

# Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025

RUANQIANQIAN (LISA) HUANG\*, UC San Diego, USA

AVERY REYNA\*, Independent, USA

SORIN LERNER, Cornell University, USA

HAIJUN XIA, UC San Diego, USA

BRIAN HEMPEL<sup>†</sup>, UC San Diego, USA

The rise of AI *agents* is transforming how software can be built. The promise of agents is that developers might write code quicker, delegate multiple tasks to different agents, and even write a full piece of software purely out of natural language. In reality, what roles agents play in professional software development remains in question. This paper investigates how *experienced* developers use agents in building software, including their motivations, strategies, task suitability, and sentiments. Through field observations (N=13) and qualitative surveys (N=99), we find that while experienced developers value agents as a productivity boost, they retain their agency in software design and implementation out of insistence on fundamental software quality attributes, employing strategies for controlling agent behavior leveraging their expertise. In addition, experienced developers feel overall positive about incorporating agents into software development given their confidence in complementing the agents' limitations. Our results shed light on the value of software development best practices in effective use of agents, suggest the kinds of tasks for which agents may be suitable, and point towards future opportunities for better agentic interfaces and agentic use guidelines.

## 1 Introduction

*I've been a software developer and data analyst for 20 years and there is no way I'll EVER go back to coding by hand. That ship has sailed and good riddance to it.*

- Developer in Our Survey (S28)

AI is rapidly changing the practice of programming. Already, about half of professional software developers are using AI tools daily [30]. Large language models (LLMs) are particularly good at writing code, and are becoming more skillful every year. Originally, in 2021, LLMs only provided coding assistance as super-charged autocomplete [12]. But more recently, their capabilities have advanced to accessing, modifying, and testing whole codebases in autonomous, step-by-step actions—we are now in the *agentic* coding era. There are many open questions about how capable these agents are and how best to use them. Anecdotally, we sometimes hear from people that they tried it once and it didn't work out so well. But this contrasts with what one reads on social media: some online users claim to use dozens of agents at once to autonomously construct massive software (e.g., [32, 41]), a claim so intriguing but potentially incredulous that it is parodied [16]. *What is really happening?*

Human studies of agentic coding are emerging but still sparse. A notable randomized trial found that experienced open source maintainers were actually slowed down by 19% when allowed to use AI [4], and an agentic system deployed in an issue tracker saw only 8% of its invocations resulting in complete success (a merged pull request) [31]. These results suggest that perhaps agentic AI is not as useful as it might first sound, but still about a quarter of professional developers report that they already use AI agents at least weekly [30]. There have been a few recent investigations [9, 11, 27, 29]

---

\*Denotes equal contribution.

<sup>†</sup>Directing author.

of “vibe coding” [17]. Although the term is sometimes used to mean any coding with AI agents, these papers investigate “vibe coding” as a *particular* form of agent use that aims for an experience of “flow and joy” by trusting the AI instead of carefully reviewing the generated code [27], “where you fully give in to the vibes”, “forget that the code even exists”, and “don’t read the diffs anymore” [17]. There is a tacit acknowledgment by its practitioners that vibing produces lower quality code [9]. As such, *vibing* may not be the most successful approach to agentic coding, and may not be how experienced developers use agents. *How, then, do experienced developers create quality software with AI agents?*

This paper is an attempt to gain insight into the current practice of agentic coding by experts, in order to understand what is and is not working. Compared to prior work, we (a) do not limit the investigation to to vibe coding, and (b) examine *experienced* developers only, in the hopes that they have enough expertise to be insightfully critical of the agentic tools in real-world use. We present a two-part study—13 field observations and a broader survey of 99 experienced developers—hoping to answer four research questions (RQs):

- RQ1 - Motivations.** What do experienced developers care about when incorporating agents into their software development workflow?
- RQ2 - Strategies.** What strategies do experienced developers employ when developing software with agents?
- RQ3 - Suitability.** What are software development agents suitable for, and when do they fail?
- RQ4 - Sentiments.** What sentiments do experienced developers feel when using agentic tools?

Our most salient finding is that, indeed, **professional developers do not vibe code. Instead, they carefully control the agents through planning and supervision.** Specifically, they are looking for a productivity boost while still valuing software quality attributes (RQ1), they plan before implementing and validate all agentic outputs (RQ2), they find agents suitable for well-described, straightforward tasks but not complex tasks (RQ3), and yet they generally enjoy using agents as long as they are in control (RQ4).

The rest of the paper is structured as follows: Sec. 2 describes the methodology of our two-part study; Sec. 3 details the findings, which are summarized in Sec. 4; we discuss the implications of our findings in Sec. 5, relate our study to prior work in Sec. 6, and conclude the paper in Sec. 7.

## 2 Methods

We define *experienced developers* as those with at least three years of professional development experience. We define *agentic tools* or *agents* as AI tools integrated into an IDE or a terminal that can manipulate the code directly (*i.e.*, excluding web-based chat interfaces).

We study the RQs through a two-part study: in depth via field observations (Sec. 2.1), and in breadth via a qualitative survey (Sec. 2.2). Our institutional review board approved both parts. Full replication package of the study materials can be found at [https://osf.io/bxwv2/?view\\_only=25dfabc544fc497dae628d1ea8996896](https://osf.io/bxwv2/?view_only=25dfabc544fc497dae628d1ea8996896).

### 2.1 Part 1: Field Observations

**Participants.** We recruited most participants via social media and personal networks, and one by snowball sampling. We imposed the following screening constraints: (1) three or more years of verifiable professional software engineering experience (full-time, part-time, and internships included); (2) prior experience with agentic AI in creating software; and (3) the ability to demonstrate a realistic task as part of work or personal side projects to be done with agentic AI during the study. Eventually, we recruited 13 participants (1 identifying as female, 12 as male) with years of

professional experience ranging from 3 to 25. [Table 1](#) details their use of agentic tools, model use, and tasks during the study; we detail their tasks later in this subsection.

**Procedure.** Each study session included two parts: a 45-minute observation and a 30-minute semi-structured interview. We conducted the studies over Zoom, recording participants' screens and audio for analysis. The first two authors took turns to run the study. Sessions were run between August 1 and October 3, 2025.

During the observational portion, participants worked on tasks of their choosing using their preferred setup. At the beginning, participants introduced their task, source of code base, agentic AI setup, and task relevance to their day-to-day work. Then, we asked participants to think aloud as they worked. We asked questions about each participant's workflow and task to gain additional insights as needed, following prior practices [\[13\]](#). After about 45 minutes of observation, we asked the participant to start wrapping up their work. We conducted a semi-structured interview afterward. Our questions involved topics relevant to all our research questions and necessary follow-ups on the observation portion. Participants received a \$100 USD gift card after the study.

**Tasks.** Prior to each study, we asked participants to bring their own tasks where they would use their usual agentic workflows; tasks need not be completed during the study and could be something from scratch or ongoing. Each participant worked on tasks that were part of their own ongoing work or side projects during the study. Specifically, five worked on production software as part of their work (P2, P3, P6, P7, P13), three demonstrated exploratory work tasks given the R&D nature of their jobs (P4, P5, P10), and five pursued side projects, including three collaborative (P1, P9, P12) and two personal (P8, P11). Through answers to the question about task relevance to their day-to-day work, five out of 13 participants (P1, P4, P5, P9, P12) reported that their tasks were outside of their professional domains, as indicated in the "Familiar?" column in [Table 1](#).

## 2.2 Part 2: Invited Survey

**Survey Structure.** We designed a 15-minute qualitative survey. After completing the survey, participants could join a drawing to win one of four \$100 USD gift cards. All questions in the survey were required, except for the last question for optional comments.

[Fig. 1](#) shows the precise wording of the non-demographic survey questions. To ground their responses, the survey asked participants to consider the last time they used agentic AI to help develop software. The first part of the survey asked 11 questions regarding that experience: (1) their task; (2) three important things they cared about when developing software in general (to prime them for the next question); (3) what they cared about when using agentic AI; (4) agentic tools they used and (5) why; (6) prompting strategies; (7) whether they used multiple agents (and if so, how); (8) how often they had to modify agent-generated code; (9) parts of the task on which agents did well/poorly; (10) suitability of agents for the considered task; and (11) enjoyment after developing software with agents. [Fig. 3](#) shows the use of agent tools, and [Fig. 4](#) reports the data from three rating questions on code modification frequency, task suitability of agents, and enjoyment after working with agents (more in [Sec. 3](#)).

The second part of the survey asked participants two open-ended questions considering their overall experiences in software development in the past year: tasks they prefer to perform (12) without and (13) with assistance from agents.

The final part of the survey collected information on demographics and software development experience. The full survey is included in the replication package linked above.

We conducted three pilots of the survey within our institution with a total of 14 participants. These pilots helped improve the survey content, ensure data quality, and measure the survey length.

**To guide you in answering the following questions, think about the last time you used AI agents to help develop software. Then, answer the following questions based on that experience.**

1. What was your task?
2. What are three important things you care about when developing software?
3. Which of these did you care about **most** when using AI agents to complete this task? If none of the above, what did you care about?
4. What tool(s) did you use to work with the agents? Select all that apply. [7 options + “Other” free response, see Fig. 3]
5. Why did you choose the above tool(s) for this task?
6. How did you prompt—was there special information you included or any prompting tricks or prompt engineering tactics you used?
- 7a. For this task, did you use one agent at a time, or did you run multiple in parallel? [2 options]
- 7b. If you used parallel agents, how did you set that up?
8. For this task, how frequently did you feel like you had to modify or change the code output generated by the agent(s)? [5 options]
9. What part(s) of the task did the agent(s) perform well on? Poorly on?
10. After finishing (or abandoning!) the task, how suitable do you feel agent(s) are for the task? [6 options]
- 11a. For this task, how much did you enjoy developing software with agents compared to without? [6 options]
- 11b. Can you explain your rating for the previous question?

**Answer the following questions thinking about your *overall experiences this year* in software development.**

12. Thinking of all the responsibilities you have as a software developer, which kinds of tasks do you prefer to perform **without assistance** from agents?
  13. Thinking of all the responsibilities you have as a software developer, which kinds of tasks do you prefer to perform **with** agents?
- Optional: Do you have any other comments or anything else you would like us to know?

Fig. 1. Non-demographic questions from survey. Unless the number of response options are noted, questions are free response.

The survey was updated between each round of feedback. The pilot results were not included in the data analyzed for this paper.

**Survey Participants.** We invited responses from GitHub users, following a recruitment process similar to prior work [20]. To find developers with experience with agentic tools, we focused on the following repositories: (1) Five agentic tools that were ranked among the top 20 apps<sup>1</sup> and open-sourced on GitHub: Kilo Code, Cline, RooCode, Cursor, and Claude Code; (2) Repositories that fell under the topics of AI/ML frameworks (e.g., AutoGPT, DeepCode), open source language models (e.g., BERT, CLIP), development frameworks (e.g., Continue, Vercel AI SDK), and other various types of development tools (e.g., Gemini CLI, PyLance); (3) Repositories tagged with agentic or agentic-coding on GitHub with at least 500 stars. The Appendix shows a complete list of these repositories. We used GitHub’s GraphQL API to extract user profiles from these repositories by querying recent commits, pull requests, issues, stargazers, and forks within the past year.

We then took the set union of all participants retrieved with public emails, removing all duplicates. This resulted in 4,141 unique GitHub users who we invited to take the survey. For response quality control, we distributed the survey via Qualtrics directly through emails, each invitee with a unique survey link (as opposed to public URLs).

249 out of 4,141 users responded to the survey, resulting in a response rate of around 6%, comparable to other research surveys in software engineering [19, 20]. Among the 249 responses, 104 responses were valid—respondents were at least 18 years of age, had at least 3 years of professional experience in software development, and used at least one agentic tool per our definition at the beginning of this section. After the analysis (described below), we belatedly discarded five responses

<sup>1</sup><https://openrouter.ai/rankings>, rankings now may differ from our references at the time of the recruitment

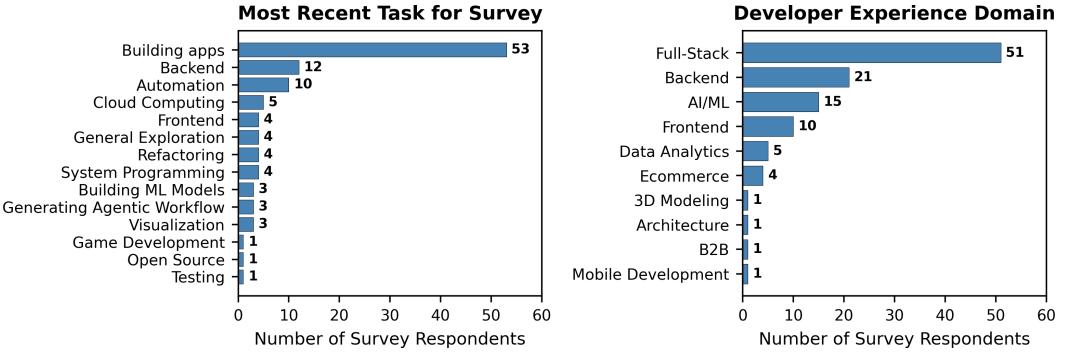


Fig. 2. Distribution of tasks and development experience. The sums exceed 99 (total # of analyzed responses) as task and experience domains may fall under multiple categories.

with highly similar answers that demonstrate, suspiciously, full use of GPT-4.1 to answer our survey questions without personalized steering (confirmed via our experiments with the same model). Our results reflect the remaining 99 survey responses, removing the five above. The 99 valid response were submitted between August 18 and September 23, 2025.

The 99 respondents self-reported years of professional development experience between 3 and 41 years (*avg.* = 12.8, *median* = 10, *sd* = 9.7). Among them, 1 self-identified as female, 1 did not disclose their gender, and the remaining 97 were male. The geographical distribution of the 99 responses is the following: North America (27), Europe (24), Asia (29), South America (13), Africa (5), Oceania (1). Since our survey reached developers worldwide, though it is in English, we received five responses in a language that is not English, which we translated to English equivalents using Gemini 2.5 Pro and cross-validated with Claude Sonnet 4.

Independent from the main data analysis below, we open-coded each task and software development experience reported in the survey. 96 out of the 99 respondents reported tasks where they used agentic tools, as shown in Fig. 2, along with the respondents' self-reported domain of development experience. The majority of survey respondents were full-stack developers, and their tasks primarily involved app building (including web apps).

### 2.3 Data Analysis

We performed two stages of data analysis to answer the four research questions. First, we followed standard thematic analysis process [5] to analyze the qualitative data and develop initial answers; because of the two-part nature of our study, we conducted open coding separately on each part, and we developed themes upon all codes from both parts. Then, we analyzed specific parts of the data in depth, as some of the emerged themes warranted further investigations.

**Observations.** All recordings were auto-transcribed by the web conferencing software. The first two authors conducted open coding of the first two sessions, using field notes as references, to develop the initial code book. Then, they each coded one half of the remaining sessions, discussing new codes until achieving agreements every two sessions.

**Survey.** In parallel to analyzing Observations, the same two authors conducted open coding of the open-ended survey responses. First, they each analyzed half of the first 30 responses to develop on a code book independent from the Observations code book. They then each coded half of the remaining responses and met afterwards to discuss notable findings and share sample data of new codes to establish agreements.

**Joint Thematic Analysis.** Given the large quantity of our qualitative data, rather than computing inter-rater reliability, both coders went through rounds of coding to reach agreement upon each code creation and coding instance, following best practices in qualitative analysis [23]. After coding concluded, the two coder-authors and another author used affinity mapping of codes from both study parts to develop themes. Specifically, we created clusters of the codes (regardless of their source) that share semantic similarities, merged clusters as needed, and developed a description of each cluster that became a *theme*. We report themes as paragraph headings throughout Sec. 3.

Two themes emerged from the first analysis stage, suggesting additional analyses to fully answer our research questions: (for RQ2) developers *control* rather than *vibe* when using coding agents—but how did they control? (for RQ3) developers mentioned using agents for several tasks—but which specific tasks are agents suitable for? To answer these questions, we performed two additional analyses on the prompts from Observations and the tasks mentioned in Survey, respectively.

**Prompt Analysis.** We transcribed the prompts (*i.e.*, user query to the agent) and plans (*e.g.*, an implementation plan in a Markdown file) from the Observations video recordings and performed open coding to identify the varieties of context referenced therein. A first and the last author coded the first session together to develop an initial code book. The two authors then each coded half of each of the remaining sessions, meeting once partway through and once at the end to add, merge, and rename codes. Due to the high volume of the context types (43), we decided to report only the top ten context types (Table 3). Separately, the last author recorded the number of words and estimated the number of steps in each prompt/plan by manually counting the number of separate things the user asked the agent to do, *e.g.*, the number of todo list items (in the case of a multi-line plan), imperative verbs (in the case of a long single-line prompt), or questions asked (in the case of a query for information). The last author also recorded, for each submitted prompt, how many steps were actually executed by the agent—this could be fewer than the number of steps requested in the prompt (*e.g.*, if the user interrupted the agent during the implementation) or more than the number of requested steps (*e.g.*, if the user instructed the agent to execute a plan written previously). Table 2 reports the results of the overall prompt analysis.

**Task Suitability Analysis.** The last author revisited the raw survey responses and made a fine-grained list of tasks mentioned by respondents throughout the questions in all of the surveys, resulting in 189 tasks. In consultation with a first author, they merged these into an initial code book of 116 task codes (manually refined from clusters suggested by OpenAI GPT-5). Then, for each task code, the last author coded the AI agent’s suitability indicated in each survey.<sup>2</sup> Finally, the last author met with the other first author to merge (and occasionally split) task codes to refine and simplify the code book, referencing code-relevant quotes pulled from the surveys by OpenAI GPT-5-mini. The final refined code book consisted of 89 task codes, of which 59 appeared in at least 5 surveys and are reported in Table 4. Each survey mentioned on average 8.5 task codes.

## 2.4 Limitations

We acknowledge several threats to the validity of our findings.

**Demographic and Sentiment Bias.** 12/13 (92%) observation participants and 97/99 (98%) survey respondents self-identified as male, both proportions slightly higher than the global stats as of 2024 (91% male)<sup>3</sup>, limiting gender diversity in our sample and potentially the generalizability of our findings. In general, there is likely a selection bias towards people positive towards AI for both

<sup>2</sup>Initially this coding was attempted with OpenAI GPT-5-nano, but it became clear that accurate coding would require prompt engineering *per task code*. Instead, coding was manual and AI was used only as a double-check to highlight codings potentially missed.

<sup>3</sup><https://www.jetbrains.com/lp/devcosystem-2024/>

the observations and the survey. The bias was deliberate because we wanted answers for RQ2 (Strategies) and RQ3 (Suitable Tasks) from those who might be having some success with AI, but a positive selection bias for AI is perhaps less appropriate for RQ4 (Sentiments).

**Sample Size and Variety.** Our 13 field observations represent a relatively small sample, primarily due to the difficulty in recruiting experienced developers who could share work-related tasks (due to confidentiality concerns) when there were no personal projects. While our goal is to surface usage patterns of agentic tools in software development, rather than full theory development, the observation findings may only achieve local saturation given the limited variety of tasks and agentic tools observed. We addressed this threat by augmenting the observations with the large-scale survey of 99 developers. Our analysis for RQ1, RQ2, and RQ4 is based on holistic thematic synthesis of the survey and observations together (RQ3 primarily relies on the variety from the survey alone).

**Survey Sampling.** Our survey recruitment, which involved scraping public emails from GitHub, is not encouraged by the platform (as reported in a prior similar survey [20]) and may introduce a self-selection bias toward developers active on public repositories; we articulated the reason for the survey invitation in every outreach email. The survey also gathered emails from a disproportionate number of AI/ML repositories, potentially biasing the tasks and respondents' usage of AI; as mentioned above, this latter bias was desirable for RQ2 and RQ3. Despite some sampling bias, respondents reported a wide variety of tasks (Fig. 2).

**Misrepresented Expertise.** There is a possibility that participants may not have been as expert as they represented. To mitigate, for the observation participants, we searched the web with their name to check that their web presence indicated 3+ years of experience (12x by LinkedIn work history, 1x by company website). Moreover, none of their behaviors during the study cast doubt on their reported expertise. For the survey, we did not perform manual verification, but (1) we did not recruit by public online posting or social media, only by direct email, and (2) respondents were not told about any precise experience cutoff as we collected responses from all years of experience.

**Short Observation Time.** Within our study, observations were limited to a single 45-minute session per participant, and thus we were unable to examine the longitudinal use of agentic tools. In addition, software development is a broad spectrum ranging from production deployment to exploratory prototyping, each category potentially involving multiple development cycles; because we prioritized the observation of realistic tasks within limited durations, we were unable to observe any particular kind of software development implemented in its full cycle. As a mitigation, we asked participants in both the observations and the survey about their overall experience with agents outside of the observed/surveyed tasks.

### 3 Results

Below we report results from our two-part study (Observations and Survey) according to our four research questions in terms of behavioral data, quotes, and survey ratings. When quoting participants, we use prefixes of P for Observations participants, and S for Survey participants.

#### 3.1 RQ1: Motivations for Using Agents in Software Development

*I'm on disability, but agents let me code again and be more productive than ever (in a 25+ year career).*

- S22

We asked both Observations and Survey participants about factors that affect their decision to incorporate agents in software development. In Observations, we asked: "When determining how to integrate agents into your workflow, what are your goals and priorities?" In Survey, we asked two consecutive questions at the beginning of the survey: we asked "What are three important

Table 1. Observational study participants, agentic tools & models used during the study, and tasks. “YoE” = years of professional software development experience; “Familiar?” = whether a task is within one’s expertise.

ID	YoE	Agentic Tool(s)	Model(s)	Familiar?	Task
P1	5	Claude Code, Windsurf	Sonnet 4, OpenAI o3, Gemini 2.5 Pro		Building a login system for data labeling platform
P2	9	Windsurf	Sonnet 3.7, GPT-5	✓	Creating a plan for a React app implementation
P3	10	Cursor	Cursor’s auto mode	✓	Improving a prototype for AI safety
P4	6	Cursor	Sonnet 3.5, GPT-5		Improving a user-facing tutorial for a dashboard
P5	9	Cursor	GPT-5, Sonnet 4		Debugging UI for analyzing object detection algorithms
P6	11	Cursor	Sonnet 4, GPT-5	✓	Generating an API and relevant tests
P7	3	GitHub Copilot	Sonnet 4	✓	Building an ML detection pipeline
P8	6	GitHub Copilot	Sonnet 3.5, GPT-4o	✓	Building app that visualizes file histories
P9	3	Kilo Code, Terragon	Sonnet 4	✓	Transferring Radix UI design assets to Base UI
P10	15	Claude Code, Cursor	Sonnet 4, GPT-5	✓	Debugging data pipeline & adding UI features
P11	25	Claude Code, Codex	Sonnet 4, codex-1	✓	Building a Ruby-based card game
P12	9	Cursor	Sonnet 4		Planning to enhance a chatbot & refactoring code
P13	20	Claude Code	Sonnet 4.5	✓	Debugging a production testing suite

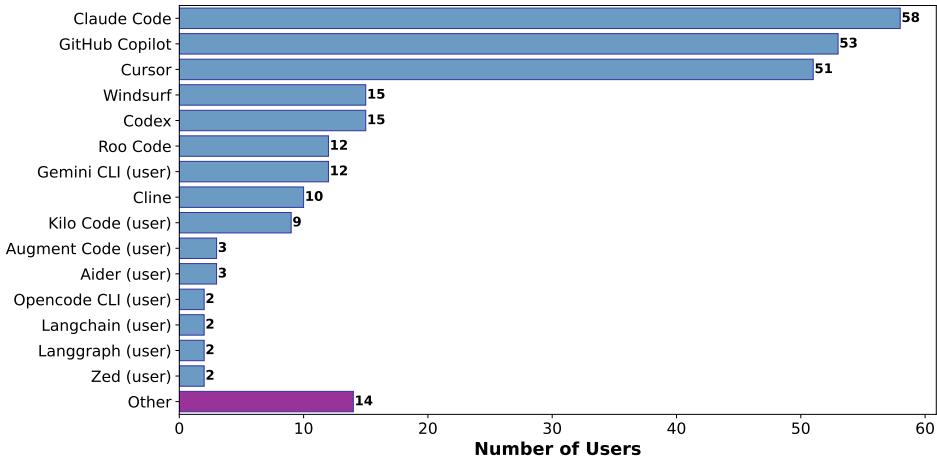


Fig. 3. Use of agentic coding tools in Survey. Numbers sum up to more than 99 as one user may report more than one tool use. Tools suffixed with “(user)” are not within the provided options and reported via a text box. “Other” represent self-reported tools with counts of 1.

things you care about when developing software?” to situate for the next question, “Which of these did you care about most when using AI agents to complete this task? If none of the above, what did you care about?”. Although potentially leading, the goal of the first question was to encourage participants to think about their development goals more broadly rather than focusing on the immediate moment when they choose to type an AI prompt, to avoid answers to the second question that all said “producing the right code” without explaining what, more specifically, was meant to be “right”. Below we report answers to the second question only. We also asked why they chose their particular agentic tools (reported in Fig. 3). We found the following common themes.

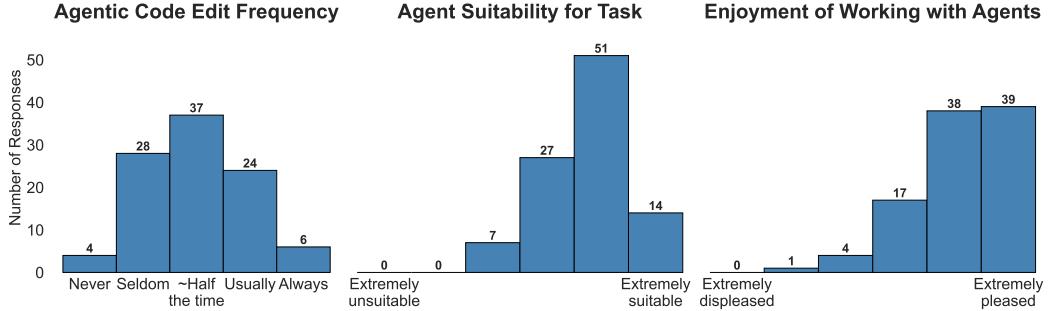


Fig. 4. Survey ratings of modification frequency of agentic code, agent suitability for the task in reflection, and enjoyment of working with agents compared to without.

**Personal Productivity.** Experienced developers valued agentic tools for improving the speed of software development (9x Observations, 35x Survey). Some participants directly comment on agents' productivity boost to their general workflows (4x Observations, 9x Survey) and to the specific tasks they worked on (12x Survey). S59 opined that "*I think my productivity has increased ten-fold (seems exaggerated but it feels like that)*". Many attributes of agentic tools affect their effectiveness in improving productivity, including their ease of discovery and low barrier to entry (3x Observations, 8x Survey), being integrated into existing tools (3x Observations, 11x Survey), and executing user requests at high quality (6x Observations, 36x Survey). Having access to customization (e.g., configuring user rules) (4x Survey) and choice of state-of-the-art models (3x Survey) was another decision factor on using agentic tools. Some participants especially appreciated getting a productivity boost from agentic tools at low to no cost (2x Observations, 15x Survey). Finally, some participants started using agentic tools purely out of curiosity after hearing about these tools from colleagues and influencers (5x Observations, 36x Survey).

**Maintaining Software Quality Attributes.** In addition to personal productivity, experienced developers also valued many software quality attributes when working with agents (6x Observations, 67x Survey). We did not specifically ask about software quality attributes during Observations, nevertheless four participants (P2, P4, P8, P12) mentioned things about the software that they valued, including modularity (P8, P12), maintainability (P2, P8), compatibility (P2, P4), and correctness (P2); two participants (P6, P13) explicitly translated their preferences for code readability and test-driven development to user rules (more in Sec. 3.2).

In Survey, when asked what they cared about when developing with agents, 67 respondents mentioned software quality attributes, in contrast to 37 respondents mentioning non-quality attributes (e.g., productivity enhancement), and 9 valued unspecified quality in software (e.g., "best practices", S56). Fig. 5 shows the quality attributes mentioned by survey respondents. The most mentioned qualities are correctness (e.g.,

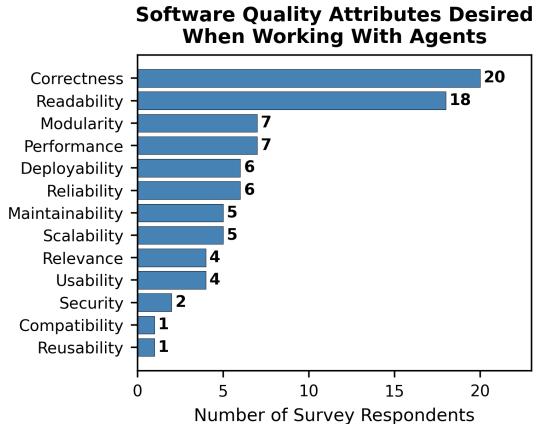


Fig. 5. Distribution of software quality attributes reported by survey respondents.

*“that it wouldn’t [screw] up my code” - S71) and readability (e.g., “most AI agents produce messy and unnecessarily long code when not given enough instructions” - S53).*

The Observations and Survey together suggest that maintaining software quality is a concern of experienced developers when using agents.

**Takeaway 1:** Experienced developers appreciate agentic tools for providing boost in productivity, while simultaneously valuing existing software quality attributes when working with agents.

### 3.2 RQ2: Strategies for Using Agents in Software Development

*I am a software engineer, I prompted by applying the lessons of software engineering to narrative. I described what good would look like, I described concrete used experiences that this should power, I explained the economics behind costs, and I gave a spec of what I needed implemented. This uses most prompt engineering tricks. There is templates and examples and semantic guessing and every kind of thing you can imagine. But to a degree, it’s just good communication. I also always told it to chill out and stop claiming victory so soon. It’s embarrassing.*

- S88

Table 1 shows the agentic tool use across Observations participants, with Cursor (6/13) and Claude Code (4/13) used the most. Fig. 3 shows that in Survey, the top agentic tools used were Claude Code (58/99), GitHub Copilot (53/99), and Cursor (51/99). Participants demonstrated or described strategies they employed when using agentic tools in software development, implying a desire for controlling agent behavior, which we detail in subsequent paragraphs.

**Controlling Software Design and Implementation.** In Observations, we saw that *all* participants, when assisted by agents, adopted strategies to oversee the software design, implementation, or both (if occurred in the same task) regardless of their familiarity with the task domains.

11/13 Observations participants created new software features (except P5 and P13, whose tasks were entirely debugging). We found that regardless of task familiarity, all 11 participants controlled the design of new features to be implemented, asking the agent to develop a draft plan before human revisions (P1, P12) or creating the design plan completely by themselves (P2, P3, P4, P6, P7, P8, P9, P10, P11). Having full control over the design planning was particularly important when the software requirements came from other stakeholders (P2, P12), and when the design included proprietary information (P2, P8). Even when asking the agent to generate a base design plan for implementing a feature in an unfamiliar task domain, P1 always reviewed the plan based on his general software engineering expertise, ensuring each step of the plan implied incremental changes that he could *“test [...] in an integrated way.”*

All 13 Observations participants controlled the software implementation at some level. Three participants (P1, P4, P5), all of whom working with unfamiliar tasks, specified implementation requirements and let the agents drive the implementation, not necessarily reviewing the agent-generated code but closely monitored the program outputs. Although not reviewing the code directly, they still approached the AI in a skeptical and controlling manner, evidenced by rejecting dependencies the agents would try to install (P1, P4) and responding to AI failure by manually tracing through misbehaving code with a debugger (P5). Nine participants (P2, P3, P6, P7, P8, P9, P10, P11, P13), although letting agents generate significant portions of code through prompts, carefully reviewed every agentic change. Five of them (P3, P6, P7, P11, P13) provided the agent with revision feedback after reviewing to keep agentic context consistent: *“I like to keep talking*

Table 2. Summary of observed prompting strategies and verification tactics. “Plan Files”=writing/saving plans to external files, “Context Files”=using local files to maintain agent context/memory across sessions, “Max Size”=number of words in the largest prompt or plan, “Max Steps”=steps in the largest prompt or plan, “SE/P”=steps executed per prompt. Note the number of steps executed in one prompt can exceed the largest plan size if, e.g., a prompt references a prior plan and also gives extra steps to do.

ID	Plan Files?	Context Files?	Max Size	Max Steps	Max SE/P	Mean SE/P	Verification Strategies
P1	✓	✓	917	70	6	2.2	Checked UI functionality, manual testing, iterating with agent
P2	✓	✓	619	71	5	1.9	Verified plan output line-by-line against PRD requirements
P3			225	5	3	1.5	Tested changes in UI, reported back to chat
P4			84	4	4	1.8	Checked changes in UI/IDE, prompted agent as needed
P5			55	3	3	1.6	Tested agent changes in UI, reported results
P6			125	9	13	5.0	Verified plan line-by-line for correct requirements
P7			91	2	2	1.3	Reviewed code, made manual edits, stopped if incorrect
P8			43	3	2	1.1	Reviewed generated code for structure and syntax
P9			275	2	1	1.0	Reviewed PRs, made edits based on GitHub diff
P10			288	7	7	3.0	Checked progress in UI/IDE, inspected dev tools
P11	✓	✓	189	11	12	3.5	Tested functionality in terminal, iterated with agent
P12			48	3	5	2.3	Rotated between output, changes, and artifact
P13			558	3	3	1.4	Reviewed output via linter feedback and test execution
			Min	43	2	1.0	
			Mean	270.5	14.8	5.1	2.1
			Max	917	71	13	5.0

with the AI just because it keeps everything in the chat context [...] because if I change something in the editor [...] I actually need to tell it that I did change something in the editor while it wasn't looking.” (P6) The remaining four (P2, P8, P9, P10) manually modified the code themselves, as P10 put it: “I think it saves me time in the long run, and it also saves me from re-explaining everything later.” Finally, unlike the above 12 participants, P12 approached the *implementation* without agents (when refactoring an unfamiliar code base), instead using the agent to explain the code base architecture to facilitate his refactoring work—at one point, the agent (which was supposed to focus on drawing architectural diagrams) annoyed him by accidentally changing a file he was actively working on: “well, that's not nice.” Table 2 details sample verification strategies participants used to check for the agent-generated code, such as manual testing and reporting back bugs, reviewing code line-by-line, running test suites through the terminal, and reviewing via linter feedback.

In Survey, although we did not explicitly ask about preference for control in software design and implementation, 50/99 respondents mentioned driving the architectural requirements and design, and reported, on average, modifying agent-generated code about half the time (3.0/5, 1=“Never”, 5=“Always”, see Fig. 4).

**Controlling Agent Behavior through Prompting Strategies.** From both Observations and Survey, we identified varying prompting strategies experienced developers employed for better agentic outputs. Most participants agreed that prompts should include clear context and explicit instructions (12x Observations, 43x Survey), e.g. S59 said, “[My] prompts [for modifying existing code] are always very specific, with as much info as possible for example - filenames, function names, variable names, error messages etc. I focus on ensuring prompt has clear statements on what I the LLM to look at and what I want it to do. Small and specific modifications.” Developers demonstrated/reported specific prompting mechanisms including screenshots (3x Observations), file references (5x Observations), examples (1x Observations, 5x Survey), step-by-step thinking (1x Observations, 3x Survey), and

Table 3. Top 10 types of context included in prompts executed by each participant.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
UI or Design Term	✓	✓	✓	✓	✓			✓	✓				
Technical Term	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Domain Object	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓
Reference to Input File	✓	✓	✓				✓	✓	✓	✓	✓	✓	✓
Specific Library or API	✓	✓		✓		✓	✓	✓	✓		✓	✓	
Interaction	✓	✓	✓	✓	✓			✓	✓	✓			
New Feature or Requirement	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		
Reference to Step in Plan	✓	✓	✓	✓	✓								
Reference to Output File	✓	✓	✓	✓		✓		✓			✓		
Purpose of Feature	✓	✓		✓	✓					✓	✓	✓	

external information via Model Context Protocols (MCPs) (1x Observations, 5x Survey), feeling the necessity to “*treat AI as a smart kid that knows nothing about you and outside world*” (S75). Some even used external agents just to improve the quality of prompts (3x Survey). In addition, several participants applied user rules (3x Observations, 18x Survey) to, e.g., enforce project specifications (2x Observations, 5x Survey), provide language-agnostic guidelines (8x Survey), or correct agent behavior based on prior interactions (2x Observations), “*applying the lessons of software engineering to narrative*” as quoted at the beginning of this subsection (S88).

Opinions on the length of a chat differed. Some kept a chat long to keep all necessary contextual information (2x Observations, 29x Survey) or to intentionally interact with agents iteratively (2x Observations, 8x Survey) “*because you don’t exactly know what it doesn’t know*” (P6). Prompting interactively was sometimes necessary as developers noticed agent misinterpretation of previous requirements and needed to provide additional information (2x Survey). Others keep both prompts

Let's add a new feature that lets a user score New Feature or Requirement annotations Domain Object from 1 to 4.

In the UI, a user can score an annotation by hovering over that annotation Interaction and pressing the key 1, 2, 3, or 4. When this happens, the score should be updated in the frontend dataset. The UI should display annotation scores with a

small indicator in a bubble on the bottom-left of UI and Design Terms each annotation highlight. For now, we will ignore the annotations panel and focus entirely on the main content panels. As always, when the user presses save, the annotations data should be persisted to the backend. This includes the new annotation scores.

Here are the steps to follow to implement this:

1. Update the annotation data type. Add an optional “quality” field to both the TextLabel(Technical Term) and the TextMapping(Technical Term) types.
2. Add new keybindings(Technical Term) to the frontend that listen for the keys 1, 2, 3, and 4. UI and Design Terms Take a look at how keyEventUtils(Technical Term) is set up and properly update the handleApplicationLevelHotkeys(Technical Term) function to support the new listener. Implement the listener callback within App.tsx(Reference to Output File).
3. Update the UI rendering to draw the score indicators, when a score exists, on each annotation highlight in the main content panels.

We will do these steps one at a time. Please do just step 1 now. Reference to Step in Plan

Fig. 6. An example prompt from P3, giving the agent seven types of context (of the ten types reported in Table 3) to implement a new full-stack feature for user score annotations. Colored bubble text indicates context type; the figure only notates select examples of each, with more instances of some present.

and conversational contexts short and clear (5x Observations, 3x Survey) to focus on individual small tasks (2x Observations, 2x Survey) and avoid waiting long (2x Observations, 1x Survey).

Despite the varying (sometimes differing) strategies, prompting agents remained primarily a self-taught process with “*trial and error*” (P12): 13 Survey participants mentioned trying and mixing different strategies as their experience with agents evolved.

To gain a better flavor of what participants meant by prompting with “clear context and explicit instructions”, in addition to reviewing strategies reported by Survey respondents (Fig. 7 in the Appendix), we further analyzed the prompts from Observations. We found that Observations participants incorporated a wide variety of context in their prompts, including the following top 10 types of context (Table 3): UI elements, technical terms, domain-specific objects, reference files, specific libraries, expected UI behavior, new features or requirements, relevant steps in existing plans, files to change, and purpose of request. Fig. 6 shows an example prompt from P3 that employs seven of the top 10 types of context. Here, P3 carefully prompted to (hopefully) avoid repetitively editing the generated code. To reduce ambiguity in the prompt, P3 mentioned data types, domain objects (*i.e.*, application-specific entities), particular interactions with the UI, and the specific output file to be edited. P3 further constrained the agent by requesting, “*Please do just step 1 now*”, to prevent the agent from proceeding with the next steps prematurely.

To understand the overall complexity of prompts, we also measured prompt and plan sizes for all participants. The largest plan or prompt used by each participant is listed in Table 2, in number of words (“Max Size”) and number of steps (“Max Steps”); Sec. 2.3 details our measurement approach. Three participants (P1, P2, P11) developed plans in files with more than 10 steps each. P1 and P2 developed their plans in collaboration with the agent, and the plans were notably large, with more than 70 steps. Despite the large plans, P1 and P2 only trusted the agent to implement smaller chunks of the plan at a time. Specifically, we counted the number of steps executed in each prompt (“SE/P” in Table 2), the maximum number of steps executed in a prompt (“Max SE/P”), and the mean (“Mean SE/P”), finding that P1 and P2 never had the agent work on more than 5 or 6 steps at a time. Overall, participants on average asked the agent to work on only 2.1 steps at a time. So, while some participants were comfortable developing large plans for agents, to maintain control those participants only instructed the agents to work on subsets at a time, after which they would verify before continuing.

**Controlling Agent Behavior Outside of Prompts.** Below we report non-prompting user strategies for controlling agent behavior. Because we did not explicitly ask about these behaviors in Survey, we report the findings largely from Observations while interleaving relevant Survey data. All Observations participants oversaw agent performance in plan and code generation. Some techniques were already used in software development: testing agent implementation by execution or systematic testing (P3, P4, P8, P10, P12, P13), documenting agent progress via version control for easy rollbacks (P4, P10, S73, S85), and code validation by reading (P2, P6, P10). Other techniques were more relevant to proactive human intervention, including monitoring agent thinking process while waiting (P4, P5, P6, P10), manually decomposing complex tasks into concrete implementation steps for the agent (P4, P7, P10), revising code to implement good coding practices (*e.g.*, readability) as a demonstration for the agent (P2, P6), and overriding agent behavior with human expertise (P1, P2). Notably, five Observations participants felt more willing to test their code systematically while using agents (P1, P4, P6, P12, P13). P6 reinforced his preference for test-driven development by having agents generate test cases for every agentic change, citing a higher test coverage than before because “*it's part of the workflow now*.” However, testing agent-generated code has not yet been fully automated, especially for testing tasks like UI interactions (P4, P10), database creation (P8), and remote applications like Google Apps Script (P10). S97 summarizes the importance of

reviewing agent behavior for better control: “*I definitely think [agents] are a productivity booster. But I make sure to read every line of code before accepting.*”

Debugging, as a result of agent-generated incorrect code, remained output driven, though some participants offloaded bug analysis and reasoning to agents (P4, P10). When it comes to debugging at the system level, participant strategies vary: P12 and S15 would not even “*waste [their] time*” (P12) with agents, while P10 still trusted the agents after recognizing their potential to hallucinate.

**Managing Multiple Agents through Traditional Version Control.** Some experienced developers (4x Observations, 31x Survey) used multiple agents to create software, though still configured manually (6x Survey), to parallelize different implementation tasks (3x Observations, 17x Survey) and complement the abilities among agents (1x Observations, 10x Survey). They used traditional version control (e.g., git) to compare and merge work from different agents (2x Observations, 4x Survey).

**Controlling Agent through Software Engineering Expertise.** Participants cited their existing software development expertise as a critical strategy for effectively using agents when creating software (11x Observations, 65x Survey). Working with agents required not only awareness of their capabilities (P1, P5) but also code comprehension and debugging skills (P2, P3, P6, P7, P12, P10) and clarity in translating product specs to code (P1, P2, P4, P8, P10, P13). Survey participants agreed upon the importance of “*first understanding coding and then using AI*” (S2), e.g., “*my observation so far is spending time to understand code already generated helps massively*” (S59). Such domain expertise with code could further help articulate an accurate implementation plan for the agents before any code generation (3x Observations, 13x Survey).

**Takeaway 2:** When working with agents, experienced developers *control* the software design and implementation by prompting and planning with *clear context and explicit instructions*, and letting agents work on only a few tasks at a time. Outside of prompting, experienced developers leverage established software development best practices, such as validation and version control, as well as their engineering expertise to incorporate agentic changes with supervision.

### 3.3 RQ3: Agentic Task Suitability

*There's almost nothing I won't use these for. Even for one line items I feel confident in implementing myself, I like walking through it with the agent. It's a second set of eyes on the work I'm doing.*

- S8

Below, we report results for task suitability mainly from survey responses augmented with observations as needed. On average, Survey respondents rated the agents’ suitability for their tasks as 4.73/6 (1=“Extremely unsuitable”, 6=“Extremely suitable”, see Fig. 4, we used a 6-point scale to force respondents to not be neutral).

Software development is much more than just writing code, and agents may be applicable to related tasks. Our survey design (Fig. 1) did not prescribe what these specific tasks might be, but the respondents mentioned many tasks, about 8.5 per response. Table 4 lists the 59 fine-grained task codes mentioned in at least 5 surveys, grouped by the surveys’ general sentiment about agent suitability for each task: “Maybe suitable”, “Controversial”, and “Maybe unsuitable”. Task codes with at least a 2.5:1 ratio of suitable to unsuitable responses are categorized as “Maybe suitable”, those with less than 1:2.5 as “Maybe unsuitable”, and the remainder as “Controversial”. The 2.5:1 (resp. 1:2.5) threshold is somewhat arbitrary but selected based on the author’s judgment. Below

Table 4. Agentic task suitability and sentiments, based on number of survey responses mentioning a task as suitable or unsuitable for AI agents. The 59 tasks below were mentioned in at least 5 of the 99 surveys. Tasks are grouped into “Maybe suitable”, “Controversial”, and “Maybe unsuitable” based on the ratio of responses (imbalance of at least 2.5:1 to be “suitable” and 1:2.5 to be “unsuitable”, the remaining are “controversial”). Tasks within a category are ordered by total number of surveys. Note that “Refactoring” and “Debugging” are subdivided into simple/general/complex, although complex debugging did not receive five mentions and is not reported below. Bars depict the number of survey responses indicating **suitability (S)** or **unsuitability (U)** for a task, **S:U** are the counts, and the final column gives example quotes from the surveys.

Task	↔ U S →	S:U Evidence Example(s)
<b>Maybe suitable for...</b>		
Accelerating productivity		35:2 “Saves a significant amount of time and improves my efficiency” (S21)
Small/simple/straightforward tasks	1:1	33:1 Prefer for “the grunt work” (S25), “tasks that don’t require...innovations” (S30)
Following well-defined plans		28:2 Prefer for “concrete, well-defined, and implementation-focused” tasks (S73)
Generating new code		27:2 Prefer for “coding” (S41); “quickly write code based on [my] prompts” (S50)
Tedious/repetitive tasks		26:0 Prefer for “repetitive [tasks]” (S65), “[tasks] I can do without even thinking” (S66)
Scaffolding or boilerplate		25:0 Prefer for “writing outline of the code” (S53); agent did well at “boilerplate” (S60)
Writing tests	1:1	19:2 Agent did “well on writing simple tests” (S74)
Refactoring (general) or code improvement	1:1	18:3 “Refactoring” (S18,S61,S68,S76,S85); “restricting code” (S65)
All or most all tasks		20:0 Which...do you prefer to perform with agents? “Everything” (S49)
Writing/updating documentation		20:0 Prefer for “writing documentation” (S64)
Backend development		14:1 “Software Engineering Backend”, agent “Suitable”, user “Moderately pleased” (S98)
Debugging (simple) or simple fixes		12:3 Prefer for “bug fixes which I already ‘know’ what the solution is” (S64)
Refactoring (simple) or modifying code or cleanup		14:0 Prefer for “code changes” (S9) or “long Typescript cleanup sessions” (S59)
Starting or setting up a new project		10:4 Prefer for “green field development” (S47) or “creating initial folder structure” (S61)
Frontend development		10:3 Did well at “generating flutter...code following instructions” (S60)
Prototyping or small projects		12:0 Did well on “first prototypes” (S27) and “simple small projects” (S77)
Explaining/analyzing code/APIs/errors		10:1 “[creating] a code map” (S22); good at “explaining how something works” (S59)
Exploring alternatives or experimenting		7:0 Prefer for “throwaway demos/experiments” (S24), “architecture exploration” (S47)
Helping the developer learn		6:1 “Code explanations are helping me learn new things” (S59)
Looking up knowledge for the developer		6:1 “Helpful for research” (S25); prefer for “learning & knowledge lookup” (S76)
Data processing/analysis		5:2 Did well at “creating helpers to process data” (S19)
Collaboratively talking out problems		6:0 “I really like bouncing ideas off AI to see what things it thinks of that I didn’t.” (S7)
Writing helper functions		6:0 “It is good at generating...utility functions” (S55)
Testing		5:0 “Good at loop iteration on running - fixing tests” (S18)
LLM following a knowledge base		4:1 Did well at “understanding context from extensive knowledge base” (S85)
Using major/common frameworks/libraries		4:1 “Batteries all included frameworks...have a bizarrely powerful synergy with AI” (S88)
<b>Controversial</b>		
High level plans (e.g. project or architecture design)	1:1	13:23 Dislike for “big picture’ design” (S9); S28 uses AI for “bigger design and planning”
Debugging (general)		12:8 Did well on “analy[zing] bugs” (S36); “no LLMs [debugged] well” (S57)
Brainstorming or conceptualization		11:5 “For a new problem, I just engage with some LLM, get ideas” (S92)
UI/HTML/CSS		9:4 “It [saves] time to do UI layout” (S26); dislike for “fine tuning CSS/HTML” (S23)
Understanding project architecture		6:7 “Confusion on environment” (S52); use “to understand...sections in...code base” (S59)
Long context sessions or between-task consistency		4:8 “Memory...doesn’t do well. Context management is not great sometimes.” (S17)
Code review		7:4 Prefer for “PR reviews” (S9); dislike for “reviews and quality assurance” (S31)
Writing feature-level implementation plans		6:3 Prefer for “PRD generation” (S55); “planning...went off the rails frequently” (S39)
Creative vision or creative tasks		4:4 Did well on “content generation” (S41); dislike for “tasks [needing] creativity” (S56)
Understanding task context		3:4 Poor at “reading GitHub issues” (S74); “excels when provided with...context” (S85)
Domains/libraries unfamiliar to the developer		4:2 “For an unknown task...it gives a great headstart to look into solutions.” (S51)
Creating configurations		3:3 Dislike for “Ansible dev” (S29); prefer for “creating configurations” (S34)
Version control		2:4 Dislike for “code setup in git” (S17); “git commands I mostly do myself” (S79)
Handling many files at once		2:3 “It poorly perfom when the codebase is modularized” (S75)
<b>Maybe unsuitable for...</b>		
One-shotting code without modification/verification		5:23 Did “poorly on...clean, production-ready code without my manual refactoring” (S31)
Integrating with existing code; legacy code	1:1	3:17 Did “poorly on adding new features to existing functionality” (S33)
Complex tasks (not necessarily big)		3:16 Dislike for “heavily logical” tasks (S78); or “implementing new complex logic” (S61)
Business logic or tasks requiring domain knowledge		2:15 Dislike “when I start a customer software tailored with business logic” (S45)
Writing performant code		3:9 Poor on “performance optimization for high-load scenarios” (S91)
Replacing human expertise or decision making	1:1	0:12 “I like coding alongside agents. Not vibe coding. But working **with**” (S96)
Handling CI/deployment infrastructure		3:8 Dislike for “deployment” (S97); or for “CI/CD pipeline configuration” (S85)
Handling details		2:7 “Poor performance in handling my data scenario requirements” (S11)
Big tasks (not necessarily complex)		1:7 “I struggled with large-scale code generation and refactoring” (S73)
High-stakes or privacy-sensitive tasks		0:8 Dislike for “high-risk codes” (S53); or for “handling sensitive...codebases” (S58)
Security-critical code		2:5 Dislike for “APIs keys” (S12) or “security-critical code reviews” (S85)
Vague/unclear/open-ended tasks		0:7 Poor with “abstracted prompt” (S30); or “sweeping changes...without a plan” (S59)
Using uncommon or private libraries/languages		1:5 “Some internal platforms I use at work that agents don’t know much about” (S8)
Complex logic		0:6 Poor at “complex multi-step business logic without [my guidance]” (S31)
Choosing technologies/frameworks/libraries		1:4 Dislike for “hosting and technology/framework/library decisions” (S42)
Refactoring (complex) or improving architecture		1:4 Dislike for “core changes to the structure of the project” (S16)
Tasks highly familiar to the developer		1:4 Dislike for “few line changes with deep domain knowledge” (S94)
Database design/management		0:5 Dislike for “database design” (S42), “database management” (S50)
New API versions after LLM’s knowledge cutoff		0:5 “Specific Azure APIs were not used [probably] due to API changes” (S32)

we discuss all the tasks mentioned by at least 10 surveys, grouped by theme, with a few other less-mentioned tasks discussed as thematically appropriate. Task codes are written as **task name (S:U)** where **S** and **U** are the number of survey responses indicating **suitability** and **unsuitability**. The reader may find value by considering their own agentic AI use as they read: tasks where others are reporting success are potential opportunities to try agents in one's own practice.

**Agents Accelerate Straightforward, Repetitive, Scaffolding Tasks.** A large number of survey respondents reported they found agents helpful for **accelerating productivity** (35:2), with many reporting that they preferred using agents on **all or most all tasks** (20:0). For example, for the question asking what tasks they prefer to perform *without* agents, S55 said, “*Hardly anything that I can think of, I see that AI assistants have improved [my] experience everywhere in general*”. Unsurprisingly, respondents mentioned agents were suitable for the expected use case of **generating new code** (27:2). More specifically, respondents found agents suitable for **small/simple/straightforward tasks** (33:1) and **tedious/repetitive tasks** (26:0), that agents “*removed the mundane*” (S52), although, as we mention below, this suitability for simple tasks does not extend to *complex* tasks. Agents are also quite helpful at writing **scaffolding or boilerplate** (25:0): S7 preferred using agents for “*writing boilerplate code that we wish we didn't have to write in the first place*”, and S94 noted “[the agent] can scaffold things well”. Agents can assist with **starting or setting up a new project** (10:4), e.g. S98 reported the “*agent perform[s] well when building anything prototype from zero until become one small apps*”, and S61 preferred using agents when “*creating initial folder structure*”, although S27 noted that from-scratch performance is better when **using major/common frameworks/libraries** (4:1) “*like Tailwind CSS / Radix UI*”. With agents' skills at handling simpler tasks, some respondents noted that agents allowed them to shift their thinking to a higher level: S64 reported, “*Something I love about agentic coding is thinking more about the architecture and feature requirements rather than implementation details. I just review the code*”, S66 said they “*allowed me to focus on solving problems and refining the final output*”, and S89 noted “*they save you time to focus on business [by handling] implementation details*.”

**Suitable for Following Well-Defined Plans.** A key feature of instruction-following coding agents, compared to autocomplete-based AI, is the agents' ability to follow multi-step instructions, as S25 put it: “*A few months ago, you had to break down tasks to be pretty small in order to get quality output. But the new models are surprisingly good at developing even larger tasks.*” Consequently, a large number of respondents reported the agents were successful at **following well-defined plans** (28:2), e.g. “*Once the plan exists, the agent can work really cool and I'm just wondering how he does such great things!*” (S16), although many respondents emphasized the need for clear prompting, e.g. “*If given clear instructions they do my work*” (S44), and S7 noted, “*AI can really go off the rails and you spend more time putting it back than you gained, if you don't stay pretty specific with most things*”. Fig. 7 in the **Appendix** lists quotes from respondents that emphasize the need for clarity in successfully prompting the agents. A few respondents reported agents were good at **following a knowledge base** (4:1), e.g. S85 set up a knowledge/ directory in their project with “*structured documentation*” to guide the agent, and four respondents mentioned using the Context7 MCP Server [33] that fetches the latest library documentation for the agent.

Conversely, “*if you don't lead [or] plan then you will [get] stuck*” (S89): several respondents mentioned the agents performed poorly with **vague/unclear/open-ended tasks** (0:7), and a few mentioned trouble with **long context sessions** or **between-task consistency** (4:8), or the agent **understanding task context** (3:4), e.g. S87 noted the agent “*focused too much on specific details of the code without considering the context, which resulted in many more details being changed than required*”, although these sentiments were not universal, such as S85's success with their knowledge base approach above.

**Suitable for General SWE Tasks: Writing Tests, General Refactoring, Documentation, Simple Debugging.** Software engineering (SWE) is a diverse discipline that involves more than just writing implementation code. As mentioned, one of the goals of this survey was to discover if experienced developers found AI agents suitable for other SWE tasks, such as test writing, refactoring, writing documentation, and debugging. Note we subdivided refactoring and debugging into simple/general/complex tasks codes. Sure enough, a notable number of respondents found agents suitable for [writing tests](#) (19:2), e.g. *“it saves time to have AI sketch tests I can refine”* (S31) and *“they are surprisingly good at writing general tests for already existing code”* (S65). Similarly, simple and general refactoring were also viewed as suitable for agents (14:0 for [refactoring \(simple\)](#) or [modifying code or cleanup](#) and 18:3 for [refactoring \(general\) or code improvement](#)), e.g. *“I often say ‘refactor this function’”* (S68). A notable number of respondents mentioned using agents for [writing/updating documentation](#) (20:0), although the responses did not elaborate on their specific writing process. Respondents also found agents suitable for [debugging \(simple\)](#) or [simple fixes](#) (12:3), e.g. *“it was highly effective at fixing a bug when I could point it to a specific failing test”* (S73), as well as for [explaining/analyzing code/APIs/errors](#) (10:1), e.g. *“agents are good at breaking down confusing stack traces and suggesting possible causes”* (S31).

Nevertheless, beyond simple debugging or general refactoring, opinions were more mixed. [Debugging \(general\)](#) (12:8) was controversial, e.g. S27 preferred agents for *“simple bug fixes”* but dispreferred them for *“bug fixing”*. S57 had trouble debugging, with LLMs *“often causing more problems than they initially found”*, whereas S97 preferred using agents for *“debugging (but they can get stuck in debug loops)”* and S85 noted they use agents for *“bug fixes with clear reproduction steps”*. “Complex” debugging did not meet the 5 response reporting threshold.

And although suitable for [general refactoring](#) (18:3), agents were perceived negatively for [refactoring \(complex\)](#) or [improving architecture](#) (1:4): e.g. S16 dispreferred agents for *“any core changes to the structure of the project”*, and S79 avoided agents for *“complicated, many-step, boundary-crossing changes across codebases that have to be coordinated”*. As task complexity increases, agents may become less suitable.

**Suitable for Backend and Some Frontend Work.** Both backend and frontend developers reported that agents were suitable for their work (14:1 and 10:3 for [backend development](#) and [frontend development](#) respectively). For example, S98 felt *“moderately pleased”* with the agents in backend tasks, while S60 appreciated the agents for *“generating flutter [...] code following instructions.”*

Within frontend work, however, specific mentions of [UI/HTML/CSS](#) (9:4) were positive but not universally so: While some participants found that *“The UI part of my work (SwiftUI) was done well”* (S95), and that  *“[agents] help me save time to do UI layout”* (S26), S23 disliked agents for *“fine tuning CSS/HTML”*, and S24 said *“there were often little things in the UI that were not right that were time consuming to fix”*. Notably, in Observations, P1 shared his failure of using agents to implement the desired visual aesthetics of an animated web banner, struggling with accurately specifying his intent: *“I [tried], I don’t know, 30 times [...] And no matter how I prompted it, [it] just wouldn’t achieve what I was looking for.”* These experiences suggest that agents may be generally helpful with UI construction but not for controlling specific details of the visual design.

**Suitable for Prototyping and Experimenting.** Prototyping was a common theme among responses, with respondents finding agents suitable for [prototyping or small projects](#) (12:0), with a number specifically mentioning that the agents were suitable for [exploring alternatives or experimenting](#) (7:0): S15 wrote, *“building a framework/prototype of the solution (let’s say to around 2k LoC) is straight forward with an LLM and can even be vibe coded efficiently”*, while S27 reported the agent performed well on *“creating first prototypes”*, S24 uses agents for *“throwaway demos/experiments”*. These responses suggest agents may be suitable for prototypes and one-off experiments.

**Controversial for Plan Development.** The most interesting debate in the survey responses is whether or not agents are suitable for creating **high level plans** (e.g. project or architecture design) (13:23). A majority of the respondents that mentioned high level planning were negative about using agents for it, sometimes out of concerns that agents are unsuitable for **business logic or tasks requiring domain knowledge** (2:15) and cannot **replace human expertise or decision making** (0:12), discussed in more detail below. Agents also may not be suitable for **choosing technologies/frameworks/libraries** (1:4), and developers reported mixed experience with the agents' ability to understand project architecture (6:7). For example, S68 will not use agents for “*system design! I never trust LLMs for systematic issues*” and S25 said “*with architecture in general I use AI much less, other than being helpful for research*”. With naive delegation to the agent, “*a lot of architectural decisions that the Agent inherently chose can come to bite back. This boils down to domain expertise and supplying your Agent with strict requirements before building the initial infrastructure*” (S15). To prevent its poor architectural choices from biting back, S59 will tell the agent “*my tech stack (at times I drop npm module names that I want in my tech stack to steer it - prevents rework)*”.

Although no respondent suggested that agents could replace human judgment, some still used agents for **high level plans** (e.g. project or architecture design) (13:23), perhaps in part because developers can get assistance from agents short of full delegation: agents can assist with **brainstorming or conceptualization** (11:5) or **collaboratively talking out problems** (6:0), “*it’s like full time rubber ducking*” (S7) or “*like asking a co-developer for questions and insights—without the fear of being judged.*” (S58). P13 mentioned that he “*explicitly asked [the agent] to challenge [him] and [...] double check what [he was] proposing.*” Similarly, S35 said they prefer *not* to use agents for “*high-level planning, but I use AI even for that. Helps me avoid blind spots. However, it’s a lot of back-and-forth, so I don’t leave agents to work on that without my input.*” S97 will “*talk to Gemini 2.5 Pro for design/architecture, then implement with Claude Code...I’ve found it very effective to do design/architecture first, then ask the LLM to output Markdown source code (.md file)... with details of the design.*” And P12 explained that when he needs to architect systems, agents are a “*lifesaver*” because he can ask them to draw the context of the system and how it fits in with a particular feature he has in mind, allowing him to get a visual representation of what the system looks like, expediting his design process. These experiences suggest that agents can be a collaborator, albeit perhaps not an autonomous driver, in helping developers design the system and architecture.

At a less high level, a handful of respondents used agents for **writing feature-level implementation plans** (6:3). For example, S55 prefers agents for “*PRD generation, breaking down [tasks]*”, and S97 relies heavily on agent generated plans from the architecture level (mentioned above) all the way down to feature implementation, they “*ask Claude Code to read the design...to update CLAUDE.md, and to create a SPECIFICATION.md file with all of the design choices made, and a TASKS.md file containing an implementation plan. Then I move onto implementation and refer to TASKS.md, like ‘Please implement Task 1.1...’, etc*”. Three respondents were more negative about feature-level planning by agents, mentioning that e.g. “*planning...went off the rails frequently*” (S39).

**Unsuitable for Business Logic, Tasks Requiring Domain Knowledge, and Human Decision Making.** A notable number of developers reported that agents are unsuitable for **business logic or tasks requiring domain knowledge** (2:15). For example, S78 avoided agents for “*heavily logical or business rules parts*” and S95 avoided them for “*very specific business logic that requires a lot of context*”. This avoidance is related to the unsuitability of agents for **replacing human expertise or decision making** (0:12): that is, 12 respondents mentioned the need for a human in the loop or the value of human expertise. “*Try to be an expert in area, if you are [a] software arch[itect] you know things that can improve the models results, you are the pilot the model are copilots and you need to share [with] them the expertise that you have*” (S12). S59 emphasized, “*The decision making on*

*implementing a agent generated plan/option/modification lies with me - I take my time thinking about its implication before I give a thumbs up.*" Similarly, S62 noted, "You need to master certain skills and judgment to verify whether the generated code is correct". No respondent indicated that agents are suitable for completely autonomous operation, instead the more commonly expressed idea was that of S83: "I do everything with \*assistance\* but never let the agent be completely autonomous—I am always reading the output and steering".

**Unsuitable for Perfect Code Generation.** Human oversight is still needed because, as many respondents indicated, LLMs are still unsuitable for one-shotting code without modification/verification (5:23) and can have trouble handling details (2:7). That is, many respondents noted that generated code required revision, verification, or clean up. "Almost everything it performs great on. It just does not one shot things" (S49), but "sometimes the agents generate messy code" (S58) and the agent could suffer from "over engineering, not as detailed aware, [it] keep repeating itself" (S94). An oft-mentioned problem was integrating with existing code or handling legacy code (3:17), e.g. S80 found the agent might needlessly create duplicate code, it "often wrote custom code for pre-existing functions in core libraries", and S64 reported the agent "has difficult[y] ensuring features integrate within the entire architecture end to end". In Observations, 8 participants said that agents could sometimes do more than asked, explaining that agents can create unnecessary files, rewrite entire sections of code, or install packages that are not needed for the task at hand. Put bluntly, P7 said, "I do not quite enjoy the fact that it will just like start changing and changing stuff without me actually understanding what's going on". A couple survey respondents mentioned trouble with legacy code, e.g. "if I want to explore or understanding other legacy codebases, my experience is not so good" (S92) and S53 avoids agents when "dealing with legacy or high-risk codes; I'd rather go to StackOverflow for those cases". A few respondents noted agent troubles with using uncommon or private libraries/languages (1:5) or with new API versions after LLM's knowledge cutoff (0:5) (although to mitigate this issue, as mentioned above, at least four respondents use an MCP server so the agents can look up the latest documentation). Although agents cannot one-shot code, a few respondents still used agents for code review (7:4), e.g. S9 prefers to use agents for "tickets & PR reviews" and S91 similarly uses "Claude Code as a pull request reviewer in GitHub for code quality assurance". Three Observations participants also used agents for PR review on GitHub (P5, P6, P9). Nevertheless, 4 survey respondents expressed worry: e.g. S76 avoids agents for "critical code reviews". With the mixed reliability of agentic code generation, S13 took reviewing responsibility upon themselves as a matter of accountability: "Ultimately, I am responsible for all code and deliverables, so I will independently review the content generated by the agent".

**Unsuitable for Complex Tasks.** In contrast to the suitability of agents for small/simple/straight-forward tasks (33:1) and tedious/repetitive tasks (26:0) discussed above, agents become less suitable as task complexity increases. Respondents reported agents were less appropriate for complex tasks (not necessarily big) (3:16), specifically for complex logic (0:6) and, as mentioned previously, for refactoring (complex) or improving architecture (1:4). For example, S4 "never finished the task" because "my use case is complicated and [the agent] performed poorly throughout most of it", S90 reported "if the function is complex (requires several dependencies), agent tends to hallucinate", and S16 found "if the task is complex, like splitting lots of changes in a work directory to a bunch of smaller commits, it can [get] stuck and do bad things". Unsuitability was also reported for big tasks (not necessarily complex) (1:7): e.g. S55 found that "when large changes need to be made, agent often misses the following the guidelines and at times marks incorrect changes". In contrary, P3 from Observations explained how he completed a migration to React in three days with agents, "It would have taken probably two weeks of my own time... And at the end of three days, I don't think we've caught a single bug or...design mistake ever since then". Which kinds of tasks are seemingly complex

but still suitable for agents is an open question—we speculate the React migration may have worked because React is a common library, and the migration is scaffolded from the prior code.

**Misc Unsuitable: Performance-Critical, Deployment, High-Stakes, and Security-Critical Situations.** A handful of other tasks were mentioned as unsuitable for agents. Some respondents reported that agents were bad at writing performant code (3:9), e.g. S43 avoids agents for “high performance structural design”, S91 found the agents were poor at “performance optimization for high-load scenarios”, and S55 noted “they often tend to write code that is less performant”. Respondents also avoided agents for handling CI/deployment infrastructure (3:8), generally preferring to handle the broader infrastructure setup themselves. Agents were also avoided for high-stakes or privacy-sensitive tasks (0:8) or security-critical code (2:5) because their sensitive nature leaves little room for error, e.g. S41 avoids agents for “tasks such as high stakes decision making, sensitive data handling without safeguard etc” and S64 avoids agents when “implementing critical components that need to be performant or secure”.

**Takeaway 3a:** Experienced developers find agents *suitable* for accelerating straightforward, repetitive, and scaffolding tasks if prompted with well-defined plans. Beyond writing new code and prototyping, these suitable tasks include writing tests, documentation, general refactoring and simple debugging.

**Takeaway 3b:** But, as task complexity increases, agent suitability decreases. Experienced developers find agents *unsuitable* for tasks requiring domain knowledge such as business logic, and no respondent said agents could replace human decision making, in part because the generated code is not perfect on the first shot.

**Takeaway 3c:** Experienced developers *disagree* about using agents for software planning and design. Some avoided agents out of concern over the importance of design, while others embraced back-and-forth design with an AI.

### 3.4 RQ4: Sentiments Towards Agents

*It felt like driving a F1 car. While it also felt like getting stuck in traffic jam a lot, I still felt optimistic about it.*

- S24

On average, Survey participants rated their enjoyment of developing software with agents as 5.11/6 (1=“Extremely displeased”, 6=“Extremely pleased”) as compared to without agents (Fig. 4). All Observations participants shared similar positive sentiments. The subsequent paragraphs detail reasons for their overall enjoyment and insights for the future of agentic software development.

**Happiness and Curiosity.** Observational participants explicitly felt positively after working with agents (P3, P7, P8), pointing out that its success in completing a task made them happy with the outcome and the time they saved. P7 explained, “If it’s [...] a simple enough task, [...] it most likely will get correct and then I will have something neat and cool that I can just use directly. And [...] I feel good.” With these positive feelings, some found coding fun again: “I’ve had more fun working in this codebase, [...] especially since I started using Cursor, than I’ve had writing code in a long time.” (P3) Similarly, S8 wrote, “This has made code fun again. I’m producing things that I didn’t have time or energy to do before. It’s like rediscovering computers again for the first time.” Two Observations participants (P10, P11) and two Survey respondents (S30, S59) chose to use agents out of pure curiosity about their coding abilities. P10 and S59 further shared a sense of relief brought by their

agentic coding workflows, as S59 put: “*I am less stressed as compared to my days without an agent. I am mostly thinking how to structure my prompts.*”

**Need for Humans to Stay in the Loop.** While there are positive sentiments towards coding agents, experienced developers prefer to *control* agent behavior in building software (5x Observations). P2 was concerned about whether “*we’re at the point where [...] I can hand off the responsibility of software engineering [...] to an AI [...] as a software engineer.*” Similarly, experienced developers (6x Observations) discussed the necessity of human understanding of a code base is for production, especially when reviewing agent-generated code. This further extends to software maintainability: “[agents] can also be used as a crutch to [...] have people who don’t know what they’re doing generate something that seemingly works, but it is kind of a nightmare to maintain anything.” (P4) Overall, a proper software engineering foundation is critical for controlling the agent (5x Observations and 5x Survey), in terms of not only evaluating the generated code but also correcting the agent when its output misaligns with the developer’s intent. Ultimately, “*you still do have to like think about it yourself [...] and evaluate of what the LLM is telling us is a good approach.*” (P8)

**Trust in Agent’s Capabilities.** Some experienced developers do trust the agent’s output, but only after checking the code in some capacity (5x Observations), and will correct it or even manually code something if there has been a dip in quality (3x Survey). This goes both ways for those building projects within or outside of their domain expertise. For those building software outside of their typical domains of expertise (P1, P4, P5, P12), they acknowledged the agents’ abilities in generating more usable code than themselves, but only leveraging such benefits to create what they want more effectively. Even those with tasks within their expertise echoed this sentiment (P2, P3, P6, P8), hopeful for the AI’s capabilities of allowing them to build software in domains they are not familiar with. As P2 explicitly commented, “*agents are gonna do a faster and better job than I would ever do [...] in terms of [...] creating websites.*”

**Agents for Collaboration.** Some experienced developers view the agent as a means of elucidating thought processes for the software they are building (4x Observations, 12x Survey). P12 mentioned using agents as a “*rubber ducky*”, a common expression in software engineering where the developer explains their thought processes in natural language to try and figure out the next steps for their coding tasks. More broadly, agents are collaborators for brainstorming and more abstract tasks, where users are not only writing code with the agent but also thinking about the big picture (5x Observations, 34x Survey). Notably, experienced developers preferred staying in control even in these use cases, driving the conversation and thought processes themselves and viewing agents more as sources of inspiration (12x Survey). As S83 summarized, “*I do everything with assistance but never let the agent be completely autonomous—I am always reading the output and steering.*”

**Agentic Coding as the Future.** Experienced developers felt that AI-assisted coding workflows are only going to become more ubiquitous in future software development (4x Observations, 13x Survey). Some developers found it difficult to return to the pre-agent era: “*I don’t want to go back to writing code manually. Now it seems like such a waste*” (S35); “*there is no way I’ll EVER go back to coding by hand*” (S28). Some even urged developers to “*get into agentic AI as soon as possible otherwise you will be fall behind soon.*” (P13) On a concluding note, sentiments or trust towards agents aside, experienced developers felt that agents brought them closer to the software they wanted to create (5x Observations), echoing the high enjoyment ratings from the survey (Fig. 4).

**Takeaway 4:** Experienced developers enjoy working *with* agents as source of collaboration, rather than delegating work to the agents completely.

## 4 Results Summary

We began this paper with an excerpt from S28 saying “good riddance” to manual coding. The full quote is itself a nice summary of our results:

*No matter how you slice it, agents are extremely accelerating. I've been a software developer and data analyst for 20 years and there is no way I'll EVER go back to coding by hand. That ship has sailed and good riddance to it. One word of warning to the young ones getting into the business – it's still really important to know what you're doing. The agents are great for a non-technical person to create a demo, but going beyond that into anything close to production-ready work requires a lot of supervision. :)*

- S28

Our study shows that **experienced developers generally enjoy working with agents** in developing software by **controlling agent behavior** through strategic plans and supervision, rather than *vibing* with agents. Specifically, our results answer our research questions (Sec. 1) as follows:

**RQ1 - Motivations.** Experienced developers appreciate the boost in development speed that agents could bring, while still valuing fundamental software quality attributes.

**RQ2 - Strategies.** With these values, experienced developers control both the software design and implementation when using agents, leveraging pre-existing knowledge about software development, prompting strategies, and established techniques for quality assurance (e.g., validation and version control), while letting agents work on only a few tasks at a time.

**RQ3 - Suitability.** Through employing these strategies, experienced developers identify that agents may be best for accelerating straightforward, repetitive, and scaffolding tasks, including e.g. writing tests, documentation, general refactoring and simple debugging. But as task complexity increases, agent suitability decreases. Experienced developers avoid agents for business logic and do not yet find agents suitable for completely autonomous operation. Opinions about using agents for high level planning are mixed.

**RQ4 - Sentiments.** Combining their expertise with software and understanding of agent capabilities, experienced developers generally enjoy working with agents, perceiving agentic tools as valuable source of collaboration in software development—rather than replacement of humans—that requires wise decision making and supervision from the human.

## 5 Discussion

Below we discuss why experienced developers are not vibing (Sec. 5.1), consider possible actions for current software developers (Sec. 5.2), and discuss future work directions (Sec. 5.3).

### 5.1 Why Don't Pros Vibe?

*I like coding alongside agents. Not vibe coding. But working \*\*with\*\*.*

- S96

We found that experienced developers strategically *control* agent behavior in software development, rather than *vibing*. That is, they do not let agents drive their software design or implementation, and do not “fully give in to the vibes, embrace exponentials, and forget that the code even exists” [17].

Why, then, do experienced developers refuse to code with *vibes*? We can think of four possible reasons: (1) experienced developers value software engineering principles, which they have received from education and enforced every day at work, something difficult to automatically implant in

agents even with prompts and user rules (6x Observations); (2) experienced developers often work on production software, rather than “throwaway weekend projects” [17], that leads to impact on real users and/or involves other stakeholders, many of whom decide the software requirements and affect the design (P2, P12); (3) when working with familiar code bases or tech stacks, especially with pre-defined design requirements, there is little room for exploratory coding, and developers often have better context than agents, such that delegating implementation to agents could instead lead to frustrating iterations; (4) for unfamiliar tasks, when agentic solutions go wrong, developers found it frustrating that “*it [could take] a lot of time to resolve them*” (S13). These results imply that expertise—when available—supersedes vibes and drives the development of quality software.

## 5.2 Considering One's Own Agent Use

*I never again want to give two [poops] about the specific best way to quijibo the toaster in dingledangle framework v0.21. The agent reads the latest docs and then is forced to comply.*

- S88

A goal of this work is to paint a better picture of what is and is not working with agentic coding. Some readers may themselves be developers wondering if they are using agents to their full capabilities—indeed, this was one of our own motivations for this work. Although our studies were only designed to gain some insight into current agentic use, rather than developing a complete account of best practices, our findings do suggest a few basic recommendations and potential strategies for improving one's own control of agents.

A first recommendation for agent use is to practice prompting clearly, specifically, and in detail as in e.g. Fig. 6. Prompts can potentially be long and include many steps, but vague prompts will not work. A second recommendation is to not expect to be able to vibe, because current agents cannot *autonomously* manage the development of large software. Human expertise remains essential: apply the lens of software engineering to ensure code quality and project structure. Be sure the project remains under control.

Beyond these high level recommendations, another potential use of this paper may be to compare the tasks in Table 4 with one's own practice, considering which tasks one might try using agents for. While our study population generally agreed that agents were not fit for complex tasks or core business logic, there are a large variety of other tasks for which agents might be useful, including scaffolding components, test writing, refactoring, writing documentation, simple debugging, project setup, experiments, explaining code and errors, knowledge lookup, “rubber duck” conversations, and knowledge lookup. As such, it may be worthwhile to try agents for these tasks. Still, it likely takes practice to learn how to use agents well (*“the more experience I have with it, the more my personal workflow yields better results”* - S28), so in an optimistic view agents may even be fit for the “controversial” tasks in Table 4, like planning, brainstorming, or project understanding—perhaps only part of the surveyed population had developed effective prompting strategies for these tasks, so maybe the limiter was not agent ability but user prompting skill. These scenarios as well might be opportunities for trying agents.

Many of our study participants reported that agents are skilled at following well-defined plans. In Observations we did see some evidence of long plans (P1 and P2 had plans with 70 and 71 steps, respectively; see Table 2), although, despite occasionally long plans, participants ensured that agents implemented plans piece-wise in chunks of manageable size (e.g., “Please do just step 1 now” in Fig. 6). Drafting longer plans, perhaps in a file, is another strategy to try.

Note that, psychologically, trying AI and then failing hurts trust in AI more than success builds trust [36]. In other words, if the readers (or ourselves) try agents for a new task where it does not work right away, it may be advisable to expect that initial discouragement and keep iterating.

### 5.3 Future Work

*This is the future of development, and it's very fun.*

- S25

We see two broad directions for future work: improvements to the interfaces for agentic coding, and further detailed studies of developer use of agents to derive best practices.

**Better Interfaces for Controlling Agents.** One avenue for future work is to improve agentic tools for scenarios where they are not as helpful. Our investigation revealed some areas where agents still struggle, specifically:

- (1) Automated testing, particularly testing in a remote or realistic environment, is difficult.
- (2) UI debugging through prompting is tedious.
- (3) It is difficult to understand why an agent fails or hangs.

Future work can discover or design prompts and interfaces to improve these pain points.

Similarly, because this investigation shows that experienced developers plan in order to better control agent output, there is an opportunity to improve interfaces for the planning step. Careful interface design might guide expert programmers, inexperienced with AI agents, into more successful workflows. A variation of that interface might help scaffold novices' workflows, so they learn to address the same considerations as experts and prompt more effectively.

**Developing Best Practices.** To our knowledge, there is not yet any rigorously developed set of best practices for using coding agents. Although this study produced an initial account of agent use by experienced developers, these usage patterns may not necessarily be optimal. As noted, Becker et al. [4] observed a 19% performance drop when AI use was allowed for coding. Nonetheless, it is an open question whether those developers, or the participants in our study, have discovered how to use agents to their full potential. As a partial mitigation, this was not a first-use study—we aimed to recruit developers who were active users of AI—but we did not specifically screen for productivity with AI.

To rigorously develop best practices for using agents, future work might focus more narrowly on developers who are most successful with AI agents. It might be tricky to find these people because development productivity is notoriously hard to measure. One method might be to ask people about their most productive colleagues, aiming to recruit developers well-esteemed by their peers. Alternatively, one could recruit developers who (1) have developed highly automated setups that (2) have a proven record of shipping working software; the presumption here is that more automation means more productivity, so long as the end result still works. After gathering best practices from these “most productive” developers, a further point of study is to, like Becker et al. [4], validate that these practices *actually* accelerate developers, potentially in a greater variety of settings than Becker et al. [4], which only considered large, open-source projects with experienced maintainers.

Ultimately, further research is needed to confirm the most productive agentic usage patterns and relevant scenarios, and to verify that coding agents are measurably helpful for productivity.

## 6 Related Work

Programmer experience research for the past few years has felt like living in a whirlwind—sometimes it feels like AI is changing the status quo almost daily. Below we venture to describe some of the recent whirlwinds: we discuss AI agents for software development, recount usability studies of generative AI for coding, and compare recent empirical investigations of vibe coding in particular.

## 6.1 Agents for Coding

With the rapid improvement in transformer-based LLMs since their inception [35] and the large amount of open-source software publicly available for training, LLMs were soon applied to programming, initially as a super-charged autocomplete with Github Copilot in 2021 [12] powered by the OpenAI's Codex model [7], and later as a conversational partner with the advent of instruction-following LLMs [25]: it is hard to overstate how earth-shattering the public launch of ChatGPT was in 2022 [24].

LLMs improved steadily, with LLMs gaining the ability to think aloud [37] and autonomously plan and enact multi-step workflows [28]. LLM coding ability reached a milestone with the announcement of Devin in 2024 [38], which claimed to be “the first AI software engineer”, followed soon thereafter in academia by SWE-agent [40] targeting “end-to-end software engineering”.<sup>x</sup> Instead of a local autocomplete, these new systems were agentic: autonomously taking step-by-step actions to read, modify, and test whole codebases by iteratively invoking tools (e.g. search, read, edit, run) and fixing errors similar to a human’s edit-run-debug cycle.

Besides broadly capable agents like Devin and SWE-agent, other AI agents have been developed for perhaps every conceivable part of software development, including requirements engineering, debugging, and testing, as reported in surveys [15, 21]. But currently, the widely deployed systems in practical use (Cursor, Github Copilot Agent, Claude Code, Codex CLI) present a broadly capable model by default rather than task-specific agents, although users may still provide specialized prompts to invoke specialized agents (e.g. Claude Code Subagents [2]). But although the model of a broadly capable autonomous agent is widespread and was the first paradigm to make significant progress [40] on the SWE-bench software engineering benchmark suite [14], autonomous choices may not be necessary for such benchmark coding tasks: a carefully crafted pipeline of fixed steps can perform competitively with state of the art [39]. This counter-result, which is effectively a case of extreme prompt engineering, highlights that prompting strategy matters *a lot* for success with current LLMs. Consequently, we would expect large variations in user success with LLMs as well, based on each user’s prompting expertise. Below we discuss some of the usability research on generative AI use for coding.

## 6.2 Coding GenAI Usability

**Prompt Quality** In the studies of generative AI use for coding, the correlation between prompt quality and success is a repeated theme in both the autocomplete and the agentic eras. In the autocomplete era, Lucchetti et al. [22] found that prompts may be robust to word choice but success rate drops dramatically if the prompt is missing key information. This finding is mirrored by Liang et al. [20]’s survey of 410 developers on GitHub, finding the most common prompting strategy is providing “clear explanations”. The students in Akhoroz and Yildirim [1]’s survey also suggested that output quality is improved by using specific prompts, decomposing problems into steps, and iteratively improving a prompt. Expertise also makes a difference: the lab study in Feldman and Anderson [10] showed that non-programmers cannot prompt as successfully as even novice programmers. Similar patterns hold in the agentic era. In lab tasks, Eibl et al. [8] found that key information was left out of 65% prompts, noting that “many participants in our study struggled with crafting ‘good’ prompts”. And in a deployment of an agentic system for resolving real-world software issues, Takerngsaksiri et al. [31] note their number 1 takeaway is that LLM helpfulness “heavily relies on a detailed input description”. Participants in our study as well highlighted that prompts should include clear context and explicit instructions (12x Observations, 43x Survey). Prompt quality is a key bottleneck in LLM usability.

**AI Utility.** At an even broader view, there is the question of whether LLMs are measurably helpful to programmers, as the extra burdens of prompting and code review may overwhelm any gains.

From the autocomplete era, quantitative studies of this question differ in their results. A couple controlled lab studies showed positive results, those with Copilot were 56% faster on a single task in Peng et al. [26] ( $n=95$ , mid-2022), and when experienced Python developers were asked to implement API endpoints for a web server, those with Copilot were much more likely to submit solutions that passed unit tests (60.5% versus 39.5%,  $p < 0.01$ ,  $n=202$ ) [3]. But on the three lab tasks in Vaithilingam et al. [34], participants with Copilot were broadly similar in success rate and speed compared to those without ( $n=24$ ,  $\leq$  Jan 2022), although participants preferred to use Copilot; the authors suspect the lack of measurable gain is because of buggy generated code leading to an increase in debugging time, with participants generally underestimating the effort to fix generated code. In a private retrospective analysis of bug rate and pull request (PR) throughput among a large sample of real-world developers, those with access to Copilot produced more bugs than those without, without any notable difference in PR throughput [18].

In the agentic era, developers believe agentic AI is helpful: of those using AI agents 69% believe agents have increased their productivity [30]. But, quantitative support for gains is currently lacking. Becker et al. [4] studied 16 experienced open source maintainers, allowing them to use or not use AI tools as they performed their usual work (condition randomized per-task,  $n=246$  tasks). While these developers estimated that AI access made them 20% faster, on the whole they were in fact 19% slower when they could use AI. The non-helpfulness may be because this setup was a worse-case scenario for AI: large, complex repositories, where developers are intimately familiar with the code and have tacit knowledge unavailable to the AI, while the developers are also over-optimistic about AI utility [4]. The study measured a low accept rate for generations, <50%, which accords with the 25% accept rate of the agentic tool in Takerngsaksiri et al. [31].

More work is needed to understand when and why AI tools are actually helpful. While our study lists scenarios where experienced developers think AI agents are or are not helpful (Table 4), this study is not a controlled verification of whether agents are measurably accelerating these tasks.

### 6.3 Empirical Studies of Vibe Coding

With the skill of LLMs increasing to the point that AI agents can produce and modify full codebases, it is now becoming possible to, perhaps, disengage with the code entirely and only write prompts. Karpathy [17] dubbed this approach “vibe coding”, where “you fully give in to the vibes, embrace exponentials, and forget that the code even exists...it’s not really coding - I just see stuff, say stuff, run stuff”. The possibility that formal code could become incidental to crafting software is tantalizing, as it potentially opens up the door for non-programmers to write software. A few recent studies explore this new vibe coding phenomena.

Pimenova et al. [27] investigated social media posts and interviewed several people who had tried vibe coding in order to better define and characterize the phenomena. Definitions of vibe coding differ—e.g. how much manual code review is allowed?—but the key feature seems to be aiming for “flow and joy” in the development experience by trusting the AI [27]. This mindset (in our interpretation) leads vibe coders to avoid manual review because it is annoying and kills the vibes. Both Pimenova et al. [27]’s and Fawzy et al. [9]’s investigations of online writeups on vibe coding found that practitioners are aware of the potential lower quality of vibe code, and calibrate their use of it accordingly—even Karpathy’s original tweet introduced vibing only in the context of “throwaway weekend projects” [17]. But, especially with the current AI hype, Fawzy et al. [9] worry that non-programmers become “vulnerable developers” if they try to vibe code software for a setting in which they do not have the expertise to manage the real-world responsibilities of their potentially incorrect code.

The necessity of expertise is supported by Sarkar and Drosos [29]’s analysis of online livestreams of vibe coders ( $n=4$ , 8.5 hours): all four developers studied performed some amount of code review, but relied on their programming expertise to skim the code quickly rather than scrutinizing line-by-line. They also relied on their expertise to choose when to switch to manual coding, e.g. for one-line edits or for debugging. Overall, Sarkar and Drosos [29] believe “vibe coding still requires significant human expertise...especially when the vibes are off”. Similarly, students with different coding expertise prompt differently, in a lab study of vibe coding by CS1 ( $n=9$ ) and upper-level SWE students ( $n=10$ ), the more advanced students provided more context in their prompts and were more likely to work with code [11]. It does not yet seem that pure vibe coding can produce software beyond a certain complexity, Chandrasekaran [6] tried re-creating the same app using three different prompting styles, their best results were by following a less vibey, more disciplined approach, leading them to hypothesize that “production-grade quality requires deliberate oversight”.

Our study did not set out to study vibe coding in particular, but the experienced developers we observed did not vibe. When implementing new features, while they might ask the AI for a first-draft plan, they always planned. And they were careful about the code produced: 69% (9/13) carefully reviewed every change, this accords with the open source developers in Becker et al. [4], 75% of whom said they read *every* line of AI-generated code.

With the continuing improvement of LLMs, vibe coding may be the future, even by non-programmers. But, for complex software, it is not the present.

## 7 Conclusion

*I think AI agents are amazing as long as you are the driver & reviewing its work.  
AI agents become problematic once you’re not making them adhere to engineering principles that have been established for decades.*

- S64

To gain insight into the current practice of agentic coding by experts, we studied experienced developers (3+ years of professional experience) through 13 field observations supplemented with a broader survey of 99 respondents. We aimed to learn about their values and motivations for using agents, workflow strategies, what agents are suitable for, and developer sentiments about using coding agents. We find that experienced developers do not currently vibe code. Instead, they carefully *control* the agents through planning and active supervision because they care about software quality. Although they may not always read code to validate agentic output, they are careful not to lose control and do not let agents run completely autonomously, particularly having the agents only work on a few tasks at a time. They generally enjoy using agents, finding them suitable for accelerating straightforward, repetitive, and scaffolding tasks, including a large variety of software engineering tasks like writing tests, documentation, refactoring, and simple debugging; and yet opinions are mixed about writing plans with agents, and they do not use agents for core business logic or complex tasks. AI capabilities and interfaces are changing fast: this work paints a picture of what is and is not working now, serving as a reference point both to calibrate expectations about current AI as well as to anchor comparison in future years to see how much AI has improved since 2025. As of now, the AIs are not taking over yet—experienced developers are still in control.

## Acknowledgments

Our thanks to Bryan Min, Saketh Kasibatla, and Matthew Beaudouin-Lafon for helpful feedback on the study design and recruitment.

## References

- [1] Mehmet Akhoroz and Caglar Yildirim. 2025. Conversational AI As a Coding Assistant: Understanding Programmers' Interactions with and Expectations From Large Language Models for Coding. *CoRR* abs/2503.16508 (2025). arXiv:2503.16508 doi:10.48550/ARXIV.2503.16508
- [2] Anthropic. 2025. Claude Docs: Subagents. <https://web.archive.org/web/20251003125209/https://docs.claude.com/en/docs/clause-code/sub-agents>
- [3] Jared Bauer. 2024. Does GitHub Copilot improve code quality? Here's what the data says. The Github Blog. <https://github.blog/news-insights/research/does-github-copilot-improve-code-quality-heres-what-the-data-says/>
- [4] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI On Experienced Open-Source Developer Productivity. *CoRR* abs/2507.09089 (2025). arXiv:2507.09089 doi:10.48550/ARXIV.2507.09089
- [5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. doi:10.1191/147808706qp063oa Publisher: Routledge \_eprint: <https://www.tandfonline.com/doi/pdf/10.1191/147808706qp063oa>
- [6] Premanand Chandrasekaran. 2025. Can vibe coding produce production-grade software? Thoughtworks Blog. <https://www.thoughtworks.com/insights/blog/generative-ai/can-vibe-coding-produce-production-grade-software>
- [7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebbgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained On Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 <https://arxiv.org/abs/2107.03374>
- [8] Philipp Eibl, Sadra Sabouri, and Souti Chattopadhyay. 2025. Exploring the Challenges and Opportunities of AI-assisted Codebase Generation. *CoRR* abs/2508.07966 (2025). arXiv:2508.07966 doi:10.48550/ARXIV.2508.07966
- [9] Ahmed Fawzy, Amjad Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review. arXiv:2510.00328 doi:10.48550/ARXIV.2510.00328
- [10] Molly Q. Feldman and Carolyn Jane Anderson. 2024. Non-Expert Programmers In the Generative AI Future. In *Symposium on Human-Computer Interaction for Work (CHIWORK)*. <https://doi.org/10.1145/3663384.3663393>
- [11] Francis Geng, Anshul Shah, Haolin Li, Nawab Mulla, Steven Swanson, Adalbert Gerald Soosai Raj, Daniel Zingaro, and Leo Porter. 2025. Exploring Student-AI Interactions In Vibe Coding. *CoRR* abs/2507.22614 (2025). arXiv:2507.22614 doi:10.48550/ARXIV.2507.22614
- [12] GitHub and Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. GitHub Blog. <https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/>
- [13] Ruanqianqian Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities . In *International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE55347.2025.00232 <https://doi.ieee.org/10.1109/ICSE55347.2025.00232>
- [14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *International Conference on Learning Representations (ICLR)*. <https://openreview.net/forum?id=VTF8yNQM66>
- [15] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs To LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. *CoRR* abs/2408.02479 (2024). arXiv:2408.02479 doi:10.48550/ARXIV.2408.02479
- [16] Greg Kamradt. 2025. I Vibe More Than You: 10x vibe for 0.1x work. <https://www.ivibemorethanyou.com/>
- [17] Andrej Karpathy. 2025. There's a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists... <https://x.com/karpathy/status/1886192184808149383>
- [18] Uplevel Developer Labs. 2024. Can Generative AI Improve Developer Productivity? <https://resources.uplevelteam.com/gen-ai-for-coding>
- [19] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 435–447. doi:10.1109/ICSE48619.2023.000047
- [20] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey On the Usability of AI Programming Assistants: Successes and Challenges. In *International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1145/3597503.3608128>

- [21] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *CoRR* abs/2409.02977 (2024). arXiv:2409.02977 doi:10.48550/ARXIV.2409.02977
- [22] Francesca Lucchetti, Zixuan Wu, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2025. Substance Beats Style: Why Beginning Students Fail To Code with LLMs. In *Nations of the Americas Chapter of the Association for Computational Linguistics (NAACL)*. <https://doi.org/10.18653/v1/2025.nacl-long.433>
- [23] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (Nov. 2019), 23 pages. doi:10.1145/3359174
- [24] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. OpenAI Blog. <https://web.archive.org/web/20230203201352/https://openai.com/blog/chatgpt/>
- [25] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models To Follow Instructions with Human Feedback. In *Conference on Neural Information Processing Systems (NeurIPS)*. [http://papers.nips.cc/paper\\_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html)
- [26] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI On Developer Productivity: Evidence From GitHub Copilot. *CoRR* abs/2302.06590 (2023). arXiv:2302.06590 doi:10.48550/ARXIV.2302.06590
- [27] Veronica Pimenova, Sarah Fakhoury, Christian Bird, Margaret-Anne Storey, and Madeline Endres. 2025. Good Vibrations? A Qualitative Study of Co-Creation, Communication, Flow, and Trust in Vibe Coding. arXiv:2509.12491 doi:10.48550/ARXIV.2509.12491
- [28] Toran Bruce Richards. 2023. Auto-GPT: An Autonomous GPT-4 Experiment. <https://github.com/Significant-Gravitas/AutoGPT/blob/7c0bdcedfadfe6d74b691294a40e0497f47a3eb/README.md>
- [29] Advait Sarkar and Ian Drosos. 2025. Vibe Coding: Programming Through Conversation with Artificial Intelligence. In *Psychology of Programming Interest Group (PPIG 2025)*. <https://doi.org/10.48550/arXiv.2506.23253>
- [30] Stack Overflow. 2025. 2025 Developer Survey: AI. <https://survey.stackoverflow.co/2025/ai/>
- [31] Wannita Takerngsaksiri, Jirat Pasuksmit, Patananom Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2024. Human-In-the-Loop Software Development Agents. *CoRR* abs/2411.12924 (2024). arXiv:2411.12924 doi:10.48550/ARXIV.2411.12924
- [32] thermo. 2025. how i set up 12+ claude 4 code agents to work in parallel mostly autonomously for several hours without conflicts or slop code. <https://x.com/DionysianAgent/status/1929532241534767168>
- [33] Upstash. 2025. Context7 MCP - Up-to-date Code Docs For Any Prompt. <https://github.com/upstash/context7>
- [34] 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 *Conference on Human Factors in Computing Systems (CHI)*. <https://doi.org/10.1145/3491101.3519665>
- [35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Conference on Neural Information Processing Systems (NeurIPS)*. <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fb0d053c1c4a845aa-Abstract.html>
- [36] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and Designing for Trust In AI-powered Code Generation Tools. In *Conference on Fairness, Accountability, and Transparency (FAccT 2024)*. <https://doi.org/10.1145/3630106.3658984>
- [37] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning In Large Language Models. In *Conference on Neural Information Processing Systems (NeurIPS)*. [http://papers.nips.cc/paper\\_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html)
- [38] Scott Wu. 2024. Introducing Devin, the first AI software engineer. <https://cognition.ai/blog/introducing-devin>
- [39] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *CoRR* abs/2407.01489 (2024). arXiv:2407.01489 doi:10.48550/ARXIV.2407.01489
- [40] John Yang, Carlos E. Jimenez, Alexander Wetzig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Conference on Neural Information Processing Systems (NeurIPS)*. [http://papers.nips.cc/paper\\_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html](http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html)
- [41] Steve Yegge. 2025. I've been using up to a dozen coding agents at a time for nearly a week straight. Just added my 13th... [https://x.com/Steve\\_Yegge/status/1965991591367295403](https://x.com/Steve_Yegge/status/1965991591367295403)

## A Appendix

Below are a full list of the GitHub repositories used to invite survey respondents (Sec. A.1) and quotes from survey respondents about prompt strategies (Sec. A.2).

### A.1 Survey Recruitment Sources

Table 5. GitHub repositories from which we invited participants by querying recent commits, pull requests, issues, stargazers, and forks within the past year, following a similar methodology to Liang et al. [20].

Category	Repository
<i>AI/ML Frameworks &amp; Tools</i>	All-Hands-AI/OpenHands, HKUDS/DeepCode, OpenBMB/ChatDev, Significant-Gravitas/AutoGPT, aider-ai/aider, anthropics/claude-code, gpt-engineer-org/gpt-engineer, microsoft/autogen, microsoft/guidance, TabbyML/tabby, TransformerOptimus/SuperAGI, crewAIInc/crewAI, joaomdmoura/crewAI
<i>Development Tools &amp; IDEs</i>	cursor/cursor, cline/cline, stackblitz/bolt.new, Kilo-Org/kilocode
<i>Language Models &amp; AI Research</i>	EleutherAI/gpt-neo, EleutherAI/gpt-neox, facebookresearch/llama, google-research/bert, openai/CLIP, openai/gpt-2, openai/whisper, microsoft/CodeBERT, salesforce/CodeT5
<i>Data Science &amp; Libraries</i>	numpy/numpy, pandas-dev/pandas, scikit-learn/scikit-learn, plotly/plotly.py, chroma-core/chroma
<i>Development Frameworks</i>	continuedev/continue, jerryliu/llama_index, run-llama/llama_index, vercel/ai, torantulino/auto-gpt, yoheinakajima/babyagi
<i>Microsoft Development Tools</i>	microsoft/JARVIS, microsoft/pylance, microsoft/unilm
<i>Other Development Tools</i>	jupyter/notebook, google-gemini/gemini-cli, openai/chatgpt-retrieval-plugin

## A.2 Survey Quotes About Prompting

Usually, I just provide them with the **task details**, but sometimes I need to specify the steps. (S21)

I give **clear instructions** and ask for specific output format (S23)

I first had it **help me develop a "whitepaper"** describing the project and planning out the development in phases (that could be independently evaluated). Then I had it execute the phases. (S38)

It needed some iteration to understand the code and I **provided more guidelines and it got better and better**. (S43)

If given **clear instructions** they do my work (S44)

I do use a lot of prompt tricks, but overall just good communication and explaining in detail. Like, **overexplaining in a LOT of detail**. (S49)

Very tailored prompting. Needs a coding protocol. Needs an SDLC [software development life cycle] protocol. **Lots of context** on code structure, includes PRD, includes coding plan, includes architecture plan, HITL [human in the loop] after each task to avoid drift. (S52)

Usually I give brief explanation of the task on the top, followed by a **list of details**. Also I try to write prompts in Markdown format. (S53)

Please create new feature doing x and y based on base class BackgroundProcess from bin/BackgroundProcess.php and **example implementation** featureA from bin/featureA.php (S57)

The prompts are always very specific, with **as much info as possible** for example - Filenames, function names, variable names, error messages etc. I focus on ensuring prompt has clear statements on what I [want] the LLM to look at and what I want it to do. Small and Specific modifications (S59)

(1) Defined **clear instructions** first. Split to multiple files for maintainability,

(2) Brainstorming first, then start the task,

(3) Refer to files directly [in] the prompt to provide more context if needed (S60)

Before starting a task I give a very thorough prompt of the requirements and context & **I ask for explicit clarification** if any details aren't clear to ensure the agent is on the same page as me. (S64)

As someone who values scalability and security, I make sure to cover **every single edge case** I can think of. I also provide the agent with any sample resources I need it to use for reference, such as code structures and style guides...the agent performs well when it receives **detailed prompts**. (S66)

My prompts were simply about stating the requirements clearly and explaining the problem **thoroughly**. I found that **longer prompts** tend to work better than shorter ones (S72)

Always **treat AI as a smart kid that knows nothing about you and outside world** (more like introvert). So you must write the context, write strict rules (do's and don't), write the guides, and finally write the task. (S75)

1. Direct instruction following

2. Test driven approach

3. Dividing big projects into small tasks and creating **todo lists** (S76)

I maintain a CLAUDE.local.md file with project context and development principles, use descriptive commit messages for context, and leverage git worktrees for parallel development. I focus on **clear, specific requests with relevant file paths and context**, often referencing recent commits and PRs to provide the AI with understanding of the current work (S85)

If you have **clear plan**...would save you much time (S89)

If I provide the right context or my **exact requirement** or function signature, I get more than a 80% success rate. (S92)

Fig. 7. Select quotes from survey respondents who found agents performing well at **following well-defined plans (28:2)**. A common theme is providing clear instructions with as much information as possible. Bolded emphasis is ours for readability.