Need Help? Designing Proactive AI Assistants for Programming

Valerie Chen
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
valeriechen@cmu.edu

Alan Zhu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
alanzhuyixuan@gmail.com

Sebastian Zhao
University of California Berkeley
Berkeley, California, USA
sebbyzhao@berkeley.edu

Hussein Mozannar
Microsoft Research
Redmond, Washington, USA
hmozannar@microsoft.com

David Sontag
Massachusetts Institute of Technology
Boston, Massachusetts, USA
dsontag@mit.edu

Ameet Talwalkar
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
atalwalk@andrew.cmu.edu

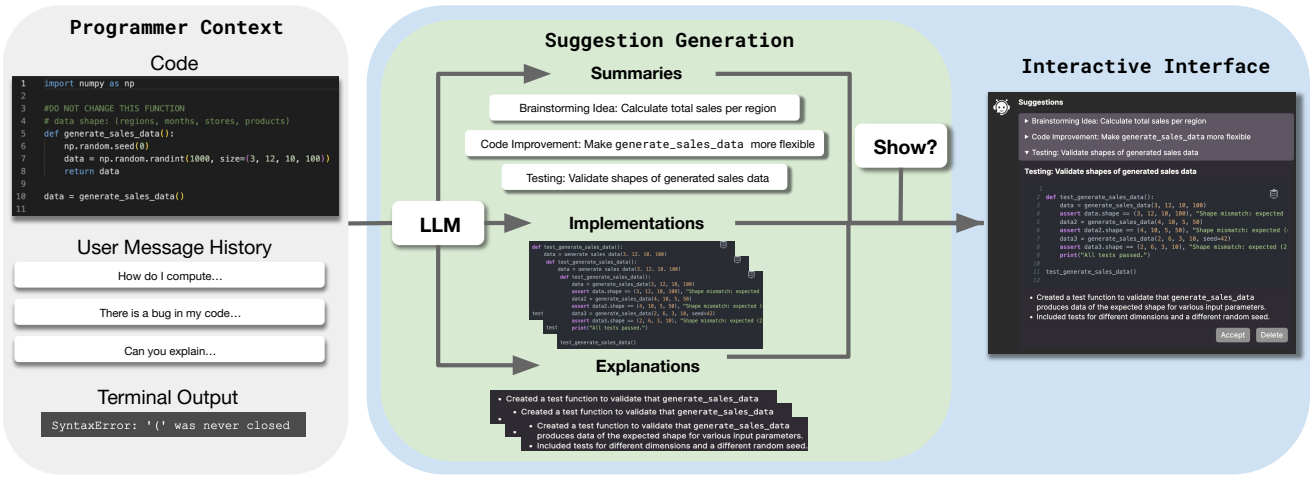

Figure 1: The implementation of a proactive chat assistant for programming. We introduce a proactive assistant that takes in the programmer's context, which includes the current code, user message history, and optionally terminal output, generates a set of suggestions, which include a summary, implementation, and explanation of implementation, and then determines whether it is timely to show the user in the interactive interface.

Abstract

While current chat-based AI assistants primarily operate reactively, responding only when prompted by users, there is significant potential for these systems to proactively assist in tasks without explicit invocation, enabling a mixed-initiative interaction. This work explores the design and implementation of proactive AI assistants powered by large language models. We first outline the key design considerations for building effective proactive assistants. As a case study, we propose a proactive chat-based programming assistant that automatically provides suggestions and facilitates their integration into the programmer's code. The programming context provides a shared workspace enabling the assistant to offer more relevant suggestions. We conducted a randomized experimental study examining the impact of various design elements of the proactive assistant on programmer productivity and user experience. Our findings reveal significant benefits of incorporating proactive chat assistants into coding environments, while also uncovering important nuances that influence their usage and effectiveness.

CCS Concepts

• **Human-centered computing** → **User studies**; • **Software and its engineering** → **Collaboration in software development**.

Keywords

AI-assisted Programming, Proactivity, Mixed-Initiative Interaction

CHI '25, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3714002

ACM Reference Format:
Valerie Chen, Alan Zhu, Sebastian Zhao, Hussein Mozannar, David Sontag, and Ameet Talwalkar. 2025. Need Help? Designing Proactive AI Assistants for Programming. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3706598.3714002

1 Introduction

Chat-based AI assistants such as ChatGPT [39] or Claude [3], enable people to accomplish some of their tasks via prompting: the user crafts a message with sufficient context about their task and then receives an AI response, they continue iterating this process until they're satisfied with the result. Common use cases of chat-based AI assistance include writing tasks [25, 26], idea generation [45], programming tasks [35, 52], and web navigation among other tasks [30, 55]. Interacting with these assistants requires two points of effort from the human: first porting over the workspace context, e.g., copying the document or code file, and second describing the task in natural language. Assuming the assistant had access to the working context of the user, a natural question is whether it could *infer the task they want to solve.* If the assistant can reliably provide suggestions relevant to the user's task, then it might be advantageous for it to generate suggestions automatically. The assistant would now be *proactive* and result in a mixed-initiative interaction [20]. In this paper, we attempt to design effective proactive chat-based assistants powered by large language models (LLMs) for programming.

Virtual assistants have been explored in many commercial products including Apple Siri and Amazon Alexa [33]; in fact, one of the most well-known and first proactive virtual assistants is Microsoft's Office Assistant, with the English version commonly referred to as Clippy. The Office Assistant would proactively offer its assistance based on inference of the user actions in their Word document. While the user experience with Clippy was less than ideal [7], more recent examples of proactive assistants in the form of autocompletion for text-based tasks have been more successful. For instance, GitHub Copilot [16] provides in-line code suggestions with LLMs. Copilot has been well received by programmers and randomized studies have shown increases in programmer productivity [41, 49]. However, autocomplete-based proactive assistance only considers local suggestions based on the user's cursor position. In contrast, a chat-based proactive assistant could suggest modifying the entire code or pointing out bugs in earlier parts of the code, in addition to being able to complete the code. Moreover, it is clear that the particular design details of a proactive assistant are crucial to its effectiveness.

We begin by outlining design considerations for designing chat-based proactive assistants with LLMs to address challenges of proactivity [9] by building on previous guidelines for human-AI interaction and mixed-initiative interaction [2, 20, 25]. Key considerations include supporting efficient evaluation and utilization of proactive suggestions as well as timing of suggestions based on context. The domain of programming presents a good opportunity to study proactivity given the success of LLM-based assistants for programming and a well-defined work context, which consists of the code file(s). This enables us to overcome the challenge of observing the human's workspace by integrating the assistant into the programmer's integrated development environment (IDE). Based on these design considerations, we build a proactive assistant for programming and integrate it into a chat interface inside a code editor, as shown in Figure 1. The chat interface operates like a standard interface where the user can type messages and receive responses; however, the assistant can now send proactive messages without a corresponding user request. The timing of the proactive suggestions depends on the human's recent activities in the editor and their interactions with the assistant responses. The proactive messages are distinguished from regular assistant messages, with the former only displaying a high-level description of the message unless the user chooses to expand the message. Finally, each assistant message can be integrated into the code via a "preview" button that highlights the changes the message will make to the code.

We perform a controlled user study where we evaluate programmer usage of our proactive assistant against a baseline ChatGPT-like assistant. We use the web-based code editor RealHumanEval [37] to experiment as it allows us to easily conduct studies. In the study, we include multiple conditions varying different aspects of the proactive assistant, including the timing of the suggestions and the ability to integrate suggestions into the programmer's code, to understand the effects of changing different design decisions. Our study demonstrates the benefits of proactivity in chat assistants for code as proactive assistants increase the number of tasks completed by 12-18%. However, our study also shows that proactive assistants need to be designed carefully as small changes can lead to differing user experiences—for example, increasing the frequency of suggestions can negatively impact user experience, reducing participant preference for the proactive assistant over baseline by half, despite productivity gains. Additionally, small changes in proactive assistant capabilities, including the ability to integrate suggestions, can significantly change user interaction and coding patterns.

In summary, this paper contributes:

- A proactive chat assistant system, comprising an interactive interface, suggestion content based on the user's code, and logic to appropriately time suggestions (Section 4). We also outline a set of design considerations when building proactive assistants (Section 3 and Section 7.1).
- A controlled user study characterizing the benefits of our proactive chat assistant on downstream user metrics (e.g., productivity and user experience), highlighting the potential for proactivity to be included in future coding assistants, as well as demonstrating the potential negative effects of changing certain proactive features (Section 6.1 and Section 6.2). We also analyze fine-grain user behavior when interacting with different proactive assistants (Section 6.3).

2 Related Work

2.1 The Ecosystem of Programming Assistants

A growing number of tools powered by large language models (LLMs)—i.e., programming assistants—are available to developers to generate or edit code and answer queries. Programmers are increasingly writing code with AI assistants like Github Copilot [16] and Cursor [13] and are using chat assistants like ChatGPT [39] or Claude [3] in place of online Q&A communities like Stack Overflow [52]. Programming assistants typically involve one of two types of support: autocomplete suggestions are used to quickly write more code based on the programmer's current code context, while chat dialogue can help answer questions that range from debugging errors to explaining documentation. Programming assistants surrounding code completion have been the focus of prior work, providing insight into how people use LLM-generated

code completions [6, 12, 35, 41, 43, 49] and improving completions by providing explanations [53] and determining when to best show [36, 50]. While there has been less focus on investigating the use of chat assistants for programming, it is important to note that they complement assistance provided by code completions [28] and are the focus of this work.

Prior studies investigating chat dialogue for programming assistance have always looked at the setting where people initiate questions to the assistant [10, 23, 24, 38, 44, 52]. In particular, Chopra et al. [10] found that people spent a significant amount of time constructing prompts and gathering and expressing their context to ChatGPT and did not enjoy being "slowed-down" through this process. Additionally, Nam et al. [38] noted that people often had a hard time finding a good prompt for the chat assistant that could give them the desired response. Further, Mozannar et al. [37] highlighted how the burden remains for programmers to appropriately incorporate an assistant response into their code. These observations highlight the hurdles to a streamlined integration of chat assistants like ChatGPT in a developer's day-to-day workflow: the need to provide sufficient context about the developer's problem and the need for developers to manually input and decide what they want to ask and then act upon the assistant response. Even while some programming assistants (e.g., Cursor [13], Github Copilot [16]) have begun building chat assistants into the integrated development environment (IDE), they still require human input. In this work, we evaluate whether incorporating an aspect of proactivity into chat assistants can lead to more benefits in terms of productivity and user experience. Our work aligns with existing tools for programming that provide chat support in that we still allow developers the option to ask questions when they so choose. Different from prior work, our proactive chat assistant also periodically recommends suggestions and even allows developers to integrate those suggestions into their code.

While the focus of this work is on designing proactive *chat-based* assistants, we overview other forms of proactive assistance in the existing programming literature. As mentioned in Section 1 and discussed extensively in Section 2.1, the most well-known and widely used form of proactive assistance is in-line code completions (e.g., GitHub Copilot [16]) [28]. Aside from proactive code completions, multiple related works have studied proactive assistance to address various issues including fixing error messages and understanding LLM generations. For example, a set of more traditional approaches explored the use of adaptive feedback—drawn from existing examples or code repair tools—to help students resolve error messages in submitted programs [1, 18, 19]. Recent approaches have explored live programming techniques to "preview" the outcomes of AI-generated code using projection boxes [14, 27]. Our work investigates whether LLM-powered proactive assistants can address multiple issues at once. We return to these studies and other aforementioned studies on chat-based assistance for programming to discuss how our findings inform the design of proactive programming assistants in Section 7.1.3.

2.2 Background on Proactive Assistants

Research on proactive assistants has appeared in many forms, ranging from physical robots [5, 42, 56] to virtual chat assistants on phones or computer applications [15, 17, 29, 32]. Given our target application of programming, we focus our discussion on chat assistants. Even within proactive chat assistants, the goals of prior proactive assistants vary depending on the context they are used. For example, support-based assistants use proactivity to provide motivation and continued dialogue [15, 32], while education-related assistants can proactively offer relevant support or explanations for students [51]. Most relevant to our domain are the set of information-finding assistants that leverage proactive messaging to provide additional and useful information [4, 21] and planning assistants can proactively help people reason about their decisions and identify overlooked alternative decisions or rationales [17]. The goal of our proactive assistant is to improve a programmer's productivity while maintaining a good user experience; it requires blending multiple goals of existing proactive assistants. In Section 3, we discuss further how these downstream metrics inform the design considerations for our proactive assistant.

Many prior deployed proactive assistants have failed or received significant negative reaction because the actual capabilities of these systems do not meet user expectations [33, 34], as many of the systems relied on pre-set messaging or simple models [15, 29]. With the advent of modern LLMs, and their growing usage as the backbone of agents [40, 54, 59], there is significant potential to revisit how we design more capable proactive assistants. In recent work, ComPeer [32] leveraged GPT-4 to build a proactive assistant for mental health support that handles and acts on a longer term memory of user dialogue. Aside from mental health applications, we believe LLMs can be particularly well suited for proactive assistants for productivity goals, particularly when they are already being used as regular chat assistants (e.g., ChatGPT). The increased model capability and ability to handle long contexts would be suitable for capturing complex environments in which they are completing tasks, which has been relatively unexplored in prior chat assistant literature relating to productivity [9]. In Section 4, we discuss the design of our proactive system which leverages off-the-shelf state-of-the-art LLMs to propose suggestions based on the user environment and even incorporate these suggestions into user code.

3 Design Considerations for Proactive Coding Assistants

To formulate design considerations for the core functionality and behavior of the proactive assistant, we revisit existing guidelines on designing mixed-initiative user interfaces and human-AI interactions [2, 20] as well as a survey of known challenges of designing proactive chat assistants [9]. We highlight five design considerations that merit further attention to capture the benefits and challenges of proactivity.

Harnessing the benefits of proactivity. Horvitz [20] emphasizes the importance of "developing significant value-added automation" when developing effective mixed-initiative systems. Given prior work highlighting the limitations of existing chat assistants for coding contexts [10, 37, 38] (as discussed in Section 2.1), the value-add of introducing proactivity would be an improvement in programmer productivity (e.g., in terms of time and effort) and in the user experience (e.g., reducing perceived "slow-down"), as

compared to a non-proactive chat assistant. Concretely, we propose the following two design considerations:

(1) Support efficient evaluation. Proactive suggestions should be provided in a way such that programmers can efficiently interact with them. This involves providing a manageable amount of information to the user about the suggestions so that they can quickly decide if they are worthwhile. This might involve hierarchically scaffolding the suggestions so that the programmer can obtain more information about the suggestion if needed.

(2) Support efficient utilization. If the user decides the suggestion is useful, the proactive assistant should make it easy for the programmer to utilize it. For example, the proactive assistant should make it as easy as possible for them to take action corresponding to the suggestion and incorporate it into the programmer's code. If the programmer decides the suggestion is not useful, they should be able to easily dismiss the assistant.

Handling challenges of proactivity. Prior works on proactive chat assistants have demonstrated that there is considerable nuance in developing a proactive chat assistant [9], as untimely and irrelevant proactive messages may compromise the success of the interaction timing [29, 46–48]. We incorporate these challenges in our design considerations:

(3) Show contextually relevant suggestions. Proactive chat assistants are increasingly built into complex environments where users often perform a number of different tasks. For example, programming tasks can range from implementation to debugging and testing, just to name a few, which all would require different interventions and information from the assistant. Proactive assistants should make sure to surface information that is relevant to the current context and recent user interactions and messages.

(4) Incorporating user feedback. The proactive assistant should adapt its suggestions and timing as it interacts with the user. The user should be able to accept and reject proactive suggestions, and the decision to accept or reject suggestions should influence future decisions of the proactive assistant. This is especially important as different programmers may want different levels of proactive help from the assistant based on their experience for instance.

(5) Time suggestions based on context. Programmers perform various tasks during their work and are often in a state of flow [11]. The proactive assistant should consider what the programmer is currently doing and their workflow before it provides a suggestion [6]. For example, showing suggestions while the user is actively coding can disrupt the programmer's flow and lead to disruptions. Horvitz [20] discusses the importance of fallback behavior which can allow for a reduced frequency of suggestions.

In the following section, we introduce an implementation of a proactive assistant for programming by building on the design considerations above.

4 Proactive Assistant Implementation

In this paper, we build a proactive assistant for programming tasks based on the design considerations in Section 3. Our proactive chat assistant distinguishes itself from online AI assistants such as ChatGPT by first having access to the user's work context and second by proactively proposing suggestions to the user versus only passively waiting for user requests. When embedding the proactive assistant into the user's work context, which is the programmer's IDE in this setting, the proactive assistant will have access to the current code in any of the user's files, the terminal outputs, and prior user queries to the AI assistant. To turn the context into suggestions, we discuss different components that comprise our proactive assistant and how we instantiate each design consideration, as shown in Figure 1:

- *Interactive interface*: In Section 4.1, we overview the interface through which the assistant presents the generated suggestions in a way that makes it easy for users to interact with, understand, and invoke when needed (Design Considerations 1-2).
- *Suggestion generation*: In Section 4.2, we discuss how the assistant takes the current context and generates a set of proactive suggestions that is timely and relevant (Design Consideration 3) which take into account user feedback (Design Consideration 4). In Section 4.3, we discuss how to appropriately determine when suggestions are shown to users (Design Consideration 5).

4.1 Interactive Interface

The interface of the proactive assistant, shown in Figure 2, is built into the standard chat interface, allowing programmers to use all normal chat functionality, while the assistant periodically provides suggestions to programmers. Furthermore, this means the proactive assistant does not take up additional real estate on the screen, particularly when programmers may already have multiple files open side-by-side.We describe the different features of the proactive suggestion and their respective design considerations:

4.1.1 Suggestion summary. Towards fulfilling design consideration 1, the proactive assistant facilitates efficient evaluation of the suggestion by providing a summary to allow users to quickly get an idea of whether or not the suggestion is relevant. The summary consists of a single sentence that starts with the type of the suggestion e.g., a bug fix, a new feature, and then a description of the suggestion, as shown in the headings of Figure 2 (A). The drop-down style of the interface allows users to easily expand and condense a suggestion and quickly browse the various suggestions.

4.1.2 Details of suggested implementation. If the summary of the suggestion seems relevant to the user, they can expand the suggestion for more details as shown in Figure 2 (A). Depending on the type of suggestion, they may receive a code snippet and/or a text description (e.g., a suggestion that explains a code snippet may not include code). The interface also allows users to copy the suggestion into the editor (design consideration 2). We limit the text description of the implementation to a few bullet points to facilitate efficient interaction (design consideration 1).

4.1.3 Preview suggested implementation in code. While users have the freedom to copy the suggestion code into the editor and decide how they want to use the suggestion, the proactive assistant also has a "preview" functionality, which will allow users to see how the assistant would incorporate it into their code (design consideration 2). The suggested implementation is computed via another call to the LLM, where it is given the proactive suggestion and the user's code and produces a new code that incorporates the suggestion.

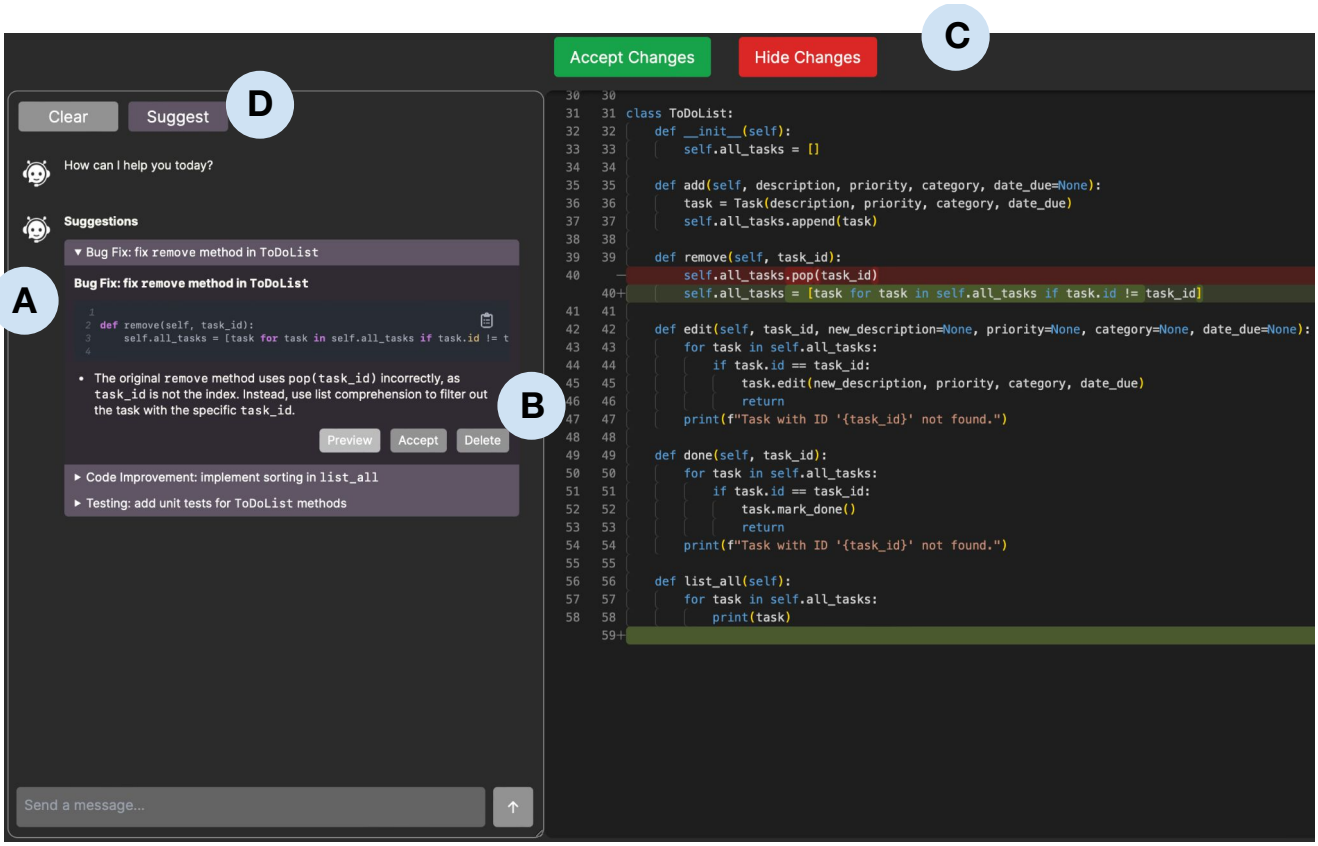

Figure 2: A walkthrough of the proactive assistant interface for coding. (A) Overview of suggestions, where the assistant provides a short description, and users can choose to expand a suggestion for more details (this can involve implementation and a brief explanation); (B) Buttons that allow participants to preview the implementation in their code, accept a suggestion to ask follow-up questions on, or delete the suggestion; (C) Integrating a suggestion into the editor via a diff format, where users can decide if they want to accept or hide the changes; (D) Invoking suggestions, where users can also request suggestions from the assistant (for example, if they did not like the original suggestion or wanted a suggestion before the assistant provided one).

Each time a suggestion is previewed requires another LLM inference call. As such, we only preview a suggestion when invoked by the user, rather than automatically previewing *every* suggestion, which helps to reduce the latency that users experience. This new code file generated is then shown to the user via a diff editor based on the Monaco Editor [1] which tells the user what lines were added or removed once the suggestion was incorporated, as shown in Figure 2 (C). The user has the option to accept or revert all changes and can additionally only a subset of the changes and edit freely before clicking on the "accept changes" button. In the Supplementary Material, we include the prompt used to incorporate edits into the current code snippet.

4.1.4 Asking follow-up questions based on suggestion. Since the proactive assistant is a part of the chat interface, we create the option for users to add a proactive suggestion to the chat history (via the "accept" button) or remove it (via the "delete" button). Clicking on the accept button shown in Figure 2 (B) would add the expanded suggestion to the chat message in a way that looks visually similar to the rest of the user's messages. This allows future proactive suggestions to condition on the user's accept or delete actions addressing design consideration 4 (incorporating user feedback). This also allows the user to conveniently ask follow-up questions on the proactive suggestion. This may be helpful as it is common for users to follow up on LLM responses to clarify or correct the model.

4.1.5 Manually requesting suggestions. While the nature of a proactive assistant suggests that the assistant will be the one who determines when to provide suggestions to the user, it is unrealistic to expect an assistant always to anticipate when users might want suggestions (design consideration 5). As such, we include a button, as shown in Figure 2 (D), allowing users to request suggestions as they wish. This can also allow us in future work to better learn when the proactive assistant should intervene.

[1] https://microsoft.github.io/monaco-editor/

Suggestion Type	Example of developer question

Explaining existing code	What does this do?
	`model = GPTLanguageModel()`
	`m = model.to(device)`

Brainstorming new functionality	Based on the following OCaml code for an s-expression evaluator, write a parser for the tokens defined. Note that all operations must be binary and bracketed, with no concept of operator precedence. `[code context]`
Completing unfinished code	The following is some C code for binding a "hello" C function to an Io method. Can you complete the code by the comment: `[code context]`
Pointers to documentation	I've saved this in 'scripts/align-import.py'. Remind me how to use 'find' to run it on every '*.lean' file in a subdirectory?
Debugging (Latent errors)	Why is my redirect not working? Here is my client side code `[code context]` Everything else works as intended, except that it will not redirect. What is the issue here?

Debugging (Runtime errors)	`[code context]`
	Test suite failed to run
	Cannot find module 'libp2p' from 'src/shared/libp2p_node/index.ts'
	`[remainder of error trace]`
	Do you have any idea why I am getting this error?

Adding unit tests	Could you create Jest unit tests for this function? `[code context]`
Improving efficiency and modularity	The following is a kernel of an algorithm. It uses Apple's metal api for matrix operation. I think it can be improved to make it run faster. `[code context]`

Table 1: Suggestion types informed by dataset analysis. We identify categories of suggestions that a proactive assistant can surface by coding developer questions to ChatGPT in the DevGPT [52] dataset. We incorporate these suggestion types into the design of our proactive assistant.

4.2 Suggestion Content

For the proactive assistant to be useful, it must generate appropriate suggestion content (design consideration 3). This means that the assistant should have knowledge of the types of suggestions that users benefit from and that the assistant should surface the right type of suggestion at the right time, depending on the user's code context.

To identify the different types of suggestions that users benefit from for coding contexts, we look to DevGPT [52], a dataset of conversations between programmers and ChatGPT, as a source of real-world questions. While other chat datasets also contain in-the-wild programming-related questions (e.g., WildChat [57], LMSYS-Chat-1M [58]), we selected DevGPT in particular because the majority or all questions contain user code context in addition to the user question and LLM response, which allows us to manually inspect whether we think it would be reasonable for a proactive assistant to anticipate this question from the current code context. We sample 100 questions and their code contexts from the DevGPT data and manually create a label for the "category" of the question. We then cluster labels to identify common themes across questions. The goal of surveying existing questions is to ensure we obtain a reasonable set of potential suggestion types to prompt the proactive assistant, however, this set may not be exhaustive.

From this annotation process, we identify eight categories of questions that proactive assistants can help with: *explaining existing code*, *brainstorming new ideas or functionality*, *completing unfinished code*, *providing pointers to syntax hints or external documentation*, *identifying and fixing bugs* (which include both latent and runtime errors), *adding unit tests*, and *improving code efficiency and modularity*. Annotations were roughly distributed across all categories where each appeared 15%, 19%, 18%, 13%, 10%, 9%, 6%, 10% of the time respectively—note that since this dataset consists of only ChatGPT conversations and may not represent the actual distribution of user questions in practice.In Table 1, we include example user questions that fall under each category.

4.3 Timing of Suggestions

The proactive assistant must display suggestions at the appropriate time so they are not obstructive and distracting to users (design consideration 5). A study by Barke et al. [6] characterized that interaction with programming assistants consists of two modes:

acceleration, where the assistant is primarily used to help the programmer implement what they already intend to write; and exploration, where the programming assistant assists a programmer in identifying and planning out goals. We propose the following conditions for when proactive suggestions should be shown to users based on an estimation of when the user is in acceleration or exploration mode:

- *During acceleration:* Suggestions are not requested when the user is interacting with a previous suggestion, typing in the chat, sending a message, or waiting for a response. The suggestion timer resumes 5 seconds after the user stops typing to account for a typing break. If the user would like a suggestion, they can still request it.
- *During exploration:* When the user is idle—likely due to planning and brainstorming, the assistant provides suggestions after 5 seconds, limited to every 20 seconds since the last suggestion, interaction, or chat. If the user starts coding before suggestions are shown, due to potential latency in the model query, we do not display the suggestions.

To detect mode based on user interactions, we implement listeners and corresponding timers in the editor to track whether the user is actively taking actions (e.g., writing code or chat messages). Based on pilot studies with 4 participants, we determined that 5 seconds was a reasonable estimates of how long people tend to pause when coding and 20 seconds was a reasonable guess at how much time was needed to check out the proactive suggestions.

Additionally, we allow the proactive suggestion to be shown *during debugging* when users are trying to edit the current code for any bugs or improve the performance. Debugging is distinct from the previous two states because the user is neither trying to write more code nor planning the next steps. The proactive assistant immediately provides a suggestion when the user runs or submits code that leads to an error because the user is not actively coding.

4.4 Generating Suggestions

To generate a set of suggestions, we discuss how the assistant makes a call to the LLM with the current user context to generate a set of suggestions. In this work, all LLM calls are made to GPT-4o [39], a state-of-the-art LLM. In pilot studies, we experimented with smaller, faster models (e.g., GPT-4o-mini) but observed that the quality (e.g., correctness of suggested code, relevance of suggestions) of suggestions to be lower. We overview what inputs from the user context are used to generate each type of suggestion and include the full prompts in the Supplementary Material.

Standard suggestions. Standard suggestions are the ones that periodically show up while the user is writing code. To generate these suggestions, the LLM call includes a list of items: any prior chat messages, a system prompt, and a general prompt. We include the prior messages to avoid generating suggestions that may repeat previously asked questions and to make the model aware of prior user-assistant interactions. Then, we scaffold the suggestion types identified in Section 4.2 into the system prompt to encourage the model to consider a more diverse set of suggestions. Finally, we include the current code and instructions for how to return the set of generated suggestions in the main prompt. Figure 3 (left) shows how all inputs are combined to generate standard suggestions.

Debugging suggestions. Debugging suggestions are the suggestions that are only triggered when the user runs their code. The LLM call again includes prior messages, but instead of the system prompt, we simply include all other information in one main prompt. This main prompt includes the current code, a short instruction to ask the model to focus on any error outputs, error statements from the terminal, and instructions for how to return the set of generated suggestions. Note that debugging suggestions have a more restricted set of suggestions that the assistant can generate. Figure 3 (right) shows how all inputs are combined to generate debugging suggestions.

The response from the LLM will then be parsed into individual suggestions and displayed in the exact order in which they are returned in the interactive interface as described in Section 4.1. While we do not explicitly incorporate an additional mechanism to rank suggestions, we verify based through pilot tests that the order of suggestion types is not always the same.

5 Study Design

We conducted a controlled user study to understand the effectiveness of a proactive assistant on productivity and investigate how people interact with the assistant. Our research questions are as follows:

RQ1: *What is the effect of the proactive assistant on user productivity?* The goal of a proactive chat assistant is to anticipate potential user needs or even surface questions the user has not considered. As such, we expect the assistant to have a measurable downstream effect on productivity measures.

RQ2: *What is the effect of the proactive assistant on user experience?* In addition to productivity measures, we hope the user has a favorable experience when interacting with proactive assistants.

RQ3: *How do participants interact with different proactive assistants?* A proactive assistant comprises several design considerations, we analyze how users interact with different variants of our proactive assistant to understand the effect of these design decisions on user behavior.

5.1 Participants

We recruited a total of 65 students via university mailing lists. The inclusion criteria for the study are that they must be based in the U.S., be older than 18 years old, and have experience programming. Since we design tasks that are written in Python, we require participants to have a baseline level of Python knowledge. Among our participants, 60% reported their Python knowledge as intermediate, 31% had advanced Python experience, and 9% were beginners. Moreover, 19% self-report that they are daily users of AI tools for programming (including GitHub Copilot and ChatGPT), 46% report that they use these tools at least once if not multiple times a week, and 35% rarely use AI tools for programming. The gender breakdown of our participants was 34% female and 66% male. The student population was distributed between 38% undergraduate and 62% graduate students, skewing towards more experienced programmers.

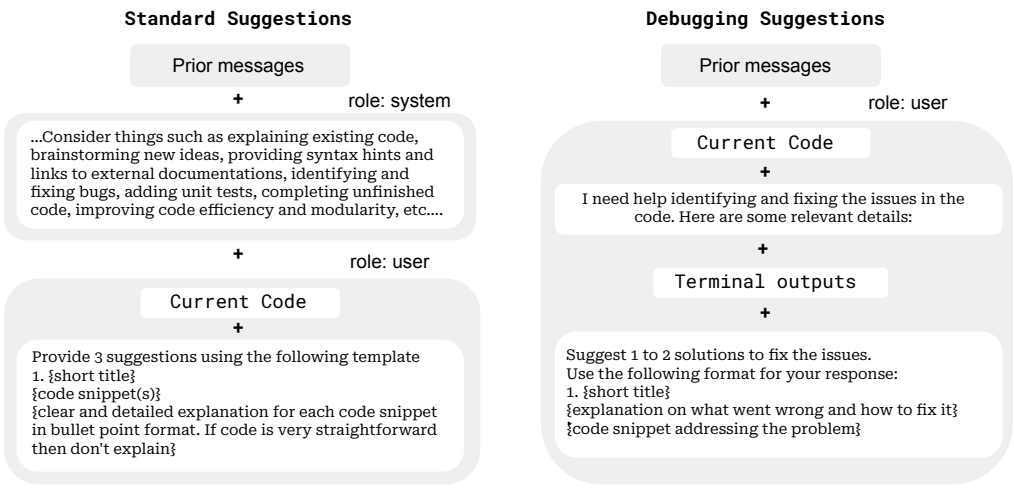

Figure 3: Prompting the assistant for suggestions. An overview of how different inputs (e.g., prior messages, current code, and terminal outputs) are incorporated in the generation of standard and debugging suggestions, the former which is shown generally and the latter which is only triggered when code is run. Full prompts for both types of suggestions are provided in the Supplementary Material.

5.2 Baseline

Our study compares proactive assistants to a non-proactive chat-based assistant baseline. Every participant in our study interacts with a standard chat assistant that is incorporated into the coding environment but does not have access to the user's code. This is similar to the set-up that has been studied in prior papers [24, 37, 52] and mimics how people would use ChatGPT. The interaction paradigm for the baseline is as follows: the user needs to type in their query to obtain a response from the chat assistant. The chat assistant can respond in natural language and provide code snippets.

In the baseline (as well as our experimental conditions), we do not include any code autocomplete tools to not introduce a potential confounding factor in our analysis as not all participants are experienced in using autocomplete tools. Further, we believe the proactive assistant can complement code autocompletion tools, which consist of local proactive changes, while the proactive assistant provides suggestions at a more global level.

5.3 Experimental Conditions

We consider three experimental conditions to explore the effect of different proactive chat assistant designs:

Condition 1: Our proactive chat assistant (`Suggest and Preview`*):* In this condition, the participant interacts with the proactive assistant described in Section 4. This condition encompasses all features of the baseline condition, i.e., the participant still has the option to send chat messages to the assistant at any point while also receiving proactive suggestions.

Condition 2: Our proactive chat assistant without preview feature (`Suggest`*):* In this condition, the participant interacts with the proactive assistant described in Section 4, without the preview feature, which allows the assistant to suggest direct edits to the user's code. Similar to Condition 1, the participant can still send chat messages to the assistant at any point while also receiving proactive suggestions.

Condition 3: Persistent proactive chat assistant (`Persistent Suggest`*):* In this condition, the participant interacts with a variant of our proactive chat assistant. While the proactive assistant in this condition looks at face value to be the same as the proactive assistant in Condition 2 (`Suggest`), we modified a few different parameters discussed in Section 4: reducing the amount of time that an assistant waits to provide a suggestion (from 20 seconds to 5 seconds), increasing the number of suggestions shown (from 3 to 5), and removing guiding prompts.

The three conditions allow us to investigate whether (1) the level of proactivity affects user productivity and user experience (comparing `Suggest and Preview` and `Suggest`) and whether (2) small changes in the design of a proactive assistant can make a sizeable difference in user productivity and experience (comparing `Suggest` and `Persistent Suggest`). Recall we fix the LLM backbone across all conditions to be GPT-4o [39], a state-of-the-art LLM.

5.4 Experimental Platform

The study was conducted online and asynchronously at the participant's own time. The interface we used to deploy the study is an extension of the open-source platform RealHumanEval introduced in [37]. Conducting the study on the web allows for easy assignment of participants to experimental conditions, the duration of the study, and the tasks participants solve. Finally, the web interface reduces installation issues or any potential incompatibility. While the web interface is not the traditional IDE that a user may typically use for their day-to-day development, it is reminiscent of online coding platforms (e.g., Leetcode) that participants are generally familiar

with. The editor in our web interface is also the same Monaco editor as in Visual Studio Code, a popular IDE. We also added a check in the post-study form to make sure that participants did not have any issues using the interface.

5.5 Procedure

Before participating in the study, each participant filled out a consent form. The study has been approved by our institution's review board (IRB). We adopt a between-subject setup to compare the three proactive chat variants and a within-subjects setup to compare each proactive variant to the non-proactive baseline. This means each participant will interact with both a proactive assistant and the baseline chat condition. We randomize the order in which participants interact with either a proactive assistant or the baseline chat assistant. The total amount of time the participant spent coding for the study was 40 minutes, with participants spending 20 minutes in each condition.

5.5.1 Onboarding. Before participants can access the proactive assistant, they are provided with a short tutorial that describes how to interact with the proactive suggestions. The tutorial highlights how the proactive assistant will provide a high-level summary of suggestions that can be expanded upon and how to ask follow-up questions. The tutorial also warns participants that the suggested implementations may not always be correct and to carefully verify the suggestions. In the Appendix, we provide a copy of the onboarding instructions that participants had access to.

5.5.2 Coding Tasks. In our study, participants engage in 20-minute sessions focusing on two types of programming problems designed to reflect real-world engineering scenarios. The first type of problem aims to test system-building skills. For instance, participants are tasked with enhancing an "online store" class object implementation. Given an initial starter code, they must implement additional features specified through natural language instructions, brainstorm a new feature, debug potential issues, and write tests for these new features. The second type of problem challenges participants to work with unfamiliar packages, simulating situations where engineers must quickly adapt to new tools. An example of this is manipulating a dataframe using various NumPy functions. Unlike previous studies that primarily assessed puzzle-solving abilities through Leetcode-style problems [37, 49], our approach aims to evaluate skills more directly relevant to practical software engineering tasks. In the Appendix, we provide task descriptions for each of the tasks used in the study.

The problem types are randomized across conditions and in both proactive/baseline conditions to avoid confounding task difficulty or type with the helpfulness of proactive assistants. Since each participant experienced both the baseline and proactive conditions, we distributed the 4 tasks in a symmetrical manner. This means that if the participant started with a system-building task, they would see the other system-building task when the proactive assistant was switched on or off. This scheme also ensures a relatively similar number of task types in both proactive and baseline conditions. Participants decide when they are satisfied with their current attempt on the given task and would like to move on to another question. As such, we had a second question of the other task type queued up for the participant to work on for the remainder of the time within that condition. In sum, each participant attempted two tasks of the same type across the baseline and proactive conditions, and *additionally* one or two of the remaining tasks of the different task type when time permitted.

5.5.3 Post-task survey. The study concluded with a post-task survey that asked participants about their experience with both the proactive and baseline conditions. While we log telemetry data in both conditions to measure user productivity, we use the post-task survey to measure user perceptions. In the Appendix, we provide a list of the post-task survey questions.

5.6 Measurements

To answer our research questions, we measured the following:

(1) *Telemetry.* Our interface logs all user behavior from coding to using the chat assistant: anytime the code is updated by the user, the interface saves the updated code. Further, any chat messages and associated responses are logged. Finally, any interactions with the proactive assistant, including expanding suggestions, accepting a suggestion, and requesting suggestions are all recorded. Using the collected telemetry data, we compute the the number of sub-tasks completed. We can also obtain fine-grained interaction behavior to identify common interaction patterns.
(2) *Perceived usefulness.* In the post-task survey, we ask participants to rate their interactions with both the proactive and baseline chat assistants. Participants were then asked to explain their ratings.
(3) *Open-ended feedback.* In the post-task survey, we also asked additional questions that further dive into what aspects of the proactive assistant participants preferred and what could be improved.

5.7 Analysis Approach

To measure the effect of proactive assistants on user productivity (**RQ1**), we computed the number of sub-tasks completed. We build a linear model which incorporates the experimental condition, which is either baseline or one of the proactive models, and the coding problem as fixed effects. We also qualitatively inspect whether the code written contains automated test cases. To understand the effect of proactive assistants on user experience (**RQ2**), we measure participant ratings of the baseline and proactive assistant. We then computed a binary measure of whether participants preferred the proactive assistant to the baseline and ran an ANOVA with Tukey HSD. We also analyze participant justifications to interpret the quantitative results. For all tests, the threshold for statistical significance was $\alpha = 0.05$. To begin to understand how participants use the different proactive assistants (**RQ3**), we measure the frequency of interactions with the various components of the proactive assistants as logged in the telemetry. We consider this portion of the results an exploratory analysis to identify trends that distinguish proactive conditions. We present the results of our study in the following section.

6 Results

6.1 RQ1: Proactive Assistants Generally Improve User Productivity

Number of sub-tasks completed. When comparing the number of sub-tasks completed by participants, we find that on average participants with proactive assistants are more productive with a baseline chat assistant: we observe a 12.1%±5.1%, 18%±5.8%, and 11.6%±5.0% increase in the percentage of test cases passed for `Suggest and Preview`, `Suggest`, and `Persistent Suggest` respectively compared to baseline. Figure 4 provides a more granular view by task. We find that the improvements in the number of test cases are significant across *all* proactive variants, where $p = 0.01$ for `Suggest and Preview`, $p = 0.002$ for `Suggest`: $p = 0.002$, and $p = 0.02$ for `Persistent Suggest`. We explore whether different groups of participants benefit from proactive suggestions differently (Appendix C): we do not observe any significant differences based on Python expertise or AI tool usage frequency but do observe a significant difference based on gender (i.e., women tend to benefit more from proactive assistance).

Number of test cases written. In addition to the increase in the number of test cases passed, we also observe an increase in the amount and quality of test cases written. Note that writing tests was part of the instructions provided to users. Focusing on Task Type B, where it is important to test how different components of the designed system work together, we annotate user code for whether they incorporated test cases and find that only 13.3% do in the baseline setting while 33.3% do in the proactive settings. We believe that providing proactive suggestions can help create a virtuous cycle where including more test cases helps participants surface issues in their code, thus also increasing the number of test cases passed.

6.2 RQ2: User Experience with Proactive Assistants Varies by Implementation.

User preferences across proactive conditions. While we observe generally uniform benefits of proactive assistants across the different conditions in terms of user productivity, we see more variation in terms of user experience across the conditions. As shown in Figure 5, we measure whether participants preferred the proactive assistant over the baseline non-proactive assistant. We find that the vast majority of participants prefer the proactive variant over the baseline for both the `Suggest` and `Suggest and Preview` proactive assistants (90% and 80% of participants respectively). In contrast, less than half of participants (47%) in the `Persistent Suggest` proactive condition preferred having the proactive assistant. We find that both `Suggest` and `Suggest and Preview` variants are statistically different than the `Persistent Suggest` variant ($p = 0.005$ and $p = 0.03$ respectively). On average, participants had a neutral view of the baseline assistant in terms of usefulness. As such, participants viewed both the `Suggest` and `Suggest and Preview` variants on average as net beneficial.

Comparing participant responses across proactive variants. Participants in the `Persistent Suggest` condition tended to not prefer the proactive assistant because they often found the assistant to be *"distracting"* and *"annoying"*. One participant noted that *"the non-proactive chat assistant was best because it didn't interrupt what I was doing."* and another participant *"found it annoying because it distracted [them] from working with the non-proactive answers."* In contrast, participants in the `Suggest` and `Suggest and Preview` conditions tended to have a positive view of the suggestions provided by the proactive assistant compared to the baseline. One participant noted that they preferred the *"non-proactive was pretty useless because it didn't have any of the context regarding what I was trying to do or the task; I could only really use it for pure syntax like what I would usually search StackOverflow for. Proactive was better because it had more context."* and another stated that *"I really wasn't sure what to ask for with the non-proactive chat"* since the baseline chat *"required manual input to generate advice"*.

Further, we identified two reasons for why participants did not prefer proactive suggestions across proactive conditions. The first reason was concerns about irrelevant suggestions, where participants thought *"it was hard to clearly understand what their use cases were"* and *"they did not seem useful to the particular tasks"*. Interestingly, participants were more concerned about relevancy, and did not mention the correctness at all, potentially due to the limited scope of tasks considered in the study. The second reason was that participants were generally unfamiliar with a proactive assistant compared to the baseline chat, saying *"I was already familiar with how to code with chatbot assistants."*. Further, they already knew how to use the baseline chat well as *"it feels similar to gpt-4o and copilot"* and they knew what scope of tasks the chat was helpful for (e.g., *"very small and simple cases"*). Next, we analyze how user perception translates into the actual usage of the proactive assistant.

6.3 RQ3: How Do Participants Use Different Proactive Assistants?

To better understand the effect of different design decisions on how participants use proactive assistants, we break down user interactions with proactive suggestions by condition. First, we compare `Persistent Suggest` and `Suggest` conditions to explore the effect of suggestion timing. Second, we compare `Suggest` and `Suggest and Preview` conditions to explore the effect of the level of proactivity. Finally, we investigate what kinds of suggestions participants tend to accept or reject. The following discussion should be viewed as an exploratory analysis; a later addition of statistical tests via Student t-tests did not show significant differences between conditions.

6.3.1 The effect of suggestion timing. Decreasing the amount of time that the proactive assistant waits to provide suggestions means that far more suggestions were shown in the `Persistent Suggest` condition than in the `Suggest` condition. As such, we observe that participants tended to expand more suggestions in the `Persistent Suggest` condition compared to the `Suggest` condition (9.5 ± 2.4 times per task versus 6.4 ± 2.5 times). However, proactive suggestions were copied more often in the `Suggest` condition compared to `Persistent Suggest` (3.1 ± 1.5 as compared to 2.1 ± 0.9 times). Figure 6 shows example telemetries of participants in each condition that demonstrate this behavior. These trends suggest that when a proactive assistant provides too many suggestions, participants can be easily distracted and thus still want to view the suggestions at a

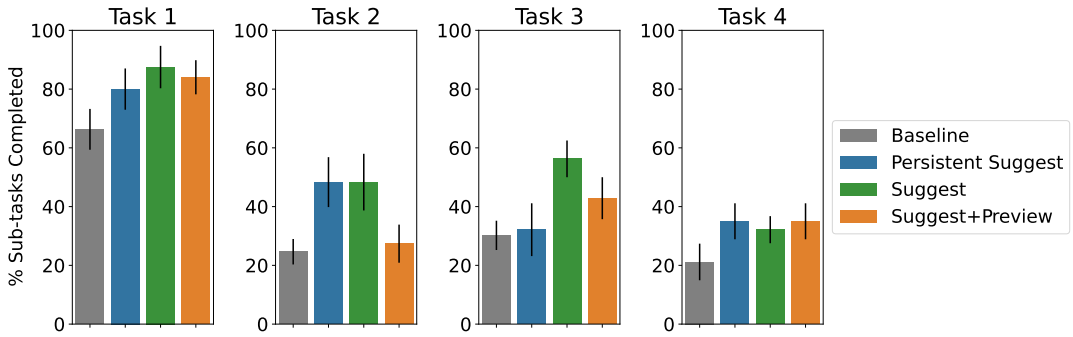

Figure 4: Percentage of sub-tasks completed correctly. Comparing baseline chat to proactive assistants across the four tasks, where Task 1 and Task 2 are system-building questions and Task 3 and 4 are ones where participants work with new packages and functionality. We report average performance and standard error. While performance varied by task, we observed that all variants of proactive assistants tended to increase the number of test cases passed compared to the baseline chat assistant across the board.

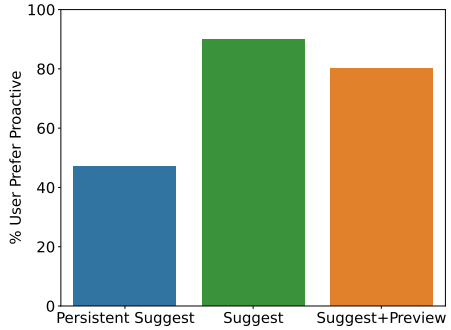

Figure 5: Comparing participant perception of proactive versus baseline. User experience is important to the adoption of proactive assistants. We compare how often participants preferred a proactive assistant to the baseline variant in each condition and find that participants generally preferred the proactive variant only in the `Suggest` and `Suggest and Preview` conditions.

similar frequency. However, when the suggestions are provided too often, participants are less likely to make use of the suggestions.

6.3.2 The effect of level of proactive assistant involvement. The ability to preview and incorporate suggestions into code led to a change in user interactions with proactive assistants. For example, after expanding a suggestion, further interactions with suggestions were more than twice as likely to be through the preview functionality rather than copying code and integrating it into the editor by themselves (3.1 ± 1.5 in the `Suggest and Preview` condition compared to 0.8 ± 0.5 `Suggest` condition)—an example of such an interaction pattern is shown in Figure 6. This may also have led to an increase in manual requests for suggestions, which happened more often in the `Suggest and Preview` condition, where participants made on average 2.9 ± 0.9 manual requests in the `Suggest and Preview` compared to an average 1.7 ± 0.7 requests in the `Suggest` condition. These results suggest that participants may gravitate towards options that reduce their manual efforts to make changes, which include utilizing the preview functionality.

6.3.3 What kinds of suggestions do participants tend to accept? While most proactive interactions fall under expanding, copying, or previewing suggestions, as discussed in Section 6.3.1 and 6.3.2, there were a handful of interactions where participants accepted or rejected suggestions. We identified a total of 75 interactions where 69 were accepts and 6 rejects. The most commonly types of accepted suggestions were those on *brainstorming new functionality* and *debugging (latent errors)*, with 18 occurrences each, while the least likely accepted suggestions were *explaining existing code* and *pointers to documentation*, with only 1 occurrence each. These trends suggest that participants are likely to accept more "actionable" suggestions, rather than ones that may be merely informative. Interestingly, *improving efficiency and modularity* suggestions were most commonly rejected suggestion types (with 5 out of 6 occurrences and the other being a *completing unfinished code* suggestion). This behavior may have been influenced by the particular study design which emphasized code correctness rather than optimized or well-written code, suggesting that different suggestions may be preferred by programmer's with different goals. Full results are provided in Appendix C.

7 Discussion

7.1 Design Implications For Proactive Coding Assistants

7.1.1 Revisiting design considerations. Given the findings from our experimental study, we now revisit whether the design considerations we posed in Section 3 are useful guides for designing proactive coding assistants. We map qualitative feedback from participants' post-task survey responses to the five design considerations.

(1) Support efficient evaluation. We observe that participants generally felt that the proactive assistant provided suggestions that had *"clear answers while giving specific examples"* and allowed them to quickly *"get a baseline understanding of the problem."*, which provides evidence that the suggestions were presented in a way that

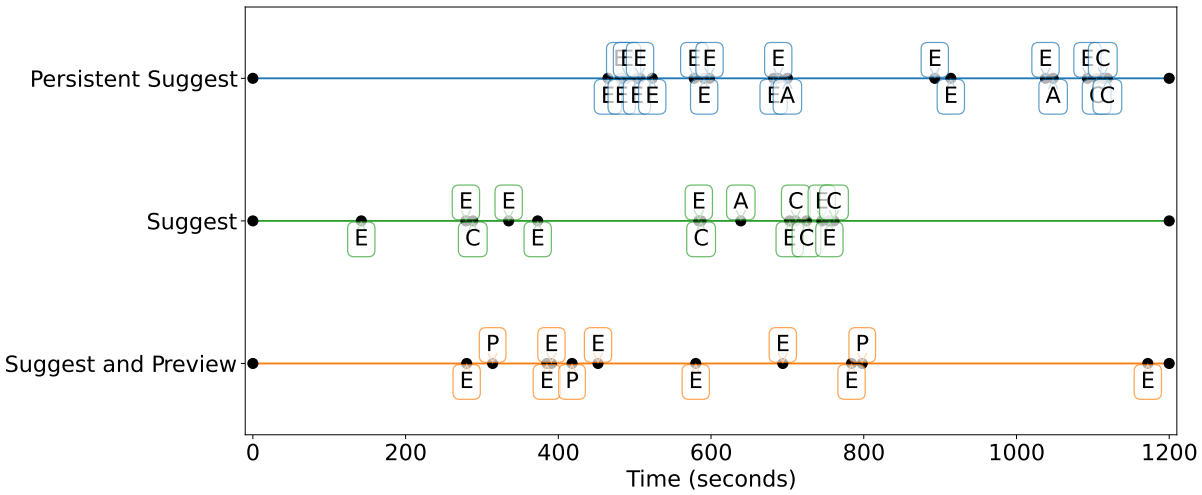

Figure 6: Comparing interactions with different proactive conditions. Visualizing sampled participant trajectories from coding with different proactive assistants. We denote "expanding" a suggestion with E, "accepting" a suggestion with A, and "copying" a suggestion with C—all of which are available in every condition, and "previewing" a suggestion with P, which is only available in the `Suggest and Preview` condition. In the `Persistent Suggestion` condition, participants tend to expand suggestions without utilizing them further. In contrast, participants tend to copy expanded suggestions in the `Suggest` condition while participants in the `Suggest and Preview` condition tend to preview suggestions.

allowed participants to efficiently understand them and determine whether to use them. Further, the suggestions were not too distracting; another participant noted that the suggestions helped to *"pinpoint specific issues while not being so obnoxious [they] couldn't ignore it to focus on other tasks."*

(2) Support efficient utilization. While both the baseline non-proactive and proactive assistants can help *"find errors in my code and write functions"*, a participant noted that they are *"able to work faster while using the one with suggestions."* In particular, many participants noted that proactive suggestions were *"much easier and faster to just click on a roughly correct suggestion instead of typing out specifically what I wanted to know"* and *"a lot of the suggestions can be directly pasted into the code."*. Further, proactive suggestions that can be directly incorporated into user code via the preview function are *"immediately executable"*, and participants felt that it improved their general workflow efficiency.

(3) Show contextually relevant information. Not only did participants believe that the proactive assistant often suggested relevant information (e.g., *"it gave me suggestions which were related to what I wanted"*), but the assistant also often *"suggested things I didn't consider and didn't know to ask for such as refactoring code and spelling errors"*. Identifying errors seemed to be a common feature that participants liked, as *"it took care of most of the errors and allowed me to focus mainly on writing my code"*. Overall, participants felt the proactive suggestions were *"more personalized"* and *"more tailored"* to their needs. In particular, a participant contrasted the proactive chat assistant with other programming assistants such as code completion tools as *"[code completion] only fills the code after my cursor and don't provide insights on the already implemented code."* This reinforces the idea that proactive chat assistants can complement commonly used tools like GitHub Copilot.

(4) Incorporating user feedback. Interestingly, this was the only design consideration that received mixed feedback from participants. Some participants believed *"the ability to accept or reject the code it suggested was really nice"* and *"intuitive"* to use while others felt like they were *"unnecessary"* and *"interacting with them distracts me from the task."* Future work may consider incorporating user feedback implicitly rather than requiring users to accept or delete suggestions. Related to our findings from Section 6.3.3, different kinds of suggestions may be preferred at different points of a programmer's workflow.

(5) Time services based on context. Aside from the `Persistent Suggest` condition where participants felt suggestions popped up too often (as discussed in Section 6.2), participants had positive perceptions of the proactive suggestion timings, noting that the proactive assistant was *"always ready"* and comparing the proactive assistant to *"a second set of eyes"* or *"a second person solving [the problem] for me by my side."* Additionally, multiple participants suggested that the timing of suggestions can even enable them to *"catch issues early"* or *"fix errors early on so that they don't require refactoring the entire codebase later"*.

7.1.2 Expanding design considerations. We also asked participants how the proactive assistant can be improved and synthesized these responses to identify three new design considerations. We do not include features that were already accounted for in the different variants (e.g., participants in the `Persistent Suggest` condition wanted less frequent suggestions, and participants in the `Suggest` condition wanted to be able to incorporate suggestions into the code). These expanded design considerations present opportunities for future work to improve proactive coding assistants.

(6) Allow users to decide when they want proactive assistance. Multiple participants across conditions recommended that they would like the option for when to have the proactive assistant on (e.g., *"an opt-in feature"* or *"could change the frequency"*), particularly because *"sometimes I found it useful and sometimes I didn't"*. Since the goal of our study was to investigate the usage patterns and effects of a proactive assistant, providing the user the option to turn off the proactive assistant would not have been possible. This may be a particularly necessary feature in certain contexts where the user is working on less standard coding tasks, the user may choose to turn off proactive suggestions. Prior evaluations of LLMs have shown that models are more accurate or hallucinate less on tasks akin to those that have appeared in the training data [22, 31]. This feature could also be important for long-term usage and facilitating appropriate trust and reliance on AI assistance .For example, a participant was concerned about potential over-reliance on proactive suggestions, which *"can lead to a lack of understanding of a codebase. Therefore, I don't want constant suggestions."*

(7) Incorporate additional context when generating suggestions beyond user code. While many participants already appreciated the benefits of proactive suggestions, they had further ideas for improving the suggestion content by incorporating other aspects of user behavior. Incorporating additional context can not only better customize suggestions to user needs, but may also be beneficial in settings where the user is working on long context tasks.For example, participants suggested that the proactive assistant could take into account *"the functions I was using"*, *"what the user is focusing on, typing, or clicking,"* or *"recently updated code regions."* Another option suggested to incorporate further context would be to allow users to provide a natural language of their task—i.e., *"if I can enter the task description somewhere as a constant part of the prompt [as] some suggestions have a slight misunderstanding of the problem"*.

(8) Flexibly varying the amount of information shown. Since the current proactive assistant provides one implementation option and provides a fixed number of suggestions each time, participants noted the possibility of scaling the amount of information up and down. For example, the assistant could provide more implementation options in the proactive suggestions, e.g., *"multiple ways of solving a problem,"* particularly when different factors may matter in different contexts (e.g., runtime). However, there may also be times when the assistant could scale down the number of suggestions shown (e.g., *"if everything was good, then tell me that everything is good."*)

7.1.3 Comparison of findings to prior work. Finally, we discuss our findings in the context of related, prior literature on AI-assisted programming and reflect on how our results should implicate the design of future AI coding assistants. In particular, we focus on related work of two categories: findings on chat-based programming assistants and findings on other forms of proactive programming assistants.

Implications on chat-based programming assistants. In a prior study on chat assistance, Ross et al. [44] found that the "acceptance rate" of non-proactive chat requests, as measured by code copies, was higher than the acceptance rate of proactively generated code completions. However, our findings suggest that proactive chat assistants can generally increase user productivity. While these results may appear to be at odds, that is not necessarily the case as proactive suggestions can have a lower acceptance rate but still help users indirectly. Relatedly, Nam et al. [38] and Mozannar et al. [37] both highlighted issues with and the burden of appropriately prompting the chat assistant to receive helpful responses. Our study provides evidence that a proactive assistant can reduce such a burden with proactive suggestions surfacing ideas that the user may not have thought of. In sum, our findings suggest that chat-based assistants may benefit from incorporating *some* aspects of proactivity.

Implications on proactive programming assistants. Traditional approaches to providing proactive feedback relied on collecting relevant training examples and thus were largely limited in scope in terms of the types of feedback that the system could provide [18, 19]. In contrast, we find that with modern LLMs, our proactive assistant could provide a diverse set of suggestions that were utilized to varying degrees by participants at different points of solving the programming task. This suggests the potential for leveraging current and even more capable models in the future as an engine for surfacing high-quality suggestions. Ferdowsi et al. [14] showed previewing AI suggestions and their impacts on existing code facilitates user validation of these suggestions. While we do observe benefits from the `Suggest and Preview` condition, participants saw the biggest benefit from the `Suggest` condition where there was no preview functionality. A possible reason for this may be due to the way we present previews of code edits to users in our interface, which may not be optimal for user validation and is a ripe direction for future work.

7.2 Limitations and Future Work

There are a few limitations to our study results and setup which we discuss here. First, our study was time-limited and consisted of tasks given to the participants rather than the tasks being self-motivated by the participants. The coding tasks we relied on do not span the entire set of tasks a professional programmer might encounter in their work. The tasks were also easily measured in the number of test cases passed, which may not be the best metric in practice. Second, the participants in our study were entirely comprised of undergraduate and graduate students, which is not fully representative of the audience that uses AI tools for programming. Third, our implementation of the proactive assistant was integrated into a web-based coding environment which does not contain the full features of an enterprise IDE. Fourth, the conditions that we considered in our controlled lab study were limited to chat-based systems, since not all participants had experience using GitHub Copilot. Altogether, we encourage the reader to consider the limitations of our study design when interpreting how our findings may generalize to various real-world programming applications. For example, while we observe that participants did not mention the correctness of suggestions, this may be an important consideration for developers who work on safety-critical code when interacting with a proactive assistant.

Our study presents multiple opportunities for future work. First, further work is necessary to understand the nuances of proactive assistants on a wider range of user backgrounds and coding tasks.

Additionally, future work could consider evaluating the proactive chat assistants in the presence of other programming tools like code completion to understand the interplay between different tools and their relative effects on productivity. There are also multiple promising directions in how proactive assistants can be improved to handle longer contexts, rank suggestions, more clever ways to trigger suggestions, and reduce potential hallucinations in assistant suggestions.

8 Conclusion

In this paper, we explore the design and implementation of proactive AI assistants powered by large language models for chat-based programming assistants. Our prototype considers a set of design considerations that balance the benefits of proactivity while handling its challenges. In a randomized experimental study, we evaluated the impact of various design elements of the proactive assistant on programmer productivity and user experience. We find that proactive assistants can generally increase the number of tasks completed and improve the user experience while coding. Furthermore, we provide insight into how aspects of the design including the effect of suggestion timing and the level of assistant involvement change interaction patterns. These results allow us to revisit the design implications for proactive coding assistants, validating the importance of the design considerations while introducing new considerations that allow for further customization and adaptivity to the user needs. Altogether, this work demonstrates the value of proactivity in AI coding assistants and identifies key directions for further study to further enhance user productivity and experience.

References

[1] Umair Z Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*. 139–150.

[2] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–13.

[3] Anthropic. 2023. Meet Claude. https://www.anthropic.com/claude

[4] Sandeep Avula, Gordon Chadwick, Jaime Arguello, and Robert Capra. 2018. Searchbots: User engagement with chatbots during collaborative search. In *Proceedings of the 2018 conference on human information interaction & retrieval*. 52–61.

[5] Jimmy Baraglia, Maya Cakmak, Yukie Nagai, Rajesh Rao, and Minoru Asada. 2016. Initiative in robot assistance during collaborative task execution. In *2016 11th ACM/IEEE international conference on human-robot interaction (HRI)*. IEEE, 67–74.

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

[7] Nancy Baym, Limor Shifman, Christopher Persaud, and Kelly Wagman. 2019. Intelligent failures: Clippy memes and the limits of digital assistants. *AoIR Selected Papers of Internet Research* (2019).

[8] 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.

[9] Ana Paula Chaves and Marco Aurelio Gerosa. 2021. How should my chatbot interact? A survey on social characteristics in human–chatbot interaction design. *International Journal of Human–Computer Interaction* 37, 8 (2021), 729–758.

[10] Bhavya Chopra, Ananya Singha, Anna Fariha, Sumit Gulwani, Chris Parnin, Ashish Tiwari, and Austin Z Henley. 2023. Conversational Challenges in AI-Powered Data Science: Obstacles, Needs, and Design Opportunities. *arXiv preprint arXiv:2310.16164* (2023).

[11] Mihaly Csikszentmihalyi and Reed Larson. 2014. *Flow and the foundations of positive psychology*. Vol. 10. Springer.

[12] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2024. The Productivity Effects of Generative AI: Evidence from a Field Experiment with GitHub Copilot. (2024).

[13] Cursor. 2023. Cursor - The AI Code Editor. https://www.cursor.com/

[14] Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 143, 8 pages. https://doi.org/10.1145/3613904.3642495

[15] Kathleen Kara Fitzpatrick, Alison Darcy, and Molly Vierhile. 2017. Delivering cognitive behavior therapy to young adults with symptoms of depression and anxiety using a fully automated conversational agent (Woebot): a randomized controlled trial. *JMIR mental health* 4, 2 (2017), e7785.

[16] Github. 2022. GitHub copilot - your AI pair programmer. https://github.com/features/copilot

[17] Ken Gu, Ruoxi Shang, Tim Althoff, Chenglong Wang, and Steven M Drucker. 2024. How Do Analysts Understand and Verify AI-Assisted Data Analyses?. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–22.

[18] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

[19] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[20] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 159–166.

[21] Jiaxiong Hu, Jingya Guo, Ningjing Tang, Xiaojuan Ma, Yuan Yao, Changyuan Yang, and Yingqing Xu. 2024. Designing the Conversational Agent: Asking Follow-up Questions for Information Elicitation. *Proceedings of the ACM on Human-Computer Interaction* 8, CSCW1 (2024), 1–30.

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[23] 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. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–23.

[24] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Z Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. (2024).

[25] Mina Lee, Katy Ilonka Gero, John Joon Young Chung, Simon Buckingham Shum, Vipul Raheja, Hua Shen, Subhashini Venugopalan, Thiemo Wambsganss, David Zhou, Emad A Alghamdi, et al. 2024. A Design Space for Intelligent and Interactive Writing Assistants. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–35.

[26] Mina Lee, Percy Liang, and Qian Yang. 2022. Coauthor: Designing a human-ai collaborative writing dataset for exploring language model capabilities. In *CHI Conference on Human Factors in Computing Systems*. 1–19.

[27] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[28] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 605–617.

[29] Q Vera Liao, Matthew Davis, Werner Geyer, Michael Muller, and N Sadat Shami. 2016. What can you do? Studying social-agent orientation and agent proactive interactions with an agent for employees. In *Proceedings of the 2016 acm conference on designing interactive systems*. 264–275.

[30] Jessy Lin, Daniel Fried, Dan Klein, and Anca Dragan. 2022. Inferring Rewards from Language in Context. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8546–8560.

[31] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).

[32] Tianjian Liu, Hongzheng Zhao, Yuheng Liu, Xingbo Wang, and Zhenhui Peng. 2024. ComPeer: A Generative Conversational Agent for Proactive Peer Support. *arXiv preprint arXiv:2407.18064* (2024).

[33] Ewa Luger and Abigail Sellen. 2016. " Like Having a Really Bad PA" The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 5286–5297.

[34] Christian Meurisch, Cristina A Mihale-Wilson, Adrian Hawlitschek, Florian Giger, Florian Müller, Oliver Hinz, and Max Mühlhäuser. 2020. Exploring user expectations of proactive AI systems. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 4 (2020), 1–22.

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

[36] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. When to show a suggestion? Integrating human feedback in AI-assisted programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 10137–10144.

[37] Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers. *arXiv preprint arXiv:2404.02806* (2024).

[38] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[39] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/

[40] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442* (2023).

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

[42] Zhenhui Peng, Yunhwan Kwon, Jiaan Lu, Ziming Wu, and Xiaojuan Ma. 2019. Design and evaluation of service robot's proactivity in decision-making support process. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13.

[43] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That It Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (nov 2023), 31 pages. https://doi.org/10.1145/3617367

[44] 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. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[45] Orit Shaer, Angelora Cooper, Osnat Mokryn, Andrew L Kun, and Hagit Ben Shoshan. 2024. AI-Augmented Brainwriting: Investigating the use of LLMs in group ideation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–17.

[46] Annika Silvervarg and Arne Jönsson. 2013. Iterative development and evaluation of a social conversational agent. In *Proceedings of the Sixth International Joint Conference on Natural Language Processing*. 1223–1229.

[47] Ella Tallyn, Hector Fried, Rory Gianni, Amy Isard, and Chris Speed. 2018. The ethnobot: Gathering ethnographies in the age of IoT. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–13.

[48] Carlos Toxtli, Andrés Monroy-Hernández, and Justin Cranshaw. 2018. Understanding chatbot-mediated task management. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–6.

[49] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. 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*. 1–7.

[50] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. 2023. Generation probabilities are not enough: Exploring the effectiveness of uncertainty highlighting in AI-powered code completions. *arXiv preprint arXiv:2302.07248* (2023).

[51] Rainer Winkler and Julian Roos. 2019. Bringing AI into the classroom: Designing smart personal assistants as learning tutors. (2019).

[52] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2023. DevGPT: Studying Developer-ChatGPT Conversations. *arXiv preprint arXiv:2309.03914* (2023).

[53] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight anchored explanations of just-generated code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–15.

[54] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[55] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* (2024).

[56] Yu Zhang, Vignesh Narayanan, Tathagata Chakraborti, and Subbarao Kambhampati. 2015. A human factors analysis of proactive support in human-robot teaming. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3586–3593.

[57] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. WildChat: 1M ChatGPT Interaction Logs in the Wild. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=Bl8u7ZRlbM

[58] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. LMSYS-Chat-1M: A Large-Scale Real-World LLM Conversation Dataset. arXiv:2309.11998 [cs.CL]

[59] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).

A User Study Details

A.1 Instructions

General instructions. Welcome to python coding study!

- You will be writing code in Python only, and use only standard python libraries in addition to numpy.
- You will have 40 minutes total where you will try to solve as many coding tasks as possible one at a time.
- It is NOT allowed to use any outside resources to solve the coding questions (e.g. Google, StackOverflow, ChatGPT), your compensation is tied to effort only.

Baseline condition. You will write code in the interface above: a code editor equipped with an AI assistant chatbot. The chatbot does not have access to either the code editor or the task description.

- Please write python code only in the editor. We only support standard python3.8 packages and numpy.
- You can run your code by pressing the "Run" button and the output will be in the output box at the bottom in grey.
- Press the "Submit" button to submit your code. You can only submit your code once for each task.
- You are provided with a chat interface that you can use to help you with the coding tasks.
- Pressing the "Clear" button on the top of the chat window will clear the current chat history. Chat history will be automatically cleared when you move to the next task.
- If the chat output contains a code snippet, you can click the icon to copy the code to your clipboard. Please be aware that its output is not always correct.

Proactive condition. You have access to the Proactive Coding Assistant. In addition to the regular chatbot functionality, the assistant will occasionally provide suggestions when you are stuck! Please try to use it in the study if the suggestions seem helpful. Here's how:

Click on a suggestion title for the following options:

- Previewing: Click "Preview" to see how the proactive assistant incorporates the suggestion into your code, which can then accept or hide the changes. Use the copy icon to copy the code to your clipboard.
- Accepting: click "Accept" to add the expanded suggestion to the chat window and ask follow-up questions.
- Deleting: click "Clear all" to remove all current suggestions. Use "Delete" to remove a single expanded suggestion.

Other features:

- If a regular chat message contains code, you can also have the proactive assistant help you incorporate into your code.
- You can request suggestions by clicking on the "Suggest" button at the top of the chat window, or using the shortcut [Ctrl + Enter] (Windows) or [Cmd + Enter] (Mac) in the code editor.
- After running your code, the assistant analyzes the output or error messages and offers debugging suggestions. You can interact with these suggestions in the same way.

Warning: the assistant does not have access to the task description, but it has access to your code editor and can provide context-aware suggestions based on your code and cursor position. Suggestions are not always perfect, and the code provided may be inaccurate. In addition the suggestions may use packages that we do not support (we only support numpy). Always review suggestions thoroughly before integrating them into your code.

You can review this tutorial again by clicking on the "Show Instructions" button at the top of the page, then clicking "Next".

A.2 Post-study Questionnaire

- Overall, how useful was the non-proactive chat assistant (e.g., the one that did not automatically provide suggestions)? (on a scale of 1-10, where 1 is least helpful and 10 is most helpful)
- Overall, how useful was the proactive chat assistant (e.g., the one that automatically provided suggestions)? (on a scale of 1-10, where 1 is least helpful and 10 is most helpful)
- Briefly explain the reasoning for your ratings above. (free response)
- What kind of suggestions did you find most helpful? (free response)
- In what ways could the proactive chat assistant suggestions be improved?
- In what ways could the proactive chat assistant interface be improved?

B Task Design

B.1 Tasks

We consider a total of 4 tasks in our study, where each task comprises multiple sub-tasks.

Task 1: Storefront.

> You are a freelance software engineer hired to design the backend of an online store. To get started, we have provided starter code for the Store class.
> Your goal is to add more functionality to the online store to make it more complete. Please complete Sub-Task 1 first, then you can complete Sub-Tasks 2 and 3 in any order.
> Sub-Task 1: major additions - write the Order and Product class, which should work with the current code in the Store class
> Sub-Task 2: add the following functionality to the Store class - write function ‘apply_discount_to_order(self, order_id, discount)‘ - write function ‘check_order_status(self, order_id)‘
> Sub-Task 3: create one additional feature that you think might be good to have
> Make sure to write test cases to demonstrate that the online store works as intended.
> You can only submit your code once for this task. Please only submit your code after you have completed as many sub-tasks as you can.

Task 2: To-do List.

You are an application developer working on the backend of a to-do list app. You are working with some existing code your colleague left off. Each task takes in a ‘description‘ (string), ‘category‘ (string), ‘priority‘ (string, ‘"High"‘, ‘"Medium"‘, or ‘"Low"‘), and optional ‘date_due‘ (datetime object).
Your goal is to add more functionality to the to-do list to make it more complete. You can complete the subtasks in any order.
Sub-Task 1: minor fixes - when a task is overdue, change the ‘__str__‘ function of the task to reflect the task is overdue by adding ‘(OVERDUE)‘ after the due date - when adding a new task, do not allow the user to add it if the due date is in the past - there seems to be a bug in the code, please find and fix it - the efficiency of editing a task can be improved, please edit this functionality
Sub-Task 2: add features - modify the ‘list_all(self)‘ function by adding an argument ‘show_completed‘ which prints only unfinished tasks when set to False - add a function ‘list_by_priority(self)‘ to ToDoList which prints the task in order of priority, from high to low
Make sure to write test cases to demonstrate that the to-do list works as intended.
You can only submit your code once for this task. Please only submit your code after you have completed as many sub-tasks as you can.

Task 3: Sales analysis.

You are a data analyst working for a global retailer. You are working with product sales data from the past year.
The sales data is a 4-dimensional numpy array with shape (3, 12, 10, 100) and the following structure:
- Axis 0: Different regions
- Axis 1: Different months
- Axis 2: Different stores
- Axis 3: Product sales figures

Given the described sales data, you need to use ONLY numpy packages to complete the following sub-tasks:
1. write function ‘total_sales_per_region(data)‘ that calculates total sales number for each region
2. write function ‘cumulative_sales(data)‘ that computes the cumulative sales over time for each product across all regions and stores
3. write function ‘top_products_by_sales(data, k)‘ that computes top k best selling product id for each month across all regions and stores
4. write function ‘temporal_correlation(data)‘ that calculates pairwise temporal correlations over time for each product
Your goal is to complete the above sub-tasks. Make sure to write test cases to demonstrate that the code works as intended.
You can only submit your code once for this task. Please only submit your code after you have completed as many sub-tasks as you can.

Task 4: Weather trends.

You are a data analyst working for a meteorological department. Your task is to analyze temperature data for a given month to gain insights into weather patterns and trends.
Given temperature data for a month, you need to use ONLY numpy packages to complete the following sub-tasks:
1. fill out ‘classify_temps(data)‘ which classifies each day's temperature into categories such as 'Freezing', 'Moderate', and 'Hot'.
2. write function ‘clip_temps(data)‘ which clips any extreme temperature values to ensure they fall within -10 and 40.
3. write function ‘compute_moving_avg(data, window_size)‘ which calculates the moving average of temperatures over a specified window_size (e.g., 7 days).
4. write function ‘compute_weekly_avg(data)‘ which calculates weekly average temperatures.
Your goal is to complete the above sub-tasks. The provided code is incomplete and may not be fully correct.
Make sure to write test cases to demonstrate that the code works as intended.
You can only submit your code once for this task. Please only submit your code after you have completed as many sub-tasks as you can.

C Additional results

Do the benefits of proactivity vary by user background? As discussed in Section 6.1, we generally observe benefits across all proactive conditions. A natural follow-up is to analyze whether user benefits vary depending on user background. As part of background and demographic information, we collected participants' self-reported Python proficiency, self-reported AI tool experience, and gender. For the purposes of this analysis, we combine all proactive conditions given the sample size per condition of each subgroup. We compute the difference in performance per participant between the proactive and baseline condition and build a linear model with the background and demographic information as fixed effects. We do not find any statistically significant effect due to Python experience or AI tool usage—this may be due to insufficient sample size per subgroup (e.g., there are only a small number of beginner programmers). However, we do observe a significant difference ($p = 0.04$) due to gender where women tend to benefit more from proactive suggestions by 24.6%. This observation aligns with prior work that showed that software design could have different impacts on different genders [8]. In Figure 7, we visualize the percentage of sub-tasks completed correctly broken down by each of the sub-groups.

What kinds of suggestions do participants tend to accept? As discussed in Section 6.3.3, of the 69 accepted suggestions, the most

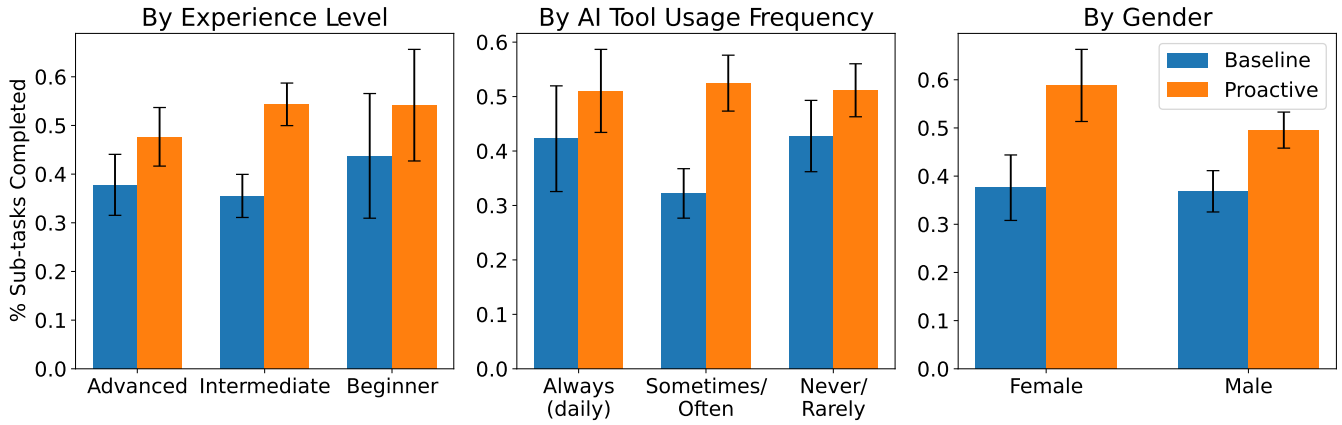

Figure 7: Percentage of sub-tasks completed correctly broken down by sub-groups. Comparing baseline chat to proactive assistants across different sub-groups as defined by Python programming experience, AI tool usage frequency, and gender. We report mean performance (by percentage sub-tasks completed) as well as standard error).

commonly types of accepted suggestions were those on *brainstorming new functionality* and *debugging (latent errors)*, with 18 occurrences each. The least likely accepted suggestions were *explaining existing code* and *pointers to documentation*, with only 1 occurrence each. Other number of occurrences of other suggestions were 12 for *completing unfinished code*, 3 for *debugging (runtime errors)*, 9 for *adding unit tests*, and 8 for *improving efficiency and modularity*. We also observe that all accepted suggestions contain code snippets.