a foundational understanding of how developers interact with LLM-generated code, yet several limitations must be acknowledged. The analysis is based on self-reported survey data, which may not fully reflect actual usage or debugging practices due to recall bias or subjective interpretation. Moreover, the participant pool is geographically limited to developers in Bangladesh, which may limit the generalizability of the findings across different regions or development environments.

The study also lacks direct observational data or automated logging of developer activity, which could offer a more precise view of how LLMs are used in real-time. Additionally, our classification of debugging methods—such as print statements, interactive tools, or LLM-assisted fixes—may not capture the full complexity of debugging workflows. We also do not evaluate the technical quality of the generated code, such as measuring correctness or identifying specific vulnerabilities. Finally, this study captures only a snapshot in time; future work should explore how these practices evolve as LLM tools become more advanced and widely adopted.

Future research could benefit from broader, more diverse samples, mixed-method approaches including observational studies or code audits, and longitudinal data to better understand the long-term integration of LLMs into software development workflows.

6 Conclusion

This study explores how software developers use Large Language Models (LLMs) for code generation and the strategies they employ to debug the resulting code. Our findings show that while LLMs are becoming a common part of developers' workflows, they are not without challenges. Developers primarily rely on LLMs for generating small code snippets, but they often face issues such as incorrect logic, syntax errors, inefficiencies, and security vulnerabilities. To address these problems, developers use a range of debugging techniques—from traditional methods like print statements to more advanced tools and even prompting the LLM itself for corrections.

The results highlight a growing dependency on LLMs alongside a critical need for careful evaluation and debugging of their output. As LLMs continue to evolve, understanding how developers interact with these tools is essential for improving their design and reliability. This work contributes to that understanding and lays the groundwork for future research into more effective integration of LLMs into professional software development practices.

7 Acknowledgements

We would like to thank all the developers who took the time to participate in our survey and share their valuable insights. Their thoughtful responses made this study possible. We also appreciate those who helped distribute the survey within their networks, enabling us to reach a diverse group of participants. This work would not have been possible without their generous contributions and support. We would also like to thank Prof. Janice for her constructive feedback and valuable suggestions, which greatly helped improve the quality and focus of this work.

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] Open AI. 2022. Introducing ChatGPT. (2022). https://openai.com/index/chatgpt/

[3] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 162–168.

[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Giuseppe Desolda, Andrea Esposito, Francesco Greco, Cesare Tucci, Paolo Buono, and Antonio Piccinno. 2025. Understanding User Mental Models in AI-Driven Code Completion Tools: Insights from an Elicitation Study. *arXiv preprint arXiv:2502.02194* (2025).

[8] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[9] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. (2021). https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer

[10] GitHub. 2025. GithubCopilot. https://github.com/features/copilot. May 2025 version.

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[12] Saki Imai. 2022. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[13] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).

[14] Yichen Li, Yun Peng, Yintong Huo, and Michael R Lyu. 2024. Enhancing llm-based coding tools through native integration of ide-derived static context. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 70–74.

[15] Amanda Liu and Michael Coblenz. 2023. Debugging techniques in professional programming. Plateau Workshop.

[16] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2022. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1580–1598.

[17] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[18] Meta. 2025. Codellama. https://github.com/meta-llama/codellama. May 2025 version.

[19] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

[20] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.

[21] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2136–2148.

[22] OpenAI. 2025. ChatGPT. https://chatgpt.com/. May 2025 version.

[23] OpenAI. 2025. Codex. https://github.com/openai/codex. May 2025 version.

[24] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. doi:10.1109/SP46214.2022.9833571

[25] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[26] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 23–30.

[27] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. A Study on Developer Behaviors for Validating and Repairing LLM-Generated Code Using Eye Tracking and IDE Actions. *arXiv preprint arXiv:2405.16081* (2024).

[28] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[30] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. Core: Resolving code quality issues using llms. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 789–811.

[31] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[32] Zhou Yang, Jieke Shi, Prem Devanbu, and David Lo. 2024. Ecosystem of large language models for code. *ACM Transactions on Software Engineering and Methodology* (2024).

[33] Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352* (2019).