

Harness-Native Software Engineering: The Control Plane of Coding Agents

Chaitanya Mishra
Independent Researcher

March 2026

Abstract

Coding-agent systems are increasingly shaped by infrastructure outside the base model: repository guidance, tool permissions, execution sandboxes, persistent memory, verification loops, recovery mechanisms, and delegation logic. For long-horizon software tasks, those design choices often determine what an agent can see, what it may do, and whether it can recover from failure. This paper introduces Harness-Native Software Engineering as a framework for analyzing that shift and formalizes the surrounding runtime layer as the Agent Harness Control Plane, with eight functions: context ingress, action mediation, execution substrate, state persistence, verification and review, recovery and debugging, delegation and coordination, and governance and audit.

Drawing on recent system documentation, benchmark papers, configuration studies, adoption analyses, and security research, the paper advances five claims. Coding-agent performance is harness-sensitive. Recoverability matters alongside one-shot success. The effective security boundary sits at the harness interfaces through which tools, protocols, permissions, and sandboxes are exposed. Teams are externalizing agent behavior into version-controlled artifacts such as manifests, skills, and custom-agent definitions. And benchmark rankings can change when harness conditions or task realism change. The paper's contribution is conceptual and methodological: a reusable unit of analysis for coding agents, a comparative matrix of representative systems, a reporting instrument called the Harness Condition Sheet, a threat model for the harness layer, and a harness-sensitive protocol for future empirical work.

Keywords. coding agents; agentic software engineering; harness engineering; evaluation methodology; long-horizon software tasks; recoverability; repository context; agent security

1. Introduction

Coding agents are no longer well-described as autocomplete systems with larger prompts. The leading public systems now operate through terminals, IDE agents, cloud runners, pull-request workflows, persistent memory files, custom-agent manifests, approval policies, hooks, checkpoints, and subagents. Codex exposes approval modes, project guidance via `AGENTS.md`, cloud tasks, skills, Model Context Protocol (MCP) integration, and experimental multi-agent workflows. Claude Code exposes persistent project memory, auto memory, hooks, plugins, and custom subagents. GitHub Copilot's coding agent works through issue-to-pull-request execution in GitHub-hosted environments, with self-review, security scanning, and cross-agent memory. Gemini CLI exposes skills, checkpointing, subagents, hooks, and sandboxing. Cursor, OpenHands, and OpenClaw surface analogous structures around background execution, review agents, typed tool systems, isolated workspaces, and multi-agent routing [2, 3, 4, 5, 11, 12, 13, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27].

Recent research points in the same direction from a different angle. IDE-Bench replaces raw terminal interaction with an IDE-native tool ecosystem and a Dockerized test harness. VeRO introduces an evaluation harness for agents that optimize other agents and argues that uncontrolled execution makes such evaluation

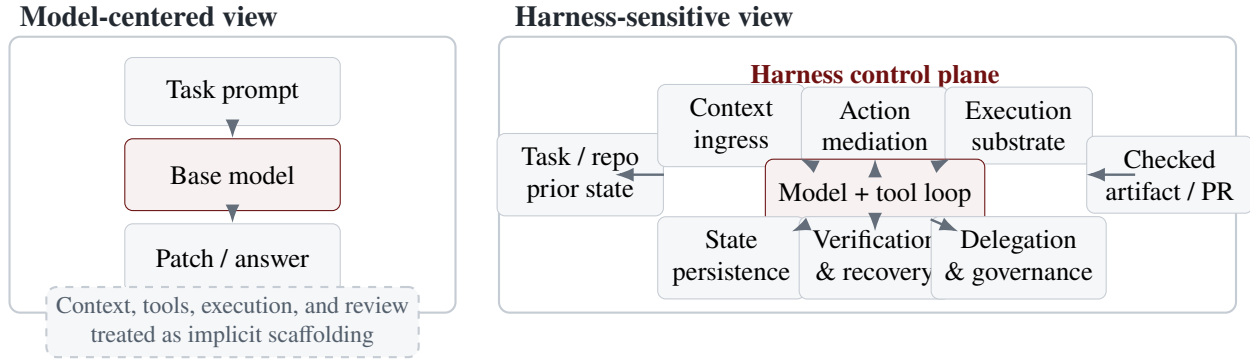


Figure 1. From model-centered to harness-sensitive evaluation. The right panel makes explicit the runtime layer that governs what the agent sees, what it may do, and how its outputs are checked.

unreliable. AutoHarness studies automatic synthesis of the harness itself. Anthropic’s January 2026 note on agent evaluation states the issue directly: when one evaluates an “agent,” one is evaluating a harness and a model working together. OpenAI’s 2026 Codex engineering report likewise frames progress as harness engineering: environment design, verification loops, repository legibility, feedback channels, and orchestration choices [28, 29, 30, 9, 1, 8].

The literature, however, is still fragmented. Benchmark papers often compare systems under only partially specified runtime conditions. Configuration studies focus on context files, skills, and subagents as artifacts. Security work concentrates on protocols, tool invocation, shell access, and prompt injection. Adoption work studies pull requests, commit traces, and repository-level patterns. Each cluster is useful. None provides a field-level language that makes these results mutually intelligible. The same layer now mediates capability, failure, cost, and governance, yet the field still lacks a stable name for it.

This paper calls that regime *Harness-Native Software Engineering*. The term refers to a stage of agentic coding in which realized capability, safety, cost, and continuity are increasingly determined by the *harness*: the model-external layer that curates context, mediates actions, provisions execution, persists state, runs verification, supports recovery, coordinates delegation, and enforces governance. The central claim is not that models no longer matter. It is that for many deployment-like coding tasks, especially long-horizon repository work, the base model is no longer the only informative unit of analysis.

The argument is deliberately narrower than broad visions of AI-native or agentic software engineering. Prior vision papers describe a larger transition toward AI teammates and software engineering 3.0 [57, 58]. Harness-Native Software Engineering focuses on a more specific question: where do the decisive variables sit *inside* modern coding-agent systems? The answer advanced here is that they increasingly sit in the control plane around the model.

Contributions. The paper makes five contributions.

1. It introduces Harness-Native Software Engineering as a field-level term for agentic coding systems whose realized behavior is primarily mediated by the harness.
2. It formalizes the Agent Harness Control Plane as an eight-function decomposition of that model-external control layer.
3. It synthesizes recent evidence across runtime systems, benchmark design, recovery and debugging, configuration artifacts, security work, and repository-scale adoption.

4. It proposes a reporting instrument, the Harness Condition Sheet, for benchmark and deployment descriptions.
5. It lays out a harness-sensitive research program based on fixed-model harness ablations, recoverability metrics, unsafe-action reporting, and condition-aware benchmark comparisons.

The rest of the paper proceeds as follows. Section 2 organizes the recent literature into coherent clusters. Section 3 formalizes the problem. Section 4 defines the Agent Harness Control Plane. Section 5 explains the analytical method. Section 6 uses comparative and case-based analysis to evaluate the framework. Sections 7 and 8 develop benchmark and security implications. Section 9 examines repository traces and organizational adoption. Sections 10–12 discuss implications, limits, and next steps.

2. Related Work

2.1 Runtime systems and exposed control surfaces

The most immediate evidence for a harness-centric framing comes from how recent coding-agent systems are built and documented. Codex is not described merely as a language model with file-editing tools. It is documented as a software agent that runs in a loop, can be configured through repository guidance and config layers, can delegate work to cloud tasks, can use MCP servers, can restrict or expand network access, and can participate in multi-agent workflows [2, 3, 4, 5, 6, 7]. Claude Code, GitHub Copilot’s coding agent, and Gemini CLI present comparable surfaces: persistent memory, hooks, subagents, review loops, checkpointing, sandboxes, and policy controls [11, 12, 13, 14, 17, 19, 20, 21, 22, 23]. OpenHands and OpenClaw make the same shift explicit in open tooling, exposing agent reasoning loops, workspace abstraction, typed tool actions, isolated workspaces, and routing across external harnesses [25, 26, 27].

This matters because it changes what counts as system design. In earlier code-assistant work, the system boundary often ended at prompt construction and a handful of tools. In current coding agents, the system boundary includes stateful directories, policy files, staged approvals, external protocols, self-review routines, and environment templates. The model remains central, but it is increasingly embedded in a programmable runtime that determines which observations reach it, what actions are legal, and how those actions are checked.

2.2 Harnesses as explicit research objects

A concept is usually worth naming when it begins to appear independently in both systems practice and research. That is now true of the harness. OpenAI’s February 2026 engineering report uses the term directly and describes progress in terms of repository legibility, environment shaping, observability, testing loops, and workflow structure rather than prompt phrasing alone [1]. Anthropic’s long-running-agent report describes a specialized harness built from the Claude Agent SDK primitives, explicitly separating initialization from execution and using persistent files and git state to carry progress across context windows [8]. Anthropic’s evaluation note then generalizes the point: an agent evaluation is an evaluation of a harness-model pair, not of the model in isolation [9].

Recent preprints extend that move into formal evaluation. IDE-Bench uses an IDE-native tool interface and a Dockerized test harness to measure engineering collaboration rather than raw terminal manipulation [28]. VeRO defines an evaluation harness for agent optimization, with versioned snapshots, budget-controlled execution, and structured traces [29]. AutoHarness treats the harness as a synthesizable program component and studies whether an LLM can write that layer automatically [30]. These works do not yet produce a

field-wide theory of coding-agent architecture, but they establish that the harness is no longer an informal implementation detail.

2.3 Recovery, debugging, and long-horizon continuity

Recovery has become a revealing fault line in agentic coding. Anthropic’s long-running harness is built around the fact that many software tasks exceed a single context window and that the agent must recover bearings repeatedly. The system therefore uses an initializer phase, explicit setup scripts, progress files, git commits, and a requirement to work incrementally through a feature list [8]. Debug2Fix takes a more targeted approach: it integrates interactive debuggers into a coding agent through a debug subagent and reports improvements of more than 20% in some settings, while also showing that better tooling can allow weaker models to match or exceed stronger ones [31]. AgentStepper shifts the focus from task completion to developer understanding and shows that debugging the *agent trajectory* itself benefits from a debugger-like interface with breakpoints, stepwise execution, and live manipulation of prompts and tool invocations [32].

These works jointly suggest that validated failure is not an edge case. It is a normal operating condition in long-horizon coding. If that is true, then recovery mechanisms are not auxiliary. They are first-order capability multipliers. A model that can generate plausible edits but cannot preserve coherent progress across failure states is less deployment-relevant than a somewhat weaker model inside a better recovery loop.

2.4 Benchmark realism and evaluation instability

Benchmark work over the last year has become much sharper about what current coding-agent scores do and do not mean. The Agentic Benchmark Checklist paper documents setup and reward-design errors that can lead to up to 100% relative over- or under-estimation, and it reports that applying the checklist to CVE-Bench reduces overestimation by 33% [33]. Saving SWE-Bench shows that rewriting benchmark prompts into realistic user-style requests can cut apparent capability by more than 50% for some systems [34]. FeatureBench reports that a state-of-the-art agentic model achieving 74.4% on SWE-bench solves only 11.0% of its feature-development tasks [35]. Terminal-Bench focuses on realistic terminal work and keeps frontier systems below the performance levels implied by more stylized coding benchmarks [36]. SWE-EVO shifts from isolated issue resolution to software evolution over many files and iterations, reporting a 21% resolution rate for GPT-5 with OpenHands versus 65% on SWE-Bench Verified [37]. ABC-Bench, RepoMod-Bench, OmniCode, and SWE-rebench all make complementary points about environment fidelity, execution, scale, task breadth, and contamination [39, 40, 41, 38].

The standard interpretation of this literature is that benchmarks need to get harder or more realistic. That interpretation is correct but incomplete. Many of these changes also alter the effective harness condition under which the agent is evaluated: task phrasing, available tools, hidden verification, environment setup, continuity horizon, and degrees of freedom around testing. A harness-sensitive framework provides a way to explain why benchmark conclusions shift so sharply under those changes.

2.5 Configuration artifacts and externalized context

A second literature cluster examines how teams configure coding agents in practice. The exploratory study of 2,926 repositories finds eight configuration mechanisms and identifies three dominant trends: context files dominate the landscape, AGENTS.md is emerging as an interoperable standard, and advanced mechanisms such as skills and subagents remain shallowly adopted [42]. Agent READMEs adds a large-scale empirical study of 2,303 context files and shows that these artifacts behave more like evolving configuration code than static documentation; developers emphasize build instructions, architecture, and implementation detail,

while security and performance appear much less often [43]. That asymmetry matters: it suggests that teams use context files primarily to make agents functional, not to constrain them safely.

The AGENTS.md evaluation literature is especially useful because it documents disagreement. One study of 10 repositories and 124 pull requests finds that AGENTS.md is associated with lower median runtime (28.64%) and reduced output-token consumption (16.58%) while preserving comparable task completion [44]. Another finds that context files often reduce success rates while increasing inference cost by more than 20%, even though they encourage broader exploration and more extensive testing [45]. The interesting point is not which study “wins.” The interesting point is that repository-visible instructions are already strong enough to change behavior, cost, and exploration patterns measurably. That is what one would expect if the harness has become a software artifact in its own right.

Work on codified context makes the same point from a single-system engineering perspective. A recent case study of a 108,000-line C# distributed system describes a three-component context infrastructure combining a hot-memory constitution, many specialized agents, and a cold-memory knowledge base [46]. Whether or not one adopts that design, the paper is evidence that persistent, explicit, and multi-layered context is becoming an operational substrate for coding agents rather than a prompt-writing trick.

2.6 Security boundaries and protocol attack surfaces

Security work is now difficult to reconcile with a model-only account of risk. The prompt-injection SoK on agentic coding assistants synthesizes 78 studies and reports that adaptive attacks against current defenses can exceed 85% success, while organizing attacks around delivery vectors, modalities, and propagation behavior [47]. Breaking the Protocol argues that MCP has architectural vulnerabilities at the protocol level, introduces MCPBench, and reports 23–41% attack amplification relative to equivalent non-MCP integrations, with a proposed protocol extension reducing attack success from 52.8% to 12.4% in the authors’ experiments [48]. “Your AI, My Shell” evaluates high-privilege coding editors with 314 payloads covering 70 MITRE ATT&CK techniques and reports attack success rates up to 84% for malicious command execution [49]. SEC-bench shifts from security incidents to capability evaluation and shows that current LLM agents achieve at most 18.0% success on proof-of-concept generation and 34.0% on vulnerability patching for real-world software-security tasks [50].

Official system documentation reacts at the same layer. Codex documents sandbox modes, network controls, project trust, approval policies, and warnings about prompt injection when internet access is enabled [6, 7]. GitHub Copilot’s coding agent emphasizes customizable environments, restricted permissions, and approval gates [13, 14]. Gemini CLI documents sandboxing and policy-governed action modes [23, 19]. The common target is neither raw next-token generation nor code quality in the abstract. It is the harness interface: where the agent reads, what it trusts, what it may execute, and which actions cross human or policy boundaries.

2.7 Adoption, repository traces, and workflow evidence

Repository-scale evidence now makes clear that coding agents are not a niche phenomenon. AIDev aggregates 932,791 agent-authored pull requests from 116,211 repositories and 72,189 developers across five prominent agents [51]. Agentic Much studies 129,134 projects and estimates adoption between 15.85% and 22.60% in the first half of 2025, arguing that the visible traces left by coding agents make large-scale analysis feasible [52]. A causal study comparing autonomous agents with IDE-based AI assistants finds front-loaded velocity gains when agents are the first observable AI tool in a project, but it also reports persistent quality risks, with static-analysis warnings and cognitive complexity increasing by roughly 18% and 39% [53].

A large-scale study of agent-generated pull requests finds substantial differences in commit count and files touched relative to human PRs, suggesting that agent-authored work is not just a larger version of ordinary human change sets [54]. Studies of review and integration add more texture: core developers more consistently require verification before acceptance, reviewer engagement strongly correlates with successful integration, and agent-authored pull requests exhibit distinctive review dynamics [55, 56].

These studies matter for this paper because they show where the harness leaves traces. Context files, custom-agent manifests, PR descriptions, review loops, memory systems, and issue-to-PR workflows are all observable artifacts of the control plane. Adoption therefore does not merely show that agents are used. It shows *how* teams are operationalizing them.

2.8 Missing lens

Taken together, these clusters describe a coherent shift that the current vocabulary does not fully capture. Runtime papers show exposed control surfaces. Recovery work shows that continuity depends on external state and debugging infrastructure. Benchmarks show large swings when runtime conditions become more realistic. Configuration studies show that agent behavior is being externalized into version-controlled artifacts. Security work shows that practical risk resides at protocol, tool, and shell boundaries. Adoption work shows those same artifacts entering repositories and review workflows at scale. The missing lens is a framework that explains why these findings belong together. The next sections propose that framework.

3. Problem Formulation

Let a coding-agent system be represented as

$$\mathcal{A} = \langle M, H, E \rangle,$$

where M denotes the base model or routed model set, H denotes the harness, and E denotes the execution environment. A task x is drawn from a distribution \mathcal{T} over repository-level software-engineering tasks. Running \mathcal{A} on x produces a trajectory τ consisting of observations, model inferences, tool calls, file edits, environment transitions, review events, and possible human interventions.

The crucial move is to treat H as an object of analysis rather than as undifferentiated scaffolding. A coding agent rarely observes x directly. It observes a harness-mediated view of x : issue text after preprocessing, repository files after search and selection, memory artifacts, context files, tool schemas, policy constraints, and environment state. Likewise, the agent rarely acts directly on the task. It acts through harness-mediated mechanisms: tool invocation, shell execution, file write gates, approval policies, cloud runners, review loops, and delegation channels.

Definition 1 (Harness). The *harness* is the model-external layer that determines which context is visible, which actions are available, where execution occurs, what state persists, how verification is performed, how failures are recovered, how work is delegated, and where governance is enforced.

That definition is intentionally broader than prompt construction and slightly narrower than the entire sociotechnical deployment stack. It includes the runtime elements that shape the agent’s realized behavior on a task. It does not include everything about the surrounding organization.

Definition 2 (Harness-Native Software Engineering). Harness-Native Software Engineering is the regime of agentic coding in which the realized capability, safety, cost, and continuity of a coding system are primarily mediated by the harness rather than by the base model alone.

Table 1. Core notation used throughout the paper.

Symbol	Meaning
\mathcal{A}	Coding-agent system
M	Base model or model-routing policy
H	Harness, treated here as the control plane
E	Execution environment
$x \sim \mathcal{T}$	Task drawn from a task distribution
τ	Realized agent trajectory on a task
h	Concrete harness condition used in a run or benchmark
VTS	Verified task success
RR	Recovery rate
UAR	Unsafe action rate
HIB	Human intervention burden
Δ_H	Harness sensitivity under a fixed model

The word “primarily” is important. The claim is comparative, not absolute. A stronger model can still dominate a weaker one. The question is whether, for deployment-like coding tasks, changes in the harness are large enough and common enough to deserve first-class treatment.

To make that comparative claim explicit, define the utility of a model-harness pair on a task as

$$U(M, h, x) = \alpha \cdot \text{VTS}(M, h, x) - \beta \cdot \text{Cost}(M, h, x) - \gamma \cdot \text{UAR}(M, h, x) - \delta \cdot \text{HIB}(M, h, x),$$

where $\alpha, \beta, \gamma, \delta > 0$ encode task- or organization-specific priorities. The exact utility function can change across settings. What matters is that it treats correctness, cost, safety, and intervention as joint outputs rather than separate side metrics.

Definition 3 (Harness sensitivity). For a fixed model M , task distribution \mathcal{T} , and two harness conditions h_1 and h_2 , define

$$\Delta_H(M; h_1, h_2, \mathcal{T}) = \mathbb{E}_{x \sim \mathcal{T}} [U(M, h_1, x) - U(M, h_2, x)].$$

A task regime is *harness-sensitive* when $|\Delta_H|$ is frequently non-trivial and can rival the gain from a model swap under a fixed harness.

The idea is not new in spirit. Anthropic’s evaluation note already says that an agent evaluation is an evaluation of harness plus model [9]. The contribution here is to turn that intuition into a reusable unit of analysis for coding agents specifically.

A second concept is recoverability. Long-horizon software work rarely proceeds without entering failure states: failing tests, broken builds, incorrect assumptions, policy-denied actions, or tool errors. Define the *recovery rate* as

$$\text{RR} = \frac{|\{\tau : \tau \text{ enters a validated failure state and later reaches verified success}\}|}{|\{\tau : \tau \text{ enters a validated failure state}\}|}.$$

This metric differs from raw task success because it treats failure as part of the trajectory rather than as a terminal label.

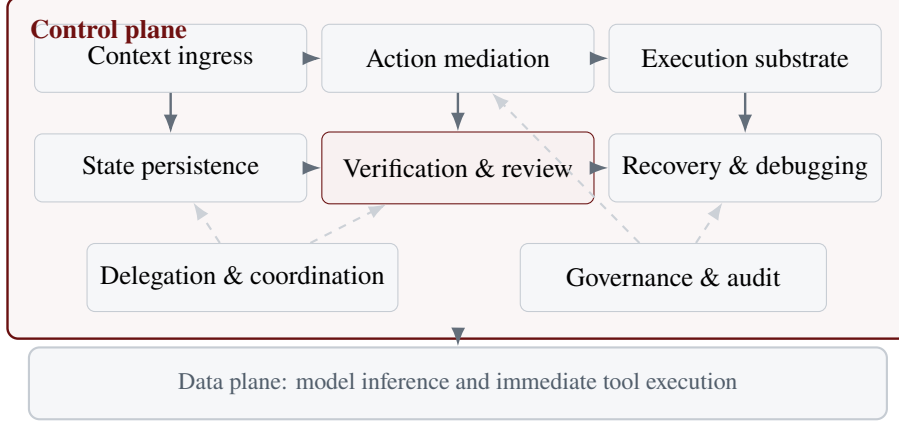


Figure 2. The Agent Harness Control Plane decomposed into eight control-plane functions. The functions are analytically distinct but operationally coupled; the model/tool loop sits beneath them as the data plane.

For safety and governance, define the *unsafe action rate* as

$$\text{UAR} = \frac{1}{N} \sum_{i=1}^N \mathbf{1} [\text{run } i \text{ attempts a policy-disallowed or security-relevant unsafe action}] .$$

Define *human intervention burden* as

$$\text{HIB} = \frac{\text{approvals} + \text{manual corrections} + \text{manual restarts}}{\max(1, \text{successful tasks})} .$$

These quantities are crude but necessary. An agent that “solves” tasks only by repeatedly crossing governance boundaries or requiring heavy babysitting is less deployment-relevant than a benchmark table might suggest.

The final conceptual move is to distinguish *data plane* from *control plane*. The data plane is the local loop of model inference and immediate tool execution. The control plane is the persistent orchestration layer that decides what context enters the loop, what actions are legal, where those actions run, how state persists, how outcomes are verified, how failures are repaired, and how delegation and human authority are structured. Harness-Native Software Engineering is the claim that, for many coding tasks, the control plane has become the right place to locate the field’s decisive variables.

4. The Agent Harness Control Plane

Definition 4 (Agent Harness Control Plane). The Agent Harness Control Plane is the tuple

$$H = \langle C, A, X, S, V, R, D, G \rangle,$$

where C is context ingress, A is action mediation, X is execution substrate, S is state persistence, V is verification and review, R is recovery and debugging, D is delegation and coordination, and G is governance and audit.

The choice of eight functions is meant to be analytically useful rather than metaphysically final. The decomposition separates variables that are often bundled together in product narratives but that affect evaluation, security, and adoption in different ways.

4.1 Context ingress

Context ingress covers everything that determines what the agent sees before or during action: task descriptions, repository files after search and retrieval, issue and pull-request state, memory artifacts, context files, skills, and auxiliary documentation. Codex documents project guidance via `AGENTS.md`; Claude Code documents persistent `CLAUDE.md` files and auto memory; Gemini distinguishes between persistent workspace context and on-demand skills; configuration studies show that these repository-visible artifacts now appear across major tools and are increasingly standardized [3, 11, 20, 42, 43].

Treating context ingress as a first-class control-plane function clarifies two facts that otherwise look contradictory. First, context files can improve efficiency by reducing rediscovery work, as seen in the `AGENTS.md` efficiency study [44]. Second, they can degrade success and increase cost when they inject unnecessary or poorly scoped requirements, as seen in the negative `AGENTS.md` evaluation [45]. These are not mutually exclusive findings. They show that context ingress is an active design variable. In a harness-native regime, repository context is neither neutral background nor merely prompt text. It is executable guidance that shapes search, testing, and constraint following.

4.2 Action mediation

Action mediation includes tool exposure, tool schemas, permission models, protocol bindings, approval gates, and policy filters. The relevant question is not just whether a tool exists. It is whether the agent can discover it, invoke it correctly, route through a protocol server, escalate privileges, or act without human approval. Codex, GitHub Copilot, Gemini CLI, and OpenClaw all expose explicit surfaces at this layer: approvals, MCP or ACP bindings, custom agents, routing, and policy modes [2, 4, 6, 13, 14, 19, 23, 26].

This layer is also where practical capability and risk meet. A debugger hidden behind poor tool descriptions may never be used. A shell exposed without trust boundaries can become an attack surface. MCP servers can extend reach, but protocol trust becomes part of the system's threat model [48, 47]. The harness therefore does more than provide tools. It regulates the legal action space of the agent.

4.3 Execution substrate

Execution substrate refers to the place where code runs and side effects occur: local terminal, editor sandbox, container, worktree, CI runner, or cloud task environment. The shift from chat assistants to repo-native agents is largely a shift in execution substrate. GitHub Copilot's coding agent starts in a customizable GitHub Actions environment. Codex delegates work to cloud tasks with configurable environments and network policies. OpenHands structures agent execution around workspaces and agent servers. OpenClaw isolates agents into separate workspaces and sessions. IDE-Bench and ABC-Bench show why this matters for evaluation: environment setup and execution fidelity are not decorations but determinants of what the agent can validate [13, 2, 7, 25, 27, 28, 39].

Execution substrate often sits hidden in benchmark reporting. Yet a model paired with a rich, reproducible runner is a materially different system from the same model paired with a bare tool sandbox. Harness-Native Software Engineering treats that distinction as central rather than incidental.

4.4 State persistence

State persistence covers anything that survives beyond a single context window: memory files, checkpoints, progress notes, git history, resumed threads, and saved sessions. Anthropic's long-running harness uses explicit progress files and git commits to maintain continuity across context windows [8]. Claude Code

documents persistent project instructions and auto memory [11]. Gemini CLI exposes checkpointing that captures a snapshot before file modification [21]. GitHub describes cross-agent memory that allows coding agent, CLI, and code review to share repository-specific insights [17]. Codified Context extends the same logic into a larger architecture of hot and cold memory [46].

Without this layer, long-horizon autonomy is often illusory. The agent appears persistent only because humans repeatedly restate or reconstruct context. A harness-native view makes explicit that continuity is achieved by moving state from the ephemeral model context into durable external artifacts.

4.5 Verification and review

Verification and review include tests, linters, browser checks, self-review, PR review, security scanning, and CI gating. OpenAI’s internal Codex case study emphasizes repeated testing, self-review, and review loops inside the development process rather than after it [1]. GitHub’s coding agent now includes self-review and built-in security scanning [14]. Cursor’s Bugbot reviews pull requests automatically [24]. Anthropic’s long-running harness constrains agents to make incremental, testable progress through explicit feature lists [8]. Review studies show that integration outcomes depend strongly on review engagement and coordination signals, not only on the initial code patch [56, 55].

The key point is structural. Verification is not just an endpoint metric. It is a control-plane mechanism that changes subsequent action. A failing test may trigger rollback, delegation, or more targeted search. A self-review step may catch an omission before human review. When verification is weak or absent, the harness loses one of its main correction channels.

4.6 Recovery and debugging

Recovery and debugging capture how the system responds after a wrong turn. Anthropic’s harness explicitly assumes that long tasks will cross many context windows and uses startup scripts, progress notes, and small validated steps to recover coherence [8]. Debug2Fix shows that interactive debuggers, exposed through a debug subagent, can shift benchmark outcomes materially [31]. AgentStepper shows that developers need analogous capabilities for debugging the agent program and the trajectory it produces [32]. Checkpointing in Gemini CLI is a lighter version of the same idea: enable reversible edits and safe experimentation [21].

The function is distinct from verification, even though the two interact closely. Verification answers “is the current state acceptable?” Recovery answers “given that it is not, how does the system resume productive work without losing trajectory coherence?” That difference is essential for long-horizon tasks.

4.7 Delegation and coordination

Delegation and coordination concern how work is partitioned across subagents, specialist tools, human reviewers, or external harnesses. Codex documents experimental multi-agent collaboration [5]. Claude Code and Gemini CLI both document custom subagents [12, 22]. GitHub’s mission control surfaces assignment, steering, restart, and oversight for parallel coding-agent tasks [15]. GitHub’s engineering guidance on multi-agent workflows argues that many failures arise from shared-state ambiguity, weak handoffs, and missing coordination structure rather than from pure model incompetence [16]. OpenClaw routes requests to separate agent workspaces or to external harnesses through ACP sessions [27, 26].

This function matters because specialization and parallelism are increasingly common. It also introduces new failure modes: duplicated work, conflicting edits, inconsistent assumptions, and policy fragmentation across agents. A control-plane vocabulary is needed to describe those failures precisely.

4.8 Governance and audit

Governance and audit include branch protections, admin requirements, sandbox rules, repository trust, audit trails, approval policies, and provenance. Codex documents project trust, sandbox modes, internet controls, and approval policies [6, 7]. GitHub’s coding agent centers human approval and enterprise-facing workflow controls [13, 14]. Gemini documents sandboxing and policy-governed modes [23]. Security work makes clear why this layer matters: it is the place where prompt injection, protocol abuse, and malicious command execution become bounded or unbounded [47, 48, 49].

Governance is often described in enterprise language and therefore dismissed as secondary. For coding agents it is not secondary. It is part of the architecture that determines whether an agent can be trusted to act at all.

4.9 Why a new term is needed

One could describe these eight functions separately without coining a new term. The reason to coin Harness-Native Software Engineering is that the functions are now empirically entangled. Context files affect search and testing. Protocol bindings affect both capability and attack surface. Checkpointing affects recovery and governance. Review loops affect acceptance, cost, and security. Benchmark scores shift when these conditions change. A new term is justified only if it organizes the field better than existing language. The claim here is that “coding agent” is too broad, “context engineering” is too narrow, and “tool use” is too shallow. Harness-Native Software Engineering names the specific architectural regime in which these functions jointly determine realized software-engineering behavior.

5. Methodology

This is a conceptual and methodological paper grounded in public evidence. It does not report a new controlled benchmark run. It instead constructs a framework and evaluates it against recent system documentation, engineering reports, benchmark papers, security analyses, configuration studies, and adoption datasets. That choice trades experimental novelty for analytical breadth. Given the pace of change in the area, that trade is preferable to presenting weak or non-reproducible pseudo-experiments.

The source corpus was selected to cover four evidence classes. The first class is *system-surface evidence*: official documentation and engineering reports that show what frontier coding-agent systems actually expose in their runtime and policy layers [1, 2, 8, 13, 19, 25, 27]. The second class is *benchmark evidence*: papers that stress the gap between stylized evaluation and repository-level or long-horizon engineering [33, 34, 35, 36, 37, 39, 40, 41, 38, 28, 29]. The third class is *artifact and workflow evidence*: configuration studies and repository-scale adoption work that reveal how the harness becomes visible in practice [42, 43, 44, 45, 51, 52, 53, 55]. The fourth class is *security evidence*: papers that attack protocols, skills, tools, and shell surfaces directly [47, 48, 49, 50].

The paper uses these sources in four distinct ways. *Empirical observations* are direct claims reported in the cited sources, such as benchmark drops, documented system features, or measured attack success. *Synthesis across sources* combines observations from multiple clusters into broader patterns, for example the link between benchmark instability and under-specified harness conditions. *Theoretical interpretation* appears where the paper proposes concepts such as control plane, harness sensitivity, or recovery dominance. *Speculation* is reserved for the research agenda and future empirical predictions. Keeping these layers separate is important because the paper is strongest as a framing and methodology contribution, not as a substitute for new experiments.

Table 2. Representative systems through the Agent Harness Control Plane lens. The entries are intentionally high-level; the goal is to locate exposed control-plane functions rather than to exhaust product detail.

System	Modality	Context ingress	Action / execution	Persistence / recovery	Verification / delegation / governance
Codex	CLI + cloud	AGENTS.md, config layers, skills	approvals, MCP, cloud tasks, sandbox modes	resumed tasks, project config, internet controls	self-review loops, experimental multi-agents, admin controls
Claude Code	CLI / SDK	CLAUDE.md, auto memory	hooks, plugins, MCP, tool permissions	memory, git-oriented long-horizon harness patterns	subagents, policy settings, harness-based eval framing
GitHub Copilot coding agent	GitHub-native	issue / PR state, custom agents, agentic memory	customizable Actions environments, repo permissions	workflow-linked memory across coding agent, CLI, code review	self-review, security scanning, mission control, human approval
Gemini CLI	CLI	workspace context, skills	tools, hooks, MCP, policy modes, sandboxing	checkpointing, subagents	experimental subagents, restore points, governed action modes
Cursor	Editor + background agent	project rules, memories	editor tools, background agent, MCP integration	background execution, memories	Bugbot review, product-level access controls
OpenHands	SDK / CLI / cloud	conversations, skills, retrieval	typed tools, workspace abstraction, agent server	explicit state management in SDK	extensible orchestration and validation flows
OpenClaw	Router / gateway	per-agent workspace files, notes, routing bindings	ACP sessions, isolated workspaces	session and agent directories	multi-agent routing, per-agent sandbox and tool restrictions

Representative systems were chosen to maximize architectural coverage rather than market share. The comparison set includes Codex, Claude Code, GitHub Copilot coding agent, Gemini CLI, Cursor, OpenHands, and OpenClaw. This set spans terminal-native, editor-native, cloud-native, repo-native, and open orchestration settings. It also includes both product systems and open frameworks, which helps separate surface branding from architectural pattern.

The validation criteria are fourfold. *Explanatory breadth* asks whether the framework can organize the major recent literatures without forcing them. *Discriminative power* asks whether the framework distinguishes systems in ways that matter for capability, risk, or evaluation. *Predictive adequacy* asks whether the framework helps explain observed benchmark instability, recovery effects, and security boundary shifts. *Design utility* asks whether the framework yields useful concrete outputs, such as the Harness Condition Sheet, a threat model, or proposed metrics for future work.

6. Comparative Analysis

6.1 Comparative system matrix

Table 2 summarizes representative systems through the Agent Harness Control Plane lens. The point is not to produce a leaderboard. It is to show that current coding-agent systems increasingly differ less by whether they have a harness and more by how explicit, programmable, and governable that harness is.

Three patterns stand out. First, context is increasingly externalized into durable artifacts rather than left inside transient prompts. Second, execution is increasingly split across substrates—local terminals, cloud sandboxes, CI-style runners, and protocol-connected tools. Third, governance is moving upward into explicit policy and audit surfaces. These are the structural features that motivate the control-plane language.

6.2 Proposition 1: Harness sensitivity

Proposition 1 (Harness sensitivity). *For many repository-level and long-horizon coding tasks, there exist harness conditions h_1 and h_2 such that $|\Delta_H(M; h_1, h_2, \mathcal{T})|$ is comparable to, or larger than, the utility gain obtained by replacing M with a stronger model under fixed h .*

The strongest direct evidence comes from results that disturb intuitive model orderings. Debug2Fix reports that adding a debug subagent and interactive debugger integration can materially improve bug-fixing performance and can allow weaker models to match or exceed stronger ones in some settings [31]. This is exactly the kind of result the proposition predicts: a harness change reorders realized performance without changing the task family.

The AGENTS.md literature provides a second, subtler form of evidence. One study finds lower runtime and lower output-token consumption with comparable completion [44]; another finds lower success and higher cost but more extensive exploration and testing [45]. The disagreement is not noise to be ignored. It shows that manifest quality and repository-level instructions are themselves consequential experimental variables. The same model, pointed at the same repository class, can behave differently under different guidance layers.

Practice reports make the same point without formal ablations. OpenAI’s internal Codex case study describes early failures not mainly as model limitations but as missing environment structure, missing feedback loops, and insufficient repository legibility [1]. Anthropic’s long-running harness likewise improves continuity by restructuring initialization, state handoff, and incremental verification [8]. The interpretation advanced here is theoretical rather than experimental: once a model crosses a threshold of basic repository competence, realized behavior becomes strongly harness-sensitive.

6.3 Proposition 2: Recovery dominance

Proposition 2 (Recovery dominance). *On long-horizon coding tasks, recoverability becomes a major determinant of verified success once basic repository competence is present.*

Anthropic’s long-running harness is built around the assumption that hours- or days-long work will repeatedly lose context, hit partial failures, and require re-entry. The harness responds by externalizing state into progress files, setup scripts, git commits, and structured feature lists [8]. Debug2Fix and AgentStepper show that runtime investigation and trajectory debugging are not luxuries but productive capabilities for both the agent and the developer of the agent [31, 32]. These systems are all attempts to turn failure from a terminal event into a manageable transition.

Benchmark results support the same interpretation. FeatureBench’s 11.0% success rate on end-to-end feature work, Terminal-Bench’s low performance on realistic command-line tasks, and SWE-EVO’s sharp drop from SWE-Bench Verified performance all indicate that current agents struggle most when the task requires sustained adaptation rather than a single clean fix [35, 36, 37]. A model-only reading says that the task is simply harder. A harness-native reading says that long-horizon difficulty is inseparable from weak recovery mechanisms.

6.4 Proposition 3: Artifact externalization

Proposition 3 (Artifact externalization). *As coding-agent adoption matures, teams increasingly encode agent behavior into repository-visible and version-controlled artifacts such as context files, skills, custom agents, policies, and memory structures.*

The configuration literature is already clear on the first half of this claim. Context files dominate configuration mechanisms across tools, and AGENTS.md is emerging as an interoperable artifact [42]. Agent READMEs shows that these files evolve frequently and contain instruction types that resemble operational policy as much as documentation [43]. Product systems now treat these artifacts as standard surfaces: AGENTS.md, CLAUDE.md, GEMINI.md-style context, skills, custom agents, and cross-agent memory are all documented product features [3, 11, 20, 18, 17].

Adoption data supports the second half. AIDev and Agentic Much both show that coding agents already leave strong traces in repositories, pull requests, commits, and configuration artifacts [51, 52]. In a model-centric regime, most customization would stay hidden inside prompts or UI settings. In a harness-native regime, customization becomes shareable infrastructure, which is exactly what repository-visible manifests and skills provide.

6.5 Proposition 4: Boundary displacement

Proposition 4 (Boundary displacement). *For deployed coding agents, the dominant actionable security boundary has shifted from the model surface to the harness interfaces that regulate tools, protocols, permissions, and execution.*

Security work gives this proposition unusually sharp evidence. The prompt-injection SoK argues that coding-assistant risk is fundamentally amplified by access to tools, shells, and protocol ecosystems [47]. Breaking the Protocol argues that MCP’s vulnerabilities are architectural and protocol-level rather than isolated implementation mistakes [48]. AIShellJack shows that high-privilege coding editors can be remotely induced to execute malicious commands with attack success rates up to 84% [49]. None of these failures can be understood well by looking only at next-token prediction. They sit at the harness boundary: what untrusted content can influence, what tool it can reach, and which permissions are ambient.

Official documentation converges on the same diagnosis from the defense side. Codex warns that internet access can turn prompt injection into data exfiltration or unsafe edits and therefore recommends limiting network scope aggressively [7]. GitHub and Gemini likewise foreground sandboxes, environment controls, and policy-gated actions [13, 23]. If the practical defenses live in the harness, then the practical boundary does too.

6.6 Proposition 5: Evaluation inversion

Proposition 5 (Evaluation inversion). *Benchmarks that under-specify or unrealistically fix the harness condition can systematically misestimate real deployment capability, and ranking conclusions may invert when the harness condition changes.*

The Agentic Benchmark Checklist, Saving SWE-Bench, FeatureBench, Terminal-Bench, and SWE-EVO already show large swings in measured capability when task setup, realism, horizon, and hidden validation change [33, 34, 35, 36, 37]. IDE-Bench, VeRO, ABC-Bench, RepoMod-Bench, and SWE-rebench reinforce the same point from different directions: realistic execution, hidden tests, scale, contamination control, and structured runtime conditions all matter [28, 29, 39, 40, 38]. The harness-native interpretation is straightforward. Those conditions are not peripheral metadata. They are pieces of the evaluated system.

This proposition has a practical consequence. Claims such as “model M achieves $p\%$ on benchmark B ” should often be read as shorthand for “model M under harness condition h achieves $p\%$ on benchmark B .” When h is weakly specified, comparison claims become harder to interpret than current practice admits.

6.7 Boundary conditions

The framework does not imply that the harness always dominates. Very short, self-contained editing tasks can remain strongly model-limited. Static code-generation settings with minimal environment interaction also fit the framework less well. A very weak model can still fail despite a strong harness. The claim of Harness-Native Software Engineering is therefore regime-specific: it is about deployment-like coding tasks that involve repository state, tool use, verification, or continuity over time.

6.8 Stress tests for the taxonomy

A useful taxonomy should survive contact with representative task families.

Feature development. FeatureBench stresses context ingress, execution substrate, and verification. A system needs architectural context, the ability to run or simulate the application, and hidden or independent validation to avoid passing only visible tests [35].

Long-running bug fixing. Anthropic’s long-horizon case stresses state persistence and recovery. Without durable progress artifacts and re-entry structure, the agent repeatedly spends context budget rediscovering state [8].

Security remediation. SEC-bench and prompt-injection work stress action mediation and governance. Security performance depends not only on secure coding knowledge but also on the harness’s ability to constrain unsafe actions and isolate untrusted content [50, 47, 49].

Parallel agent workflows. GitHub’s mission control and multi-agent engineering guidance stress delegation, coordination, and governance. Shared-state ambiguity and weak handoffs become visible even when individual subtasks are within model capability [15, 16].

These stress tests do not prove the framework, but they show that the decomposition remains interpretable across distinct task regimes.

7. Benchmark and Evaluation Protocol

Current coding-agent benchmark reporting usually includes the model, token budget, and sometimes a terse description of available tools. That is not enough. If the harness condition can change realized behavior materially, then benchmark papers need a disciplined way to state what was actually evaluated.

Definition 5 (Harness Condition Sheet). A Harness Condition Sheet is a structured report of the concrete harness condition used in a benchmark run or deployment study. At minimum it records context artifacts, tool and protocol exposure, permission and network policy, execution substrate, persistence features, verification stack, recovery mechanisms, delegation topology, human approval rules, and relevant cost or time budgets.

Table 3 maps prominent benchmark families to the Agent Harness Control Plane functions they stress. No single benchmark spans the entire control plane. That is not a flaw by itself. It becomes a flaw only when benchmark results are interpreted as if they were broader than their stressed functions justify.

The Harness Condition Sheet matters for three reasons. First, it increases reproducibility. Second, it helps separate model effects from harness effects. Third, it discourages a common failure mode in benchmark communication: treating under-reported runtime conditions as if they were negligible. If one model is evaluated

Table 3. Benchmark families and the control-plane functions they primarily stress.

Benchmark	Primary task class	What it makes more realistic	Main Agent Harness Control Plane functions stressed
IDE-Bench	IDE-native engineering tasks	Structured IDE tools and private-codebase realism	Context ingress, action mediation, execution substrate
Terminal-Bench	Command-line tasks	Real CLI execution and professional terminal workflows	Action mediation, execution substrate, recovery
FeatureBench	Feature implementation	Multi-commit, end-to-end feature work	Context ingress, execution substrate, verification
SWE-EVO	Software evolution	Long-horizon multi-file modifications	State persistence, recovery, verification
ABC-Bench	Backend engineering	Containerized services and external API validation	Execution substrate, verification, governance
RepoMod-Bench	Repository modernization	Large-scale repository translation with hidden tests	Context ingress, execution substrate, verification
SEC-bench	Security tasks	Authentic security workflows in isolated environments	Action mediation, governance, verification
VeRO	Agent optimization	Versioned agent snapshots and budget-controlled traces	State persistence, verification, governance
OmniCode	Broader SWE tasks	Task diversity across languages and categories	Context ingress, verification
SWE-rebench	Fresh task collection	Contamination-aware evaluation and task freshness	Context ingress, execution substrate, reporting discipline

with richer retrieval, broader shell access, stricter self-review, or more permissive retry logic, then a fair comparison requires that those differences be visible.

A minimal Harness Condition Sheet for coding-agent benchmarks should contain at least the following fields:

1. base model(s), routing logic, and reasoning settings;
2. interface mode (CLI, IDE, cloud, PR-native, or hybrid);
3. context artifacts loaded (AGENTS.md, skills, memories, issue state, PR state);
4. tools and protocol bindings (MCP, ACP, browser automation, debugger, package managers);
5. permission and approval policy;
6. internet and external-resource policy;
7. execution substrate (local machine, container, CI runner, cloud task);
8. persistence features (checkpointing, progress files, resumed sessions);
9. verification stack (tests, linters, self-review, security scans, CI);
10. recovery mechanisms (rollback, restart, checkpoint restore, debug subagents);
11. delegation topology (single agent, specialist subagents, multi-agent, human escalation);
12. budget constraints (token, wall-clock, retry count, compute cost).

A harness-sensitive empirical agenda then becomes possible. At minimum, future work should perform fixed-model ablations over: (i) repository guidance quality, (ii) verification depth, (iii) sandbox and network policy, (iv) persistence and checkpointing, (v) debugger access, and (vi) delegation topology. The dependent variables should extend beyond task success to include RR, UAR, HIB, token and wall-clock cost, and rates of human override.

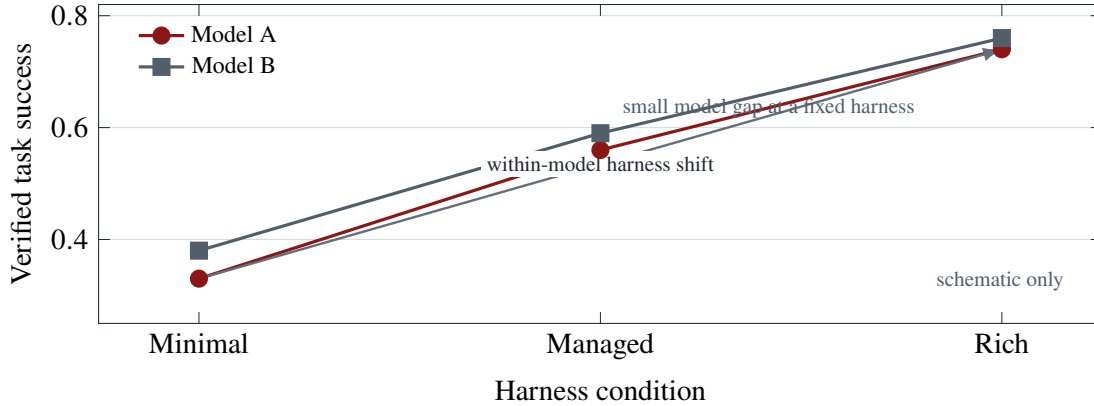


Figure 3. Schematic illustration of harness sensitivity. A shift in harness condition can move a fixed model more than the gap between two models evaluated under the same condition.

Table 4. Illustrative threat mapping from attack vector to Agent Harness Control Plane function.

Threat vector	Main Harness function	Agent Control Plane	Typical failure mode	Representative evidence
Poisoned issue / README / web content	Context ingress		Untrusted instructions become effective policy or action requests	Prompt-injection SoK [47]
Malicious shell or package command	Action mediation	me-governance +	Prompted command execution with high privilege	AIShellJack [49]
Compromised MCP server or metadata	Action mediation	media-	Capability spoofing, prompt injection, trust propagation	MCP security analysis [48]
Over-permissive network access	Governance + execution substrate		Exfiltration or unsafe dependency access	Codex internet-access guidance [7]
Weak review / CI controls	Verification governance	+ governance	Harmful patches or security regressions slip through	Copilot coding-agent docs [13, 14]
Persistent poisoned memory or context files	State persistence	persis-	Malicious or stale instructions survive session boundaries	Context-artifact literature [43, 46]

8. Security and Governance

If the harness is the control plane of a coding agent, then the threat model must center on harness interfaces rather than on the model alone. The relevant assets include repository contents, credentials, environment variables, shell access, network reachability, memory artifacts, review channels, and provenance data. The relevant entry points include issue text, pull-request comments, retrieved web content, context files, protocol servers, plugin or skill metadata, tool outputs, and transitive content fetched by tools.

The attacker goals documented in recent work are concrete: induce malicious command execution, exfiltrate data, escalate privilege, poison memory or context, manipulate protocol trust, or degrade review and verification workflows [47, 48, 49]. These goals map cleanly onto control-plane functions. Prompt injection against a retrieved README is primarily a context-ingress problem. Malicious shell execution is an action-mediation and governance problem. MCP impersonation or privilege misrepresentation is an action-mediation and protocol-governance problem. Unsafe persistence of attacker-controlled instructions is a state-persistence problem.

A harness-native security posture follows directly from this mapping.

Minimize ambient authority. Default-deny policies, read-only modes, restricted network access, and narrow tool allowlists matter because the agent’s action space is the immediate attack surface. Codex and Gemini both document constrained modes and network controls for this reason [6, 7, 23].

Treat external content as untrusted code-adjacent input. Issue text, PR discussions, fetched documentation, and tool outputs should not be treated as benign natural language. The security literature shows that untrusted natural-language content can become an effective control signal when it flows into the agent loop without origin-aware guardrails [47, 48].

Make state reversible. Checkpointing, explicit rollback, and durable git checkpoints are not just convenience features. They limit blast radius after bad actions and make incident recovery practical [8, 21].

Keep verification independent when possible. A harness that only asks the agent to decide whether it succeeded is structurally weak. Hidden tests, external API checks, CI gates, and human review provide partial independence from the model’s self-assessment [39, 40, 14].

Audit delegation. Multi-agent systems introduce new trust boundaries. Routing decisions, agent identities, and per-agent permissions should be logged and constrained, especially when specialized agents have different tool or network access [16, 27].

The broader point is architectural. Security is not a separate annex to coding-agent design. It is built into the same control-plane choices that shape capability and cost.

9. Workflow Integration and Organizational Implications

Repository-scale studies show that coding agents are now entering real development workflows in visible ways. AIDev documents nearly a million agent-authored pull requests across more than one hundred thousand repositories [51]. Agentic Much estimates very high early adoption, between 15.85% and 22.60% across 129,134 studied projects [52]. These studies matter because they shift the discussion from demos and benchmark scores to evidence of sustained repository-level use.

A harness-native view helps explain the adoption pattern. What teams adopt is not just a model endpoint. They adopt an operational package: how tasks are described, how repository context is surfaced, how execution is provisioned, how review is staged, what memory persists, and how agent behavior is externalized into manifests or custom agents. That is why repository-visible artifacts matter so much. They are the durable parts of the harness that teams can inspect, version, and share.

The pull request is emerging as a particularly important coordination interface. Agent-authored work often arrives not as a suggestion inside the IDE but as a branch, patch, or pull request that must survive review and verification. Studies of core and peripheral developers show that both groups review and modify agent-authored PRs differently, with core developers more consistently insisting on verification and peripheral developers more likely to merge without CI checks [55]. A study of collaboration signals finds that reviewer engagement is more predictive of successful integration than simple iteration intensity, while force pushes and larger change sets correlate with lower merge probability [56]. These are control-plane facts. They speak to governance, review structure, and coordination stability.

The quality picture is mixed. The longitudinal causal study by Agarwal et al. finds front-loaded velocity gains but persistent maintainability risks, including increases in static-analysis warnings and cognitive complexity [53]. The large-scale PR study by Ogenwot and Businge shows that agentic PRs differ substantially from human PRs in commit count and change structure [54]. These findings cut against simplistic narratives of either universal acceleration or universal failure. They instead suggest that the harness used for review, validation, and integration determines whether apparent throughput turns into acceptable software.

The growth of agentic memory systems reinforces the same point. GitHub’s cross-agent memory is explicitly framed as repository-specific knowledge shared across coding agent, CLI, and code review [17]. Claude Code’s auto memory and persistent project instructions perform a similar function [11]. These are not merely convenience features. They are ways of stabilizing behavior over repeated interactions with the same codebase.

Several organizational implications follow.

Behavior becomes policy-bearing infrastructure. When project rules move into `AGENTS.md`, custom-agent definitions, skills, and memory layers, they stop being informal team norms and become machine-consumable policy. That raises the quality bar for those artifacts and makes their review process consequential.

Review remains central. The empirical literature does not support eliminating human oversight. It supports making review more structured and more selective. Agents can generate candidate work at scale; acceptance still depends heavily on review and verification quality [56, 55].

Selective deployment is rational. The causal evidence suggests diminishing returns and persistent quality risks in some settings [53]. That argues against blanket organizational adoption and in favor of task-specific harness design: narrow permissions, well-defined validation, and explicit boundaries for when an agent should or should not act autonomously.

Provenance becomes more important. As agent-authored work moves through pull requests, memories, and subagents, organizations need traceability not just for compliance but for debugging and accountability. Governance is easier when the harness itself makes actions legible.

10. Discussion

10.1 Why Harness-Native Software Engineering is not redundant

Several adjacent terms already circulate in the literature and in practice. The closest is *context engineering*. Anthropic’s context-engineering note correctly identifies context as a finite and central resource [10]. The configuration literature reaches similar conclusions from repository artifacts [42, 43]. But context engineering is only one slice of the present framework. It does not, by itself, capture approval policy, checkpointing, execution substrate, review loops, protocol attack surface, or multi-agent coordination. Harness-Native Software Engineering contains context engineering but is not reducible to it.

A second adjacent term is *repo-native agency*. That phrase captures the important shift from chat surfaces to agents working inside real repositories. The present paper agrees with that diagnosis. It adds that repository state alone is still not the right abstraction. Two systems can both be repo-native while differing radically in permissions, persistence, debugging ability, and governance. Those differences belong to the harness.

A third adjacent frame is the broader literature on AI-native or agentic software engineering [57, 58]. That literature addresses a larger field transition: new human-agent roles, changes in lifecycle structure, and the emergence of AI teammates. Harness-Native Software Engineering is narrower. It asks where capability and risk now reside within coding-agent systems. Its answer is architectural rather than civilizational.

10.2 Objections

Objection 1: The framework overstates the harness and understates the model. That objection would be decisive only if the paper claimed that the model no longer matters. It does not. The claim is regime-specific and comparative. For short or static tasks, the model may dominate. For long-horizon repository work, the cited literature suggests that harness condition is often large enough to deserve equal analytical status.

Objection 2: “Harness” is just engineering plumbing, not a research concept. The recent literature weakens that objection. Once harnesses become explicit objects in engineering reports, evaluation papers, configuration studies, and security analyses, they stop being mere plumbing. They become the place where many of the field’s operative variables now live [1, 9, 29, 30].

Objection 3: The argument relies too heavily on product documentation. Official documentation is not used here as evidence of objective superiority. It is used as primary evidence of what features and control surfaces major systems actually expose. The framework is grounded in those docs *plus* independent benchmark, configuration, security, and adoption work.

Objection 4: The work is conceptual rather than experimental. Correct. The contribution is a framing, a decomposition, and a methodology. The paper earns its keep only if those things help organize current evidence and improve future experimentation. The Harness Condition Sheet, the proposed metrics, and the threat model are intended to be concrete outputs of that kind.

10.3 What the framework predicts

If the framework is useful, several predictions follow.

First, future benchmark papers will increasingly report harness details because model-only reporting will look insufficient. Second, recoverability metrics will become more informative than one-shot pass rates on long-horizon tasks. Third, more organizational customization will move into version-controlled artifacts rather than hidden prompts. Fourth, security incidents will continue to cluster at protocol, tool, and execution boundaries until those layers receive stronger standardization and attestation. Fifth, competition among coding agents will shift toward the design quality of the harness as much as toward the base model.

None of these predictions is guaranteed. They are offered as falsifiable consequences of the framework, not as certainty.

11. Limitations

This paper is conceptual and methodological. It does not report new controlled experiments, and its strongest claims therefore remain propositions rather than universal empirical laws. The framework should be read as a disciplined synthesis and a research agenda, not as the last word on causal magnitude.

The source base mixes official documentation, engineering reports, and recent preprints. That is appropriate for a fast-moving topic but has drawbacks. Documentation describes exposed features, not necessarily real-world reliability. Preprints can change. Product surfaces also move quickly enough that any static comparative matrix will age.

A second limitation is selection bias. The representative systems studied here are public and well-documented. Private enterprise harnesses or proprietary internal systems could differ in important ways. The framework is intended to generalize beyond the selected systems, but its empirical grounding is necessarily constrained by public evidence.

A third limitation is that the Agent Harness Control Plane decomposition is not unique. Another researcher might split or merge some functions differently. The test of the decomposition is usefulness rather than ontological finality.

Finally, the control plane and the model are not independent in the strongest causal sense. Better models can exploit richer harnesses more effectively, and richer harnesses can mask or amplify model weaknesses. Harness-Native Software Engineering therefore should not be read as a replacement for model analysis. It is a complement that becomes increasingly necessary as agentic coding systems mature.

12. Conclusion

Coding agents have reached a point where the model alone is no longer the most informative unit of analysis for many software-engineering tasks. Recent systems expose explicit layers for context, tools, execution, persistence, verification, recovery, delegation, and governance. Recent benchmark, configuration, security, and adoption work all point toward the same conclusion from different directions: those model-external layers now mediate much of what the agent can do, how safely it can do it, and how credibly its behavior can be measured.

This paper names that regime Harness-Native Software Engineering and formalizes the Agent Harness Control Plane as its central analytical object. The claim is not that models no longer matter, nor that every coding task is harness-dominated. The claim is that for repository-level and long-horizon engineering work, the harness has become too consequential to remain background scaffolding.

If that claim is right, the next phase of coding-agent research should report harness conditions explicitly, ablate them seriously, and treat them as first-class design variables. That would yield a more faithful science of coding agents than model-only comparison tables can provide.

A. System Profiles

A.1 Codex

Codex presents one of the clearest public examples of a harness-native system. Project-scoped guidance is surfaced through `AGENTS.md` and config layers. Actions are mediated through approvals, sandbox modes, network controls, MCP servers, and optional cloud tasks. Execution can happen locally or in configured cloud environments. Codex also documents skills and experimental multi-agent workflows, making delegation an explicit runtime choice rather than an informal prompt pattern [2, 3, 4, 5, 6, 7].

The OpenAI engineering report adds a broader systems perspective. The report repeatedly frames progress in terms of repository legibility, observability, testing, review, and environment design, which fits the Agent Harness Control Plane decomposition closely [1].

A.2 Claude Code

Claude Code emphasizes persistent project knowledge and operational structure. `CLAUDE.md` and auto memory provide context ingress and long-lived state. Hooks, plugins, and MCP govern tool and workflow extension. Subagents expose delegation explicitly. Anthropic’s long-running harness shows how these primitives can be composed into a specialized system for multi-session continuity [11, 12, 8, 9].

Claude Code is a useful example because it makes plain that context and continuity are operational surfaces. The project-specific memory model is not merely personalization. It is part of the system’s control plane.

A.3 GitHub Copilot coding agent

GitHub Copilot’s coding agent is strongly PR-native. The default flow begins from an issue or task, runs in a customizable GitHub Actions environment, and culminates in a pull request. Self-review, security scanning, custom agents, and mission control all surface the harness directly. GitHub’s public writing on multi-agent failures is especially notable because it frames coordination breakdowns as system-structure problems rather than as generic model weakness [13, 14, 15, 16, 17, 18].

This system illustrates the governance-heavy end of the design space. Approval, review, and workflow integration are part of the first-class product experience.

A.4 Gemini CLI

Gemini CLI is terminal-native but unusually explicit about harness features. Skills, checkpoints, hooks, subagents, and sandboxing are all documented. The distinction between workspace context and on-demand skills is analytically useful because it separates persistent background from conditional expertise [19, 20, 21, 22, 23].

Gemini CLI also shows how recovery can be treated as a built-in control-plane capability rather than as an afterthought. Checkpointing is a small but important example of making reversibility part of the runtime.

A.5 Cursor

Cursor illustrates how editor-native systems are converging toward the same runtime patterns. Background agents, memories, Bugbot review, and one-click MCP installation all move the product beyond inline assistance toward a richer harness [24]. The system makes the editor the primary interaction surface, but many of the operative variables still sit in the same control-plane categories described in this paper.

A.6 OpenHands

OpenHands is important because it exposes agent structure without product packaging. Its architecture documentation foregrounds the reasoning loop, state management, workspace abstraction, and separation between core SDK, tools, sandbox, and agent server [25]. This makes it a useful reference case for the open infrastructure end of the coding-agent ecosystem.

A.7 OpenClaw

OpenClaw is analytically useful because it routes across multiple isolated agents and can dispatch into external coding harnesses through ACP sessions [26, 27]. It makes visible a version of the future in which the main “agent” is partly a conductor over other harnesses. That is still compatible with Harness-Native Software Engineering; it simply makes delegation and governance even more central.

B. Harness Condition Sheet Template

Table 5. A practical Harness Condition Sheet template for coding-agent benchmarks and deployment reports.

Field	What to report
Base model(s)	Model name, routing logic, reasoning settings, temperature or deterministic mode, best-of- N if used
Interface mode	CLI, IDE, cloud task, PR-native, hybrid, or orchestrated multi-surface workflow
Context artifacts	AGENTS.md, CLAUDE.md, workspace context files, memories, skills, custom agents, issue / PR state, retrieval corpora
Tool inventory	Available tools, debugger access, browser automation, test runners, package managers, search tools, build tools
Protocol bindings	MCP, ACP, A2A, remote agents, internal APIs, external services
Approval policy	Read-only or write access, tool approval mode, human confirmation points, escalation rules
Internet policy	Off, allowlist, unrestricted, cached search, or environment-specific network profile
Execution substrate	Local machine, container, worktree, CI runner, cloud environment, or mixed mode
Persistence features	Checkpointing, progress files, resumed threads, git checkpoint policy, repository memory
Verification stack	Visible tests, hidden tests, self-review, security scans, browser checks, CI, external API validation
Recovery mechanisms	Retry policy, rollback, checkpoint restore, debug subagents, restart scripts, failure summaries
Delegation topology	Single agent, specialist subagents, conductor pattern, multi-agent parallelism, human handoff
Governance and audit	Audit logs, branch protections, provenance records, admin requirements, repository trust model
Budgets	Token cap, wall-clock limit, retry count, compute budget, evaluation budget

C. Evaluation Metrics

Table 6. Suggested metrics for harness-sensitive evaluation.

Metric	Definition and use
Verified Task Success (VTS)	Fraction of tasks accepted by hidden or independent verification rather than by the agent’s own assertion alone
Recovery Rate (RR)	Fraction of trajectories that later succeed after entering a validated failure state; useful for long-horizon work
Unsafe Action Rate (UAR)	Fraction of runs that attempt policy-disallowed or security-relevant unsafe actions
Human Intervention Burden (HIB)	Approvals, manual corrections, and manual restarts per successful task; captures operational babysitting cost
Continuity Efficiency (CE)	A deployment-sensitive success-per-cost metric, for example $CE = VTS / (\text{tokens} + \lambda \cdot \text{wall-clock})$
Review Friction	Number of review rounds, review latency, or ratio of accepted to revised agent-authored patches
Verification Depth	Count or quality tier of independent checks performed before acceptance

D. Glossary of Adjacent Terms

Table 7. Disambiguation of nearby terms.

Term	Meaning in relation to this paper
Agentic software engineering	Broad field concerned with how agents participate in software engineering; wider than this paper’s focus
AI-native software engineering / SE 3.0	Vision of a larger lifecycle transition toward AI teammates; broader and more strategic than Harness-Native Software Engineering
Context engineering	Design of what enters the model’s context; a strict subset of the control plane analyzed here
Repo-native agency	Agents acting directly on repository state; compatible with Harness-Native Software Engineering but not sufficient to characterize the harness
Harness / scaffold	Model-external orchestration layer; this paper uses <i>harness</i> because it better captures execution, policy, and persistence
Control plane	Persistent layer that determines context, action legality, execution, recovery, and governance
Data plane	Local loop of model inference and immediate tool execution

E. Benchmark Crosswalk

Table 8. Extended benchmark crosswalk.

Benchmark	Horizon	Core realism move	Main blind spots relative to full Agent Harness Control Plane
IDE-Bench	Medium	IDE-native structured tool ecosystem and private repositories	Limited coverage of organization-level governance and long-run memory
Terminal-Bench	Medium	High-skill real CLI tasks	Less emphasis on PR-native review and repository-specific memory
FeatureBench	Medium to long	End-to-end feature implementation with executable environments	Less explicit focus on permission and governance variation
SWE-EVO	Long	Software evolution over many files and iterations	Narrower language and ecosystem coverage
ABC-Bench	Medium	Backend lifecycle including service deployment and API checks	Less direct emphasis on repository context artifacts
RepoMod-Bench	Long	Large-scale repository translation with implementation-agnostic hidden tests	Specialized task family; weaker direct coverage of human review
OmniCode	Short to medium	Broader task-type diversity across languages	Lighter emphasis on long-horizon continuity and recovery
SEC-bench	Medium	Authentic security-engineering tasks in isolated environments	Focuses on security capability, not general workflow adoption
VeRO	Medium	Evaluating agents that optimize agents using versioned snapshots and traces	Meta-agent setting differs from standard developer workflow
SWE-rebench	Variable	Fresh task collection and contamination-aware evaluation	More focused on data pipeline and freshness than on governance surfaces

F. Threat Scenarios

F.1 Poisoned issue text with broad shell access

A repository issue contains an apparently helpful reproduction command that silently exfiltrates local data to an attacker-controlled endpoint. If the agent operates with broad shell authority, open network access, and no approval gate, the attack crosses context ingress, action mediation, execution substrate, and governance all at once. Codex’s internet-access documentation explicitly warns that prompt injection can leak data when network access is enabled [7]. AIShellJack shows that the broader class of malicious-command attacks is not hypothetical [49].

F.2 Compromised MCP server

A tool server advertises capabilities it should not have and returns prompt-injecting content through bidirectional sampling. This is precisely the sort of architectural gap highlighted in the MCP security analysis [48]. The main lesson is that protocol trust is part of the harness, not a side channel outside it.

F.3 Persistent poisoned memory

A context or memory artifact accumulates stale or malicious instructions over time. Because the content is durable and automatically reloaded, the attack survives across sessions and can become harder to distinguish from legitimate project guidance. This is one reason why the quality and provenance of memory artifacts matter as much as their mere existence [43, 46].

F.4 Parallel agents with inconsistent authority

Two specialist agents operate on the same repository with different tool restrictions and overlapping responsibilities. One agent proposes a change that the other silently assumes is already valid; the conductor merges outputs without a shared verification checkpoint. GitHub’s writing on multi-agent failures makes clear that such coordination breakdowns are common in practice [16]. Here the problem is not model reasoning alone but weak delegation and governance structure.

References

- [1] OpenAI. Harness engineering: leveraging Codex in an agent-first world. OpenAI, February 2026. <https://openai.com/index/harness-engineering/>.
- [2] OpenAI Developers. Codex CLI documentation. 2026. <https://developers.openai.com/codex/cli>.
- [3] OpenAI Developers. Custom instructions with AGENTS.md. 2026. <https://developers.openai.com/codex/guides/agents-md>.
- [4] OpenAI Developers. Customization: project guidance, skills, MCP, and multi-agents in Codex. 2026. <https://developers.openai.com/codex/concepts/customization>.
- [5] OpenAI Developers. Multi-agents in Codex. 2026. <https://developers.openai.com/codex/multi-agent>.
- [6] OpenAI Developers. Codex security. 2026. <https://developers.openai.com/codex/security>.
- [7] OpenAI Developers. Agent internet access. 2026. <https://developers.openai.com/codex/cloud/internet-access>.
- [8] Anthropic. Effective harnesses for long-running agents. Anthropic Engineering, November 2025. <https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents>.
- [9] Anthropic. Demystifying evals for AI agents. Anthropic Engineering, January 2026. <https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents>.
- [10] Anthropic. Effective context engineering for AI agents. Anthropic Engineering, September 2025. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>.
- [11] Anthropic. How Claude remembers your project. Claude Code documentation, 2026. <https://code.claude.com/docs/en/memory>.
- [12] Anthropic. Create custom subagents. Claude Code documentation, 2026. <https://code.claude.com/docs/en/sub-agents>.
- [13] GitHub. GitHub Copilot coding agent 101: Getting started with agentic workflows on GitHub. GitHub Blog, September 2025, updated January 2026. <https://github.blog/ai-and-ml/github-copilot/github-copilot-coding-agent-101-getting-started-with-agentic-workflows-on-github/>.
- [14] GitHub. What’s new with GitHub Copilot coding agent. GitHub Blog, March 2026. <https://github.blog/ai-and-ml/github-copilot/whats-new-with-github-copilot-coding-agent/>.
- [15] GitHub. How to orchestrate agents using mission control. GitHub Blog, December 2025. <https://github.blog/ai-and-ml/github-copilot/how-to-orchestrate-agents-using-mission-control/>.

- [16] GitHub. Multi-agent workflows often fail. Here’s how to engineer ones that don’t. GitHub Blog, March 2026. <https://github.blog/ai-and-ml/generative-ai/multi-agent-workflows-often-fail-heres-how-to-engineer-ones-that-dont/>.
- [17] GitHub. Building an agentic memory system for GitHub Copilot. GitHub Blog, January 2026. <https://github.blog/ai-and-ml/github-copilot/building-an-agentic-memory-system-for-github-copilot/>.
- [18] GitHub. Your stack, your rules: introducing custom agents in GitHub Copilot for observability, IaC, and security. GitHub Blog, December 2025. <https://github.blog/news-insights/product-news/your-stack-your-rules-introducing-custom-agents-in-github-copilot-for-observability-iac-and-security/>.
- [19] Gemini CLI Team. Gemini CLI documentation. 2026. <https://geminicli.com/docs/>.
- [20] Gemini CLI Team. Agent Skills. 2026. <https://geminicli.com/docs/cli/skills/>.
- [21] Gemini CLI Team. Checkpointing. 2026. <https://geminicli.com/docs/cli/checkpointing/>.
- [22] Gemini CLI Team. Subagents (experimental). 2026. <https://geminicli.com/docs/core/subagents/>.
- [23] Gemini CLI Team. Sandboxing in the Gemini CLI. 2026. <https://geminicli.com/docs/>.
- [24] Cursor. Bugbot, Background Agent access to everyone, and one-click MCP install. Cursor 1.0 changelog, June 2025. <https://cursor.com/changelog/1-0>.
- [25] OpenHands. Architecture overview: OpenHands Software Agent SDK. OpenHands documentation, 2026. <https://docs.openhands.dev/sdk/arch/overview>.
- [26] OpenClaw. ACP Agents. OpenClaw documentation, 2026. <https://docs.openclaw.ai/tools/acp-agents>.
- [27] OpenClaw. Multi-Agent Routing. OpenClaw documentation, 2026. <https://docs.openclaw.ai/concepts/multi-agent>.
- [28] S. Mateega et al. IDE-Bench: Evaluating Large Language Models as IDE Agents on Real-World Software Engineering Tasks. arXiv preprint arXiv:2601.20886, 2026. <https://arxiv.org/abs/2601.20886>.
- [29] V. Ursekar et al. VeRO: An Evaluation Harness for Agents to Optimize Agents. arXiv preprint arXiv:2602.22480, 2026. <https://arxiv.org/abs/2602.22480>.
- [30] AutoHarness authors. AutoHarness: improving LLM agents by automatically synthesizing a code harness. arXiv preprint arXiv:2603.03329, 2026. <https://arxiv.org/abs/2603.03329>.
- [31] S. Garg and Y. Huang. Debug2Fix: Supercharging Coding Agents with Interactive Debugging Capabilities. arXiv preprint arXiv:2602.18571, 2026. <https://arxiv.org/abs/2602.18571>.
- [32] R. Hutter and M. Pradel. AgentStepper: Interactive Debugging of Software Development Agents. arXiv preprint arXiv:2602.06593, 2026. <https://arxiv.org/abs/2602.06593>.
- [33] Y. Zhu et al. Establishing Best Practices for Building Rigorous Agentic Benchmarks. arXiv preprint arXiv:2507.02825, 2025. <https://arxiv.org/abs/2507.02825>.
- [34] S. Garg, B. Steenhoek, and Y. Huang. Saving SWE-Bench: A Benchmark Mutation Approach for Realistic Agent Evaluation. arXiv preprint arXiv:2510.08996, 2025. <https://arxiv.org/abs/2510.08996>.

- [35] Q. Zhou et al. FeatureBench: Benchmarking Agentic Coding for Complex Feature Development. arXiv preprint arXiv:2602.10975, 2026. <https://arxiv.org/abs/2602.10975>.
- [36] Terminal-Bench Team. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. arXiv preprint arXiv:2601.11868, 2026. <https://arxiv.org/abs/2601.11868>.
- [37] M. V. T. Thai et al. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios. arXiv preprint arXiv:2512.18470, 2025. <https://arxiv.org/abs/2512.18470>.
- [38] I. Badertdinov et al. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. arXiv preprint arXiv:2505.20411, 2025. <https://arxiv.org/abs/2505.20411>.
- [39] J. Yang et al. ABC-Bench: Benchmarking Agentic Backend Coding in Real-World Development. arXiv preprint arXiv:2601.11077, 2026. <https://arxiv.org/abs/2601.11077>.
- [40] X. Li et al. RepoMod-Bench: A Benchmark for Code Repository Modernization via Implementation-Agnostic Testing. arXiv preprint arXiv:2602.22518, 2026. <https://arxiv.org/abs/2602.22518>.
- [41] A. Sonwane et al. OmniCode: A Benchmark for Evaluating Software Engineering Agents. arXiv preprint arXiv:2602.02262, 2026. <https://arxiv.org/abs/2602.02262>.
- [42] M. Galster et al. Configuring Agentic AI Coding Tools: An Exploratory Study. arXiv preprint arXiv:2602.14690, 2026. <https://arxiv.org/abs/2602.14690>.
- [43] W. Chatlatanagulchai et al. Agent READMEs: An Empirical Study of Context Files for Agentic Coding. arXiv preprint arXiv:2511.12884, 2025. <https://arxiv.org/abs/2511.12884>.
- [44] J. L. Lulla et al. On the Impact of AGENTS.md Files on the Efficiency of AI Coding Agents. arXiv preprint arXiv:2601.20404, 2026. <https://arxiv.org/abs/2601.20404>.
- [45] T. Gloaguen et al. Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? arXiv preprint arXiv:2602.11988, 2026. <https://arxiv.org/abs/2602.11988>.
- [46] A. Vasilopoulos. Codified Context: Infrastructure for AI Agents in a Complex Codebase. arXiv preprint arXiv:2602.20478, 2026. <https://arxiv.org/abs/2602.20478>.
- [47] N. Maloyan. Prompt Injection Attacks on Agentic Coding Assistants: A Systematic Analysis of Vulnerabilities in Skills, Tools, and Protocol Ecosystems. arXiv preprint arXiv:2601.17548, 2026. <https://arxiv.org/abs/2601.17548>.
- [48] N. Maloyan and D. Namiot. Breaking the Protocol: Security Analysis of the Model Context Protocol Specification and Prompt Injection Vulnerabilities in Tool-Integrated LLM Agents. arXiv preprint arXiv:2601.17549, 2026. <https://arxiv.org/abs/2601.17549>.
- [49] Y. Liu et al. “Your AI, My Shell”: Demystifying Prompt Injection Attacks on Agentic AI Coding Editors. arXiv preprint arXiv:2509.22040, 2025. <https://arxiv.org/abs/2509.22040>.
- [50] H. Lee, Z. Zhang, H. Lu, and L. Zhang. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. arXiv preprint arXiv:2506.11791, 2025. <https://arxiv.org/abs/2506.11791>.

- [51] H. Li et al. AIDev: Studying AI Coding Agents on GitHub. arXiv preprint arXiv:2602.09185, 2026. <https://arxiv.org/abs/2602.09185>.
- [52] R. Robbes et al. Agentic Much? Adoption of Coding Agents on GitHub. arXiv preprint arXiv:2601.18341, 2026. <https://arxiv.org/abs/2601.18341>.
- [53] S. Agarwal, H. He, and B. Vasilescu. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. arXiv preprint arXiv:2601.13597, 2026. <https://arxiv.org/abs/2601.13597>.
- [54] D. Ogenrwot and J. Businge. How AI Coding Agents Modify Code: A Large-Scale Study of GitHub Pull Requests. arXiv preprint arXiv:2601.17581, 2026. <https://arxiv.org/abs/2601.17581>.
- [55] S. T. Cynthia, J. K. Das, and B. Roy. Are We All Using Agents the Same Way? An Empirical Study of Core and Peripheral Developers Use of Coding Agents. arXiv preprint arXiv:2601.20106, 2026. <https://arxiv.org/abs/2601.20106>.
- [56] C. Nachuma and M. Zibran. When AI Teammates Meet Code Review: Collaboration Signals Shaping the Integration of Agent-Authored Pull Requests. arXiv preprint arXiv:2602.19441, 2026. <https://arxiv.org/abs/2602.19441>.
- [57] A. E. Hassan et al. Towards AI-Native Software Engineering (SE 3.0): A Vision and a Roadmap. arXiv preprint arXiv:2410.06107, 2024. <https://arxiv.org/abs/2410.06107>.
- [58] A. E. Hassan et al. Agentic Software Engineering: Foundational Pillars and a Research Roadmap. arXiv preprint arXiv:2509.06216, 2025. <https://arxiv.org/abs/2509.06216>.