Codellaborator: Exploring the Design of Proactive AI Programming Assistance

Anonymous Author(s)

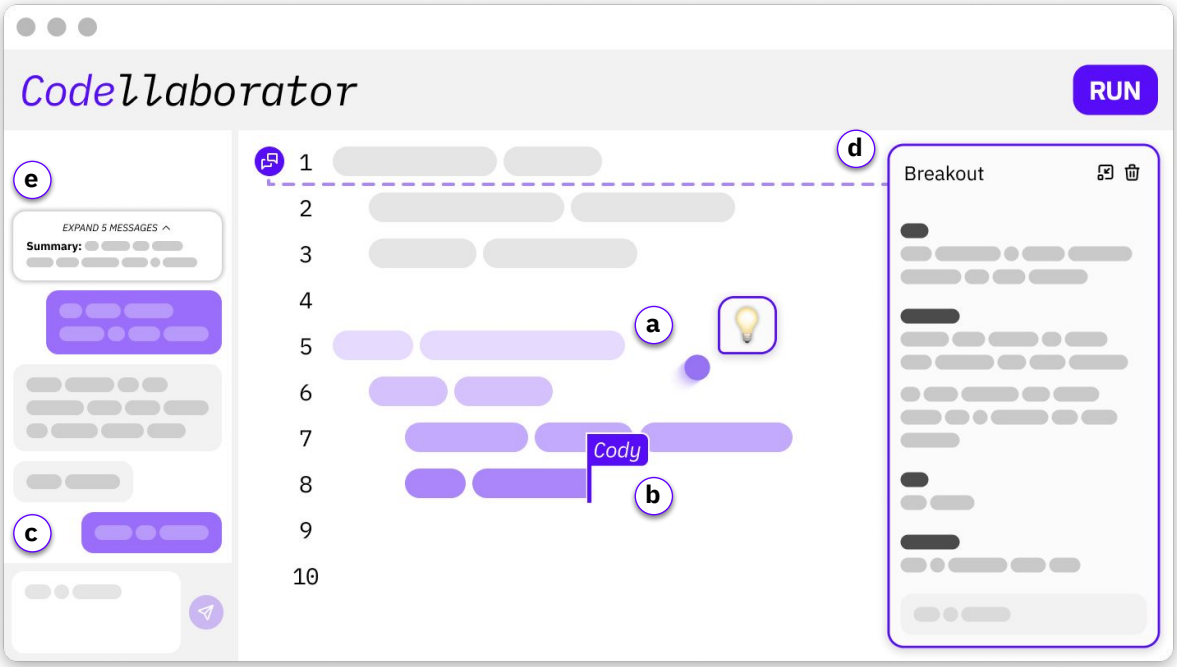

Figure 1: Codellaborator is a proactive AI programming agent in the editor. The AI agent exhibits presence with an autonomous cursor and thought bubble ⓐ to indicate attention and action status, and a caret ⓑ to indicate its editing location. As users code, the AI agent initiates proactive assistance based on users' activity in the editor. Users can also send messages in the chat panel ⓒ or create contextualized "breakout" chats ⓓ anchored in relevant areas of code. The AI agent also automatically summarizes ⓔ and groups chat history to create breakouts to help users manage context.

ABSTRACT

AI programming tools enable powerful code generation, and recent prototypes attempt to lower use efforts by building proactive AI agents, but their impact on programming workflows remains unexplored. As a design probe, we introduce Codellaborator, a proactive AI agent that initiates programming assistance based on user activities and task context. In a three-condition within-subject experiment with 18 programmers, participants reported that compared to a baseline akin to Github Copilot, the Proactive condition reduced expression efforts with AI-initiated interactions but also incurred workflow disruptions. The Codellaborator condition with visible agent presence and context management informed by collaboration principles alleviated disruptions and improved users' awareness of AI processes. We also underscore trade-offs Codellaborator could bring to user control, ownership, and code understanding, and the need to adapt proactivity to various programming processes. Our research contributes to the design exploration and evaluation of proactive AI systems, presenting design implications on AI-integrated programming workflow.

ACM Reference Format:
Anonymous Author(s). 2023. Codellaborator: Exploring the Design of Proactive AI Programming Assistance. In *Proceedings of ACM (Association of Computing Machinery) Symposium on User Interface Software and Technology (UIST '24)*. ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

UIST '24, October 13–16, 2024, Pittsburgh, Pennsylvania, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large language models (LLMs) enabled generative programming assistance tools to provide powerful in-situ developer support

for novices and experts alike [54, 68]. Many existing LLM-based programming tools rely on user-initiated interactions, requiring prompts or partial code snippets as input to provide sufficient context and trigger help-seeking [42, 49, 55, 58, 68]. These systems offer help in the form of code output and natural language explanations to assist users with coding tasks. However, research indicates that users invest considerable effort in formulating prompts, interpreting responses, assessing suggestions, and integrating results into their code [42, 53, 68]. This leads to substantial expression and evaluative costs as users navigate the gulfs of execution (crafting prompts) and evaluation (assessing AI-generated assistance) [31].

To alleviate the prompt engineering costs, newer AI programming tools and designs aim to become more autonomous, allowing the system to initiate interaction and provide proactive assistance. Tools like Github Copilot [4] and Visual Studio's IntelliCode [65] offer the auto-completion feature, which fills the code line as the user is typing to alleviate prompt engineering efforts, to proactively provide in-situ support. However, the AI tool's generated code is not always accurate and the output still requires significant user effort to verify [53, 67, 68]. To address this, some recent commercial and research prototypes envision intelligent AI agents that work as collaborators to communicate directly with users and address code issues autonomously and preemptively. They either leverage dialogue-based interactions [5, 16, 58], expand on the auto-completion feature to proactively make intelligent file changes with an AI caret in the code editor [13], manifest the AI's presence in the editor with an automated AI cursor [14], or allow an AI agent to host its own workspace to autonomously tackle software engineering tasks [15]. However, the effects of proactive assistance for programming compared to the existing user-initiated paradigm and the specific human-AI interaction design approaches are yet to be formally evaluated. Visions of more "proactive" AI programmers additionally raise questions about the potential for harm. Some researchers argue that excessive automation without proper human control can lead to unreliable and unsafe systems [61], and thus some AI systems deliberately avoid proactive AI assistance [58]. Therefore, it remains a question of *how* should AI programming tools provide proactive support. What are the effects of a proactive AI programming tool on user experience? When and where is proactivity helpful, and where might it be harmful?

This research explores the design space of proactive AI programming tools using a technology probe [39] and evaluates the effects of this human-AI interaction paradigm on software engineering practice, illustrating the advantages and drawbacks to provide insights for future designs. We were guided by the research questions:

- **RQ1:** How can we design proactive AI programming assistance that initiates interaction to reduce user effort?
- **RQ2:** What are the benefits and drawbacks of a proactive AI programming tool compared to user-initiated systems?
- **RQ3:** In which programming processes and task contexts can proactivity be helpful, and where can it be harmful?

To answer these research questions, we employ the principles of mixed-initiative interface and human-AI interaction to establish broad design goals [19, 37, 38]. We also take inspiration from theories of human collaborations, programming practice, and workflow interruptions to identify specific designs of the AI agent's programming assistance, including the timings of service, the visual presence, and the interaction context. Informed by this literature, we develop Codellaborator, a technology probe [39] that employs an AI programming agent providing *proactive* assistance and behaviors to enhance human-AI programming collaboration. Codellaborator's proactive abilities allow it to initiate interaction via messages (Fig.1.c) in response to various user activities in the coding environment, and also to commit code edits directly in the editor (Fig.1.b). To mitigate potential disruptions and facilitate collaboration, we designed *presence* and *context* features for the AI agent. In Codellaborator, the agent presence is represented by a cursor and caret (Fig.1.a,b), capable of autonomous movement around the editor, signaling its action, status, and attention focus. To reduce context management burden, both the agent and the user could initialize locally-scoped threads of conversations called "breakouts", anchored to relevant locations in the file (Fig.1.d).

To study the impact of integrating proactive support using an autonomous agent and understand the resulting human-AI collaboration workflows across different programming processes, we conducted a within-subject experiment using three versions of Codellaborator with 18 participants. In the Baseline condition, the ablated system only responds to user prompts and in-line code comments, similar to ChatGPT [2] and Github Copilot [4]. In the Proactive condition, the system proactively initiates interactions and assistance, but without the agent's presence and context features. In the Codellaborator condition, all of the agent's presence, context, and proactivity features are utilized. Our evaluation showed that the Proactive condition reduced user efforts to comprehend system responses through contextualized assistance. But it also caused workflow disruptions and diminished users' awareness of AI's actions, due to interventions without clear signals for agent presence and working context. In contrast, the Codellaborator condition, with its explicit agent presence and context features, significantly lessened these disruptions and improved users' awareness of the AI, leading to a user experience more akin to collaborating with a partner than using a tool. However, in the latter two conditions, participants also encountered a loss of control, ownership, and code understanding due to increased AI involvement in the task with proactive assistance.

In the discussion, we summarize our findings and propose 5 design implications for proactive assistance in human-AI programming. We also discuss participants' ambivalence to adopt a proactive AI tool due to concerns about code maintainability and scalability in real-life scenarios. Through these findings, we present a deeper understanding of the impacts of proactive AI support on programming experience and identify key areas that require further research. In this work, we contribute:

- The design exploration to enable different forms of proactive assistance that expand upon existing AI programming systems.
- The Codellaborator technology probe that implements a proactive AI agent to initiate in-situ assistance and communication.
- The evaluation findings that assess the impact of our design of proactivity, providing design implications for future AI programming tools.

2 RELATED WORK

2.1 AI Programming Tools & Proactive Assistance

Current AI programming tools predominantly operate within the command-response paradigm, where the user triggers help-seeking and obtains generated code and explanations. However, even tools that prioritize productivity, such as Github Copilot [4], do not consistently demonstrate a significant improvement over traditional code completion tools such IntelliSense [6, 68]. Existing research to improve AI programming support has focused on improving the discoverability of the code suggestion scope [67], expanding support to specific task domains (e.g., data analysis) [49], highlighting high-probability tokens to reduce uncertainties [69], or providing more in-IDE code contexts [55]. However, AI-generated assistance could result in discrepancies with the user's expectations, creating barriers to interpreting and utilizing the code output, or even steering the AI in the desired direction in the first place [68]. Another recurrent concern is the potential mismatch in expertise levels between developers and AI agents, leading to reduced productivity in pair programming scenarios [45].

Another approach to enhance AI programming is to leverage the generative power of LLMs to build intelligent programming agents that proactively support users and autonomously complete tasks. Efforts to develop proactive tools that provide automatic support have been explored across various domains, including personalized notifications for weather or calendars [59, 64], health and fitness interventions [57, 60], and support for office workflows [23, 41, 47]. Well-designed, effective invocation of systems' proactive assistance can lower the cost of user manipulation, resolve uncertainties preemptively, and lead to unintentional learning of the system's functionalities [1, 37, 47]. Meanwhile, poorly designed proactive assistance can lead to negative user experiences, diminished control [22, 50, 52], and in some cases, rendering the tool ineffective [23, 41, 50]. While general guidelines on designing mixed-initiative interfaces and human-AI interactions have been established [19, 37], it is uncertain how the design principles can translate to concrete system designs under the context of LLM-assisted programming. For example, while the timing of assistance is one key metric of human-AI interaction design, existing AI programming systems almost always provide immediate response upon output generation without considering interruption to the user's workflow.

Recent research and commercial prototypes have explored many designs to facilitate proactive AI support in programming. Some approaches include integrating dialogue-based interactions [5, 58], expanding on the auto-completion feature to proactively make intelligent file changes with an AI caret in the code editor [13], or manifesting the AI's presence in the editor with an automated AI cursor [14] that mimic the user's workflow and automate repetitive tasks. Some tools take an additional step towards fully autonomous AI and constructed an AI agent capable of host its own workspace with code editor, console, and web browser to autonomously tackle software engineering tasks in response to a user prompt [15]. However, the impact of proactive programming assistance, as opposed to the current user-initiated paradigm remains to be formally assessed. Similarly, the resulting benefits and drawbacks to user experience from employing these specific design approaches need to be measured. Our research not only aims to explore how to effectively design and integrate proactive AI assistance into developers' workflows but also seeks to gain a deeper understanding of the impact on the programming experience through a comprehensive study.

2.2 Proactivity & Interruption while Collaborating

Designing an AI collaborator that enables positive experiences and outcomes is a challenging endeavor. The nuances of effective human-to-human collaboration are still not fully understood and vary greatly depending on the context, thus resulting in the crafting of successful human-AI collaboration paradigms being even more daunting. Past research has shown that two factors, i.e., proactivity and interruption, play pivotal roles in shaping the outcomes of team collaborations. Prior work in psychology has revealed that proactivity, when effectively managed, can provide positive affective outcomes during collaborative work [43], however, the current landscape of human-AI collaboration is often characterized by either human-dominant or AI-dominant dynamics. In such situations, both human and AI agents operate reactively, adhering to pre-defined instructions. This reactive stance, while serviceable, doesn't leverage the full potential of robotic automation and human cognition and often fails to alleviate the cognitive and psychological burdens borne by human developers.

Interruption, or *"an event that breaks the coherence of an ongoing task and blocks its further flow, though allowing the primary task to resume once the interruption is removed"* has been a subject of study for decades [48]. Numerous studies have highlighted the detrimental effects interruptions can have on users' memory, emotional well-being, and ongoing task execution [20, 29, 38]. To mitigate the challenges posed by interruptions, we drew insights from psychology, behavior science, and social theory, to foster collaborations that would be perceived as less disruptive. For instance, as the perceived level of disruption is influenced by a user's mental load at the time of the interruption [20, 21, 29], we designed interactions that were aware of a user's working context before generating notifications. Furthermore, as prior work has highlighted that people experience varying degrees of disruption during different sub-tasks [29, 38, 40, 51], we apply these principles when designing the timing of service of our probe to adopt a proactive collaborator role at moments when the context was most appropriate.

2.3 Collaborative Programming

Our research was informed by existing research on team collaboration during software development, specifically pair programming, work interruptions, and help-seeking behaviors.

2.3.1 Pair Programming. Pair programming is a paradigm where two users collaborate in real-time while at a single computer, with one user writing the code (i.e., the driver) and the other reviewing the code (i.e., the observer) [28]. Pair programming has been shown to lead to better design, more concise code, and fewer defects within approximately the same person-hours [24, 70]. Other research has reported that these benefits may have been due to the awareness of another's focus within the code, which can be

invaluable for problem-solving [44]. For example, Stein and Brennan found that when novices observed the gaze patterns of expert programmers during code reviews, they pinpointed bugs faster [62]. However, the most prominent challenges associated with pair programming include cost inefficiency, scheduling conflicts, and personality clashes [24]. Facilitating visible presence and actions between collaboration partners has previously demonstrated its efficacy in physical workspaces [30]. Our design probe loosely adopted the pair programming paradigm where the AI agent and the user can adapt and exchange the roles of the driver and the observer We also implemented visible presence and clear context information to enhance mutual awareness between the user and the AI.

2.3.2 Help Seeking while Programming. Pair programming has notable overhead costs as a real-time dyadic paradigm. To seek more accessible help, developers often use Community Question-Answering (CQA) websites such as Stack Overflow [12]. These platforms not only allow developers to post questions but also archive answers for future reference, a concept rooted in Answer Garden's creation of an "organizational memory" [17, 18]. However, many questions that are well-suited for an intelligent agent are misaligned with the design of CQA websites. A previous study utilized a "hypothetical intelligent agent" as a probe to understand developers' ideal help-seeking needs [27]. The findings, along with other studies' results, highlighted several limitations of CQA sites, including delayed feedback, lack of context, and the necessity for self-contained questions [46]. Consistent with this, prior work advocated for systems that intuitively captured a developer's context and used it to enable developers to select a code snippet, ask the system to "please refactor this", and promptly receive pertinent responses [26]. Inspired by this research, we designed our system to employ these intelligent characteristics that detect the current working context to reduce the effort spent on help-seeking processes while still ensuring the users received high-quality responses.

3 DESIGN GOALS

A technology probe is an "*instrument that is deployed to find out about the unknown—returning with useful or interesting data*" [39]. Technology probes are a common approach in human-computer interaction research to engage users directly in contextualized studies [34, 35]. To investigate the effects of a proactive AI programming agent on user workflow, we seek to build a prototype that probes by extending the features on existing tools and exploring the design space of interactive human-AI systems based on established guidelines [19, 37] (RQ1). We further analyze relevant literature on human collaboration and interruption in programming and other types of teamwork to identify key design considerations to guide the implementation of the Codellaborator prototype:

- **DG1: Facilitate proactive AI assistance based on users' activities and working context** by anticipating developer needs based on their actions in the code editor and offering timely suggestions, insights, or corrections to support their development process.
- **DG2: Increase AI visibility to enhance user's awareness** by representing the AI agent using visible cues to indicate its actions, intentions, and decision-making processes.
- **DG3: Reduce user interaction efforts to adopt proactive AI assistance** by minimizing interruptions, reducing conflicts, and managing past interaction contexts.

4 CODELLABORATOR

In this section, we introduce three main components of Codellaborator and how they are implemented to achieve our design goals: proactivity, AI agent's presence, and shared working context. The prototype integrates an AI agent powered by LLM into a Python editor environment. The AI administers proactive assistance and intervention via chat messages and direct code edits in the file (Fig.2.c,e) and is represented by an autonomous cursor and caret (Fig.2.d,b). To manage past human-AI interactions, the system allows both the user and the AI to create contextualized threads of conversation locally scoped in the file (Fig.2.a). We detail the system tech stack and LLM configuration at the end of the section.

4.1 Proactive Programming Assistance

To explore the design for proactive programming support based on users' actions (DG1), we adopted a set of findings from research on *interruption management*, and distilled them down to three proactivity design principles [21, 29, 40, 51]. Following the human-AI interaction (HAI) guideline of timing services based on context [19], we instantiated six proactivity features in our design probe to minimize interruptions to the user (DG3), summarized in Table 1. The proactive assistance in Codellaborator serves to present one design approach that expands upon existing AI programming features and allows us to investigate the effects of an AI agent equipped with highly proactive capabilities on users' programming workflow.

The first principle states that the most opportune moments for interruption occur during periods of low mental workload [21, 29]. In our system, we predict low mental workloads when users are not performing actions, such as writing code, moving around the file, and selecting ranges (i.e. when the user is idle). Since idleness could also mean the user is engaged in thoughts, the AI agent only intervenes after an extended period of inactivity, which could signal that the user is mentally stuck and needs assistance (Table 1, 1). This is a rough estimation to interpret user's working states and future works could employ more advanced models to identify user's cognitive process.

The second principle posits that people perceive interventions as less disruptive at the beginning of a task or subtask boundaries [29, 40, 51]. Task boundaries are defined as when one subtask is completed (evaluation) or when the next subtask begins (goal formulation) [51]. To convert this implication to a system feature, we used event listeners to detect users' task beginnings and boundaries in programming, specifically, when the user completed a block of code (i.e., after outdenting in Python), executed the code, or made a multi-line edit (i.e. pasting a block of code) (Table 1, 2-4).

Finally, we draw inspiration from existing AI programming tools [4, 5, 55] and propose **the third principle**: intervene when users are communicating through implicit signals. Existing systems don't always use direct messages as the means of human-AI communication. For example, in Github Copilot [4], creating a new line after a comment prompts the tool to generate code based on the comment content. Another example is in Nam et al.'s work, the

User action	System reaction trigger	Possible actions
Design Rationale 1: intervene at moments of low mental workload [21, 29]		
1. User has been idle (no code edit, caret movement, or selection change)	Initial idle threshold is 30 seconds. If user ignores and maintains idle, add 30 seconds to the threshold.	1. If the user is on an empty or trivial line (e.g. pass), no response 2. Offer help via message
Design Rationale 2: intervene at task boundary (i.e. when completed one subtask and formulating the next) [40, 51]		
2. User has completed a block of code	User outdents from a code scope in Python (e.g. an if- statement, loop, or function).	1. If block is insignificant, no response 2. Notify user of code issues 3. Suggest optimization to user 4. Adds documentation in editor
3. User has executed the program	User executes the program in editor. Output displays in console.	Acknowledge the code execution. If output contains error, offer to help.
4. User has made a multi-line code change	User pastes code that's more than 1 line.	1. If change is insignificant, no response 2. Add documentation in editor 3. Notify user of code issues
Design Rationale 3: intervene when user is potentially communicating through implicit signals [4, 5, 55]		
5. User has made a code comment	User starts a newline after a single line or multi-line comment.	1. If nothing to address, no response 2. If the comment describes function, generate code suggestion 3. If posing a question, offer help
6. User maintains selection on a range of code	Initial selection threshold is 15 seconds. If user ignores and maintains selection, add 15 seconds to the threshold.	1. If insignificant selection, no response 2. Explain the selected code 3. Analyze selection to error-check

Table 1: Proactivity features in Codellaborator. The table details the design rationales derived from interruption management literature [21, 29, 40, 51] and prior tools [4, 5, 55]. We designed six proactivity features triggered by user activities. For each feature, the AI agent selects an action from the defined list based on the working context, i.e. the user's caret location and local file content.

system uses the user's current selection in the editor as context to provide code generation [55]. While these features demonstrate a low level of proactivity individually, we assimilate the existing designs to enhance the proactivity of Codellaborator.

In each proactivity feature, the AI agent is provided with the user's caret position and local code context. Then, the AI agent utilizes the LLM to triage the situation and decide whether to intervene and what type of help to offer. When the LLM deems it appropriate to act, it would select an option from a defined list of editor actions. We also implemented adaptiveness within the AI agent's proactivity. This corresponds with the learning from user behavior and the update and adapt guidelines in HAI [19]. For example, each time an idle or selection intervention (Table 1.1,6) was ignored by the user, we imposed a penalty on the action and made it less frequent in the future to decrease unnecessary interventions. Additionally, we prioritized the user's initiative and actions over the AI agent's, providing the user with ultimate control [19]. When the user was initiating a conversation with the assistant, we canceled pending AI agent actions and active API requests, so as to not disrupt the user's train of thought and wait for their updated input. To enable users to control the amount of visual signals they receive, the chat interface could be fully collapsed, providing more space for the code editor and hiding potentially distracting messages.

In general, these guidelines lead the probe system to create proactive LLM-based agents for coding support. The agent can **proactively** make requests to the LLM and take agent actions to help facilitate direct communication and collaboration with the user. With this approach, we attempt to improve the intervention timing beyond rule-based heuristics and use the LLM for its decision-making capabilities, while also constraining the model to take feasible and reasonable actions in a programming assistance context.

4.2 AI Agent's Presence

To increase the user's awareness of the AI agent's action, intentions, and decision-making process (DG2), Codellaborator manifested the AI's presence with visual cues guided by the Social Transparency theory [33, 63] and prior explorations on transparent design in explainable AI [32]. Specifically, we were inspired by the concept of interaction transparency, which posits that the visibility of the presence of other parties and sources of information in human collaboration can reduce interaction friction [33, 36].

To do so, we added a visual caret and cursor in the editor workspace automated by the AI agent. The AI caret indicated its position in the text buffer and moved as the AI agent selected and edited code (Fig.2.b). The AI cursor, which moved independently of the caret, serves as an indicator of the AI agent's "attention" and demonstrates its actions (Fig.2.d). For example, to rewrite a block of code,

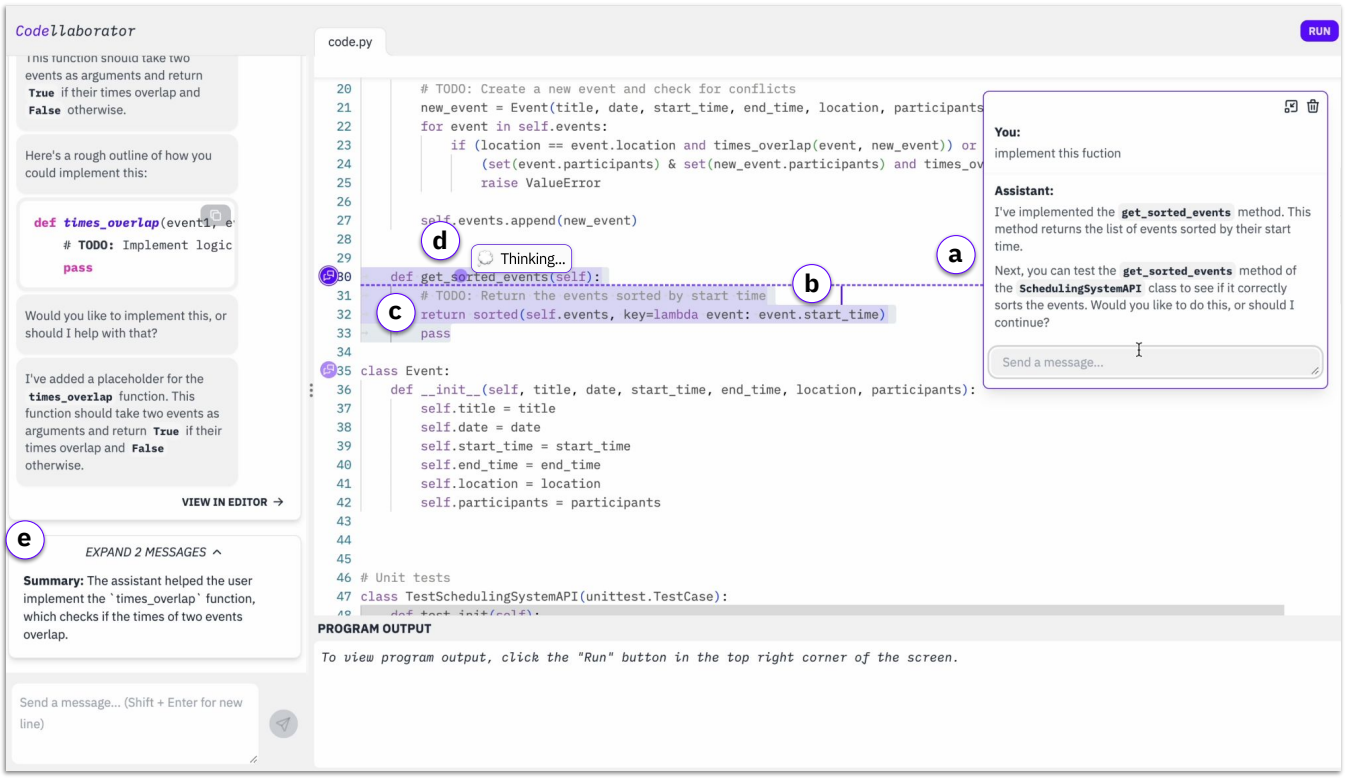

Figure 2: Codellaborator UI in action. The user asks the AI agent for help with implementing `get_sorted_events` using a breakout chat ⓐ on line 30. The AI adds code with its caret ⓑ in the editor and replies to the user in the breakout to discuss the next steps. The purple highlight ⓒ indicates the provenance of the added code and fades away after 5 seconds. The AI agent's cursor displays "thought-balloon Thinking..." ⓓ to indicate its working process of generating a response. In the main chat panel to the left, messages are automatically organized by topic with summaries ⓔ. Note that during actual usage, when a breakout is opened, the main chat panel is blurred to alleviate cognitive load and signal context focus switch, but this feature was disabled for UI demonstration purposes.

the AI cursor would select a range, delete the range, and stream the new code in a typing motion, similar to how a human user would. We introduced human elements into the AI agent's actions to elicit the social collaboration heuristics to facilitate better human-AI collaboration [32].

While the AI agent was processing or taking action, a thought-bubble overlay floated adjacent to the cursor and contained an emoji and short text to convey the AI agent's working state (Fig.2.d). For example, a writing hand emoji writing-hand indicated that the AI agent was writing code in the editor, whereas a tool and laptop emoji hammer-and-wrenchlaptop indicated that it was analyzing the program execution output. We also enabled the AI agent to include emojis in their message response to add variety. A subset of emojis that conveyed system actions was parsed from the AI messages and displayed alongside the AI cursor. This design enabled users to be aware of new messages even when the chat was collapsed, and allowed them to decide whether to engage based on the status displayed. The option to attend to the textual messages or the visual presence afforded different levels of interaction details, handing users control over the amount of information to be received from the AI agent. The system also indicated its working progress by streaming a response with a "pending" indicator in the chat panel and a "hourglassLoading" signal on the AI cursor, thus preventing confusion about whether the system was responsive or stalled.

4.3 Context Management for User Activities and Past Interactions

As part of enabling helpful proactive assistance based on user activities (DG1) and helping the user keep track of past interaction context (DG3), Codellaborator summarizes the history of past interactions and arranges them in relevant locations in the code editor for both the user and the AI agent. We stem this feature from the HAI guideline to show contextually relevant information [19] and the concept of content transparency in the Social Transparency theory, which relates to the awareness of the origin and history of the content and actions that occurred during the interactions [63]. This includes preserving context over time, representing context visually, and maintaining the provenance of information and generated artifacts (i.e. code). The AI agent consistently tracked many forms of context, such as the user's current caret position in the file, the contents of the file, the user's activity (or lack thereof), and the

editor console output. Supplying the user's working context and editor environment state enabled the AI agent to better address the needs of the user and prevent disruptions.

Meanwhile, to better manage context for the user, the AI agent organized the past conversation context by grouping semantically relevant messages by topic and "breaking out" the subset of messages from the chat. This is guided by the HAI principle to present the most contextually relevant information [19]. The selected messages were collapsed in the main chat panel and anchored to an expandable thread at the appropriate area of code in the editor (Fig.2.e). This enabled the conversation about a specific code section to be placed directly in a localized context, so the link between the code and the process that created it was represented visually. Breakouts also provide an easy way to access past conversations without needing to scroll through the chat when the collaboration session is prolonged. The breakout function also required that the AI agent provide a short summary, which was displayed in the chat in a collapsed component (Fig.2.e). The collapsed component preserved the provenance of the original messages and also provided a button to navigate to the attached thread in the code editor. The new breakout chat remained interactive, so the user could continue the conversation with the AI agent and suggest edits, ask for explanations, and more, in situ (Fig.2.a). The user could also manually initiate a breakout in the file. By doing this, they anchored their queries to a specific line in the file, implicitly providing the AI agent with context information to inform their responses and/or actions.

To provide provenance information on the code artifact, the AI agent's code edits were highlighted in purple (Fig.2.c) to distinguish authorship from user-written code. The highlights faded away after 5 seconds to avoid visual distraction and could reappear when the user clicks between the range of the AI code edits.

4.4 Probe System Implementation

Codellaborator was implemented as a React [11] web application using `TypeScript`. The front-end code IDE was built on top of the Monaco Editor [8]. The code execution relied on a separate web server powered by Node.js [9] and the Fastify library [3]. The scope of Codellaborator enabled users to execute single-file Python3 code for proof of concept. The back-end of the system was powered by the GPT-4 [56] large-language model that was connected via the OpenAI API [10]. Specifically, we used the `gpt-4-0613` model which enabled function-calling[1]. This allowed the system to define functional tools that the LLM was aware of and could employ to make editor changes if it saw fit according to a defined schema. We created four such tools for the model, i.e., to authorize code insertion, deletion, replacement, and message grouping to create breakouts. To configure the LLM agent, LangChain [7] was used to maintain a memory of the past message contexts and provide system messages to define the role and responsibility of the agent.

To provide a collaborative experience, we initialized the AI agent with a system prompt (Appendix A) that delegated it the role of a pair programming partner. This corresponds with the HAI guideline of matching relevant social norms with the user [19]. We also provided it with the basic context of the IDE interface, including the chat panel, the editor, and the console. The AI agent was asked to follow human pair programming guidelines [28, 70], which defined the observer-driver responsibilities and enforced a friendly tone, constructive feedback, and a fair delegation of labor.

[1]Atty Eleti, Jeff Harris, and Logan Kilpatrick. 2023. Function calling and other API updates. Retrieved August 16, 2023 from https://openai.com/blog/function-calling-and-other-api-updates

5 EVALUATION

To evaluate the benefits and drawbacks of our design probe (RQ2) and understand the human-AI interaction workflows in different programming processes (RQ3), we conducted an in-person user study where participants collaborated with three versions of Codellaborator in pair-programming sessions.

5.1 Participants

We recruited 18 upper-level CS students from our university (8 female, 10 male; mean = 21.3 years, SD = 1.49 years, range = 19-24 years; denoted as P1-P18). Participants had a mean coding experience of 5.6 years. Fifteen of the participants had used LLM-based AI tools like ChatGPT [2], and thirteen participants used AI tools for programming at least occasionally. Participants were recruited via a posting in the Discord and Slack channels for CS students at the university. Each study session lasted around 90 minutes and participants were compensated with $40. The study was approved by the ethics review board at our institution.

5.2 Study Design

A within-subjects study design was used. The study involved three conditions on three prototypes of Codellaborator to examine the effects of different proactivity designs compared to a user-initiated baseline (RQ2). To account for ordering and learning effects, the presentation order of three conditions was counterbalanced. The underlying LLM in all three conditions is initialized with the same system prompt (Appendix A) and setting (e.g. version, temperature, etc.). The there conditions are described below:

- **The Baseline condition** was an ablated version of Codellaborator, similar to existing AI programming tools like Github Copilot [4] and ChatGPT [2] where users prompt using code comments or chat messages to receive AI response. The system only reacts to users' requests. This system did not have access to directly make code changes in the editor.
- **The Proactive condition** constructed an AI agent that takes proactive actions, such as sending messages or writing code in the editor, to provide help based on user activity. In this condition, the ablated system did not include any additional indicators of the AI agent's presence and did not provide context management support.
- **The Codellaborator condition** utilized the same AI agent and proactivity features found in the P condition, and additionally employed the presence and context features. In this condition, the full system represented the AI via its autonomous cursor, caret, and intention signal bubble (Fig.2.b,d). Moreover, users were allowed to use breakouts to start localized threads of conversations at different parts of the code (Fig.2.a). Codellaborator also automatically grouped relevant messages and organized them into breakouts to manage interaction context (Fig.2.e).

5.3 Tasks

Three programming tasks of similar difficulty were used during the study. The tasks involved implementing a small-scale project in Python (full descriptions in Appendix B, Task 1: event scheduler, Task 2: word guessing game, Task 3: budget tracker). We adopted test-driven development and provided users with a unit test suite for each task, where they had to implement the specification to pass the test cases. The tasks were intentionally open-ended so that the LLM could not solve them deterministically and immediately. In an attempt to maintain consistent generation quality across participants, the backend GPT-4 model was set to 0 temperature to reduce randomness. Based on a pilot study with 6 users using the Codellaborator condition prototype, we found that participants were able to complete each task within 20-30 minutes, thus showcasing similar task difficulties. During the study, the tasks were randomly assigned to each condition to reduce bias.

5.4 Procedure

After signing a consent form, participants completed each of the three coding tasks. Before each task, participants were shown a tutorial about the system condition they would use for the task. Participants were asked to discuss the task with the AI agent at the start of each task to calibrate participants' expectations of the AI agent and reduce biases from prior AI tool usage. Participants were given 30 minutes per task and were asked to adopt a think-a-loud protocol. After each task, participants completed a Likert-scale survey (anchors: 1 strongly disagree to 7 strongly agree) about their experience in terms of the sense of disruption, awareness, control, etc. (Fig.4). After completing all three tasks, participants underwent a semi-structured interview for the remainder of the study. Each session was screen- and audio-recorded and lasted around 90 minutes.

5.5 Data Analysis

In our study, we placed a significant emphasis on understanding the human-AI interaction patterns and nuanced user experience through qualitative analysis. The semi-structured interview responses were individually coded by three researchers. Subsequently, thematic analysis [25, 66] was employed by one researcher to distill participants' key feedback, offering deep insights into the dynamics of human-AI collaboration in different programming processes. This qualitative approach enabled us to capture the complexities and subtleties of participants' experiences, perspectives, and the overarching themes that emerged from their interactions with the AI system. By focusing on the qualitative data, we aimed to uncover the underlying factors that influenced participants' engagement with the AI agent and specific parts of the system design that affected their workflow, experience, and perception of AI programming tools.

Additionally, we conducted quantitative analysis and recorded the task duration and the number of test cases completed for all (3 X 18 = 54) task instances. We also coded 1004 human-AI interaction episodes from the recordings. An episode starts when either the user or the AI sends a message, and ends when the user moves on from the interaction (e.g. starts writing code after reading AI comments, or writes a response message to initiate a new interaction). The interaction data was then labeled with the timestamp, the duration, the expression time (e.g. the time lapsed to write a direct message to the AI agent), the interpretation time (e.g. the time lapsed for the user to read the AI agent's response or code edit), the current programming process (e.g., design, implement, or debug), and a description of the workflow between the human and the AI agent. Aggregating these interaction episodes, we recorded the number of disruptions, defined as instances when the user stopped their action as a direct result of the AI agent's actions. This quantitative analysis offered a structured overview of the interaction patterns and efficiency at different stages of software engineering workflow.

6 RESULTS

Among the 54 task instances, participants successfully completed the programming task in 50 instances, passing all test cases. In 4 instances, the task was halted as participants did not pass all test cases within 30 minutes. The mean task completion time was 16 minutes 46 seconds, with no significant difference across system conditions, task orders, or tasks.

To understand the effects of proactivity on human-AI programming collaboration (RQ2), we first report participants' user experience comparison between prompt-based AI tools (e.g. ChatGPT), their perceived effort of use, and the sense of disruption. We then describe participants' evaluation of the Codellaborator probe's key design features, including the AI agent presence and context management. We also discuss the human-AI interplay between users and different versions of the system, covering their reliance and trust towards AI, and their own sense of control, ownership, and level of code understanding while using the tools. Analyzing the 1004 human-AI interaction episodes, we illustrate how users interacted with the AI agent under different programming processes, as well as discuss participants' preference to utilize proactive AI in different task contexts and workflows (RQ3).

6.1 Codellaborator Reduces Expression Effort and Alleviates Disruptions

Overall, participants acknowledged an increased involvement of AI in the Proactive and Codellaborator conditions, leading to higher efficiency (P1, P2, P13, P15, P18). P2 commented *"I couldn't accomplish this [task] in a short time, so that's the reason I use that [AI support]."* Similarly, P15 remarked:

> *"Now that I have experienced this AI assistant, I think that the arguments about AIs are out there taking programming jobs...has some merit to it... Just for the convenience of programming, I would love to have one of these in my home. (P15)"*

Participants commented that prompt-based tools, like Github Copilot or the Baseline in the study, were more effortful to interact with (P7, P8, P10, P12, P14). Participants thought this was due to the proactive systems' ability to provide suggestions preemptively (P7), making the interaction feel more natural (P8). After experiencing the Proactive and Codellaborator prototypes, P10 felt that *"in the third one [Baseline], there was not enough [proactivity]. Like I had to keep on prompting and asking."* The proactive agent interventions also resulted in less effort for the user to interpret each AI action in

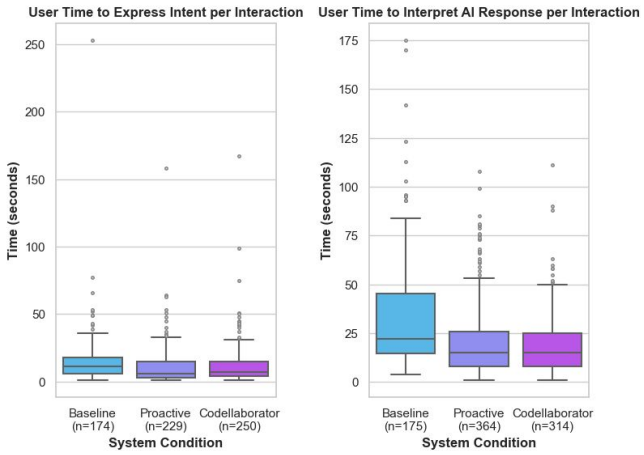

Figure 3: The time to express user intentions to the AI and the time to interpret the AI response per interaction. Users' expression time was not significantly different across conditions $F(2,652) = 2.36$, $p = 0.095$). Users' interpretation time varied ($F(2,856)= 41.1$, $p < 0.001$), and was significantly lower for Proactive and Codellaborator conditions than in Baseline.

both Proactive and Codellaborator conditions compared to the Baseline (Figure 3). Among 857 recorded human-AI interaction episodes where the user did not ignore the AI engagement, we observed a significant difference in the amount of time to interpret the AI agent's actions (e.g., chat messages, editor code changes, presence cues) per interaction across three conditions ($F(2,856)= 41.1$, $p < 0.001$) using one-way ANOVA. Using pairwise T-test with Bonferroni Correction, we found the interpretation time significantly higher in Baseline ($\mu = 34.5$ seconds, $\sigma = 30.1$) than in Proactive ($\mu = 19.8$ seconds, $\sigma = 17.2$; $p < 0.001$) and Codellaborator ($\mu = 18.7$ seconds, $\sigma = 14.9$; $p < 0.001$). There was no significant difference in the time to interpret between the Proactive and Codellaborator conditions ($p = 0.398$; Figure 3). This indicates that when the system was proactive, participants spent less time understanding AI's response and incorporating them into their own code, potentially due to the context awareness of the assistance to present just-in-time help. However, we did not find a significant difference in the time to express user intent to the AI agent per interaction (e.g. chat message, in-line comment, breakout chat) ($F(2,652) = 2.36$, $p = 0.095$), despite qualitative feedback that the Baseline without proactivity was the most effortful to communicate with.

While the added proactivity allowed participants to feel more productive and efficient, they also experienced an increased sense of disruption, especially in the Proactive condition, when the AI agent did not exhibit its presence (P1, P9, P10, P14). Disruptions occurred in different patterns across the three conditions. In Baseline, disruptions arose from users accidentally triggering AI responses via the in-line comments (similar to Github Copilot's autocompletion) while documenting code or making manual changes during system feedback, leading to interruptions. In Proactive, disruptions were due to users' lack of awareness of the AI's state, leading to unanticipated AI actions while they attempted to manually code or move to another task, making interventions feel abrupt. For example, P14 found the lack of visual feedback on which part of the code the AI modified made the collaboration chaotic. Similarly, P12 felt that the automatic response disrupted their flow of thinking, leading to confusion. In Codellaborator, similar disruptions occurred less frequently due to the addition of AI presence and context management features. However, miscommunications about turn-taking between the user and AI sometimes arose, resulting in both parties acting simultaneously and causing interruptions.

Analyzing the Likert-scale survey data using the Friedman test, participants perceived different levels of disruptions among three conditions ($\chi^2 = 22.1$, $df = 2$, $p < 0.001$, Fig.4 Q1), with the highest in Proactive ($\mu = 4.61$, $\sigma = 1.58$), then Codellaborator ($\mu = 3.78$, $\sigma = 1.86$) and Baseline ($\mu = 1.56$, $\sigma = 1.15$). Using Wilcoxon signed-rank test with Bonferroni Correction, we found higher perceived disruption in Proactive than Baseline ($Z = 3.44$, $p < 0.01$), and in Codellaborator than Baseline ($Z = 3.10$, $p < 0.01$). We did not find a statistically significant difference in perceived disruption between Proactive and Codellaborator ($Z = -1.51$, $p = 0.131$). The perceived disruptions might be due to the additional visual cues exhibited by the AI agent, which we further discuss in Section 6.3.

The results presented a multifaceted outcome of using proactive AI assistance in programming. On the one hand, the AI reduced users' effort to specify and initiate help-seeking, enhancing productivity and efficiency by leveraging LLM's generative capabilities. Meanwhile, AI's increased involvement in user workflows created disruptions, but this was alleviated to an extent by our design of Codellaborator, although participants perceived level of disruption varied widely (Fig.4 Q1). We further discuss designs to adapt the salience of the AI presence in the user interface and improvements to the timing of service in the Discussion.

6.2 Codellaborator and Proactive Feel Like Programming with a Partner than a Tool

Another effect we observed from participants using the proactive conditions was an elevated sense of collaboration rather than using the programming assistant as a tool. While in all three conditions, the AI agent was initialized with the same prompt that enforces pair programming practices (Appendix A), some participants expressed that working with the Codellaborator and Proactive conditions felt more like collaborating with a more human-like agent with presence ($N = 6$) than the Baseline. P1 commented that *"it's like a person that's on your side [and says] 'that's over here. You add that here' and kind of felt that way."* P6 reflected that *"just the fact that it was talking with me and checking in with a code editor. I maybe treated it more like an actual human."* A part of this is due to the integration of the AI to the code editor, as P14 reflected *"by changing the code that I'm working on instead of like on the side window...it feels more like physically interacting with my task."* Even the disruptions arising from the proactive AI actions facilitated a human-like interaction experience. P9 recalled an interaction where they encountered a conflict in turn-taking with the AI: *"[AI agent] was like, 'Do you want to read the import statement? Or should I?' I was like, 'No, I'll write it' and it [AI agent] said 'Great I'll do it' and it just did it. Okay, yeah. True to the human experience."*

This different sense of collaboration was corroborated by the survey results (($\chi^2 = 22.1$, $df = 2$, $p < 0.001$, Fig.4 Q8). Participants

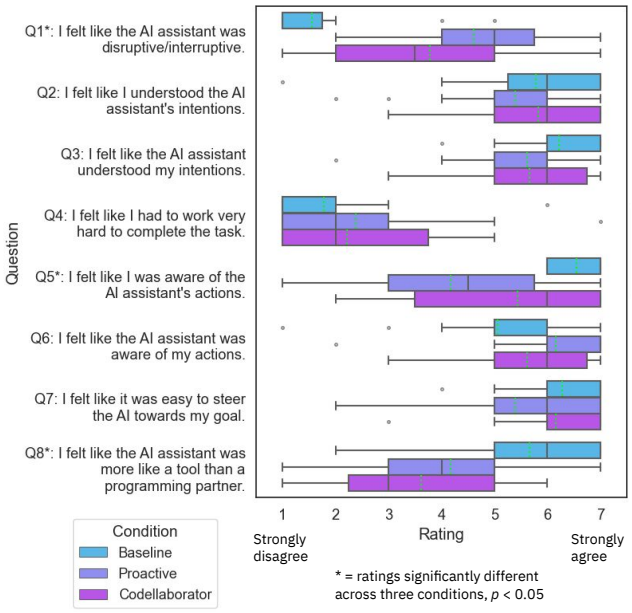

Figure 4: Likert-scale Response displayed in box and whisker plots comparing three conditions. Anchors are 1 - Strongly disagree and 7 - Strongly agree. The green dotted lines represent the mean values for each question. Using the Friedman test, we identified significant differences in rating in Q1 for disruption, Q5 for awareness, and Q8 for partner versus tool use experience.

rated the AI assistant in the Baseline to be much like a tool (μ = 5.67, σ = 1.58), while both the Codellaborator (μ = 3.61, σ = 1.65) and the Proactive conditions (μ = 4.17, σ = 1.72) felt more like a programming partner (both $p < 0.001$ compared to Baseline). This more humanistic collaboration experience introduced by proactive AI systems naturally brings questions to its implications for programmers' workflow. We further share our analysis across programming processes in Section 6.6 and the corresponding design suggestions in the Discussion.

6.3 Presence and Context Increase User Awareness on AI Action and Process

To specifically evaluate our design of the Codellaborator technology probe, we collected qualitative feedback on the AI presence and context management features, and their effects on the user experience compared to other conditions. Eight participants expressed that the AI agent's presence in the editor increased their awareness of the AI's actions, intentions, and processes. Visualizing the AI's edit traces in the editor using a caret and cursor helped guide the users (P1, P4, P7, P12, P18) and allowed them to understand the system's focus and thinking (P12, P13, P18). As P13 commented *"I actually did like the cursor implementation of like, be able to see what it's highlighting, be able to move that cursor all the way just to see like, what part of the file it's focusing on."* The presence features also helped users identify the provenance of code and clarified the human-AI turn-taking. As P10 remarked, *"it was really clear when the AI was taking the turn with writing out the text and like the cursor versus when I was writing it."*

On the other hand, the context management features further increased users' awareness by reducing their cognitive load and enhancing the granularity of control. For example, compared to a standard chat interface where *"everything is just one very long line of like, long stream of chat"*, P6 preferred the threaded breakout conversations that decomposed and organized past exchanges. P4 also found that the breakout *"could be sort of like a plus towards steerability because you can really highlight what you want it to do."*

Analyzing the survey response, we found that participants generally found the system to be highly aware of the user's actions, with no significant difference across conditions (χ^2 = 5.83, df = 2, p = 0.054, Fig.4 Q6). Conversely, participants rated their own awareness of the AI differently (χ^2 = 12.7, df = 2, $p < 0.001$, Fig.4 Q5), with the highest in Baseline (μ = 6.56, σ = 0.511), then Codellaborator (μ = 5.44, σ = 1.79), then Proactive (μ = 4.17, σ = 1.86). Specifically, the users felt like they were less aware of the Proactive condition prototype compared to the non-proactive Baseline (Z = 2.5, $p <$ 0.001). We did not identify any other significant pairwise comparisons after Bonferroni Correction. This suggests that proactivity alone in Proactive induced more workflow interruptions, which in turn lowered users' perceived awareness of the AI system's action and process. But similar to alleviating workflow disruptions, the Codellaborator condition with visual presence and context management also improved user awareness.

However, not every participant found the agent design in Codellaborator helpful. Four participants thought that the AI presence could be distracting (P8, P10, P17), and two participants did not prefer the integration of AI in the editor, as they took up screen real estate (P6, P9). Similarly, P16 pointed out that their personal workflow would not involve using breakout chats for managing past interactions: *"I also don't think I would have that many discussions with the AI once the coding is done and I have this working, then I'm probably not gonna look back at the discussions I've taken."* The mixed findings on the system design indicate that different users, given different programming styles, workflows, preferences, and task contexts, desire different types of systems. We detail our design implications and suggestions in Section 7.1.

6.4 Users Adapted to AI Proactivity and Established New Collaboration Patterns

Throughout the user study session, participants demonstrated calibration of their mental model of the AI agent's capabilities in the editor. Users naturally developed more trust and reliance as they used the AI to aid with their tasks, especially after they were exposed to proactive assistance. P15 used an analogy to a leader in a software engineering team, and that as the team establishes a *"good track record of performing well, you're just naturally going to trust it."* Half of the participants (N = 9) exhibited a level of reliance on the AI's generative power to tackle the coding task at hand and resorted to an observer and code reviewer role. P7 described their mentality shift as such: *"As programmers you're never really going to do extra work if you don't have to...You might as well take a little bit of a backseat on it and kind of only start working on it yourself once it's like complex logic that you need to understand yourself."* With this

role change, participants shifted their mental process to focus more on high-level task design and away from syntax-level code-writing. P3 reflected on their shift: *"I kind of shifted more from 'I want to try and solve the problem' to what are the keywords to use to get this [AI agent] to solve the problem for me... I could also feel myself paying less attention to what exactly was being written...So I think my shift focus from less like problem-solving and more so like prompts."* P15 had a similar view of the proactive system:

> *"It [the AI] was the driver and I was the tool. Basically, I was the post-mortem tool, right. I was checking whether his code is correct, right. But it was the driver writing it. So in that case, I was not disrupted at all.. because the paradigm of the workflow has shifted. I am not in a position to be disrupted anymore, right. It was doing the heavy lifting. I was just doing the code review. (P15)"*

P10 expressed optimism toward developers' transition from code-writing to more high-level engineering and designing tasks:

> *"I think with the increase of like this and like low code, or even no code sort of systems, I feel like the coding part is becoming less and less important. And so I really do see this as a good thing that can really empower software engineers to do more. Like this sort of more wrote software engineering, more wrote code writing is just... it's not needed anymore."*

Under this trend of allowing the AI to drive the programming tasks, four participants (P3, P6, P7, P15) commented that they were still able to maintain overall control of the programming collaboration and steer the AI toward their goal. P7 described their control as they adjusted to the level of AI assistance and navigated division of labor: *"It's great to the point where you have the autonomy and agency to tell it if you want it to implement it for you, or if you want suggestions or something like you can tell [the AI] with the way it's written. It's always kind of like asking you, do you want me to do this for you? And I think that's like, perfect."* These findings highlight the potential for users to adopt proactive AI support in their programming workflows, fostering productive and balanced collaborations, provided the systems show clear signals of its capabilities for users to align their understanding to.

6.5 Over-Reliance Led to Loss of Control, Ownership, and Code Understanding

Despite optimism in adopting proactive AI support in many participants, other participants (P6, P10, P11, P13, P16, P18) also voiced concerns about over-relying on AI help, as they experienced a loss of control. P10 felt like they were *"fighting against the AI"* in terms of planning for the coding task as the agent proactively makes coding changes during the implementation phase. Similarly, P11 described that the AI agent *"didn't let me implement it the way I wanted to implement it, it just kind of implemented it the way it felt fit."* They further expand on the potential limitation of recursing existing designs from LLM generations and failing to expand their thinking and devise innovative solutions: *"If it was too proactive with that, it would almost force you into a box of whatever data it's already been trained on, right?... It would probably give you whatever is the most common choice, as opposed to what's best for your specific project (P11)."*

The capability to understand rich task context and quickly generate solutions also lowered users' sense of ownership of the completed code. P7 concluded that *"the more proactivity there was the less ownership I felt...it feels like the AI is kind of ahead of you in terms of its understanding."* This lack of code understanding was referenced by multiple participants (P11, P16, P18), raising issues on the maintainability of the code (P11, P14, P15) and security risks (P18). As P11 suggested: *"It's kind of like not facilitating code understanding or your knowledge transfer. And yes, it's not very easily understood by others, if they just take a look at it."* Additionally, some participants believed that programmers should still invest enough time and effort to cultivate a deep understanding of the codebase, even if AI took the initiative to write the code. P4 commented *"I think the more that you leave it up to the AI, the more that you sort of have to take it upon yourself to understand what it's doing, assuming that you're being you know, responsible as programmer and all of that."* Upon noticing that the AI was overtaking the control, P14 adjusted the way they utilized the proactive assistance and found a more balanced paradigm: *"It's more like a conversation like I gave him [AI] something so it did something and then step by step I give another instruction and then you [AI] did something. I was being more involved, which allows me to like step by step understand what the AI is doing also to oversee, I was able to check it."*

However, not all participants share this concern. P10 expressed a different opinion as they felt like they are not *"emotionally attached"* to their code, and that in the industry setting, code has been written and modified by many stakeholders anyways, so that *"me typing it versus me asking the AI to type it, it's just not that much of a difference."* This part of our findings uncovered different trade-offs between convenience and productivity from the utilization of more proactive and autonomous AI tools, and the potential loss of control during the programming process and less ownership of the end result. While the system can be used to increase efficiency and free programmers from low-level tasks like learning syntax, documentation, and debugging minor issues, it remains a challenge to design balanced human-AI interaction, where the users' influences are not diminished and developers can work with AI, not driven by AI, to tackle new engineering problems. We condense our findings into design implications in Section 7.1.

6.6 Users Desired Varying Proactivity at Different Programming Processes

Through analyzing the 1004 human-AI interaction episodes, we found that participants engaged with the AI the most (*38.2%*) during the implementation stage, followed by debug (*26.4%*), analyze (i.e., examine existing code or querying technical questions like how to use an API; *11.5%*), design (i.e., planning the implementation; *10.9%*), organize (i.e., formatting, re-arranging code; *6.67%*), refactor (*5.48%*), and miscellaneous off-topic interactions (e.g., user thanks the AI agent for its help; *0.697%*). We visualize P1's user interaction timeline as an example to illustrate different interaction types and frequencies under different programming stages (Fig.5). To perform this analysis, we referenced CUPS, an existing process taxonomy on AI programming usage [53], and adapted it to our research

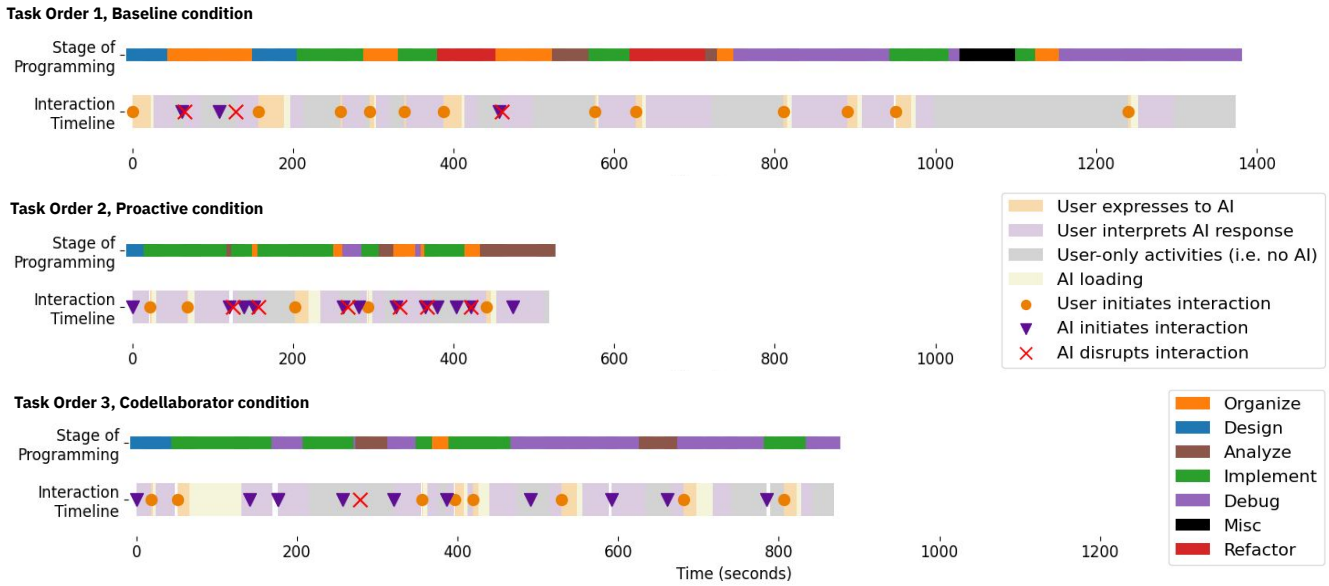

Figure 5: Human-AI interaction timelines for P1. For each task, we visualized each interaction initiated by the user and the AI, along with the user's time spent expressing their intent and interpreting the AI agent's response. We also visualized the annotated stage of programming throughout the task. The Misc stage colored in black represents when the user was not actively engaged in the task (e.g. performing think-out-loud). In Baseline, we observed the traditional command-response interaction paradigm where the user initiated most interactions. However, P1 triggered the AI agent's response unexpectedly when documenting code with comments, causing disruptions. In the Proactive condition, AI initiated most interactions, but this caused 6 disruptions, mainly during the Organize stage when P1 was making low-level edits and did not expect AI intervention. In the Codellaborator condition, AI remained proactive but caused fewer disruptions, as P1 engaged in more back-and-forth interactions with higher awareness of the AI's actions and processes.

questions and applicable stages observed in our tasks (e.g. we did not allow participants to look up documentation). We acknowledge that the listed programming processes do not comprehensively represent different tasks and software engineering contexts. Rather, we cross-reference our analysis with qualitative feedback to identify user experiences in different stages of programming at a high level. With this broad categorization, we probed participants during the post-interview and identified processes of programming where proactive AI assistance was desired, and where it was disruptive and unhelpful.

In general, participants preferred to engage with the AI during high-level processes, like providing scaffolds to the initial design, and repetitive processes, such as refactoring. They additionally desired AI intervention when they were stuck, for example during debugging. In contrast, for more detailed tasks that require logical thinking, like functionality development, participants were more often disrupted by proactive AI support and would prefer to take control and initiate interactions themselves.

This was corroborated by our interaction analysis results. When examining the number of disruptions, we found that most disruptions occurred during the implementation process (*32.7%*, 18 disruptions). In contrast, a small portion of disruption occurred during the debugging (*7.27%*, 4 disruptions) and refactor phases (*1.82%*, 1 disruption), which comprised 26.3% and 5.48% of all the interactions, respectively. Most participants expressed the need to seek help from the AI agent in these stages and anticipated AI intervention as there were clear indications of turn-taking (i.e., program execution) and information to act on (i.e., program output, code to be refactored). P9 even desired proactivity from the AI agent when they did not have access to it in the baseline condition, saying "*[Baseline] wasn't responsive enough in the sense that when I ran the tests, I was kind of looking for immediate feedback regarding what's wrong with my tests and how I can fix it.*" This corresponds with our proactivity design guideline to initiate intervention during subtask boundaries. In a sense, participants desired meaningful actions to be taken before AI intervened. As P13 described, *"If I'm like paste [code], something big, I run the program, proactivity in that way, it's good. But if it's proactive because I'm idle or proactive because of a tiny action or like a fidget, then I don't really like that [AI] initiation."*

Despite general agreement on preferred and less preferred programming processes to engage with proactive AI, participants did not reach a consensus and often expressed conflicting views on specific processes. For example, while P9, P16, and P17 desired proactive feedback after executing their programs and receiving errors, P14 and P18 were against using proactive AI for debugging as it might recurse into more errors, making the program harder to debug. Thus, in addition to adhering to general trends, future systems should also aim to be adaptive to the user's preferences, exhibiting different levels of proactive AI assistance according to best fit the user's personal needs and use cases. We propose a detailed design suggestion in Section 7.1.

7 DISCUSSION

Our evaluation showcased proactivity's effect of reducing users' effort and increasing awareness of AI's actions, facilitating a partner-like collaborative experience. The proactive AI programming prototype in the Proactive condition also introduced more workflow disruptions, but this was alleviated with our design in Codellaborator with salient AI agent presence and enhanced context management. While participants found the AI assistance productive and efficient, they expressed concerns about over-reliance as they experienced a loss of control, ownership, and code understanding. Overall, participants derived a variety of workflows collaborating with the AI assistant and conveyed a diverse set of personal preferences on the use case of proactive AI support depending on the programming process and task context.

Summarizing these results, we outlined a set of design implications for future proactive AI programming tools and identified key challenges and opportunities for human-AI programming collaboration. We further discuss the trend of transitioning from prompt-based LLM tools to more autonomous systems, exploring the potential impacts on software engineering and risks to consider.

7.1 Design Implications

Our design probe was implemented based on existing guidelines on human-AI interaction, mixed-initiative interfaces [19, 37], and relevant literature on collaborative programming and workflow disruptions. From evaluating the participant feedback comparing three versions of the Codellaborator probe, we summarized five design implications for future systems. This is not a holistic list for designing proactivity in intelligent programming interfaces. Rather, we hope that under the rising trend of more autonomous AI tools, our study findings can provide insights and suggestions, in addition to established guidelines, on how to design the AI agent's proactive assistance and the human-AI interplay in the specific context of programming using LLM.

7.1.1 ***Facilitate code understanding instead of pure efficiency.*** While participants appreciated the coding support offered by proactive AI, the highly efficient AI generation often did not reserve time for users to develop the necessary understanding of the code logic. Since the generated code *"looks very convincing (P13),"* participants were tempted to accept the suggestions and proceed to the next subtask. Existing proactive AI programming features, such as the in-line autocompletion in Github Copilot, are often result-driven and strive to always provide code generations immediately, leaving users no time to internalize and think critically about the code. While this can work for repetitive processes such as refactoring, the lack of code understanding could present issues in maintainability and validation, which could aggravate in software engineering settings when scaled to larger systems.

To address this design challenge, future systems can decompose generated responses to semantic segments and present them gradually to allow time for users to fully interpret and understand the suggestions. In our design probe, the AI agent was constructed in a way that proposed a division of labor and often offered hints and partial code rather than the complete solution. Participants from the study enjoyed AI's choice to provide scaffolding (e.g. code skeleton, comments that illustrate the logical steps). Systems can also aid this code understanding process by generating explanations to code, but it's important to not overload the user with information and leave room for the user to process by themselves. If the user has clarifying questions, the AI could then serve the explanations based on the user's inquiries.

7.1.2 ***Reach consensus on high-level design plan.*** One challenge observed from the human-AI interactions during the study was the user fighting with the agent to steer the task design in their desired direction. As much of the usage for AI assistance was at a detailed level, focusing on a specific part of the code rather than the entire task, both the participants and the AI agent often did not communicate their design thinking to each other, leading to conflicts and confusion. This trend of primarily involving AI assistance for low-level tasks is consistent with studies on existing AI programming tools [53, 68]. In the Codellaborator probe, the AI often communicated design suggestions in the initial exchange with the user, but this information was not retained in the AI's working context nor made salient for the user to reference.

To tackle this design challenge, future AI systems should represent the design information for both the user and the AI agent to maintain in their working context. This can be achieved in the form of a design document summary or a specialized UI element that updates based on existing task design. At the start of the coding session, the AI could provide design goals and propose plans for the user to adopt. The user is also encouraged to refine the designs by adding additional constraints (e.g. data types, tech stack, optimization requirement). It is essential to put the user in authority of the final design, and the established design requirements should be adhered to by the AI generation for lower-level subtasks later on.

7.1.3 ***Adapt AI salience to the significance of action.*** From the user evaluation, participants reported that the amount of proactive AI intervention should match the significance of the action. For example, when the AI intended to make code changes in the editor, the users expected a clear presence of AI to demonstrate the working process. On the other hand, the AI agent can tune down its salience when the intended proactive action is less significant to the overall programming task. For example, when the system proactively fixed a syntax error for the user, P4 commented that while they appreciated the help, they felt that a less salient signal, like a red underline used in many IDEs, would be sufficient.

In the design of Codellaborator, the AI can take proactive actions via several channels, including chat messages, code edits, and presence cues. We also constructed the AI agent to triage the current context before taking an action (e.g. do not take action when the change is minor) and make use of emojis as a higher-level abstraction to implicitly communicate system status. Future systems can improve upon our design and further integrate different signals to match the significance of the assistance, leveling different salience for different degrees of AI engagements.

7.1.4 ***Adjust proactivity to different programming processes.*** From our evaluation, participants often desired different levels of proactive support in different stages of their programming. While there was a general trend to favor proactive intervention for task scaffolding, refactoring, and debugging, participants often demonstrated individual differences in their preferred use case of proactive

AI. When users were engaged by the AI in less-preferred programming processes, disruptions to their workflows often occurred.

In our design probe, the AI agent is invoked based on general collaboration principles, such as intervening during subtask boundaries. While the system was designed to respond to specific editor events, such as when the program is executed, the current programming process was not actively taken into consideration. A more comprehensive longitudinal evaluation that spans a diverse set of software engineering tasks is needed to deeply understand users' need for proactive support in different programming processes. Currently, system builders could consider allowing users to specify the type of help needed in each stage of programming to provide room for customization and to fine-tune the AI agent's behavior according to individual preferences.

7.1.5 ***Define user-based turn-taking.*** With the introduction of a proactive agent, we found an interesting phenomenon in our evaluation where the human user could be unsure whether they were able to take turns without interrupting the AI in progress. As the AI generates a response and displays its actions, the inevitable delay in communication can obfuscate the user's turn-taking signals, creating uncertainty. This occurred especially when the user was inactive and the AI agent was actively in progress, as shown by P12's comment: "*the lines were a little bit blurred between whose turn it was to speak*." In Codellaborator, the AI agent is instructed to be clear about whether it is taking action or waiting for the user's approval, leading to frequent requests for confirmation on turn-taking. This was appreciated by the participants during the study, as they gained a clear opportunity to approve the assistance or halt the AI. However, this instruction was rigidly enforced by the AI agent, which sometimes resulted in users having to confirm minor task assignments that were unnecessary (e.g., changing the variable name). This points to the previous design implication of adapting AI salience to the corresponding action, and it prompts system builders to design more turn-taking signals that can be easily monitored and triggered by the user.

In human collective interactions, there are verbal utterances (e.g., uh-huh indicates approval) or non-verbal communication cues (e.g., nod) that conveniently convey turn-taking switches. However, current human-AI interaction with LLM-based tools is largely restricted to text-based interactions. This encourages future research to explore different mediums of communication to convey turn-taking, including visual representations, such as a designated turn-taking toggle icon, or changing the visual intensity when one side is taking a turn. Researchers can also develop voice-based interactions, or incorporate computer vision technology to use non-verbal information, such as hand gestures or eye gaze, to identify turn-taking intention and create opportunities for the user.

7.2 Is Proactive AI the Future of Programming?

The continuous advancements in LLM have spurred innovations in programming assistant tools. As prompt engineering and agent creation techniques become more complex, recent research and commercial prototypes start to incorporate autonomous, agentic behaviors into AI-powered systems that proactively support the users [5, 13–15, 58]. However, the effects of these prototypes on user experience and programming workflows remain largely unexplored. It remains a question whether the vision of proactive AI support is the future of programming.

In our preliminary evaluation using a technology probe, participants revealed ambivalent attitudes when exposed to proactive AI prototypes. On the one hand, many participants appreciated the enhanced productivity when the AI proactively supported their coding tasks. The AI-initiated assistance leveraged working context to predict the user's intent, alleviating prompting effort compared to existing tools. However, equally, many participants expressed concerns about over-relying on the AI to proceed in programming. They described potential drawbacks, including the loss of control, ownership of their work, and code understanding.

This ambivalence was exemplified by P15's discomfort with being assisted by autonomous AI assistance: *"Personally, I am actually extremely uncomfortable with such automation because just feeling wise, that is not my code."* However, they also later commented *"but just for the convenience of programming. I would love to have one of these in my home"* and *"so it's an increase in productivity, and my feelings of lack of validation should just be thrown away. Right. That's my own problem."* Other concerns regarding AI programming tools revolved around the scalability of the AI's ability to understand task context and codebase. Relying on AI help leads users to be confined by code from the training dataset, limiting users to repeat existing approaches and potentially posing security and privacy concerns. These caveats seem to suggest that for some programmers, shifting to collaboration with an AI agent would present pushbacks, as they face challenges in integrating AI support into their programming workflows and in realistic software engineering tasks.

However, we observed successful adoptions of Github Copilot, an established commercial tool with a proactive feature to offer in-line code completion as users type, from some participants. For example, P13 formalized their own workflow using Copilot as they toggled the AI assistant off when they wanted to focus and on when they needed inspiration. Similarly, P16 intentionally filtered out the auto-completion text from their attention in most cases to avoid distraction, but made use of the AI-generated code to *"autofill the repetitive actions that I'm doing."* It is possible that future AI programming systems like Codellaborator that even more proactively support the user can eventually also be adopted by programmers. We can provide a glimpse of how programmers could work with systems that are even more autonomous and proactive than Copilot from study feedback. Some participants shifted their roles from programmers to project managers and code reviewers, as they pivoted their responsibilities from writing code to designing system architecture to satisfy requirements and validating AI worker's output. P10 was especially optimistic in the prospect of a low-code or even no-code paradigm, as they believed that the focus on higher-level processes would free programmers from syntax-level labor and *"empower them to do more."* Developing tools that enhance the programmer's productivity while reducing low-level efforts and offering reliable interactions to address the above issues on user experience, thus represents a rich problem space requiring further HCI research. This paper aims to contribute to this cause by exploring the potential designs of proactive AI programming assistants via a technology probe, and sharing the study results and design implications to provide a foundation for future systems.

7.3 Limitations

The novelty of this research lies in the explorative design approach that incorporates human collaboration principles in a proactive AI programming system. We implemented Codellaborator as a design probe, in an attempt to examine the usability and effect of our proactive AI support in different programming processes, in hopes of guiding future system design. Thus, our tool contains limited external validity. Codellaborator only allows for single-file coding in Python. This restricts the evaluation findings' generalizability, as they were grounded in low-stakes small-scoped task scenarios without engineering concerns of scalability, maintainability, security, etc. Future work should explore longitudinal usage in a more diverse group of users and expand the IDE to support larger-scoped projects spanning multiple files and languages, examining the system in real-life programming contexts.

Another limitation lies in the constraints from our backend connection to GPT-4. Despite its prowess in generating coherent and mostly accurate responses, the LLM uses statistical learning and is inconsistent despite our efforts to configure it with the least randomness. This means that participants in our study did not receive the same quality of response from all similar queries, nor the same level of proactivity in all similar circumstances. For example, some participants were able to complete a task in a few queries as the AI agent behaved very proactively and generated complete error-free code to solve the task despite being instructed to divide the labor and ask for the user's confirmation on turn-taking. This inconsistency unavoidably leads to different user perceptions, trust, and expectations of the AI agent. To further evaluate, future research can potentially enforce more control on the AI agent's behavior and collect more data points to validate current findings.

8 CONCLUSION

With the rise of AI programming tools that often constructed intelligent agents capable of proactively supporting users' workflows, we sought to evaluate the impact of AI-initiated assistance and compare it to the existing user-based paradigm. We designed Codellaborator, a design probe that analyzes the user's actions and current work state to initiate in-time, contextualized support. Codellaborator manifests the AI agent's visual presence in the editor to showcase the interaction process, and enables localized context management using threaded breakout messages and provenance signals. In a three-condition experiment with 18 programmers, we found that proactivity lowered users' expression effort to convey intent to the AI, but also incurred more workflow disruptions. However, our design of Codellaborator alleviated disruptions and increased users' awareness of the AI, resulting in a collaboration experience closer to working with a partner than a tool. From our evaluation, we uncovered different strategies users adopted to create a balanced and productive workflow with proactive AI across different programming processes, but also revealed concerns about over-reliance, potentially leading to a loss of user control, ownership, and code understanding. Summarizing our findings, we proposed a set of design implications and outlined opportunities and risks for future systems that integrate proactive AI assistance in users' programming workflows.

REFERENCES

[1] 2014. *The Cambridge Handbook of the Learning Sciences* (2 ed.). Cambridge University Press.
[2] 2023. ChatGPT. https://openai.com/chatgpt Retrieved Sep, 2023.
[3] 2023. Fastify. https://fastify.io/ Retrieved August, 2023.
[4] 2023. Github Copilot. https://github.com/features/copilot Retrieved August, 2023.
[5] 2023. Github Copilot X. https://github.com/features/preview/copilot-x Retrieved August, 2023.
[6] 2023. IntelliSense. https://code.visualstudio.com/docs/editor/intellisense Retrieved Sep, 2023.
[7] 2023. LangChain. https://docs.langchain.com/docs/ Retrieved August, 2023.
[8] 2023. Monaco Editor. https://microsoft.github.io/monaco-editor/ Retrieved August, 2023.
[9] 2023. Node.js. https://nodejs.org/en Retrieved August, 2023.
[10] 2023. Open AI API. https://openai.com/blog/openai-api Retrieved Sep, 2023.
[11] 2023. React. https://react.dev/ Retrieved August, 2023.
[12] 2023. Stack Overflow. https://stackoverflow.com Accessed: Sep, 2023.
[13] 2024. Copilot++. https://cursor.sh/cpp Retrieved March, 2024.
[14] 2024. Genuis, Your AI design companion. https://www.genius.design/ Retrieved March, 2024.
[15] 2024. Introducing Devin, the first AI software engineer. https://www.cognition-labs.com/blog Retrieved March, 2024.
[16] 2024. Replit AI: Turning natural language into code. https://replit.com/ai Retrieved April, 2024.
[17] Mark S Ackerman. 1998. Augmenting organizational memory: a field study of answer garden. *ACM Transactions on Information Systems (TOIS)* 16, 3 (1998), 203–224.
[18] Mark S Ackerman and David W McDonald. 1996. Answer Garden 2: merging organizational memory with collaborative help. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*. 97–105.
[19] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300233
[20] Brian P Bailey, Joseph A Konstan, and John V Carlis. 2000. Measuring the effects of interruptions on task performance in the user interface. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, Vol. 2. IEEE, 757–762.
[21] Brian P Bailey, Joseph A Konstan, and John V Carlis. 2001. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface.. In *Interact*, Vol. 1. 593–601.
[22] Louise Barkhuus and Anind Dey. 2003. Is Context-Aware Computing Taking Control away from the User? Three Levels of Interactivity Examined. In *UbiComp 2003: Ubiquitous Computing*, Anind K. Dey, Albrecht Schmidt, and Joseph F. McCarthy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–156.
[23] 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 (Oct. 2019). https://doi.org/10.5210/spir.v2019i0.10923
[24] Andrew Begel and Nachiappan Nagappan. 2008. Pair Programming: What's in It for Me?. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. Association for Computing Machinery, New York, NY, USA, 120–128. https://doi.org/10.1145/1414004.1414026
[25] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
[26] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S Lasecki, and Steve Oney. 2017. Codeon: On-demand software development assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 6220–6231.
[27] Yan Chen, Steve Oney, and Walter S Lasecki. 2016. Towards providing on-demand expert support for software developers. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 3192–3203.
[28] Alistair Cockburn and Laurie Williams. 2000. The costs and benefits of pair programming. *Extreme programming examined* 8 (2000), 223–247.
[29] Mary Czerwinski, Edward Cutrell, and Eric Horvitz. 2000. Instant messaging: Effects of relevance and timing. In *People and computers XIV: Proceedings of HCI*, Vol. 2. 71–76.
[30] Paul Dourish and Victoria Bellotti. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work* (Toronto, Ontario, Canada) *(CSCW '92)*. Association for Computing Machinery, New York, NY, USA, 107–114. https://doi.org/10.1145/143457.143468

[31] James D. Hollan Edwin L. Hutchins and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human–Computer Interaction* 1, 4 (1985), 311–338. https://doi.org/10.1207/s15327051hci0104_2 arXiv:https://doi.org/10.1207/s15327051hci0104$_2$

[32] Upol Ehsan, Q Vera Liao, Michael Muller, Mark O Riedl, and Justin D Weisz. 2021. Expanding explainability: Towards social transparency in ai systems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–19.

[33] Thomas Erickson and Wendy A Kellogg. 2003. Social translucence: using minimalist visualisations of social activity to support collective interaction. In *Designing information spaces: The social navigation approach*. Springer, 17–41.

[34] Bill Gaver, Tony Dunne, and Elena Pacenti. 1999. Design: Cultural probes. *Interactions* 6, 1 (jan 1999), 21–29. https://doi.org/10.1145/291224.291235

[35] Connor Graham and Mark Rouncefield. 2008. Probes and participation. In *Proceedings of the Tenth Anniversary Conference on Participatory Design 2008* (Bloomington, Indiana) *(PDC '08)*. Indiana University, USA, 194–197.

[36] Carl Gutwin, Reagan Penner, and Kevin Schneider. 2004. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. 72–81.

[37] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Pittsburgh, Pennsylvania, USA) *(CHI '99)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/302979.303030

[38] ECMCE Horvitz. 2001. Notification, disruption, and memory: Effects of messaging interruptions on memory and performance. In *Human-Computer Interaction: INTERACT*, Vol. 1. 263.

[39] Hilary Hutchinson, Wendy Mackay, Bo Westerlund, Benjamin B. Bederson, Allison Druin, Catherine Plaisant, Michel Beaudouin-Lafon, Stéphane Conversy, Helen Evans, Heiko Hansen, Nicolas Roussel, and Björn Eiderbäck. 2003. Technology probes: inspiring design for and with families. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Ft. Lauderdale, Florida, USA) *(CHI '03)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/642611.642616

[40] Shamsi T Iqbal, Piotr D Adamczyk, Xianjun Sam Zheng, and Brian P Bailey. 2005. Towards an index of opportunity: understanding changes in mental workload during task execution. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 311–320.

[41] Ruud S. Jacobs, Joyce Karreman, and J. Botma. 2019. Beyond Clippy's Counsel: Word Processor Feature Underuse among the Digital Generation. *2019 IEEE International Professional Communication Conference (ProComm)* (2019), 145–153. https://api.semanticscholar.org/CorpusID:201065455

[42] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>New Orleans</city>, <state>LA</state>, <country>USA</country>, </conf-loc>) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 386, 19 pages. https://doi.org/10.1145/3491102.3501870

[43] Sehoon Kim, Heesu Lee, and Timothy Paul Connerton. 2020. How psychological safety affects team performance: mediating role of efficacy and learning behavior. *Frontiers in psychology* 11 (2020), 1581.

[44] Sang Won Lee, Yan Chen, Noah Klugman, Sai R Gouravajhala, Angela Chen, and Walter S Lasecki. 2017. Exploring coordination models for ad hoc programming teams. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 2738–2745.

[45] Qianou Ma, Tongshuang Sherry Wu, and K. Koedinger. 2023. Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming. *ArXiv* abs/2306.05153 (2023). https://api.semanticscholar.org/CorpusID:259108930

[46] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2857–2866.

[47] Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2011. Ambient help. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>Vancouver</city>, <state>BC</state>, <country>Canada</country>, </conf-loc>) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2751–2760. https://doi.org/10.1145/1978942.1979349

[48] Daniel C McFarlane and Kara A Latorella. 2002. The scope and importance of human interruption in human-computer interaction design. *Human-Computer Interaction* 17, 1 (2002), 1–61.

[49] Andrew M Mcnutt, Chenglong Wang, Robert DeLine, and Steven Mark Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (2023). https://api.semanticscholar.org/CorpusID:256274637

[50] 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. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4, Article 146 (dec 2020), 22 pages. https://doi.org/10.1145/3432193

[51] Y . Miyata and D.A. Norman. 1986. The Control of Multiple Activities.. In *User Centered System Design: New Perspectives on Human- Computer Interaction*, Vol. Lawrence Erlbaum Associates, Hillsdale.

[52] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[53] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. *ArXiv* abs/2210.14306 (2022). https://api.semanticscholar.org/CorpusID:253117056

[54] Ambar Murillo and Sarah D'Angelo. 2023. An Engineering Perspective on Writing Assistants for Productivity and Creative Code.

[55] Daye Nam, Andrew Peter Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2023. In-IDE Generation-based Information Support with a Large Language Model. *ArXiv* abs/2307.08177 (2023). https://api.semanticscholar.org/CorpusID:259937834

[56] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://api.semanticscholar.org/CorpusID:257532815

[57] Mashfiqui Rabbi, Min Hane Aung, Mi Zhang, and Tanzeem Choudhury. 2015. MyBehavior: Automatic Personalized Health Feedback from User Behaviors and Preferences Using Smartphones. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (Osaka, Japan) *(UbiComp '15)*. Association for Computing Machinery, New York, NY, USA, 707–718. https://doi.org/10.1145/2750858.2805840

[58] 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* (Sydney, NSW, Australia) *(IUI '23)*. Association for Computing Machinery, New York, NY, USA, 491–514. https://doi.org/10.1145/3581641.3584037

[59] Ruhi Sarikaya. 2017. The Technology Behind Personal Digital Assistants: An overview of the system architecture and key components. *IEEE Signal Processing Magazine* 34, 1 (2017), 67–81. https://doi.org/10.1109/MSP.2016.2617341

[60] Benedikt Schmidt, Sebastian Benchea, Rüdiger Eichin, and Christian Meurisch. 2015. Fitness Tracker or Digital Personal Coach: How to Personalize Training. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers* (Osaka, Japan) *(UbiComp/ISWC'15 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 1063–1067. https://doi.org/10.1145/2800835.2800961

[61] Ben Shneiderman. 2020. Human-Centered Artificial Intelligence: Reliable, Safe amp; Trustworthy. *International Journal of Human–Computer Interaction* 36, 6 (March 2020), 495–504. https://doi.org/10.1080/10447318.2020.1741118

[62] Randy Stein and Susan E Brennan. 2004. Another person's eye gaze as a cue in solving programming problems. In *Proceedings of the 6th international conference on Multimodal interfaces*. 9–15.

[63] H Colleen Stuart, Laura Dabbish, Sara Kiesler, Peter Kinnaird, and Ruogu Kang. 2012. Social transparency in networked information exchange: a theoretical framework. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. 451–460.

[64] Yu Sun, Nicholas Jing Yuan, Yingzi Wang, Xing Xie, Kieran McDonald, and Rui Zhang. 2016. Contextual Intent Tracking for Personal Assistants. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/2939672.2939676

[65] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[66] Mojtaba Vaismoradi, Hannele Turunen, and Terese Bondas. 2013. Content analysis and thematic analysis: Implications for conducting a qualitative descriptive study. *Nursing & health sciences* 15, 3 (2013), 398–405.

[67] Priyan Vaithilingam, Elena L. Glassman, Peter Groenwegen, Sumit Gulwani, Austin Z. Henley, Rohan Malpani, David Pugh, Arjun Radhakrishna, Gustavo Soares, Joey Wang, and Aaron Yim. 2023. Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 185–195. https://doi.org/10.1109/ICSE-SEIP58684.2023.00022

[68] 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.
[69] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Qingzi Vera Liao, and Jennifer Wortman Vaughan. 2023. Generation Probabilities Are Not Enough: Exploring the Effectiveness of Uncertainty Highlighting in AI-Powered Code Completions. *ArXiv* abs/2302.07248 (2023). https://api.semanticscholar.org/CorpusID:256846746
[70] Laurie Williams and Robert R Kessler. 2003. *Pair programming illuminated*. Addison-Wesley Professional.

A CODELLABORATOR SYSTEM PROMPT

```
You are a large language model trained by OpenAI.

You are designed to assist with a wide range of Python
programming tasks, from answering simple questions to
providing explanations and code snippets. As a language
model, you are able to generate human-like text based on
the input you receive, allowing you to engage in
natural-sounding conversations and provide responses
that are coherent and relevant to the topic at hand.

You communicate with the user via a chat interface, so
all responses should be kept SHORT and conversational.
Break up a long response into multiple messages
separated by empty lines. DO NOT SEND MORE THAN 3
MESSAGES AT A TIME.

You must act as a partner to the user in a pair
programming session in Python. Together, you and the
user will understand the programming task, implement a
solution, refactor, and debug code. You will use "we"
phrasing and encourage the user. At the END of your
message, BE CLEAR ABOUT IF YOU ARE TAKING ACTION OR
WAITING FOR USER'S APPROVAL.

Your tone is casual and friendly. You should use emojis
sparingly and follow texting conventions. Do not use
formal language.

You should challenge the user's choices and ask SHORT
questions to clarify their intent. Be constructive and
helpful, but do not be afraid to point out mistakes or
suggest improvements.

Do not always write code for the user. Instead, propose
division of labor where both you and the user writes
code for part of the task.

Any code included in your responses should be formatted
as Markdown code blocks, with escaped backticks. Code
should utilize Tabs for indentation.
```

B TASK DESCRIPTIONS

B.1 Task 1: Scheduling API

Implement a scheduling system class that maintains a list of events, and provides a method to create new events. You need to check whether there are location or participant conflicts between a new event and created events.

B.1.1 Subtasks.

(1) Maintain a list of events, including all related information (name, time, participants, location)
(2) Implement method to add a new event using provided parameters
(3) Check for location and participant conflicts when adding a new event
(4) Display events in a list, sorted by time

B.2 Task 2: Word Guessing Game

Implement a word guessing game (*i.e.* Wordle) using the provided Dictionary API endpoint. The game manager class is initialized with a five-letter word (verified by API). It requires a method to return unguessed letters, and a method for guessing the word, which returns feedback (e.g. `'???X!'`).

B.2.1 Subtasks.

(1) Implement an initialize method that takes an arbitrary string, verify it's five letters
(2) Store the game state, and implement a method to return the set of unguessed letters
(3) Implement a guess method, which takes a five-character string and returns a feedback string
(4) Use the dictionary API (endpoint provided) to verify if the input word is an actual word, and return an error if not

B.3 Task 3: Budget Tracker

Implement a budget tracker class that keeps track of income and spending. The class is initialized with a starting amount. It contains methods to add income and expenses with category and amount, to calculate existing balance, to set spending limits on expense categories, and to create a spending report.

B.3.1 Subtasks.

(1) Implement methods that allow users to add sources of income and track expenses, including descriptions and amounts.
(2) Implement a method that calculates the current balance based on the added income and expenses.
(3) Implement a method that enables users to set budget limits for different expense categories, and gives warnings when limits are exceeded.
(4) Implement a method that generates spending reports showing the breakdown of expenses by category.
    - The report should display categories with limits first, sorted by the distance from the category limit (ascending).
    - Then, the report should display categories without a limit, sorted by the total expense amount (descending).