Towards Effective Validation and Integration of LLM-Generated Code

Ningzhi Tang, ntang@nd.edu, *University of Notre Dame, Notre Dame, IN, USA*

***Abstract*—Recent advances in large language model (LLM)-based code generation tools have shown the potential to lower the barrier to programming and improve developer productivity. However, validating the generated codes and integrating them into projects present significant challenges, especially for developers with limited expertise or domain-specific knowledge. I present a novel approach to help developers understand and modify LLM-generated code to align with their intentions, informed by my empirical study of developer behaviors when working with LLM code generation models. In the future, I propose to extend the work to include support for project-level code validation, dynamic program behavior integration, developer behavioral and cognitive context modeling, and computer science education.**

***Index Terms*—large language model, debugging, developer behavior, code generation.**

I. Background

The increasing use of large language model (LLM)-powered programming tools, such as GitHub Copilot, has shown the potential to generate high-quality code snippets, thus improving productivity [1]. However, the generated code may not always align with the developers' intentions, posing challenges for integrating the code into their projects [2]. Furthermore, developers' intents should be gradually clarified during the programming process, necessitating good interaction techniques to iteratively validate and modify LLM suggestions [3].

Compared to traditional debugging, developers face unique challenges when validating and integrating LLM-generated code in three aspects: code understanding, information acquisition, and attention focus. *Firstly*, developers must exert additional effort to understand the code, as it can be lengthy [3], contain excessive control structures, or use unfamiliar APIs [1]. Conversely, they would be already familiar with the structure and details of code written by themselves. *Secondly*, LLMs typically generate continuous local code snippets (e.g., a method or class) based on existing contexts. This requires developers to quickly validate and decide whether to accept, modify, or discard them. In contrast, traditional debugging may require more effort in seeking, relating, and collecting information throughout the project to locate bugs [4]. *Thirdly*, our empirical study found that, compared to human-written code, LLM-generated code often excels in detail, but falls short in logical structure, necessitating different validation and modification strategies [5].

These unique challenges in the code validation and integration process may be even more difficult for novice developers, turning LLM into a "liability" instead of an "asset" [6]. LLMs hold the promise of lowering the barriers for novice developers, or developers who are only experts in other languages/frameworks, by allowing them to describe their needs in natural language and prompt the LLM for the desired output. However, compared to experts, they may lack sufficient expertise to understand the code, determine its correctness, and make modifications [7]. This gap calls for research efforts in empirical studies and tool development for code generation.

Although LLMs introduce new challenges, they also show the potential to alleviate them, though not perfectly. From the *validation* perspective, our previous study found that developers use Copilot to generate natural language explanations for code, helping them understand and validate it [5]. However, these generated explanations are scattered and have an uncontrolled level of detail. From the *integration* perspective, 40% of developers modify or rewrite prompts to make the LLM edit or regenerate the code [1]. However, the descriptions for regenerating requirements may still be ambiguous and fail to express developers' intent accurately. These issues imply the need for research into dynamic levels of abstraction of developer intents and better support for developers to iteratively disambiguate their intents with code generation models.

II. Our Approach and Progress to Date

In this section, we present our progress. We developed CodeGRITS [8], an eye-tracking and IDE behavior-logging toolkit to track developer behavior. Using it, we studied how developers validate and repair LLM-generated code. From these insights, we proposed a tool to assist in this process.

A. CodeGRITS: Developer Behavior and Eye Tracking in IDE

Tracking developers' programming behavior can help us quantitatively understand their software development process [9]. We developed CodeGRITS, a JetBrains plugin that can simultaneously track developers' IDE interactions and eye movements. It collects raw gaze data from eye-tracking devices and interprets them into the corresponding code tokens. The tracked data are exported in XML files. CodeGRITS supports multiple IDEs (e.g., IntelliJ IDEA, PyCharm) and interprets all IDE-supported languages' abstract syntax tree (AST). We have open-sourced CodeGRITS[1] with a website (e.g., usage guide, output format) and JavaDoc documentation.

B. Empirical Insights into LLM Code Validation and Repair

To investigate how developers validate and repair LLM-generated code and to examine the impact of code provenance awareness during these processes, we conducted a lab study with 28 participants, who were tasked with validating and repairing code generated by GitHub Copilot in three software projects [5], [10]. We collected data using CodeGRITS,

[1]https://codegrits.github.io/CodeGRITS/

combined with cognitive workload assessments and semi-structured interviews. Our results indicate that without explicit information, developers often fail to recognize the provenance of the code, which can affect their performance, behaviors, and cognitive workload. Developers generally employ similar validation and repair strategies for LLM-generated code but exhibit distinct behaviors such as frequent switching between code and comments, and differing attentional focuses.

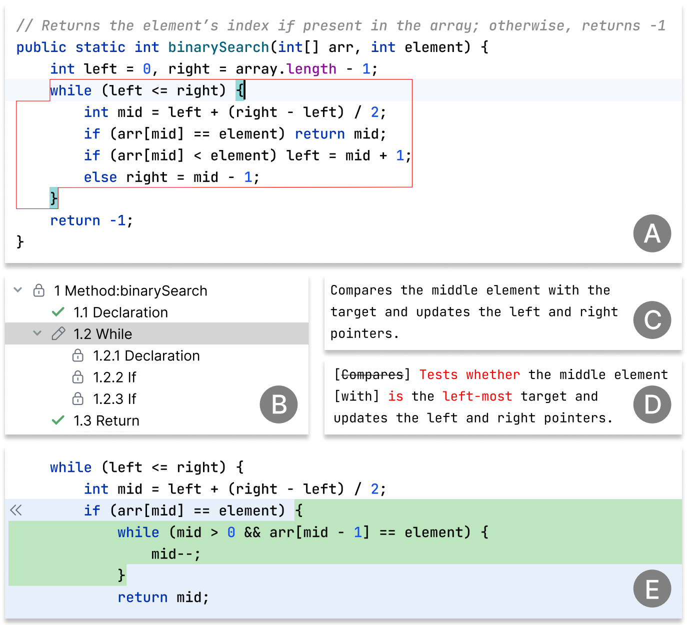
C. Multi-Level Explanation & Procedurally Prompted Editing

Based on insights from previous research and our study, we designed an IDE plugin (Fig. 1) to help developers understand LLM-generated code through multi-level code summarization and further edit it using procedural prompts.

(a) *Multi-Level Explanation.* Given a segment of LLM-generated code, e.g., a method **(A)**, our prototype decomposes it into a tree structure based on its AST. Then it generates natural language explanations for each node to aid understanding **(B)**. Developers can freely explore different levels of explanation by clicking on either the code or a tree node. The corresponding code snippet will be highlighted with a red box, and the corresponding explanation will be displayed in a panel **(C)**.

(b) *Procedurally Prompted Editing.* Conversely, developers can edit the multi-level code summaries to express their intentions, using these modifications as prompts for the LLM to refine unsatisfactory code **(D)**. Changes to the summaries will be highlighted in red text. The procedural prompts can more accurately express developers' intents compared to merely changing the original descriptive prompts for the LLM. The differences between the LLM-edited and previous code will be visualized, allowing developers to decide whether to accept the changes **(E)**.

The multi-level summaries allow developers to understand the code at various detail levels. Developers can also modify it to procedurally prompt the LLM to refine the code, enabling more accurate expressions of their intentions. We will conduct a usability study to evaluate our prototype's effectiveness.



```
// Returns the element's index if present in the array; otherwise, returns -1
public static int binarySearch(int[] arr, int element) {
    int left = 0, right = array.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == element) return mid;
        if (arr[mid] < element) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Compares the middle element with the target and updates the left and right pointers.

[~~Compares~~] Tests whether the middle element [with] is the left-most target and updates the left and right pointers.

```
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == element) {
            while (mid > 0 && arr[mid - 1] == element) {
                mid--;
            }
            return mid;
```

Fig. 1. Framework of the designed prototype.

III. Proposed Future Work

Our proposed system can potentially help developers validate and integrate LLM-generated code, but it currently presents several limitations in its scope and applications. We plan to extend it in four areas: (1) project-level code validation, (2) dynamic program behavior integration, (3) developer context personalization, and (4) computer science education.

Firstly, we will expand the scope of our system from local code snippets to cross-file, project-level code. While developers currently use LLMs mainly for generating the former, future tools may handle larger codebases [9]. Exploring this expansion will further enhance their programming workflow.

Secondly, we will expand our system to understand both static and dynamic program behaviors. Besides static code, dynamically understanding the program, e.g., inserting print statements or using an IDE debugger to inspect runtime behavior, is a crucial part of debugging [10]. Similarly, in front-end development, developers verify implementations by inspecting webpages or GUI visualizations. For AI modeling, debugging involves logging changes in loss, accuracy, or other metrics.

Thirdly, we plan to design personalized systems based on developers' behavioral and cognitive contexts [8]. CodeGRITS can track developers' IDE interactions and eye movements, providing rich contextual information about their behavioral focus and cognitive states. Incorporating such context can offer personalized support for their software development processes.

Finally, we are also interested in exploring how our system can help novices learn programming. Validating LLM-generated code may be more challenging for beginners, hindering the democratization of programming [7]. We will design educational systems to teach them to use LLMs effectively, ensuring they become "assets" rather than "liabilities".

References

[1] J. T. Liang *et al.*, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *ICSE*, 2024.
[2] H. Mozannar *et al.*, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *CHI*, 2024.
[3] S. Barke *et al.*, "Grounded copilot: How programmers interact with code-generating models," *ACM Program. Lang.*, no. OOPSLA1, 2023.
[4] A. J. Ko *et al.*, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, 2006.
[5] N. Tang *et al.*, "Developer behaviors in validating and repairing llm-generated code using ide and eye tracking," in *VL/HCC*, 2024.
[6] A. M. Dakhel *et al.*, "Github copilot ai pair programmer: Asset or liability?" *J. Syst. Softw.*, 2023.
[7] S. Nguyen *et al.*, "How beginning programmers and code llms (mis) read each other," in *CHI*, 2024.
[8] N. Tang *et al.*, "Codegrits: A research toolkit for developer behavior and eye tracking in ide," in *ICSE-Companion*, 2024.
[9] A. Bansal *et al.*, "Programmer visual attention during context-aware code summarization," *arXiv preprint arXiv:2405.18573*, 2024.
[10] N. Tang *et al.*, "An empirical study of developer behaviors for validating and repairing ai-generated code," in *PLATEAU*, 2023.