Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild

Aayush Kumar*, Yasharth Bajpai*, Sumit Gulwani†, Gustavo Soares†, Emerson Murphy-Hill‡
*Microsoft, Bengaluru, India; †Microsoft, Redmond, WA, USA; ‡Microsoft, Sunnyvale, CA, USA
{t-aaykumar, ybajpai, sumitg, gustavo.soares, emerson.rex}@microsoft.com

***Abstract*—Software Engineering Agents (SWE agents) can autonomously perform development tasks on benchmarks like SWE Bench, but still face challenges when tackling complex and ambiguous real-world tasks. Consequently, SWE agents are often designed to allow interactivity with developers, enabling collaborative problem-solving. To understand how developers collaborate with SWE agents and the barriers they face in such interactions, we observed 19 developers using an in-IDE agent to resolve 33 open issues in repositories to which they had previously contributed. Participants successfully resolved about half of these issues, with those solving issues incrementally having greater success than those using a one-shot approach. Participants who actively collaborated with the agent and iterated on its outputs were also more successful, though they faced challenges in trusting the agent's responses and collaborating on debugging and testing. Our findings suggest that to facilitate successful collaborations, both SWE agents and developers should actively contribute to tasks throughout all stages of the software development process. SWE agents can enable this by challenging and engaging in discussions with developers, rather than being conclusive or sycophantic.**

***Index Terms*—Agent-based systems, software development tools, collaboration, communication, developer experience**

I. Introduction

Software developers are increasingly incorporating generative artificial intelligence (AI) tools into their work to boost productivity [1]. Initially, tools like inline code completions and chat assistants have aided developers by generating code snippets and providing on-demand answers. More recently, there has been a surge in the adoption of *software engineering agents* (SWE agents)—tools specifically designed to autonomously perform complex development tasks with the capability to iteratively refine their outputs based on environmental feedback. They not only generate code, but can also execute tool calls and act on these tools' results [2].

SWE agents have shown promising results on established benchmarks of software development tasks, such as fully autonomously solving many of the GitHub issues in SWE-Bench [3]. However, they still struggle to solve certain issues due to factors such as the complexity and ambiguity of the issues, lack of tacit knowledge, and limited access to the full development environment [4], [5]. To overcome these limitations, many SWE agents are designed to interact with developers, enabling a collaborative approach to problem-solving. This interaction allows developers and agents to share responsibilities such as understanding the codebase, localising, coding, debugging, testing, and reviewing each other's work.

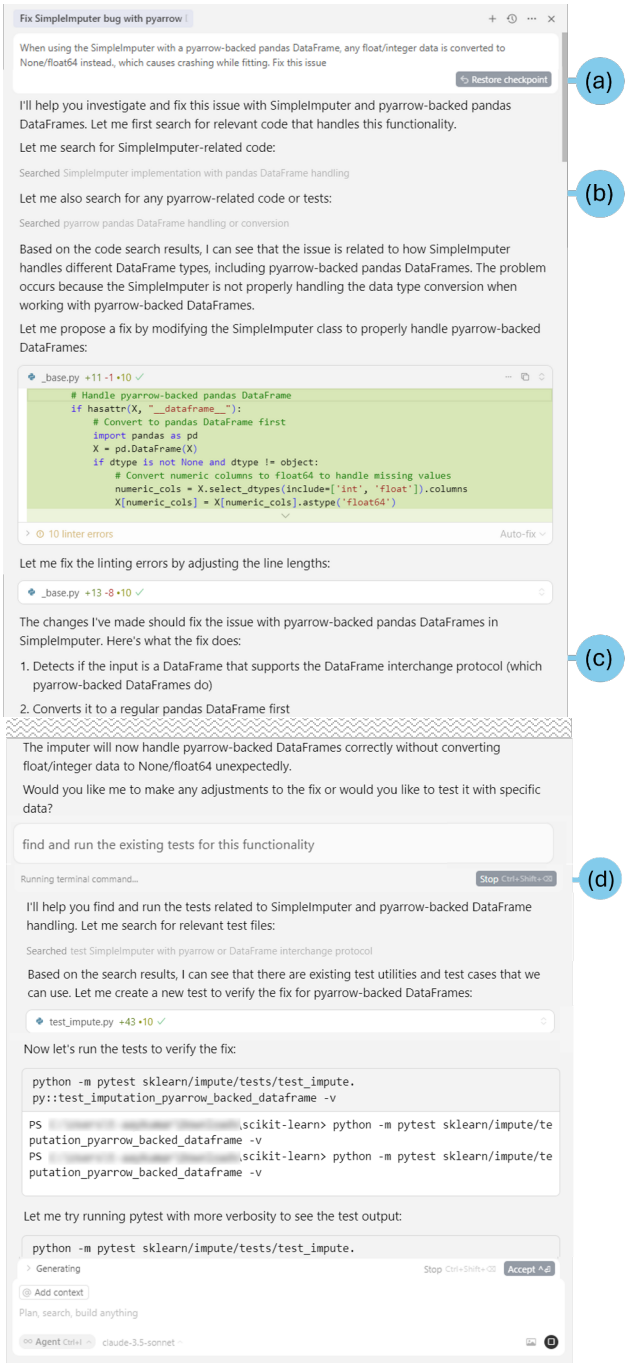

Fig. 1: Example usage of Cursor Agent (not from our study)

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00043

Figure 1 shows an example of one such SWE agent – Cursor Agent in the Cursor integrated development environment (IDE) – responding to a developer's prompt that has been copied from a GitHub bug report [6] (this specific bug was chosen purely to illustrate the usage of in-IDE SWE agents and was not part of our study). In the top part of the figure, the agent searches the codebase, describes the issue based on this search, and tries to implement a fix. After prompting, the agent then adds a unit test and runs it. While reviewing the agent's changes, the developer notices a functional call erroneously changed by the agent (Figure 2, bottom), and rejects that part of the change, before reviewing the remaining 5 changes.

There is reason to believe that collaborations between agents and developers will not be smooth and painless. For non-agentic tools, prior research has found that developers sometimes struggle with understanding and trusting the AI's output, hindering effective collaboration [7]–[9]. Is this also the case for SWE agents? On one hand, unlike non-agentic tools that generate a single response and thus act as a black box, agents can provide developers with information about their process by showing their work as they go. Further, the capability of agents to make decisions autonomously allows for developers to express higher-level goals rather than low-level instructions. On the other hand, the broader scope of agents' decisions and subsequent actions can exacerbate trust issues and place a higher cognitive load on the developer of understanding both the agent's reasoning trajectory as well as the mutations it performs to the codebase. The answer is yet unclear, due to a lack of research on real-world usage of SWE agents [2].

In this paper, we begin to address this gap by observing 19 open-source software developers use SWE agents to solve real-world open issues of their choosing. Through this study, we aim to address the following research questions:

RQ1: How do software developers collaborate with SWE agents to resolve open issues in active codebases?

RQ2: What barriers do developers face when collaborating with SWE agents to resolve open issues in active codebases?

RQ3: What factors influence participants' success when collaborating with SWE agents to resolve open issues in active codebases?

In answering these questions, this study contributes the first empirical investigation of developer-agent collaboration in resolving real-world issues.

II. Related Work

A. Agentic Tools for Software Development

A single definition of an agentic tool has not yet been clearly established and evolves as new tools are introduced [10]. Yang and colleagues state that a language model 'acts as an agent when it interacts with an environment by iteratively taking actions and receiving feedback' [11], while Jin and colleagues mention that the ability to 'select the optimal solution from multiple homogeneous results' is fundamental to an agentic

Fig. 2: A suggested code edit diff by the agent.

tool [12]. Aside from generating textual responses, agents can perform actions such as searching for files, modifying code, and running terminal commands by calling tools. Further, agents can iterate on their own output, enabling them to complete tasks without the need of human intervention [2].

In software development, there has been a rapid rise in the number of agentic tools in research and in practice [10]. Many of these tools aim to solve end-to-end software engineering tasks, as reflected in the evolution of benchmarks such as SWE-bench [3], a set of real previously-solved GitHub issues, and LiveCodeBench [13], which adds new issues over time. Similarly, SWE-Lancer [4] uses real-world freelance software engineering tasks to evaluate AI tools. While these benchmarks aim to evaluate real-world performance, they all test the *autonomous* performance of agentic tools, rather than their performance with a human-in-the-loop. As Liu and colleagues report, most research-based agentic systems target maximum automation, where human involvement is limited to specification of the problem statement [2].

At the same time, several agentic tools have been integrated into IDEs, such as VSCode Agent Mode [14], Windsurf Cascade [15] and Cursor Agent [16]. Unlike fully autonomous agentic systems, these tools are designed to operate with a human-in-the-loop. Although such tools are rapidly emerging and evolving, at the time of this writing, we know of no empirical research about how developers use such in-IDE agentic tools. This paper aims to fill this gap.

B. Developer Experiences with AI Tools

Prior research has explored the strategies that developers apply when using AI [17], especially those related to prompt engineering [18]–[20]. Prior work has also found that several development tasks are time-consuming or challenging when working with AI developer tools, including communicating the required context to AI tools [8], [17], verifying AI-generated code recommendations [21], and debugging incorrect AI-generated code [9]. Developer studies are mixed about whether AI increases developer efficiency [9] or decreases it overall [22]. Studies have also found that developers often use AI tools as learning aids and information sources [9], [23], and have concerns about the security [9], [22], quality and

reliability [7], [21] of generated code. While these studies have richly explored developer interactions with AI code completion [8], [9], [21], [24] and AI-based chat [25]–[28], there is a dearth of developer experience studies on the use of SWE agents. Yet, SWE agents offer a unique interaction paradigm, differing from non-agentic tools in their capability to perform complex tasks autonomously and take feedback from the environment [2].

Such capabilities are also found in software bots, which can make asynchronous edits to codebases. Prior developer experience research has found that these tools sometimes perform unsolicited actions and produce too much information [29], [30]. Largely pre-dating the current wave of LLM-based AI, such bots typically operate on strict instructions; in contrast, SWE agents can make decisions autonomously, potentially complicating effective collaboration [31].

Prior work has argued for the importance of building design guidelines for agents to facilitate effective interaction mechanisms [2], [32]. Recent work has begun to address this gap for SWE agents. While Epperson and colleagues discussed the challenges developers face when debugging autonomous multi-agent systems [33], Pu and colleagues studied the effect of proactivity in developers' interactions with agentic tools, reporting that contextual awareness and salience of the agent's actions are critical to avoid workflow disruptions [34]. We build on this line of work by studying interactions with SWE agents in real-world tasks.

C. Communication in Software Development Teams

Prior work has found that effective communication of knowledge in software engineering is a complex problem that requires active involvement from all collaborators. Rus and colleagues report that the majority of software engineering knowledge is *tacit*, that is, gained through personal experience, rather than explicitly documented. This can make it difficult for new team members to work effectively, as they lack the knowledge that current team members have [5]. Gonçalves and colleagues correspondingly reported that developers spent the most amount of time in collaborative work if they were a new team member or if they had business/customer knowledge [35]. Prior literature further suggests that a lack in streamlined, synchronous communication can result in mistrust and frustration among team members, particularly in distributed software development teams [36], [37]. It is thus important for both managers [38] as well as junior software developers [39] to be effective communicators and collaborators.

Kuttal and colleagues [40], [41] found through Wizard-of-Oz studies that developer-agent interactions tend to suffer due to a lack of extended discussions and the agent not understanding non-verbal cues. Extending this line of work, we aim to leverage findings on communication barriers in software development teams to investigate such barriers in collaborations between developers and in-IDE SWE Agents.

III. Study Design

To answer our research questions, we aimed to observe developers collaborating with a SWE agent in a highly ecologically valid setting (in 'the wild'). To achieve this, we asked 19 professional developers to use Cursor Agent to resolve issues in a codebase to which they have previously contributed.

A. Tool Selection

In selecting the appropriate SWE agent for our study, we considered the spectrum of human-AI collaboration capabilities, ranging from highly autonomous agents to those that enable flexible interactions with developers. We reviewed several agents, starting with fully autonomous options like AutoCodeRover [42], and moving to agents that allow a moderate level of human-AI collaboration. These moderate collaboration tools often operate outside the IDE and primarily rely on user interaction through prompting, such as Devin [43], Aider [44],OpenHands [45] and GitHub Copilot Workspaces [46]. Finally, we considered agents that operate within an IDE, facilitating high collaboration not only through prompting but also by providing developers access to the full development environment. Examples include Cursor Agent [16], VSCode Agent Mode [14], Windsurf Cascade [15], Cline [47], and Amazon Q Developer [48].

We decided to focus on IDE-based agents as they offer more flexibility for developers to choose whether to rely fully on the agent or closely collaborate on tasks. Moreover, IDE-based AI tools like GitHub Copilot, Cursor, and Windsurf have gained significant popularity in recent years.

To select among these candidate tools, we assessed the types of features and interaction mechanisms present in each of these tools (as of April 30, 2025). These features included those related to flexibility in interactions (modelessness, ability to preview code changes, ability to backtrack within conversations, ability to synchronously edit the same file as the agent), context used by the agent (files opened in IDE, recent user actions) and the interpretability of the agent's outputs (code change explanations, visualization of agent's reasoning). We found that Amazon Q was lacking some of these features, such as providing explanations of its outputs. VSCode Agent Mode was newly released at the time of our study and not yet as mature as other options. Although Cline is a capable SWE agent, the grey literature suggests that it has a 'steeper learning curve, less polished UI than Cursor or Windsurf' [49], making it less appropriate for a study with limited time for participants to learn the tool.

This left Cursor Agent and Windsurf Cascade as our final candidates. Both of these tools contained almost all of the features that we found in our candidates. We conducted pilot testing using both tools with a small convenience sample of developers and found the tools comparable in capabilities and usability. However, Cursor Agent offered one feature that made it particularly suitable for our study: automatic incorporation of users' currently open files and cursor position as context for the agent. Therefore, we selected Cursor Agent as our study tool. Participants used Cursor v0.47 with Claude 3.5 Sonnet.

TABLE I: Participant Demographics

Dimension	Details
Professional Role	Software Engineer I/II: 6; Senior Software Engineer: 7; Principal Software Engineer/Manager: 5; Associate Consultant: 1
Professional Experience	0-2 years: 1; 3-5 years: 4; 6-10 years: 6; 11-15 years: 3; $\geq$16 years: 5
Gender	Men: 14; Women: 5
Age	18-25: 3; 26-35: 10; 36-45: 4; 46-55: 2
Race	White: 6; South Asian: 5; Asian: 2; Black or African American: 2; American Indian or Alaska Native: 1; Hispanic or Latino: 1; Jewish: 1; Multiracial: 1
Region	US: 10; India: 2; Kenya: 2; Israel: 2; Germany: 2; China: 1

B. Tasks

We sought software engineering tasks for this study that:

- reflected developers' typical day-to-day tasks;
- participants were motivated to solve thoroughly;
- could be completed in less than an hour, enabling us to analyze end-to-end agent-participant collaboration; and
- would not use proprietary code, which participants may not be allowed to show us as researchers or to send to a third-party AI tool.

Thus, we asked participants to work on recent open issues (such as bugs or feature requests) in open-source repositories that they had worked on before. We applied the following guidelines to select suitable repositories that were:

- of significant size, that is, with more than 50 source code (i.e., non-documentation and non-configuration) files;
- actively maintained (i.e., commits in the prior month);
- software centric, thus excluding repositories such as a collection of study materials or tutorials [50]; and
- starred over 500 times on GitHub, an indicator of the maturity of the repository [51].

Further, we limited our selection to open-source repositories that were part of two major GitHub organizations that are managed by our company. This enabled us to select participants based on internal-only information about repository contributors, like seniority and geographic location, and to communicate more easily with those contributors.

While most participants selected their first task based on the list of open issues in their project, some (five) participants selected tasks to work on before the start of the session, some of which were not GitHub issues. We allowed them to continue as they had planned. We further asked participants to choose issues that could be solved based on code edits, rather than configuration or documentation fixes, to ensure the agent could provide meaningful assistance.

If participants had time to work on multiple tasks during the session, after they completed their first task, we collaborated with them to choose subsequent issues to ensure diversity in task type (e.g., if the first issue was a bug, we encouraged participants to next select a feature request) and difficulty (e.g., if the first issue was completed easily, we encouraged participants to next select an issue that seemed more challenging).

C. Participants

After shortlisting repositories, we reached out to a total of 43 contributors who were employees of our company and who had contributed at least one commit to the repository in the previous 4 months. We aimed to recruit a diverse sample of participants by reaching out to participants belonging to different regions, genders, levels of seniority and with different levels of activity in the repository. 26 candidates declined or ignored our invitation to participate. We recruited two participants based on recommendations from other invitees who were unavailable. In total, 19 participants accepted our invitation and completed the study. Participants demographics are presented in Table I. 18 of our participants reported having experience with Github Copilot Chat in their regular programming work, while 3 participants reported using Cursor.

D. Protocol

Each study session was a meeting between the participant and the first author, conducted using a video conferencing tool. Sessions were scheduled for 60 minutes, with some variability based on configuration issues and participants' availability beyond the scheduled time. To ensure that such variations would not affect participants' behavior, we informed them that the objective of the study was to observe their interactions with the agent rather than to solve as many issues as possible. Further, we did not analyze time-dependent variables such as action durations and counts, focusing instead on the presence or absence of these actions (as described in Section III-F).

At the start of each session, the study administrator asked the participant to fill out a pre-study questionnaire and consent form. Following this, the study administrator gave a tutorial of the Cursor IDE and its interface by working through a demonstration task to ensure that participants understood the tool, its usage, and its features. Participants were instructed to think aloud and to focus on solving the issue, rather than assessing the AI, to create a more realistic environment. They were also instructed to try to use the AI for all their tasks, and make small manual changes if necessary – this instruction was added after some pilot study participants stopped using the agent altogether after switching to manual editing.

Participants worked on their tasks by remotely controlling the study administrator's screen. We performed all the necessary set up for the participant's repository on the study administrator's system. We chose not to use the participants' systems to eliminate the burden on participants on having to setup the Cursor IDE with their codebase. However, in one case, the participant worked on their own computer because there were configuration issues with the setup on the study administrator's system and because the participant had previously installed and worked on the Cursor IDE.

Once participants completed an issue, the study administrator first asked the participant if they were satisfied enough with the fix to submit it as a pull request. If not, participants were asked to keep working to reach that level of satisfaction (if possible within the scope of the session) before moving on to the next issue. We did not ask participants to create

and submit a pull request since that would require them to enter their GitHub credentials onto the study administrator's system. Nonetheless, so that participants could create a pull request with their credentials, we sent them the diff of their work as a patch after the session.

At the end of each session, participants filled out a post-study questionnaire containing questions about their perceptions of the tool.

E. Data Collection

We collected several sources of data during the study, as described below.

1) Session recordings: To understand participants' usage of the agent in the session, we used video, audio, and screen recordings to qualitatively code both participant actions as well as the details of their interactions with the agent.

a) Participant Action Codebook: We coded participant actions by adapting the CUPS taxonomy, previously used for understanding common programmer actions when using AI code completion [21]. We modified this taxonomy to add states unique to collaborating with SWE agents, such as following the trajectory of the agent's execution and backtracking to a previous point in the conversation (Figure 1a).

b) Chat Trajectory Codebook: We also examined the trajectories of the developer-agent interactions in our study by coding information on the context the participant provides to the agent (e.g., files, code snippets, and external links) as well as the information they gave to and sought from the agent in their prompts. To create our codebook, we performed open coding on two participants' traces, then iterated on this codebook by annotating trajectories for two further participants to create the final version of our codebook.

To validate each of our codebooks, one author coded a study session by applying codes to time segments across the session recording. These codes were then masked, and another author recoded the same session by applying codes to the corresponding time segments. We then computed Cohen's κ, yielding inter-rater reliability scores of 0.92 and 0.89 for the participant action and chat trajectory codebook respectively, indicating near perfect agreement [52].

2) Pre-Study Questionnaire: This questionnaire consisted of optional questions about participant demographics and prior experience with AI tools. We asked questions on prior experience to explore how this may be related to participants' use of the tool [26]. These questions were based on previous studies of developer-AI interaction [21], [26]. We also included a question about participants' perceptions of AI tools for programming in terms of usability [27] by adapting the Technology Acceptance Model questionnaire [53].

3) Post-Study Questionnaire: This questionnaire contained questions about the usability of the tool; the perceived role of the agent in the development process [27]; the importance of the design features of the tool [26], where we drew features from the tool assessment in Section III-A; and an open-ended question for participants to describe any notable observations while using the tool.

F. Analysis Approach

Based on the relevance of actions to the immediate state of the chat with the agent, we interpret the presence and ordering of participant actions at prompt-level granularity (e.g., verifying code changes, stopping the agent) or issue-level granularity (e.g., writing code, running tests) and analyze codes on chat interactions at the prompt-level granularity. Since participants were instructed to think aloud and study sessions varied in duration, we do not analyze the durations of participants' activities, nor do we analyze the absolute counts of these activities to avoid biases toward participants who performed actions repetitively in shorter intervals. We also note specific utterances and scenarios that indicate barriers to effective collaborations, reporting only those that recur across different participants.

While our results are largely qualitative, we include some numerical data throughout to give the reader a fuller sense of the trends that we observed. However, given the exploratory nature of the study and the relatively small sample size, we caution the reader against broad extrapolation or generalization. Congruently, we present findings descriptively, without applying inferential statistics.

Towards helping to ensure the validity of our findings, we sent an early version of our results to all participants as a member check [54]. Our study protocol has undergone privacy assessment and review by our organization. To aid in replication, qualitative codebooks and study materials can be accessed in our Supplementary Material.

G. Limitations

One limitation of our study is that we recruited participants solely from one company. Participants' interactions and perceptions of the tool are likely influenced by the corporate culture and organizational incentives at this company, which is actively promoting AI products and experiences. Likewise, we acknowledge our positionality as researchers in that company, and the inherent unconscious biases and incentives that this environment introduces in our interpretation of the study data.

Another limitation is small sample size (19 participants); we accepted this limitation so that we could perform time-consuming, thorough action and trajectory coding which does not easily scale up to more participants. However, since participants in our study worked on open issues that did not have predefined solutions, we cannot measure or understand the effect of the complexity of these tasks on their performance. Further, external factors such as participant demographics may have affected the results of our study, though we have no evidence for this: upon performing a logistic regression with success as the dependent variable and the predictors being age, gender, professional role, years of experience and number of commits to the repository in the 4 months leading up to the study; none of the predictors were statistically significant.

Most participants in our study had not previously used Cursor or any other SWE agent, and thus our results mostly reflect early usage, rather than regular long-term use. Our study is also subject to observer effects; as participants knew

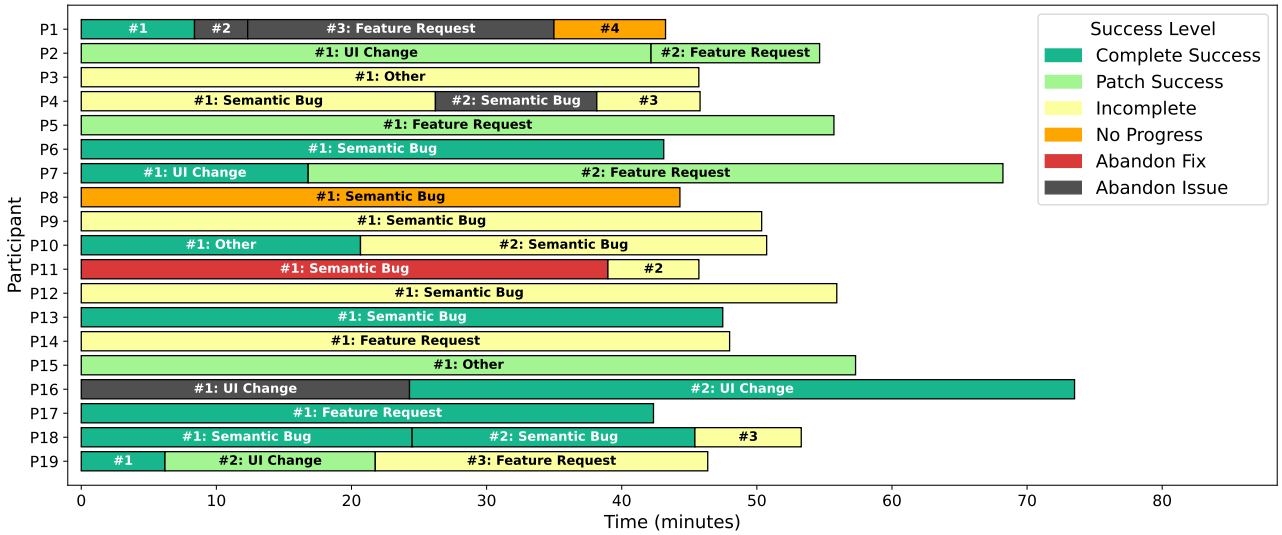

Fig. 3: Issue Timeline by Participant with Success Rates. Success Levels are detailed in Section IV-C.

that we were studying their interactions, they might may have used the tool differently than in a private setting. Likewise, our instruction that participants try to use AI for all their tasks means that, in the wild, developers would likely be more prone to abandonment. Variations in the duration of study sessions might also have affected participants' behavior. Furthermore, while model call latency was the overall dominant latency bottleneck, our decision to have participants use a remote machine induced an additional UI latency which may have hindered their user experience.

Our choice of Cursor in early 2025 also has an effect on generalizability; while we found Cursor to have the most interaction features compared to other in-IDE SWE Agents and we anticipate our findings would apply to such tools, our findings may not generalize to non-IDE based SWE Agents. Further, since AI tools and models are constantly evolving, some findings reflect present-day limitations of agents and their underlying models.

IV. Results

Figure 3 visualizes participants' success and time spent as they submitted 269 prompts to the agent across 33 issues. Participants were satisfied with the final patch for 16 of these tasks, while 4 of these tasks were abandoned for external reasons (such as participants being unconvinced of the issue's validity) for a total success rate of 55% $(16/(33-4))$. In this section, we describe the strategies participants applied, the challenges they faced while collaborating with the agent, and finally how these strategies and other factors were associated with participants' success.

A. RQ1: How developers collaborate with SWE agents

In this subsection, we describe the patterns and strategies we observed in participants' collaborations with the agent.

1) Delegation Strategies: We observed that participants broadly followed one of two approaches to delegate work –

In one approach (the *one-shot strategy*), participants provided the agent with the entire issue description and asked it to solve the issue. Ten out of 17 participants who worked on GitHub issues followed this strategy. This is a high risk, high reward approach:

- If the agent is able to generate a comprehensive fix and verify its changes, the participant can resolve the issue with little additional effort.
- If the agent is unable to generate a comprehensive fix or verify its changes, participants must then manually both understand the generated fix and then iterate on it to get to a valid fix. We observed that this iteration can be time-consuming, as the it can entail not only understanding the code change itself, but also understanding the reasoning behind the change, navigating between code changes, and understanding changes to test files.

In the other approach (the *incremental resolution strategy*), participants manually divide the task into sequential sub-tasks and ask the agent to solve each sub-task sequentially in separate prompts. When the participant is satisfied with the solution to one sub-task, they then ask the agent to work on the next sub-task. Although this approach is safer than the one-shot strategy because mistakes made by the agent can be reviewed and caught earlier, it requires more proactive involvement and interaction from the participant to divide the issue into these sub-tasks. As evidence, participants who used this strategy used an average of 11.0 prompts per issue, compared to 7.0 for the one-shot strategy. Further, these participants were more likely to manually read existing code to enhance their own understanding while working on issues, doing so in 83% (15 out of 18) of the issues they worked on as compared to 60% (9 out of 15) for other participants.

One participant (P7), who had used Cursor before, explained that they preferred different strategies in different scenarios: *"For an issue that is very contained, I trust it to come up*

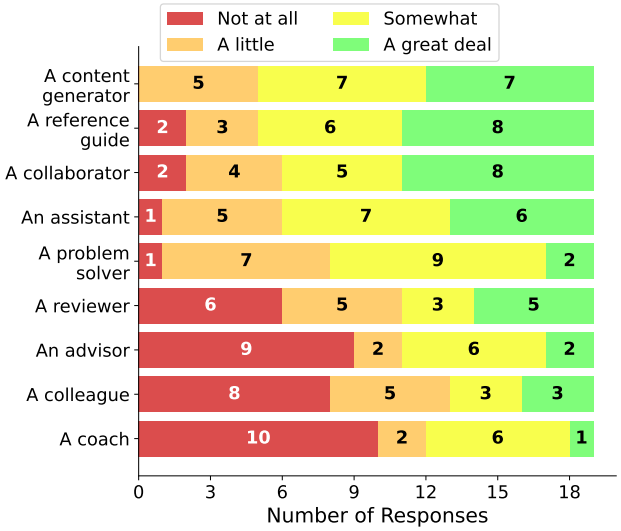

Fig. 4: Post-study questionnaire responses on Agent Perception

with the changes itself, but for issues that are more involved with different parts of the codebase I feel have to point it in the right direction so that anything sensible comes out at the end". However, we also observed that these approaches are not mutually exclusive; when delegating the entire issue to the agent via the one-shot strategy, an incomplete solution may require the participant to prompt the agent with extra steps via incremental resolution. As P3 mentioned, *"The AI agent was able to generate good structure... once more context was needed, it committed errors that need more handholding and additional time to be spent on guiding it"*.

2) Providing Expert Information: We observed that participants provided two different types of information to the agent to help it make the desired code changes:

- *Contextual information* from the issue description and the environmental context (such as test logs). Eg: *"There are build errors in your solution"*
- *Expert information* relating to code implementation or convention that cannot be observed directly from context or the issue description and is based on the developer's prior knowledge of the repository. Eg: *"Usually we don't set an upperbound. I think 3.10 should be fine now"*

Participants' patterns of providing such information to the agent mirror those of software engineering managers. Kalliamvakou and colleagues [38] report that when assigning work, great managers in software engineering leave the implementation details to the engineer, providing actionable feedback when required. Analogously, participants tended to provide less expert information to the agent up-front, and more as feedback; they provided expert information in 49% of prompts that asked the agent to generate a new code change, compared to in 66% of prompts that ask the agent to refine a code change (i.e., iterate upon a previously generated change). This relationship was even stronger for the 12 participants in managerial roles – they provided expert insights in 45% of their prompts seeking new changes versus 75% when seeking to refine changes. As shown in Figure 4, the analogy is reflected in the post-study questionnaire, in which participants report viewing the agent more as an assistant (at a lower level in the organizational hierarchy) than as a colleague (at the same level) or advisor (at a higher level).

While all participants usually tended to provide expert information in feedback prompts rather than up-front, we found that experience factors moderated how often participants provided such information. For example, participants who had more familiarity with the repository tended to provide expert information more often – those with over 10 commits to the repository in the four months leading up to the study provided such insights 70% of time when seeking any code changes as compared to 48% for other participants. Similarly, participants who had previous experience with Cursor also provided more expert insights, doing so in 81% of prompts seeking code changes. Further, participants applying the one-shot strategy provided expert insights less often (in 47% of their prompts seeking code changes) as compared to those applying the incremental resolution strategy (who did so in 64% of their prompts seeking code changes).

3) Requested Tasks: We observed that participants not only requested the SWE agent to make code changes to resolve the task specified in the GitHub issue but also requested assistance in sub-tasks corresponding to other activities of the software development process. While 50% of participants' prompts requested code changes, 27% of their prompts sought explanations of details related to the existing codebase (code comprehension), 16% of their prompts sought to run tests and 11% of their prompts sought explanations related to changes made by the agent (code review). Note that the above categories are not mutually exclusive. This distribution of prompts reflects participants' perceptions of the agent as reported in the post-study questionnaire (Figure 4). Participants viewed the agent the most as a content generator; prompts requesting code generation were most common. Since 'reference guide' was a close second, participants also often used prompts to ask the agent about the codebase. On the other hand, participants did not view the agent as a reviewer, and were thus less likely to ask it to run tests or review its own code.

4) Manual Actions: Participants' perceptions of the agent shaped not only the types of prompts they provided the agent, but also the role the participants themselves took in the collaboration, tending to work on debugging and testing more than on writing code. Since participants reported not thinking of the agent as a code reviewer (Figure 4), they often took the responsibility to debug and test code upon themselves, doing so manually in 21 out of 33 issues by running tests or examining execution logs. In the case of bugs concerning UI changes, it was also easier for the participant to test changes manually by simply observing the state of the UI rather than uploading images to the chat for the agent to interpret.

On the other hand, since participants tended to view the agent more as a content generator, they were more likely to let the agent handle all the code changes to be made, manually writing code in only 14 issues. Further, in 10 of these issues,

participants' code interventions were limited to editing code suggested by the agent rather than adding new functionality.

Some participants expressed desire for AI tools to provide more support in non-code generation actions such as localizing, debugging, and testing, since these aspects of the software development process were often critical to resolving issues. P12 expressed that that the agent was not particularly helpful in localizing their issue, mentioning that *"I think the main part [of solving the issue] was finding where to put the code, so it might have been marginally faster to do it myself"*. Similarly, participants mentioned the challenges they faced when collaborating with the agent for debugging; for instance, P8 responded in the post-study questionnaire that the agent was *"not too great or useful at the debugging and converting-a-repro-into-a-test loop"*, and P11 responded that *"It's puzzling to me why [the agent is] so good at generating code but bad at dealing with basic environmental commands"*. Participants' manual actions thus reflected not only their perceptions of the agent, but also a forced response to the constrained capabilities of the agent as a collaborator.

5) Reviewing Agent Outputs: The agent provided three types of outputs to communicate with participants:

- *Execution*: a trace of the agent's work that appears while the agent is working, including details like files read and commands executed (the bulk of Figure 1).
- *Change Explanation*: at the end of a trace, typically the agent summarized the change it made to the participant's codebase (Figure 1c).
- *Change Diff*: inside the participant's editor, the agent provides a number of inline diffs as proposed changes that the participant can accept or reject (Figure 2).

We observed that participants tended to first track the agent's execution live as-it-happened before reviewing explanations and diffs, with a preference for reading diffs. While Cursor Agent allows for users to synchronously perform manual tasks in parallel to its own execution, participants instead followed the agent's execution as it happened in real time for 84% of their prompts. Usually, this live analysis was sufficient for participants to understand the agent's *process*; after following the agent's execution live, participants only reviewed the process that the agents followed (such as searching the codebase and reading files, in Figure 1b) for 7% of the agent's responses. In contrast, for the majority of cases (specifically, after 75% of the agent's responses), participants reviewed the agent's *outputs* (diffs or explanations) after following its execution in real time.

When reviewing code outputs, participants preferred to verify diffs over change explanations. While participants followed the agent's execution in real time for 95% of the responses that included code changes, these code changes often required further verification – participants reviewed diffs after 67% of such agent responses, while they only reviewed explanations after 31% of these responses. Only for 8% of responses with code changes did participants review code explanations without reviewing code diffs. This preference to review code directly is also reflected in the post-study questionnaire, in

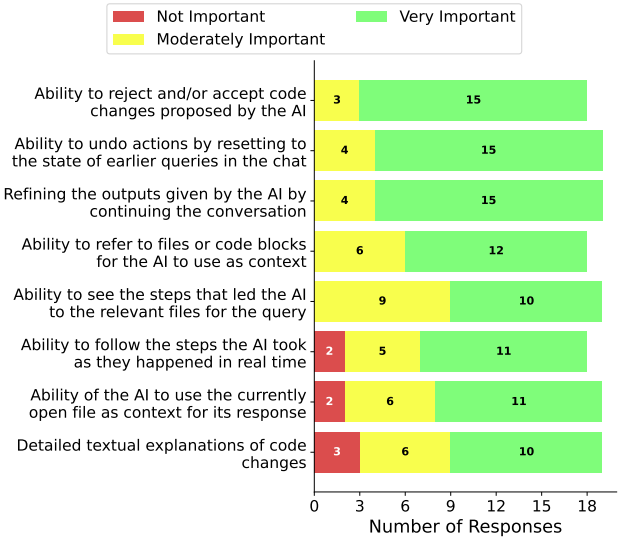

Fig. 5: Post-study questionnaire responses on Agent Features

which participants ranked the ability to see detailed explanations of code changes as the least important feature among the different features of the agent (Figure 5). Manual verification of the agent's outputs was usually sufficient for participants to have clarity on the agent's code changes – they only asked for explanations of generated changes in 16% of their follow-up prompts.

Participants who had previous experience with Cursor tended to focus even less on explanations than their peers, without any apparent impact on their success (Section IV-C2). Like other participants, they followed the agent during 95% of responses containing a code change, and further verified diffs in 66% of these responses. However, they only read explanations after 18% of agent responses with code changes, as compared to 35% for other participants.

6) Iteration Patterns: Participants in our study often iterated on the agent's outputs to get to a satisfactory solution, preferring to refine faulty patches rather than starting over. Iteration was required more often when working on feature requests – participants used an average of 11.2 prompts when working on feature requests as compared to 6.3 when working on bugs. Overall, participants used an average of 8.2 prompts per issue, out of which 52% of prompts that sought code changes sought refinements to previous changes by the agent, indicating that the agent's first outputs were often unsatisfactory and required further modifications. Despite this, participants only rejected code changes 10% of the time after the agent generated code changes. Further, even though the agent allowed for participants to reset the chat to earlier points in the conversation (Figure 1a) – a feature we mentioned in the warm-up task – participants only used this feature in 15% of the issues they worked on. Additionally, participants only terminated the agent's execution before completion (Figure 1d) in 11% of prompts. Thus, participants preferred to iterate on imperfect changes rather than starting afresh by stopping or

resetting the agent.

Despite this proclivity to iterate on imperfect outputs, some participants indicated that they did not think that iterating was always a good strategy. P6 noted that the agent was *"not understanding the implications of previously written bad code, and doubling down by adding more code instead of understanding the underlying issue."* Similarly, P14 mentioned of the agent in their post-study questionnaire that *"if you continue to ask it to fix issues that it hallucinated on you might end up wasting time following in on the wrong path. It would be better to just explain the task in more detail or revert back to a previous breakpoint."*

B. RQ2: Barriers to Developer-Agent Collaborations

1) Lack of tacit knowledge: The agent was sometimes an ineffective collaborator because it lacked *tacit* knowledge, which is gained through personal experience and is undocumented [5]. Unlike with expert insights related to implementation details, it may be difficult for participants to know *when* to discuss tacit knowledge, and even when they do, it may be difficult for them to articulate it in writing. Since the agent cannot understand non-verbal clues, this may hamper effective collaborations [40].

For example, in the case of P7 (who had prior experience with Cursor), the participant was able to interpret a temporal cue but unable to communicate their finding to the agent – when a terminal command running unit tests that was started by the agent seemed to be taking a long time to complete execution, the participant realized based on their experience with the repository that the tests were failing. However, rather than stopping the agent's execution and explaining this in a prompt, they chose to work on the issue manually while the agent came to this realization itself. In other cases, the agent suffered from a lack of social context. For example, the agent once offered an explanation of the codebase that seemed to contradict comments on the issue by a maintainer of the repository, making participant P8 skeptical of the agent's correctness. Prior knowledge of this context might have led the agent to question its assumptions before generating such an explanation.

2) Unsolicited Actions: We observed that the agent was sometimes excessively proactive, making changes beyond the scope of the participant's prompt. We found that the agent made unsolicited code changes after 38% of participants' prompts that did not seek any code changes. In some cases, this was productive. For example, P5 mentioned that the agent *"anticipated what I wanted to do"*. In other cases, this led to miscommunication. For example, when the agent made changes beyond what the prompt specified, P16 was forced to reject the change and write a follow-up prompt to specifically instruct the agent to only make the requested change. P7 mentioned that the agent *"always tries to generate new code, even though the right thing to do here might be [something else]"*.

We further observed that the agent ran terminal commands in 10% of prompts that did not request to run any. As further evidence that these commands were unwanted, participants were nearly three times more likely to stop the agent prematurely when its response included terminal commands (61% vs. 21%). Unlike in code changes, where the agent does not apply proposed changes until explicitly accepted and can undo its changes, terminal executions can permanently change the state of the environment and the codebase. For example, while P12 was reviewing a test log file, the agent ran some new tests on the terminal, even though this was not instructed in the prompt. This led to the test log being deleted, derailing the participant's workflow. Similarly, P11 had to abandon an issue due to the agent corrupting the code environment by running a long series of terminal commands.

3) Sychophancy: Sometimes, the tendency of the agent to agree with whatever the participant mentioned in their latest prompt led to a mistrust of the agent. In particular, after the agent produced a diff and an explanation for that diff, participants were hesitant to trust the agent when in subsequent responses it offered a seemingly contradictory explanation for the same fix. As P15 mentioned, *"If I say that 'your change is wrong', it will revert that change. That makes me lose confidence in it"*. Similarly, P16 expressed in the post-study questionnaire that the agent *"can be confused easily and mislead the developer so [the agent] can't be trusted blindly"*. Bansal and colleagues [32] have also noted that consistency in the agent's outputs in critical to establishing trust in the agent. P18 applied an alternate approach that avoided such issues, explaining that they *"tend to treat [the agent] like a human, where I don't want to give it the answer, I want it to think about it, and come to a conclusion itself. It might have insights I don't know myself"*. They did this by asking the agent suggestive questions (e.g., *are there any negative consequences of...*) rather than instructing the agent. This allowed the agent to retrospect while making refinements rather than blindly changing its assumptions based on the participant's latest instructions. This approach worked well for the participant - they were able to solve the issue as well as find additional related problems with the codebase (unnecessary type checking) that they did not know about previously.

4) Overconfidence: Kuttal and colleagues [40] report that AI agents for programming should be designed to respond with uncertainty when appropriate. Consistent with this finding, we found in our study that participants sometimes mistrusted the agent due to its certainty that the changes it made in its latest response correctly solved the issue. As P9 noted in the post-study questionnaire, *"It never stopped thinking it knew how to solve the problem even though it was wrong"*. Even when the agent generated correct changes and explanations, its confidence in the alignment of these outputs to the issue sometimes led the participant to be hesitant of the agent. P18 noted about a generated fix that *"it works, but it's not really what I want"*. Similarly, upon reviewing a description of codebase files that agent suggested the user to analyze, P10 mentioned, *"These things are true, but not necessarily things I want to modify [to solve the issue]"*. Khati and colleagues [55] similarly report that AI trust metrics for developers include not

only correctness, but also alignment with their intentions.

Participants were particularly hesitant when the agent made many changes. Participants stopped the agent's execution prematurely in 39% of the responses in which the agent performed more than 3 actions (such as terminal commands or code changes), as compared to only 9% when the agent performed 3 or fewer actions. Participants sometimes expressed their concerns over losing control of the agent – while P10 mentioned that *"I don't feel super comfortable when it makes edits and I can't track exactly what and where"*, P14 noted about an issue that they were not able to successfully complete that *"the problem I had with the agent is that it took over, I had to force delete things and it created a mess"*.

C. RQ3: Factors Associated with Success

We considered participants to be successful in solving the issue if they were completely satisfied with the fix so as to submit it as a pull request (*Complete Success*, 10 issues) or if they were satisfied with the generated patch but needed to run some configuration steps or tests outside the scope of the study session before submitting the change as a pull request (*Patch Success*, 6 issues). Participants made progress but were unable to fully complete 10 issues (*Incomplete*), did not make any progress in 2 issues (*No Progress*), and abandoned 1 issue due to an inability to make any progress towards the fix (*Abandon Fix*). Although the total number of issues that participants worked on (N=33) is too low to conclusively reason about participants' success in resolving issues, we observed two main factors associated with success. We summarize the effect of these factors in Table II, and describe them in detail below.

1) Collaboration Strategies: Participants were more likely to be successful in collaborations with the agent when they took a more active role in this collaboration.

For example, participants were more likely to successfully resolve issues when they communicated more with the agent – the average number of prompts used for successful issues was 10.3 as compared to 7.1 for unsuccessful issues. Iterating on the agent's outputs was also a factor in success – participants were only successful in 30% of the issues in which they did not ask for code refinements.

Notably, participants who applied the one-shot strategy (Section IV-A1) were less successful in solving issues – they only succeeded in 38% of the issues for which they provided the agent with the entire issue description. On the other hand, participants who applied the incremental resolution strategy were successful in 83% of the issues they worked on. Similarly, participants who provided the agent with expert insights (Section IV-A2) tended to perform better than those who simply provided the agent with environmental context. Participants succeeded in 14 out of 22 (64%) issues in which they supplied such expert insights to the agent, as compared to only 2 out of the 7 (29%) issues in which they did not provide any expert knowledge.

Beyond providing insights to the agent, we also found that participants were more successful when they manually wrote code rather than just relying on the agent to do so. Participants were successful in 73% of the issues in which they wrote code manually, as compared to in 36% of the issues for which they did not write any code manually. This result did not extend to manual debugging and testing – participants who manually worked on debugging or testing code for an issue were successful in 53% of such issues, as compared to 60% for issues in which they did not perform manual debugging or testing.

TABLE II: Participant Success vs Study Characteristics

Factor	Attribute	Success Rate
Collaboration Strategies		
Delegation Strategy	One-Shot	38%
	Incremental Resolution	**83%**
Manual Actions	**Manual Code Writing/Editing**	**73%**
	No Manual Code Edits	36%
Insights Provided	**Expert Insights**	**64%**
	No Expert Insights	29%
Iteration Pattern	**Requested Code Refinements**	**68%**
	No Code Refinements Requested	30%
Conversation Length	Less than 10 prompts	50%
	10 prompts or more	**64%**
External Factors		
Task Type	Bugs	38%
	Other Issues (Features, UI Fixes, etc.)	**69%**
Experience with Cursor	**Had Prior Experience**	**75%**
	No Prior Experience	52%

2) External Factors: Participants' success was influenced by external factors relating to the tool and the task, such as prior experience and nature of the issue. The 3 participants with previous experience with Cursor had more success (3 out of 4 issues resolved), compared to others' 52% success-rate.

Participants were most successful solving bugs that only concerned user interface changes (5 out of 5 issues resolved), followed by refactoring/variable renaming issues (2 out of 3 issues resolved), then feature requests (4 out of 8 issues resolved), and finally bugs (5 out of 13 issues resolved). These results suggest that participants may have struggled to collaborate with the agent to work on steps specific to bug resolution such as localization and debugging.

While benchmarks suggest that autonomous agents perform differently when working on code in different programming languages [56], our human-in-the-loop study suggests that this was not an important factor in participants' success. Participants had similar success rates – succeeding about half the time – across different languages (Python: 4 of 9 resolved successfully; TypeScript: 5 of 8; C++ 3 of 6; Cmake: 1 of 2; and Java 0 of 1), with the exception of C# (3 of 3).

V. Discussion

A. Collaborating Actively across the Development Process

A recent blog post by Anthropic [57] discusses the prevalence of the 'feedback loop' interaction pattern, in which users collaborate with agents to work on code tasks simply by providing the agent with feedback from the environment. In

our study, we observed a similar interaction pattern, as many of participants' prompts requesting code changes consisted only of environmental context or details of the original issue description. However, our results suggest that such interactions are less effective than those in which developers provide their own insights to the agent (Section IV-C1). Thus, when collaborating with agents, developers should not restrict their role to 'human routers' [58] that simply provide environmental feedback, but rather actively collaborate with them to achieve better results. Participants in our study often took a managerial role in these collaborations, which suggests that agents may be more likely to perform the role of junior engineers in developer-agent software teams. This in turn suggests that the junior developer workforce may be more impacted by the introduction of AI into software development teams. Future work can explore how SWE agents can adopt the communication strategies and behaviors exhibited by high-performing subordinate engineers, and whether adopting these strategies affects developers' perceptions of such agents.

Our results further suggest that developers should actively collaborate with the agent in all stages of the software engineering process, not only in those stages for which the agent is not particularly helpful, such as debugging and testing. Even though participants primarily used the agent to generate code changes, they were more successful when they collaborated with the agent by also manually writing code. On the other hand, even though participants expressed misgivings about the agent's ability to help with debugging (Section IV-A4), manually debugging and testing code did not appear to improve their likelihood of success.

Thus, developer-agent collaborations were most successful when both the developer and the agent actively contributed to issue resolution. In line with prior findings [33], as the agent was not as supportive in debugging and localizing, it was often unable to actively contribute to this part of the issue resolution process, and thus participants struggled to successfully resolve bugs. While software engineering benchmarks have clear, well set up tasks and environments for agents to work on, real development can involve messy environmental setup and debugging steps before the developer is ready to make code changes. However, agents often fail at these tasks, as reflected in their worse performance on more realistic benchmarks [4]. For SWE agents to be useful for developers in their day-to-day tasks, they must empower users across all the steps of the software engineering process, especially the ones that are messy and ambiguous.

B. Calibrating Output Scope

AI agents are designed to solve complex issues autonomously, and have the tools to do so [2]. However, we found that this design can backfire – in our study, participants who applied the one-shot delegation strategy were often unsuccessful (Section IV-C1). When the agent tries and fails to successfully implement complex changes, it is difficult for the developer to debug where the problems in the buggy patch arise from. Further, real-world complex issues often require expert information that only an experienced developer might know. Yet, knowing what information could be relevant may be difficult for developers to foresee, and providing all such necessary information in a single prompt might be tedious for developers, particularly in the case of complex issues. Thus, interactive agents should leverage the presence of the developer as a collaborator by internally calibrating the scope of their outputs when working on an issue. This could enable the developer to provide expert insights when needed, and effectively investigate any problems with the agent's output. Bajpai and colleagues [25] similarly found that in-IDE AI copilots are more effective when debugging if they focus on gathering information and communicating with the user before suggesting a fix. Such a pipeline, as also supported by [59], has the advantages of the incremental resolution strategy without requiring additional effort from the developer to manually analyze the issue or intervene in the fix unless necessary.

C. Collaborating through Discussion, Not Sycophancy

Friction arising from the agent's lack of tacit knowledge is difficult to anticipate for both the agent and the developer (Section IV-B1). This friction may occur regardless of how much context the agent gathers, meaning that agents may always produce incorrect patches in some cases.

In contrast to pair programming, where developers spend considerable time discussing code logic and implementation details [40], [60], the agent rarely engaged in problem-solving discussions. Instead, despite initial confidence in its changes, the agent immediately agreed with whatever the developer said in the latest prompt. While participants using incremental strategies were more successful, the agent contributed little to the solution quality since it simply followed the user's instructions.

However, when participant P18 invited the agent to challenge their opinions, they uncovered novel insights about the existing codebase, a synergy highlighted in recent surveys of LLM-backed SE agents [2]. This suggests that agents designed to challenge rather than merely obey the developer – by engaging in substantive discussion – might not only increase trust and improve solution quality, but further promote developer growth through critical thinking [61].

VI. Conclusion

In this paper, we've seen that developers can collaborate with a SWE agent to solve real-world software engineering tasks, but also that such a collaboration is fraught and that success is not guaranteed. While the capabilities of large language models and of the agents that build upon them expands at a dizzying rate, the capabilities of the humans who work with them are relatively static. The strategies, challenges, and interaction patterns we document here are therefore likely to remain relevant even as SWE agents rapidly evolve. By understanding and addressing these enduring human factors, we hope this work contributes to a future where developer-agent collaboration is both effective and empowering.

REFERENCES

[1] "Stack Overflow Developer Survey 2024," 2024. Accessed: March 11, 2025.

[2] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large Language Model-Based Agents for Software Engineering: A Survey," 2024.

[3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," 2024.

[4] S. Miserendino, M. Wang, T. Patwardhan, and J. Heidecke, "SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering?."

[5] I. Rus and M. Lindvall, "Knowledge management in software engineering," *IEEE Software*, vol. 19, no. 3, pp. 26–38, 2002.

[6] "scikit-learn GitHub Issue 31373." https://github.com/scikit-learn/scikit-learn/issues/31373. Accessed: 2023-12-05.

[7] M. Khemka and B. Houck, "Toward Effective AI Support for Developers: A survey of desires and concerns.," *Commun. ACM*, vol. 67, p. 42–49, Oct. 2024.

[8] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," *Proc. ACM Program. Lang.*, vol. 7, Apr. 2023.

[9] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[10] S. Casper, L. Bailey, R. Hunter, C. Ezell, E. Cabalé, M. Gerovitch, S. Slocum, K. Wei, N. Jurkovic, A. Khan, P. J. K. Christoffersen, A. P. Ozisik, R. Trivedi, D. Hadfield-Menell, and N. Kolt, "The AI Agent Index," 2025.

[11] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," 2024.

[12] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future," 2025.

[13] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code," 2024.

[14] "VSCode Agent Mode." https://code.visualstudio.com/docs/copilot/chat/chat-agent-mode. Accessed: 2025-05-30.

[15] "Windsurf Cascade." https://windsurf.com/cascade. Accessed: 2025-05-30.

[16] "Cursor Agent." https://docs.cursor.com/chat/agent. Accessed: 2025-05-30.

[17] A. Sergeyuk, S. Titov, and M. Izadi, "In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review," in *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments*, IDE '24, (New York, NY, USA), p. 95–100, Association for Computing Machinery, 2024.

[18] V. A. Braberman, F. Bonomo-Braberman, Y. Charalambous, J. G. Colonna, L. C. Cordeiro, and R. de Freitas, "Tasks People Prompt: A Taxonomy of LLM Downstream Tasks in Software Verification and Falsification Approaches," 2024.

[19] Y. Sasaki, H. Washizaki, J. Li, N. Yoshioka, N. Ubayashi, and Y. Fukazawa, "Landscape and Taxonomy of Prompt Engineering Patterns in Software Engineering," *IT Professional*, vol. 27, no. 1, pp. 41–49, 2025.

[20] S. K. K. Santu and D. Feng, "TELeR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks," 2023.

[21] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[22] C. A. Gonçalves and C. T. Gonçalves, "Assessment on the Effectiveness of GitHub Copilot as a Code Assistance Tool: An Empirical Study," in *Progress in Artificial Intelligence* (M. F. Santos, J. Machado, P. Novais, P. Cortez, and P. M. Moreira, eds.), (Cham), pp. 27–38, Springer Nature Switzerland, 2025.

[23] E. A. Haque, C. Brown, T. D. LaToza, and B. Johnson, "Information Seeking Using AI Assistants," 2024.

[24] E. Paradis, K. Grey, Q. Madison, D. Nam, A. Macvean, V. Meimand, N. Zhang, B. Ferrari-Church, and S. Chandra, "How much does AI impact development speed? An enterprise-based randomized controlled trial," 2024.

[25] Y. Bajpai, B. Chopra, P. Biyani, C. Aslan, D. Coleman, S. Gulwani, C. Parnin, A. Radhakrishna, and G. Soares, "Let's Fix this Together: Conversational Debugging with GitHub Copilot," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–12, 2024.

[26] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help With Code Understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[27] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "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*, IUI '23, (New York, NY, USA), p. 491–514, Association for Computing Machinery, 2023.

[28] S. Mohamed, A. Parvin, and E. Parra, "Chatting with AI: Deciphering Developer Conversations with ChatGPT," in *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, (New York, NY, USA), p. 187–191, Association for Computing Machinery, 2024.

[29] L. Erlenhov, F. G. d. O. Neto, and P. Leitner, "An empirical study of bots in software development: characteristics and challenges from a practitioner's perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), p. 445–455, Association for Computing Machinery, 2020.

[30] M. Wessel, I. Wiese, I. Steinmacher, and M. A. Gerosa, "Don't Disturb Me: Challenges of Interacting with Software Bots on Open Source Software Projects," *Proc. ACM Hum.-Comput. Interact.*, vol. 5, Oct. 2021.

[31] B. Chopra, Y. Bajpai, P. Biyani, G. Soares, A. Radhakrishna, C. Parnin, and S. Gulwani, "Exploring Interaction Patterns for Debugging: Enhancing Conversational Capabilities of AI-assistants," 2024.

[32] G. Bansal, J. W. Vaughan, S. Amershi, E. Horvitz, A. Fourney, H. Mozannar, V. Dibia, and D. S. Weld, "Challenges in Human-Agent Communication," 2024.

[33] W. Epperson, G. Bansal, V. C. Dibia, A. Fourney, J. Gerrits, E. E. Zhu, and S. Amershi, "Interactive Debugging and Steering of Multi-Agent AI Systems," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, (New York, NY, USA), Association for Computing Machinery, 2025.

[34] K. Pu, D. Lazaro, I. Arawjo, H. Xia, Z. Xiao, T. Grossman, and Y. Chen, "Assistance or Disruption? Exploring and Evaluating the Design and Trade-offs of Proactive AI Programming Support," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, (New York, NY, USA), Association for Computing Machinery, 2025.

[35] M. Gonçalves, C. Souza, and V. Gonzalez, "Collaboration, Information Seeking and Communication: An Observational Study of Software Developers' Work Practices," *J. UCS*, vol. 17, pp. 1913–1930, 01 2011.

[36] V. Casey, "Developing Trust In Virtual Software Development Teams," *Journal of Theoretical and Applied Electronic Commerce Research, ISSN 0718-1876, Vol. 5, Nº. 2, 2010, pags. 41-58*, vol. 5, 08 2010.

[37] P. Lenberg, R. Feldt, and L. G. Wallgren, "Behavioral software engineering: A definition and systematic literature review," *Journal of Systems and Software*, vol. 107, pp. 15–37, 2015.

[38] E. Kalliamvakou, C. Bird, T. Zimmermann, A. Begel, R. DeLine, and D. M. German, "What Makes a Great Manager of Software Engineers?," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 87–106, 2019.

[39] A. Niva, J. Markkula, and E. Annanperä, "Junior Software Engineers' International Communication and Collaboration Competences," *IEEE Access*, vol. 11, pp. 139039–139068, 2023.

[40] S. K. Kuttal, B. Ong, K. Kwasny, and P. Robe, "Trade-offs for Substituting a Human with an Agent in a Pair Programming Context: The Good, the Bad, and the Ugly," in *Proceedings of the 2021 CHI Conference on*

Human Factors in Computing Systems, CHI '21, (New York, NY, USA), Association for Computing Machinery, 2021.
[41] P. Robe and S. K. Kuttal, "Designing PairBuddy—A Conversational Agent for Pair Programming," *ACM Trans. Comput.-Hum. Interact.*, vol. 29, May 2022.
[42] "AutoCodeRover: Autonomous Program Improvement, author=Yuntong Zhang and Haifeng Ruan and Zhiyu Fan and Abhik Roychoudhury," 2024.
[43] "Devin." https://devin.ai/. Accessed: 2025-05-30.
[44] "Aider." https://aider.chat/. Accessed: 2025-05-30.
[45] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "OpenHands: An Open Platform for AI Software Developers as Generalist Agents," 2024.
[46] "GitHub Copilot Workspace." https://githubnext.com/projects/copilot-workspace. Accessed: 2025-05-30.
[47] "Cline." https://cline.bot/. Accessed: 2025-05-30.
[48] "Amazon Q." https://aws.amazon.com/q/developer/. Accessed: 2025-05-30.
[49] R. Pahwa, "Cline vs. Windsurf vs. PearAI vs. Cursor: 2025's Top AI Coding Assistants Compared." https://medium.com/@pahwar/cline-vs-windsurf-vs-pearai-vs-cursor-2025s-top-ai-coding-assistants-compared-2b04b985df51, 2025. Medium.
[50] J. Zhang, Z. Liu, L. Bao, Z. Xing, X. Hu, and X. Xia, "Inside Bug Report Templates: An Empirical Study on Bug Report Templates in Open-Source Software," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, Internetware '24, (New York, NY, USA), p. 125–134, Association for Computing Machinery, 2024.
[51] Y. Sens, H. Knopp, S. Peldszus, and T. Berger, "A Large-Scale Study of Model Integration in ML-Enabled Software Systems," 2025.
[52] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.
[53] P. Silva, "Davis' technology acceptance model (TAM)(1989)," *Information seeking behavior and technology adoption: Theories and trends*, pp. 205–219, 2015.
[54] S. L. Motulsky, "Is member checking the gold standard of quality in qualitative research?," *Qualitative Psychology*, vol. 8, no. 3, p. 389, 2021.
[55] D. Khati, Y. Liu, D. N. Palacio, Y. Zhang, and D. Poshyvanyk, "Mapping the Trust Terrain: LLMs in Software Engineering – Insights and Perspectives," 2025.
[56] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li, S. Liu, Y. Xiao, L. Chen, Y. Zhang, J. Su, T. Liu, R. Long, K. Shen, and L. Xiang, "Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving," 2025.
[57] Anthropic, "Anthropic Economic Index: AI's Impact on Software Development," 2025. Accessed: 2025-05-19.
[58] K. at Wharton, "Can 'Deep Work' Really Work?," June 2016. Accessed: 2025-05-31.
[59] S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, and E. Horvitz, "Guidelines for Human-AI Interaction," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, (New York, NY, USA), p. 1–13, Association for Computing Machinery, 2019.
[60] L. Prechelt and U. Stärk, "Types of Cooperation Episodes in Side-by-Side Programming," in *Proceedings of the 21st Annual Workshop of the Psychology of Programming Interest Group (PPIG 2009)*, 2009.
[61] A. Sarkar, "AI Should Challenge, Not Obey," *Commun. ACM*, vol. 67, p. 18–21, Sept. 2024.