A Disciplined Framework for Human-in-the-Loop LLM Coding Agents

Ștefan-Daniel Wagner, Eduard-Cristian Popovici, Laurențiu Boicescu and Traian-Cristian Ureche

***Abstract*—Large language model coding agents can now write working code for much of everyday programming work, and "vibe coding," where the developer gives in to the model and stops tracking the code, has grown up around that capability. It works for prototypes and fails where production fails: edge cases, silent regressions, unexplained changes. We argue that capability is no longer the bottleneck for such work. Discipline is. Building on primary sources from the tool makers and on Karpathy's framing of software as a shifting paradigm, we recast disciplined AI-assisted engineering as resource management on a machine whose scarcest resource is its own context window. Four mechanisms reinforce each other: engineer the context, separate the roles so the worker is not its own grader, make critical parts deterministic, and verify before you ship. We distinguish this human-in-the-loop practice from autonomous multi-agent research, and ground it in a single practitioner's self-logged corpus of roughly 680 sessions across about 40 projects, read against independent studies. Between two snapshots ten days apart, always-on context fell 53%. We price the practice in tokens and dollars and say when it pays. We expect the value of discipline to rise, not fall, as models improve.**

***Index Terms*—agentic AI, LLM coding agents, human-in-the-loop software engineering, context engineering, verification, software engineering practice**

I. Introduction

Large language models (LLMs) can write working code, and they get more skillful at it every year [1]. On SWE-bench, a benchmark of 2,294 tasks drawn from real GitHub issues, a model is handed a codebase and an issue and asked to produce the fix end to end, with no human in the loop [2]. In a controlled experiment, freelance developers given an AI pair programmer finished a scoped implementation task 55.8% faster than the control group [3]. The question that drove this field for years, can the model do it, is now largely settled for most everyday programming work.

A culture grew up around that capability. Andrej Karpathy named it in early 2025: "vibe coding," where you "give in to the vibes, embrace exponentials, and forget that the code even exists" [4]. As a way to build a throwaway prototype on a Friday night, vibe coding is wonderful. As a way to build software that other people depend on, it fails in the places production always fails: the edge case nobody prompted for, the silent regression, the change no one can explain three weeks later. The erosion predates the name: an industry analysis of 211 million changed lines over 2020 to 2024, the years in which AI assistance spread, finds copy-pasted code rising from 8.3% to 12.3% of changes while refactoring fell from about 25% to under 10% [5].

Practitioners already know this. In a 2025 study of how professional developers actually use AI agents, the experienced ones do not vibe. They stay in control, reviewing, constraining, and verifying what the agent produces [1]. A 2025 randomized controlled trial sharpens the point: 16 experienced open-source developers working on their own mature repositories were 19% slower when allowed early-2025 AI tools, even though they believed afterward that the tools had sped them up by 20% [6]. The capability that wins on a scoped, self-contained task does not automatically survive contact with code that someone must maintain. The gap between the demo and the deployment is a gap in discipline.

That is the argument of this paper. *Capability is no longer the bottleneck. Discipline is.* For most such work, the frontier model is good enough. What decides whether you ship something trustworthy is the practice you put around it. This is an engineering claim with concrete mechanisms behind it, and the body of the paper is those mechanisms.

We group the discipline into four habits that reinforce each other. We engineer the context, because the model's context window is a small, degrading resource and most failures trace back to filling it badly [7], [8]. We separate roles, so that the agent doing the work is not the one grading it [8], [9]. We make the parts that must not vary deterministic, by moving them out of the model's judgment and into code [10], [11]. And we refuse to ship what we cannot verify [8].

We formalize these four habits as an operational framework, CRDV (context, roles, determinism, verification), rather than a new system. The framework's principles are vendor-neutral, though every instantiation we show is from one toolchain, Claude Code, because its mechanisms (standing context files, on-demand skills, context-isolated sub-agents, deterministic hooks) are documented in enough detail to reason about precisely. We also draw on our own experience running LLM agents in a production setting, reported in anonymized form from a single-practitioner corpus of roughly 680 sessions across about 40 projects (Section IV-F).

We make four contributions. 1) We ground the framework in a measurable property of the machine rather than in taste: recall degrades as the context window fills, so each CRDV mechanism earns its place by how it manages that scarce resource or guards against its decay (Section II). 2) We locate CRDV against the autonomous multi-agent line of work on one primary axis, the controller, from which the other contrasts follow: their pipelines optimize an unattended model for benchmark pass rate, ours optimizes a supervised agent for production trust (Section III). 3) We specify the four mechanisms precisely enough that a team can adopt them incrementally, each tied to a primary source and an honest

account of its limits (Section IV). 4) We give an honest account of what the framework costs, in tokens and money where our corpus measures it and qualitatively in latency and orchestration, and of when it pays (Sections IV-F and V).

II. From Software 1.0 to 3.0, and the Constraint That Comes With It

To see why discipline is the bottleneck, look at the kind of machine we are now programming.

Karpathy drew the first line in 2017. Software 1.0 is the code we write by hand, in languages like C or Python. Software 2.0 is code "written by the optimization," where the source is no longer instructions but the weights of a neural network learned from data [12]. The program still exists. We just stopped writing it directly.

In 2025 he drew a second line. LLMs, he argued, are a new kind of computer that we program in English: the prompt is the program, and the model is something like a new operating system sitting where the CPU used to [13]. Call it Software 3.0. The shift matters for our argument because it changes where bugs come from. In Software 1.0 a bug is a wrong instruction. In Software 3.0 a bug is just as often a context problem: the model was capable of the right answer but was never given, or could no longer see, the information it needed.

That points at the defining constraint of this machine. A model reasons over a context window, the bounded span of tokens it can attend to at once, and that window is both finite and lossy. Anthropic's own guidance is blunt about it: performance degrades as the context fills, and the model starts "forgetting" earlier instructions or making more mistakes, so the context window is "the most important resource to manage" [8]. The effect is measurable, not folklore, and not only the vendor's own finding: recall falls as the token count grows [7], and models use long contexts unevenly, with accuracy degrading when the relevant information sits in the middle [14].

This gives discipline a precise meaning. Anthropic defines context engineering as "the set of strategies for curating and maintaining the optimal set of tokens" during inference, and frames the goal as finding "the smallest possible set of high-signal tokens that maximize the likelihood of some desired outcome" [7]. Read that as a resource-management problem and the whole practice falls into place. The window is scarce, so we spend it deliberately.

The natural response to a scarce window is to stop loading everything up front. Rather than memorize a whole codebase, an agent can keep an external index and pull in only what each step needs, much as people use file systems and bookmarks instead of holding everything in their heads [7]. Agent Skills make this concrete through multi-level progressive disclosure: a skill exposes only its name and a one-line description until the agent decides it is relevant, and only then loads the body and any linked files [11]. The amount of knowledge that can sit behind such a skill is, in Anthropic's words, "effectively unbounded," because almost none of it occupies the window until it is needed [11].

So the discipline we describe is what careful resource management looks like on a machine whose scarcest resource is the very thing that holds your instructions.

III. Related Work

The dominant research line treats LLM software engineering as a problem of autonomy: how far can a system get on its own?

The benchmarks set that frame. SWE-bench measures whether a system can resolve real GitHub issues end to end, with no human in the loop [2]. Systems built to climb it push on the agent's interface to the computer. SWE-agent shows that a purpose-built agent-computer interface, rather than a raw shell, sharply improves an agent's ability to navigate a repository, edit files, and run tests [15]. AutoCodeRover takes a different route to the same autonomous goal, combining the model with code search over the abstract syntax tree and spectrum-based fault localization to produce a patch [16]. RepairAgent goes further and lets a finite state machine decide when the model should gather information, draft a patch, or validate one, repairing 164 Defects4J bugs, 39 of them beyond what prior techniques had fixed [17]. This work is impressive and complementary to ours, but its unit of success is the unattended fix, scored against a benchmark. Its state machine is determinism in our sense applied inside an unattended loop: the regimes divide on who controls the loop, not on whether code structures it.

A parallel line scales autonomy across multiple agents. MetaGPT encodes standardized operating procedures into the prompts of role-specialized agents so that they can verify intermediate results and reduce errors [18]. ChatDev runs design, coding, and testing as specialized agents that coordinate through language alone [19]. These frameworks are the closest prior art to the role separation we advocate in Section IV, and we borrow their core insight that distinct roles catch each other's mistakes. We differ in the controller. Their pipelines are autonomous by design. Ours keeps a human at the wheel, with the agent roles serving the engineer rather than replacing them.

A survey of the field documents how rapidly language models have spread across software engineering tasks [20], and by contrast it makes the gap we address plain: the literature is rich on autonomous capability and thin on the everyday discipline a practitioner uses to ship trustworthy software with these tools today. Fan et al. reach a matching conclusion from the survey side: their review "reveals the pivotal role that hybrid techniques (traditional SE plus LLMs) have to play in the development and deployment of reliable, efficient and effective LLM-based SE" [21], and CRDV is that hybrid written down as practice. Recent empirical work points the same way: studied in the wild, experienced developers keep control of the agent rather than hand it the keys [1]. Table I summarizes the contrast. Our contribution is the discipline that makes a human-supervised agent dependable, which is the regime most production teams are actually in.

TABLE I
TWO REGIMES: THE AUTONOMOUS LINE MAXIMIZES WHAT AN AGENT DOES UNATTENDED, CRDV WHAT A SUPERVISED AGENT DOES DEPENDABLY.

Axis	Autonomous agents [15]–[19]	CRDV (this paper)
Controller	model-driven	human-in-the-loop
Optimizes for	benchmark pass rate	production trust
Verification	role-separated, inside the model loop	role-separated, plus deterministic gates
Unit of reuse	agent roles, standard procedures	context files, skills, hooks, sub-agents
Evidence	benchmark scores	single-practitioner corpus

IV. THE DISCIPLINE

The CRDV framework is four mechanisms that hold each other up: engineer the context, separate the roles, make the critical parts deterministic, and verify before you ship. Figure 1 shows how they fit a single loop of work. A fifth habit, preferring the simplest thing that works (Section IV-D), is not a mechanism of the framework but the guardrail that keeps the four from becoming machinery for its own sake. None is exotic. Their power is in being applied every time, which is exactly what an undisciplined "vibe" workflow fails to do.

A. Engineer the Context

Everything starts with what the model can see. Because the context window is scarce and lossy (Section II), the first job is to put high-signal tokens in and keep noise out [7].

The sharpest lever here is the line between what is always loaded and what is fetched on demand. A standing instructions file, such as `CLAUDE.md`, is loaded every session and occupies the window from then on, so it should hold only what applies broadly. Knowledge that is relevant only sometimes belongs in a skill that the agent loads on demand, "without bloating every conversation" [8]. Skills make that cheap through multi-level progressive disclosure: until a skill is triggered, only its name and short description sit in the window [11]. The discipline is to treat every line of always-on context as a recurring tax and to push everything else behind on-demand loading.

Tool design is part of the same budget. Results land back in the context window, so a tool that dumps low-level identifiers or sprawling output spends the window on the agent's behalf. Well-designed tools "return only high signal information" and name their parameters unambiguously [22]. A tool is a context decision as much as an action.

In our own practice, a long conversation would start reintroducing a bug we had fixed an hour earlier, the classic sign of a full window and of earlier instructions decaying. The fix was to start fresh with a curated context holding only the relevant files and the one task at hand.

B. Separate the Roles

The second habit follows from the first: if context is scarce, do not make one agent hold all of it.

A sub-agent runs in its own context window, with its own system prompt and tool access, and returns only a result to the caller [9], [23]. That keeps exploration out of the main conversation, so a side quest that reads twenty files does not flood the window the orchestrator has to keep clear [9]. And a sub-agent can explore widely yet hand back a short, distilled summary, often a thousand or two tokens, which cuts how much the orchestrator must read to benefit from the work [7].

Role separation also fixes a subtler problem: the agent that wrote the code is the worst judge of it. The bias is documented beyond our corpus. Early LLM-as-judge evaluation named it self-enhancement bias but could not confirm it [24]. Later work established it directly: LLM evaluators recognize and favor their own generations, scoring their own outputs higher than others' even when human annotators judge them equal [25]. A reviewer running in a fresh sub-agent context sees only the diff and the criteria we give it, not the reasoning that produced the change, so it evaluates the result on its own terms [8]. This is the same logic as a verification step that asks a fresh model to refute a result rather than confirm it, so that the worker is not its own grader [8]. A second, context-isolated reviewer with a blunt instruction to find what is wrong catches a class of errors that the author, primed by its own intent, reads straight past.

C. Make the Critical Parts Deterministic

Some things must happen every time, and "the model usually remembers" is not good enough.

Instructions in a context file are advisory. The model may follow them or, late in a long session, may not. Hooks are different by design: they are deterministic and fire on an event whether or not the model chose to act, which is why they suit actions that must happen with "zero exceptions" [8], [10]. We have watched this class of gate earn its keep: an agent attempted a delegated operation with per-action approvals disabled, and the permission layer denied it, because the human had authorized the delegation but not the removal of the approval gate. Consent to a task is not consent to the task with its safety off, and that distinction held because it was enforced in code, not remembered by a model. We are careful not to oversell this. Determinism here is a design intent, not a proof of zero failure. The same documentation notes edge cases where matching can fail open, so hooks reduce reliance on model judgment rather than removing it entirely [10].

The same instinct applies to logic that should not be rederived by a token predictor each time. Bundling a script with a skill lets the agent run deterministic code instead of regenerating the steps, which makes the workflow consistent and repeatable in a way generation cannot guarantee [11]. The order of operations matters too: in our practice a new safeguard is first exercised by hand inside real sessions, and only then encoded into defaults and an advisory hook, so the tooling automates a discipline that already works rather than promising one that might. Our own enforcement layer is a handful of lifecycle hooks, and every version-controlled one carries a paired unit test: the guardrails are themselves under

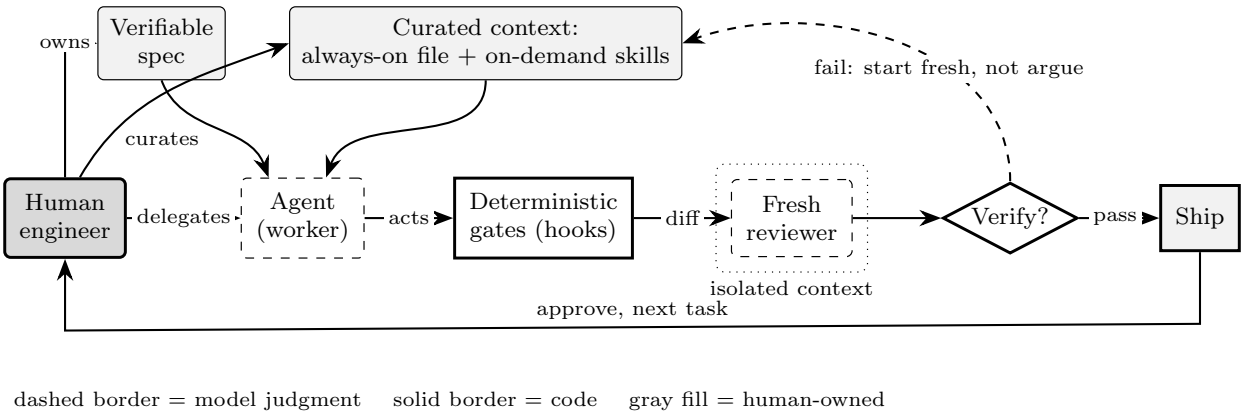

Fig. 1. The CRDV loop. The human owns the spec and curates the context. The agent acts. Deterministic gates and a fresh-context reviewer (dotted: isolated context) check the result before it ships. A failed check restarts with a curated context.

verification. The rule of thumb is simple: if a step must be exact, move it out of the model's judgment and into code.

D. Prefer the Simplest Thing That Works

Discipline is not the same as machinery, and it is easy to confuse the two.

Anthropic's own guidance for building agents recommends "finding the simplest solution possible, and only increasing complexity when needed" [26]. The useful distinction is between a workflow, where the steps are orchestrated through predefined code paths, and an agent, where the model directs its own process [26]. Agentic autonomy buys flexibility but trades away latency, cost, and predictability, so it should be spent only where the task genuinely needs it [26]. In practice we reach for a fixed workflow first and grant the model open-ended autonomy last, because a predictable pipeline is easier to verify than a free-running agent. The discipline includes the discipline not to over-engineer.

E. Verify, or Do Not Ship

The four habits converge on one rule. The difference between a session you have to watch and one you can walk away from is whether you gave the agent a check it can run: a test, a build, a screenshot to compare [8]. Stated plainly, if you cannot verify it, do not ship it [8]. The human side of the loop needs the same structure: in a survey of 319 knowledge workers, higher confidence in generative AI was associated with less critical thinking [27]. The more the tool is trusted, the less its output gets questioned, which is exactly when an independent check earns its keep. Vibe coding forgets that the code exists. Disciplined practice insists that every change carry the evidence of its own correctness. Verification is the thread running through all of it: the spec is verifiable, the reviewer is independent, the critical parts are deterministic, and nothing lands without a check.

F. Lessons and Measurements from Practice

We close the framework with our own use of it, which lines up with what others report about professionals controlling rather than vibing [1]. The corpus is one practitioner's self-logged case record of production use across about 40 projects, reviewed retrospectively. It records our own tool use rather than data gathered from other people, and the agent writes a structured transcript for every session, so collection needed no separate instrumentation. We count a session as one user-initiated transcript carrying at least one model response, and a project as one version-controlled repository or, where a working directory has none, one top-level working area. Always-on context is the token count of the standing instructions loaded at every session start, and a session counts as reset when the practitioner cleared the context rather than arguing with it. That yields roughly 680 sessions over the two months of transcripts the agent retains. Those sessions spawned about 2,100 context-isolated sub-agent runs, some three per session, which is role separation (Section IV-B) showing up as a habit rather than an aspiration. The practice began months earlier, but the agent does not keep those transcripts, so nothing we report reaches back past the retained window. One script recomputes every count and token total below directly from the transcripts, and we will release it with the camera-ready. Three failure patterns recurred whenever a result needed rework. In a context failure, the model had the capability but not the right window. In a review miss, the author agent graded its own work. In a verification gap, nothing checked the result before it shipped. No fourth pattern maps to determinism, and by construction it cannot: determinism is where the fixes landed, and a recurring failure stops recurring when its safeguard moves into code. The patterns are the practitioner's own reading, with no per-pattern tally. What the corpus quantifies is the context-hygiene metrics below, from a single setting, so we read them as direction, not as population estimates.

The context failure was, in our reading, the most common of the three: the model could have done it, but we had fed it the wrong window. Treating the window as the budget of Section IV-A moved every context-hygiene measure we track between two snapshots 10 days apart. Always-on context

shrank 53%, from about 11,100 tokens to about 5,200. The share of sessions we reset with a fresh context rather than argue with rose from about 11% to about 44%, and the rate at which a session re-read material it had already seen fell from about 15% to about 9%. New procedural knowledge moved to on-demand skills instead of always-on memory: before the change, memory writes outnumbered skill writes four to one, and afterward skill writes were the majority. These are within-subject trends that moved alongside the discipline, not a controlled comparison.

The highest-value mechanism, by the same reading (a self-assessment we return to in Section V), was the independent reviewer, because it caught the mistakes the authoring agent was primed not to see. The permission-layer incident of Section IV-C is the same separation applied to task scope rather than to code.

The third pattern sets the boundary: the discipline paid for itself only on work that is verifiable and that someone will depend on. For a throwaway script, vibing is genuinely fine, and pretending otherwise wastes effort. In our corpus that boundary hardened into the routing policy of Table II: the more a task's failure costs, the more capable the model tier, up to the frontier, and the tighter the human gate. The honest boundary of our claim is there: discipline earns its overhead when the cost of being wrong is real.

V. Discussion: What It Costs and When It Pays

A discipline that only ever helped would not need defending, so it is worth being clear about the costs.

The habits buy reliability with overhead. Role separation means running more than one context and paying for the orchestration around it. And although a sub-agent returns a small summary, the orchestrator still receives and stores it, so the saving is reduced consumption, not a free budget [7]. Loading context on demand has its own price: runtime exploration is slower than reading pre-computed data, which is why the sensible strategy is usually a hybrid rather than pure just-in-time retrieval [7]. Autonomy costs too, in latency and money, which is the explicit trade-off behind preferring the simplest workflow that works [26]. Verification is the cheapest of the four to add and the easiest to skip under deadline, which is why we treat it as non-negotiable. Delivery telemetry agrees: the 2024 DORA report estimates a 1.5% drop in delivery throughput and a 7.2% drop in delivery stability for every 25% increase in AI adoption, a pattern its authors attribute, tentatively, to changelists growing faster than the discipline of small batches and robust testing [28].

The token bill is the cost practitioners see least and pay most. Across a trailing 45-day window of the two months of retained transcripts, the practice consumed about 27 billion tokens, about 0.6 billion a day, nearly all of them cached context replayed on every turn rather than fresh prompt text. Priced at the vendor's published per-token rates for the models actually used, that window would cost on the order of $32,000. It did not, because a flat-rate subscription absorbed it, and because prompt caching bills a replayed token at a tenth of a new one. We report the figure because the discipline's economics presently rest on a price that does not track the cost of serving it, and nothing in the engineering guarantees that gap persists. A practice that would be unaffordable at metered rates is a practice with a dependency worth naming.

Limitations and threats to validity. The discipline also has limits we do not want to paper over. Determinism via hooks is a design intent rather than a guarantee, for the reason given in Section IV-C [10]. Our evidence is of two kinds with different strengths: primary documentation of the mechanisms, which is solid, and the single-practitioner corpus of Section IV-F, which is not a controlled comparison and supports a directional read at best. Internal validity is the sharpest worry, because the same practitioner applied the discipline and read its outcomes, which invites a confirmation bias that only a blinded, multi-practitioner replication would rule out. The risk is not hypothetical: METR's participants were measurably slower with AI yet still reported a 20% speedup afterward [6], so we treat our own sense of improvement as a claim to check, not as evidence. Construct validity is strained because the measured window is short, 10 days between snapshots, and it overlaps the period in which we were writing about the discipline, so the trends, the reset rate above all, may partly reflect attention rather than mechanism. External validity is bounded by the toolchain: the principles behind CRDV are not vendor-specific, but we have demonstrated them on one only, and the exact features we cite live in documentation that changes. The corpus itself is not released, because review anonymity and the confidentiality of the underlying projects both preclude it, though the script that recomputes every number we report from it will be. Independent empirical work showing that professionals control rather than vibe [1] supports the direction of our argument. The open test is a controlled study with several practitioners, a non-disciplined control condition, and outcome classification blind to condition.

When does the discipline pay? The boundary is the one we drew in Section IV-F: it earns its overhead on work that is verifiable and that someone will depend on. For a prototype that will be read once and deleted, the overhead is waste, and vibe coding is the rational choice. For software that ships, the asymmetry flips, because the cost of a silent regression dwarfs the cost of an independent review. Table II is that asymmetry written down as policy: each class of task gets the cheapest model tier and the loosest human gate that its failure cost allows.

Better models raise the ceiling on capability but do not repeal the context window, the value of an independent grader, or the need for a check, so we expect the relative value of discipline to rise as raw capability becomes a commodity.

VI. Conclusion

For most everyday programming work, the model is no longer the limiting factor. What separates a convincing demo from software people can depend on is the practice around the model, and that practice is learnable. We have set it out as an operational framework of four reinforcing mechanisms,

TABLE II
ROUTING POLICY, HARDENED IN PRACTICE: ESCALATE THE MODEL TIER, UP TO A FRONTIER CEILING, AND KEEP TIGHTENING THE HUMAN GATE AS THE COST OF FAILURE GROWS.

Task class	Model tier	Human gate
read-only exploration	small	none (auto)
mechanical, skill-shaped	mid	spot-check
routine build on a spec	frontier	one plan approval, shippable checkpoints
architecturally hard	frontier	checkpoint per sub-decision, adversarial self-review
full-corpus audits	frontier	pre-split by the human, never one autonomous pass

CRDV: engineer the context, separate the roles, make the critical parts deterministic, and verify before you ship. Each rests on a measurable reality of the machine we now program, above all the small and lossy context window, and each carries an honest cost that decides when it is worth paying. In the one setting where we measured it, always-on context fell 53%. The promise of vibe coding is that you can forget the code exists. The lesson of production is that someone always has to remember. As models keep improving, the capability gap will close, and the discipline gap is the one that will still be worth your attention.

REFERENCES

[1] R. Huang, A. Reyna, S. Lerner, H. Xia, and B. Hempel, "Professional software developers don't vibe, they control: AI agent use for coding in 2025," *arXiv preprint arXiv:2512.14012*, 2025.

[2] 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?" in *Proc. ICLR*, 2024, arXiv:2310.06770.

[3] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[4] M. S. Smith. (2025, Apr.) Engineers are using AI to code based on vibes. IEEE Spectrum. Accessed 2026-07-10. [Online]. Available: https://spectrum.ieee.org/vibe-coding

[5] GitClear, "AI Copilot Code Quality: 2025 look back at 12 months of data," Industry research report, 2025, accessed 2026-07-10. [Online]. Available: https://www.gitclear.com/ai_assistant_code_quality_2025_research

[6] J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the impact of early-2025 AI on experienced open-source developer productivity," *arXiv preprint arXiv:2507.09089*, 2025.

[7] Anthropic. (2025, Sep.) Effective context engineering for AI agents. Accessed 2026-06-07. [Online]. Available: https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

[8] ——. (2026) Best practices for Claude Code. Official documentation, living document, accessed 2026-06-07. [Online]. Available: https://code.claude.com/docs/en/best-practices

[9] ——. (2026) Create custom subagents. Claude Code documentation, living document, accessed 2026-06-07. [Online]. Available: https://code.claude.com/docs/en/sub-agents

[10] ——. (2026) Automate actions with hooks. Claude Code documentation, living document, accessed 2026-06-07. [Online]. Available: https://code.claude.com/docs/en/hooks-guide

[11] B. Zhang, K. Lazuka, and M. Murag. (2025, Oct.) Equipping agents for the real world with Agent Skills. Anthropic. Accessed 2026-06-07. [Online]. Available: https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

[12] A. Karpathy. (2017, Nov.) Software 2.0. Accessed 2026-06-07. [Online]. Available: https://karpathy.medium.com/software-2-0-a64152b37c35

[13] S. Wang. (2025, Jun.) Andrej Karpathy on Software 3.0: Software in the age of AI. Latent Space. Accessed 2026-07-10. [Online]. Available: https://www.latent.space/p/s3

[14] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024.

[15] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Proc. NeurIPS*, 2024, arXiv:2405.15793.

[16] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proc. ACM ISSTA*, 2024, pp. 1592–1604.

[17] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," in *Proc. IEEE/ACM ICSE*, 2025, pp. 2188–2200.

[18] S. Hong, M. Zhuge, J. Chen *et al.*, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *Proc. ICLR*, 2024, arXiv:2308.00352.

[19] C. Qian, W. Liu, H. Liu, N. Chen *et al.*, "ChatDev: Communicative agents for software development," in *Proc. ACL*, 2024, pp. 15 174–15 186.

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, 2024, article 220.

[21] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *Proc. IEEE/ACM Int. Conf. Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53.

[22] K. Aizawa *et al.* (2025, Sep.) Writing effective tools for agents - with agents. Anthropic. Accessed 2026-06-07. [Online]. Available: https://www.anthropic.com/engineering/writing-tools-for-agents

[23] Anthropic. (2025) Building agents with the Claude Agent SDK. Accessed 2026-06-07. [Online]. Available: https://claude.com/blog/building-agents-with-the-claude-agent-sdk

[24] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang *et al.*, "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in *Proc. NeurIPS Datasets and Benchmarks Track*, 2023, arXiv:2306.05685.

[25] A. Panickssery, S. R. Bowman, and S. Feng, "LLM evaluators recognize and favor their own generations," in *Proc. NeurIPS*, 2024, arXiv:2404.13076.

[26] Anthropic. (2024, Dec.) Building effective agents. Accessed 2026-06-07. [Online]. Available: https://www.anthropic.com/engineering/building-effective-agents

[27] H.-P. Lee, A. Sarkar, L. Tankelevitch, I. Drosos, S. Rintel, R. Banks, and N. Wilson, "The impact of generative AI on critical thinking: Self-reported reductions in cognitive effort and confidence effects from a survey of knowledge workers," in *Proc. ACM CHI*, 2025.

[28] DORA, "Accelerate State of DevOps Report 2024," Google Cloud DORA research report, 2024, accessed 2026-07-10. [Online]. Available: https://dora.dev/research/2024/dora-report/