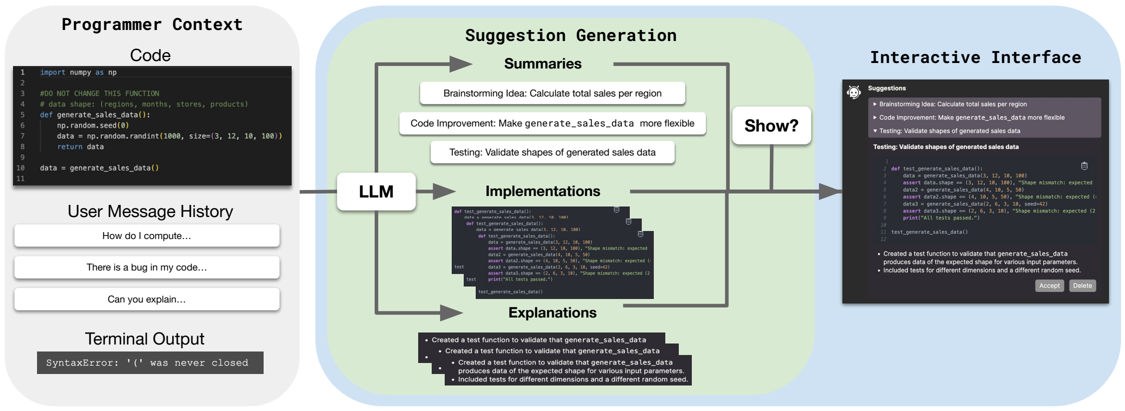
Need Help? Designing Proactive AI Assistants for Programming

VALERIE CHEN, Carnegie Mellon University, USA
ALAN ZHU, Carnegie Mellon University, USA
SEBASTIAN ZHAO, UC Berkeley, USA
HUSSEIN MOZANNAR, Microsoft Research, USA
DAVID SONTAG, Massachusetts Institute of Technology, USA
AMEET TALWALKAR, Carnegie Mellon University, USA



```
import numpy as np

#DO NOT CHANGE THIS FUNCTION
# data shape: (regions, months, stores, products)
def generate_sales_data():
    np.random.seed(0)
    data = np.random.randint(1000, size=(3, 12, 10, 100))
    return data

data = generate_sales_data()
```

```
SyntaxError: '(' was never closed
```

▸ Brainstorming Idea: Calculate total sales per region
▸ Code Improvement: Make generate_sales_data more flexible
▾ Testing: Validate shapes of generated sales data

Testing: Validate shapes of generated sales data

```

def test_generate_sales_data():
    data = generate_sales_data(3, 12, 10, 100)
    assert data.shape == (3, 12, 10, 100), "Shape mismatch: expected
    data2 = generate_sales_data(4, 10, 5, 50)
    assert data2.shape == (4, 10, 5, 50), "Shape mismatch: expected (
    data3 = generate_sales_data(2, 6, 3, 10, seed=42)
    assert data3.shape == (2, 6, 3, 10), "Shape mismatch: expected (2
    print("All tests passed.")

test_generate_sales_data()
```

- Created a test function to validate that generate_sales_data produces data of the expected shape for various input parameters.
- Included tests for different dimensions and a different random seed.

Fig. 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.

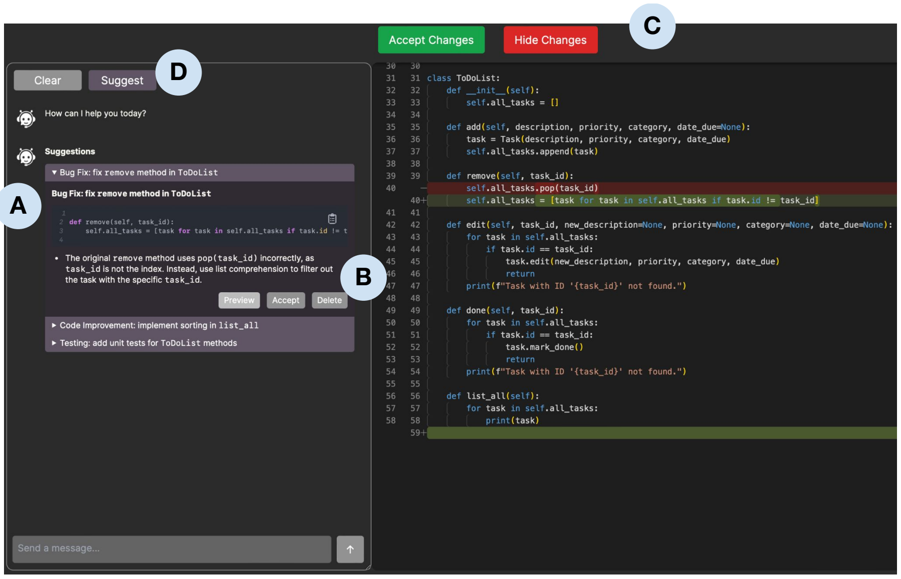
CCS Concepts: • **Human-centered computing → User studies**; • **Software and its engineering → Collaboration in software development**.

Additional Key Words and Phrases: AI-assisted Programming, Proactivity, Mixed-Initiative Interaction

1 Introduction

Chat-based AI assistants such as ChatGPT [8] or Claude [1], 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. 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

Authors' Contact Information: Valerie Chen, valeriechen@cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA; Alan Zhu, alanzhuyixuan@gmail.com, Carnegie Mellon University, Pittsburgh, PA, USA; Sebastian Zhao, sebbyzhao@berkeley.edu, UC Berkeley, Berkeley, CA, USA; Hussein Mozannar, hmozannar@microsoft.com, Microsoft Research, Redmond, WA, USA; David Sontag, dsontag@mit.edu, Massachusetts Institute of Technology, Cambridge, MA, USA; Ameet Talwalkar, talwalkar@cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA.



How can I help you today?

Suggestions

▼ Bug Fix: fix `remove` method in `ToDoList`

Bug Fix: fix `remove` method in `ToDoList`

```

def remove(self, task_id):
    self.all_tasks = [task for task in self.all_tasks if task.id != t

```

- The original `remove` method uses `pop(task_id)` incorrectly, as `task_id` is not the index. Instead, use list comprehension to filter out the task with the specific `task_id`.

▶ Code Improvement: implement sorting in `list_all`

▶ Testing: add unit tests for `ToDoList` methods

```

class ToDoList:
    def __init__(self):
        self.all_tasks = []

    def add(self, description, priority, category, date_due=None):
        task = Task(description, priority, category, date_due)
        self.all_tasks.append(task)

    def remove(self, task_id):
-       self.all_tasks.pop(task_id)
+       self.all_tasks = [task for task in self.all_tasks if task.id != task_id]

    def edit(self, task_id, new_description=None, priority=None, category=None, date_due=None):
        for task in self.all_tasks:
            if task.id == task_id:
                task.edit(new_description, priority, category, date_due)
                return
        print(f"Task with ID '{task_id}' not found.")

    def done(self, task_id):
        for task in self.all_tasks:
            if task.id == task_id:
                task.mark_done()
                return
        print(f"Task with ID '{task_id}' not found.")

    def list_all(self):
        for task in self.all_tasks:
            print(task)
+
```

Fig. 2. **A walkthrough of the proactive assistant interface for coding.** (A) Overview of suggestions, consisting of a short description, implementation, and a brief explanation; (B) 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.

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 [5]. 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 [6]; 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 [3], more recent examples of proactive assistants in the form of autocompletion for text-based tasks have been more successful. For instance, GitHub Copilot [4] provides in-line code suggestions with LLMs. Copilot has been well received by programmers and randomized studies have shown increases in programmer productivity [9, 10]. 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, the particular design details of a proactive assistant are crucial to its effectiveness.

2 Proactive Assistant Implementation

In this paper, we build a proactive assistant for programming tasks. 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.

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. 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. 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 proactive assistant also has a "preview" functionality, which will allow users to see how the assistant would incorporate it into their code. 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, 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. 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.

Suggestion Content. To identify the different types of suggestions that users benefit from for coding contexts, we look to DevGPT [11], a dataset of conversations between programmers and ChatGPT, as a source of real-world questions. 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.

Timing of Suggestions. The proactive assistant must display suggestions at the appropriate time so they are not obstructive and distracting to users. 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 [2] . 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.

3 Study Design

We recruited a total of 65 students with diverse programming experience and levels of regular AI usage. Our study compares proactive assistants to a non-proactive chat-based assistant baseline.

Experimental Conditions. In the `Suggest and Preview` condition, the participant interacts with the proactive assistant described in Section 2. In the `Suggest` condition, the participant interacts with the proactive assistant described in Section 2, without the preview feature. Finally, in the `Persistent Suggest` condition, the participant interacts with a variant of `Suggest` condition, but with a reduced amount of time that an assistant waits to provide a suggestion (from 20 seconds to 5 seconds) and increase in the number of suggestions shown (from 3 to 5). We fix the LLM backbone across all conditions to be GPT-4o [8], a state-of-the-art LLM.

Procedure. 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 [7]. Before participating

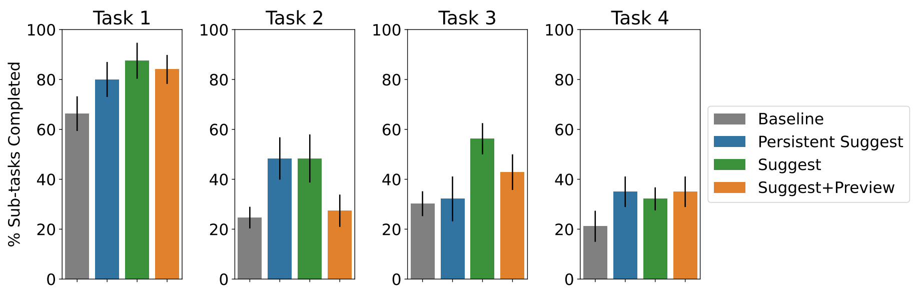


Fig. 3. **Percentage of sub-tasks completed correctly.** Comparing baseline chat to proactive assistants across the four tasks, where Task 1 and Task 2 are system-building questions and Task 3 and 4 are ones where participants work with new packages and functionality. We report average performance and 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.

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.

4 Results

Proactive Assistants Generally Improve User Productivity. 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 3 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`.

User Experience with Proactive Assistants Varies by Implementation. 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. 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."*. 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"*.

References

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

[2] 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.

[3] 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).

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

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

[6] 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.

[7] 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).

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

[9] 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).

[10] 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.

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