Position: Humans are Missing from AI Coding Agent Research

Zora Zhiruo Wang[1*] **John Yang**[2*] **Kilian Lieret**[3*]
Alexa Tartaglini[2] **Valerie Chen**[1] **Yuxiang Wei**[4]
Zijian Wang **Lingming Zhang**[4] **Karthik Narasimhan**[3]
Ludwig Schmidt[2] **Graham Neubig**[1] **Daniel Fried**[1] **Diyi Yang**[2]

[1]Carnegie Mellon University [2]Stanford University [3]Princeton University
[4]University of Illinois Urbana-Champaign

[*]Equal Contribution

zhiruow@andrew.cmu.edu, johnby@stanford.edu

Abstract

Recent progress in AI coding agent research has led to rapid improvements in agents' ability to autonomously perform complex software engineering tasks, from editing large codebases to executing long-horizon development workflows. As these systems make strides, however, the primary bottleneck to practical usefulness increasingly shifts away from pure task-solving capability, and toward challenges in how users communicate with, supervise, and trust agents. In this position paper, we argue for a reorientation from autonomous to *human-centered* coding agents: systems designed not only to complete tasks, but to collaborate effectively with people. We identify four core interaction-level dimensions that characterize the human-agent task-solving loop: task alignment, verifiability, steerability, and adaptability. Finally, we outline concrete research directions to advance these dimensions, including user-involved coding environments, comprehensive verification mechanisms, and principled measures of human-agent interaction quality.

Position

AI coding agent research is fixated on autonomous task completion, treating benchmark difficulty as a proxy for practical value. However, as coding agents have graduated from academic papers to industry products, we hypothesize that the next meaningful breakthroughs lie *not* in what agents can do solo, but in their interplay with human developers and users.

1. Introduction

Recent advances in large language models (LLMs) and agent frameworks have led to rapid progress in AI coding research. Coding agents can now modify real codebases, resolve issues in complex repositories (Jimenez et al., 2024), and execute multi-step software engineering workflows (Chan et al., 2025) that were previously out of reach for automated systems. Across benchmarks and live deployments, newer models continue to outperform earlier ones with greater autonomous execution capabilities.

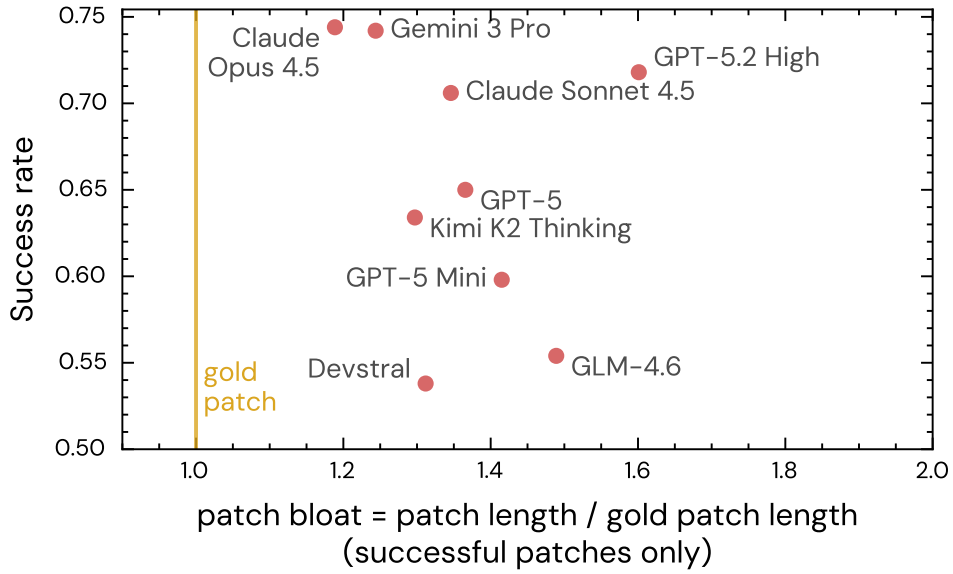

Figure 1: **Agent-generated patches are consistently longer than "ideal" gold patches**, posing a challenge to *verifiability*. In SWE-bench Verified, this pattern persists despite high success rates.

However, as agent autonomy increases, the primary bottleneck to practical usefulness is shifting (Chen et al., 2025b). While earlier systems were limited by their inability to generate correct code, modern agents can produce plausible solutions, as shown by the surging scores on SWE-bench verified in Figure 1. Empirical studies increasingly highlight failures not in producing executable code, but in misunderstanding user intent (Jiang et al., 2022; Liu et al., 2023), producing outputs that are difficult to interpret or verify (Kumar et al., 2025; Treude and Gerosa, 2025), and behaving in ways that are hard to control or predict over time (Chen et al., 2025b; Liang et al., 2024; Mozannar et al., 2024a). For instance, agent-produced patches can be bloated and hard to verify, hindering their practical utility. As coding agents increasingly function as general-purpose agents (Soni et al., 2025) that program to solve diverse real-world tasks beyond software engineering (Wang et al., 2025d), these interaction challenges become even more pronounced. Consequently, as agent capabilities improve, the bottleneck shifts from "can it work?" to "can humans understand, trust, and work with it?", making human-centered design essential for translating capability into practical usefulness.

This mismatch reflects a growing disconnect between agent research and real-world deployment. Much of the field has focused on advancing autonomy, measured by success rate on harder benchmarks. Yet real-world programming is rarely a one-shot activity. Instead, it unfolds through iterative interaction, partial delegation, evolving goals, and continuous human oversight. In this setting, the goal of developing coding agents is not to replace human labor, but to augment complementary human strengths such as initiative and judgment. Crucially, human-centered interaction is not a temporary workaround for immature models: it enables new forms of collaboration in which agents and users jointly discover emerging workflows. As a result, utility depends not only on whether an agent can complete a task, but on whether humans can effectively work with it to shape outcomes and future work (§2).

We argue that this shift in bottlenecks calls for a corresponding adjustment in how coding agents are designed and evaluated: from maximizing agent-solo autonomy to maximizing human-centered usefulness (Position 1). To make this notion concrete, we identify four key dimensions that characterize human-centered coding agents—task alignment, steerability, verifi-

cation, and adaptability—and formalize them with measurable definitions. We choose these dimensions as interaction primitives that together span the agent task-solving cycle (Figure 2): from interpreting user intent, executing actions under human control, exposing outputs for human assessment, to improving behaviors across repeated use. Although they may not constitute an exhaustive list of desirable system properties, these foundational capabilities could support higher-level concepts such as collaboration (§3).

To address these challenges, we outline a few research directions, including building scalable models of human users, enabling more efficient and task-aware verification, defining measurable interaction-centric evaluation signals, and exploring applications for AI coding agents beyond traditional software engineering. Together, these efforts call for new infrastructure, benchmarks, and evaluation settings that incorporate human interaction as an essential component, rather than treating it as an external afterthought (§4).

While we argue that human-centered design is essential for coding agents, this view is not without contention. We engage with common counter-arguments and clarify why these perspectives do not eliminate the need for human-centered agent research (§5). Collectively, they point toward a research agenda centered not on replacing human developers, but on building AI systems that meaningfully augment us.

2. AI for Code Today

Recent coding agent research has largely converged on a single objective: *increasing agent autonomy*, typically measured by an agent's ability to complete standalone software engineering tasks (Chowdhury et al., 2024; Jain et al., 2024). Under this framing, "better" agents are those that can handle harder benchmarks (Deng et al., 2025; Merrill et al., 2026), execute longer horizons (Zhao et al., 2024a), and achieve higher end-to-end success rates. This autonomy-centric perspective has shaped several dominant research directions.

First, substantial effort has gone into *constructing more difficult benchmarks*, which demands understanding across languages (Yang et al., 2025a) and modalities (Yang et al., 2024b), execution in longer horizons (Li et al., 2024; Zhao et al., 2024a) and more complex environments (Chan et al., 2025; Yang et al., 2025b). While these benchmarks are all designed to challenge agents with increasingly complex engineering tasks, they may lead the community to overweigh tasks at the tail-end of the difficulty spectrum.

Second, to solve these increasingly difficult problems, researchers have focused on building *sophisticated agent frameworks* that scaffold LMs with crafted pipelines (Xia et al., 2024) and tools (Wang et al., 2025a; Yang et al., 2024a). Despite differences in architectures, these frameworks share a common goal of pushing agents toward fully autonomous engineering, often limiting human control, which can ultimately constrain their practical utility.

Third, the field has invested in specialized environments and data curation recipes to enable *scalable training with effective verification*. For instance, SWE-Gym established a demonstration-based training formula (Pan et al., 2025), SWE-smith and R2E-Gym automated environment scaling and verification (Jain et al., 2025b; Yang et al., 2025a), and later work diversifies task coverage (Sonwane et al., 2025; Zhu et al., 2025). These training reinforce the autonomy-centric loop, which continuously synthesizes more difficult tasks and trains agents to improve on them.

Although these research have drastically expanded agent capabilities, the assumption that continuing these efforts yields proportional practical gains remains largely unexamined. This gap stems from a mismatch between benchmarks optimized for autonomous behavior and real-

world settings that inevitably involve human interaction. If we continue to prioritize isolated autonomy as the primary axis of progress, research risks overlooking how coding agents are actually used: through communication with humans, oversight and intervention from humans, and adaptation alongside humans. As a result, the field may be optimizing in a direction of diminishing returns, pursuing ever-harder tasks while marginal utility gains shrink, when the gradient toward practical impact points elsewhere: human-agent interaction.

3. What Users Want

Practical deployment of coding agents reveals bottlenecks that differ from those emphasized in automation-oriented agent development. Industry reports and developer surveys from major coding tool providers consistently highlight how users are often less limited by agents' code correctness than by challenges in communicating with the agent, consuming and verifying its outputs, steering its behavior, and the agent adapting to an evolving codebase and user base.

These practical concerns motivate a shift toward human-centered coding systems design. We articulate four key pillars for building effective human-centered coding agents. In addition to qualitative characterization, we formalize these pillars with computable metrics that enable systematic evaluation and optimization. Our goal is not merely conceptual framing, but to provide actionable targets that can be reported in experiments and optimized by future systems.

3.1. Task Alignment

Definition. Task alignment refers to the process by which humans and agents establish and maintain a shared task understanding through mutual modeling (Clark and Brennan, 1991). In AI-assisted coding, users must externalize intent through NL instructions, relevant contexts, or constraints; while agents must infer latent goals, resolve under-specification, surface assumptions, and signal uncertainty. Effective alignment is therefore not a single capability, but a coordinated behavior: dynamically integrating interpretation, clarification, and revision as the task unfolds.

Formulation. Given an NL instruction q_H from user H, the agent C forms an internal task specification $z_C = \text{interpret}_C(q_H|\mathcal{T}) \in \mathcal{Z}$ situated in environment $\mathcal{T}$ with task contexts, where $\mathcal{Z}$ denotes a structured intent space. Task alignment measures distance between the user-intended specification z_H and the agent's inferred z_C,

$$G(H, C; q_H) = \text{sim}_Z(z_H, z_C)$$

where z_H can come from human annotation, and sim measures task similarity (cosine similarity, lexical overlap).

Motivation. Task alignment is key to a smooth user experience for coding agents. Industry has recognized this, producing best-practice guides on prompt engineering, instruction files, and structured frameworks for agent interaction (Cursor, 2026; Ellich and Etcovitch, 2025). Studies confirm the stakes: developers who communicate more effectively with agents see measurably better outcomes (Sarkar, 2025). When communication fails, the consequences compound. Incorrect assumptions, wasted computation, and repeated correction cycles degrade both productivity and trust (Mozannar et al., 2024a). Today, users largely adapt to agents, learning to phrase requests in ways they understand. A significant pain point is the "clarification spiral." Given a task, models launch into implementation without inferring unstated constraints and make incorrect assumptions. Users then reject results and relay additional asks, only to

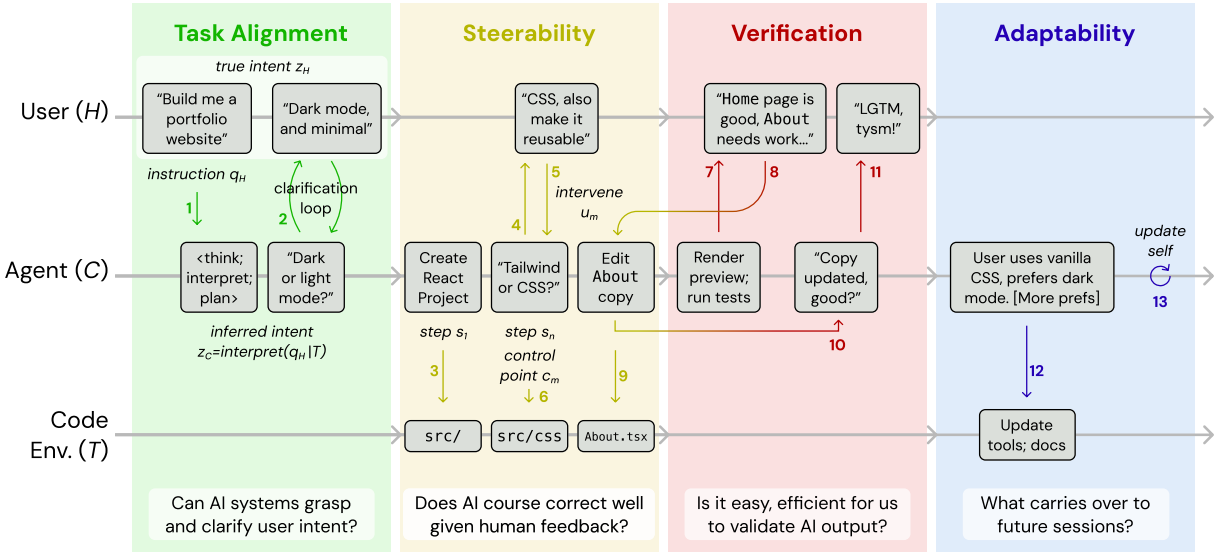

Figure 2: **The Human-Coding Agent Collaboration Loop.** We ground our four pillars in a unified interaction loop between a user, coding agent, and code environment. Using a concrete running example (building a personal portfolio website), we illustrate how task alignment, steerability, verification, and adaptability surface as distinct phases in a single session. The numbers denote an order in which user/agent/codebase interactions occur. Though depicted sequentially for clarity, these phases interleave in practice – task alignment questions naturally may arise during steering; verification may cause an agent to revisit intent and course-correct (step 7).

watch new misunderstandings emerge. What we ultimately want that remains unfulfilled is the reverse: agents that properly model users and interpret tasks.

Research Gap. Coding dialogue is fundamentally more demanding than traditional task-oriented dialogue (Chi et al., 2025b). Whereas a basic request like hotel booking simply requires satisfying a fixed schema of options, programming tasks frequently necessitate diverse communication acts (Ross et al., 2023).

Despite widespread use, open conversation data between humans and AI coding systems remains starkly lacking. Datasets like WildChat and OpenAssistant capture general dialogue patterns (Köpf et al., 2023; Zhao et al., 2024b), but developer dialogue is heavily grounded in external artifacts (code, error logs, documentation) and shaped by tacit expectations around style, architecture, and implementation conventions. Today, benchmarks sidestep the study of such intricacies entirely: pass@k collapses all signal about communicative quality into a single bit, discarding everything about how mutual understanding was (or was not) achieved.

Without open coding conversation data or widely adopted open-source tooling, systematic and quantitative evaluation of grounding quality remains prohibitively out of reach (Vijayvargiya et al., 2025). Communication is instead assessed through anecdotal reports of user frustration (Jiang et al., 2022; Liu et al., 2023), leaving us without benchmarks to reliably track progress over time.

3.2. Steerability

Definition. Steerability concerns an agent's ability to expose and respond to human control signals throughout task execution. Rather than optimizing solely for uninterrupted autonomy (Horvitz, 1999), steerable agents must recognize meaningful decision points in a task, operate at appropriate levels of abstraction, and structure execution to expose decision boundaries at which human intervention can shape downstream behavior. This enables users to redirect subgoals and adjust execution strategies, while preserving agent autonomy over low-level actions.

Formulation. Given a user task q_H, the agent executes an action trajectory $\tau = (s_1, \cdots, s_N)$ where s_i denotes atomic actions. The agent induces a higher-level segmentation $\mathcal{G}_C(\tau) = \{g_1, \cdots, g_M\}$, where each segment $g_m = (\tau_{k_m:k_{m+1}}, c_m)$ consists of a contiguous sub-trajectory and an associated control point $c_m \in C$ (e.g., branching choices, confirmation prompts). To measure agent's responsiveness to human control, let user apply an intervention u_m at the control point c_m, yielding a modified trajectory $\tau' = \text{exec}_C(\tau, u_m)$. Comparing to a reference trajectory $\tau^*_{u_m}$ reflecting the intended behavioral change, we define response steerability as

$$S(H, C) = \mathbb{E}_{(m,u_m)}[\text{sim}_\tau(\tau', \tau^*_{u_m})]$$

While steerability is fundamentally defined by how agent behavior responds to human interventions, we could additionally measure structural alignment of exposed control points as a diagnostic signal. Let $\mathcal{G}^*$ denotes a reference segmentation derived from expert demonstration, we define structural steerability as $S_{\text{struct}}(C) = \text{sim}(\mathcal{G}_C(\tau), \mathcal{G}^*)$, where sim measures alignment of both segmentation boundaries and control point content.

Motivation. Users of AI coding systems consistently express a need to control when and how it acts throughout a task (Kalliamvakou, 2025). Depending on context, users may operate at different points along the granularity spectrum (Sapkota et al., 2025): at times specifying high-level intent for agents to explore broadly via "vibe code" and rapid prototyping (Cursor, 2026; Kalliamvakou, 2024); and at other times intervening at fine-grained levels to make precise, localized changes. However, these needs cannot be met by systems that treat autonomy as a single global setting. Instead, practical steerability requires agents to identify meaningful control points in the task-solving process—moments where alternative execution paths, tradeoffs, or commitments arise—and to expose those choices to the user. By structuring execution around these control points, agents allow users to direct what happens next, rather than forcing them into a binary accept-or-reject role only after a fully specified plan has already been carried out. While task alignment governs what the user means, steerability governs what the system does next.

Research Gap. Steerability remains weakly supported in current coding agent systems: existing work is largely descriptive or peripheral, either documenting developer behavior shifts as agent automation increases (Chen et al., 2025b) or designing human-in-the-loop co-planning (Mozannar et al., 2025), but neither provides a formal account of how agents should structure execution to preserve user control. At present, we lack principled approaches for agents to identify and expose meaningful control points to users, leaving systems trapped between a false dichotomy between full agent autonomy and constant human supervision.

3.3. Verification

Definition. Verification refers to a user's ability to assess if a coding agent's outputs are correct with respect to task requirements. This presupposes that users can meaningfully consume and

understand the agent's artifacts, including code outputs, execution traces, and intermediate reasoning (Vaithilingam et al., 2022). In practice, verification determines whether the outputs satisfy both explicit requirements and implicit constraints (Fakhoury et al., 2024). In other words, verification is about whether agent deliverables expose sufficient structure and evidence for humans to accurately judge correctness.

Formulation. Given a user instruction q_H, the agent produces output o, including intermediate artifacts and final output. A human verifier produces a potentially imperfect judgment $s_H = \text{verify}(o, z_H)$ due to agent output verifiability. Let $y^*(o, z_H)$ denote the ground-truth verifiers (correctness, efficiency, etc.), we define verifiability as the expected agreement between human and gold judgments:

$$V(H, C) = \mathbb{E}_{(o,z_H)}[\mathcal{A}(s_H, y^*(o, z_H))]$$

where $\mathcal{A}$ denotes an agreement metric such as accuracy.

Motivation. Users cannot trust what they cannot verify. Recent surveys find that of the 84% of developers now using AI coding tools, nearly half do not trust outputs while two-thirds report that half-baked solutions lead to heavier debugging burdens (Stack Overflow, 2025). Importantly, current works establish that a user's trust in coding tools is continually recalibrated through repeated verification (Sapkota et al., 2025), a process that shifts meaningfully with task stakes and user expertise (Wang et al., 2024a). For instance, while experienced developers prefer to inspect low-level implementation details, novice programmers may rely on higher-level natural language explanations and observable artifacts (Barke et al., 2023; Gu et al., 2024). Today, the burden is uneven, with less experienced developers reporting both the highest productivity gains and greatest struggle to review coding agent outputs (Sonar, 2026). Supporting verification across this spectrum is therefore critical for enabling reliable and sustained human–agent collaboration.

Research Gap. Unit testing is the dominant verification paradigm in current benchmarks given their precision and reproducibility. However, in practice, it captures only a narrow notion of functional correctness and often shifts the verification burden onto users. Humans rely on a much broader range of strategies to establish trust in code and the systems that generate it, such as code inspection, execution tracing, and iterative dialogue (Mozannar et al., 2024b). This mismatch suggest that while tests are valuable, they may be insufficient for supporting verification in the wild.

First, tests do not always reduce verification effort. Developers rarely construct test suites before prompting AI tools (Sonar, 2026); instead, tests are generated with code. When AI is producing both, the tests no longer serve as independent evidence for building trust. Rather, in addition to code, users must now check if tests capture intent – a lateral shift, not reduction, in cognitive load. To be clear, unit tests are valuable when constructed strategically, by encoding requirements once to avoid repeated verification. But using tests as the default mechanism for any agent output creates work that many interactions do not warrant.

Second, while unit testing has its place, it can be ill-suited for non-technical users and tasks beyond traditional software engineering. Code is increasingly the action space for general-purpose agents, powering exciting use cases such as data analysis (Gu et al., 2024), interface design (Yuan et al., 2025), and tasks on the edge of imagination (e.g., raising a plant[1]). Consequently, outputs are becoming increasingly heterogeneous and context-dependent, making universal pass/fail criteria hard to define (Barr et al., 2014).

Rather than relying solely on test-centric workflows, verification should surface evidence

[1]See the AutonCorp Biodome Project, where Claude autonomously manages a tomato plant via IoT sensors.

in human-interpretable ways, such as visual previews and interactive summaries. Verification needs to be designed in ways that reduce user burden and make agent behavior easier to assess, especially for users without deep programming expertise.

3.4. Adaptability

Definition. We define adaptability as an AI coding agent's ability to maintain and update itself using accumulated experience, to improve future performance while preserving previously acquired capabilities (Wang et al., 2024b). This includes updating persistent memory such as user preferences (Letta, 2025) and task specifications (Wang et al., 2025c), as well as action policies by acquiring and refining reusable skills (Wang et al., 2025b). Beyond storing historic context, highly adaptive agents leverage experience to improve how they perceive the environment and take actions, generalize to related tasks while avoiding undesirable drift.

Formulation. Let τ depict a distribution (user task, preference, etc.), on which the agent aggregates task experiences $e \sim \tau$ and updates itself $C^k = \text{adapt}(C, \{e\}_1^k)$. We quantify adaptability as the improvement over task sessions

$$A(C^k) = \mathbb{E}_{\tau \sim \mathcal{T}}\left[\text{Perf}(C^k) - \text{Perf}(C^0)\right]$$

where $\text{Perf}(\cdot)$ denotes a task performance metric. This formulation captures the agent's ability to improve through experience toward a target distribution.

Motivation. Software engineering is inherently adaptive: Codebases are living artifacts that developers tend to, expand, refactor, and repurpose over time. Users increasingly expect the same continuity and growth from their AI coding tools, yet a common pain point is the need to re-establish context and re-state execution details every session, often described as "prompt fatigue" (Lessard, 2025). Early efforts in building extensions that auto-generate documentation to maintain persistent memory, suggest reductions in user cognitive load and more efficient, personalized interactions (Katz, 2026). Yet adaptability extends beyond remembering contexts to upgrading their actions, by iteratively developing their skill repertoire and reusing them effectively. However, the design of such systems is still in its infancy. Today's agent "memory" and "skills" are often little more than indexed markdown files (Anthropics, 2025), rather than mechanisms for sustained learning, leaving the promise of genuinely adaptive coding agents largely unrealized.

Research gap. Today's coding agents are developed and evaluated on isolated, one-off tasks, with no incentive for learning from prior sessions or accumulating user-specific context. Even when tasks share a repository or creator (e.g., 850 SWE-bench tasks from Django), solving one has no effect on a later task. Simply designing harder problems that require more turns to solve does not obviate the need for persistent context across sessions, a key to sustained support for human engineers. After all, beyond functional code, repositories embody collaboration across developers, accumulating a collective of conventions, preferences, and institutional knowledge over time. Systems that internalize this are most effective at reducing the cognitive burden of repeating context over multiple runs.

When adaptability is studied, it optimizes for agents, not users. Agent Skill Induction (Wang et al., 2025b) and Live-SWE-agent (Xia et al., 2025) show that agents can acquire reusable skills and improve task performance across sessions, but the metric remains squarely on task success, with no notion of whether or how a human copilot might benefit. The gaps come to light when collaboration with a human-in-the-loop extends beyond a single session. A developer tells an agent to use `logging` instead of `print`. Does the lesson persist to next week, or

vanish by the morning? If contradictory preferences are introduced, can the agent sort out the conflicts and ask for confirmation if needed? As discussed in *Motivation*, practitioners are already improvising solutions (`AGENTS.md` file, custom rule sets) with promising results. But systematically evaluating whether current and future proposals improve multi-session human-agent collaboration remains uncharted.

3.5. Isn't there more?

Beyond these four pillars, readers may naturally wonder about other properties, such as safety or proactivity. As discussed in §1, we view additional directions as complementary rather than foundational. They either emerge as compositions of core dimensions (e.g., safety depends on grounding and verification working together) or reflect system-level design choices, not underlying interaction capabilities. Find discussions of several such properties in §B.1.

4. Closing the Gap

To tackle the gaps from §3, rather than prescribing isolated fixes per area, we consider: "What infrastructure missing today blocks progress across multiple dimensions?"

A common bottleneck across all pillars is that our formulations require sampling distributions of human behavior – intent, judgments, interventions – that current open-source pipelines do not provide at scale. Furthermore, we lack task structures that truly require interaction to succeed and demand verification approaches beyond unit tests, particularly in application domains beyond software engineering.

We identify four high-leverage research directions, each targeting a bottleneck that stalls multiple fronts simultaneously. For each, we discuss how progress is being throttled, then propose potential solutions.

4.1. Scale human modeling

Today, researchers face an unfortunate dilemma when evaluating human-AI collaboration: either run expensive human studies that do not scale, or default to fully autonomous benchmarks that sidestep the human entirely. The machine learning community has largely chosen the latter. But for genuine progress on human-centered coding agents, there is no shortcut. The formulations in §3 all require sampling from human distributions: specification ($\mathcal{G}$) necessitates user intent z_H, verification ($\mathcal{V}$) depends on human judgments s_H, and steerability ($\mathcal{S}$) relies on human intervention u_m.

To address this, the first path is to construct executable environments with (simulated) users and establish training pipelines accordingly. This begins with developing *user simulators* that faithfully model humans as active software developers and users (Yao et al., 2024). A key challenge is that current simulators prompted to "act like a human" behave homogeneously and overly cooperative (Li et al., 2017). More prompting alone is not sufficient: faithful simulation requires modeling personal preferences, expertise, and realistic failure modes, since real users often hold implicit expectations they may not always verbalize and change their minds mid-task (Naous et al., 2025; Shi et al., 2025). One approach is to leverage large-scale developer profiles and interactions available on GitHub (Figure 3), such as pull request histories, even real user-agent interaction data, which together capture user behavioral and preferential patterns to train more realistic interlocutors.

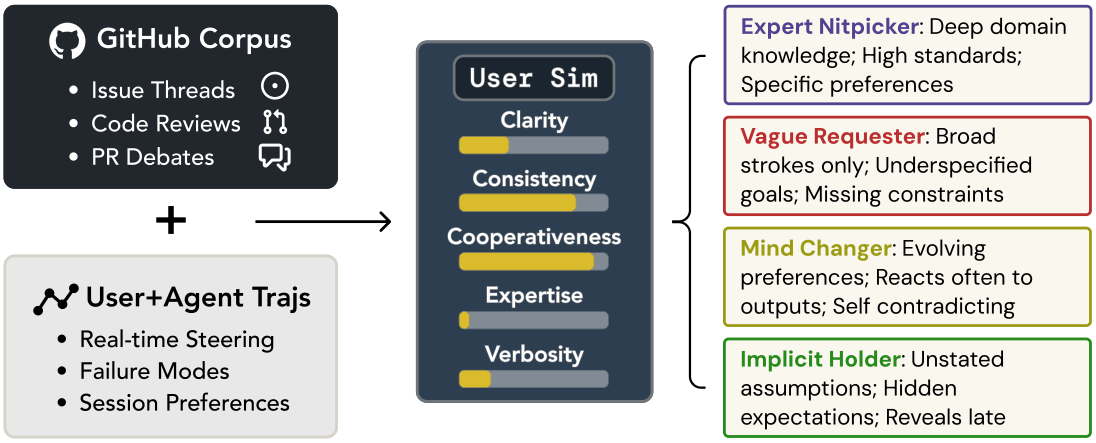

Figure 3: **Generating Diverse Simulated Users.** We envision combining GitHub interaction data with observed human-agent trajectories to parameterize user simulators along dimensions like clarity, consistency, and expertise.

A complementary approach is to build widely adopted user-facing platforms that *collect interaction data as a byproduct of real use*. For instance, Copilot Arena integrates into IDEs and gathers large-scale preference signals on daily coding tasks that substantially diverge from model rankings from static benchmarks (Chi et al., 2025a). We encourage researchers and industry to partner and share anonymized interaction data to enable scalable research.

Ultimately, measuring and improving the fidelity of LM-based simulators remains an open challenge. How do we evaluate alignment to user intent and behavior, ensure diversity without collapsing to stereotypes, and effectively combine simulated and real interaction signals?

4.2. Enable efficient oversight

As discussed in §3.2 and §3.3, today's users verify agent outputs in tedious and manual ways. Proxies of code correctness, such as unit tests, scale poorly in the wild and may not be well-suited to the task or end user's expertise.

Rather than placing the full burden on users, coding systems should proactively support oversight (Chen et al., 2025c; Raghavendra et al., 2026; Sun et al., 2025; Zhao et al., 2025). Recent work has explored generate relevant tests without explicit requests (Chen et al., 2022; Jain et al., 2025a; Mündler et al., 2024) and self-checking outputs prior to delivery (Chen et al., 2023; Ni et al., 2023).

Supplanting such traditional methods, we envision verification becoming a dynamic, protean procedure. Conditioned on task and user, agents should reason about notions of quality and surface appropriate artifacts to make validation tractable, as imagined in Figure 4. A data pipeline implementation might warrant a `.md` file summarizing key steps (e.g., how were missing values handled). On the other hand, code for a webpage should be presented visually for humans to assess design quality at a glance.

Several open questions remain. How should we evaluate the quality of verification approaches themselves? What communicative cost can users bear before cognitive load degrades their judgment? And when should agents self-verify versus surface as artifacts for human review?

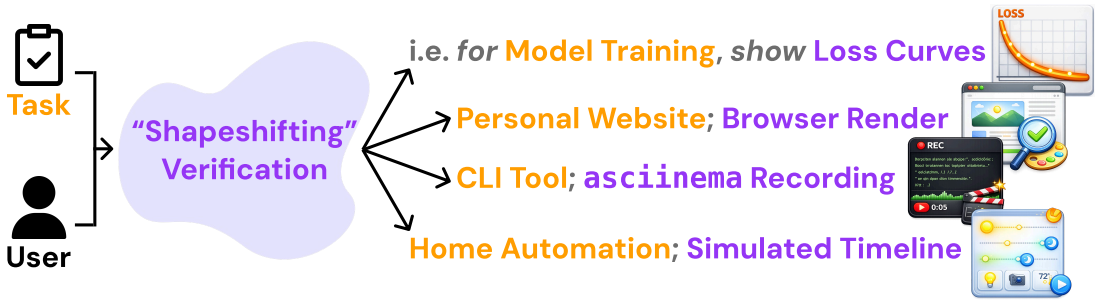

Figure 4: **Context should determine the "shape" of oversight.** Rather than defaulting to unit tests or code diffs, AI systems should surface artifacts that make validation intuitive.

4.3. Define measures for interaction

The current focus on task resolution leaves interaction quality unmeasured and unoptimized. §3's formulations all involve expectations over human behavior, such as intent (z_H), judgments (s_H), or interventions (u_M). We hypothesize that operationalizing such latent variables has value. How many turns does it take to recover z_H? How often is u_M required? How much human effort does s_H demand?

Fortunately, we need not start from scratch. Decades of HCI and software engineering research offer rich precedent for defining and operationalizing interaction quality from user studies (Sarkar et al., 2022; Sergeyuk et al., 2024; Sun et al., 2022). Mozannar et al. (2024a) introduces the CUPS taxonomy, a categorization of programmer-AI interaction into 12 discrete states (e.g., "Prompt Crafting", "Editing Recent Suggestions"), which they then use to measure the distribution of time and transitions across these states. Barke et al. (2023) distinguishes acceleration mode (user knows what to do, measures time-to-completion) from exploration mode (user uncertain, measures validation effort and suggestion-browsing depth). Chen et al. (2025a) proposes the PULSE framework to operationalize user satisfaction as a training signal, using models to predict satisfaction from interaction traces and project the effect of agent design choices. Literature in such fields is ripe with such gems; model trainers and agent builders could port and operationalize them at scale.

Beyond adapting existing frameworks, there is gold to be found in large-scale trajectory analysis. Cursor's Tab RL work (Jackson et al., 2025) illustrates this: by mining 400M+ daily accept/reject decisions, they discovered that *when* to suggest matters as much as *what* to suggest. This insight is invisible to pass-rate benchmarks. Infrastructure for agent trajectory analysis (Bouzenia and Pradel, 2025; Dunlap, 2025; Meng et al., 2025) presents an exciting opportunity to rapidly define and validate human-centered interaction metrics like collaboration quality and user satisfaction.

4.4. Go beyond software engineering

Coding agents are general agents (Soni et al., 2025). Any task expressible programmatically becomes fair game. Manage a stock portfolio. Control a smart home. Tend a greenhouse. Plan a wedding. Compose music. People are already using coding agents for these and more. This shift demands that agent design move beyond tools optimized for professional developers to support everyday users.

What makes such domains compelling is that the four pillars arise intrinsically. Grounding user intent is essential: when an agent managing a portfolio is told "be more conservative", this

means different things from a retiree versus a college graduate. Orchestrating a smart home foregrounds verification; if “unlock the front door" or “turn on the stove" leads to mistakes, the damage can’t be undone with a simple `git revert`. Steerability is indispensable for planning tasks, where flights get canceled, open slots get booked, and user preferences change. Consider monitoring a greenhouse. The right watering schedule depends on this morning’s humidity, but optimal planting strategy is best informed by last season’s growing cycles. An agent must digest feedback signals spanning hours to months, deciding which timescales to attend for each decision. This is adaptability at its purest.

5. Alternative Views

We dedicate the following section to addressing meaningful points of contention to arguments put forth by our position.

(a) *Given the rapid rate of progress, AI will take over coding. Studying how humans interact and collaborate with AI coding systems is a fleeting need.*

Anthropic CEO Dario Amodei famously stated in March 2025 that in the near future, AI will write 90% of code (CFR, 2025), a declaration representative of a general sentiment that ongoing improvements in models’ coding capabilities will eventually stamp out any need for human effort in software development (Collins, 2024; Simon, 1965).

We posit that whether or not this future materializes, the study of how humans participate and benefit from an AI-enhanced coding loop will remain relevant. Even if Dr. Amodei’s prediction bears out, humans do not exit the loop. As agents carry out increasingly complex tasks, more decisions must be made, many of which are personalized, organization-specific, or depend on information only a specific individual possesses. The locus of effort simply shifts upstream, from implementation to specification and evaluation. Code, after all, is simply the language we use to communicate with machines – one that continues to evolve, from assembly to C to Python. Assuming agents bear more responsibility for writing Python, developers may instead convey intent with natural language or perhaps a new formalism balancing human expressiveness with programmatic precision. But the task of coding, fundamentally how we communicate with machines, does not vanish; it transforms.

Present-day trends already reflect this. Concurrent with the emergence of more end-to-end coding agents (Anthropic, 2025; OpenAI, 2025), tools designed to aid developers, not replace them, have proliferated: for reviewing (Graphite, 2025; Greptile, 2025), editing front-ends (Ginsberg and Lu, 2025), completion (GitHub), and debugging (Levin et al., 2024; Zhou et al., 2025a). The spectrum of tools coming to market suggests that users not only prefer copilots in some cases, but also desire different interfaces for different tasks. Both in-IDE auto-complete and a visual editor can be used to modify webpage elements, but which a developer reaches for depends on the task, their expertise, and preferences. Beyond tools, the outcomes also embed personal taste. Should a button be red or blue? Styled with inline CSS or reusable classes? Accessed via a modal or new page?

(b) *Human interaction and evaluation are too costly to scale.*

Human evaluation for natural language generation systems is challenging (Celikyilmaz et al., 2020). Studies are expensive and time-consuming, particularly for tasks requiring heavy domain expertise like coding (Howcroft et al., 2020). A lack of standards around how to design, execute, and report human evaluations can easily lead to irreproducible results (Van der Lee et al., 2021).

These concerns, while valid, can be overcome. As covered in §4, just as the NLP community developed scalable proxies of human feedback to improve instruction following and alignment (e.g., reward models, LM-as-judge (Dubois et al., 2023; Zheng et al., 2023)), metrics such as cyclomatic complexity (McCabe, 1976), cognitive complexity (Campbell, 2018), maintainability index (Oman and Hagemeister, 1992) and readability (Buse and Weimer, 2008, 2009) are computable and tangibly alleviate maintenance burden. More ambitiously, recent works have showcased the viability of benchmarks with user simulators (Yao et al., 2024) and preference collection through organic usage (Chiang et al., 2024) as paths towards scalable human-centered evaluation. The challenge of standardizing such nascent approaches is precisely why now is the time to act (Shao et al., 2025b).

(c) *Capabilities first. Human-centered concerns are product problems that model builders shouldn't get distracted by.*

The Bitter Lesson argues that general methods leveraging computation ultimately supersede manually crafted approaches based on human knowledge of a domain (Sutton, 2019). If anything, the current market reflects this division of labor. Researchers optimize capabilities, then product builders graft the appropriate user interfaces downstream.

Our position is not anti-scaling. Rather, we agree, with the important caveat that what we're actually scaling toward matters. If we continue optimizing solely for autonomous coding agents, we will produce just that. Better collaborators will not emerge for free (Shao et al., 2025a; Shen et al., 2025; Weston and Foerster, 2025; Wu et al., 2025; Zhou et al., 2025b). The leap from GPT-3 to ChatGPT illustrates this point. Incorporating human preferences into the training objective itself transformed a research artifact into a gripping conversationalist (Ouyang et al., 2022). Deferring human-centered concerns to downstream product work risks entrenching interaction patterns that are difficult to undo.

6. Conclusion

We contend that the omission of consideration for humans in AI coding agent research threatens to undermine the very utility these systems seek to provide. Left unchecked, the gap between AI coding agent research and real-world utility may very well widen. The research community must decide whether to optimize for leaderboards or the people who actually use these systems as well. Otherwise, we run the risk of building ever-more-capable coding agents that fewer people know how to wield.

Impact Statements

We highlight two potential impacts of this position paper on the research community and broader society.

First, our work aims to point out a potential symbiosis between HCI research and agent development in the ML community. While HCI studies how people interact with AI coding systems, much agent research continues to prioritize autonomous task completion. By introducing measurable human-centered objectives, we provide a shared framework that can foster closer integration across ML, HCI, software engineering, and related fields.

Second, our position reframes automation in software work from maximizing autonomy to supporting meaningful human interaction. Rather than replacing programmers, human-centered coding agents can amplify human judgment and initiative while remaining controllable

and interpretable. By advocating for accessible interaction and verification mechanisms, this approach also broadens access to programming capabilities for non-experts, enabling wider participation in AI-powered tools. More in §B.2.

Acknowledgments

We would like to thank members of Stanford NLP, the SALT Lab, and the Language Technologies Institute at CMU for their helpful discussions and insightful feedback on the project. We would also like to thank Mike Merrill, Ofir Press, Alexander Wettig, Wenting Zhao, and many attendees of the Deep Learning for Code (DL4C) workshop for providing thoughts and opinions that were formative to the shape of the arguments we put forth. Zora Zhiruo Wang is supported by Google PhD Fellowship. This work was supported by an HAI grant, DSO lab, Open Philanthropy, Schmidt Sciences, and a grant under the NSF CAREER IIS-2247357 and ONR N00014-24-1-2532.

References

Anthropic. Claude code, 2025. URL `https://code.claude.com/docs/en/overview`.

Anthropics. Skills. `https://github.com/anthropics/skills`, 2025.

S. Barke, M. B. James, and N. Polikarpova. Grounded copilot: How programmers interact with code-generating models. Proceedings of the ACM on Programming Languages, 7(OOPSLA1): 85–111, 2023.

E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. IEEE transactions on software engineering, 41(5):507–525, 2014.

I. Bouzenia and M. Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. arXiv preprint arXiv:2506.18824, 2025.

R. P. Buse and W. R. Weimer. A metric for software readability. In Proceedings of the 2008 international symposium on Software testing and analysis, pages 121–130, 2008.

R. P. Buse and W. R. Weimer. Learning a metric for code readability. IEEE Transactions on software engineering, 36(4):546–558, 2009.

G. A. Campbell. Cognitive complexity: An overview and evaluation. In Proceedings of the 2018 international conference on technical debt, pages 57–58, 2018.

A. Celikyilmaz, E. Clark, and J. Gao. Evaluation of text generation: A survey. arXiv preprint arXiv:2006.14799, 2020.

CFR. Ceo speaker series with dario amodei of anthropic, 2025. URL `https://www.cfr.org/event/ceo-speaker-series-dario-amodei-anthropic`.

J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, L. Weng, and A. Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL `https://arxiv.org/abs/2410.07095`.

B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397, 2022.

V. Chen, R. Malhotra, X. Wang, J. Michelini, X. Zhou, A. B. Soni, H. H. Tran, C. Smith, A. Talwalkar, and G. Neubig. How can we assess human-agent interactions? case studies in software agent design. arXiv preprint arXiv:2510.09801, 2025a.

V. Chen, A. Talwalkar, R. Brennan, and G. Neubig. Code with me or for me? how increasing ai automation transforms developer workflows. arXiv preprint arXiv:2507.08149, 2025b.

V. Chen, A. Zhu, S. Zhao, H. Mozannar, D. Sontag, and A. Talwalkar. Need help? designing proactive ai assistants for programming, 2025c. URL https://arxiv.org/abs/2410.04596.

X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128, 2023.

W. Chi, V. Chen, A. N. Angelopoulos, W.-L. Chiang, A. Mittal, N. Jain, T. Zhang, I. Stoica, C. Donahue, and A. Talwalkar. Copilot arena: A platform for code llm evaluation in the wild, 2025a. URL https://arxiv.org/abs/2502.09328.

W. Chi, V. Chen, R. Shar, A. Mittal, J. Liang, W.-L. Chiang, A. N. Angelopoulos, I. Stoica, G. Neubig, A. Talwalkar, et al. Edit-bench: Evaluating llm abilities to perform real-world instructed code edits. arXiv preprint arXiv:2511.04486, 2025b.

W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, B. Zhu, H. Zhang, M. Jordan, J. E. Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. In Forty-first International Conference on Machine Learning, 2024.

N. Chowdhury, J. Aung, C. J. Shern, O. Jaffe, D. Sherburn, G. Starace, E. Mays, R. Dias, M. Aljubeh, M. Glaese, et al. Introducing swe-bench verified. arXiv preprint arXiv:2407.01489, 2024.

H. H. Clark and S. E. Brennan. Grounding in communication. 1991.

B. Collins. Nvidia ceo predicts the death of coding - jensen huang says ai will do the work, so kids don't need to learn, 2 2024. URL https://www.techradar.com/pro/nvidia-ceo-predicts-the-death-of-coding-jensen-huang-says-ai-will-do-the-work-so-kids-dont-need-to-learn.

Cursor. Best practices for coding with agents. Cursor Blog, 2026. URL https://cursor.com/blog/agent-best-practices. Accessed: 2026-01-10.

X. Deng, J. Da, E. Pan, Y. Y. He, C. Ide, K. Garg, N. Lauffer, A. Park, N. Pasari, C. Rane, K. Sampath, M. Krishnan, S. Kundurthy, S. Hendryx, Z. Wang, V. Bharadwaj, J. Holm, R. Aluri, C. B. C. Zhang, N. Jacobson, B. Liu, and B. Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Y. Dubois, C. X. Li, R. Taori, T. Zhang, I. Gulrajani, J. Ba, C. Guestrin, P. S. Liang, and T. B. Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. Advances in Neural Information Processing Systems, 36:30039–30069, 2023.

L. Dunlap. StringSight: Automatically analyze your model traces. https://github.com/lisadunlap/StringSight, 2025. Accessed: 2026-01-20.

B. Ellich and J. Etcovitch. WRAP up your backlog with GitHub copilot coding agent. GitHub Blog, 2025. URL https://github.blog/ai-and-ml/github-copilot/wrap-up-your-backlog-with-github-copilot-coding-agent/. Accessed: 2026-01-11.

S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri. Llm-based test-driven interactive code generation: User study and empirical evaluation. IEEE Transactions on Software Engineering, 2024.

J. Ginsberg and R. Lu. A visual editor for the cursor browser, 2025. URL https://cursor.com/blog/browser-visual-editor.

GitHub. Github copilot · your ai pair programmer. URL https://github.com/features/copilot.

Graphite. Graphite: The next generation of code review., 2025. URL https://graphite.com/.

Greptile. Greptile, 2025. URL https://www.greptile.com/.

K. Gu, R. Shang, T. Althoff, C. Wang, and S. M. Drucker. How do analysts understand and verify ai-assisted data analyses? In Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, pages 1–22, 2024.

J. He, C. Treude, and D. Lo. Llm-based multi-agent systems for software engineering: Literature review, vision and the road ahead, 2025. URL https://arxiv.org/abs/2404.04834.

S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

E. Horvitz. Principles of mixed-initiative user interfaces. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems, pages 159–166, 1999.

D. M. Howcroft, A. Belz, M.-A. Clinciu, D. Gkatzia, S. A. Hasan, S. Mahamood, S. Mille, E. van Miltenburg, S. Santhanam, and V. Rieser. Twenty years of confusion in human evaluation: NLG needs evaluation sheets and standardised definitions. In B. Davis, Y. Graham, J. Kelleher, and Y. Sripada, editors, Proceedings of the 13th International Conference on Natural Language Generation, pages 169–182, Dublin, Ireland, Dec. 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.inlg-1.23. URL https://aclanthology.org/2020.inlg-1.23/.

J. Jackson, P. Kravtsov, and S. Jain. Improving cursor tab with online rl, 2025. URL https://cursor.com/blog/tab-rl.

K. Jain, G. Synnaeve, and B. Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2025a. URL https://arxiv.org/abs/2410.00752.

N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

N. Jain, J. Singh, M. Shetty, L. Zheng, K. Sen, and I. Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. arXiv preprint arXiv:2504.07164, 2025b.

E. Jiang, E. Toh, A. Molina, K. Olson, C. Kayacik, A. Donsbach, C. J. Cai, and M. Terry. Discovering the syntax and strategies of natural language programming with generative language models. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, pages 1–19, 2022.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

E. Kalliamvakou. A developer's second brain: Reducing complexity through partnership with AI. GitHub Blog, 2024. URL https://github.blog/news-insights/research/a-developers-second-brain-reducing-complexity-through-partnership-with-ai/. Accessed: 2026-01-10.

E. Kalliamvakou. The new identity of a developer: What changes and what doesn't in the AI era. GitHub Blog, 2025. URL https://github.blog/news-insights/octoverse/the-new-identity-of-a-developer-what-changes-and-what-doesnt-in-the-ai-era/. Accessed: 2026-01-10.

J. Katz. Dynamic context discovery. Cursor Blog, 2026. URL https://cursor.com/blog/dynamic-context-discovery. Accessed: 2026-01-10.

A. Köpf, Y. Kilcher, D. Von Rütte, S. Anagnostidis, Z. R. Tam, K. Stevens, A. Barhoum, D. Nguyen, O. Stanley, R. Nagyfi, et al. Openassistant conversations-democratizing large language model alignment. Advances in neural information processing systems, 36:47669–47681, 2023.

A. Kumar, Y. Bajpai, S. Gulwani, G. Soares, and E. Murphy-Hill. Why ai agents still need you: Findings from developer-agent collaborations in the wild. arXiv preprint arXiv:2506.12347, 2025.

J. Lessard. Do AI coding assistants really save time to developers? Axify Blog, 2025. URL https://axify.io/blog/are-ai-coding-assistants-really-saving-developers-time. Accessed: 2026-01-10.

Letta. Letta code: The memory-first coding agent, 2025. URL https://github.com/letta-ai/letta-code.

N. G. Leveson. Engineering a safer world: Systems thinking applied to safety. The MIT Press, 2016.

K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund. Chatdbg: An ai-powered debugging assistant. arXiv preprint arXiv:2403.16354, 2024.

B. Li, W. Wu, Z. Tang, L. Shi, J. Yang, J. Li, S. Yao, C. Qian, B. Hui, Q. Zhang, Z. Yu, H. Du, P. Yang, D. Lin, C. Peng, and K. Chen. Prompting large language models to tackle the full software development lifecycle: A case study, 2024. URL https://arxiv.org/abs/2403.08604.

X. Li, Z. C. Lipton, B. Dhingra, L. Li, J. Gao, and Y.-N. Chen. A user simulator for task-completion dialogues, 2017. URL https://arxiv.org/abs/1612.05688.

J. T. Liang, C. Yang, and B. A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In Proceedings of the 46th IEEE/ACM international conference on software engineering, pages 1–13, 2024.

M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon. "what it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, pages 1–31, 2023.

T. J. McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

K. Meng, V. Huang, J. Steinhardt, and S. Schwettmann. Introducing docent. https://transluce.org/introducing-docent, March 2025.

M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, pages 1–16, 2024a.

H. Mozannar, V. Chen, M. Alsobay, S. Das, S. Zhao, D. Wei, M. Nagireddy, P. Sattigeri, A. Talwalkar, and D. Sontag. The realhumaneval: Evaluating large language models' abilities to support programmers, 2024b. URL https://arxiv.org/abs/2404.02806.

H. Mozannar, G. Bansal, C. Tan, A. Fourney, V. Dibia, et al. Magentic-ui: Towards human-in-the-loop agentic systems. arXiv preprint arXiv:2507.22358, 2025. URL https://arxiv.org/abs/2507.22358.

N. Mündler, M. N. Mueller, J. He, and M. Vechev. SWT-bench: Testing and validating real-world bug-fixes with code agents. In The Thirty-eighth Annual Conference on Neural Information Processing Systems, 2024. URL https://openreview.net/forum?id=9Y8zUO11EQ.

T. Naous, P. Laban, W. Xu, and J. Neville. Flipping the dialogue: Training and evaluating user language models, 2025. URL https://arxiv.org/abs/2510.06552.

A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin. Lever: Learning to verify language-to-code generation with execution. In International Conference on Machine Learning, pages 26106–26128. PMLR, 2023.

P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In Proceedings Conference on Software Maintenance 1992, pages 337–338. IEEE Computer Society, 1992.

OpenAI. Codex: One agent for everywhere you code, 2025. URL https://openai.com/codex/.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. Advances in neural information processing systems, 35:27730–27744, 2022.

J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang. Training software engineering agents and verifiers with swe-gym, 2025. URL https://arxiv.org/abs/2412.21139.

M. Raghavendra, A. Gunjal, B. Liu, and Y. He. Agentic rubrics as contextual verifiers for swe agents. arXiv preprint arXiv:2601.04171, 2026.

S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In Proceedings of the 28th International Conference on Intelligent User Interfaces, pages 491–514, 2023.

R. Sapkota, K. I. Roumeliotis, and M. Karkee. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. arXiv preprint arXiv:2505.19443, 2025. URL https://arxiv.org/abs/2505.19443.

A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn. What is it like to program with artificial intelligence? arXiv preprint arXiv:2208.06213, 2022.

S. K. Sarkar. Ai agents, productivity, and higher-order thinking: Early evidence from software development. Available at SSRN 5713646, 2025.

A. Sergeyuk, S. Titov, and M. Izadi. In-ide human-ai experience in the era of large language models; a literature review. In Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments, pages 95–100, 2024.

Y. Shao, V. Samuel, Y. Jiang, J. Yang, and D. Yang. Collaborative gym: A framework for enabling and evaluating human-agent collaboration, 2025a. URL https://arxiv.org/abs/2412.15701.

Y. Shao, H. Zope, Y. Jiang, J. Pei, D. Nguyen, E. Brynjolfsson, and D. Yang. Future of work with ai agents: Auditing automation and augmentation potential across the u.s. workforce, 2025b. URL https://arxiv.org/abs/2506.06576.

S. Z. Shen, V. Chen, K. Gu, A. Ross, Z. Ma, J. Ross, A. Gu, C. Si, W. Chi, A. Peng, J. J. Shen, A. Talwalkar, T. Wu, and D. Sontag. Completion ≠ collaboration: Scaling collaborative effort with agents, 2025. URL https://arxiv.org/abs/2510.25744.

Q. Shi, C. E. Jimenez, S. Dong, B. Seo, C. Yao, A. Kelch, and K. Narasimhan. Impersona: Evaluating individual level lm impersonation, 2025. URL https://arxiv.org/abs/2504.04332.

H. A. Simon. The shape of automation for men and management, volume 13. Harper & Row New York, 1965.

Sonar. State of code: Developer survey 2026, Jan. 2026. URL https://www.sonarsource.com/the-state-of-code/.

A. B. Soni, B. Li, X. Wang, V. Chen, and G. Neubig. Coding agents with multimodal browsing are generalist problem solvers. arXiv preprint arXiv:2506.03011, 2025.

A. Sonwane, I. White, H. Lee, M. Pereira, L. Caccia, M. Kim, Z. Shi, C. Singh, A. Sordoni, M.-A. Côté, and X. Yuan. Bugpilot: Complex bug generation for efficient learning of swe skills, 2025. URL https://arxiv.org/abs/2510.19898.

Stack Overflow. 2025 stack overflow developer survey. https://survey.stackoverflow.co/2025/, 2025. 49,000+ respondents from 177 countries.

J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz. Investigating explainability of generative ai for code through scenario-based design, 2022. URL https://arxiv.org/abs/2202.04903.

W. Sun, X. Zhou, W. Du, X. Wang, S. Welleck, G. Neubig, M. Sap, and Y. Yang. Training proactive and personalized llm agents, 2025. URL https://arxiv.org/abs/2511.02208.

R. Sutton. The bitter lesson. Incomplete Ideas (blog), 13(1):38, 2019.

C. Treude and M. A. Gerosa. How developers interact with ai: A taxonomy of human-ai collaboration in software engineering. In 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), pages 236–240. IEEE, 2025.

P. Vaithilingam, T. Zhang, and E. L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Chi conference on human factors in computing systems extended abstracts, pages 1–7, 2022.

C. Van der Lee, A. Gatt, E. Van Miltenburg, and E. Krahmer. Human evaluation of automatically generated text: Current trends and best practice guidelines. Computer Speech & Language, 67:101151, 2021.

S. Vijayvargiya, X. Zhou, A. Yerukola, M. Sap, and G. Neubig. Interactive agents to overcome ambiguity in software engineering, 2025. URL https://arxiv.org/abs/2502.13069.

R. Wang, R. Cheng, D. Ford, and T. Zimmermann. Investigating and designing for trust in ai-powered code generation tools. In Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency, pages 1475–1493, 2024a.

X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025a. URL https://arxiv.org/abs/2407.16741.

Z. Wang, D. Fried, and G. Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024b. URL https://arxiv.org/abs/2401.12869.

Z. Z. Wang, A. Gandhi, G. Neubig, and D. Fried. Inducing programmatic skills for agentic tasks, 2025b. URL https://arxiv.org/abs/2504.06821.

Z. Z. Wang, J. Mao, D. Fried, and G. Neubig. Agent workflow memory. In Forty-second International Conference on Machine Learning, 2025c. URL https://openreview.net/forum?id=NTAhi2JEEE.

Z. Z. Wang, Y. Shao, O. Shaikh, D. Fried, G. Neubig, and D. Yang. How do ai agents do human work? comparing ai and human workflows across diverse occupations, 2025d. URL https://arxiv.org/abs/2510.22780.

J. Weston and J. Foerster. Ai & human co-improvement for safer co-superintelligence. arXiv preprint arXiv:2512.05356, 2025.

S. Wu, M. Galley, B. Peng, H. Cheng, G. Li, Y. Dou, W. Cai, J. Zou, J. Leskovec, and J. Gao. Collabllm: From passive responders to active collaborators, 2025. URL https://arxiv.org/abs/2502.00640.

C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

C. S. Xia, Z. Wang, Y. Yang, Y. Wei, and L. Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly?, 2025. URL https://arxiv.org/abs/2511.13646.

J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024a. URL https://arxiv.org/abs/2405.15793.

J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, D. Yang, S. I. Wang, and O. Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

J. Yang, K. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang. Swe-smith: Scaling data for software engineering agents, 2025a. URL https://arxiv.org/abs/2504.21798.

J. Yang, K. Lieret, J. Yang, C. E. Jimenez, O. Press, L. Schmidt, and D. Yang. Codeclash: Benchmarking goal-oriented software engineering, 2025b. URL https://arxiv.org/abs/2511.00839.

S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

M. Yuan, J. Chen, Y. Hu, S. Feng, M. Xie, G. Mohammadi, Z. Xing, and A. J. Quigley. Towards human-ai synergy in ui design: Supporting iterative generation with llms. ACM Transactions on Computer-Human Interaction, 2025.

S. Zhao, A. Zhu, H. Mozannar, D. Sontag, A. Talwalkar, and V. Chen. Codinggenie: A proactive llm-powered programming assistant, 2025. URL https://arxiv.org/abs/2503.14724.

W. Zhao, N. Jiang, C. Lee, J. T. Chiu, C. Cardie, M. Gallé, and A. M. Rush. Commit0: Library generation from scratch, 2024a. URL https://arxiv.org/abs/2412.01769.

W. Zhao, X. Ren, J. Hessel, C. Cardie, Y. Choi, and Y. Deng. Wildchat: 1m chatgpt interaction logs in the wild. arXiv preprint arXiv:2405.01470, 2024b.

L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in neural information processing systems, 36:46595–46623, 2023.

A. Zhou, M. Ramesh, and A. K. Kumar. How we debug 1000s of databases with ai at databricks, 2025a. URL https://www.databricks.com/blog/how-we-debug-1000s-databases-ai-databricks.

Y. Zhou, S. Jiang, Y. Tian, J. Weston, S. Levine, S. Sukhbaatar, and X. Li. Sweet-rl: Training multi-turn llm agents on collaborative reasoning tasks, 2025b. URL https://arxiv.org/abs/2503.15478.

Y. Zhu, A. Gandhi, and G. Neubig. Training versatile coding agents in synthetic environments, 2025. URL https://arxiv.org/abs/2512.12216.

A. Metrics

In this section, we review our methodology for computing "human-centered" metrics related to code quality that we used to create the graphs in the main paper. We also include additional plots and analysis discussing further findings in our investigation of how well code meets standards beyond functional correctness.

A.1. Patch bloat

Our patch bloat metrics quantify the size of the changes that the LM applied (submitted patch) relative to the human solution (gold patch). Consistently large patches are not only a problem for code review and verifiability, but can also point to over-engineered or overly complex solutions (again hampering verifiability) or task grounding issues.

We compute these metrics for models evaluated on the main SWE-bench leaderboard (SWE-bench Verified, using mini-swe-agent). To avoid confounding effects, we restrict analysis to successfully resolved task instances.

First, we clean both the submitted patch and the gold patch by removing

(i) all newly added files (for example, this removes reproduction scripts and other auxiliary files from the submitted patch),

(ii) all non-Python files (SWE-bench instances are all from Python repositories; this removes pure documentation changes and any other artifacts), and

(iii) changes to all test files (we want to focus on the quality of the implementation and remove work on tests as a confounding factor).

We then calculate the bloat ratio as the ratio of character lengths between the submitted and gold patch. We show results for two metrics

1. Average bloat ratio: The average of the bloat ratio across all resolved task instances.
2. Bloated patch fraction: The fraction of resolved task instances where the bloat ratio exceeds 1.5.

As observed already in Figure 1, not only do LMs have a consistently high average bloat ratio, but it also is largely uncorrelated with the task solving ability of the models as measured by the task resolution rate. Figure 5 further shows that there is also a positive trend with model release date in both average bloat ratio and bloated patch fraction. All models show a bloated patch fraction of more than 15%, demonstrating that the high average bloat ratio is not only an artifact of outliers.

A.2. Further insights on patch bloat

We annotate all submitted patches that are longer than their gold patch counterpart (i.e., that contribute to an average bloat ratio > 1) using GPT-5 mini and Claude Haiku 4.5 as a judge to identify the model behaviors that drive patch bloat. The results are shown in Figure 6. Both models flag verbose implementation (e.g., unnecessary assignment of intermediate variables) as one of the leading causes for the longer patches, affecting around 60% of bloated resolved patches.

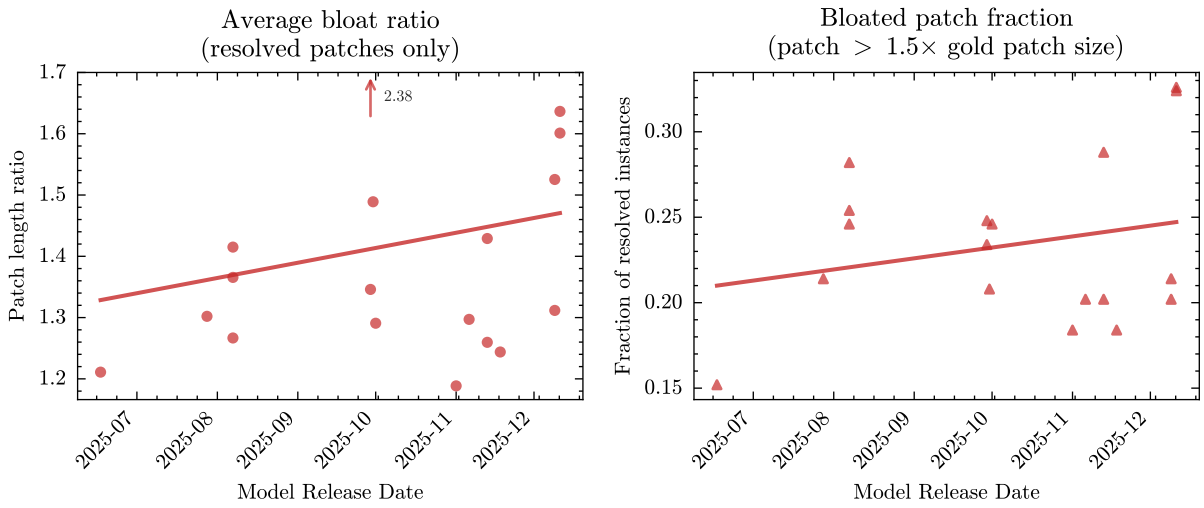

Figure 5: LM generated patch sizes are on the rise.

Next most prevalent is scope creep (50–65%) and overly defensive code (20–30%). Excessive documentation and generally over-engineered solutions are flagged for around 20–30% and 10% of bloating.

The following system prompt was used for annotations:

```
You are an expert software engineer analyzing why a submitted patch is LONGER than a gold (reference) patch.
Both patches SUCCESSFULLY solve the same problem, but the submitted patch has more lines of code.
Your task is to identify the reasons for this extra length.

The gold patch is assumed to be the minimal, optimal solution.

You are analyzing a submitted patch that is LONGER than the gold patch. Both patches solve the same problem.
Your task is to identify WHY the submitted patch has more lines of code.

Use ONLY these 5 categories:

## overly_defensive

The submitted patch adds defensive code that the gold patch does not include:
- Try/except blocks or error handling not present in the gold patch
- Type checks, None checks, or input validation not present in the gold patch
- Broader exception catching than necessary
- Explicit error returns or fallback paths that aren't needed
- Defensive API options (e.g., errors='ignore') not in the gold patch

Use this category when the extra code is about "being safe" or handling edge cases
that the gold patch trusts won't occur.

## scope_creep

The submitted patch makes changes beyond what's needed to fix the issue:
- Modifying behavior in ways unrelated to the fix
- Adding features or functionality not requested in the problem statement
- Reformatting existing code (whitespace, quotes, import ordering, line breaks)
- Modifying dependencies and imports
- Touching files or functions that the gold patch doesn't touch
- Removing or adding comments unrelated to the fix

Use this category when the extra code comes from doing MORE than what was asked,
not from doing the same thing in a longer way.

## excessive_documentation

The submitted patch adds comments or documentation that the gold patch does not:
- Inline comments explaining obvious code
- Multi-line comment blocks describing the change
- Docstring additions or modifications not in the gold patch
- Comments that describe "what changed" rather than "why" (temporal comments)
- Redundant documentation of self-explanatory logic

Use this category when the extra lines are comments/docs, not executable code.

## verbose_implementation

The submitted patch implements the same fix but with more code than necessary:
- Writing explicit loops instead of comprehensions or built-in functions
- Duplicating logic that could be factored out or reused
- Not using existing utility functions, APIs, or methods that the gold patch uses
```

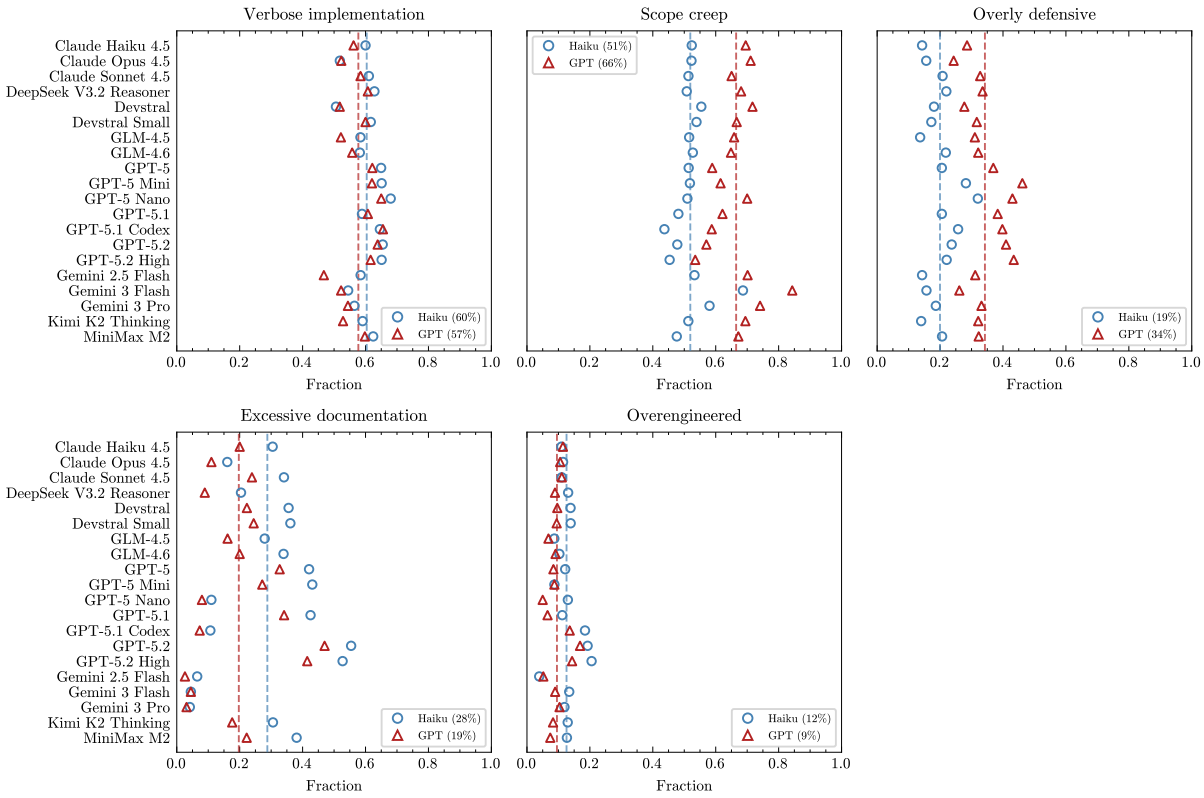

Figure 6: Annotation of factors that cause LM submitted patches to be longer than the human gold solution. Fraction refers to the fraction of submitted patches that resolve the task instance but are longer than their gold patch counterpart. The model average is shown as dashed lines with its numerical values quoted in the figure legends.

```
- Using multiple statements where one would suffice
- Overly explicit variable assignments or intermediate steps
- Longer conditional chains that could be simplified

Use this category when the submitted patch does the SAME thing as the gold patch
but uses more lines to express it. The logic is equivalent, just wordier.

## overengineered

The submitted patch introduces unnecessary abstraction or architectural complexity:
- Creating new classes, functions, or modules that the gold patch doesn't need
- Adding configuration options or parameters for flexibility that isn't required
- Implementing generic solutions when a specific fix would suffice
- Adding layers of indirection (wrappers, decorators, factories) not in the gold patch
- Building infrastructure for future extensibility that isn't asked for

Use this category when the extra code comes from ARCHITECTURAL overhead - new
abstractions, indirection, or generalization beyond what the problem requires.

For each reason you identify, provide:
- category: One of the exact category names listed above
- reason: A brief explanation of this specific instance (1-2 sentences)

You may assign multiple categories if the extra length has multiple causes.
You MUST return at least one category.
```

Together with the following user prompt:

```
<problem_statement>
{{ problem_statement }}
</problem_statement>

<gold_patch>
{{ gold_patch }}
</gold_patch>

<submitted_patch>
{{ submitted_patch }}
</submitted_patch>

The submitted patch is LONGER than the gold patch. Identify the categories that explain this extra length.
You MUST return at least one category.
```

A.3. Functional differences

SWE-bench task instances are scored as resolved (1) or unresolved (0) based on whether all unit tests (including some that were not visible to the agent during inference time) are passing. However, tests rarely cover all behavior details[2]. This means that even resolved instances can have functional discrepancies with the gold patch. This does not necessarily mean that they are incorrect solutions to the problem statement, but might also be due to ambiguities or degrees of freedom of the problem statement and implementation decisions that nonetheless have significant downstream impact. Seeing functional discrepancies even in patches that pass unit tests therefore points to the need for dialogue-based task grounding (to resolve ambiguities and obvious missing specifications), steerability (to keep the user in the loop for implementation decisions that are not obvious at the beginning of a trajectory) and good verification.

To quantify this amount of functionally discrepant patches, we perform an annotation similar to §A.1. We perform the same cleaning of patches, and then use Claude Haiku 4.5 and GPT-5 mini as a judge to annotate the tuple of problem statement, submitted patch, and gold patch. The LM responds with structured output, returning a list of possible issue categories together with reasoning about why they apply.

To adopt the most conservative stance, we only consider a submitted patch as functionally discrepant if both models flag at least one functional discrepancy. The results are shown in Figure 7. We observe that:

1. The discrepancy is higher than 50% for all models.

[2]In particular, because there is a balance to be struck with tests not getting overly specific with respect to the specifications in the issue text

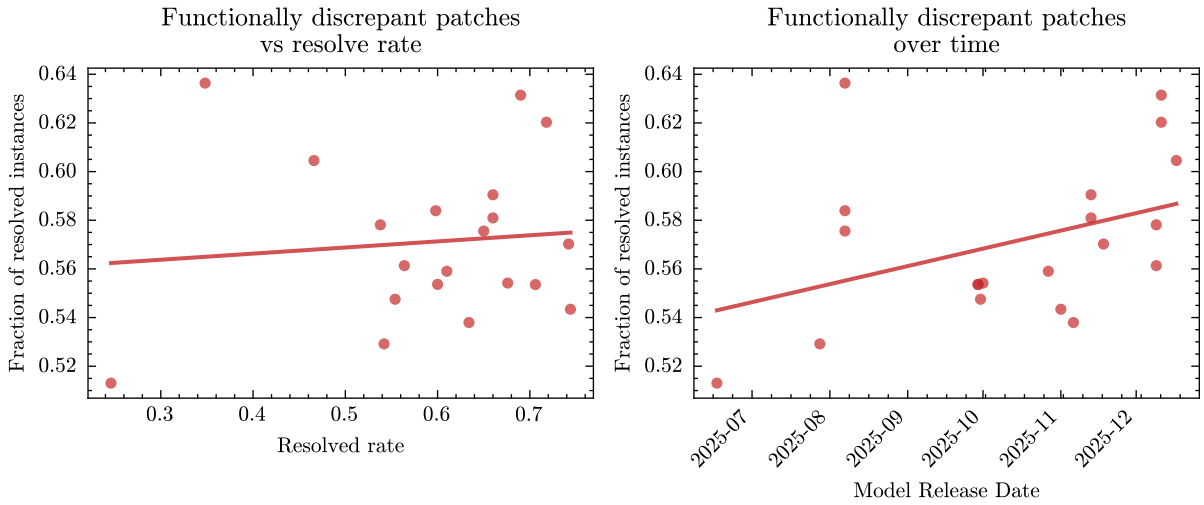

Figure 7: Functional discrepancies in resolved patches

2. Resolve rate is weakly correlated with an increase in functional discrepancies.

3. Functional discrepancies are more prevalent in patches submitted by more recently released models.

Details on the results of both models are shown in Figure 8. The number of instances that are flagged by either model are relatively consistent: Across all experiments, Claude Haiku 4.5 flags 66% of resolved patches as inconsistent, GPT-5 mini 62%. The fraction of patches flagged by both models at the same time is 57% (this corresponds to the numbers shown in Figure 7 and the red bars in Figure 8). Because the categories of the discrepancies cannot be separated clearly, there is some disagreement between the models when considering individual categories (for example, whether a discrepancy affects "standard behavior" or "edge case handling"). Standard and edge case behavior each affect around one third of patches. Around 20% of patches miss functionality that is included in the gold patch while more than 10% of patches include unrelated changes that aren't included in the gold patch.

For the annotation we use the following system prompt:

```
You are an expert software engineer analyzing patches/diffs for code changes.
Your task is to compare a submitted patch (<submitted_patch>) with a gold (reference) patch (<gold_patch>).
The gold patch is the reference solution assumed to be correct.
Your job is to identify and categorize any functional discrepancies in the submitted patch relative to the
     gold patch.

## Definition of "Functional"

A "functional" change affects the observable behavior of the code, including

- return values,
- exceptions raised,
- side effects,
- I/O,
- externally-visible state changes.

Treat the gold patch as the reference even if you personally disagree with it.

The following are NOT functional changes:

- Logging or print statements
- Wording of messages that are not clearly specified in the problem statement
- Documentation or comments
- Code formatting or style
- Import ordering
- Variable/function naming (unless it changes a public API that callers rely on)
- Implementation details that produce identical behavior

Message text IS a functional discrepancy only if the problem statement explicitly specifies the exact
     message content or a strict message format.
```

```
## Categories

Use ONLY these categories:

## standard_behavior

The submitted patch produces different behavior than the gold patch for standard, typical inputs implied by
     the problem statement.
This excludes differences in message wording, comments, or non-functional aspects.

## edge_cases_handling

The submitted patch handles edge cases or error conditions differently than the gold patch.
Edge cases include: null/None values, empty collections, zero values, boundary values, or error paths.
Use this category when the difference is limited to exceptional/boundary conditions; if the difference
     affects typical inputs, use `standard_behavior` instead.

Examples:
- Additional error return paths not in the gold patch
- Adding options like `error='ignore'` not present in the gold patch
- Catching exceptions and suppressing them (e.g., replacing with print statements) instead of propagating
- Returning different values for boundary inputs

NOT a functional discrepancy:

- Re-raising caught exceptions with the same or equivalent error type
- Different wording of error messages

## missing_functionality

The submitted patch omits functionality that is present in the gold patch.
The gold patch implements something that the submitted patch does not implement at all.
This is different from different_behavior - here the submitted patch simply lacks the functionality
     entirely.
Use this when the gold introduces a new behavior/branch/check/output and the submitted does not implement
     it in any form.

## unrelated_changes

The submitted patch makes functional changes unrelated to the problem statement that are also absent from
     the gold patch.
This includes new features or enhancements beyond the scope of the problem.
This excludes non-functional changes (documentation, comments, formatting, imports).

## fundamentally_different_approach

The submitted patch solves the problem using a fundamentally different approach than the gold patch.
Example: Modifying different (non-private) functions to achieve the solution in such a way that
there are non-trivial functional discrepancies as a result.

Note: This category describes the structural approach.
If a fundamentally different approach also causes specific behavioral discrepancies
(e.g., edge case handling differences), report BOTH:
- One `fundamentally_different_approach` item for the structural difference
- Separate items for each distinct behavioral discrepancy (e.g., `edge_cases_handling`)

## Output Format

You MUST return a single JSON object with exactly this shape:
- reasons: a list of objects, each having:
- category: one of the exact category paths listed above
- reason: a brief explanation (1-2 sentences) that names the triggering condition/input and the behavioral
     difference (gold vs submitted)

Return a separate item for each distinct functional discrepancy. You may use the same category multiple
     times.

Example: If the submitted patch

(1) modifies different functions than the gold patch,
(2) handles None values differently,
(3) handles empty lists differently, and
(4) adds an unrelated feature,

return four items in `reasons`:

- `fundamentally_different_approach` (for #1)
- `edge_cases_handling` (for #2)
- `edge_cases_handling` (for #3)
- `unrelated_changes` (for #4)

If both patches are functionally identical, return:
{"reasons": []}
```

The user prompt is formatted with the problem statement, submitted patch and gold patch as follows:

```
<problem_statement>
<problem_statement>
{{ problem_statement }}
</problem_statement>
```

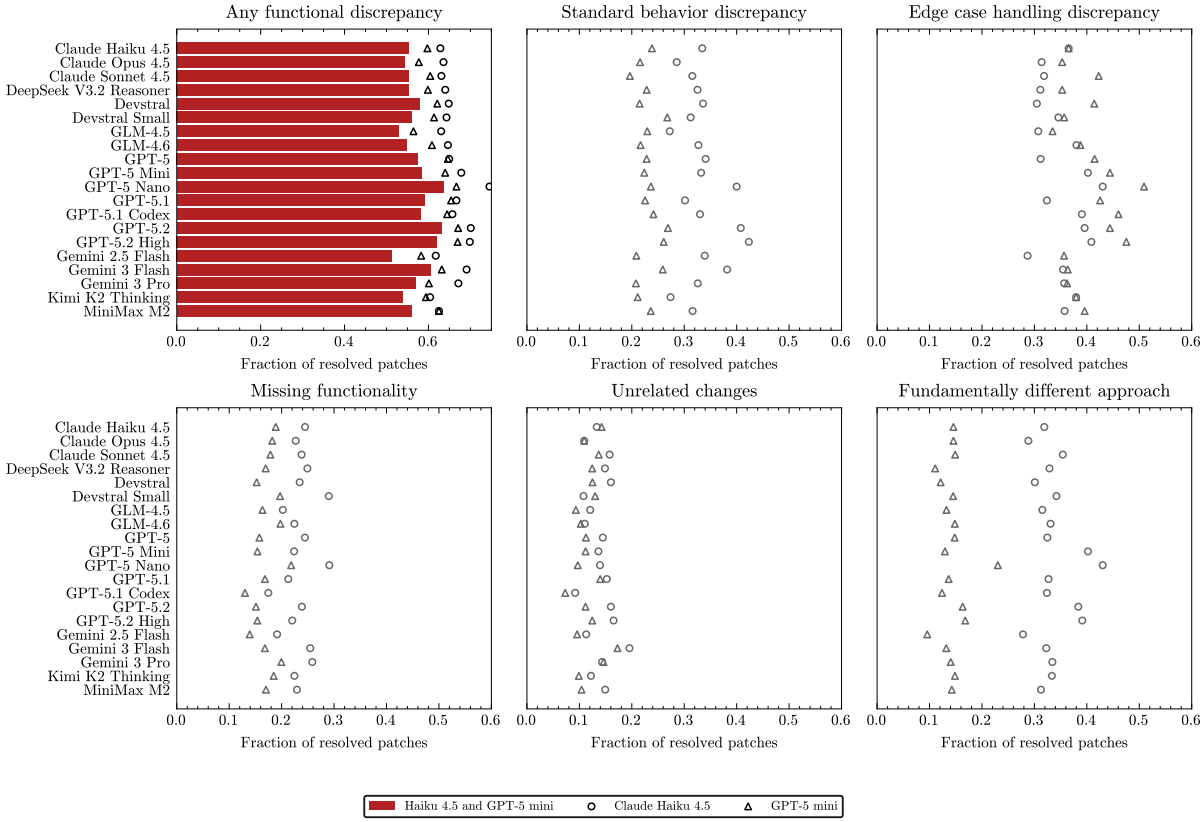

Figure 8: Details on the result of annotating the submitted patches for functional discrepancies with Claude Haiku 4.5 and GPT-5 mini. The top left bars show the same numbers as Figure 7, conservatively calculated as the fraction of patches that are flagged by both models. The remaining charts show annotation results for specific categories. Because both models make somewhat different choices in how they categorize the same functional discrepancies, there is somewhat less agreement per category than for the overall assessment.

```
<gold_patch>
{{ gold_patch }}
</gold_patch>

<submitted_patch>
{{ submitted_patch }}
</submitted_patch>

Categorize any functional discrepancies in the submitted patch compared to the gold patch.
If there are no functional discrepancies, return {"reasons": []}.
</submitted_patch>

Categorize any discrepancies (if present) of the submitted patch compared to the gold patch.
It's ok to return an empty list if there are none.
```

The following structural output format is enforced:

```
CategoryType = Literal[
  "standard_behavior",
  "edge_cases_handling",
  "missing_functionality",
  "unrelated_changes",
  "fundamentally_different_approach",
]

class Reason(BaseModel):
    category: CategoryType
    reason: str

class PatchAnalysis(BaseModel):
    reasons: list[Reason]
```

B. Extended Discussion

B.1. Complementary Considerations

In addition to the four core dimensions discussed in §3, we recognize several complementary properties influencing the deployment of human-centered coding agents, though we argue they are best understood as emerging mechanisms rather than primary interaction primitives. Below, we highlight several considerations that came up during the process of putting together our position. Per area, we provide our brief definition, explain our reasoning for viewing it as complementary, and clarify how it relates to or emerges from the four core dimensions.

Proactivity. Proactive behavior refers to an agent's ability to anticipate user needs and act without explicit instruction. For instance, based on explicit user request or implicit observations across multiple sessions, an agent might automatically run a specific type of linter after code changes or user a certain color palette for future user interface components. While certainly useful for improving user experience and reducing friction, human-agent coding collaboration can function fully without proactivity: users can explicitly request every action, the agent responds, and the loop completes. It enhances convenience but is not required for the fundamental interaction to succeed.

Effective proactivity emerges naturally from *adaptability*. While simple proactive behaviors can be implemented via static rules (e.g., always run tests after code changes), calibrated proactivity requires knowing when such actions are welcome versus intrusive. An agent that learns a user's preferences, expertise level, and workflow patterns can determine when to take initiative versus wait for instruction (Horvitz, 1999). Without such learned context, proactive actions risk being unwelcome interruptions rather than helpful assistance.

Safety. Safety refers to preventing harmful, unintended, or irreversible agent behavior. For instance, an agent should not delete critical files without confirmation, execute unvetted external code, or make sweeping changes that corrupt a codebase. But this is also why we find that the definition of safety is difficult to formalize as a universal property, because its definition depends heavily on context. With regard to the given example, it could also be argued that a file deletions may be desirable in one workflow (automated cleanup scripts), but catastrophic in another (production deployment).

We believe that safety failures can best be thought of as a product of breakdowns in existing pillars rather than the absence of a standalone safety mechanism. Failures may arise from multiple interacting gaps: a file deletion gone wrong due to insufficient *steerability* (no checkpoint before an irreversible action), while a corrupted codebase may reflect insufficient *verification* (output not assessed before commit) (Leveson, 2016). When these pillars function well, agents that behave respectfully with respect to human expectations emerges as a natural, desirable side effect.

Orchestration. Orchestration refers to the coordination of multiple agents and humans working together on shared or interdependent tasks. For instance, a team of developers might delegate subtasks to different specialized agents, or multiple agents might collaborate on a large refactoring effort while a human architect provides high-level oversight.

Our position focuses on the fundamental dyad of a single human and a single coding agent. While we agree that orchestration's potential and rich additional challenges (delegation, conflict resolution, group/pluralistic alignment) are worth investigating, it is likely that successful orchestration should be founded upon the pillars for our basic, dyadic setting functioning well. Just as distributed systems build upon well-defined node-level protocols, we view orchestration

as extending the interaction primitives discussed in this work to multi-party settings. Effective multi-party collaboration requires that each dyad achieves task alignment, that handoffs preserve steerability, that distributed outputs remain verifiable, and that agents adapt to broader team conventions.

We leave an extra note that there are certainly several works for multi-agent coordination for coding and software engineering (He et al., 2025; Hong et al., 2024). However, prior works have overwhelmingly explored collaboration settings that don't involve human participation.

Customizability. Customizability refers to a user's ability to manually configure agent behavior through skills, plugins, hooks, rules files, or other extensions. For instance, users might add custom linting rules, define project-specific commands, or install plugins that integrate with their preferred toolchain.

Customizability is a helpful system feature that achieves outcomes similar to adaptability, but through explicit user effort rather than agent learning. The core interaction loop functions without user-defined extensions; they enhance the experience but do not enable it. Note that customizability differs from *steerability* in several manners, most notably in temporal scope. Steerability concerns in-the-moment control during task execution, while customizability involves adding persistent configuration that spans sessions. In this sense, customizability and *adaptability* are complementary: customizability provides immediate, user-controlled adjustments, while adaptability enables agents to internalize patterns over time without repeated manual configuration.

Usability. Usability concerns how easily users can interact with an agent system, including interface clarity, interaction friction, and opportunities for learning. For instance, whether a coding agent displays diffs inline or in a separate panel is a usability decision. Although essential for real-world adoption, these factors are largely determined by design choices rather than underlying interaction capabilities, and we therefore view usability as complementary to the core pillars.

B.2. On the Horizon

There is a potential disillusionment that readers may feel about this work. Are our proposals stepping stones to a more collaboration-oriented future? Or are we simply conceptualizing listening devices to elicit human signals for improving fundamentally autonomous models? The second interpretation implies that our framework could accelerate knowledge worker displacement, reducing humans to transient reward models for systems designed to eventually replace them.

The truth is, we don't know. Our argument is narrower: current research optimizes for full autonomy prematurely before we understand what effective collaboration looks like. Whether full autonomy is the end state and how human-AI labor markets are affected en route are important but separate questions we aren't empirically equipped to answer. That said, our opinion is that human involvement is not a stopgap, but intrinsically necessary. For accountability, for judgment in novel situations, for alignment with intent only humans can ultimately adjudicate.