Using an LLM to Help With Code Understanding

Daye Nam
Carnegie Mellon University
U.S.A.
dayen@cs.cmu.edu

Andrew Macvean
Google, Inc.
U.S.A.
amacvean@google.com

Vincent Hellendoorn
Carnegie Mellon University
U.S.A.
vhellend@andrew.cmu.edu

Bogdan Vasilescu
Carnegie Mellon University
U.S.A.
vasilescu@cmu.edu

Brad Myers
Carnegie Mellon University
U.S.A.
bam@cs.cmu.edu

ABSTRACT

Understanding code is challenging, especially when working in new and complex development environments. Code comments and documentation can help, but are typically scarce or hard to navigate. Large language models (LLMs) are revolutionizing the process of writing code. Can they do the same for helping understand it? In this study, we provide a first investigation of an LLM-based conversational UI built directly in the IDE that is geared towards code understanding. Our IDE plugin queries OpenAI's GPT-3.5 and GPT-4 models with four high-level requests *without* the user having to write explicit prompts: to explain a highlighted section of code, provide details of API calls used in the code, explain key domain-specific terms, and provide usage examples for an API. The plugin also allows for open-ended prompts, which are automatically contextualized to the LLM with the program being edited. We evaluate this system in a user study with 32 participants, which confirms that using our plugin can aid task completion more than web search. We additionally provide a thorough analysis of the ways developers use, and perceive the usefulness of, our system, among others finding that the usage and benefits differ significantly between students and professionals. We conclude that in-IDE prompt-less interaction with LLMs is a promising future direction for tool builders.

ACM Reference Format:
Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Building and maintaining software systems requires a deep understanding of a codebase. Consequently, developers spend a significant amount of time searching and foraging for the information they need and organizing and digesting the information they find [30, 31, 34, 42, 46, 50]. Understanding code, however, is a challenging task; developers need to assimilate a large amount of information about the semantics of the code, the intricacies of the APIs used, and the relevant domain-specific concepts. Such information is often scattered across multiple sources, making it challenging for developers, especially novices or those working with unfamiliar APIs, to locate what they need. Furthermore, much of the relevant information is inadequately documented or spread across different formats and mediums, where it often becomes outdated.

With the growing popularity of large language model (LLM) based code generation tools [2, 4, 7], the need for information support for code understanding is arguably growing even higher. These tools can generate code automatically, even for developers with limited coding skills or domain knowledge. This convenience comes at a cost, however – developers may receive code they don't understand [27, 70]. Indeed, early research on LLM code generation tools has found that developers have a harder time debugging code generated by the LLM and easily get frustrated [39, 62].

Fortunately, LLMs also provide an opportunity in this space, namely by offering on-demand *generation-based information support* for developers faced with unfamiliar code. Compared to general web search queries [65], LLM prompts can allow developers to provide more context, which can enable them to receive information that more precisely aligns with their specific needs, potentially reducing the time spent on sifting through the information obtained from the web to suit their particular requirements. Developers have indeed taken to web-hosted conversational LLM tools, such as ChatGPT, for programming support en masse, but this setup requires them to both context switch and copy the relevant context from their IDEs into the chat system for support.

To explore the potential for generation-based information support directly in the developer's programming environment, we developed a prototype in-IDE LLM information support tool, GILT (Generation-based Information-support with LLM Technology). GILT is capable of generating on-demand information while considering the user's local code context, which we incorporate into the prompts provided to the LLM behind the scenes. This way, we also introduce a novel interaction method with the LLM, *prompt-less interaction.* This option aims to alleviate the cognitive load associated with writing prompts, particularly for developers who possess limited domain or programming knowledge.

As there is still little knowledge about how to best use an LLM for information support (as opposed to just code generation), we evaluate the effectiveness of our prototype tool in an exploratory

ICSE 2024, April 2024, Lisbon, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

user study with 32 participants tasked with comprehending and extending unfamiliar code that involves new domain concepts and Python APIs for data visualization and 3D rendering – a challenging task. Our study quantitatively compares task completion rates and measures of code understanding between two conditions – using the LLM-backed assistant in-IDE versus directly searching the web in a browser – and qualitatively investigates how participants used the tools and their overall satisfaction with this new interaction mode. Concretely, we answer three research questions:

- RQ1: To what extent does GILT affect developers' understanding, task completion time, and task completion rates when faced with unfamiliar code?
- RQ2: How do developers interact with GILT, and to what extent does that differ between the participants?
- RQ3: How do developers perceive the usefulness of GILT?

Our results confirm that there are statistically significant gains in task completion rate when using GILT, compared to a web search, showing the utility of generation-based information support. However, we did not find the utility gains in terms of time and understanding level of code, leaving room for further improvement. We also discovered that the degree of the benefit varies between students and professionals, and investigate potential reasons behind this.

2 RELATED WORK

2.1 Studies on Developers Information Seeking

In every phase of modern software engineering, developers need to work with unfamiliar code, and how well they learn such code influences their productivity significantly. Therefore, researchers have studied to understand how developers learn and comprehend unfamiliar code, especially how they search for and acquire information, as developers need a variety of kinds of knowledge [30, 41, 58, 59, 68]. Particularly, lots of research was done on the information seeking strategies of developers, mostly in general software maintenance [22, 30, 35] or web search settings [11, 52]. Researchers have also studied challenges developers face [18, 32, 51, 63, 65], including difficulties in effective search-query writing, information foraging, and applying the information to their own tasks. In this work, we explore a way of supporting developers' information needs with generation-based information support using LLMs.

Other efforts were made to understand developers' information seeking within software documentation, which is the main source of information for developers when they learn to use new APIs or libraries. Researchers cataloged problems developers face when using documentation [13, 53, 54] and identified types of knowledge developers report to be critical [44, 53, 54, 61], which were taken into account when we designed GILT for our study.

2.2 Studies on LLM-based Developer Tools

The potential and applicability of LLM-based AI programming tools have been actively studied by many researchers. Numerous empirical studies [21, 26, 37, 57] evaluated the quality of code or explanations generated by LLMs, to test the feasibility of applying LLM into development tools [40, 60, 70] and to Computer Science education [26, 57]. Several studies have also compared LLM-generated code and explanations with those authored by humans without LLM assistance [26, 37, 49], demonstrating that LLMs can offer reasonably good help for developers or students with caution.

Fewer studies have specifically explored the *usefulness* of LLM-based programming tools [8, 29, 39, 47, 55, 56, 62, 66, 70, 70] with actual users or their usage data, and many of these studies have focused on code generation tools like CoPilot [4]. For instance, Ziegler et al.[70] analyzed telemetry data and survey responses to understand developers' perceived productivity with GitHub Copilot, revealing that over one-fifth of suggestions were accepted by actual developers. Several human studies were also conducted. Vaithilingam *et al.* [62] compared the user experience of GitHub Copilot to traditional autocomplete in a user study and found that participants more frequently failed to complete tasks with Copilot, although there was no significant effect on task completion time. Barke [8] investicated further with a grounded theory analysis to understand *how* programmers interact with code-generating models, using Github Copilot as an example. They identified two primary modes of interaction, acceleration or exploration, where Copilot is used to speed up code authoring in small logical units or as a planning assistant to suggest structure or API calls.

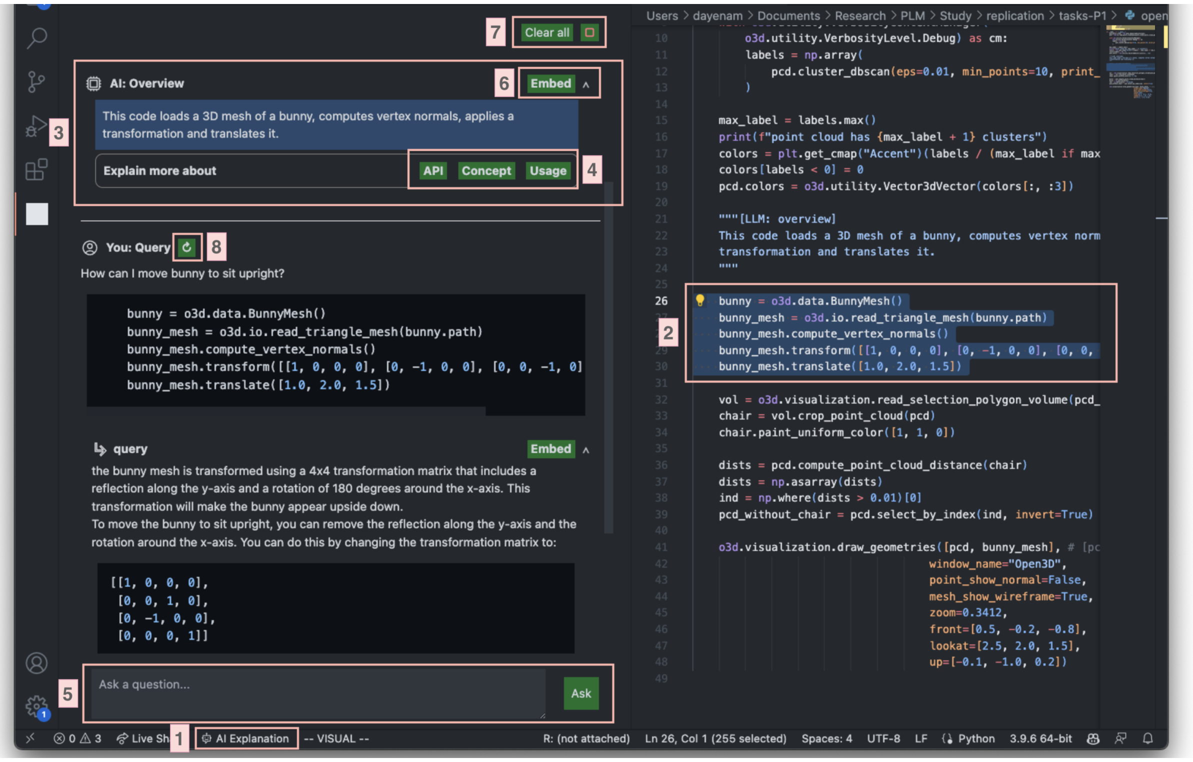
Although these studies have increased our understanding of the usefulness and usability of AI programming assistants in general, and some of the insights apply to information support, they do not show the opportunities and challenges of LLM-based tools as information support tools, with a few following exceptions [43, 55]. MacNeil et al. [43] examined the advantages of integrating code explanations generated by LLMs into an interactive e-book focused on web software development, with a user study with sophomores. They found students tend to find LLM-generated explanations to be useful, which is promising, but the study was focused on providing one-directional support in an introductory e-book which is different from user-oriented need-based information support. The Programmer's assistant [55] is the closest to our work. The authors integrated a conversational programming assistant into an IDE to explore other types of assistance beyond code completion. They collected quantitative and qualitative feedback from a human study with 42 participants from diverse backgrounds and found that the *perceived* utility of the conversational programming assistance was high. In our work, we focus on the utility of LLM-based tools to satisfy information needs for *code understanding*, and take a step forward to test the actual utility of an LLM-integrated programming tool by assessing performance measures such as completion rates, time, and participants' code understanding levels.

3 THE GILT PROTOTYPE TOOL

We iteratively designed GILT to explore different modes of interaction with an LLM for information support. GILT is a plugin for the VS Code IDE (Figure 1) that considers user context (the code selected by the user) when querying a LLM for several information support applications.

3.1 Interacting with GILT

There are two ways to interact with the plugin. First, users can select parts of their code (Figure 1-②) and trigger the tool by clicking on "AI Explanation" on the bottom bar (Figure 1-①), or using



AI: Overview

This code loads a 3D mesh of a bunny, computes vertex normals, applies a transformation and translates it.

Explain more about

You: Query

How can I move bunny to sit upright?

```
bunny = o3d.data.BunnyMesh()
bunny_mesh = o3d.io.read_triangle_mesh(bunny.path)
bunny_mesh.compute_vertex_normals()
bunny_mesh.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0]
bunny_mesh.translate([1.0, 2.0, 1.5])
```

query

the bunny mesh is transformed using a 4x4 transformation matrix that includes a reflection along the y-axis and a rotation of 180 degrees around the x-axis. This transformation will make the bunny appear upside down.
To move the bunny to sit upright, you can remove the reflection along the y-axis and the rotation around the x-axis. You can do this by changing the transformation matrix to:

```
[[1, 0, 0, 0],
 [0, 0, 1, 0],
 [0, -1, 0, 0],
 [0, 0, 0, 1]]
```

Ask a question...

```
        o3d.utility.VerbosityLevel.Debug) as cm:
        labels = np.array(
            pcd.cluster_dbscan(eps=0.01, min_points=10, print_
        )

    max_label = labels.max()
    print(f"point cloud has {max_label + 1} clusters")
    colors = plt.get_cmap("Accent")(labels / (max_label if max
    colors[labels < 0] = 0
    pcd.colors = o3d.utility.Vector3dVector(colors[:, :3])

    """[LLM: overview]
    This code loads a 3D mesh of a bunny, computes vertex norm
    transformation and translates it.
    """

    bunny = o3d.data.BunnyMesh()
    bunny_mesh = o3d.io.read_triangle_mesh(bunny.path)
    bunny_mesh.compute_vertex_normals()
    bunny_mesh.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0,
    bunny_mesh.translate([1.0, 2.0, 1.5])

    vol = o3d.visualization.read_selection_polygon_volume(pcd_
    chair = vol.crop_point_cloud(pcd)
    chair.paint_uniform_color([1, 1, 0])

    dists = pcd.compute_point_cloud_distance(chair)
    dists = np.asarray(dists)
    ind = np.where(dists > 0.01)[0]
    pcd_without_chair = pcd.select_by_index(ind, invert=True)

    o3d.visualization.draw_geometries([pcd, bunny_mesh], # [pc
                                      window_name="Open3D",
                                      point_show_normal=False,
                                      mesh_show_wireframe=True,
                                      zoom=0.3412,
                                      front=[0.5, -0.2, -0.8],
                                      lookat=[2.5, 2.0, 1.5],
                                      up=[-0.1, -1.0, 0.2])
```

Figure 1: Overview of our prototype. (1) A trigger button; (2) code used as context when prompting LLM; (3) code summary (no-prompt trigger); (4) buttons for further details; (5) an input box for user prompts; (6) options to embed information to code (Embed) and a hide/view button; (7) options to clear the panel (Clear all) and an abort LLM button; (8) a refresh button.

"alt/option + a" as a shortcut, to receive a summary description of the highlighted code (Overview). They can then explore further by clicking on buttons (Figure 1-④) for API (API), domain-specific concepts (Concept), and usage examples (Usage), which provide more detailed explanations with preset prompts. The API button offers detailed explanations about the API calls used in the code, the Concept button provides domain-specific concepts that might be needed to understand the highlighted code fully, and the Usage button offers a code example involving API calls used in the highlighted code.

Users can also ask a specific question directly to the LLM via the input box (Figure 1-⑤). If no code is selected, the entire source code is used as context (Prompt); alternatively, the relevant code highlighted by the user is used (Prompt-context). The model will then answer the question with that code as context. GILT also allows users to probe the LLM by supporting conversational interaction (Prompt-followup). When previous LLM-generated responses exist, if a user does not highlight any lines from the code, the LLM generates a response with the previous conversion as context. Users can also reset the context by triggering the tool with code highlighted, or with the Clear all button.

3.2 Our Design Process and Decisions

Focus on understanding. We intentionally did not integrate a code generation feature in the prototype as we wanted to focus on how developers *understand* code.

In-IDE extension. Besides anticipating a better user experience, we designed the prototype as an in-IDE extension to more easily provide the code context to the LLM – participants could select code to use as part of the context for a query.

Pre-generated prompts. We designed buttons that query the LLM with pre-generated prompts (*prompt-less* interaction) to ask about an API, conceptual explanations, or usage examples, as shown in Figure 1-④. We chose these based on API learning theory [18, 33, 44, 59], expecting this may particularly assist novice programmers or those unfamiliar with the APIs/domains or the LLM, as writing efficient search queries or prompts can be difficult for novices [17, 19, 32]. At the same time, we also expected that this could reduce the cognitive burden of users in general in formulating prompts.

For Overview and the buttons API, Concept, Usage, we came up with prompt templates after a few iterations. To more efficiently provide the context to LLM, we used the library names and the list of API methods included in the selected code, such as "Please provide a [library name] code example, mainly showing the usage of the following API calls: [list of API methods]" for Usage.

Unrestricted textual queries. Users can also directly prompt the LLM (Figure 1-⑤), in which case GILT will automatically add any selected code as context for the query. Internally, the tool adds the selected code as part of the user prompt using pre-defined templates, and requests the LLM to respond based on the code context.

Need-based explanation generation. The tool is pull-based, i.e., it generates an explanation only when a user requests it. Similar to many previous developer information support tools, we wanted to reduce information overload and distraction. We expect that if and when enough context can be extracted from the IDE, hybrid (pull + push) tools will be possible, but this would require more research.

Iterative design updates. We ran design pilot studies and updated our prototype accordingly. For example, we made the code summary as the default action for the tool trigger with code selection, after seeing pilot participants struggling to find parts of code to work on due to their unfamiliarity with libraries and domains. We updated the prompt-based interaction with LLM to support a conversational interface, based on the pilot participants' feedback that they wanted to probe the model based on their previous queries to clarify their intent or ask for further details. Finally, we opted to use GPT-3.5-turbo instead of GPT-4 as planned, after discovering that the response time was too slow in the pilot studies.

Interactivity and consistency over benchmark accuracy. For the underlying LLM, we used GPT 3.5-turbo. We used OpenAI Node.js Library to query the language model. We set the default sampling temperature as 0.2, which is low, to minimize the variations between the LLM output quality.

4 HUMAN STUDY DESIGN

Participants. We advertised our IRB-approved study widely within the university community (through Slack channels, posted flyers, and personal contacts) and to the public (through Twitter, email, and other channels). We asked each participant about their programming experience and screened out those who reported having a "not at all" experience. We did not ask about their professional programming experience, as the target users of our information support tools are not limited to professional developers. To minimize the possibility of participants knowing solutions, we specifically sought out participants who had not used the libraries included in our study. We accepted participants into the study on a rolling basis to capture a range of programming experience and existing domain knowledge. We compensated each participant with a $25 Amazon Gift card. We recruited 33 participants and conducted 33 studies in total. However, we had to exclude data from one participant from the analysis because they did not follow the instructions for using the extension. In the end, we had 9 women and 23 men participants. Among them, 16 participants identified themselves primarily as students, 1 as software engineer, 2 as data scientists, and 13 as researchers. In the analysis, we divided the participants into two groups (students vs. professionals) based on this. 24 participants had experience with ChatGPT, 15 with Copilot, 5 with Bard, while 7 participants reported no prior use of any LLM-based developer tools. In terms of familiarity with such tools, 14 participants stated that they have either used AI developer tools for their work or always use them for work.

Tasks. The tasks were designed to simulate a scenario in which developers with specific requirements search the web or use existing LLMs to generate code and find similar code that does not precisely match their needs. For each task, we provided participants with a high-level goal of the code, start and goal outputs, and a starter code file loaded into the IDE. In this way, participants had to understand the starter code we provided and make changes to it so that the modified code met the goal requirements. Each task consisted of 4 sub-tasks, to help reduce participant's overhead in planning, as well as to measure the task completion ratio in the analysis. There were some subtle dependencies between the sub-tasks, so we advised participants to follow the order we gave, but they were free to skip. Completing each sub-task required a small change, ranging from a single parameter value update to an addition of a line. The difficulty levels of the sub-tasks varied, but we intentionally designed the first sub-task to be easy so that participants can onboard easily. Sub-tasks also came with start and goal outputs and descriptions (see Figure 2). We did not include tasks that required strong programming knowledge, because our goal was to assess how well participants could understand the code. For the same reason, we provided participants with starter code that was runnable and bug-free.

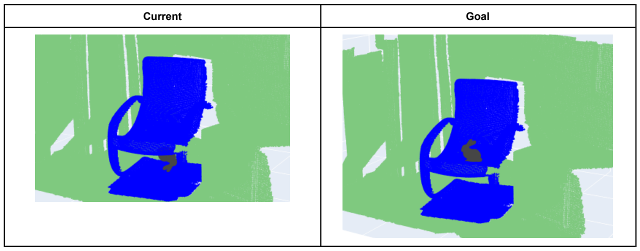


Figure 2: A 3D-rendering example sub-task (open3d-3). With these start and goal outputs, we asked the participants to "Make the bunny sit upright on the chair." See Figure 1 for the corresponding starter code and the tool output.

Our tasks cover both a common and less common domain that a Python developer might encounter in the wild. We chose two domains: data visualization and 3D rendering. These two tasks also allowed participants to easily check their progress, as they produce visual outputs that are comparable with the given goal. For the data visualization task, we used the Bokeh [1] library and asked participants to edit code that visualizes income and expenses data in polar plots. Understanding this code required knowledge of concepts related to visualizing data formats, marks, and data mapping. In the 3D rendering task, we used the Open3d [6] library. This task required knowledge of geometry and computer graphics. Participants were asked to edit code that involved point cloud rendering, transformation, and plane segmentation.

When selecting libraries, we intentionally did not choose the most common ones in their respective domains, to reduce the risk of participants knowing them well. Choosing less common libraries also helped reduce the risk of an outsized advantage of our LLM-powered information generation tool. Responses for popular libraries can be significantly better than those for less commonly used ones, as the quality of LLM-generated answers depends on whether the LLM has seen enough relevant data during training.

The Bokeh starter code consisted of 101 LOC with 11 (6 unique) Bokeh API calls, and the starter code for the Open3D task consisted of 43 LOC with 18 (18 unique) Open3D API calls. The tasks were designed based on tutorial examples in each API's documentation. In the starter codes, we did not include any comments in the code to isolate the effects of the information provided by our prototype or collected from search engines. All necessary API calls were in the starter code so participants did not need to find or add new ones. In the task descriptions, we tried to avoid using domain-specific or library-specific keywords that could potentially provide participants with direct answers from either search engines or GILT. For instance, we opted to use "make the bunny...", instead of "transform the bunny..." which may have steered participants towards the function `transform` without much thought.

The full task descriptions, starter code, solution for the demo, and the actual tasks are available in our replication package.

Experimental Design. We chose a within-subjects design, with participants using both GILT (treatment) and a search engine (control) for code understanding, but they did so on different tasks. This allowed us to ask participants to rate both conditions and provide comparative feedback about both treatments.

The control-condition participants were not allowed to use our prototype, but they were free to use any search engine to find the information they needed. The treatment-condition participants were encouraged to primarily use our prototype for information support. However, if they could not find a specific piece of information using our prototype, we allowed them to use search engines to find it. This was to prevent participants from being entirely blocked by the LLM. We expected that this was a realistic use case of any LLM-based tool, but it rarely happened during the study. Only 2 participants ended up using search engines during the treatment condition, but they could not complete the tasks even with the search engines.

We counterbalanced the tasks and the order they were presented to participants to prevent carryover effects, resulting in four groups (2 conditions x 2 orders). We used random block assignments when assigning participants to each group. Participants were assigned to each group to balance the self-reported programming and domain experience (data visualization and 3D rendering). For every new participant, we randomly assigned them to a group that no previous participant with the same experience level had been assigned. If all groups had previous participants with the same experience level, we randomly assigned the participant to any of them.

Study Protocol. We conducted the study via a video conferencing tool and in person, with each session taking about 90 minutes; in-person participants also used the video conferencing tool, for consistency. At the beginning of the study, we asked participants to complete a pre-study survey, collecting their demographic information, background knowledge, and experience with LLMs. We also estimated their general information processing and learning styles using a cognitive style survey [24] categorizing participants into two groups per dimension: comprehensive / selective information processing and process-oriented learning / tinkering. The participants were then asked to join our web-based VS Code IDE hosted on GitHub CodeSpaces [3], which provided a realistic IDE inside a web browser with edit, run, test, and debug capabilities, without requiring participants to install any software locally [16]. We then showed them a demo task description and explained what they would be working on during the real tasks. Before their first task in the treatment condition, we provided a tutorial for our plugin using the demo task, introducing each feature and giving three example prompts for the LLM. For the control condition, we did not provide any demo, as we expected every participant to be able to use search engines fluently. For each task, we gave participants 20 minutes to complete as many sub-tasks as they could. During the task, we did not use the think-aloud protocol because we wanted to collect timing data. Instead, we collected qualitative data in the post-survey with open-ended questions. We also collected extensive event and interaction logs during the task. After each task, we asked participants to complete a post-task survey to measure their understanding of the provided code and the API calls therein. At the very end, we asked them to complete a post-study survey where we asked them to evaluate the perceived usefulness and perceived ease of use of each code understanding approach and each feature in GILT. We based our questionnaire on the Technology Acceptance Model (TAM) [36], NASA Task Load Index (TLX) [25], and pre- and post-study questionnaires that were previously used in similar studies [55, 59]. See our replication package for the instruments.

We conducted 33 studies in total, with 33 participants. The initial 18 studies were conducted on a one-on-one basis, while some studies in the latter half (involving 15 participants) were carried out in group sessions, with two to five participants simultaneously and one author serving as the moderator. We took great care to ensure that participants did not interrupt each other or share their progress. As mentioned before, we excluded one participant's data and used 32 participants' for the analysis. We discovered this issue after the study, as this participant was part of the largest group session (with five participants).

5 RQ1: EFFECTS OF GILT

In this section, we report on the effectiveness of using GILT in understanding unfamiliar code.

5.1 Data Collection

Code understanding. To evaluate the effectiveness of each condition, we used three measurements: (1) Task completion time: to complete each sub-task; (2) Task progress: we rated the correctness of the participants' solution to each sub-task and measured how many sub-tasks they correctly implemented; and (3) Understanding level: we cross-checked participants' general understanding of the starter code by giving them sets of quiz questions about the APIs included in the starter code. Each set contained three questions, requiring an understanding of the functionalities of each API call. To measure the effect of using GILT and search engines, we excluded the sub-tasks data if participants *guessed* the solution without using the tool (i.e., zero interaction with the tool) or search engines before completing it (i.e., no search queries).

Prior knowledge. To control for prior knowledge, we used self-reported measures of participants' programming and domain experience. We expected more programming experience, especially in the specific domain, to lead to faster understanding of code.

Experience in AI developer tools. Crafting effective prompts for LLM-based tools requires trial and error, even for NLP experts [17, 20, 67]. Therefore, we asked participants about their experience with LLM-based developer tools. We expected participants' familiarity with other AI tools to affect their usage of the

Table 1: Summaries of regressions estimating the effect of using the prototype. Each column summarizes the model for a different outcome variable. We report the coefficient estimates with the standard errors in parentheses.

	Progress	*Time(s)*	*Underst.*	Progress	
	(1)	(2)	(3)	*Pros*	*Students*
Constant	0.41	312.65	−1.81**	−0.38	1.82**
	(0.49)	(185.33)	(0.89)	(0.68)	(0.83)
Domain	0.13*	23.14	0.41***	0.16	0.04
experience	(0.07)	(25.40)	(0.12)	(0.09)	(0.11)
Program.	−0.10	−23.67	0.20	0.01	−0.37*
experience	(0.12)	(43.53)	(0.22)	(0.17)	(0.21)
AI tool	−0.01	7.70	−0.09	0.07	−0.10
familiarity	(0.07)	(27.04)	(0.14)	(0.11)	(0.10)
Uses GILT	0.47***	−9.10	0.29	0.57**	0.29
	(0.16)	(57.26)	(0.28)	(0.22)	(0.25)
R^2	0.173	0.022	0.202	0.341	0.137
Adj. R^2	0.117	−0.046	0.148	0.243	0.010

Note: *p <0.1; **p <0.05; ***p <0.01.

LLM-based information support tool, especially the use of free-form queries, and lead to more effective use of the extension than participants without such experience.

5.2 Methodology

To answer RQ1, we compared the effectiveness of using a GILT with traditional search engines for completing programming tasks by estimating regression models for three outcome variables. For task progress and code understanding, we used quasi-Poisson models because we are modeling count variables, and for the task completion time, we used a linear regression model.

To account for potential confounding factors, we included task experience, programming experience, and LLM knowledge as control variables in our models. Finally, we used a dummy variable (`uses_GILT`) to indicate the condition (using GILT vs. using search engines). We considered mixed-effects regression but used fixed effects only, since each participant and task appear only once in the two conditions (with and without GILT). For example, for the task completion time response, we estimate the model:

```
completion_time ~ domain_experience + programming_experience
                                 + AI_tool_familiarity + uses_GILT
```

The estimated coefficient for the `uses_GILT` variable indicates the effect of using GILT while holding fixed the effects of programming experience, domain experience, and LLM knowledge.

5.3 Results

Table 1 columns (1)-(3) display the regression results for three response variables. The task progress model (Table 1-(1)) shows a significant difference between the two conditions, with participants in the GILT condition completing statistically significantly more sub-tasks (0.47 more, $p < 0.01$) than those who used search engines, controlling for experience levels and AI tool familiarity. This indicates that GILT may assist users in making more progress in their tasks compared to search engines.

On the other hand, models (2) and (3) fail to show any significant difference in completion time and code understanding quiz scores between conditions. This suggests that users in the GILT condition do not complete their tasks at a sufficiently different speed or have a sufficiently different level of understanding than those in the control group, given the statistical power of our experiment.

In summary, the results suggest that GILT may help users make more progress in their tasks without changing, for better or worse, their speed and code understanding abilities.

5.4 Additional Analysis

After observing the significant effect of GILT on task progress, we dove deeper to examine whether all participants benefited equally from the tool. To do this, we divided the participants into two distinct groups based on their self-reported occupations (professionals and students) and estimated the effects of GILT usage in each group.[1] We opted for these groups as we did not have any prior theoretical framework to guide our grouping choices, and it provided a simple yet effective approach to group participants with multiple dimensions, including programming experience, skills, and attitude toward programming.

Although both groups were more successful when using the tool, there were notable differences in their performance gains. To better understand these variations, we estimated coefficients for each group (Table 1-Pros and -Students) and observed that the impact of GILT was significant only in the Pros group model. Specifically, professionals completed 0.57 more sub-tasks with GILT support compared to when they used search engines, whereas students did not experience significant gains. These findings suggest that the degree of benefit provided by GILT may vary depending on participants' backgrounds or skills.

6 RQ2: GILT USAGE

In this section, we focus on how participants interacted with GILT, their perception of the importance of different features, and how different factors correlate with the feature usage.

6.1 Usage of Features

To analyze in more detail how participants actually used the tool, we instrumented GILT and recorded participants' event and interaction logs. The logs allowed us to count the number of times participants triggered each feature, and in what order. To supplement the usage data, participants were asked to rate the importance of each feature in a post-task survey. We used these ratings to triangulate our findings from the usage data.

Figure 3 summarizes the sequences of GILT features used by participants in the treatment condition. On average, to complete their tasks in this condition, participants interacted with the LLM via GILT 15.34 times. The number of interactions per participant ranged from a minimum of 5 to a maximum of 23. The `Overview` feature was the most frequently used method to interact with the LLM, with

[1] We considered but decided against, modeling interaction effects as they would have required more statistical power.

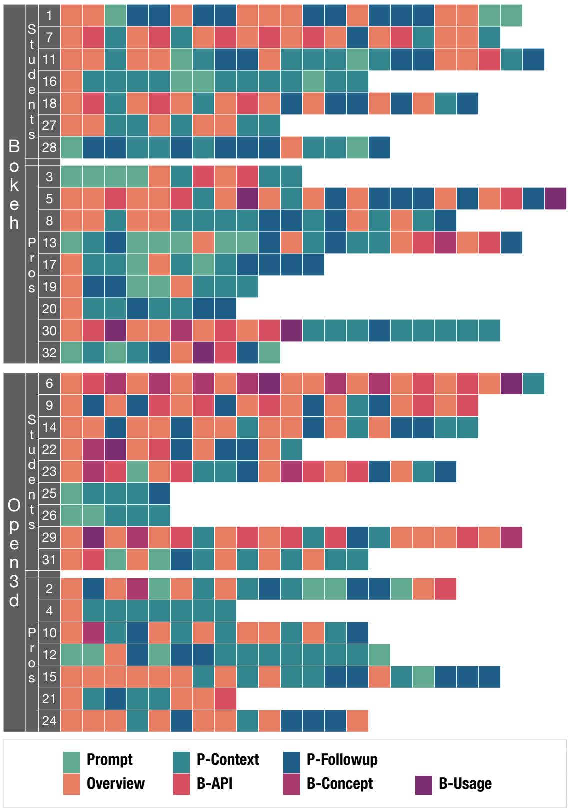


Figure 3: The sequences of feature usage in GILT. Each row corresponds to an individual participant, and the color cells are arranged chronologically, from left to right.

Table 2: Frequencies of n-grams used differently in prompts by professionals and students. For clarity, we only include n-grams used uniquely by one of the two groups, with a frequency difference of more than 2. If multiple n-grams shared the same longer n-gram, we report only the superset.

Sub-task	n-gram	Pro.	Stu.
bokeh-2	('align', 'text')	3	0
	('flip', 'label')	0	3
bokeh-3	('annular', 'wedge')	6	0
	('grid', 'annular', 'wedge')	3	0
	('first', 'pie')	3	0
	('pie', 'chart')	3	0
	('add', 'legend')	0	4
	('tell', 'line', 'need', 'change')	0	3
o3d-3	('sit', 'upright', 'chair')	4	0
	('make', 'bunny', 'sit', 'upright', 'chair')	3	0

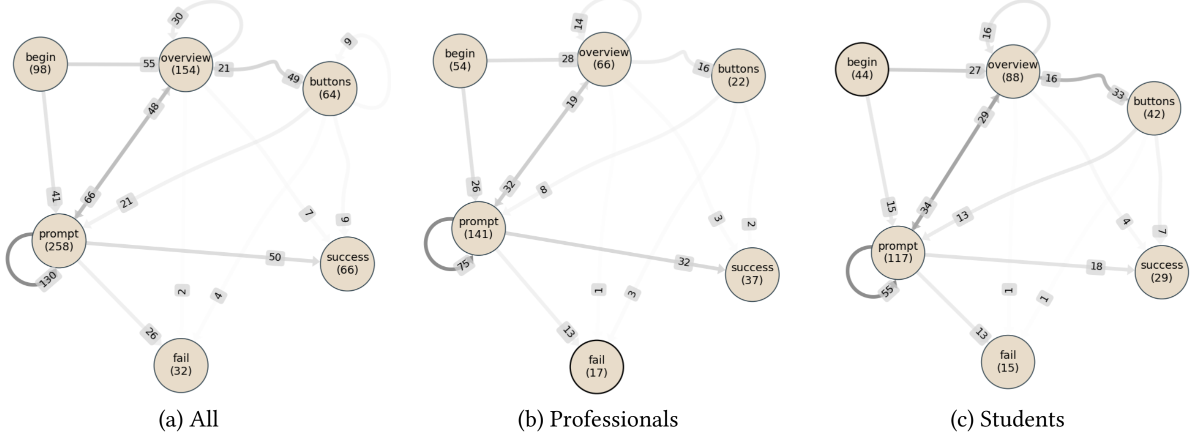
an average of 4.76 activations per participant. Many participants also used `Overview` as their first feature, possibly because it requires minimal effort, with just a single click, in contrast to other features that necessitated the formulation of queries by participants, and perhaps also because some of the buttons (e.g., `Concept`) required first using the `Overview` feature. Participants also frequently used `Prompt-context` (4.12 times) and `Prompt-followup` (2.88 times). General prompting without code context was used less frequently (1.27 times). While participants generally used buttons less frequently, some used them more frequently than queries (e.g., P29), indicating personal preferences in prompt-based and prompt-less interactions. Specifically, the `API` button was used 1.24 times, the `Concept` button 0.45 times, and the `Usage` button 0.24 times on average.

To further investigate participants' interaction with the tool, we created transition graphs (Figure 4) using sequences of feature use events for each sub-task, using both the sub-tasks successfully completed by participants and those that resulted in failure (due to incorrect answers or timeouts). Out of the potential total of 128 sub-tasks (32 participants × 4 sub-tasks), 98 sub-tasks were started before the time ran out. In understanding the transition graph, we focused on the last feature in each participant's sequence, with an assumption that when a participant completes a task, it is likely that the information from the last interaction satisfied their information needs. Among the sub-tasks they successfully completed, a substantial majority (75%) originated from prompt-based interactions. At the same time, 83% of the failed tasks were also preceded by prompt-based interactions, so prompt-based interactions were not particularly likely to result in successful information seeking.

The reported importance of the features by participants (summary included in our replication package) generally corresponds to the observed usage data. Most of the participants (97%) responded that the ability to directly ask questions to the LLM was extremely/very important, whereas their reported usefulness of the buttons varied. The reported importance of the overview feature (53% extremely/very important) was relatively low compared to the actual use, suggesting that participants may not have used the summary description provided by the overview but instead used it as context for further prompting or to activate buttons.

6.2 Professionals vs. Students

To better understand the experiences of professionals and students (see Section 5.4), we compared the transition graphs for both groups (Figure 4 (b) and (c)). Notable distinctions emerged in terms of the features more likely influencing the success and failure of sub-tasks. Specifically, for professionals, a majority (86%) of successful sub-tasks originated from `prompt`, whereas for students, this percentage (62%) was statistically significantly lower ($\chi^2(1, 66)$, $p < .05$). The success rate of prompt-based interaction was also higher among professionals (71%: 32 out of 45) compared to students (58%: 18 out of 31). Conversely, the success rate of the overview and buttons for professionals (56%: 5 out of 9) was lower than that of students (85%: 11 out of 13). These results may indicate that students, possibly with less experience in information seeking for programming, encounter challenges in formulating effective prompts compared to the professionals, and relied more on prompt-less interaction. However, we can also infer that prompt-less interaction is still not sufficient to compete with the benefits of prompt-based interaction with the current design, as they only accounted for less than 40% of the completed tasks.



(a) All (b) Professionals (c) Students

Figure 4: Transition Graphs for User Interaction. Each node displays the number of times users interacted with respective features, and each edge indicates the counted number of transitions between the connected features. For space and readability reasons, `Prompt`, `Prompt-Context`, and `Prompt-Followup` are merged into `prompt`, and `API`, `Concept`, and `Usage` are merged into `buttons`. Counts lower than 5 are omitted except for the edges connected to the 'Success' and 'Fail' nodes.

To further investigate the differences in the two groups' prompt engineering, we analyzed the text of the prompts they wrote, by comparing the frequencies of bi-, tri-, and quadrigrams in the prompts. Table 2 presents the list of n-grams that showed divergent usage between the two groups. One notable observation is that the n-grams used by the professionals group include more effective keywords, or they revise the prompts to incorporate such keywords. For instance, in the bokeh-3 sub-task, none of the participants in the student group used the critical keyword "annular wedge," which is essential for generating the information needed to solve the task, although it was used multiple times in the provided starting code. Instead, students tended to use more general keywords or keywords that had a different concept in the library (e.g., "legend") and faced difficulties in effectively revising the prompts. In addition, more participants in the professionals group demonstrated proficiency in refining their prompts by providing further specifications. For example, one participant revised the prompt from "How to change the position of the bunny to 180 degrees" to "How to transform the bunny_mesh to 180 degrees." We infer that the difference in the benefit received from GILT by the two groups can be at least partially attributed to their proficiency in prompt engineering.

Table 3: Summaries of regressions testing for associations between the user factors and the feature usage counts. Each column summarizes a regression modeling a different outcome variables. We report the coefficient estimates with their standard errors in parentheses.

	Prompt	Followup	All
	(1)	(2)	(3)
Constant	1.39***	−0.82	2.43***
	(0.31)	(0.69)	(0.27)
AI tool	0.19**	0.38**	0.11
familiarity	(0.07)	(0.15)	(0.06)
Infomation	−0.04	0.44	−0.04
Comprh.	(1.15)	(0.30)	(0.13)
Learning	0.19	0.60**	−0.12
Process	(1.14)	(0.29)	(1.13)
R^2	0.262	0.283	0.165
Adj. R^2	0.184	0.206	0.075

Note: *p <0.1; **p <0.05; ***p <0.01.

6.3 Other Factors Associated with Feature Use

During the pilot studies, we observed that participants approached the tasks differently depending on their familiarity with other LLM-based tools, and styles of information processing and learning as observed in many previous studies on software documentation and debugging [9, 44, 45]. Thus, we tested whether the GILT feature use correlates with factors other than their experience.

Hypotheses. Out of two information processing styles [12, 15, 23], people who exhibit a "selective information processor" tendency focus on the first promising option, pursuing it deeply before seeking additional information. On the other hand, people who are "comprehensive information processors" tend to gather information broadly to form a complete understanding of the problem before attempting to solve it. Based on these processing styles, we hypothesized that selective processors would utilize GILT's `Prompt-followup`, as they would prefer to use a depth-first strategy.

In terms of learning styles [12, 48], "process-oriented learners" prefer tutorials and how-to videos, while "tinkerers" like to experiment with features to develop their own understanding of the software's inner workings. Consequently, we hypothesized that tinkerers would use GILT less often, as they would prefer to tinker with the source code rather than collect information from the tool.

We also expected that participants who were already familiar with LLM-based tools would use prompt-based interaction in general (`Prompt`), especially the chat feature, more frequently, as they would already be accustomed to using chat interfaces to interact with LLMs. Conversely, we posited that participants with less experience with such tools would use the buttons more, as prompt engineering might be less familiar to them and place greater cognitive demands on them.

Methodology. To test for associations between GILT features used and the factors above we again used multiple regression analysis. We estimated three models, each focused on one particular feature. For each model, the dependent variable was the feature usage count, while participants' information processing style, learning style, and familiarity with AI developer tools were modeled as independent variables to explain the variation in usage counts.

Results. Table 3 presents the results of the regression analysis conducted for three response variables. The first model (Prompt (1)), which uses the total count of prompt-based interactions (`prompt` + `prompt-context` + `prompt-followup`), reveals that developers who are more familiar with other AI developer tools are more likely to prompt the LLMs using natural language queries. This result confirms our hypothesis that the AI tool familiarity level influences developers' use of queries. The familiarity level also has a statistically significant impact on `prompt-followup`, as shown in the Followup model (2). However, we did not find any significant impact of participants' information processing style on their use of GILT. This means that selective processors and comprehensive processors probed the LLMs similarly, as far as we can tell. The model, however, shows a statistically significant correlation between participants' learning styles and `prompt-followup` feature usage. Specifically, process-oriented learners tend to probe LLMs more frequently than tinkerers. This result might indicate that process-oriented learners are more likely to learn thoroughly before proceeding to the next step, while tinkerers tend to tinker with the code after getting the minimum amount of direction from GILT. Finally, the All model (3), which uses the total count of all GILT interactions, indicates that there is no statistically significant difference between the information styles, learning styles, and familiarity levels in terms of overall feature usage counts.

7 RQ3: USER PERCEPTIONS

In this section, we investigate how participants perceived their experience of using GILT. Specifically, we examine their perceived usefulness, usability, and cognitive load in comparison to search-based information seeking. Additionally, we explore the pros and cons participants reported, and suggestions for improving the tool.

7.1 Comparison with Web Search

We employed two wildly-used standard measures, TLX and TAM, in our post-task survey and compared them using two-tailed paired t-tests. TAM (Technology Acceptance Model) [36] is a widely used survey that assesses users' acceptance and adoption of new technologies, and TLX (Task Load Index) [25] is a subjective measure of mental workload that considers several dimensions, including mental, physical, and temporal demand, effort, frustration, and performance. The summaries of TAM and TLX comparisons can be found in our replication package.

The average scores for the [perceived usefulness, perceived ease of use] in TLX scales were [27.3, 29.75] for the control condition, and [33.49, 34.2] for the treatment condition. The paired t-tests on the TAM scores indicated that there were significant differences in perceived usefulness and perceived usability scores between the two conditions ($p < 0.001$). Specifically, participants rated GILT higher on both dimensions than they did search engines.

For TLX items [mental demand, physical demand, temporal demand, performance, effort, frustration], the average scores were [3.8, -2.1, 4.0, 1.6, 3.4, -0.1] for the control condition and [3.3, -2.5, 2.6, 3.3, 3.3, 1.0] for the treatment condition. Paired t-tests on the TLX scores revealed statistically significant differences between the tool and search engines in temporal demand ($p < 0.05$) and performance ($p < 0.05$) but not in other items. These results indicate that the participants felt less rushed when using GILT than when using search engines, and they felt more successful in accomplishing the task with the tool than with search engines, but there were no significant differences in other dimensions.

7.2 User Feedback

In the post-task survey, we asked open-ended questions regarding their general experience with using GILT compared with web search-based information seeking. Two authors conducted a thematic analysis [14] to analyze the answers. Initially, two authors separately performed open coding on the same set of 8 responses (25% of the entire data), and convened to discuss and merge the codes into a shared codebook. The first author coded the rest of the responses and discussed with the rest of the authors whenever new codes needed to be added. The codebook is available in our replication package, and we discuss some of them here.

The participants in this study reported several positive aspects of the tool, with the most notable being context incorporation. Participants valued the ability to prompt the LLM with their code as context, which allowed them to tailor the LLM's suggestions to their specific programming context, e.g., "the extension generated code that could easily be used in the context of the task I was performing, without much modification." (P5) Participants also found it extremely useful to prompt the LLM with just code context, as it allowed them to bypass the need to write proficient queries, a well-known challenge in search-based information seeking [32, 65]. P15 mentioned "*It's nice not to need to know anything about the context before being effective in your search strategy.*"

Many participants reported that using the tool helped them speed up their information seeking, by reducing the need to forage for information, e.g., "Stack Overflow or a Google search would require more time and effort in order to find the exact issue and hence would be time-consuming." (P27)

Some participants, however, reported having a hard time finding a good prompt that could give them the desired response. Combined with the need for good prompts and the limitations of LLM, this led some participants to report that the responses provided by the tool were occasionally inaccurate, reducing their productivity. P28 summarized this issue well: "*[prototype] was not able to give me the code that I was looking for, so it took up all my time (which I got very annoyed about). I think I just didn't word the question well.*"

Participants had mixed opinions on the different features of the tool, especially the buttons. Some preferred to use "*different buttons for different types of information so I didn't have to read a lot of text* *to find what I was looking for*" (P7), while others thought that was overkill and mentioned "*a simpler view would be nice.*" (P8)

Compared to ChatGPT, 17 participants (out of 19 who answered) mentioned advantages of GILT, with the `Prompt- context` feature being one of the main ones. Participants expressed positive feelings about CoPilot but acknowledged that the tool had a different role than CoPilot and that they would be complementary to each other, e.g.: "*Copilot is a tool that I can complete mechanical works quickly, but [GILT] offers insight into more challenging tasks.*" (P29)

Many participants reported that the tool would be even more useful when combined with search engines, API documentation, or CoPilot,[2] as they provide different types of information than the tool. Having the ability to choose sources based on their needs would enhance their productivity by giving them control over the trade-offs, such as speed, correctness, and adaptability of the information.

8 THREATS TO VALIDITY

One potential concern with our study design is the task and library selection. We only used tasks that show visible outputs, which might have led participants to detect potential errors more easily, compared to other tasks, such as optimization or parallel programming. However, we believe that the tasks we chose are representative of common programming errors that would need to be identified in real-world programming situations. Indeed, when we asked the participants in the post-task survey, both data visualization and 3D rendering tasks were reported to very or extremely closely resemble real-world tasks by 82% and 73% of the participants.

Similarly, the selection of libraries might have biased the study results. However, in selecting libraries for our study, we avoided using popular libraries that could unintentionally give an advantage to LLMs. We believe that the libraries we chose are of medium size and quality, and therefore represent a fair test of the LLM tools. However, it is possible that different libraries or larger codebases could produce different results.

Despite our efforts to create a controlled experience, several factors differentiate our in-IDE extension from search engines, aside from the inclusion of LLMs. For example, although previous research investigating the incorporation of search into IDE did not find a statistically significant difference between the control and treatment groups [10, 38], the in-IDE design itself may have been more helpful than access to LLMs, as it potentially reduced context-switching. Thus, further studies are needed to gain a better understanding of the extent to which each benefit of our prototype can be attributed to these differences.

Additionally, the laboratory setting may not fully capture the complexity of real-world programming tasks, which could impact the generalizability of our findings. Also, the time pressure participants could have felt, and the novelty effect in a lab setting could have changed how users interact with LLMs. Our sample size, 32, was relatively small and skewed towards those in academia. This may also limit the generalizability of our findings to more professional programmers. Thus, future research with larger, more diverse samples is necessary to confirm and expand upon our results.

Our analysis also has the standard threats to statistical conclusion validity affecting regression models. Overall, we took several steps to increase the robustness of our estimated regression results. First, we removed outliers from the top 1% most extreme values. Second, we checked for multicollinearity using the Variation Influence Factor (VIF) and confirmed that all variables we used had VIF lower than 2.5 following Johnston et al. [28].

Another potential threat to the validity of our findings is the rapid pace of technological development in the field of LLM tools. Despite our efforts to use the most up-to-date LLM available at the time of the submission, it is possible that new breakthroughs could render our findings obsolete before long.

9 DISCUSSION AND IMPLICATIONS

Comprehension outsourcing. Our analysis revealed an intriguing finding regarding participants' behavior during the study, where some of them deferred their need for code comprehension to the LLM, which was well described by one participant as *comprehension outsourcing*. These participants prompted the model at a higher level directly and did not read and fully comprehend the code before making changes. As one participant commented, "*I was surprised by how little I had to know about (or even read) the starter code before I can jump in and make changes.*" This behavior might be attributed to developers' inclination to focus on task completion rather than comprehending the software, as reported in the literature [42]. Or, participants may have also weighed the costs and risks of comprehending code themselves, and chosen to defer their comprehension efforts to the language model. While this behavior was observed in the controlled setting of a lab study and may not fully reflect how developers approach code comprehension in their daily work, it does raise concerns about the potential impact of such a trend (or over-reliance on LLMs [62]) on code quality. This highlights the importance of preventing developers who tend to defer their comprehension efforts to the LLM from being steered in directions that neither they nor the LLM are adequately equipped to handle. Studies showing developers' heavy reliance on Stack Overflow, despite its known limitations in accuracy and currency [64, 69], further emphasize the need for caution before widely adopting LLM-based tools in code development. Research on developers' motivations and reasons for code comprehension when LLMs are available will be valuable in informing future tool designs.

Need for more research in UI. In our analysis, we observed a notable trend where the professionals benefited *more* from the tool compared to students. Our examination of the prompts indicated that this discrepancy may arise because students face challenges in constructing effective queries or revising them to obtain useful information, aligning with findings in the literature on code generation using LLMs [17]. Although we provided an option to use prompt-less interaction with LLMs to reduce the difficulty in prompt engineering, a lot of participants chose to use prompt-based interaction, possibly due to their familiarity with other AI tools, the potentially higher quality of information this mode produces, or other reasons that our study did not cover. However, we find our results still promising, as we observed that students used prompt-less interaction more than the professionals and succeeded more when using the buttons than using the prompts. We believe that

[2] Notably, GitHub independently announced these enhancements to Copilot already, after we conducted our study: https://www.theverge.com/2023/7/20/23801498/github-copilot-x-chat-code-chatbot-public-beta

further research is needed, exploring various interaction options to support a diverse developer population.

Need further studies in real-world settings. One possible explanation for some of the models with null results from RQ1 and RQ2 is the artificial setting of the lab study, where participants were encouraged to focus on small, specific task requirements instead of exploring the broader information dimension. For example, participants prioritized completing more tasks rather than fully understanding the code, as reported by participant P18 in their survey response: " *[GILT] ..., which could definitely help one to tackle the task better if there weren't under the timed-settings.*" Thus, although our first study shed some light on the potential challenges and promises, to fully understand the implications of deploying this tool into general developer pipelines, it is necessary to observe how programmers use it in real-world settings with larger-scale software systems, less specific goals, and over a longer time frame. Given that GitHub recently launched CopilotX [5], a tool that offers a comparable set of features to our prototype to enhance developer experience, such research is urgently needed. We believe that our findings are a timely contribution and a good first step for researchers and tool builders in designing and developing developer assistants that effectively use LLMs.

10 CONCLUSION

We presented the results of a user study that aimed to investigate the effectiveness of generation-based information support using LLMs to aid developers in code understanding. With our in-IDE prototype tool, GILT, we demonstrated that this approach significantly enhances developers' ability to complete tasks compared to traditional search-based information seeking. At the same time, we also identified that the degree of benefits developers can get from the tool differs between students and the professionals, and the way developers interact with the tool varies based on their learning styles and familiarity with other AI tools.

Data Availability. Our supplementary material includes the replication package, including the study protocol, tasks, study data, scripts to replicate the analyses, as well as the prototype tool.

REFERENCES

[1] [n. d.]. Bokeh. about:blank. Retrieved: 2023-08-01.

[2] [n. d.]. ChatGPT|OpenAI. https://chat.openai.com/. Retrieved: 2023-08-01.

[3] [n. d.]. GitHub Codespaces. https://github.com/features/codespaces. Retrieved: 2023-08-01.

[4] [n. d.]. GitHub Copilot. https://github.com/features/copilot. Retrieved: 2023-08-01.

[5] [n. d.]. GitHub Copilot X: The AI-powered developer experience. https://github.com/features/preview/copilot-x. Retrieved: 2023-08-01.

[6] [n. d.]. Open3D – A Modern Library for 3D Data Processing. http://www.open3d.org/. Retrieved: 2023-08-01.

[7] [n. d.]. Tabnine: AI assistant for software developers. https://www.tabnine.com/. Retrieved: 2023-08-01.

[8] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. *arXiv* (2022). https://doi.org/10.48550/arxiv.2206.15000 arXiv:2206.15000

[9] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. 2006. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 231–240.

[10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 513–522.

[11] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[12] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. 2016. GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers* 28, 6 (2016), 760–787.

[13] Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software* 82, 6 (2009), 981–992.

[14] Victoria Clarke and Virginia Braun. 2013. Teaching thematic analysis: Overcoming challenges and developing strategies for effective learning. *The psychologist* 26, 2 (2013), 120–123.

[15] William K Darley and Robert E Smith. 1995. Gender differences in information processing strategies: An empirical test of the selectivity model in advertising response. *Journal of advertising* 24, 1 (1995), 41–56.

[16] Matthew C Davis, Emad Aghayi, Thomas D LaToza, Xiaoyin Wang, Brad A Myers, and Joshua Sunshine. 2022. What's (not) Working in Programmer User Studies? *ACM Transactions on Software Engineering and Methodology* (2022).

[17] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. 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, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, Maureen Doyle, Ben Stephenson, Brian Dorn, Leen-Kiat Soh, and Lina Battestilli (Eds.). ACM, 1136–1142. https://doi.org/10.1145/3545945.3569823

[18] Ekwa Duala-Ekoko and Martin P Robillard. 2010. *The information gathering strategies of API learners*. Technical Report. Technical report, TR-2010.6, School of Computer Science, McGill University.

[19] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 266–276. https://doi.org/10.1109/ICSE.2012.6227187

[20] Jean-Baptiste Döderlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale. 2022. Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic? *arXiv* (2022). https://doi.org/10.48550/arxiv.2210.14699 arXiv:2210.14699

[21] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *ACE '22: Australasian Computing Education Conference, Virtual Event, Australia, February 14 - 18, 2022*, Judy Sheard and Paul Denny (Eds.). ACM, 10–19. https://doi.org/10.1145/3511861.3511863

[22] Luanne Freund. 2015. Contextualizing the information-seeking behavior of software engineers. *Journal of the Association for Information Science and Technology* 66, 8 (2015), 1594–1605.

[23] Valentina I Grigoreanu, Margaret M Burnett, and George G Robertson. 2010. A strategy-centric approach to the design of end-user debugging tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 713–722.

[24] Md Montaser Hamid, Amreeta Chatterjee, Mariam Guizani, Andrew Anderson, Fatima Moussaoui, Sarah Yang, I Escobar, Anita Sarma, and Margaret Burnett. 2023. How to measure diversity actionably in technology. *Equity, Diversity, and Inclusion in Software Engineering: Best Practices and Insights* (2023).

[25] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[26] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. *arXiv* (2023). https://doi.org/10.1145/3568813.3600139 arXiv:2306.05715

[27] 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.

[28] Ron Johnston, Kelvyn Jones, and David Manley. 2018. Confounding and collinearity in regression analysis: a cautionary tale and an alternative procedure, illustrated by studies of British voting behaviour. *Quality & quantity* 52 (2018), 1957–1976.

[29] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. *arXiv* (2023). https://doi.org/10.1145/3544548.3580919 arXiv:2302.07427

[30] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. *29th International Conference on Software Engineering (ICSE'07)* (2007), 1–10. https://doi.org/10.1109/icse.2007.45

[31] Andrew Jensen Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (11 2006), 971 – 987. https://doi.org/10.1109/tse.2006.116

[32] Amy J. Ko and Yann Riche. 2011. The role of conceptual knowledge in API usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, Gennaro Costagliola, Amy J. Ko, Allen Cypher, Jeffrey Nichols, Christopher Scaffidi, Caitlin Kelleher, and Brad A. Myers (Eds.). IEEE, 173–176. https://doi.org/10.1109/VLHCC.2011.6070395

[33] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. 2007. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 361–370.

[34] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. 492–501.

[35] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. 2010. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2010), 197–215.

[36] Younghwa Lee, Kenneth A Kozar, and Kai RT Larsen. 2003. The technology acceptance model: Past, present, and future. *Communications of the Association for information systems* 12, 1 (2003), 50.

[37] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. *arXiv* (2023). arXiv:2304.03938

[38] Hongwei Li, Xuejiao Zhao, Zhenchang Xing, Lingfeng Bao, Xin Peng, Dongjing Gao, and Wenyun Zhao. 2015. amAssist: In-IDE ambient search of online programming resources. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 390–398.

[39] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. *arXiv preprint arXiv:2303.17125* (2023).

[40] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *CoRR* abs/2305.01210 (2023). https://doi.org/10.48550/arXiv.2305.01210 arXiv:2305.01210

[41] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[42] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 4 (09 2014), 31 – 37. https://doi.org/10.1145/2622669

[43] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2022. Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book. *arXiv* (2022). arXiv:2211.02265

[44] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2017. Application Programming Interface Documentation: What Do Software Developers Want?:. *Journal of Technical Writing and Communication* 48, 3 (07 2017), 295 – 330. https://doi.org/10.1177/0047281617721853

[45] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. How developers use API documentation: an observation study. *dl.acm.org* (2018). https://doi.org/10.1145/3274995.3274999

[46] Andre N. Meyer, Laura E. Barton, Gail C. Murphy, Thomas Zimmermann, and Thomas Fritz. 2017. The Work Life of Developers: Activities, Switches and Perceived Productivity. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1178–1193. https://doi.org/10.1109/tse.2017.2656886

[47] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *arXiv* (2022). https://doi.org/10.48550/arxiv.2210.14306 arXiv:2210.14306

[48] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of learning in novice programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55.

[49] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? *CoRR* abs/2211.03622 (2022). https://doi.org/10.48550/arXiv.2211.03622 arXiv:2211.03622

[50] David Piorkowski, Austin Z Henley, Tahmid Nabi, Scott D Fleming, Christopher Scaffidi, and Margaret Burnett. 2016. Foraging and navigations, fundamentally: developers' predictions of value and cost *(the 2016 24th ACM SIGSOFT International Symposium)*. 97 – 108. https://doi.org/10.1145/2950290.2950302

[51] Md. Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andres Lois, Sebastian Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 465–475. https://doi.org/10.1145/3196398.3196425

[52] Nikitha Rao, Chetan Bansal, Thomas Zimmermann, Ahmed Hassan Awadallah, and Nachiappan Nagappan. 2020. Analyzing web search behavior for software engineering tasks. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 768–777.

[53] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[54] Martin P Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[55] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. *arXiv* (2023). https://doi.org/10.1145/3581641.3584037 arXiv:2302.07080

[56] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv* (2022). https://doi.org/10.48550/arxiv.2208.06213 arXiv:2208.06213

[57] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations using Large Language Models. *arXiv* (2022). https://doi.org/10.48550/arxiv.2206.11861 arXiv:2206.11861

[58] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Software Eng.* 34, 4 (2008), 434–451. https://doi.org/10.1109/TSE.2008.26

[59] Kyle Thayer, Sarah E Chasins, and Amy J Ko. 2021. A Theory of Robust API Knowledge. *ACM Transactions on Computing Education* 21, 1 (2021), 1–32. https://doi.org/10.1145/3444945

[60] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant - How far is it? *CoRR* abs/2304.11938 (2023). https://doi.org/10.48550/arXiv.2304.11938 arXiv:2304.11938

[61] Gias Uddin and Martin P Robillard. 2015. How API documentation fails. *Ieee software* 32, 4 (2015), 68–75.

[62] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts* (2022), 1–7. https://doi.org/10.1145/3491101.3519665

[63] Shaowei Wang, Tse-Hsun Chen, and Ahmed E. Hassan. 2020. How Do Users Revise Answers on Technical Q&A Websites? A Case Study on Stack Overflow. *IEEE Trans. Software Eng.* 46, 9 (2020), 1024–1038. https://doi.org/10.1109/TSE.2018.2874470

[64] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24 (2019), 637–673.

[65] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22 (2017), 3149–3185.

[66] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv* (2022). https://doi.org/10.48550/arxiv.2202.13169 arXiv:2202.13169

[67] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. 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*. 1–21.

[68] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L. Glassman. 2020. Enabling Data-Driven API Design with Community Usage Data: A Need-Finding Study. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–13. https://doi.org/10.1145/3313831.3376382

[69] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th international conference on software engineering*. 886–896.

[70] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. *arXiv* (2022). https://doi.org/10.48550/arxiv.2205.06537 arXiv:2205.06537