Agentic Agile-V: From Vibe Coding to Verified Engineering in Software and Hardware Development

Christopher Koch
Independent Researcher

***Abstract*—Agentic AI coding systems can inspect repositories, plan implementation steps, edit files, call tools, run tests, and submit pull requests. These capabilities make software and hardware development faster in some settings, but current evidence does not support the simple claim that autonomous code generation automatically improves engineering outcomes. Controlled studies report productivity gains in some enterprise tasks, slowdowns in mature open-source work, moderate but heterogeneous meta-analytic effects, and persistent failures in repository setup, dependency handling, permission gating, and hardware verification. This paper argues that the central problem is no longer prompt engineering; it is engineering process control. It synthesizes evidence from agentic software engineering, GitHub-scale adoption studies, repository-level agent configuration, productivity trials, issue-resolution benchmarks, and hardware/RTL verification research. It proposes Agentic Agile-V, a process framework that uses Agile-V as the lifecycle backbone and a task-level SCOPE-V loop - Specify, Constrain, Orchestrate, Prove, Evolve, and Verify - to convert conversational intent into structured engineering artifacts and acceptance evidence. The paper contributes: (i) a taxonomy of minimum input artifacts for agentic software, firmware, and hardware work; (ii) a conversation-to-contract gate that separates exploratory dialogue from implementation; (iii) risk-adaptive feature, bug-fix, testing, and hardware workflows; and (iv) an evidence-bundle acceptance model for agent-generated artifacts. The paper concludes that agentic AI does not eliminate engineering discipline; it increases the value of requirements, constraints, traceability, independent verification, and human approval.**

***Index Terms*—Agentic AI, software engineering, Agile-V, AI-assisted development, coding agents, hardware development, firmware, verification, testing, requirements engineering, V-model, software process.**

I. Introduction

Agentic AI changes the form of engineering work. Modern coding agents can do more than complete a line of code: they can read repositories, plan work, invoke terminals, modify multiple files, run tests, prepare pull requests, and respond to review feedback. The ecosystem now includes asynchronous coding agents, repository-level instruction files, agent SDKs, multi-agent workspaces, and model-agnostic routing. Large-scale evidence shows that this is already an operational phenomenon rather than a speculative one: the AIDev dataset reports 932,791 agent-authored pull requests across 116,211 repositories and 72,189 developers [8]. Surveys on LLM-based agents for software engineering identify applications across requirements, implementation, testing, maintenance, and human-agent collaboration [2], [3].

Yet the evidence base also challenges the strongest hype. A randomized controlled trial with 96 full-time Google engineers reported an estimated 21 percent reduction in time on a complex enterprise task with AI assistance [4]. In contrast, a METR randomized trial with experienced open-source developers found that AI tools increased task completion time by 19 percent in mature repositories, despite developer expectations of speedup [5]. A 2026 meta-analysis found a statistically significant but moderate productivity effect, with substantial heterogeneity and smaller effects in open-source and enterprise contexts [6]. Repository-level configuration files can reduce runtime and token use in some settings [12], but other evaluations find that context files can reduce task success and increase cost when they add unnecessary or mismatched requirements [13]. Hardware evidence is even more cautionary: RealBench reports low pass rates on real-world Verilog generation, including 0 percent pass@1 on system-level tasks for evaluated models [27], and FIXME argues that functional verification remains underexplored despite rapid progress in LLM-aided design [28].

These findings point to a process gap. Agentic AI can generate plausible engineering artifacts faster than humans can inspect them. Therefore, the bottleneck shifts from code synthesis to specification quality, execution context, verification, traceability, and controlled iteration. The right question is not merely "How do we prompt better?" but "Which process turns natural-language intent into verifiable engineering output?"

This paper proposes Agentic Agile-V, a process framework for agentic software, firmware, and hardware development. It integrates Agile-V, a compliance-ready approach that combines Agile iteration with V-model verification and audit artifact generation [1], with a task-level SCOPE-V execution loop. The central principle is:

> Conversation is useful for discovering intent; structured artifacts are required for implementation; evidence is required for acceptance.

The contributions are fourfold:

1) A synthesis of evidence from agentic coding research, GitHub adoption studies, repository-configuration work, productivity studies, issue-resolution benchmarks, and hardware verification benchmarks.
2) A conversation-to-contract model that separates exploratory dialogue from structured execution.

3) The Agentic Agile-V process model, combining Agile-V lifecycle structure with the SCOPE-V task loop.
4) Practical workflows and risk-adaptive evidence gates for feature work, bug fixing, test generation, firmware, and hardware/RTL development.

II. Background and Evidence Base

A. From Assistants to Agents

LLM-based software agents extend standalone language models with perception, planning, memory, tool use, execution environments, and human interaction [2]. Code-generation agents differ from earlier code generators because they can decompose tasks, navigate repositories, execute tests, debug failures, and integrate changes across the software development lifecycle [3]. OpenHands, for example, emphasizes sandboxed execution, lifecycle control, model-agnostic routing, custom tools, memory management, and workspace/API integrations [21]. GitHub has also moved toward agentic workflows in which tasks can be delegated to agents that clone repositories, work in virtual environments, document their decisions, run tests, and propose pull requests for human review [22], [23].

This transition changes the engineering problem. The unit of interaction is no longer just a prompt and an answer. It is a socio-technical loop consisting of requirements, repository context, tools, permissions, tests, build environments, review practices, and release gates.

B. Productivity Is Context-Dependent

The current productivity literature supports a balanced claim. AI assistance can help, especially in well-scoped or routine tasks, but it can also create overhead through prompting, waiting, review, and correction. The Google RCT provides evidence for acceleration in an enterprise setting [4]; the METR RCT shows slowdown in familiar mature open-source work [5]; and a meta-analysis reports a moderate average effect with strong context dependence [6]. A systematic literature review also warns that developer productivity is multi-dimensional and cannot be reduced to output volume or task time alone [7]. Maintenance-burden research further suggests that AI-assisted output may shift review and rework load toward experienced developers [32].

The implication is not that agentic AI fails. It is that AI-assisted engineering is a process-sensitive intervention. Task type, codebase complexity, developer expertise, test coverage, dependency setup, and verification cost can determine whether agents help or hurt.

C. GitHub Evidence and Repository Configuration

GitHub evidence shows rapid adoption. AIDev provides a large corpus of agent-authored pull requests [8]. A task-stratified analysis of 7,156 pull requests found that task type strongly influences acceptance rates: documentation changes were accepted more often than new features, and no single agent performed best across all categories [9]. An empirical study of Claude Code pull requests found high acceptance, but also substantial human revision, especially for bug fixes, documentation, and project-specific standards [10].

Configuration artifacts are emerging as the process layer around agents. A study of 2,926 GitHub repositories found eight configuration mechanisms across tools such as Claude Code, GitHub Copilot, Cursor, Gemini, and Codex, with context files dominating and `AGENTS.md` emerging as an interoperable standard [11]. One study found that `AGENTS.md` was associated with lower runtime and lower output-token consumption [12]. Another found that context files can reduce task success and increase inference cost when they impose unnecessary requirements [13]. A factorial study of configuration-file structure found limited evidence that size, position, architecture, or local contradiction variables alone create reliable adherence effects [14].

These mixed results motivate a minimal-context principle: repository instructions should be short, current, non-contradictory, and tied to executable feedback.

D. Repository Execution Remains Hard

Real-world repository tasks remain challenging. GitTaskBench found that even the best evaluated system, OpenHands plus Claude 3.7, solved 48.15 percent of tasks, with many failures caused by environment setup and dependency resolution [15]. RepoMaster improves repository exploration by constructing graphs and pruning context, showing that repository understanding and context selection are central bottlenecks [16]. SWE-Skills-Bench found that procedural skill packages usually provide limited marginal benefit unless they match the domain and project context [17]. SWE-rebench V2 further demonstrates the importance of reproducible execution environments and reliable test suites at scale [18].

The lesson is that code context alone is insufficient. Agents need execution context: build commands, dependency setup, test commands, environment variables, toolchain information, simulator access, and clear acceptance criteria.

E. Hardware and Firmware Raise the Bar

Hardware, firmware, and embedded development have stricter failure modes. Incorrect pin mappings, register values, timing assumptions, bus behavior, reset handling, or memory layout can produce failures that are costly or unsafe. RealBench explicitly addresses the gap between simple Verilog benchmarks and real-world IP-level workflows; its low pass rates show that current LLMs are not reliable system-level hardware generators [27]. FIXME focuses on design verification and uses silicon-proven designs to evaluate functional verification capabilities [28]. Surveys of LLMs for electronic design automation and hardware/software co-design highlight both opportunities and reliability limitations [29]–[31].

For these domains, compilation is not proof. Simulation, formal checking, hardware-in-the-loop tests, timing analysis, and traceability from requirement to verification evidence are essential.

III. Method: Bounded Evidence Synthesis

This paper uses a bounded evidence synthesis rather than a quantitative meta-analysis. The evidence base is heterogeneous: randomized trials, GitHub-scale datasets, repository-configuration studies, tool papers, issue-resolution surveys, hardware benchmarks, and process frameworks measure different outcomes. Pooling them into a single effect size would obscure rather than clarify the process problem.

Sources were selected from four streams:

1) agentic software engineering surveys and tool papers;
2) empirical studies of developer productivity, agent-authored pull requests, and issue resolution;
3) repository-configuration and execution-environment studies;
4) hardware, firmware, and design-verification benchmarks.

A source was included if it addressed at least one of the following: agentic code generation, issue resolution, repository-aware execution, productivity, configuration artifacts, permission or tool gating, hardware generation, hardware verification, lifecycle traceability, or verification evidence. Industry and platform sources were used only when they documented observable tool behavior or adoption trends; peer-reviewed and preprint research was used for empirical and conceptual claims.

IV. Problem Statement

A. The Limits of Conversational Development

Conversational development is useful, but insufficient as an implementation substrate. It helps teams discover requirements, surface ambiguity, compare architectures, and identify risks. But long chat histories are not reliable engineering contracts:

- they contain superseded assumptions;
- constraints are often implicit;
- acceptance criteria are rarely executable;
- agents may overfit to recent turns;
- reviewers cannot easily audit which instruction governed a change.

Therefore, agentic work needs a conversation-to-contract gate: after discovery, the relevant intent must be converted into a reviewed execution brief.

B. Verification Debt

Agentic AI can increase the volume of code, tests, documentation, and patches. If output volume grows faster than verification capacity, teams accumulate verification debt: weak tests, hidden regressions, broad patches, unvalidated dependencies, undocumented behavior, and increased reviewer burden. The METR trial and maintenance-burden findings indicate that review and cleanup can erase perceived speedups [5], [32]. In hardware and embedded work, verification debt can become operational or physical risk.

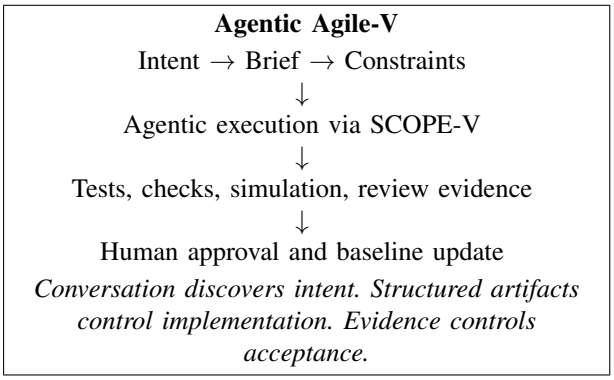

Fig. 1. High-level Agentic Agile-V model.

C. The Missing Bridge

Current tools provide execution surfaces, sandboxing, repository instructions, test execution, and pull-request workflows. Existing engineering processes provide requirements discipline, verification logic, review, and release gates. The missing bridge is a lightweight framework that tells teams what input to provide to agents, how to structure agent execution, when to test, and which evidence is required before accepting generated artifacts.

V. The Agentic Agile-V Framework

A. Overview

Agentic Agile-V has two layers. The macro layer is Agile-V: an iterative lifecycle in which each increment remains traceable to requirements, design, implementation, verification, approval, and audit evidence. The micro layer is SCOPE-V, the task-level loop used to run individual agentic tasks.

This design avoids a false choice between agility and verification. Agile iteration provides speed; V-model reasoning provides traceability; agentic execution provides implementation capacity; verification gates decide acceptance.

B. The SCOPE-V Micro-Cycle

Each agentic task passes through six steps:

Specify. Convert intent into a task brief with objective, scope, non-goals, affected modules, dependencies, acceptance criteria, and required evidence.

Constrain. Define boundaries: no public API change unless approved, no unrelated files, no new dependencies without justification, no broad refactor during a bug fix, explicit review for security-sensitive code, and mandatory preservation of hardware timing and safety constraints.

Orchestrate. Define how the agent should work: inspect first, summarize current design, propose a plan, implement small slices, run local checks, and produce a diff summary with residual risks.

Prove. Require evidence: unit tests, integration tests, regression tests, static analysis, type checks, linting, security scans, simulation, formal checks, hardware-in-the-loop results, or review checklists depending on risk.

Evolve. Feed validated learning back into repository instructions, templates, tests, and engineering baselines. Remove stale or harmful instructions.

Verify. Treat verification as recurring rather than final: before implementation, during patching, before merge, after deployment, and after field feedback.

C. Agile-V as Lifecycle Backbone

Agile-V was proposed to address a weakness of machine-speed AI-assisted engineering: lack of built-in task-level verification and regulatory traceability [1]. Its case study demonstrates feasibility in a hardware-in-the-loop setting with independent test generation and audit artifacts. This paper generalizes the idea to agentic software, firmware, and hardware development by specifying inputs, task workflows, and acceptance gates.

The design principle is:

> Agile-V controls the lifecycle; SCOPE-V controls the agentic task.

VI. Minimum Input Artifact Model

Table I defines a minimum input package. The goal is not to overload agents with context, but to provide enough structured information to avoid guessing.

VII. Conversational Discovery vs. Structured Execution

A. When Conversation Helps

Conversation is appropriate for early uncertainty: clarifying requirements, brainstorming architecture, identifying missing constraints, comparing test strategies, asking what could fail, and exploring alternative designs. In this phase the agent acts as a thinking partner. The desired output is not code first; it is a better problem statement.

B. When Structure Is Mandatory

Before implementation, the conversation must become a structured brief. This is mandatory when a task affects public APIs, safety, security, performance, hardware behavior, regulated workflows, customer-facing behavior, shared libraries, or persistent data.

The operational rule is:

> Do not let an agent implement from a long chat. Let it implement from a reviewed brief.

This rule is consistent with mixed evidence on context files. Relevant instructions can improve efficiency [12]; unnecessary or mismatched instructions can harm success [13]. The correct goal is not maximum context, but decision-relevant context.

VIII. Task Workflows

A. Feature Development

Feature work expands product intent into behavior. A recommended process is:

1) Write a feature brief: goal, non-goals, acceptance criteria.
2) Identify affected modules, APIs, data structures, and tests.
3) Ask the agent to inspect and summarize current design.
4) Require a plan before edits.
5) Implement the smallest useful slice.
6) Add or update tests alongside implementation.
7) Run targeted and regression checks.
8) Produce a diff summary, evidence bundle, and residual-risk note.
9) Require human review for architecture, security, maintainability, and edge cases.

B. Bug Fixing

Bug fixing is causal diagnosis, not feature generation. The agent should not patch immediately. A recommended process is:

1) Capture observed and expected behavior.
2) Provide reproduction steps, logs, environment, and version.
3) Ask the agent for hypotheses and missing evidence.
4) Localize likely files and call paths.
5) Create a failing regression test where possible.
6) Apply the minimal patch.
7) Run regression and nearby tests.
8) Explain why the fix works and what it does not address.

Issue-resolution surveys emphasize that realistic maintenance requires long-horizon reasoning, iterative exploration, and feedback-driven decision-making beyond single-shot generation [19], [20].

C. Testing and Review

Testing must be inside the agent loop, not after it:

- **Before implementation:** identify expected tests, edge cases, and failure modes.
- **During implementation:** run targeted tests after small changes.
- **Before merge:** require CI, static analysis, type checks, security checks, and review.
- **After merge:** monitor logs, defects, performance, and user feedback.

Permission and action gating are part of the testing problem. A stress-test of Claude Code's auto mode found that permission-gate assumptions may fail under ambiguous state-changing scenarios, especially when equivalent effects can be achieved through file edits instead of shell commands [26]. This supports explicit risk classification and human approval for state-changing or high-blast-radius actions.

D. Hardware, Firmware, and Embedded Development

Hardware-facing work requires stricter input and stricter evidence. The agent input package should include board/chip revision, datasheet excerpts, registers, pinout, clocks, protocols, memory map, toolchain, simulator, formal checker, test equipment, and rollback plan. Implementation should not be accepted without simulation, static checks, formal properties where applicable, and hardware-in-the-loop evidence for production-risk changes.

TABLE I
MINIMUM INPUT ARTIFACTS FOR AGENTIC SOFTWARE, FIRMWARE, AND HARDWARE DEVELOPMENT.

Artifact	Software Example	Firmware/Hardware Example
Intent and scope	User story, target behavior, non-goals, affected APIs	Function of module, hardware mode, peripheral role, integration boundary
Acceptance criteria	Expected inputs/outputs, UI behavior, error cases, performance target	Timing budget, protocol compliance, register behavior, waveform or HIL expectation
Architecture context	Modules, interfaces, dependencies, data model, service boundaries	Board revision, MCU/FPGA/SoC, memory map, clocks, buses, pinout, RTOS assumptions
Constraints	No API breakage, no new dependency, style and security rules	Voltage/power limits, timing constraints, interrupt rules, safety states, reset behavior
Execution context	Build commands, dependency setup, test commands, environment variables	Toolchain, simulator, synthesis tool, formal checker, test equipment, firmware flashing path
Evidence requirement	Unit/integration tests, lint, type checks, security scan, reviewer checklist	Simulation logs, formal properties, HIL logs, timing analysis, regression matrix
Risk class	Prototype, production, regulated, security-sensitive	Lab prototype, field firmware, safety-related, manufacturing release

TABLE II
RISK-ADAPTIVE ACCEPTANCE GATES FOR AGENT-GENERATED ARTIFACTS.

Risk Level	Typical Tasks	Required Evidence	Human Gate
R0: exploratory	Throwaway prototype, internal demo, draft script	Smoke test or manual run; no production credentials	Optional review
R1: routine	Documentation, small refactor, non-critical UI polish	Targeted tests, lint/type checks, diff summary	Normal code review
R2: production	Feature change, bug fix, public API, database migration	Regression tests, CI, static/security checks, rollback plan	Mandatory reviewer approval
R3: high assurance	Security, payments, medical, firmware, hardware, safety-related behavior	Traceable requirements, independent tests, simulation/formal/HIL as applicable, audit artifact	Explicit sign-off and release gate

IX. RISK-ADAPTIVE ACCEPTANCE GATES

Not all tasks require the same rigor. Table II defines four acceptance levels.

The acceptance rule is:

> Agent output is not accepted because it is plausible; it is accepted because it satisfies evidence appropriate to its risk level.

A. Evidence Bundle

For R2 and R3 tasks, the paper recommends a minimum evidence bundle:

- task brief and requirement identifiers;
- agent plan and affected files;
- executed commands and test results;
- diff summary and known residual risks;
- trace from acceptance criteria to tests;
- reviewer decision and follow-up actions;
- rollback or recovery path for production changes.

For hardware or firmware, the evidence bundle should include simulation logs, formal check results where applicable, HIL records, toolchain version, board revision, and timing or protocol evidence.

X. DISCUSSION

A. Implications for Teams

Teams adopting agentic development should avoid both overconfidence and rejection. The practical steps are:

1) Maintain minimal repository instructions such as `AGENTS.md`, but keep them short, current, and testable.
2) Use task templates for features, bugs, tests, and hardware work.
3) Require agents to inspect, summarize, and plan before editing.
4) Require tests or equivalent evidence before acceptance.
5) Separate implementation and verification agents where risk is high.
6) Track review load, defect escape, rework, and lead time rather than only code volume.

B. Implications for Tool Builders

Tool builders should optimize not only for code generation but for evidence generation. Useful tool capabilities include structured brief editors, dependency setup capture, test discovery, traceability, risk classification, permission gates, sandboxed execution, review summaries, and exportable evidence bundles. The direction of the ecosystem, including OpenHands, GitHub agent workflows, Antigravity-style artifacts,

and agent standards, is toward orchestration and observability rather than simple autocomplete [21], [23]–[25].

C. Implications for Hardware and Embedded Engineering

For embedded and hardware teams, the process should be even stricter. Agents can assist with drivers, register definitions, test harnesses, assertions, and documentation, but they must not bypass timing, protocol, safety, or HIL gates. Hardware benchmarks suggest that realistic system-level generation and verification remain difficult [27], [28], [31].

D. Threats to Validity

This paper is a synthesis and process proposal, not a new benchmark. The field is moving quickly, and point estimates from 2024 to 2026 may change as models and tools evolve. Productivity studies vary by task type, developer experience, codebase maturity, tool generation, and organizational culture. Hardware benchmarks differ in design complexity and verification rigor. The framework should therefore be validated empirically in future studies across multiple teams, repositories, tools, and hardware domains.

XI. Practical Templates

A. Feature Brief Template

A feature brief should include: objective, user-visible behavior, non-goals, affected modules, interface contracts, migration needs, compatibility constraints, security considerations, acceptance criteria, tests to add or update, and rollback path.

B. Bug Brief Template

A bug brief should include: observed behavior, expected behavior, reproduction steps, input data, logs/traces, environment, affected version, suspected area, failing test if available, constraints on fix scope, and regression-test requirement.

C. Hardware/Firmware Brief Template

A hardware or firmware brief should include: board revision, chip or FPGA variant, datasheet excerpts, register map, pinout, clock tree, bus/protocol rules, memory map, RTOS/bare-metal assumptions, timing and power constraints, safety states, toolchain, simulator, HIL setup, and acceptance evidence.

XII. Research Agenda

Future work should evaluate Agentic Agile-V empirically. Key questions include:

1) Do structured execution briefs improve agent success compared with conversational prompts?
2) What is the minimum useful content of repository instructions?
3) Does independent agentic test generation reduce defect escape?
4) Which task classes benefit from Agile-V/SCOPE-V and which are slowed by overhead?
5) Can evidence bundles reduce verification debt without eliminating productivity gains?
6) How should hardware and firmware evidence quality be measured?

XIII. Conclusion

Agentic AI is changing software and hardware development, but it does not make engineering process obsolete. The evidence rejects both extremes: agentic coding is not merely a toy, but neither is it a universal productivity multiplier. Its value depends on context, constraints, execution environments, verification gates, and human oversight.

This paper proposed Agentic Agile-V, combining Agile-V lifecycle discipline with the SCOPE-V task loop. The central message is that conversation is good for discovering intent, but structured artifacts are required for implementation. Code, tests, documentation, firmware, and hardware designs should be accepted only when they produce evidence appropriate to their risk level. The future of agentic engineering is not vibe coding at scale. It is verified engineering with agents inside the loop.

References

[1] C. Koch and J. A. Wellbrock, "Agile V: A Compliance-Ready Framework for AI-Augmented Engineering – From Concept to Audit-Ready Delivery," arXiv:2602.20684, 2026. [Online]. Available: https://arxiv.org/abs/2602.20684

[2] J. Liu et al., "Large Language Model-Based Agents for Software Engineering: A Survey," arXiv:2409.02977, 2024. [Online]. Available: https://arxiv.org/abs/2409.02977

[3] Y. Dong et al., "A Survey on Code Generation with LLM-based Agents," arXiv:2508.00083, 2025. [Online]. Available: https://arxiv.org/abs/2508.00083

[4] E. Paradis et al., "How much does AI impact development speed? An enterprise-based randomized controlled trial," arXiv:2410.12944, 2024. [Online]. Available: https://arxiv.org/abs/2410.12944

[5] J. Becker, N. Rush, E. Barnes, and D. Rein, "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," arXiv:2507.09089, 2025. [Online]. Available: https://arxiv.org/abs/2507.09089

[6] S. Maier, M. Gunzenhaeuser, J. Schweisthal, M. Schneider, and S. Feuerriegel, "A meta-analysis of the effect of generative AI on productivity and learning in programming," arXiv:2605.04779, 2026. [Online]. Available: https://arxiv.org/abs/2605.04779

[7] A. Mohamed, M. Assi, and M. Guizani, "The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review," arXiv:2507.03156, 2025. [Online]. Available: https://arxiv.org/abs/2507.03156

[8] H. Li, H. Zhang, and A. E. Hassan, "AIDev: Studying AI Coding Agents on GitHub," arXiv:2602.09185, 2026. [Online]. Available: https://arxiv.org/abs/2602.09185

[9] G. Pinna, J. Gong, D. Williams, and F. Sarro, "Comparing AI Coding Agents: A Task-Stratified Analysis of Pull Request Acceptance," arXiv:2602.08915, 2026. [Online]. Available: https://arxiv.org/abs/2602.08915

[10] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan, "On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub," arXiv:2509.14745, 2025. [Online]. Available: https://arxiv.org/abs/2509.14745

[11] M. Galster, S. Mohsenimofidi, J. L. Lulla, M. A. Abubakar, C. Treude, and S. Baltes, "Configuring Agentic AI Coding Tools: An Exploratory Study," arXiv:2602.14690, 2026. [Online]. Available: https://arxiv.org/abs/2602.14690

[12] J. L. Lulla, S. Mohsenimofidi, M. Galster, J. M. Zhang, S. Baltes, and C. Treude, "On the Impact of AGENTS.md Files on the Efficiency of AI Coding Agents," arXiv:2601.20404, 2026. [Online]. Available: https://arxiv.org/abs/2601.20404

[13] T. Gloaguen, N. Muendler, M. Mueller, V. Raychev, and M. Vechev, "Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?" arXiv:2602.11988, 2026. [Online]. Available: https://arxiv.org/abs/2602.11988

[14] D. McMillan, "Instruction Adherence in Coding Agent Configuration Files: A Factorial Study of Four File-Structure Variables," arXiv:2605.10039, 2026. [Online]. Available: https://arxiv.org/abs/2605.10039

[15] Z. Ni et al., "GitTaskBench: A Benchmark for Code Agents Solving Real-World Tasks Through Code Repository Leveraging," arXiv:2508.18993, 2025. [Online]. Available: https://arxiv.org/abs/2508.18993

[16] H. Wang et al., "RepoMaster: Autonomous Exploration and Understanding of GitHub Repositories for Complex Task Solving," arXiv:2505.21577, 2025. [Online]. Available: https://arxiv.org/abs/2505.21577

[17] T. Han et al., "SWE-Skills-Bench: Do Agent Skills Actually Help in Real-World Software Engineering?" arXiv:2603.15401, 2026. [Online]. Available: https://arxiv.org/abs/2603.15401

[18] I. Badertdinov, M. Nekrashevich, A. Shevtsov, and A. Golubev, "SWE-rebench V2: Language-Agnostic SWE Task Collection at Scale," arXiv:2602.23866, 2026. [Online]. Available: https://arxiv.org/abs/2602.23866

[19] Z. Jiang, D. Lo, and Z. Liu, "Agentic Software Issue Resolution with Large Language Models: A Survey," arXiv:2512.22256, 2025. [Online]. Available: https://arxiv.org/abs/2512.22256

[20] C. Li et al., "Advances and Frontiers of LLM-based Issue Resolution in Software Engineering: A Comprehensive Survey," arXiv:2601.11655, 2026. [Online]. Available: https://arxiv.org/abs/2601.11655

[21] X. Wang et al., "The OpenHands Software Agent SDK: A Composable and Extensible Foundation for Production Agents," arXiv:2511.03690, 2025. [Online]. Available: https://arxiv.org/abs/2511.03690

[22] T. Warren, "GitHub's new AI coding agent can fix bugs for you," The Verge, 2025. [Online]. Available: https://www.theverge.com/news/669339/github-ai-coding-agent-fix-bugs

[23] T. Warren, "GitHub is launching a hub for multiple AI coding agents," The Verge, 2025. [Online]. Available: https://www.theverge.com/news/808032/github-ai-agent-hq-coding-openai-anthropic

[24] D. Preston, "Google Antigravity is an agent-first coding tool built for Gemini 3," The Verge, 2025. [Online]. Available: https://www.theverge.com/news/822833/google-antigravity-ide-coding-agent-gemini-3-pro

[25] S. Levy, "OpenAI, Anthropic, and Block Are Teaming Up to Make AI Agents Play Nice," Wired, 2025. [Online]. Available: https://www.wired.com/story/openai-anthropic-and-block-are-teaming-up-on-ai-agent-standards

[26] Z. Ji, Z. Li, W. Jiang, Y. Gao, and S. Wang, "Measuring the Permission Gate: A Stress-Test Evaluation of Claude Code's Auto Mode," arXiv:2604.04978, 2026. [Online]. Available: https://arxiv.org/abs/2604.04978

[27] P. Jin et al., "RealBench: Benchmarking Verilog Generation Models with Real-World IP Designs," arXiv:2507.16200, 2025. [Online]. Available: https://arxiv.org/abs/2507.16200

[28] G.-W. Wan et al., "FIXME: Towards End-to-End Benchmarking of LLM-Aided Design Verification," arXiv:2507.04276, 2025. [Online]. Available: https://arxiv.org/abs/2507.04276

[29] J. Pan, G. Zhou, C.-C. Chang, I. Jacobson, J. Hu, and Y. Chen, "A Survey of Research in Large Language Models for Electronic Design Automation," arXiv:2501.09655, 2025. [Online]. Available: https://arxiv.org/abs/2501.09655

[30] C. Guo et al., "A Survey: Collaborative Hardware and Software Design in the Era of Large Language Models," arXiv:2410.07265, 2024. [Online]. Available: https://arxiv.org/abs/2410.07265

[31] Q. Xu, L. Stok, R. Drechsler, X. Wang, G. L. Zhang, and I. L. Markov, "Revolution or Hype? Seeking the Limits of Large Models in Hardware Design," arXiv:2509.04905, 2025. [Online]. Available: https://arxiv.org/abs/2509.04905

[32] F. Xu, P. K. Medappa, M. M. Tunc, M. Vroegindeweij, and J. C. Fransoo, "AI-assisted Programming May Decrease the Productivity of Experienced Developers by Increasing Maintenance Burden," arXiv:2510.10165, 2025. [Online]. Available: https://arxiv.org/abs/2510.10165

[33] M. Alenezi, "Rethinking Software Engineering for Agentic AI Systems," arXiv:2604.10599, 2026. [Online]. Available: https://arxiv.org/abs/2604.10599