

Exploring Direct Instruction and Summary-Mediated Prompting in LLM-Assisted Code Modification

Ningzhi Tang*, Emory Smith*, Yu Huang[†], Collin McMillan*, Toby Jia-Jun Li*

*{ntang, esmith36, cmc, toby.j.li}@nd.edu, [†]yu.huang@vanderbilt.edu

*University of Notre Dame, Notre Dame, IN, USA [†]Vanderbilt University, Nashville, TN, USA

Abstract—This paper presents a study of using large language models (LLMs) in modifying existing code. While LLMs for generating code have been widely studied, their role in code modification remains less understood. Although “prompting” serves as the primary interface for developers to communicate intents to LLMs, constructing effective prompts for code modification introduces challenges different from generation. Prior work suggests that natural language summaries may help scaffold this process, yet such approaches have been validated primarily in narrow domains like SQL rewriting. This study investigates two prompting strategies for LLM-assisted code modification: Direct Instruction Prompting, where developers describe changes explicitly in free-form language, and Summary-Mediated Prompting, where changes are made by editing the generated summaries of the code. We conducted an exploratory study with 15 developers who completed modification tasks using both techniques across multiple scenarios. Our findings suggest that developers followed an iterative workflow: understanding the code, localizing the edit, and validating outputs through execution or semantic reasoning. Each prompting strategy presented trade-offs: direct instruction prompting was more flexible and easier to specify, while summary-mediated prompting supported comprehension, prompt scaffolding, and control. Developers’ choice of strategy was shaped by task goals and context, including urgency, maintainability, learning intent, and code familiarity. These findings highlight the need for more usable prompt interactions, including adjustable summary granularity, reliable summary-code traceability, and consistency in generated summaries.

Index Terms—code modification, AI-assisted programming, prompting strategies, summary-mediated interaction

I. INTRODUCTION

The rise of large language models (LLMs) has transformed how developers interact with code [1], [2]. While much attention has been given to how LLMs generate new code implementations from natural language specifications [3], [4], their role in *modifying existing code*, a common yet cognitively demanding task, remains comparatively underexplored. Unlike code generation, modification requires developers to interpret existing logic, align their intent with that logic, and ensure that changes do not introduce regressions [5]–[7]. Despite complexity, recent evidence shows that editing code via natural language prompts accounts for a substantial share (831/4,188) of code-related LLM usage [8], [9], and this functionality is increasingly supported by production tools like GitHub Copilot Chat¹ and Cursor². This shift also reflects a broader

trend toward “vibe coding,” where developers delegate code modifications to AI without making manual edits³.

In such workflows, prompts serve as the primary interface through which developers communicate intent to LLMs [10], [11]. However, crafting an effective prompt remains difficult due to the inherent open-endedness of natural language [12], [13], uncertainties in LLM predictions [14], [15], and the often vague or evolving nature of developer intentions [12], [16]. These challenges become especially prominent in code modification [17], where the prompt must navigate complex, pre-existing code contexts and articulate precise changes. Developers struggle to understand unfamiliar logic, locate modification points, and express intent clearly [18], while LLMs must respond accurately without introducing side effects [14].

One promising approach to easing prompt construction is to scaffold it through editable natural language summaries of existing code, which serve as intermediate representations. Previous work in constrained domains, such as SQL rewriting [19], [20] and spreadsheet data analysis [13], [21], has explored this strategy. They show that the summaries help bridge the gap between developer intent and LLM outputs, while also improving code comprehension and giving users more control over prompt formulation. However, these approaches have largely been limited to domains with small-scale code and semantically constrained programming languages, where summary generation is achieved through predefined rule-based mappings from code to linearized instructions. Their application to broader general-purpose programming scenarios remains underexplored. In these scenarios, both code semantics and developer modification intentions are more diverse and less structured. Additionally, general-purpose code often involves abstract control flow and cross-cutting dependencies, making rule-based summaries poorly aligned with human reasoning and task-specific goals.

To bridge this gap, our work investigates how developers use two prompting techniques in *general-purpose programming* to understand the processes, trade-offs, and design implications of LLM-assisted code modification. This is enabled by LLMs’ ability to generate editable summaries from *arbitrary code*. As illustrated in Fig. 1, the two techniques are:

- **Direct Instruction Prompting:** Freely writing natural language instructions to describe desired changes.

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

²<https://www.cursor.com/>

³<https://x.com/karpathy/status/1886192184808149383>

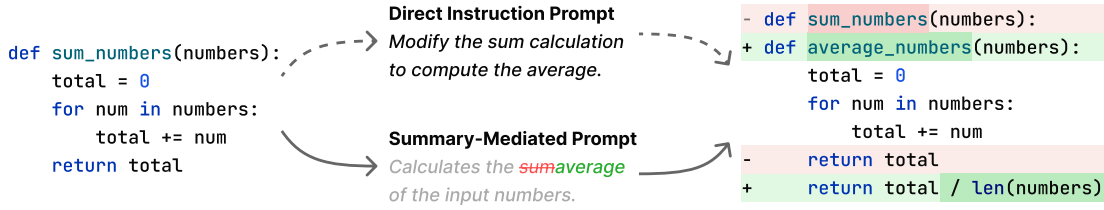


Fig. 1. An illustrative example of direct instruction (dashed arrows) and summary-mediated prompting (solid arrows) for LLM-assisted code modification.

- **Summary-Mediated Prompting:** Editing a natural language summary of existing code to specify new behavior.

We conducted an exploratory study with 15 developers (10 graduate students and 5 professionals), who performed code modification tasks across three scenarios, each containing multiple sub-requirements. To facilitate the study, we developed a prototype named PASTA (Prompt-Assisted Software TrAnsformation), integrated into JetBrains IDEs, which supports both prompting techniques. We collected interaction logs for each session. Finally, we conducted semi-structured interviews in which participants reflected on their experiences.

In each task, we asked participants to switch freely between prompting approaches and develop hybrid strategies. Unlike controlled comparisons that focus narrowly on prompting effectiveness, our open-ended design enables a richer understanding of developer behavior. This design allows us to investigate not only *which* prompting methods developers prefer, but also *how* and *why* they adapt their strategies to different task contexts, and *what* usability improvements could make summary-mediated prompting more effective in practice. Our study investigates the following research questions:

- **RQ1:** What are developers’ prompting processes and perceived characteristics of the two prompting techniques?
- **RQ2:** What task goals and contextual factors influence developers’ prompting choices?
- **RQ3:** What usability considerations and design opportunities arise in summary-mediated prompting?

We highlight the following findings.

- 1) Developers used generated summaries to understand code, narrow down modifications, and evaluate outcomes through execution or semantic reasoning. When specifying changes, they navigated trade-offs: direct instruction prompting offered flexibility and simplicity, while summary-mediated prompting provided precise terminology, broader context, and control over untouched code. Despite requiring more reading, summaries reduced typing and forced deeper understanding.
- 2) Prompting strategies varied depending on task goals and contextual factors: developers favored direct instructions when changes were urgent, simple, large in scope, or clearly defined, and used summaries to support comprehension and control when aiming for long-term maintainability, learning, or bug avoidance, or when working with unfamiliar or complex code. These needs for deeper understanding and careful validation were

amplified in industrial settings, where codebases are large, interdependent, and beyond LLM knowledge.

- 3) The usability of summary-mediated prompting depended on granularity and alignment with developers’ goals; participants called for structured formats, flexible detail levels, summary-code mappings, and consistent outputs to support efficient comprehension and accurate edits.

II. RELATED WORK

A. LLM-Assisted Code Modification

Code modification accounts for a substantial portion of software development effort [6], [7]. Researchers have focused on automating specific modification tasks using LLMs, including bug fixing [22]–[25] and refactoring [26]. For instance, Xia *et al.* [27] improved patch generation in program repair by integrating real-time feedback into LLMs. In more general settings, datasets like InstructCoder [28] and CANITEDIT [9] framed LLM code modification as mapping from (*original code + instruction*) to *modified code*. Cassano *et al.* [9] found that descriptive instructions consistently outperform vague ones, but impose a high developer workload. Shi *et al.* [29] similarly demonstrated the technical feasibility of using LLM-generated summaries for synchronized code editing, focusing on summary quality in surface-level editing cases.

While these works highlight what LLMs can accomplish for code modification, they pay limited attention to how developers actually formulate prompts in practice. In production tools like Cursor, prompting typically occurs through inline code selection or chat-based interfaces. A growing trend, popularized by Karpathy as “vibe coding,” involves describing modifications in natural language and delegating implementation details to LLMs, often without manually reviewing the resulting code. Our study fills this gap by examining developers’ prompting strategies in such workflows.

B. Developers’ Interaction with LLMs

User-centered studies have extensively examined how developers interact with LLM-powered tools and their perceived effectiveness [1], [2], [4], [30]–[34]. LLMs have been shown to improve productivity [32], [35], but the code they generate often contains functionality issues [1], [33], requiring significant developer effort to verify and repair [2], [30], accounting for up to 38% of developers’ time [34]. Despite the prevalence of code modification in LLM workflows [9] and its support in production tools, few studies have examined how developers construct and adapt prompts in these contexts.

Non-expert developers face additional barriers, including limited vocabulary for prompting and difficulty understanding generated code [36]–[38]. Modifying unfamiliar or legacy code imposes additional cognitive overhead, as developers must first comprehend existing code before making changes [39], [40]. To investigate these interactions, our study focuses on code editing tasks involving *technically unfamiliar frameworks* (e.g., TensorFlow, D3.js) in realistic programming scenarios.

Sarkar *et al.* [15] characterize the difficulty of iteratively aligning user intent with LLM outputs as an “abstraction matching” problem [13], [15]. This echoes Don Norman’s “gulf of execution” [41]: getting the system to do what the user intends. Several systems have addressed this gap in code generation through structured interaction design [42]–[47]. They target new code generation rather than code modification, which requires interpreting existing logic and making iterative changes. We fill this gap by examining interactive prompting for code transformation.

C. Prompting Strategies for LLM Programming

Many LLM-based programming systems rely on natural language prompting [10], [48], [49], but prompting remains cognitively demanding, especially for non-experts [1], [14], [50]–[52]. Users often struggle with getting started [16] and expend significant metacognitive effort throughout the process [12], [17], including decomposing tasks and adjusting prompting strategies over time. Various prompting techniques have been proposed to better align LLM outputs with human intent [11], [53]–[55]. For instance, few-shot prompting [10] and chain-of-thought reasoning [53] enhance performance by providing contextual examples and intermediate reasoning steps. In programming contexts, developers use tactics such as giving examples, stating coding goals, or iterating through multi-turn conversations [17], [56], [57].

Recent studies have explored using *intermediate representations* to scaffold prompting, such as sketches [58], visual data operations [46], and editable natural language summaries. Liu *et al.* [13] proposed “grounded abstraction matching,” translating Pandas code into natural-language utterances to support prompt refinement. Similarly, Tian *et al.* [19], [20] used stepwise SQL explanations to help users identify and correct errors. These approaches improved task accuracy and user confidence but were limited to rule-based mappings in constrained domains. Rawal *et al.* [59] further showed that natural-language versions of code improved debugging performance in algorithmic tasks, suggesting the broader potential of summaries to support modification. However, it remains unclear how well these techniques apply to general-purpose programming, where code semantics and developer intent are more complex, motivating us to investigate how they function in real-world development contexts.

III. STUDY DESIGN

We conducted an exploratory study⁴ to understand how developers modify existing code using two prompting tech-

niques: *Direct Instruction Prompting*, where users write natural language commands, and *Summary-Mediated Prompting*, where they edit LLM-generated summaries. Rather than running a controlled comparison, we aimed to observe how participants naturally adopt, combine, and adapt these strategies in realistic, iterative coding scenarios.

A. Programming Tasks

We created three tasks across diverse domains (deep learning, data visualization, and web development), implemented using Python, JavaScript, HTML, and CSS. Each task involved a familiar domain to the participants but incorporated programming techniques that they are less familiar with, allowing us to examine how technical familiarity influences prompting behavior [36]. Task design followed a structured, goal-driven process. One author first defined the domain scope and target technique. The programming scenarios were iteratively refined by the research team to balance realism and controllable complexity. Each task had a baseline implementation, followed by exploratory ideation to identify plausible enhancements in algorithm design, UI, or interaction. We selected final tasks based on: (1) completion within 20 minutes; and (2) coverage across diverse domains, code structures, and modification types, including both logic- and interface-level changes.

- **Task 1. TensorFlow Autoencoder (deep learning):** Modify a TensorFlow-based sparse autoencoder to update its model architecture, loss function, and learning schedule.
- **Task 2. D3.js LineGraph (data visualization):** Enhance the X-axis markers and point labels of a D3.js line graph visualizing skin cancer detection accuracy.
- **Task 3. Chrome Extension Translator (web development):** Improve the button effects, UI style, and front-end to back-end communication of a Chrome extension that integrates OpenAI’s API to perform machine translations.

B. Participants

We recruited 15 participants (10 male, 5 female; ages 22–30, $M = 25.1$, $SD = 2.46$) through purposive sampling [60]. All had prior experience with Python and JavaScript and were majoring in computer science or relevant engineering fields. The group included 10 graduate students and 5 professional developers, with an average of 6.94 years of programming experience. Each received a \$66 Amazon gift card as compensation for their time. All participants had used LLM tools for programming and code modification tasks (e.g., bug fixing and refactoring), such as ChatGPT (12), GitHub Copilot (11), Claude (9), DeepSeek (6), and Cursor (4).



Most participants had general domain experience across the three task scenarios (13/15, 15/15, and 14/15), but technical familiarity varied: for TensorFlow, 3 had never used it, 9 were unfamiliar, and 3 were proficient; for D3.js, 9 had never used it, 5 were unfamiliar, and 1 was proficient; for Chrome Extension, 5 had never tried, 7 were unfamiliar, and 3 were proficient. A summary of each participant’s demographics and technical background is presented in Table I.


⁴The study protocol has been approved by the IRB at our institution.

C. PASTA: Study-Enabling Prototype

To investigate how developers use two prompting techniques in real-world code modification, we developed PASTA⁵, a JetBrains IDE plugin that supports both approaches.

1) *Interface*: Fig. 2 shows the interface of PASTA. Developers begin by selecting a region of code in the editor using the mouse (left panel), then specify their intended modification using one of two prompting methods (right panel).

In summary-mediated prompting, clicking  Retrieve Summary generates a 1-3 sentence natural language description of the selected code. The summary adapts to the length and complexity of the code. Pilot testing revealed that overly detailed summaries hindered usability, so we adopted a concise summarization policy. The generated summary appears in the top text field and can be freely edited to express the intended change. Developers can click  Diff Summaries to highlight **insertions** and **deletions**, aiding summary revision. In direct instruction prompting, developers write a natural language command in the bottom text field to specify the desired modification directly.

Clicking  Commit Prompt in either mode submits the selected code, the file context, and either the edited summary or the free-form instruction to the LLM. The returned code is shown in a diff view with line- and token-level highlights, enabling developers to inspect, validate, and selectively accept changes. A loading indicator signals that the LLM is processing the prompt, providing users with timely visual feedback.

2) *Implementation*: PASTA is implemented as a plugin using the official IntelliJ SDK⁶ and is compatible with all JetBrains IDEs, including PyCharm and WebStorm, which were used in our study. The interface is integrated as a tool window panel, with functional buttons implemented via the `AnAction` interface. The summary diff feature is supported by the Java Diff Utils library⁷, while code diffing leverages the default diff package provided by the SDK. LLM-powered functionality is built on OpenAI's GPT-4o⁸ chat completions API. To improve in-context learning and ensure output consistency, we include a set of few-shot examples in each prompt.

Unlike rule-based summary systems developed for SQL or tabular data [13], [20], our approach leverages LLM generalization to describe arbitrary code without predefined templates. Although this sacrifices strict one-to-one correspondence between code and summary, it allows broader applicability across diverse programming domains.

3) *Design Decisions*: In designing PASTA, we made several key decisions to support effective prompting while maintaining compatibility with existing AI-assisted coding workflows.

Selection-Based Prompting. Contemporary AI code editors, such as Cursor and GitHub Copilot Chat, typically offer two ways to specify target code for modification: (1) using the @ symbol in a chat interface to reference code, or (2) directly

selecting code in the editor before entering a prompt. We adopt the latter selection-based approach for its simplicity, ease of integration, and natural alignment with summary generation, enabling a fair comparison between prompting techniques.

Context-Aware Interaction. Prior work shows that incorporating relevant context, e.g., current file [61], [62], call graphs [63], can improve LLM output for code tasks. Since our focus is not on comparing context strategies, we adopt an approach that includes the full content of the current file as context. This method is simple and effective for lightweight tasks, whereas how to effectively integrate broader context in large repositories remains a challenging and open question⁹.

D. Study Protocol

1) *Settings*: We conducted this study on a Windows 11 computer with PyCharm and WebStorm 2024.3 installed, along with our enabling research prototype, PASTA (Section III-C). Participants accessed the computer via Zoom's remote screen control. We instructed participants to view the study instructions on a separate device (e.g., iPad or second screen) to save screen space and prevent direct copy-pasting into prompts. During the study, we collected interaction logs from PASTA, including timestamps, selected code snippets, prompts, and LLM-generated summaries or modifications.

2) *Procedure*: Each study session lasted approximately 2 hours. We asked participants to sign a consent form and complete a pre-study questionnaire collecting their demographic information and prior experience with programming and LLM usage. We then provided a brief introduction to the study objectives, followed by a short tutorial on the two prompting ways through PASTA. Before starting the programming tasks, participants were given 5 minutes to try out both prompting methods on an example task.

The order of the three tasks was randomized to mitigate learning effects [64]. For each task, participants first read an instructional document describing the background of the task and the required modifications. They were then given 20 minutes to complete each task. Participants were free to use either prompting technique; however, they were encouraged to use PASTA to prompt (in either technique) rather than editing manually. Participants were free to ask the study coordinator how to run the code and check the results. We recorded task completion and logged the time taken for each task.

After completing each task, participants filled out a NASA Task Load Index (NASA-TLX) questionnaire [65] to self-report their cognitive workload. NASA-TLX is a widely used subjective tool that measures the user's perceived workload when performing a task. It includes six dimensions: Mental Demand, Physical Demand, Temporal Demand, Performance, Effort, and Frustration. We also asked participants to self-report their code understanding and editing effort.

After completing all tasks, participants filled out a utility evaluation questionnaire with Likert scale items to assess their experience with each prompting technique, including

⁵Code available at: <https://github.com/TTangNingzhi/PASTA>

⁶<https://plugins.jetbrains.com/docs/intellij/welcome.html>

⁷<https://java-diff-utils.github.io/java-diff-utils/>

⁸<https://openai.com/index/hello-gpt-4o/>

⁹<https://x.com/karpathy/status/1937902205765607626>

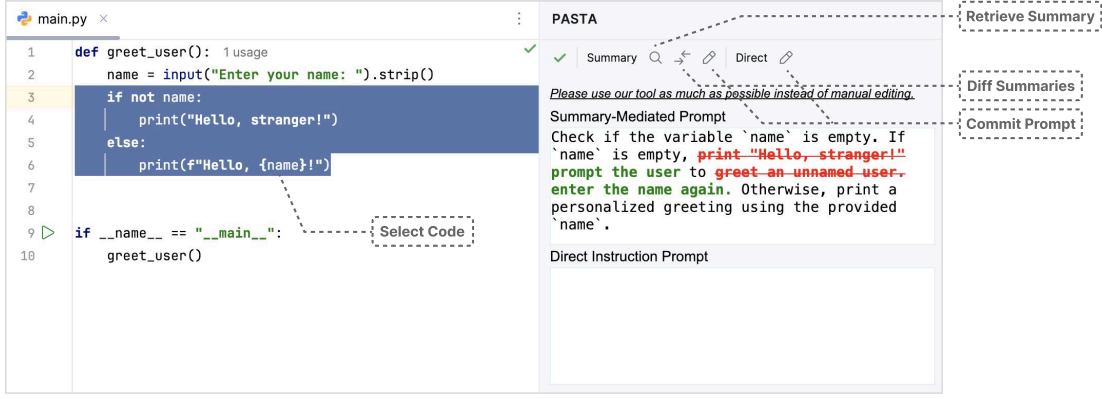


Fig. 2. Interface of the prototype plugin PASTA, integrated into JetBrains IDEs (e.g., PyCharm, WebStorm).

their ease of use, effectiveness in code comprehension and modification, sense of control, and satisfaction with LLM-generated modifications. Finally, we conducted a 25-minute semi-structured interview of participants' experiences when using the two prompting techniques. The interview covered topics including prior experience with LLM-based code modification, challenges in prompting and validation, perceived control over LLM-generated changes, and preferences between the two prompting techniques. We also explored participants' views on the usability of generated summaries and potential improvements to summary-mediated prompting.

E. Data Analysis

We transcribed all post-study interviews and conducted a qualitative analysis following open coding procedures [66], [67]. In the first stage, two authors collaboratively reviewed interview transcripts and extracted 174 meaning-rich segments related to developers' prompting strategies, decision-making processes, and challenges. These segments served as the units of analysis. Based on these segments, the authors developed an initial hierarchical codebook, consisting of high-level themes and nested subcodes. The codebook was iteratively refined through discussion. In the second stage, both authors independently coded all segments using the initial codebook. We assessed inter-coder reliability using both agreement rate and Cohen's κ , which yielded an agreement of 63.8% and $\kappa = 0.626$, indicating substantial consistency [68]. Disagreements were resolved through discussion.

To analyze the interaction logs of the two prompting choices, we used Student's t-test for group comparisons with unequal sample sizes, reporting both the mean and p -value. For Likert-scale utility ratings, we applied the Wilcoxon signed-rank test, given the ordinal nature of the data and small sample size, and reported the median and p -value.

IV. STUDY RESULTS

A. Prompting Processes and Perceived Characteristics

In this section, we present a process model of LLM-assisted code modification (Fig. 3), highlighting how the two prompting strategies support different stages, from understanding and

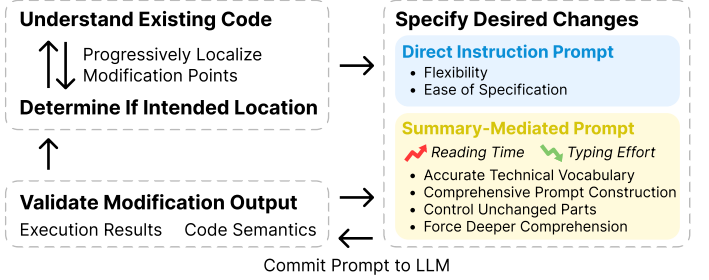


Fig. 3. A process model of developer-LLM interaction for code modification, highlighting complementary trade-offs between prompting strategies.

localization to specifying and validating changes, and analyze how summaries can first aid comprehension and later serve as a medium for constructing prompts during modification.

1) *Summaries Scaffold Code Understanding and Localization*: Before specifying changes, developers first need to understand the existing code and progressively localize the modification point (Fig. 3), a process where summaries played a key role. As P5 summarized, “Summary helped in two ways: understanding the code and deciding which part to modify.”

Summaries support comprehension of unfamiliar code (Q1). Participants found summaries helpful for understanding code that they hadn’t authored themselves (P1, P2, P3, P5, P6, P8, P9, P12, P13), noting they “saved much time compared to looking into the documentation. (P15)” P3 added, “Through the natural language description, I can infer where the bug might be,” echoing findings that natural language improves bug detection by making program logic more explicit [59]. Quantitative results also support this role: summaries were rated more helpful for understanding code (Fig. 4 Q1, 6.0 > 4.0, $p = 0.0044 < 0.01$). Usage logs confirm frequent summary use, averaging 3.07, 4.00, and 5.33 invocations per task.

Summaries enable developers to progressively localize the modification point and determine if it was the intended location (P1, P3, P5, P8, P9, P14). “Figuring out where to modify is challenging” (P14) [18], but summaries helped narrow the scope. As P9 described, “I start by summarizing the entire file to understand its purpose, then gradually narrow

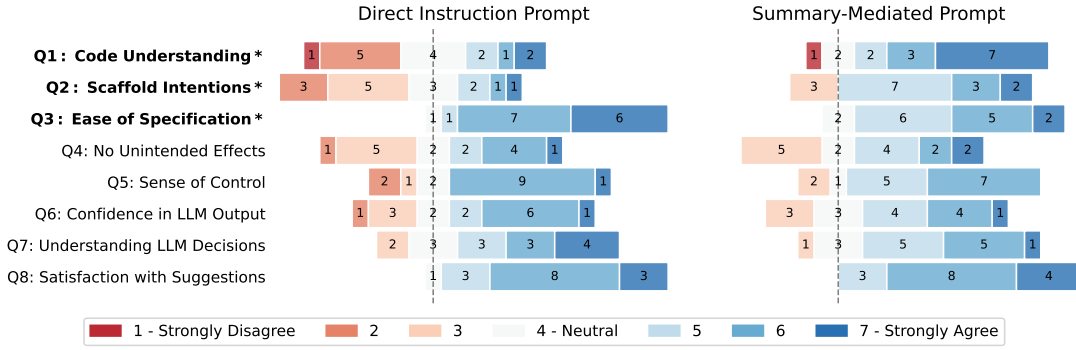


Fig. 4. Comparison of the usability of the two prompting techniques. Q1-Q3 were bolded with asterisks (*) to indicate statistically significant differences ($p < 0.05$), based on Wilcoxon signed-rank tests. No significant differences were observed for Q4-Q8.

the scope to a 50-line function that reveals internal logic”. Similarly, P14 used the summaries to “tell me what each line does” before locating the target section. As they refined the scope, participants continuously checked whether the selected code aligned with their editing intent. As P1 reflected, “I might think, ‘Okay, this function is not the one that I should modify. I need to switch to another function and check.’” This progressive narrowing process echoes cognitive models of code navigation and bug localization [5], [69], where the iterative generation of summaries serves as a lightweight strategy to support hypothesis formulation and validation.

Despite reading effort, developers frequently used summaries before choosing prompting strategies. Some participants (P1, P2, P13, P15) noted that long or detailed summaries slowed their workflow; P2 remarked, “As a non-native English speaker, it took me some time to read the summarization response.” Structuring summaries and adding visual code mappings were suggested to improve comprehension (see Section IV-C2). Despite the overhead, many participants (P1, P2, P5, P8, P9, P12, P13, P14) reported a common pattern: they first used summaries to understand the code, narrow the scope, and then selected prompting strategies accordingly.

Direct Instruction Prompt

Replace ReLU with Leaky ReLU (alpha = 0.1) in the first dense layer. Change the second layer’s activation to tanh.

Summary-Mediated Prompt

A sequential encoder with a dense layer of 128 units using Leaky ReLU activation (alpha 0.1), followed by another dense layer with sigmoid/tanh activation.

```
self.encoder = keras.Sequential([
    Dense(128, activation="relu"),
    Dense(encoding_dim, activation="sigmoid"),
    + Dense(128, activation=LeakyReLU(alpha=0.1)),
    + Dense(encoding_dim, activation="tanh"),
])
```

Fig. 5. Example prompts for fulfilling the first requirement in Task 1.

2) *Specifying Changes: Balancing Control, Effort, and Expressiveness:* Once developers identified where to modify,

they adopted two distinct strategies to specify the desired changes: direct instruction prompting (63.2% usage), which offers ease and flexibility, and summary-mediated prompting (36.8% usage), which provides scaffolding and finer control over the changes (Fig. 3). Across all subtasks, 42.2% used direct-only, 31.9% summary-only, and 25.9% mixed prompting, with respective success rates of 84.2%, 93.0%, and 80.0%.

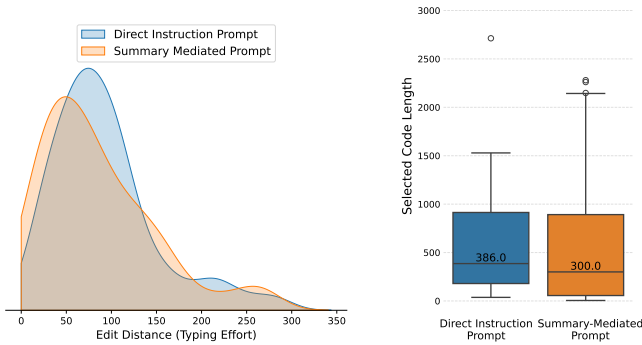
Direct instruction prompting provided strong flexibility and ease of specification, allowing developers to quickly express desired changes (Q3). Many participants found direct instruction prompting “intuitive and quicker” (P8), especially for straightforward edits. As P1 noted, “I can directly write my command [...] and I’m more confident the modifications will align with my intentions, since it’s all my own input.” This flexibility allowed developers to specify changes rapidly without relying on generated summaries (P1, P2, P7, P12, P14). Participants rated direct instruction prompting higher in “ease of specification” (Fig. 4 Q3, $6.0 > 5.0$, $p = 0.0258 < 0.05$).

However, this came with trade-offs, as developers acknowledged that direct instructions were more “prone to errors” (P1) and could result in unintended modifications (P1, P8, P9). As P9 cautioned, “if the model misunderstands my intent, it might touch unrelated parts of the code.”

Summary-mediated prompting provided accurate technical vocabulary, helping developers express changes more precisely. Using correct terminology can be challenging, especially for novice developers [37], [38]. LLM-generated summaries “use more accurate terminology, and then I can reuse those words” (P6), potentially reducing the effort of formulating instructions. As shown in Fig. 5, terms like activation and dense layer reflect domain knowledge that may not be easily recalled, especially by those less familiar with specific APIs or syntax (P6, P9, P13).

Editing summaries reduced typing effort by letting developers tweak existing text rather than writing prompts from scratch. Participants found this approach more natural and less effortful (P2, P3, P4, P6, P10, P12, P13). P3 shared, “I prefer to [...] tweak the sentence directly, rather than inventing a full prompt from scratch.”, as illustrated in Fig. 5. Fig. 6(a) shows that summary-mediated prompts

required fewer keystrokes on average, measured by Levenshtein distance¹⁰ [70] ($82.2 < 89.7$), though not significantly ($p = 0.2248$). However, this benefit depended on the usability of the generated summaries (see Section IV-C).



(a) Distribution of Levenshtein distances (top 1% outliers excluded). (b) Box plot of selected code lengths (top 1% excluded).

Fig. 6. Comparison of interaction metrics. We did not observe a statistically significant difference.

Summaries served as scaffolds for composing comprehensive prompts, helping developers capture and express intended changes (Q2) (P3, P4, P8, P10, P11, P13). Participants rated summary-mediated prompting higher for supporting intent scaffolding (Fig. 4 Q2, $5.0 > 3.0$, $p = 0.015$). P8 noted that, compared to direct instruction, editing the summary helped “*preserve the semantic integrity of these encapsulated [function] units.*” P11 also shared, “*Using the summary as a mental checklist seemed like a good idea [...] to cover more complex modifications.*” Even when the original sentence was hard to modify directly, some participants preferred to leave it unchanged and add new sentences to extend its meaning (P3, P4, P6, P9, P10). This echoes prior findings that users build on existing utterances to guide system behavior [13].

Summary-mediated prompting helped developers control unchanged parts of the code. While Q5 in Fig. 4 showed no significant difference in perceived control between prompting techniques ($6.0 > 5.0$, $p = 0.893$), interviews revealed differing interpretations of “control”: some focused on flexibility and speed, others on stability and scope. As illustrated in Fig. 5, summary-mediated prompting was helpful for constraining edits to intended areas (green/red text) and avoiding unintended changes (gray text). As P4 explained, “*If I change something small while keeping the rest [of the summary] the same, it helps constrain the LLM from modifying unintended areas.*” P9 added that summaries “*retain an overview of the full code, including untouched parts, which gives me a stronger sense of safety.*” This scoped control was especially valuable in complex tasks where developers needed to preserve existing semantics while making targeted changes.

¹⁰Levenshtein distance measures the number of character-level insertions, deletions, or substitutions needed to transform the original summary (or an empty string for direct prompting) into the final prompt.

Editing summaries forced deeper comprehension of both existing code and intended modifications. Participants noted that reading and editing summaries required effort but led to stronger comprehension before making modifications (P2, P3, P9, P12). P3 admitted, “*Reading the summary takes time, [...] but I prefer to understand every line to modify with confidence.*” This enforced comprehension clarified their intentions and improved control. As P2 reflected, “*I feel a much better sense of control because I have to fully comprehend to modify the code.*” While summary-mediated prompting demanded more reading, it encouraged deliberate and accurate edits.

3) Developers Balanced Quick Execution and Semantic Understanding When Validating LLM Modification Output: Developers used two primary strategies: executing the code for immediate feedback or examining its semantics to avoid hidden issues, consistent with prior observations [71].

Many prioritized quick execution to assess output correctness (P1, P4, P7, P9, P10, P11, P12, P13, P14, P15). P12 said, “*After I get the code, I just accept it and run it, then do my own debugging session.*” This strategy was common when time was limited and participants trusted the LLM (P7).

However, developers recognized that runtime checks alone could miss hidden errors [72], leading them to prioritize semantic understanding. P14 warned, “*GPT adds or omits elements that you actually need, and these might not be immediately noticeable in the execution.*” P3 believed that early comprehension reduces future debugging costs, noting, “*If I don’t understand the modified code and just accept it, the risk increases as the project grows.*” For long-term reliability, participants stressed the need to understand the modified code. As P4 explained, summary-mediated prompting process forced them to “*understand what the code is actually doing.*”

These strategies were not chosen arbitrarily; developers’ underlying goals, such as urgency, maintainability, or learning, shaped how they validated LLM outputs and, in turn, which prompting techniques they used. See Section IV-B1 for details.

Key findings: Developers followed an iterative workflow: with the help of summaries, they first understood the existing code and progressively localized the modification point, then validated outputs through execution or semantic reasoning. When specifying changes, they balanced two prompting strategies: direct instruction prompting, valued for its flexibility and ease of specification, and summary-mediated prompting, which supported accurate terminology, contextual completeness, and control over unchanged code. Though summary-mediated prompts required more reading, they reduced typing effort and encouraged deeper comprehension.

B. Factors Shaping Prompting Strategy Choices

Developers’ prompting choices were shaped by multiple factors, which we categorize into two groups: (1) **intentional factors**, reflecting developers’ underlying goals and motivations, and (2) **situational factors**, arising from the specific characteristics of the task and environment. We additionally

draw on insights from professional developers to examine how these factors manifest in **industrial contexts**.

1) *Intentional Factors*: Intentional factors reflect developers' personal goals, shaping not only their prompting choices but also how they validated LLM-generated modifications.

Task Urgency. Urgent tasks led developers to prefer direct instruction prompting for its speed. Under time pressure or in one-off scenarios, developers prioritized fast completion over deep code comprehension, a pattern consistent with prior work on decision-making under time pressure [73]. As P1 put it, *"I just wanted to ensure the code works."* P8 similarly noted that in urgent situations with familiar code, they would *"skip the comprehension and focus on whether the execution is correct."* Direct instruction prompting enabled quick specification and execution-based validation, avoiding the overhead of reading or editing summaries.

Codebase Maintenance. Developers preferred summary-mediated prompting to ensure code quality in support of long-term maintenance interests. When working on personal or maintainable projects, developers prioritized understanding and control over changes. P13 emphasized, *"If it's my own project that requires long-term management, then a complete understanding is necessary."* Summary-mediated prompting enforced deeper comprehension, enabling safer changes aligned with long-term maintenance interests, an essential concern for code quality and modifiability [74], [75].

Learning Goals. When aiming to learn, developers used summary-mediated prompting to build understanding. Developers often treat unfamiliar tasks as learning opportunities and rely on summaries to explore code logic. P3 shared that summaries helped them *"learn something new in just a few seconds,"* while P7 noted that in real projects, they *"would take the time to learn and analyze the generated code more carefully."* Summary-mediated prompting supported learning by encouraging deeper understanding, aligning with concerns that effective GenAI use requires active cognitive effort to prevent shallow comprehension and reasoning offloading [76].

Bug Avoidance. To avoid hidden bugs, developers favored summary-mediated prompting for early semantic validation. Participants aiming to avoid subtle errors were more cautious. As P14 warned, *"A small trick added by GPT can lead to a bug that is only discovered later."* P3 further explained, *"Debugging later may cost more than checking summaries early on."* P7 echoed this concern, *"When new errors appear, I have to go back, scrutinize my code, and debug it again, which can sometimes take longer."* By fostering early comprehension of code changes, summary-mediated prompting helped developers catch issues before they propagated.

2) *Situational Factors*: Beyond personal goals, task-related factors also shaped developers' prompting choices, influencing how they balanced efficiency and comprehension.

Code Familiarity. Familiarity with the codebase encouraged developers to favor direct instruction prompting. When familiar with the code, developers often skipped summaries and used direct instructions to specify changes efficiently because *"I knew exactly which part of the code-*

base I wanted to modify" (P5). In contrast, unfamiliar code prompted greater use of summary-mediated prompting to support understanding before edits (P5, P6, P8, P9, P10, P11, P12, P13), consistent with findings that comprehension strategies depend on prior experience and familiarity [77], [78]. Spearman correlation analysis supports this trend, though correlations were modest. Developers with higher self-reported technical familiarity retrieved fewer summaries ($\rho = -0.15$), made fewer LLM-assisted modifications ($\rho = -0.12$), selected slightly larger code spans ($\rho = 0.06$), and used direct instruction prompts more frequently ($\rho = 0.05$). They also reported lower cognitive load, with mental demand negatively correlated ($\rho = -0.26$, $p = 0.084 < 0.10$). Other NASA-TLX dimensions (e.g., physical, temporal) and perceived performance followed similar trends. These patterns suggest that code familiarity reduces both reliance on external scaffolding and the perceived effort of LLM-assisted code modification.

Task Difficulty. Complex tasks lead developers toward using summary-mediated prompting to build initial comprehension. For simple, well-understood tasks, developers wrote direct instruction prompts *"because I trust the LLM can actually solve it"* (P1, P2, P8). However, as task complexity increased, developers more often started with summaries to understand code structure and clarify requirements (P1, P2, P5, P8, P9, P12, P13, P14), reflecting the need for deeper comprehension under higher cognitive demand [77].

Edit Scope. Smaller edit scopes favored summary-mediated prompting for control; larger scopes favored direct instruction for flexibility. For small edits, developers valued the control summaries provided (P2). In contrast, broader changes led them to prefer direct instruction to avoid the overhead of reading or editing extensive summaries (P1, P2). This trend appears in Fig. 6(b), where direct instruction prompts were linked to slightly longer code spans ($386.0 > 300.0$), though the difference was not statistically significant.

Intention Clarity. Clear goals led developers to prefer direct instruction prompting. When developers knew what they wanted to change, they expressed it directly and efficiently. As P11 noted, *"If I have a clear goal, I would go direct because I can just describe it, highlight the part, and tell it what to do."* In contrast, when intentions were vague, developers used summaries to help refine and clarify their objectives.

3) *Industrial Context*: **The complexity and collaborative nature of industrial codebases increased developers' need for comprehensive understanding.** Industrial projects span large, interdependent modules where developers focus on specific features while relying on shared infrastructure and APIs [78]. Developers *"first need to understand how others' components work,"* (P8) and *"ensure that my AI modifications don't break existing functionality"* (P9). They frequently worked with unfamiliar code written by teammates, requiring extra effort to understand dependencies and integration points (P10). These complexities led developers to favor summaries for building broader system understanding (P8, P9, P11, P13), while also highlighting the importance of summary quality.

The absence of LLM knowledge about internal code

and high responsibility drove developers to write more detailed prompts and emphasize careful validation. LLMs lacked familiarity with domain-specific code and internal infrastructure, making effective prompting more challenging. As P13 noted, “*Unlike public packages, LLMs don’t have this knowledge.*” Developers compensated by crafting more detailed prompts and manually verifying outputs to ensure correctness. Accountability pressures reinforced this caution, “*The problems that arise in the industry will ultimately be traced back to the owner of the pull request*” (P13).

Key findings: Developers’ prompting choices were shaped by intentional factors. Task urgency favored direct instruction prompting for its speed and flexibility, while goals such as maintainability, learning, and bug avoidance led developers to prefer summary-mediated prompting to support deeper understanding and control. Situational factors further shaped strategy selection: developers preferred direct prompting when tasks were familiar, simple, broad in scope, or clearly defined, and turned to summaries when facing unfamiliar, complex, or ambiguous scenarios. Finally, working with industrial codebases amplified the need for understanding and validation due to large interdependent systems, collaborative workflows, and LLMs’ limited knowledge of internal infrastructure.

C. Usability Considerations and Design Opportunities

In this section, we examine developers’ experiences with the usability of summary-mediated prompting and outline opportunities for design improvement.

1) *Granularity and Goal Alignment Shape the Usability of Summaries:* **Usability depends on summary granularity and alignment with developers’ modification goals** (P4, P5, P6, P15). Participants noted that overly high-level summaries made it difficult to locate precise edit points. As P1 remarked, “*It’s sometimes challenging to determine where to insert or modify my changes [into the summary].*” P14 similarly wanted summaries “*divided into smaller parts, detailing what specific lines do.*” Conversely, overly fine-grained summaries risked fragmented edits. P6 warned, “*If A appears multiple times, and you only edit part of it, leftover fragments may remain,*” highlighting the need to balance abstraction and detail to support targeted edits without introducing inconsistencies.

When summaries exactly covered the relevant code and aligned well with developers’ goals, participants could make localized edits by simply modifying existing sentences (see Fig. 5). Otherwise, they had to append entire new sentences (P3) or fall back to direct prompting for efficiency (P12).

2) *Summary-Mediated Prompting Needs Improved Structure, Granularity, Traceability, and Consistency:* Developers suggested ways to improve the usability of summary-mediated prompting and better align it with their workflows.

Structured formats could better reflect code organization and improve readability. Participants preferred structured outlines or bullet points over free-form text to improve readability and reduce cognitive effort (P6, P8, P9, P10, P11).

As P8 noted, “*When the summary is formatted like 1, 2, 3, it becomes easier to read and understand.*” P9 added that summaries structured by module or function would make writing prompts “*feel like filling in the blanks.*”

Summaries at different granularities were seen as essential for balancing efficiency and thoroughness. Participants suggested summaries with adjustable levels of detail, allowing them to start simple and progressively expand as needed (P5, P6, P14). P5 envisioned beginning with a concise version and “*having a way to expand the summary to include more details.*” This flexibility would help developers dynamically balance brevity and depth across tasks. It also aligns with *abstraction gradient* in *Cognitive Dimensions of Notations* [79], as developers benefit from engaging with representations at varying levels of abstraction depending on their goals.

Developers emphasized improving traceability between summaries and code to support comprehension. Participants proposed visualizing mappings between summary sentences and corresponding code segments to accelerate understanding (P1, P7). P1 suggested, “*If I hover over a sentence in the summary, it could show me which part of the code it corresponds to.*” P13 further recommended highlighting summary portions that require modification to guide developers’ attention. These suggestions exemplify *closeness of mapping* and *visibility* dimensions [79]: making summary–code links explicit helps developers locate relevant context and reduce the mental effort of bridging representations.

Consistency in summary generation was critical to support iterative modifications. Participants expressed confusion about variability in LLM summaries across iterations. As P4 noted, “*Each time the summary is generated, the content is different,*” which complicates iterative editing, a common workflow in LLM-assisted programming. Inconsistent outputs disrupted developers’ expectations and made it harder to identify what changed and what remained stable, undermining both *consistency* and the *role expressiveness* of generated summaries [79]. Maintaining consistent summaries that only reflect the modified parts, just like the edited summary in Fig. 5, would help preserve developers’ mental models over time.

Key findings: Developers emphasized that the usability of summary-mediated prompting hinged on both granularity and alignment with modification goals. Summaries that were too high-level obscured edit locations, while overly fine-grained ones introduced inconsistencies. To address these issues, participants suggested concrete improvements: adopting structured formats to reflect code organization, supporting adjustable levels of detail, enabling visual mappings between summaries and code, and maintaining output consistency across iterations.

V. THREATS TO VALIDITY

Our study faces several validity threats. First, the tasks were lightweight to fit within the time constraints of a lab study [80]. Although they span diverse domains and modification types, they do not fully reflect the complexity, scale, and ambiguity of

real-world programming. This tradeoff between experimental control and ecological realism is well-documented in programmer user studies [81], [82].

Second, to focus on prompting strategies rather than bug discovery, we provided explicit modification requirements. While this improved task consistency, it may have affected natural prompting behavior, as participants responded to given goals instead of identifying changes themselves. To reduce this effect, we made the instructions intentionally indirect and prohibited copy-pasting from the instruction document to encourage more active engagement. We also asked participants to avoid manual edits and use prompting whenever possible. These constraints, while necessary for isolating prompting behaviors, may have led participants to write prompts in cases where they would typically make direct edits [83].

Third, differences in the development environment may have influenced participants' behavior. Some were more familiar with macOS and VSCode, while the study used JetBrains IDE on a Windows machine, potentially increasing cognitive load. Running the study via Zoom also introduced a slight screen delay, which may have affected navigation and typing. These factors likely did not affect prompting strategies but may have introduced minor inefficiencies during task execution.

Finally, while our prototype was designed to resemble real-world LLM-assisted coding tools (Section III-C3), its interface choices (e.g., selection-based prompting) may not reflect the full range of interactions found in AI-powered IDEs. In particular, summary-mediated prompting depends on the capabilities of the underlying LLM, which may shape how developers perceive and engage with generated summaries. Additionally, we relied on self-reported data (e.g., interviews and usability ratings), which are inherently subjective. To mitigate this, we triangulated findings with interaction logs and observations.

VI. DISCUSSION

A. Cognitive Burden of Prompting for Code Modification

Prompting for code modification is cognitively demanding, as developers must not only understand the existing code but also communicate their intended changes with clarity and precision. While code summaries offer partial support by facilitating comprehension and providing a structured scaffold, many challenges persist. Developer intentions are often vague or evolving (P7, P10, P13), and relevant context, such as call graphs or dependency structures, is difficult to express precisely in natural language (P9, P15). Although recent vibe coding tools aim to reduce prompting effort by retrieving related code and diagnostic signals, such as linter warnings and console outputs, participants in our study identified a key limitation: LLMs lack visibility into runtime program behavior. Developers found it difficult to describe issues such as unintended UI behavior (P8) or tensor shape mismatches (P14), which only emerge during execution and are hard to convey through prompts. These limitations reflect a persistent gap between what developers observe and what LLMs can interpret, highlighting the need for interaction methods that more effectively capture runtime feedback and evolving intent.

B. Comprehension Challenges in Vibe Coding Workflows

After committing prompts, developers must comprehend the resulting edits. This is particularly challenging in vibe coding, where LLMs generate or modify large amounts of code in one step, often across multiple files, leaving developers overwhelmed and unsure of what changed. Tools like Bolt¹¹ exemplify this trend, aiming to support low- or no-code development through conversational prompts and runtime previews. While many developers appreciated the efficiency, our findings indicate that comprehension remains essential, particularly for those prioritizing maintainability, learning, or bug avoidance. Several participants preferred incremental updates for greater control (P10) and expressed low trust in large or cross-file modifications (P3, P5, P7, P8, P13), highlighting the difficulty of maintaining trust and oversight in current vibe coding tools. Basic code diffs and chatbot-style explanations may be insufficient for understanding and validating large-scale LLM edits, particularly for end users with limited technical background. Future studies should examine this beyond small-snippet completions (e.g., Copilot [2]) and across different user groups. Our findings suggest that structured summaries and code-summary mappings can scaffold comprehension and align edits with developer goals. Future work should also explore integrating traditional techniques, such as automated testing, to enhance trust in multi-step modifications.

C. Natural Language Programming in the LLM Era

Natural language programming aims to let people express ideas “in the same way they think about them” [84], offering a more intuitive bridge between mental intent and computational logic. While early advocates saw it as a path to democratizing programming [85], critics argued that natural language, with its inherent ambiguity, lacks the precision required for instructing machines [86]. Modern LLMs represent a shift: rather than replacing programming languages, they can interpret vague or underspecified intent, positioning natural language as an interactive layer between human reasoning and formal code. Our study shows that treating summaries as a medium for intent scaffolding reveals several key usability factors. Further research should explore the broader roles and design spaces of natural language across diverse programming contexts.

VII. CONCLUSION

This study examined how developers construct prompts for LLM-assisted code modification, comparing direct instruction and summary-mediated prompting. Through a mixed-methods study with 15 developers across diverse programming tasks, we characterized their prompting workflows and decision-making patterns. Building on these findings, future work should explore interactive systems that better scaffold comprehension and control through code summaries designed for structural clarity, improved traceability, and adjustable granularity. In addition, longitudinal deployment studies are needed to examine how prompting strategies evolve over time and integrate into real-world development workflows at scale.

¹¹<https://bolt.new/>

To support transparency and reproducibility, we provide a replication package with the codebook, coded interview segments, interaction logs, questionnaire responses, analysis scripts, study protocol, task descriptions, and source code of PASTA, available at: <https://github.com/ND-SaNDwichLAB/direct-vs-summary-study>.

ACKNOWLEDGMENT

This research was supported in part by an AnalytiXIN Faculty Fellowship, an NVIDIA Academic Hardware Grant, a Google Cloud Research Credit Award, a Google Research Scholar Award, and NSF grants CCF-2211428, CCF-2315887, and CCF-2100035. Any opinions, findings, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

- [1] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.
- [2] N. Tang, M. Chen, Z. Ning, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "Developer behaviors in validating and repairing llm-generated code using ide and eye tracking," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 40–46.
- [3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [4] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.
- [5] D. J. Gilmore, "Models of debugging," *Acta psychologica*, vol. 78, no. 1-3, pp. 151–172, 1991.
- [6] M. Lehman and L. Belady, "Software evolution-processes of software change," 1985.
- [7] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep.*, vol. 229, p. 2013, 2013.
- [8] L. Zheng, W.-L. Chiang, Y. Sheng, T. Li, S. Zhuang, Z. Wu, Y. Zhuang, Z. Li, Z. Lin, E. P. Xing *et al.*, "Lmsys-chat-1m: A large-scale real-world llm conversation dataset," *arXiv preprint arXiv:2309.11998*, 2023.
- [9] F. Cassano, L. Li, A. Sethi, N. Shinn, A. Brennan-Jones, J. Ginesin, E. Berman, G. Chakhnashvili, A. Lozhkov, C. J. Anderson *et al.*, "Can it edit? evaluating the ability of large language models to follow code editing instructions," *arXiv preprint arXiv:2312.12450*, 2023.
- [10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [11] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.
- [12] L. Tankelevitch, V. Kewenig, A. Simkute, A. E. Scott, A. Sarkar, A. Sellen, and S. Rintel, "The metacognitive demands and opportunities of generative ai," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–24.
- [13] M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon, "“what it wants me to say”: Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–31.
- [14] C. Chen, S. Feng, A. Sharma, and C. Tan, "Machine explanations and human understanding," *arXiv preprint arXiv:2202.04092*, 2022.
- [15] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.
- [16] J. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–21.
- [17] J. T. Liang, M. Lin, N. Rao, and B. A. Myers, "Prompts are programs too! understanding how developers build software containing prompts," *arXiv preprint arXiv:2409.12447*, 2024.
- [18] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.
- [19] Y. Tian, Z. Zhang, Z. Ning, T. Li, J. K. Kummerfeld, and T. Zhang, "Interactive text-to-sql generation via editable step-by-step explanations," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 16 149–16 166.
- [20] Y. Tian, J. K. Kummerfeld, T. J.-J. Li, and T. Zhang, "Sqlucid: Grounding natural language database queries with interactive explanations," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–20.
- [21] K. Ferdowsi, J. Williams, I. Drosos, A. D. Gordon, C. Negreanu, N. Polikarpova, A. Sarkar, and B. Zorn, "Coldeco: An end user spreadsheet inspection tool for ai-generated code," in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2023, pp. 82–91.
- [22] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," *arXiv preprint arXiv:2305.04087*, 2023.
- [23] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.
- [24] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.
- [25] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.
- [26] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," *arXiv preprint arXiv:2411.02320*, 2024.
- [27] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 819–831.
- [28] K. Li, Q. Hu, X. Zhao, H. Chen, Y. Xie, T. Liu, Q. Xie, and J. He, "Instructcoder: Instruction tuning large language models for code editing," *arXiv preprint arXiv:2310.20329*, 2023.
- [29] K. Shi, D. Altunbükten, S. Anand, M. Christodorescu, K. Grünwedel, A. Koenings, S. Naidu, A. Pathak, M. Rasi, F. Ribeiro *et al.*, "Natural language outlines for code: Literate programming in the llm era," *arXiv preprint arXiv:2408.04820*, 2024.
- [30] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.
- [31] N. Tang, J. An, M. Chen, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "Codegrits: A research toolkit for developer behavior and eye tracking in ide," in *Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: Companion proceedings*, 2024, pp. 119–123.
- [32] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.
- [33] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, 2023.
- [34] H. Mozannar, G. Bansal, A. Fournay, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming,"

- in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.
- [35] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, “Productivity assessment of neural code completion,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.
 - [36] M. Q. Feldman and C. J. Anderson, “Non-expert programmers in the generative ai future,” in *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*, 2024, pp. 1–19.
 - [37] S. Nguyen, H. M. Babe, Y. Zi, A. Guha, C. J. Anderson, and M. Q. Feldman, “How beginning programmers and code llms (mis) read each other,” in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–26.
 - [38] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, “Github copilot ai pair programmer: Asset or liability?” *Journal of Systems and Software*, vol. 203, p. 111734, 2023.
 - [39] A. Von Mayrhauser and A. M. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, 1995.
 - [40] M. Balz, M. Striewe, and M. Goedicke, “Continuous maintenance of multiple abstraction levels in program code,” in *International Workshop on Future Trends of Model-Driven Development*, vol. 2. SCITEPRESS, 2010, pp. 68–79.
 - [41] E. L. Hutchins, J. D. Hollan, and D. A. Norman, “Direct manipulation interfaces,” *Human-computer interaction*, vol. 1, no. 4, pp. 311–338, 1985.
 - [42] R. Yen, J. S. Zhu, S. Suh, H. Xia, and J. Zhao, “Coladder: Manipulating code generation via multi-level blocks,” in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–20.
 - [43] M. Kazemitabaar, J. Williams, I. Drosos, T. Grossman, A. Z. Henley, C. Negreanu, and A. Sarkar, “Improving steering and verification in ai-assisted data analysis with interactive task decomposition,” in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–19.
 - [44] L. Xie, C. Zheng, H. Xia, H. Qu, and C. Zhu-Tian, “Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization,” in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–14.
 - [45] N. Tang, “Towards effective validation and integration of llm-generated code,” in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 369–370.
 - [46] D. Masson, S. Malacria, G. Casiez, and D. Vogel, “Directgpt: A direct manipulation interface to interact with large language models,” in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.
 - [47] Y. Di and T. Zhang, “Enhancing code generation via bidirectional comment-level mutual grounding,” *arXiv preprint arXiv:2505.07768*, 2025.
 - [48] E. Jiang, E. Toh, A. Molina, K. Olson, C. Kayacik, A. Donsbach, C. J. Cai, and M. Terry, “Discovering the syntax and strategies of natural language programming with generative language models,” in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–19.
 - [49] A. M. McNutt, C. Wang, R. A. Deline, and S. M. Drucker, “On the design of ai-powered code assistants for notebooks,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–16.
 - [50] H. Dang, L. Mecke, F. Lehmann, S. Goller, and D. Buschek, “How to prompt? opportunities and challenges of zero-and few-shot learning for human-ai interaction in creative applications of generative models,” *arXiv preprint arXiv:2209.01390*, 2022.
 - [51] H. Dang, S. Goller, F. Lehmann, and D. Buschek, “Choice over control: How users write with large language models using diegetic and non-diegetic prompting,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–17.
 - [52] F. F. Xu, B. Vasilescu, and G. Neubig, “In-ide code generation from natural language: Promise and challenges,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–47, 2022.
 - [53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
 - [54] S. H. Bach, V. Sanh, Z.-X. Yong, A. Webson, C. Raffel, N. V. Nayak, A. Sharma, T. Kim, M. S. Bari, T. Fevry *et al.*, “Promptsources: An integrated development environment and repository for natural language prompts,” *arXiv preprint arXiv:2202.01279*, 2022.
 - [55] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, “Prompt engineering in large language models,” in *International conference on data intelligence and cognitive informatics*. Springer, 2023, pp. 387–402.
 - [56] P. Denny, V. Kumar, and N. Giacaman, “Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language,” in *Proceedings of the 54th ACM technical symposium on computer science education V. 1*, 2023, pp. 1136–1142.
 - [57] T. Wang, N. Zhou, and Z. Chen, “Enhancing computer programming education with llms: A study on effective prompt engineering for python code generation,” *arXiv preprint arXiv:2407.05437*, 2024.
 - [58] R. Yen, J. Zhao, and D. Vogel, “Code shaping: Iterative code editing with free-form sketching,” in *Adjunct Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–3.
 - [59] R. Rawal, V.-A. Pădurean, S. Apel, A. Singla, and M. Toneva, “Hints help finding and fixing bugs differently in python and text-based program representations,” *arXiv preprint arXiv:2412.12471*, 2024.
 - [60] I. Etikan, S. A. Musa, R. S. Alkassim *et al.*, “Comparison of convenience sampling and purposive sampling,” *American journal of theoretical and applied statistics*, vol. 5, no. 1, pp. 1–4, 2016.
 - [61] C.-Y. Su, A. Bansal, and C. McMillan, “Revisiting file context for source code summarization,” *Automated Software Engineering*, vol. 31, no. 2, p. 62, 2024.
 - [62] S. Haque, A. LeClair, L. Wu, and C. McMillan, “Improved automatic summarization of subroutines via attention to file context,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 300–310.
 - [63] A. Bansal, Z. Eberhart, Z. Karas, Y. Huang, and C. McMillan, “Function call graph context encoding for neural source code summarization,” *IEEE Transactions on Software Engineering*, vol. 49, no. 9, pp. 4268–4281, 2023.
 - [64] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1994.
 - [65] S. G. Hart, “Nasa task load index (tlx),” 1986.
 - [66] M. Brod, L. E. Tesler, and T. L. Christensen, “Qualitative research and content validity: developing best practices based on science and experience,” *Quality of life research*, vol. 18, pp. 1263–1278, 2009.
 - [67] J. Lazar, J. H. Feng, and H. Hochheiser, *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.
 - [68] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *biometrics*, pp. 159–174, 1977.
 - [69] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, “How programmers debug, revisited: An information foraging theory perspective,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2010.
 - [70] V. I. Levenshtein *et al.*, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.
 - [71] G. O’Brien, “How scientists use large language models to program,” *arXiv preprint arXiv:2502.17348*, 2025.
 - [72] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
 - [73] M. R. Endsley, “Toward a theory of situation awareness in dynamic systems,” *Human factors*, vol. 37, no. 1, pp. 32–64, 1995.
 - [74] E. B. Swanson, “The dimensions of maintenance,” in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 492–497.
 - [75] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
 - [76] A. Singh, K. Taneja, Z. Guan, and A. Ghosh, “Protecting human cognition in the age of ai,” *arXiv preprint arXiv:2502.12447*, 2025.
 - [77] S. Letovsky, “Cognitive processes in program comprehension,” *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.
 - [78] A. von Mayrhauser and A. M. Vans, “Industrial experience with an integrated code comprehension model,” *Software Engineering Journal*, vol. 10, no. 5, pp. 171–182, 1995.
 - [79] T. R. Green, “Cognitive dimensions of notations,” *People and computers V*, pp. 443–460, 1989.

- [80] A. J. Ko, T. D. LaToza, and M. M. Burnett, “A practical guide to controlled experiments of software engineering tools with human participants,” *Empirical Software Engineering*, vol. 20, pp. 110–141, 2015.
- [81] J. Siegmund, N. Siegmund, and S. Apel, “Views on internal and external validity in empirical software engineering,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 9–19.
- [82] D. I. Sjöberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, “Conducting realistic experiments in software engineering,” in *Proceedings international symposium on empirical software engineering*. IEEE, 2002, pp. 17–26.
- [83] M. C. Davis, E. Aghayi, T. D. Latoza, X. Wang, B. A. Myers, and J. Sunshine, “What’s (not) working in programmer user studies?” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–32, 2023.
- [84] B. A. Myers, J. F. Pane, and A. J. Ko, “Natural programming languages and environments,” *Communications of the ACM*, vol. 47, no. 9, pp. 47–52, 2004.
- [85] J. E. Sammet, “The use of english as a programming language,” *Communications of the ACM*, vol. 9, no. 3, pp. 228–230, 1966.
- [86] E. W. Dijkstra, “On the foolishness of “natural language programming”,” *Program Construction: International Summer School*, pp. 51–53, 2005.

TABLE I
SUMMARY OF PARTICIPANT DEMOGRAPHICS AND TECHNICAL BACKGROUND.

ID	Gender	Age	Role	Experience	Deep Learning	Data Vis	Web Dev	TensorFlow	D3.js	Chrome Ext
P1	Male	27	Graduate Student	8 years	Yes	Yes	Yes	Proficient	Proficient	Proficient
P2	Male	22	Graduate Student	5 years	Yes	Yes	Yes	Unfamiliar	Never	Never
P3	Male	24	Graduate Student	6 years	Yes	Yes	Yes	Never	Never	Never
P4	Male	22	Graduate Student	5 years	Yes	Yes	Yes	Unfamiliar	Unfamiliar	Unfamiliar
P5	Female	27	Graduate Student	10 years	Yes	Yes	Yes	Unfamiliar	Unfamiliar	Never
P6	Female	23	Graduate Student	6 years	Yes	Yes	Yes	Unfamiliar	Never	Proficient
P7	Male	27	Graduate Student	7 years	Yes	Yes	Yes	Never	Never	Unfamiliar
P8	Male	25	Professional Developer	6 years	No	Yes	Yes	Never	Never	Proficient
P9	Male	24	Professional Developer	7 years	No	Yes	Yes	Unfamiliar	Never	Never
P10	Male	22	Professional Developer	6 years	Yes	Yes	Yes	Proficient	Unfamiliar	Never
P11	Male	28	Professional Developer	8 years	Yes	Yes	No	Proficient	Never	Unfamiliar
P12	Female	27	Graduate Student	5 years	Yes	Yes	Yes	Unfamiliar	Never	Unfamiliar
P13	Male	30	Professional Developer	12 years	Yes	Yes	Yes	Unfamiliar	Unfamiliar	Unfamiliar
P14	Female	22	Graduate Student	5 years	Yes	No	Yes	Unfamiliar	Never	Unfamiliar
P15	Female	27	Graduate Student	8 years	Yes	Yes	Yes	Unfamiliar	Unfamiliar	Unfamiliar

* Data Vis = Data Visualization; Web Dev = Web Development; Chrome Ext = Chrome Extension Development.

† Deep Learning, Data Vis, and Web Dev indicate whether the participant had prior experience in each domain.

‡ TensorFlow, D3.js, and Chrome Ext indicate their self-rated proficiency levels.