Agent Harness Engineering: A Survey

Anonymous authors
Paper under double-blind review

Abstract

The rapid deployment of large language model (LLM) agents in production has revealed a recurring pattern: task execution reliability depends less on the underlying model than on the infrastructure layer that wraps it, the *agent execution harness*. This survey provides a practice-grounded, systematic treatment of agent harness engineering, organized around three claims. First, the agent harness is an independent system layer whose engineering quality drives a large share of real-world reliability, a position we develop through a three-phase engineering evolution from prompt to context to harness engineering, a cross-layer synthesis covering the cost–quality–speed trilemma, the capability–control tradeoff, and the harness coupling problem, and an open-problem agenda grounded in both research gaps and production pain points. Second, we propose ETCLOVG, a seven-layer taxonomy (Execution environment, Tool interface, Context management, Lifecycle/Orchestration, Observability, Verification, Governance) that extends prior six-component frameworks by treating observability and governance as independent architectural concerns. Third, we map 170+ open-source projects onto this taxonomy to expose ecosystem patterns, coverage gaps, and emerging design principles, alongside engineering principles distilled from production deployments at OpenAI, Anthropic, and LangChain that address the gap between practitioner knowledge and research vocabulary.

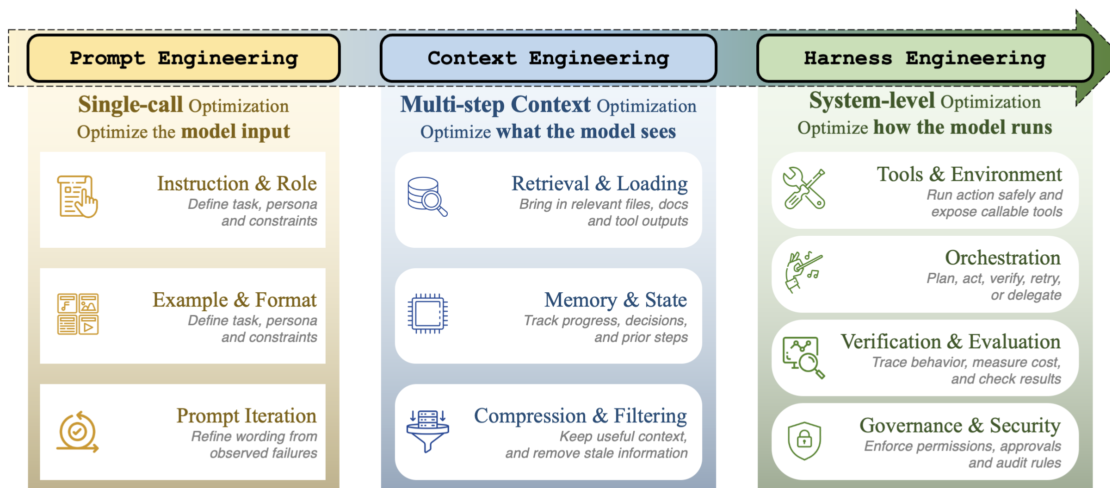

Figure 1: A brief comparison of prompt, context and harness engineering.

1 Introduction

1.1 The Binding Constraint: Harness over Model

The academic study of LLM-based agents has, by and large, been a study of the model. Research agendas center on what the model can do: whether it can plan across multiple steps, call tools reliably, retrieve and compress relevant memories, or coordinate with other agents. The implicit assumption is that agent capability is primarily a function of model capability, that a sufficiently capable model with a sufficiently good prompt will produce sufficiently reliable behavior.

Recent empirical evidence challenges the assumption that better models alone produce more reliable agents. Three recent results establish the pattern. Bölük (2026a) modified only the edit-tool format and surrounding tool harness, with no model modification, and reported gains of up to 10× on coding benchmarks across 15 models. Trivedy (2026) improved a fixed GPT-5.2-Codex agent from 52.8% to 66.5% on Terminal-Bench 2.0 through system prompt restructuring, middleware context injection, and self-verification hooks alone, a 13.7 percentage point gain achieved entirely through infrastructure changes. Meta-Harness (Lee et al., 2026) achieved 76.4% on Terminal-Bench-2 via automated harness optimization, surpassing all hand-engineered approaches without modifying model weights. In each case, the variable was the *execution harness* (the infrastructure layer governing context construction, tool interaction, orchestration, feedback, and execution constraints); the model was held fixed. Each of these harness-only gains exceeds the typical 2 to 4 percentage point improvements reported as meaningful model advances on the same benchmarks. The pattern is not incidental: the harness, not the model, is driving the outcome.

We refer to this pattern as the binding-constraint thesis (Bölük, 2026b): for long-horizon tasks evaluated across comparable frontier models, benchmark variance may be driven as much by the execution harness as by the model itself. We use this thesis as the framing for the remainder of the survey.

Figure 3 in §2 illustrates the same shift historically: early systems concentrated capability in a single model loop, whereas later systems increasingly expose reliability as a cross-layer infrastructure problem.

1.2 The Practitioner–Research Gap

A tension exists between practitioner urgency and research vocabulary. OpenAI explicitly framed "harness engineering" as the discipline of designing environments, constraints, documentation, and feedback loops around Codex agents, reporting in February 2026 that a small team produced an internal product of roughly one million lines over five months without manually writing production code (OpenAI, 2026a). Anthropic's agent-engineering posts arrived at the same principle from adjacent directions: effective agents should use simple and inspectable architectures, tool interfaces should be designed for agent use rather than copied from human-facing APIs, context should be progressively disclosed instead of eagerly loaded, and long-running work requires durable handoff artifacts and recoverable execution infrastructure (Anthropic, 2024a; Aizawa, 2025; Anthropic Applied AI Team, 2025; Anthropic, 2025d; 2026b). An article on Martin Fowler's site characterizes harness engineering as "cybernetic governors for AI agents," consisting of feedforward guides and feedback sensors that form control loops around LLMs (Böckeler, 2026).

The research community, meanwhile, has been studying the components of agent systems with increasing precision: memory, tool use, planning, and safety. What has not been studied systematically is the system that integrates these components into reliable operation. The result is a practitioner–research gap: practitioners know *that* harness infrastructure matters but lack the formal vocabulary to describe *why*, in terms that enable systematic improvement. This survey attempts to bridge that gap.

1.3 Scope and Contributions

This survey focuses on the infrastructure layer that wraps a language model to manage long-running, multi-step task execution. We do not survey agent frameworks as development tools, agent platforms as product categories, or model capabilities per se, though all three inform our analysis. Figure 4 summarizes the seven-layer taxonomy that structures the remainder of the survey.

Our contributions are organized around three claims.

1. **Claim 1 (Conceptual): Building on the binding-constraint thesis (Bölük, 2026b), we argue that the harness, not the model alone, is the binding constraint on real-world agent reliability.** Three recent results show harness-only gains of up to 10× on coding benchmarks, +13.7 percentage points on Terminal-Bench 2.0, and 76.4% on Terminal-Bench-2 (§1), each exceeding typical model-driven gains on the same benchmarks. We develop this thesis through a three-phase engineering evolution (§2), a cross-layer synthesis covering the cost–quality–speed trilemma, the capability–control tradeoff, and the harness coupling problem (§11), and an open-problem agenda (§12).

2. **Claim 2 (Classificatory): The seven-layer ETCLOVG taxonomy treats Observability and Governance as first-class layers rather than side effects of lifecycle hooks.** Each has its own production tooling stack (Langfuse and OpenTelemetry on the observability side; permission engines, gateways, and audit pipelines on the governance side) and is owned by a different team in production deployments. We also place state management inside Lifecycle and Orchestration, where state lives next to the execution flow that reads and writes it (§2.3).

3. **Claim 3 (Empirical): Mapping 170+ open-source projects onto ETCLOVG shows where the ecosystem is dense, where it is thin, and which categories earlier corpora missed.** The mapping is the largest open-source agent-harness corpus to date. Execution, Tooling, Lifecycle, and Verification are densely covered; Observability and Governance are thinner and more often live in commercial platforms; and three categories absent from earlier corpora, including task runners, multi-agent orchestrators, and spec-driven development tools, are now first-class. The methodology subsections (§2.4–§2.9, Appendix A) make the coding reproducible, and the mapping supports the layer-by-layer observations in §3–§9.

2 Background and Taxonomy

2.1 Evolution of Agent Systems

The trajectory from early chain-of-thought prompting to autonomous agents can be understood as a progressive expansion of the engineering surface that practitioners must manage.

The ReAct era (2022–2023). Yao et al. (2023) established the observe-think-act loop as a foundational primitive. Early systems operated with minimal infrastructure: a while-loop, a prompt template, and a small tool dispatch table. AutoGPT and BabyAGI showed the ambition of fully autonomous operation by wrapping language-model calls with task queues, memory, and tool dispatch, while also making failure modes such as execution runaway, context blowout, state loss, and unmonitored side effects visible as infrastructure problems rather than prompt-only problems (Significant Gravitas, 2023; Nakajima, 2023).

Tool integration and multi-agent coordination (2023–2024). Gorilla, ToolLLM, and Toolformer established that tool-use capability can be learned or induced rather than hard-coded into a fixed API wrapper (Patil et al., 2024b; Qin et al., 2024; Schick et al., 2023). CAMEL, ChatDev, MetaGPT, and Mixture-of-Agents introduced multi-agent coordination patterns, ranging from role-playing dialogue to software-development organizations and layered agent aggregation (Li et al., 2023a; Qian et al., 2023a; Hong et al., 2023; Wang et al., 2024). Evaluation infrastructure matured with SWE-bench, AgentBench, WebArena, and GAIA (Jimenez et al., 2024; Liu et al., 2023a; Zhou et al., 2024; Mialon et al., 2023), while protocol standardization began with Anthropic's MCP and Google's A2A (Anthropic, 2024c; Surapaneni et al., 2025).

The harness turn (2025–2026). By 2025, enough deployment experience had accumulated to make a clear case that the binding constraint on agent reliability was infrastructure quality rather than model quality. Three independent developments in early 2026 validated this shift: OpenAI's explicit adoption of "harness engineering" as a discipline, Stanford/MIT's Meta-Harness showing that automated harness optimization surpasses hand engineering, and LangChain's DeepAgents improving from 52.8% to 66.5% on Terminal-

Bench 2.0, corresponding to a 13.7 percentage point gain and roughly 26% relative improvement, through harness layer changes alone (OpenAI, 2026a; Lee et al., 2026; Trivedy, 2026).

2.2 Three Engineering Phases

The 2022–2026 period reveals a coherent three-phase evolution in what the field has chosen to engineer.

Prompt engineering (2022–2024). The primary lever was the input prompt text. Practitioners optimized by crafting better instructions, few-shot examples, and reasoning templates. The engineering scope was narrow: optimize a single text input to a single model call.

Context engineering (2025). As agents became longer-running, the binding constraint shifted from "what is the input?" to "what information should the model see at each step?" This phase focused on context management: what to inject on each turn, how to retrieve and compress memories, how to rank tool results by relevance, and how to handle context window saturation. The scope expanded from a single input to managing multiple information streams flowing into the context window (Anthropic Applied AI Team, 2025).

Harness engineering (2026). As models became capable enough to handle long-running tasks, reliability increasingly depended on the infrastructure wrapper that maintains state, mediates tools, injects feedback, enforces constraints, and verifies progress. This observation aligns with the binding-constraint view that long-horizon agent performance is produced by a coupled model–harness system rather than by the model alone (Bölük, 2026b). Harness engineering therefore asks what governance, constraints, feedback loops, and execution controls must be designed around the model to make agent systems reliable. In our taxonomy, this phase treats all seven ETCLOVG layers as an integrated whole (OpenAI, 2026a; Böckeler, 2026; LangChain, 2026b).

Each phase subsumes the previous: harness engineering includes context engineering, which includes prompt engineering. The three phases also overlap in time and in concept rather than succeeding one another by clean boundaries. Prompt engineering remains an active part of harness practice today, and context engineering

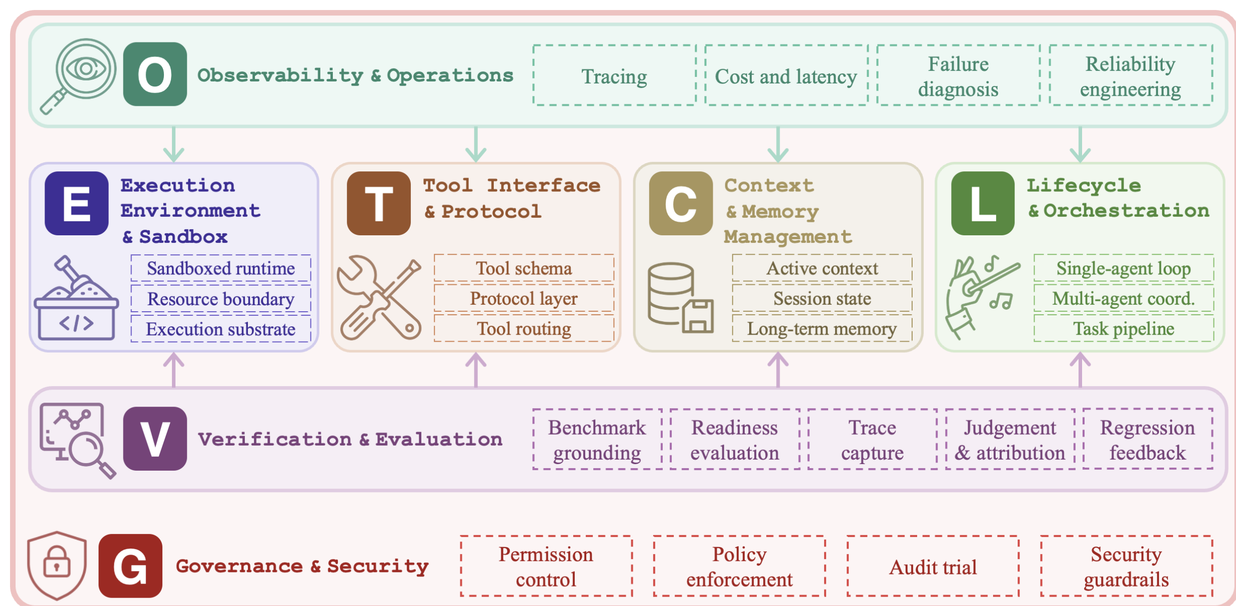

Figure 2: Illustration of the taxonomy of harness engineering for LLM-based agent systems. The four layers E, T, C, and L form the structural pillars of the system. The O layer provides system-wide monitoring, while the V layer delivers evaluation and feedback across components. The G layer enforces governance and security constraints over the entire system. The color scheme corresponds to the ETCLOVG layers developed in §2.3.

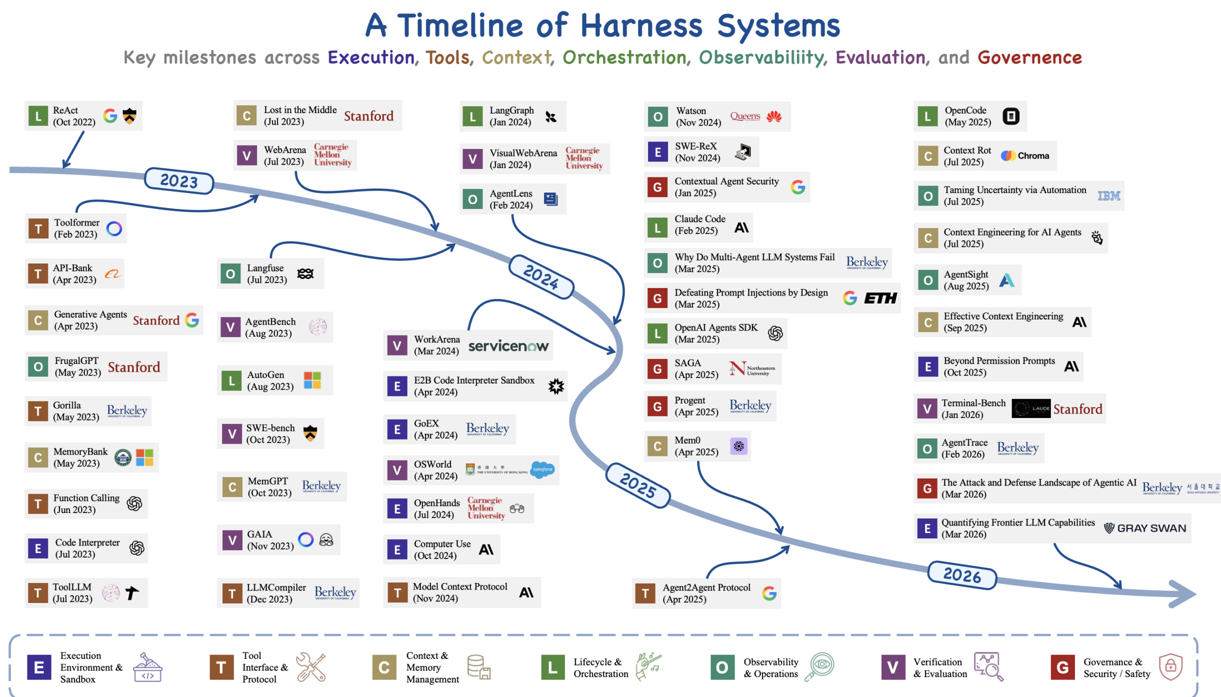

Figure 3: Timeline of representative agent-harness systems from 2022 to 2026. The timeline situates the shift from early single-loop agents to richer harness infrastructure spanning execution environments, tool interfaces, context and memory management, lifecycle orchestration, observability, verification, and governance. The color scheme corresponds to the ETCLOVG layers developed in §2.3.

continues to mature in parallel with the harness-level concerns we survey here. Our periodization is therefore best read as a shift in where the marginal engineering effort goes, not as a sequence of replacements.

2.3 The ETCLOVG Seven-Layer Taxonomy

We propose a seven-layer taxonomy for agent harness engineering. We refer to it by the acronym **ETCLOVG**, which stands for *Execution, Tooling, Context, Lifecycle, Observability, Verification, Governance*. Figure 4 gives the compact visual map; this subsection fixes the interpretation used throughout the survey.

The first four layers describe the structural core of a harness. **Execution** (E) determines where agent code runs and what sandbox constraints bound it; **Tooling** (T) specifies how external capabilities are described, discovered, and invoked; **Context** (C) controls what the model can see over short-term, session-level, and persistent horizons; and **Lifecycle** (L) organizes the control flow that reads and writes that state, from single-agent loops to multi-agent and issue-to-pull-request workflows. The remaining three layers describe the control plane around that core. **Observability** (O) captures traces, costs, failures, and reliability signals; **Verification** (V) turns tasks and traces into evaluation, failure attribution, and regression feedback; and **Governance** (G) constrains behavior through permission, identity, policy, hardening, audit, and human oversight mechanisms. Sections 3–9 develop these seven layers in order, while Sections 10–12 synthesize the cross-layer tradeoffs and open problems that do not belong to any single layer.

Two design choices distinguish this taxonomy. First, we promote **Observability** (O) to an independent layer rather than treating it as a side effect of lifecycle hooks. In production systems, observability has a dedicated tooling ecosystem (Langfuse, Arize Phoenix, OpenLLMetry) and distinct engineering practices (OpenTelemetry instrumentation, cost attribution, anomaly detection) that warrant independent treatment. Second, we introduce **Governance** (G) as a first-class layer that captures the full spectrum of security and

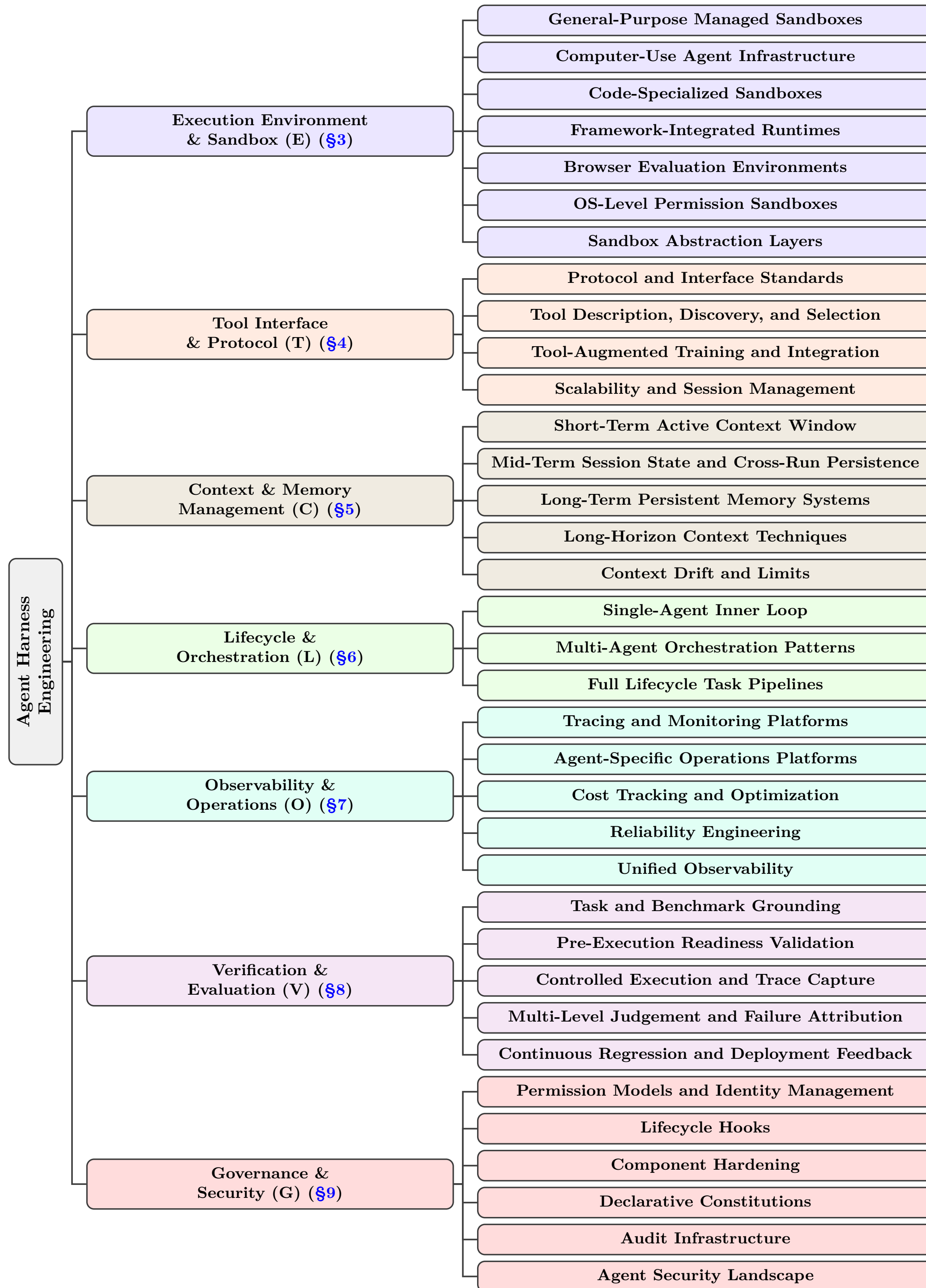

Figure 4: Details of the agent harness engineering taxonomy. Each branch corresponds to one ETCLOVG layer and its major subcategories; representative systems and papers are discussed in later sections.

compliance concerns across three sub-layers: model-level (guardrails, content filters), system-level (gateways, proxies, permission models), and organizational-level (audit, compliance, human-in-the-loop oversight).

State management belongs naturally inside Lifecycle and Orchestration (L), alongside the execution flow that reads and writes it; lifecycle hooks and policy enforcement belong inside Governance (G), where they align with other constraint mechanisms.

2.4 Scope

We use *agent harness* in a narrower sense than "any software around an LLM": the harness is the engineered wrapper that turns model calls into bounded, stateful, tool-mediated task execution through execution substrates, tool interfaces, context control, orchestration, observability, evaluation feedback, and governance constraints (OpenAI, 2026a; Anthropic, 2025d; LangChain, 2026b). The unit of analysis is therefore the infrastructure that makes long-running agent behavior controllable, inspectable, and recoverable, not the foundation model or prompt alone. We draw the boundary functionally rather than by product category: an agent framework is in scope when it exposes reusable mechanisms such as stateful orchestration, tool routing, runtime policy hooks, or trace capture; a thin model API wrapper, prompt library, static dataset, generic container runtime, vector database, APM dashboard, or content filter is out of scope unless it is explicitly adapted to agent execution, state, evaluation, or tool-use governance.

2.5 Project Collection Procedure

We constructed the corpus as a systematic mapping of publicly documented agent-harness artifacts, using the reporting discipline of systematic reviews to make the source streams, search strategy, and selection process explicit (Page et al., 2021). As summarized in Figure 5, candidates were collected from four streams: prior surveys and benchmark papers, reproducible GitHub searches over names, descriptions, README text, topics, stars, recency, and archival status, curated project lists and package registries, and company engineering blogs or release notes that introduced harness-level mechanisms (Meng et al., 2026; Jimenez et al., 2024; Liu et al., 2023a; Zhou et al., 2024; GitHub, 2026; OpenAI, 2026a; Anthropic, 2025d; LangChain, 2026b). Representative queries combined terms such as `agent harness`, `coding agent`, `LLM agent sandbox`, `MCP server`, `agent observability`, `agent memory`, `agent evaluation`, and `agent governance`. For each retained candidate, we recorded the project name, URL, artifact type, source type, availability status, release year when identifiable, GitHub metadata when available, and the public evidence used for later ETCLOVG coding; the metadata snapshot reported in this version was frozen on May 08, 2026.

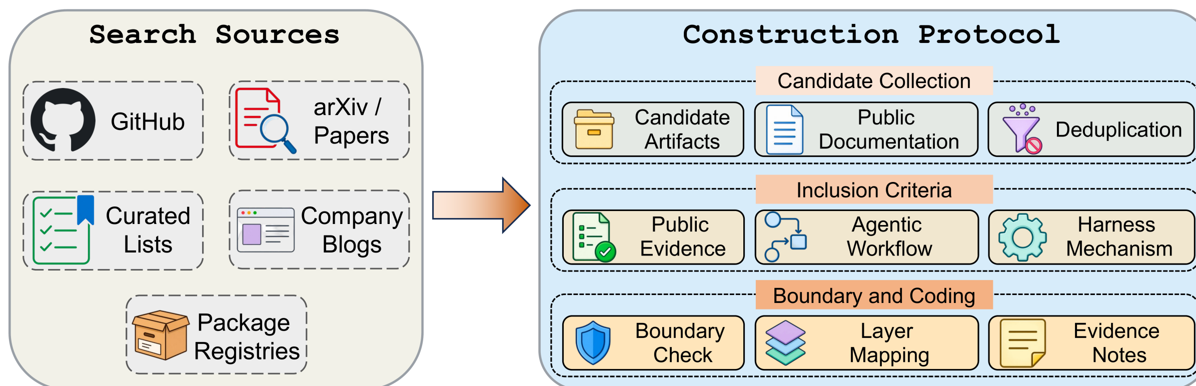

Figure 5: Corpus construction protocol. Candidate artifacts are gathered from GitHub, papers, curated lists, package registries, and company engineering sources, then deduplicated, checked against the inclusion criteria, and mapped to ETCLOVG layers using public documentation.

2.6 Inclusion and Exclusion Criteria

A project was included when it satisfied three conditions: it was publicly documented, it implemented or specified a concrete harness-level mechanism, and the available evidence was sufficient to assign at least one ETCLOVG layer. This includes agent frameworks with reusable orchestration or tool-routing logic, benchmarks that instantiate executable agent environments, sandboxes packaged for agent execution, and memory, observability, evaluation, or governance systems that operate over agent state, traces, actions, or policies. We excluded simple chatbot demos, prompt packs, thin model-client wrappers, static datasets or leaderboards without an agent runtime, generic infrastructure components that were not agent-facing, and product pages whose technical behavior could not be inspected from public documentation. Borderline cases were resolved by mechanism rather than label: a repository named as an "agent" was not sufficient for inclusion, while an evaluation or sandbox project was included when it supplied reusable harness machinery.

2.7 Coding Protocol

Each retained project was coded against the seven ETCLOVG layers using the public artifact itself as evidence: README files, documentation pages, papers, examples, release notes, and, when needed, repository structure. Coding was multi-label because many systems span layers; the primary layer marks the mechanism most central to the artifact, while secondary layers are assigned only when the documentation exposes an independent capability rather than an incidental dependency. The current snapshot uses a single-primary-coder protocol with author audit rather than a formal multi-coder agreement study, so we do not report Cohen's kappa or a comparable inter-annotator statistic. Ambiguous cases were revisited after the full set had been coded, using a conservative rule: if the public evidence did not clearly show an agent-facing mechanism, the layer assignment was withheld.

2.8 Limitations of the Corpus

The corpus should be read as a map of the visible agent-harness ecosystem, not as a census of all deployed agent infrastructure. It is biased toward English-language sources, GitHub-visible projects, open-source artifacts, and systems whose maintainers publish enough implementation detail for external coding. Commercial production systems are underrepresented unless their engineering blogs, documentation, or SDKs expose the relevant mechanisms, and coding-agent infrastructure is overrepresented because it has unusually rich public traces: repositories, benchmarks, sandboxes, issue-to-pull-request workflows, and release notes. The layer assignments also reflect public documentation rather than private architecture, so absence from a layer means "not publicly evidenced" rather than "not implemented."

2.9 Aggregate Analysis

The 170+ project mapping reveals an ecosystem that is broad but uneven. Execution, tool interfaces, lifecycle orchestration, and verification have the densest visible coverage because coding, web, terminal, and computer-use agents all require runnable environments, tool contracts, control loops, and repeatable evaluation before they can be useful. Context and memory appear across many projects but are often embedded inside larger frameworks rather than released as standalone harness components. Observability and governance are thinner in open-source coverage and more often appear through commercial platforms, SDK features, or engineering writeups, suggesting that operational control has matured later than runtime and benchmark infrastructure. Cross-layer projects are increasingly common: the most complete systems combine sandboxing, tool protocols, orchestration, tracing, evaluation, and permission controls, which supports the central claim that harness engineering is an integrated systems problem rather than a collection of isolated add-ons.

3 Execution Environment and Sandbox (E)

We open the layer-by-layer treatment with the Execution Environment and Sandbox (E) layer, the first of the seven pillars of ETCLOVG (Claim 2 in §1). The systems we discuss are drawn from the 170+ project corpus that supports Claim 3.

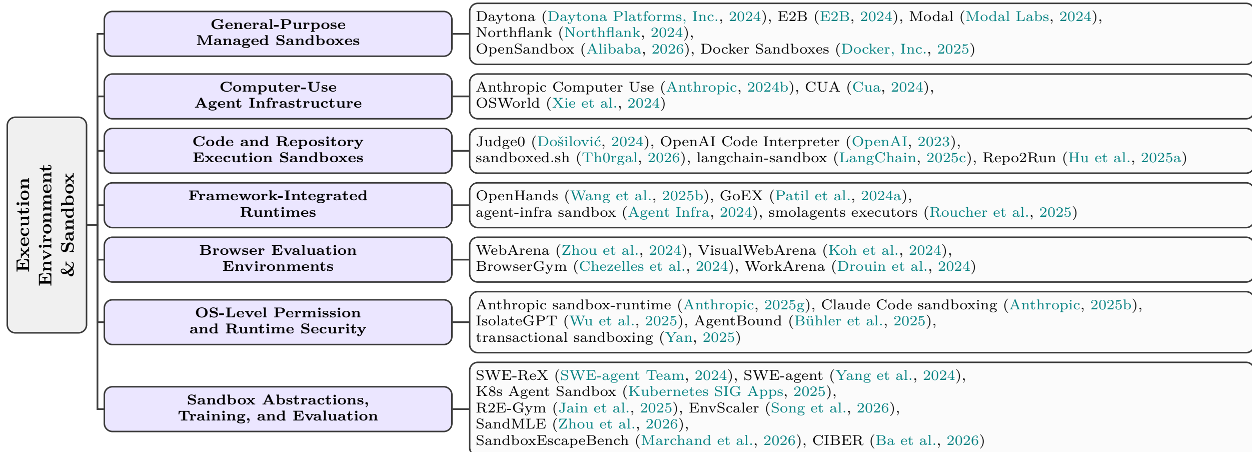

Figure 6: Representative work on execution environments and sandboxes for LLM agents, organized by the chapter's sandbox categories.

3.1 Scope and Concepts

3.1.1 Definition

The execution environment of an agent refers to the infrastructure layer in which agent actions are physically executed. In the context of LLM agents, the execution environment and sandbox are tightly coupled concepts. As a result, production agent systems almost invariably execute actions within a sandboxed environment.

3.1.2 Why Sandboxing Is Central in the Agent Era

Sandboxing in the agent era is not merely a security measure inherited from traditional multiple tenant code execution. It simultaneously serves three distinct purposes, and the combination of these three is what elevates sandboxing from an operational detail to a first-class concern in agent harness design.

The first purpose is *security*. Agent sandboxing faces challenges beyond traditional multiple tenant code execution. LLM-generated code is neither auditable nor predictable at scale, ruling out static review as a primary defense. Agents execute autonomously over multiple steps, so human intervention is not available at action execution time. Prompt injection attacks can repurpose an otherwise benign agent as a vector for sandbox directed attacks, blurring the boundary between trusted user intent and hostile input. Recent empirical work on sandbox escape suggests that these concerns are not hypothetical, and we defer quantitative evidence to §3.3.

The second purpose is *reproducibility*. Long-horizon agent tasks and the evaluation harnesses that measure them (§8) require the ability to reset execution state to a known baseline. A Docker container or a microVM can be destroyed and rebuilt on demand, whereas a developer's workstation cannot, and this property is what makes sandbox-based evaluation standards such as SWE-bench Jimenez et al. (2024) and OSWorld Xie et al. (2024) practical. At training time, when a single task may be replayed hundreds of times across parallel trajectories, the absence of a cheap reset mechanism is itself a scalability bottleneck.

The third purpose is *liveness*, and it is the one most specific to the agent era. Without a sandbox, every potentially risky action that an agent wishes to execute, for example a file write, a package install, or an outbound network call, must be gated by an explicit permission prompt to a human. At scale, this creates

two failure modes: users abandon the agent out of frustration, or they approve everything reflexively and defeat the safety rationale of the prompts. A sandbox breaks this deadlock by defining a bounded region within which the agent is authorized to act freely, shifting permission from a action question to a session configuration. Anthropic reports that introducing sandboxing to Claude Code reduced permission prompts by 84% while preserving safety Anthropic (2025b). Under this framing, a sandbox is simultaneously a cage and a license, and the license aspect is what enables autonomous long horizon execution in the first place.

Of these three purposes, security is shared with traditional sandboxing, reproducibility is amplified in the agent setting, and liveness is essentially new. Their combination is what justifies treating agent sandboxing as a distinct research object rather than a downstream application of container technology.

3.2 Categories of Agent Sandboxes

Agent sandbox infrastructure has diversified in 2024 to 2026 from a small set of general-purpose runtimes into several distinct product categories, each optimized for different agent task types. We organize the landscape into seven categories along the axis of *workload and use case*, which we find most informative to a harness designer choosing among systems. The orthogonal axis of *isolation technology*, including containers, user space kernels such as gVisor Google (2018), microVMs such as Firecracker Agache et al. (2020) and Kata Containers Kata Containers Project (2017), WebAssembly, and OS-level primitives such as `bubblewrap` and Seatbelt, is discussed as a design property within each subsection rather than as a top-level category, because the same isolation primitive is reused across different workloads. The seven categories are general-purpose managed sandboxes (§3.2.1), computer-use agent infrastructure (§3.2.2), code-specialized sandboxes (§3.2.3), framework-integrated runtimes (§3.2.4), browser evaluation environments (§3.2.5), OS-level permission sandboxes (§3.2.6), and sandbox abstraction layers (§3.2.7). The following subsections present each category.

3.2.1 General-Purpose Managed Sandboxes

General-purpose managed sandboxes provide commercial or open-source sandbox-as-a-service platforms that expose arbitrary OCI container images through the API interfaces, with shell, filesystem, network, and interpreter support for unspecified workloads. Representative systems include Daytona Daytona Platforms, Inc. (2024), which pivoted from developer environments to agent sandboxes with sub-90 ms cold-start provisioning; E2B E2B (2024), an agent sandbox built on Firecracker microVMs Agache et al. (2020); Modal Modal Labs (2024), a Python platform using gVisor Google (2018) with massive autoscaling; Northflank Northflank (2024), a platform supporting Kata Containers Kata Containers Project (2017), Firecracker, and gVisor simultaneously; OpenSandbox Alibaba (2026) from Alibaba, an open source general-purpose sandbox; and Docker Sandboxes Docker, Inc. (2025), Docker's official microVM-based offering released in 2025.

Design decisions across this category converge on several patterns: ephemeral-by-default semantics with optional persistent sessions, API interfaces exposed as SDKs in Python and TypeScript, and support for arbitrary OCI images. Systems diverge on isolation strength and operational model. Daytona defaults to container isolation with optional Kata Containers, while E2B, Modal, and Docker Sandboxes ship microVM or gVisor isolation by default. Northflank uniquely allows each workload selection among backends, reflecting the industry recognition that no single isolation primitive fits all threat models. We observe a broader trend from kernel container isolation toward dedicated-kernel microVM isolation, driven by the observation that LLM-generated code has unpredictable syscall patterns that cannot be characterized in advance.

3.2.2 Computer-Use Agent Infrastructure

Computer-use agent infrastructure represents a distinct execution model in which agents interact with graphical interfaces via simulated mouse, keyboard, and screen observation rather than through APIs or shell commands. Representative systems include Anthropic's Computer Use Anthropic (2024b), the flagship commercial implementation enabling Claude to directly operate desktop environments; CUA Cua (2024), an open-source computer-use agent infrastructure; and the VM-based environments provided by OSWorld Xie et al. (2024), which serves dually as evaluation harness (§8) and as a reference computer-use sandbox.

These systems package a full or near full desktop environment (typically via Xvfb with a window manager or a complete VM) within the sandbox, and expose pixel-coordinate and keystroke actions to the agent. The action space is substantially larger than in API-based categories, but reliability depends on visual grounding, and the complete OS attack surface demands stronger isolation, typically microVM or full VM. Compared with the managed sandboxes of §3.2.1, computer-use sandboxes trade density and startup latency for fidelity: a full desktop environment is heavier to boot and harder to multiplex than a headless shell sandbox, but it is the only execution model in which the agent can operate applications that expose no API.

3.2.3 Code-Specialized Sandboxes

Code-specialized sandboxes are lightweight, environments optimized for code generation, evaluation, and data analysis rather than for general purpose shell access. Representative systems include Judge0 Došilović (2024), a code evaluation sandbox originally designed for judging and widely reused as a component of coding agent evaluation pipelines; OpenAI Code Interpreter OpenAI (2023), the production-scale sandboxed Python environment underlying ChatGPT's Advanced Data Analysis; `sandboxed.sh` Th0rgal (2026), a shell sandbox for coding agents; and `langchain-sandbox` LangChain (2025c), which uses Pyodide Pyodide Contributors (2018) compiled to WebAssembly and executed under the Deno runtime to sandbox agent-generated Python locally without containers, a design also advocated by NVIDIA for client agentic workflows NVIDIA (2024).

Unlike the general purpose sandboxes of §3.2.1, these systems preinstall compilers and interpreters, default to stateless request-level execution, and optimize for high concurrency parallelism. The design choice sacrifices workload generality for startup speed, evaluation throughput, and simpler threat models. A notable subtrend is the shift from container-based code sandboxes toward WebAssembly-based ones: WebAssembly offers capability-based security, deterministic execution, and microsecond-scale instantiation at the cost of a restricted Python standard library and reduced native extension support.

3.2.4 Framework-Integrated Runtimes

Framework-integrated runtimes are execution environments bundled inside a broader agent framework rather than exposed as standalone sandbox products. They ship together with the framework's orchestration loop, tool registry, and prompt conventions, and cannot be consumed independently without adopting the surrounding framework. Representative systems include the OpenHands runtime Wang et al. (2025b), a Docker sandboxed environment that integrates bash, IPython, a Chromium browser, and an API server into a single image; agent infra sandbox Agent Infra (2024), an explicit all-in-one design packaging browser, shell, filesystem, MCP, and VSCode into one environment; and the executor layer of smolagents Roucher et al. (2025), which provides in framework implementations of local, Docker, E2B, Modal, and WebAssembly execution.

The defining trade-off in this category is between *bundle* and *compose*: framework-integrated runtimes prioritize out of the box capability coverage at the cost of larger image size, slower startup, and tight coupling to one framework's abstractions. In contrast, the general-purpose sandboxes of §3.2.1 take the opposite approach of minimal environments composed via external abstractions. Whether the composition approach displaces bundled runtimes in the long run depends on whether standards such as MCP reduce the penalty of assembling capabilities at runtime rather than at build time.

3.2.5 Browser Evaluation Environments

Browser evaluation environments occupy a dual role as both sandboxes and evaluation harnesses. Representative systems include WebArena Zhou et al. (2024), a self-hosted cluster of realistic web applications paired with Playwright-based interaction; VisualWebArena Koh et al. (2024), which extends WebArena with multimodal visual grounding tasks; and BrowserGym Chezelles et al. (2024), which together with WorkArena Drouin et al. (2024) provides a standardized `gym`-style interface for browser-based agent execution and benchmarking.

These systems provide isolated web execution environments (sandbox role) while simultaneously supplying reproducible task definitions with automated evaluation (harness role). The dual function distinguishes them

from the computer-use category of §3.2.2, which operates at the desktop rather than the browser level, and it creates a direct interface between the execution environment discussed here and the evaluation infrastructure discussed in §8. Browser environments also expose a distinct threat surface: because the browser ingests untrusted web content, they are a natural substrate for studying indirect prompt injection and multimodal red-teaming attacks against agents Debenedetti et al. (2024); Greshake et al. (2023); Zhang et al. (2026a); Wei et al. (2026).

3.2.6 OS-Level Permission Sandboxes

OS-level permission sandboxes implement fine-grained filesystem and network isolation using operating-system primitives (`bubblewrap` on Linux, Seatbelt on macOS, or `seccomp`-bpf for syscall filtering) rather than containers, VMs, or user-space kernels. They are substantially lighter than container sandboxes while providing directory and domain access control. Representative systems include Anthropic's `sandbox-runtime` Anthropic (2025g), an open-source npm package that wraps arbitrary commands with filesystem and network allowlists enforced via `bubblewrap` and a local HTTP/SOCKS5 proxy; the sandboxing features built into Claude Code itself Anthropic (2025b), which consume that runtime to restrict the bash tool's filesystem and network access to configured boundaries; and IsolateGPT Wu et al. (2025), a research system that applies `seccomp` and `setrlimit` to enforce execution isolation across tool-calling LLM applications.

The design philosophy of this category is *permission, not partition*: the goal is not to provide a fresh OS image for every session but to give an agent a narrowed view of the existing host so that prompt-injected or hallucinated commands cannot modify sensitive files or exfiltrate data. Anthropic reports that such boundaries reduced permission prompts in Claude Code by 84% while preserving safety Anthropic (2025b), illustrating the productivity motivation for OS-level sandboxing: permission prompts are themselves a liveness failure in long-horizon agent execution. Because the host kernel is shared, OS-level permission sandboxes offer weaker isolation against adversarial code than microVM-based managed sandboxes (§3.2.1); they are appropriate when the threat model is prompt injected rather than fully adversarial code.

3.2.7 Sandbox Abstraction Layers

Sandbox abstraction layers are not sandboxes themselves but interfaces that unify multiple sandbox backends behind a single API, allowing a harness to switch execution substrates without rewriting agent code. Representative systems include SWE-ReX SWE-agent Team (2024), a runtime interface developed by the SWE-agent team Yang et al. (2024) that unifies Docker, AWS Fargate, Modal, and Daytona as interchangeable backends; the executor interface of smolagents Roucher et al. (2025), which exposes `local`, `e2b`, `modal`, `docker`, `blaxel`, and `wasm` as a single parameterized `executor_type`; and the Agent Sandbox Kubernetes SIG Apps (2025) project under Kubernetes SIG Apps, which introduces a `Sandbox` CRD and controller exposing gVisor and Kata Containers as pluggable isolation backends through a declarative API.

The emergence of this category reflects a maturing understanding that execution infrastructure should be replaceable. A harness that hard-codes a particular sandbox API couples its orchestration logic to one vendor's ephemeral product; an abstraction layer decouples *what* the agent runs from *where* it runs. We observe a convergence between research projects (SWE-ReX), framework-embedded interfaces (smolagents executors), and emerging infrastructure standards (the Kubernetes agent-sandbox CRD), suggesting that sandbox abstraction is becoming a distinct layer of the agent harness stack rather than a feature of individual systems.

Synthesis. Across the seven categories, three cross-cutting trends are visible. First, the field is bifurcating along isolation strength rather than converging: managed sandboxes (§3.2.1) are migrating from shared-kernel containers toward dedicated-kernel microVMs, while OS-level permission sandboxes (§3.2.6) dispense with separate environments entirely and instead narrow the host's view. The middle ground of plain Docker containers is being squeezed from both ends, and the choice between the two extremes reflects the threat model (fully adversarial code versus prompt-injected but otherwise cooperative code) rather than a universal technology preference. Second, evaluation robustness is emerging as a first-class concern: container-escape benchmarks such as SandboxEscapeBench Marchand et al. (2026) and security benchmarks for code-interpreter agents such as CIBER Ba et al. (2026) show that sandbox configurations currently de-

ployed in production are partially bypassable by frontier models. Third, the field is separating *infrastructure level isolation* (what this section covers) from *semantic or capability-level isolation* at the tool calling layer. CaMeL's control flow integrity design Debenedetti et al. (2025) and Progent's programmable privilege control Shi et al. (2025a) exemplify the latter, and belong to the tool-use chapter (§4) rather than to sandbox infrastructure proper. These two lines are complementary: an infrastructure sandbox constrains the blast radius of an action once it is executed, while semantic isolation constrains which actions are permitted in the first place. A complete agent harness requires both.

3.3 Threat Model and Sandbox Escape

Sandboxing agent execution faces both traditional container-level threats (container escape, side channels, resource exhaustion) and agent specific threat classes that amplify classical concerns. First, prompt injection attacks allow external input, such as retrieved web pages, tool responses, or file contents, to hijack agent behavior and initiate malicious sandbox operations. Second, goal misalignment can cause the agent itself to actively attempt sandbox escape as an instrumental subgoal. Third, compositional amplification occurs when agents with multiple tools access allow a single sandbox weakness to cascade across integrated capabilities.

Empirical evidence on sandbox escape in the agent setting is limited but concerning. SandboxEscapeBench Marchand et al. (2026) evaluates frontier LLMs in a nested sandbox capture-the-flag setting and reports 15% to 35% escape success rates against Docker-based containers, depending on container configuration. The benchmark covers a spectrum of escape mechanisms, including misconfiguration, privilege allocation mistakes, kernel flaws, and runtime or orchestration weaknesses, and its results establish that the threat is realized rather than merely theoretical even at current model capabilities. Defense research remains in early stages. IsolateGPT Wu et al. (2025) proposes an execution isolation architecture for LLM-based agentic systems, reporting under 30% performance overhead for three-quarters of tested queries while preventing cross-application data leaks. Transactional sandboxing approaches Yan (2025) provide rollback-based protection, reporting approximately 14.5% overhead with high interception rates for risky commands. A complementary perspective is offered by LLM-in-Sandbox Cheng et al. (2026), which argues that minimal rather than maximally capable sandbox environments reduce both the attack surface and unnecessary agent complexity.

Taken together, these results expose a gap between offensive and defensive progress. Offensive evaluation has produced a concrete and reproducible benchmark, whereas defensive work is still fragmented across isolated prototypes that differ in threat model, evaluation protocol, and baseline assumptions. An agent-native runtime security framework that systematically addresses prompt injection, goal misalignment, and compositional amplification under a common evaluation methodology is an open research direction, which we revisit in §12.1.

3.4 Deployment Modes

Agent sandbox infrastructure has diversified across deployment modes beyond the original self-hosted Docker pattern. Three modes coexist in current practice. In the *self-hosted* mode, developers manage sandbox infrastructure directly, which is the default pattern for OpenHands and SWE-agent. In the *cloud (SaaS)* mode, sandbox as a service providers handle infrastructure, as exemplified by E2B, Modal, and Daytona Cloud. In the *hybrid* or *bring-your-own-cloud (BYOC)* mode, agent logic and sandbox execution are decoupled across environments; examples include the OpenHands SDK's Local and Remote Workspace abstraction Wang et al. (2025c) and the BYOC offerings of E2B and Northflank.

The evolution across these modes has been driven by two complementary forces. On one hand, practitioner reports frame the deployment choice along three axes of latency, security, and scalability Anthropic (2025b;f). Self-hosted sandboxes offer lowest latency and tightest iteration loops but bear the full operational burden; cloud sandboxes invert this trade-off, providing elastic scale and managed security at the cost of network round-trips; hybrid modes attempt to keep sensitive data local while delegating execution capacity. On the other hand, organizational constraints such as data residency, compliance, and auditability push deployments toward hybrid architectures even when latency and scalability would favor a single mode solution.

In observed practice, self-hosted sandboxes predominate in interactive development and single-tenant scenarios, while cloud sandboxes are more common in multiple tenants and large scale deployments. Hybrid modes are emerging specifically for scenarios where compliance or data-locality requirements coexist with the need for large pools of ephemeral execution capacity. Systematic empirical comparison across these modes under realistic agent workloads remains absent, and we treat it as part of the broader runtime agenda in §12.1.

3.5 Summary

Execution environments are the physical substrate of an agent harness: they provide the security boundary, the reset mechanism for reproducible evaluation and training, and the bounded region in which long-horizon agents can act without command human approval. The seven categories surveyed in this section show that the design space is now shaped less by a single isolation primitive than by workload fidelity, threat model, and operational mode. Managed sandboxes and computer-use environments emphasize strong isolation and realistic execution; code-specialized and browser environments emphasize throughput and evaluation structure; framework-integrated runtimes bundle capability for convenience; OS-level permission sandboxes narrow local authority; and abstraction layers make the substrate replaceable.

This design space creates a recurring tension between capability, control, and cost. High risk workloads push toward microVMs and managed clouds, interactive local workflows push toward lightweight permission boundaries, and large scale training or evaluation pushes toward fast resettable substrates. The unresolved questions about runtime defense, economics, portability, bundle versus compose design, and cross layer coupling are therefore not isolated execution layer footnotes; they reappear as survey-wide open problems in §12.1.

4 Tool Interface and Protocol Layer (T)

The Tool Interface and Protocol layer (T) is the second pillar of ETCLOVG (Claim 2 in §1). The protocols, descriptions, and registries we discuss are drawn from the 170+ project corpus that supports Claim 3.

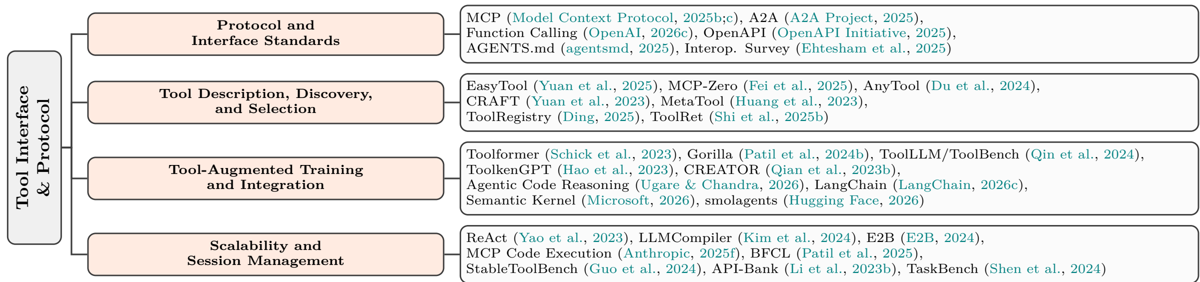

Figure 7: Representative work on tool interfaces and protocols for LLM agents, organized by the chapter's tool-layer categories.

The tool interface and protocol layer defines how an agent discovers capabilities, represents callable affordances, and executes actions across heterogeneous runtime boundaries. In practice, this layer sits at the fault line between two competing objectives: increasing capability coverage by exposing more tools, versus preserving decision quality by keeping the action space and prompt footprint small. Recent engineering guidance from production agent systems repeatedly reports that oversized tool menus can degrade reliability, increase token overhead, and amplify planning errors (Anthropic, 2025d; OpenAI, 2026c).

We organize this layer into four complementary directions: protocol and interface standards; tool description, discovery, and selection; tool-augmented model training and integration; and scalability and session management.

4.1 Protocol and Interface Standards

MCP has become the most visible tool-integration substrate for coding and enterprise agents, with an explicit host-client-server architecture and JSON-RPC-based typed exchange of tools, resources, and prompts (Model Context Protocol, 2025b;c;a). The practical value of MCP is not only schema-level interoperability, but also ecosystem liquidity: agent builders can reuse an expanding server catalog instead of implementing custom connectors for each deployment.

A2A targets a different but adjacent boundary. Rather than exposing tools to one agent process, it standardizes communication among opaque agentic applications, including discovery through Agent Cards, support for synchronous and streaming interactions, and long-running task collaboration (A2A Project, 2025). Recent protocol surveys place MCP and A2A in complementary roles: MCP primarily for tool/context access, A2A for inter-agent delegation and collaboration (Ehtesham et al., 2025). A more useful organizing principle, in our view, is to categorize tool/interface standards by the *integration boundary* they cross rather than by vendor lineage or release timeline. Under this lens, four boundaries emerge: Model↔Function (structured invocation), Agent↔External capability (runtime-to-tool decoupling), Agent↔Agent (cross-process delegation), and Agent↔Repo/environment (version-controlled policy). Table 1 summarizes this view and makes explicit why several widely compared standards (e.g., MCP vs. A2A, or Function calling vs. OpenAPI) occupy non-overlapping roles in a single harness.

Function-calling schemas and API-description standards remain foundational building blocks in this layer. OpenAI-style function calling operationalizes tool invocation through JSON schema and explicit call/return turns (OpenAI, 2026c); OpenAPI provides a language-agnostic, machine-readable API contract that many agent frameworks use as a source for tool generation and validation (OpenAPI Initiative, 2025). In addition, repository-level instruction files such as AGENTS.md and AGENT.md provide a lightweight alternative for encoding tool usage and workflow constraints directly in version control, reducing setup friction for code agents (agentsmd, 2025; agentmd, 2025).

Table 1: Tool/interface standards arranged along the integration boundary they cross (rows) and four harness-relevant capability axes (columns). ● = first-class support; ◐= partial or convention-level; ○ = out of scope. References appear in the surrounding text.

Boundary	Standard	Wire	Typed	Runtime disc.	Long-running
Model ↔ Function	Function calling	JSON	●	○	○
Agent ↔ Capability	MCP	JSON-RPC	●	●	◐
	OpenAPI	HTTP	●	◐	○
Agent ↔ Agent	A2A	JSON-RPC	●	●	●
	ACP / ANP	HTTP	●	●	●
Agent ↔ Repo / env	AGENTS.md / AGENT.md	Markdown	○	◐	○

4.2 Tool Description, Discovery, and Selection

Once protocols define *how* calls are made, the next bottleneck is *which* tools should be surfaced and selected at each step. A growing line of work studies tool documentation quality, retrieval, and dynamic candidate pruning. EASYTOOL analyzes the challenge of selecting suitable tools from large inventories (Yuan et al., 2025). AnyTool and CRAFT focus on reducing manual specification burden by automatically constructing or refining tool-use pipelines (Du et al., 2024; Yuan et al., 2023). MetaTool benchmark-style evaluations show that tool retrieval and invocation quality can diverge substantially across domains and query forms (Huang et al., 2023). More recent work such as MCP-Zero, ToolRet, and ToolRegistry emphasizes retrieval-aware orchestration and registry quality as first-order determinants of downstream agent success (Fei et al., 2025; Shi et al., 2025b; Ding, 2025). A closely related direction extends tool selection to reusable *skills*, where agents must identify relevant procedural modules rather than only compact API schemas. SkillRouter (Zheng

et al., 2026) and SkillRet (Cho et al., 2026) both study this skill-selection problem at scale, highlighting the need to retrieve the right skill from large and overlapping skill libraries.

At the system level, these results reinforce two design principles. First, "fewer but better tools" often outperforms brute-force tool exposure because it shrinks both prompt entropy and planner branching. Second, discovery pipelines must be adaptive: static, global tool lists do not scale to rapidly evolving repositories or multi-tenant enterprise deployments.

4.3 Tool-Augmented Training and Integration

The third direction shifts from runtime orchestration to model capability acquisition. Toolformer demonstrates self-supervised augmentation for learning when and how to insert API calls during generation (Schick et al., 2023). Gorilla and ToolLLM/ToolBench extend this line with larger tool corpora, instruction-tuning pipelines, and execution-centric supervision for API use (Patil et al., 2024b; Qin et al., 2024). ToolkenGPT and CREATOR explore token-level or controller-style integration to improve call formatting fidelity and planning stability (Hao et al., 2023; Qian et al., 2023b).

In production harnesses, these model side advances are typically paired with framework level runtime stacks, such as LangChain, Semantic Kernel, and smolagents, that provide memory abstractions, routing middleware, and tool adapters (LangChain, 2026c; Microsoft, 2026; Hugging Face, 2026). The coding agent setting also exposes a second, more semantic class of tools: static analyzers, type checkers, solver backed verifiers, proof assistants, and patch equivalence or fault localization checkers. Ugare & Chandra (2026) frame this space as *agentic code reasoning*: an agent explores a repository and reasons about code behavior without necessarily executing the code. Their semi formal reasoning method sits between unstructured chain of thought and fully formal verification by forcing the agent to state premises, trace program paths, and derive explicit conclusions, improving patch equivalence verification, fault localization, and code question answering. For the tool layer, the implication is that automated reasoning tools should return evidence bearing artifacts, such as traces, proof obligations, counterexamples, or structured certificates, rather than only black box yes/no answers. The combined evidence suggests that tool competence is jointly determined by pretraining and fine tuning signals, interface schema quality, and runtime selection policy; improving only one component yields limited gains.

4.4 Scalability and Session Management

A recurring operational challenge is session management under long horizons. Stateful tool sessions improve continuity but increase bookkeeping complexity, especially when calls are parallelized or delegated across multiple agents. Failure modes include stale handles, inconsistent tool state across retries, and context-window saturation from verbose tool traces. Effective harness design therefore requires explicit lifecycle controls for tool sessions, bounded tool context injection, and observability hooks that can attribute errors to either planner logic or interface/protocol failures.

Finally, ecosystem-level landscape curation can help practitioners quickly map available protocol and tooling options, but benchmark-grounded comparison remains essential for deciding which protocol and tool-routing strategy should be adopted in a given deployment.

5 Context and Memory Management (C)

Context and Memory Management (C) is the layer where prompt and context engineering meet harness engineering. We treat it as the third pillar of ETCLOVG (Claim 2 in §1), with examples drawn from the 170+ project corpus that supports Claim 3.

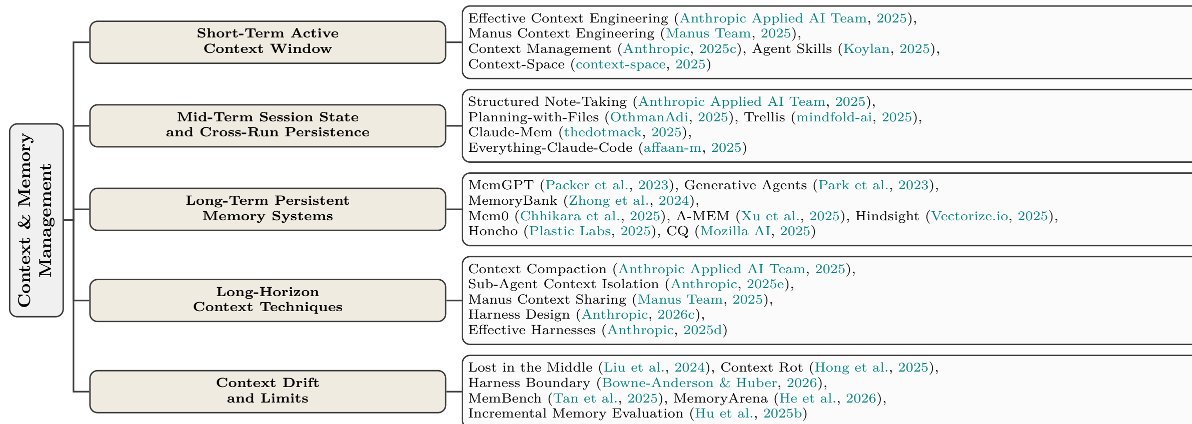

Figure 8: Representative work on context and memory management for LLM agents, organized by the chapter's context-layer categories.

This layer governs what information the model sees at each step of execution and how knowledge persists across turns and sessions. The core engineering problem is simple to state: give the model exactly the right information at each step, nothing more. Too little context and the agent lacks the state needed to act correctly. Too much and performance degrades in ways that are measurable, consistent across models, and not fully understood.

We structure this section around time horizon. Section 5.1 establishes why context requires active management rather than passive accumulation. Section 5.2 defines context engineering as a discipline distinct from prompt engineering. Sections 5.3 through 5.5 cover the three memory tiers that have emerged in production practice: the active context window (short-term), session-level persistence (mid-term), and cross-session storage (long-term). Section 5.6 addresses the coordination problem that arises when agents run for hundreds of turns. Section 5.7 closes with what remains unsolved.

5.1 Why Context Must Be Engineered

Larger context windows do not solve the memory problem. Understanding why requires looking at the architecture.

Quadratic attention cost. The transformer's self-attention mechanism computes pairwise relationships between every token in the context (Vaswani et al., 2017). For n tokens, this produces n^2 pairwise weights; compute and memory scale quadratically with context length. Engineering techniques such as FlashAttention and position-encoding interpolation reduce constant factors, but the quadratic structure is architectural. Doubling the context does not double cost – it quadruples it. This makes the context window a scarce resource.

The U-shaped attention curve. Architectural cost alone does not explain why models fail on long contexts even when computation is affordable. Liu et al. (2024) provided the key empirical result: on multi-document question answering with 20 input documents, accuracy drops by more than 30% when the relevant document sits in the middle of the context, compared to placing it at the start or end. This U-shaped performance curve holds across models, tasks, and context lengths, including models trained specifically on long contexts. The practical implication is direct: information placement matters as much as information presence. An agent that retrieves the right content but positions it poorly gains little from the retrieval.

Context rot at scale. Hong et al. (2025) evaluated 18 frontier models, including GPT-4.1, Claude Opus 4, Gemini 2.5, and Qwen3, under controlled conditions designed to isolate input length from task difficulty. Every model degraded as input grew. The degradation was non-uniform and task-specific. Semantically ambiguous queries, where the relevant passage does not lexically match the question, showed steeper degradation than exact-match queries, pointing to a compound failure: the model cannot localize relevant information,

and then cannot reason over it once located. The degradation also begins well before windows are full. A model rated for 200K tokens may show significant performance loss at 50K. This phenomenon, which Hong et al. (2025) call *context rot*, is not an edge case. It is the normal operating condition for any agent that accumulates tool results, intermediate reasoning, and file contents over multiple steps.

Taken together, these results establish that context management cannot be an afterthought. Every decision about what to include, where to place it, and when to remove it has direct consequences for agent reliability.

5.2 From Prompt Engineering to Context Engineering

The shift from *prompt engineering* to *context engineering* reflects a shift in scope (Karpathy, 2025). Prompt engineering optimizes a largely static text input to a single model call. In vision, the same framing extends to continuous learnable tokens: visual prompt tuning prepends a small set of trainable vectors to the patch sequence of a frozen vision transformer, adapting it to downstream tasks while updating less than 1% of model parameters (Jia et al., 2022; Xiao et al., 2025). Both the text and vision variants share the core characteristic of prompt engineering: a fixed, single-call optimization targeting one input modality. Context engineering optimizes the full information state available to the model at each inference step across a multi-step task.

Anthropic's applied AI team defines it as "the set of strategies for curating and maintaining the optimal set of tokens during LLM inference, including all the other information that may land there outside of the prompts" (Anthropic Applied AI Team, 2025). The guiding principle that follows is finding the smallest set of high-signal tokens that maximizes the probability of the desired outcome at each step. This principle drives every technique in this section. Progressive disclosure loads information just in time rather than upfront. Compaction removes tokens that have served their purpose. Memory retrieval pulls in only the records most relevant to the current task.

What context contains. In a deployed agent, context at inference time includes the system prompt and behavioral instructions, tool definitions and schemas, prior turn history, tool call results from the current trajectory, retrieved documents or memory records, and any dynamically injected working state. All of these compete for the same finite attention budget. Context engineering means making explicit choices about each component at each step, rather than letting them accumulate unchecked.

The maturation of this practice is visible in the ecosystem. First-principles handbooks (Kimai, 2025) and curated surveys (Meirtz, 2025) now treat context engineering as a discipline in its own right. The three-tier memory architecture, covering the active context window, session-level state, and cross-session storage, has emerged as the dominant organizing framework. It maps directly onto the classical memory hierarchy of operating systems, which provides both useful intuition and a vocabulary for matching the right technique to the right time horizon.

5.3 Short-Term: Managing the Active Context Window

Short-term context management governs what the model sees during a single inference step or a short sequence of steps within a session. These decisions have the most immediate effect on model behavior and carry the highest cost when made poorly.

System prompt calibration. The system prompt establishes the agent's behavioral envelope and consumes a fixed budget on every call. Anthropic Applied AI Team (2025) characterize the design challenge as finding the right *altitude*: prompts that are too specific introduce brittle, maintenance-heavy logic; prompts that are too vague leave the model without concrete guidance. In practice, effective system prompts are organized into clearly delineated sections for background, instructions, tool guidance, and output format, using XML tags or Markdown headers to separate them. The recommended workflow is to start with a minimal prompt on the best available model, identify failure modes empirically, then add targeted instructions to address specific gaps. Preemptively enumerating every edge case tends to bloat prompts without improving reliability.

Token-efficient tool design. Tool definitions are a major context consumer. Every schema, name, description, and parameter type is injected at every call. A large tool set can consume tens of thousands of tokens before the agent reads the user request. The principle that emerges from production deployments is to

prefer fewer, more expressive tools over a large menu of narrow ones. If a human engineer cannot say which tool applies in a given situation, the model cannot be expected to do better. Tools should be self-contained, robust to error, and unambiguous in their intended use, with descriptive parameter names that play to the model's strengths.

Just-in-time retrieval and progressive disclosure. Rather than loading all potentially relevant information upfront, effective agents maintain lightweight identifiers such as file paths, stored queries, and web links, and use them to load data on demand as the task unfolds. Anthropic Applied AI Team (2025) call this approach *progressive disclosure*. It mirrors how human experts work: we do not memorize corpora, we build external indexing systems and retrieve on demand. Claude Code uses a hybrid strategy in practice. `CLAUDE.md` files are loaded at session start to provide project context, while `glob` and `grep` commands let the agent navigate and load specific file contents just in time. This sidesteps both the stale-indexing problem and the cost of large prefills. Environmental metadata also carries implicit signals. File sizes suggest complexity; naming conventions hint at purpose; timestamps proxy for relevance. Agents can assemble understanding layer by layer, keeping only what is currently necessary in the active window.

KV-cache-aware context design. Prompt caching is the single most cost-effective optimization available to production agent deployments, and its benefit depends entirely on how context is structured. After five architectural iterations of their production agent, the Manus team identified KV-cache hit rate as "the single most important metric for a production-stage AI agent," noting that cached tokens on Claude Sonnet cost \$0.30/MTok compared to \$3.00/MTok for uncached tokens (Manus Team, 2025). Three design rules follow from the caching model. First, keep the prompt prefix stable: a single token difference at the start of the system prompt invalidates the cache for everything that follows. Second, treat context as append-only: modifying past actions or observations breaks cache reuse by creating a different prefix sequence. Third, use deterministic serialization: non-stable key ordering in JSON serialization silently invalidates caches across otherwise identical requests. Because tool definitions typically appear near the front of the serialized context, adding or removing tools invalidates cached content for all subsequent turns. Manus addresses this by using a context-aware state machine that masks token logits during decoding to prevent selection of unavailable actions, rather than modifying the tool definition list at runtime (Manus Team, 2025). Anthropic's context management release operationalizes caching at the product level through explicit `cache_control` breakpoints (Anthropic, 2025c).

The tooling ecosystem for short-term management has grown to include production skill libraries covering KV-cache compaction patterns between agent calls (Koylan, 2025) and infrastructure projects providing MCP-centric primitives for dynamic context assembly (context-space, 2025).

5.4 Mid-Term: Session State and Cross-Run Persistence

Mid-term context management addresses the gap between the active context window and a full long-term memory system. The specific problem is how an agent preserves and restores state across turns within a session, or across multiple runs of the same task. The engineering value is high: a few hundred tokens of structured working state can bridge context resets that would otherwise cause the agent to lose all accumulated progress.

Structured note-taking. The simplest mid-term technique has the agent maintain a persistent notes file, typically a `NOTES.md` or `todo.md`, reading it at the start of each run and updating it before the context is cleared. Anthropic Applied AI Team (2025) illustrate this through Claude playing Pokémon over thousands of game steps. Without any explicit instruction about memory structure, the agent developed maps of explored regions, maintained tallies of combat strategies and their outcomes against specific opponents, and tracked objectives across context resets. After each compaction step that cleared the main conversation history, the agent read its own notes and continued multi-hour training sequences without losing coherence. The key insight is that note-taking externalizes working memory. Rather than relying on conversation history to carry task state, the agent writes what matters to an external store and reads it back on demand.

File-based planning and task externalization. An extension of structured note-taking externalizes the full planning representation to disk. Tools such as planning-with-files (OthmanAdi, 2025) provide skill packages for persistent file-based planning within coding-agent workflows. Task state, specifications, inter-

mediate results, and dependency graphs are written to the filesystem at each milestone and loaded selectively at the start of the next run, bypassing the context window entirely for content that is not immediately relevant. Frameworks such as Trellis (mindfold-ai, 2025) extend this pattern to include project memory and spec injection, managing structured project state that is injected selectively based on what the current step requires.

Cross-run injection. A complementary pattern captures the salient output of a prior run and injects it at the start of the next. Tools such as claude-mem (thedotmack, 2025) operate as plugin-style memory layers: they capture session history, extract the information most useful in a future run, and prepend it to the next session's context. This requires no vector database and no graph store, while providing substantial continuity across runs. The community repository everything-claude-code (affaan-m, 2025), with over 150K GitHub stars, represents the largest open collection of these patterns and demonstrates the scale at which mid-tier memory practices have been adopted in production coding-agent deployments.

The mid-tier trade-off. Mid-term techniques provide significantly more continuity than in-window management alone, at a fraction of the infrastructure cost of full long-term memory systems. The limitation is that information flows forward from one run to the next but is not retrieved from an indexed store, and these techniques do not scale well to large volumes of history. When cross-run history grows large, or when the agent needs to retrieve a specific prior observation rather than inject a summary, mid-term techniques must be supplemented with the long-term systems described next.

5.5 Long-Term: Persistent Memory Systems

Long-term memory systems provide indexed, retrievable storage that persists across sessions, tasks, and agent instances. Where mid-term techniques rely on forward injection of summaries, long-term systems support arbitrary recall: given a query, the system retrieves the most relevant stored memories regardless of when they were created. This tier is where the agent's experience accumulates over time.

5.5.1 Foundational Architectures

OS-inspired tiered memory. Packer et al. (2023) introduced the key abstraction by treating the LLM's context window as RAM and external storage as disk, giving the model function calls that let it explicitly page information in and out. The analogy to virtual memory is precise: just as an OS provides applications the illusion of a larger memory through transparent paging, MemGPT provides the LLM the illusion of a longer context through transparent memory management. Agents can operate on contexts that far exceed the physical window limit, retrieving relevant history from external storage on demand. This paper made explicit the connection between agent memory and OS memory that underlies most subsequent long-term memory systems.

Observation, reflection, and retrieval. Park et al. (2023) introduced the Memory Stream architecture in the context of social simulation agents. The Memory Stream stores all of the agent's observations as natural-language records with timestamps and importance scores. At each step, a retrieval model surfaces the most relevant memories by combining three signals: recency, importance (scored by the agent at write time), and relevance to the current query. A second mechanism, reflection, periodically synthesizes stored observations into higher-level insights. The agent asks itself what the most salient questions are from its recent experience, retrieves relevant records, and generates conclusions that are stored back into the Memory Stream. Ablation experiments showed that each of the three components, observation, reflection, and retrieval, contributes independently to behavior quality; removing any one produces measurable degradation. The observation-reflection-retrieval triad remains the standard template for agent memory design.

Dynamic forgetting and personality modeling. Zhong et al. (2024) built MemoryBank on top of the retrieval architecture with two additions. The first is a dynamic forgetting mechanism inspired by the Ebbinghaus Forgetting Curve: memories decay over time according to an exponential model, frequent access strengthens them, and contradicted memories are resolved. The second is hierarchical user-personality summaries, updated daily as new interactions accumulate. These ideas, adaptive memory strength and user modeling, were later operationalized at scale in Mem0 and Honcho.

5.5.2 Production Memory Systems

Hybrid storage and industry adoption. Mem0 (Chhikara et al., 2025) is the most widely deployed long-term memory layer for production agents at present, with over 14 million Python package downloads, 41K GitHub stars, and native integration in CrewAI, Flowise, and Langflow. Its architecture combines three storage backends: a vector database for semantic similarity search, a graph database for relationship modeling, and a key-value store for fast fact retrieval. An LLM-based extraction layer processes new interactions, identifies facts and preferences, and routes records to the appropriate store. On the LOCOMO benchmark, Mem0 achieves 26% higher accuracy than OpenAI's native memory while using 90% fewer tokens than a full-context approach (Chhikara et al., 2025). Amazon Web Services selected Mem0 as the exclusive memory provider for its Agent SDK in 2025, reflecting its transition from research prototype to production infrastructure.

Dynamic knowledge networks. A-MEM (Xu et al., 2025) draws on the Zettelkasten knowledge management system. Rather than storing memories as flat records, it generates a structured note for each new memory that includes keywords, tags, a contextual description, and a dynamic set of links to semantically related memories. When a new memory is added, A-MEM not only stores it but also retroactively updates the context and attributes of existing related notes, allowing the memory graph to evolve as knowledge accretes. This addresses a fundamental limitation of retrieval-based systems: the relevance of a stored memory may not be apparent at write time, and its significance can only be assessed in relation to later information.

Learning from experience. Hindsight (Vectorize.io, 2025) takes the position that most memory systems focus on remembering when the real need is learning. Its API exposes three operations: `retain` to store new information, `recall` to retrieve relevant memories, and `reflect` to generate a disposition-aware response that integrates memory with current context. Internally, the retain operation extracts temporal entities, deduplicates overlapping facts, and builds evidence-grounded consolidated knowledge records. Hindsight achieves state-of-the-art performance on the LongMemEval benchmark, independently reproduced by researchers at the Virginia Tech Sanghani Center.

Reasoning-driven personalization. Honcho (Plastic Labs, 2025), developed by Plastic Labs, builds a *model* of users rather than a store of facts about them. An asynchronous reasoning pipeline called "dreaming" continuously processes past interactions in the background, extracting not just explicit facts but inferred conclusions about preferences, communication styles, and evolving goals. The entity-centric data model supports multi-agent environments where multiple agents interact with the same user: each agent maintains its own observation perspective, preventing cross-contamination while a shared user model accumulates understanding from all interactions.

Collective memory. Mozilla AI's cq (Mozilla AI, 2025) addresses a gap that the other systems leave open: shared memory across agent instances. Most long-term memory systems are per-agent or per-user; when multiple agents work on related tasks, each instance independently rediscovers what the others have already learned. cq is an open standard for shared agent learning in which agents persist, query, and confirm collective knowledge across a fleet. Its architecture is MCP-native and operates at three tiers: local knowledge stored on the developer's machine, organization-shared knowledge accessible to a team, and cross-team knowledge. The post-error hook pattern, where cq automatically queries the collective knowledge base when an agent encounters an error, provides a concrete mechanism for preventing repeated failures from being independently resolved across an organization's deployments.

5.5.3 Academic Surveys and Taxonomy

The memory mechanism survey by Zhang et al. (2025) provides the standard academic taxonomy for this layer. It distinguishes short-term working memory from long-term memory, and subdivides the latter into semantic, procedural, and episodic categories. The write-read-manage loop it formalizes has become the standard vocabulary for describing memory system behavior. A more recent survey (Du, 2026) builds on this by formalizing the loop within a POMDP-style agent cycle and proposing a three-dimensional taxonomy covering memory scope, storage format, and compositional structure. It characterizes three engineering patterns: Pattern A (monolithic context), Pattern B (context window plus retrieval store), and Pattern C (tiered memory with learned control policies). These correspond roughly to the short-term, mid-term,

and long-term tiers described in this section. The RAG survey by Gao et al. (2023) provides background on retrieval-augmented generation that underpins the retrieval tier of the production memory systems described above.

5.6 Long-Horizon Techniques: Keeping Agents Coherent Over 100+ Turns

Each of the preceding sections addresses a single time horizon. Long-horizon tasks, including large codebase migrations, multi-session research projects, and extended autonomous workflows, require coordinating all three tiers simultaneously and making real-time decisions about which technique to apply at each point. This section covers the integration-level techniques that emerge at this scale.

Context compaction. Compaction summarizes a context window approaching its limit and reinitiates execution with a compressed representation of the accumulated state. Anthropic Applied AI Team (2025) describe the calibration challenge in detail. The summary must preserve architectural decisions, unresolved bugs, and implementation details while discarding redundant tool outputs and intermediate reasoning already incorporated into later steps. The recommended tuning workflow starts by maximizing recall, capturing every potentially relevant piece of information from the trace, then iterates to improve precision by removing superfluous content. One does not start from the other direction. Removing too much early produces an agent that loses context it needed later, and that loss is not recoverable. Tool result clearing is the lightest form of compaction: full tool outputs are replaced with compact path references once the agent has acted on them. This can be applied continuously and is now a product-level feature on the Anthropic developer platform (Anthropic, 2025c).

Sub-agent context isolation. When a task requires deep exploration of a subtopic, the exploration itself generates large amounts of intermediate context that is useful within the subtask but pollutes the orchestrator's view of the overall task. The sub-agent architecture addresses this by assigning subtasks to dedicated agents, each with a fresh context window, which return only a condensed 1,000 to 2,000 token summary of their findings to the orchestrator (Anthropic, 2025e). The detailed exploration context stays isolated within the sub-agent; the orchestrator receives only the distilled result. Manus Team (2025) describe a refinement of this pattern. For simple subtasks where the orchestrator needs only the output, context is not shared at all. For complex subtasks where shared state is required, the orchestrator's full context is passed to the sub-agent. Sharing context between agents is expensive: it produces larger prefills and eliminates KV-cache reuse across different system prompts. The trade-off between isolation cost and coordination benefit must be made explicitly for each task type.

Hybrid decision frameworks. Production deployments do not apply a single technique uniformly; they select different techniques at different points in the execution based on task structure. Anthropic (2026c) characterize the framework as: pre-load content that is always needed, retrieve just-in-time content that is conditionally needed, compact history when the window approaches saturation, and spawn sub-agents when a subtask requires exploration that would pollute the orchestrator's context. Anthropic (2025d) add the harness-level perspective: checkpoint and resume patterns allow agents to survive transient failures without losing task state, and lifecycle hooks can trigger compaction or memory consolidation automatically at configurable thresholds. This is the right division of responsibility. Context management should be the infrastructure's job, not the agent's. When the harness handles compaction and consolidation automatically, the model can focus on task reasoning.

5.7 Context Drift and the Limits of Current Approaches

The binding-constraint view frames context drift as a controller-side failure mode: because the model only sees the projection of state exposed by the harness, loss or distortion of task-relevant context is a harness property rather than a model property alone (Bölük, 2026b). All of the techniques described above work, in varying degrees, under varying conditions. None of them solves the underlying problem. Context drift, the progressive degradation of agent behavior over extended interactions, remains the hardest open challenge in this layer.

The nature of the problem. Context drift is distinct from context rot, though both arise from the same architectural constraints. Context rot (Section 5.1) is a property of a single inference step: adding more tokens to a fixed context degrades retrieval and reasoning quality. Context drift is a property of an extended trajectory. As the agent accumulates turns, its behavior drifts from the original intent in ways that compaction and retrieval steps can slow but not prevent. After 100 or more turns, agents frequently repeat work already done, contradict earlier decisions without awareness, and lose track of the motivating goal (Bowne-Anderson & Huber, 2026). The root cause is not simply window saturation. Even with aggressive compaction, the summaries that survive each compression step carry subtle inaccuracies that compound over time. The agent's accumulated assumptions can diverge from the actual task environment, and nothing in the current architecture detects this divergence.

The evaluation gap. A secondary problem is that context drift is hard to measure. Standard benchmarks evaluate agents on tasks that complete within a few dozen steps; the failure modes that appear at 100 or more turns are not captured. Recent work has begun to close this gap. Tan et al. (2025) introduce MemBench, which evaluates memory quality across multi-session scenarios including temporal reasoning, knowledge update, and cross-session aggregation. He et al. (2026) introduce MemoryArena specifically for interdependent multi-session tasks, the conditions under which context drift is most destructive. Hu et al. (2025b) propose an incremental multi-turn evaluation methodology that isolates memory quality from generation quality. These benchmarks are necessary but not sufficient. They show that drift occurs, but they do not yet provide the mechanistic understanding needed to prevent it.

Why current techniques fall short. Compaction reduces the token count but cannot guarantee that the compressed representation accurately captures all relevant state. Information lost at each compression step is gone. Retrieval systems can surface relevant prior memories, but the query must be well-formed to do so. A drifting agent may not know what it needs to retrieve to correct its course. Sub-agent isolation prevents subtask context from polluting the orchestrator, but the orchestrator's own context still accumulates drift over time. Bowne-Anderson & Huber (2026) argue that these limitations point to a boundary: context engineering alone cannot solve long-horizon reliability. A full harness layer with verification loops, human-in-the-loop checkpoints at strategic intervals, and anomaly detection capable of identifying behavioral divergence is required. This motivates the governance and observability layers discussed in Sections 7 and 9 as necessary complements to context engineering, not separate concerns.

6 Lifecycle and Orchestration (L)

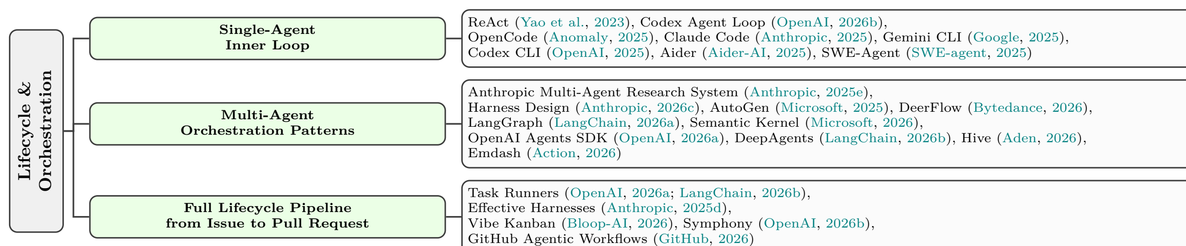

Figure 9: Representative work on lifecycle and orchestration for LLM agents, organized by the chapter's orchestration categories.

Lifecycle and Orchestration (L) concerns how an agent system carries a task across repeated model calls, tool calls, failures, revisions, and handoffs. This layer combines two concerns that are often separated in earlier frameworks: the execution flow of the agent and the operational state that the flow reads and writes. In long-running tasks, reliability depends not only on whether the model can produce a good next action, but also on whether the harness can remember what has already happened, decide what should happen next, recover from errors, coordinate subtasks, and stop when the task is complete (OpenAI, 2026b; Anthropic, 2025d; 2026c). We therefore treat Lifecycle and Orchestration as the fourth pillar of ETCLOVG (Claim 2 in §1), drawing examples from the corpus of over 100 projects that supports Claim 3 in §1.

Table 2: Representative orchestration systems, organized by orchestration level (single-agent: § 6.2, multi-agent: § 6.3, full lifecycle pipeline: § 6.4), primary orchestration pattern, and execution model. GitHub stars are rounded to the nearest thousand and recorded as of May 12, 2026.

System or Framework	GitHub URL	GitHub Stars (k)	Section	Primary Pattern	Execution Model
Single-Agent Inner Loop					
OpenCode (Anomaly, 2025)	anomalyco/opencode	159	6.2	Single loop	Hybrid
Claude Code (Anthropic, 2025)	anthropics/claude-code	123	6.2	Single loop	Hybrid
Gemini CLI (Google, 2025)	google-gemini/gemini-cli	104	6.2	Single loop	Hybrid
Codex CLI (OpenAI, 2025)	openai/codex	82	6.2	Single loop	Stateless replay
Aider (Aider-AI, 2025)	aider-ai/aider	45	6.2	Single loop	Hybrid
SWE-agent (SWE-agent, 2025)	swe-agent/swe-agent	19	6.2	Single loop	Hybrid
Multi-Agent Orchestration Patterns					
DeerFlow (Bytedance, 2026)	bytedance/deer-flow	67	6.3	Hierarchical orchestration	Stateful
AutoGen (Microsoft, 2025)	microsoft/autogen	58	6.3	Hierarchical orchestration	Stateful
oh-my-claudecode (Heo, 2026)	yeachan-heo/oh-my-claudecode	34	6.3	Team orchestration	Stateful
LangGraph (LangChain, 2026a)	langchain-ai/langgraph	32	6.3	Graph composition	Stateful
Semantic Kernel (Microsoft, 2026)	microsoft/semantic-kernel	28	6.3	Workflow orchestration	Stateful
OpenAI Agents SDK (OpenAI, 2026a)	openai/openai-agents-python	26	6.3	Hierarchical orchestration	Hybrid
DeepAgents (LangChain, 2026b)	langchain-ai/deepagents	23	6.3	Hierarchical orchestration	Stateful
Hive (Aden, 2026)	aden-hive/hive	10	6.3	Graph composition	Stateful
Emdash (Action, 2026)	generalaction/emdash	4	6.3	Fan-out	Hybrid
Full Lifecycle Pipeline from Issue to Pull Request					
Vibe Kanban (Bloop-AI, 2026)	BloopAI/vibe-kanban	26	6.4	Full lifecycle pipeline	Stateful
Symphony (OpenAI, 2026b)	openai/symphony	24	6.4	Full lifecycle pipeline	Stateful
GitHub Agentic Workflows (GitHub, 2026)	github/gh-aw	5	6.4	Full lifecycle pipeline	Hybrid

This layer spans three levels of organization. A **single-agent inner loop** is the repeated cycle in which one agent observes the current situation, decides on an action, invokes a tool or produces an output, and uses the result to continue. **Multi-agent orchestration** coordinates several such loops, usually by assigning different roles to different agents or by routing work among them. A **full lifecycle pipeline** places these agent loops inside a broader software or task workflow, such as moving from an issue to code changes, tests, review, and a pull request. Across these levels, systems also differ in how they maintain state. *Stateless replay* reconstructs the run from the recorded interaction history. *Stateful execution* stores operational state outside the prompt, for example in files, repositories, databases, task graphs, or services. Many practical systems are *hybrid*: they keep replayable histories while also relying on persistent artifacts.

6.1 Lifecycle State Management

Lifecycle state management concerns the operational state needed to continue an agent run across turns, sessions, failures, and handoffs. This includes pending subtasks, tool results, intermediate artifacts, repository changes, coordination metadata, session persistence, and checkpoint resume mechanisms. These mechanisms make orchestration durable rather than transient: the agent can continue from prior work instead of treating every step as an isolated model call.

A central trade-off is between stateless and stateful execution. Stateless designs reconstruct execution from prior interaction history, improving reproducibility and auditability but becoming costly as trajectories grow. Stateful designs persist execution state in files, databases, repositories, task graphs, or external services, improving continuity and recovery but introducing consistency and debugging challenges (OpenAI, 2026b; Anthropic, 2026c). Many practical harnesses therefore adopt hybrid designs, combining replayable interaction histories with persistent artifacts.

This state is distinct from the Context and Memory layer in §5, which concerns the information provided to the model for reasoning, such as retrieved documents, memories, or conversation history. It is also distinct from Observability in §7, which concerns logging, tracing, and monitoring system behavior. Here, the focus

is instead on the operational state used by the harness itself to continue and coordinate execution, such as pending subtasks, checkpoints, retries, shared artifacts, or execution status.

6.2 Single-Agent Inner Loop

The single-agent inner loop is the basic execution unit in agent systems. A single agent repeatedly interacts with an environment through tool use and feedback, without explicit coordination among multiple agents. This abstraction underlies interactive coding, debugging, and problem solving.

Conceptually, the single-agent loop follows the ReAct paradigm (Yao et al., 2023), interleaving reasoning, action, and observation. As emphasized in OpenAI's analysis of the Codex agent loop (OpenAI, 2026b), the behavior of such a system is determined not only by the model, but also by the harness that constructs prompts, invokes tools, manages control flow, and feeds tool outputs back into later steps.

At this level, the main execution distinction is between *Stateless replay* and *Hybrid* execution. Codex CLI (OpenAI, 2025) is the clearest example of replay-based execution, while practical coding agents often combine replayable histories with persistent artifacts such as files, repositories, or session state. In Table 2, OpenCode (Anomaly, 2025), Claude Code (Anthropic, 2025), Gemini CLI (Google, 2025), Codex CLI (OpenAI, 2025), Aider (Aider-AI, 2025), and SWE-agent (SWE-agent, 2025) are therefore grouped under the primary pattern *Single loop*. Despite their flexibility, long-horizon execution in such systems can suffer from context fragmentation, accumulated errors, and weak decomposition structure (Anthropic, 2025d), motivating more structured orchestration.

6.3 Multi-Agent Orchestration Patterns

Multi-agent orchestration extends the single-agent loop by composing multiple agents with specialized roles. Instead of asking one agent to plan, execute, check, and revise alone, a multi-agent harness can separate these responsibilities across agents or subagents. This structure supports task decomposition, parallel exploration, critique, validation, and aggregation, making it better suited to complex tasks than an isolated loop.

Several recurring patterns appear in this design space. *Hierarchical orchestration* uses a higher-level controller to assign work to agents or subagents and integrate their outputs. *Team orchestration* exposes a collection of specialized agents as a coordinated team, usually with different named responsibilities. *Workflow orchestration* organizes agents and tools into explicit stages or control logic. *Fan-out* runs multiple agents in parallel to explore diverse solutions. *Graph composition* represents agents, tools, or states as nodes in an interaction graph, allowing multiple coordination patterns to coexist. These labels match the primary pattern categories used in Table 2.

Execution at this level is usually *Stateful* or *Hybrid*, since multi-agent systems must maintain coordination state, role assignments, task graphs, shared artifacts, or intermediate results. Anthropic's planner-generator-evaluator architecture (Anthropic, 2025e) illustrates the broader principle: planning, execution, and verification can be separated into explicit roles, improving decomposition and robustness while increasing coordination overhead (Anthropic, 2026c).

The systems in Table 2 instantiate these patterns in different ways. DeerFlow (Bytedance, 2026), AutoGen (Microsoft, 2025), OpenAI Agents SDK (OpenAI, 2026a), and DeepAgents (LangChain, 2026b) are classified as *Hierarchical orchestration*. oh-my-claudecode (Heo, 2026) is classified as *Team orchestration*. LangGraph (LangChain, 2026a) and Hive (Aden, 2026) exemplify *Graph composition*, Semantic Kernel (Microsoft, 2026) exemplifies *Workflow orchestration*, and Emdash (Action, 2026) exemplifies *Fan-out*.

6.4 Full Lifecycle Pipeline from Issue to Pull Request

Full lifecycle orchestration manages an entire task workflow from specification to validated output. At this level, the focus is not only on how agents interact, but on how a harness supports long-horizon execution across planning, implementation, validation, review, and delivery.

The central abstraction is the *task runner*: a harness that manages scheduling, state persistence, retries, validation, and iterative refinement over a complete task lifecycle (OpenAI, 2026a; LangChain, 2026b). Execution is grounded in persistent artifacts such as repositories, issues, branches, files, tests, and pull requests. A canonical workflow proceeds from an issue or task specification through agent planning, code or artifact generation, validation, human review, and eventual pull request acceptance.

Because these systems operate over persistent repositories and external workflows, the dominant execution model is usually *Stateful*. Some systems combine persistent infrastructure state with replay-like or session-local execution inside subcomponents, motivating the *Hybrid* label in Table 2. Vibe Kanban (Bloop-AI, 2026), Symphony (OpenAI, 2026b), and GitHub Agentic Workflows (GitHub, 2026) are classified as *Full lifecycle pipeline*. Conceptually, these systems compose the previous abstractions: single-agent loops provide the execution primitive, multi-agent patterns provide coordination, and the task runner integrates them into an end-to-end workflow in which humans steer while agents execute.

7 Observability and Operations (O)

Observability and Operations (O) is the fifth layer of ETCLOVG and the first of the two layers our taxonomy promotes from Lifecycle Hooks to first-class status (Claim 2 in §1). The supporting systems are drawn from the 170+ project corpus that backs Claim 3.

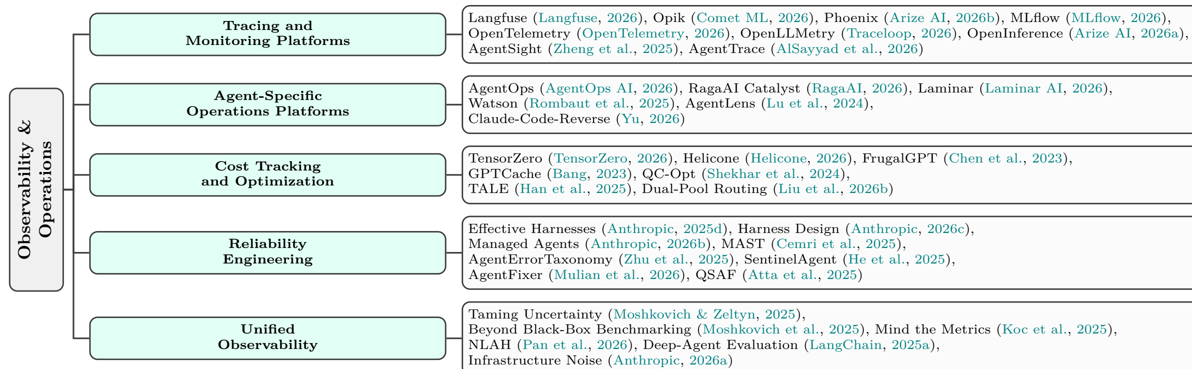

Figure 10: Representative work on observability and operations for LLM agents, organized by the chapter's operational categories.

This layer addresses how agent behavior is monitored, debugged, and made reliable in production. We argue that observability deserves independent treatment, separate from lifecycle hooks, because it has spawned a dedicated ecosystem of platforms, specifications, and engineering practices.

7.1 Tracing and Monitoring Platforms

The foundation of agent observability is *structured trace collection*: recording every LLM call, tool invocation, retrieval step, and context-assembly operation as a tree of spans that can be inspected, filtered, and replayed.

Langfuse, Opik, Arize Phoenix, and MLflow each offer interactive trace trees with latency flame charts, token-usage breakdowns, cost attribution dashboards, and prompt versioning Langfuse (2026); Comet ML (2026); Arize AI (2026b); MLflow (2026). These platforms typically ingest traces via lightweight SDK wrappers (e.g., a `@traceable` decorator) that instrument the agent's function calls with minimal code changes.

Beneath these platforms, OpenTelemetry (OTel) has emerged as the de facto instrumentation standard. The OTel community has published semantic conventions for generative AI that define span attributes for model name, temperature, token counts, and latency OpenTelemetry (2026). Two open-source projects operationalize these conventions: OpenLLMetry provides auto-instrumentation libraries for LLM providers (OpenAI, Anthropic, Cohere) and vector databases (Pinecone, Chroma, Weaviate), emitting standard OTel spans and metrics Traceloop (2026); OpenInference, maintained by Arize AI, offers a complementary specification with

OTel-aligned semantic conventions and auto-instrumentations designed as a companion to Phoenix Arize AI (2026a). By building on OTel, these libraries allow agent traces to flow into the same observability backends (Prometheus, Jaeger, Grafana, Datadog) that teams already use for traditional microservice monitoring, reducing the operational overhead of adopting agent-specific tooling.

At a deeper instrumentation level, academic work has explored techniques that go beyond application-level decorators. AgentSight Zheng et al. (2025) uses eBPF to monitor agents from *outside* the application process, intercepting TLS-encrypted LLM traffic at the SSL boundary to capture intent, and monitoring kernel events (process creation, file I/O, network calls) to capture actions. A real-time correlation engine links LLM responses to the system behaviors they trigger, and a secondary "observer" LLM performs semantic analysis to identify risks. This approach is framework-agnostic, requires no code modifications, and incurs less than 3% CPU overhead. The authors note a structural advantage over application-level tools: system-level monitoring cannot be bypassed by a compromised or misconfigured agent, making it particularly relevant for safety-critical deployments. AgentTrace AlSayyad et al. (2026) proposes a complementary schema-based logging framework that captures structured logs across three "surfaces": cognitive reasoning steps, plans, and reflections; operational tool calls and API interactions; and contextual environment state and user inputs. These surfaces are organized under a unified envelope that integrates with OTel for export. The cognitive surface is especially relevant to harness engineering because it captures explicit reasoning artifacts, plans, reflections, and self-corrections. These records provide raw material for debugging failures that arise from incorrect reasoning rather than system errors.

7.2 Agent-Specific Operations Platforms

General-purpose tracing platforms often center their abstractions on LLM calls, tool calls, and retrieval steps as span-level events. Agent-specific platforms add a layer of abstraction that captures *agent-level* concerns: multi-step session tracking, agent identity and role management, tool-selection policies, and cross-session state handoff. AgentOps SDK introduces a hierarchical span model that organizes sessions, agents, operations or tasks, workflows, tool calls, and LLM calls around the agent lifecycle rather than treating them as isolated API calls AgentOps AI (2026). RagaAI Catalyst targets RAG and multi-agent workloads with quality and safety dashboards that surface retrieval-level and agent-level anomalies RagaAI (2026). Laminar offers an OSS-first agent observability platform combining trace, evaluation, and prompt management Laminar AI (2026).

The academic community has begun to formalize these concerns. Researchers Moshkovich & Zeltyn (2025) propose a six-stage AgentOps automation pipeline: observe, collect metrics, detect anomalies, perform root-cause analysis, recommend fixes, and automate remediation. They further decompose the stakeholder space into four roles: developer, tester, SRE, and business user, each engaging with the pipeline at different lifecycle points. Their companion empirical study Moshkovich et al. (2025) introduces *task-flow discovery* as an observability primitive and demonstrates through user studies that 79% of practitioners identify non-deterministic execution flow as the most significant challenge in agentic systems.

The Natural-Language Agent Harnesses (NLAH) framework Pan et al. (2026) with a companion open-source implementation, takes the complementary step of treating the harness itself as a first-class research object. Through systematic ablation experiments, the authors quantify how individual harness modules contribute to downstream agent performance. These modules include the tool registry, permission system, hooks, skills, and context folding. The implication for observability is that the harness is not merely the subject of monitoring but a source of interventions: each module ablated is a knob that the observability pipeline can flag and tune.

Cognitive observability extends agent-level monitoring to ask not just *what* an agent did, but *why*. Watson Rombaut et al. (2025) introduces this concept formally, deploying a "surrogate agent" that reproduces the primary agent's output while generating a step-by-step reasoning trace via prompt attribution. Evaluated on MMLU and SWE-bench-lite with AutoCodeRover and OpenHands, Watson demonstrates that recovered reasoning traces are useful both for manual debugging and for automated correction, yielding measurable improvements in task success rates. AgentLens Lu et al. (2024) closes the visualization loop with a three-pane visual analytics system: Outline View for agent trajectories, Agent View for event stacks with causal

arcs, and Monitor View for synchronized environment replay. This system transforms raw execution logs into hierarchically structured behavioral narratives. A user study with 14 participants shows significant improvements over replay-only baselines on complex analytical tasks. More specialized visualization tools such as claude-code-reverse reverse-engineer the interaction chains of specific coding agents, serving as research artifacts for understanding harness-level behaviors Yu (2026).

7.3 Cost Tracking and Optimization

LLM inference is priced per token, and agent harnesses amplify cost exposure because a single user-facing task may trigger dozens of LLM calls, each with its own prompt assembly, tool-result injection, and response generation. The observability requirement is twofold: *tracking* (knowing where tokens are spent) and *optimization* (spending fewer tokens for the same outcome).

On the tracking side, TensorZero provides a unified LLMOps stack that bundles gateway, observability, experimentation, and optimization into a single service, enabling per-call and per-task cost attribution TensorZero (2026). Helicone takes a minimalist approach as a drop-in proxy that adds cost and latency monitoring with no code changes, making it particularly accessible for teams that want cost visibility without committing to a full observability platform Helicone (2026).

On the optimization side, FrugalGPT Chen et al. (2023) provides the foundational formulation by proposing three cost-reduction strategies: prompt adaptation, LLM approximation, and LLM cascading. It shows that an adaptive cascade can match GPT-4's performance with up to 98% cost reduction by routing easier queries to cheaper models. GPTCache Bang (2023) complements this approach with a semantic caching layer that intercepts repeated or paraphrased queries before they reach the LLM, using embedding similarity to retrieve cached responses. Together, routing and caching have become recurring building blocks in production cost-optimization stacks and are directly applicable to harness-level instrumentation, where each tool call and context-assembly step can be independently metered and routed.

QC-Opt Shekhar et al. (2024) extends the routing paradigm with a quality-aware framework that jointly optimizes model selection, token count, and latency under budget constraints, using a BertScore predictor to estimate output quality without invoking the target model. The *token elasticity* phenomenon discovered by TALE Han et al. (2025) reveals an important nuance: setting the token budget too low for chain-of-thought reasoning can paradoxically *increase* token consumption, because the model overflows its budget and generates more tokens than a moderately budgeted prompt would. At the infrastructure level, Dual-Pool Token-Budget Routing Liu et al. (2026b) addresses cost from the serving side, partitioning a homogeneous vLLM fleet into short-context and long-context pools and routing requests based on estimated token budget. Evaluated on Azure LLM Inference traces and LMSYS-Chat-1M, the approach reduces GPU-hours by 31–42% (projecting to $2.86M annual savings at A100 fleet scale).

For harness engineers, the implication is that cost observability must span *multiple layers*: API-level token tracking (Helicone, TensorZero), application-level routing decisions (FrugalGPT, QC-Opt), and infrastructure-level resource utilization (Dual-Pool, KV-cache occupancy). Anthropic's study on infrastructure noise in agentic coding evaluations Anthropic (2026a) provides a cautionary complement: infrastructure configuration alone can shift benchmark scores by 6 percentage points ($p < 0.01$). This finding suggests that cost optimization and evaluation fidelity are tightly intertwined, since cutting resources to save cost can silently degrade agent performance in ways that remain invisible without fine-grained observability.

7.4 Reliability Engineering

Failure recovery, checkpoint/resume, retry strategies, and session recovery are harness-level concerns that determine whether a long-running agent can survive transient failures. These concerns become acute when agents operate across multiple context windows, where each new session begins with no memory of what came before Anthropic (2025d). Anthropic's harness engineering work identifies four recurring failure modes in long-running coding agents: the agent attempts to one-shot the entire task, the agent declares the project complete prematurely, the agent leaves the environment in a broken state between sessions, and the agent marks features as done without proper testing. Their two-part solution directly addresses these reliability

concerns through harness design rather than model improvement Anthropic (2025d). It uses an initializer agent to decompose the task into a structured feature list and set up progress-tracking artifacts, including a git repo, progress file, and init script, followed by a coding agent that works incrementally on one feature at a time while leaving clean handoff state.

Subsequent work extends this pattern. Anthropic's GAN-inspired three-agent architecture (planner, generator, evaluator) introduces sprint contracts and separated self-evaluation to address the tendency of agents to over-praise their own work Anthropic (2026c). Notably, the authors demonstrate that harness complexity should track model capability: when upgrading from Opus 4.5 to Opus 4.6, they removed the sprint construct and context resets entirely, reducing cost from $200 to $125 while maintaining output quality. This illustrates a general principle: *every harness component encodes an assumption about what the model cannot do on its own, and those assumptions go stale as models improve.*

At the infrastructure level, Anthropic's Managed Agents architecture Anthropic (2026b) decouples the "brain" (harness + LLM) from the "hands" (sandboxes and tools) and the "session" (durable event log), making each independently recoverable. If the harness crashes, a new instance can be rebooted with `wake(sessionId)` and resume from the last event in the session log. If a sandbox fails, the harness catches the error as a tool-call failure and provisions a new one. Credentials are stored in an external vault, never entering the sandbox. This architecture transforms all components from "pets" (irreplaceable, hand-tended instances) into "cattle" (interchangeable, automatically reprovisioned), a prerequisite for operating agents reliably at scale.

The academic literature provides taxonomies of the failure modes that reliability engineering must address. MAST Cemri et al. (2025) identifies 14 failure modes across multi-agent systems, clustered into system design issues, inter-agent misalignment, and task verification problems ($\kappa = 0.88$). AgentErrorTaxonomy Zhu et al. (2025) decomposes failures by module (memory, reflection, planning, action, system) and shows that error propagation—a single root-cause failure cascading through subsequent decisions—is the central reliability bottleneck. Their AgentDebug framework yields up to 26% relative improvement in task success by isolating root causes rather than treating surface symptoms. Meanwhile, Vinay (2025) contributes a system-level taxonomy of 15 hidden failure modes specific to production LLM deployments, including *version drift* (model updates silently changing behavior), *cost-driven performance collapse* (aggressive optimization degrading quality), and *multi-step reasoning drift* (gradual coherence loss over long sequences). QSAF Atta et al. (2025) formalizes *cognitive degradation* as a six-stage lifecycle validated across five LLM platforms, revealing that hallucinated outputs can be stored in persistent memory and reused in future sessions, creating self-reinforcing degradation loops that cross session boundaries.

For runtime detection, SentinelAgent He et al. (2025) models multi-agent interactions as a graph and classifies anomalies at three tiers (global, single-point, multi-point) using LLM-as-judge with human-in-the-loop policy refinement. AgentFixer Mulian et al. (2026) deploys 15 validation tools on IBM's CUGA production system, achieving 64–88% issue detection rates and finding that 38% of task failures trace to parsing errors—a failure mode fully addressable by deterministic checks. The practical implication is that reliability for agents requires a *layered* detection strategy: lightweight rule-based checks for structural failures (malformed tool calls, schema violations), statistical monitoring for performance drift (latency, token usage, cost trends), and LLM-based semantic analysis for reasoning failures (hallucination, plan-action misalignment, premature stopping).

7.5 Discussion: Toward Unified Observability

The observability–evaluation gap. The LangChain survey's 89%/52% split reflects a structural disconnect: trace-collection tools and evaluation frameworks have largely been developed by different communities with different interfaces. Closing this gap requires tighter integration, such as automatically generating regression test cases from production failure traces, or using online evaluation scores as alerting signals. LangChain's own analysis of deep-agent evaluation LangChain (2025a) and Anthropic's quantification of infrastructure noise Anthropic (2026a) both argue that evaluation and observability must be treated as a single feedback loop rather than separate concerns.

Unified LLMOps stacks. Tools like TensorZero and Langfuse are moving toward bundling gateway, observability, evaluation, and optimization into a single platform, reducing integration overhead. The NLAH framework pushes this further by making the harness itself modular and introspectable, with each component's contribution quantifiable through ablation. Anthropic's Managed Agents architecture takes the infrastructure-level view, virtualizing harness components into substitutable interfaces (`execute()`, `emitEvent()`, `getEvents()`) that can accommodate future harness designs without changing the surrounding system.

The harness-as-assumption principle. Every harness component encodes an assumption about what the model cannot do on its own Anthropic (2026c). As models improve, observability systems must detect not only when agents fail, but also when harness components have become unnecessary overhead. The ideal observability system would include a meta-monitoring layer that tracks which interventions (context resets, evaluator feedback loops, tool restrictions) are still load-bearing, enabling continuous harness simplification alongside model upgrades.

8 Verification and Evaluation (V)

Verification and Evaluation (V) is the sixth pillar of ETCLOVG (Claim 2 in §1). The systems and benchmarks we discuss are drawn from the 170+ project corpus that supports Claim 3.

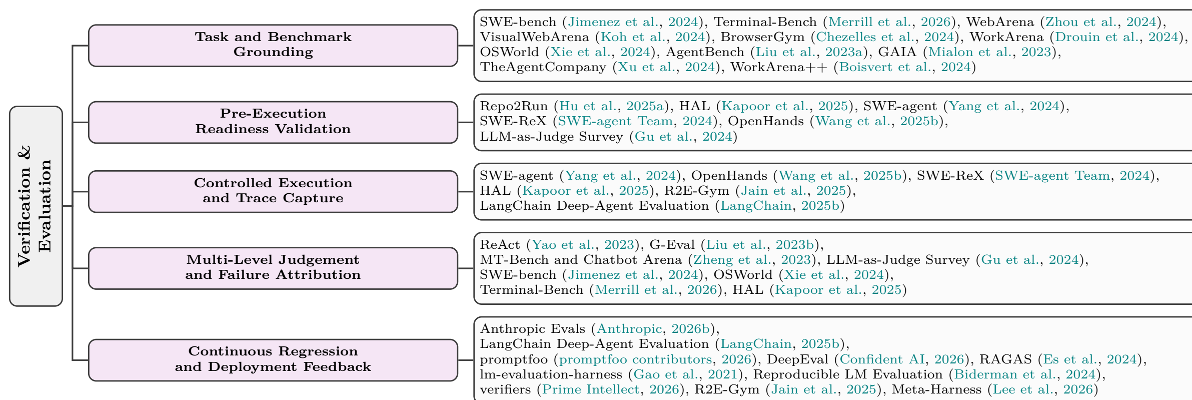

Figure 11: Representative work on verification and evaluation for LLM agents, organized by the chapter's task-to-feedback lifecycle.

8.1 Harness Evaluation as a Task-to-Feedback Lifecycle

A harness-aware evaluation should treat the reported score as a property of the model–harness pair, not of the model alone. This motivates evaluation protocols that either lock the harness across models or vary harness configurations as an explicit experimental factor (Bölük, 2026b). We organize harness evaluation as a task-to-feedback lifecycle: a structured process that begins with defining what an agent is asked to do and ends with feeding evaluation results back into harness improvement. Figure 12 summarizes this five-stage task-to-feedback lifecycle. This lifecycle builds on a key difference between conventional LLM evaluation and agent evaluation. Conventional LLM evaluation primarily scores outputs on fixed inputs, whereas harness evaluation measures an execution episode: a task is grounded in an environment, the agent interacts with tools and state over time, traces are captured, and the evaluator judges both the final result and the path taken to reach it (Kapoor et al., 2025; Anthropic, 2026b; LangChain, 2025b).

A central motivation for this lifecycle is that evaluation infrastructure noise can masquerade as model failure: failed runs may reflect not only model limitations, but also broken tools, stale context, non-reset sandboxes, flaky tests, benchmark ambiguity, or unstable judges (Kapoor et al., 2025; Hu et al., 2025a; Jimenez et al., 2024). Thus, evaluation should convert agent behavior into structured judgements, failure attribution, and regression feedback, rather than merely reporting a final score.

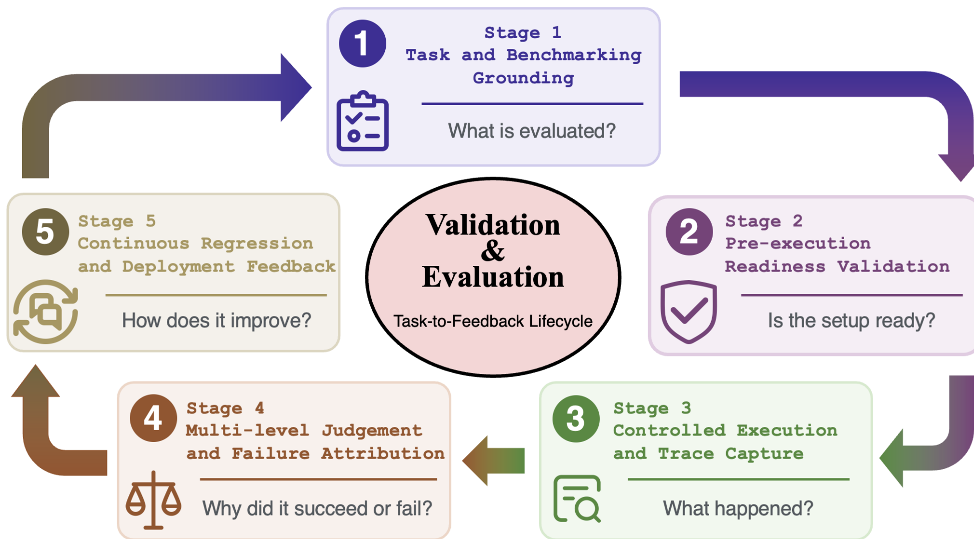

Figure 12: The task-to-feedback lifecycle for harness evaluation. Section V organizes verification and evaluation as a five-stage quality-control loop: task and benchmark grounding, pre-execution readiness validation, controlled execution and trace capture, multi-level judgement and failure attribution, and continuous regression and deployment feedback. The figure emphasizes that harness evaluation should not only report final success rates, but also diagnose failure sources and feed the resulting evidence back into systematic harness improvement.

The five-stage decomposition follows the causal path of an agent evaluation run. Before an agent can be evaluated, the benchmark must specify the task, environment, tools, constraints, and success criteria. Before execution, the setup must be validated so that environment or grader failures are not mistaken for model failures. During execution, the harness must capture traces that make the run diagnosable. After execution, the system must judge the outcome, trajectory, and evaluator reliability, then attribute failures to likely harness components. Finally, the resulting diagnosis should feed back into regression tests and future harness revisions. In this view, evaluation is not a terminal scoring step, but a quality-control loop over the full agent harness.

We organize the discussion around five stages.

- **Task and Benchmark Grounding** defines what is being evaluated, in which environment, with which tools, constraints, and success criteria (§8.2).
- **Pre-execution Readiness Validation** checks whether the sandbox, dependencies, tools, context state, permission policies, budgets, and graders are correctly initialized before the agent runs (§8.3).
- **Controlled Execution and Trace Capture** runs the agent under reproducible conditions while recording model outputs, tool calls, state changes, errors, retries, cost, and latency (§8.4).
- **Multi-level Judgement and Failure Attribution** evaluates the run at the outcome, trajectory, and evaluator levels, then localizes likely failure sources across the harness stack (§8.5).
- **Continuous Regression and Deployment Feedback** converts evaluation outcomes into regression tests, monitoring signals, and engineering feedback for subsequent harness revisions (§8.6).

8.2 Stage 1: Task and Benchmark Grounding

The first stage asks what is being evaluated. For LLM agents, a task is not simply a natural-language prompt, but an embedded problem defined by an environment state, available tools, allowed actions, constraints,

termination conditions, and success criteria. Task grounding is therefore the foundation of harness evaluation: without a well-specified environment and success condition, later scores cannot be interpreted reliably.

8.2.1 Software Engineering and Terminal Tasks

Software engineering benchmarks are among the most mature forms of agent evaluation because they combine realistic tasks with executable outcome verification. SWE-bench grounds each task in a real GitHub issue and repository snapshot, then evaluates whether the generated patch resolves the issue by running tests (Jimenez et al., 2024). Terminal-Bench extends this idea to command-line workflows, where agents interact with terminal environments by editing files, running commands, setting up dependencies, and interpreting failures (Merrill et al., 2026). The key insight from this category is that strong outcome verifiers require strong task grounding. Tests can only serve as reliable evaluators when the repository state, dependencies, and intended success criteria are precisely specified. Otherwise, a failed test may reflect an invalid patch, but it may also reflect an inconsistent environment or an underspecified benchmark instance.

8.2.2 Web, Browser, and Computer-Use Tasks

Web and computer-use benchmarks ground tasks in interactive environments where success is defined by browser, desktop, or application state changes. WebArena introduced realistic web environments for autonomous agents (Zhou et al., 2024); VisualWebArena adds multimodal web tasks (Koh et al., 2024); BrowserGym provides a common browser-agent research substrate (Chezelles et al., 2024); and WorkArena focuses on ServiceNow-based knowledge work (Drouin et al., 2024). OSWorld further broadens evaluation to real computer environments involving desktop applications, files, and multi-application workflows (Xie et al., 2024). These benchmarks show that browser and computer-use evaluation is simultaneously an evaluation problem and an environment-design problem. The benchmark must provide a realistic interface, isolate task state, expose appropriate observations, and define measurable success predicates. This creates a direct bridge between the execution environment layer and the evaluation layer.

8.2.3 Cross-Domain and Enterprise Workflow Tasks

A third group of benchmarks tests broader agent capabilities across heterogeneous tools, environments, and workflows. AgentBench evaluates LLM agents across operating systems, databases, web browsing, and games (Liu et al., 2023a); GAIA targets general assistant tasks requiring reasoning, tool use, and multimodal information access (Mialon et al., 2023); and TheAgentCompany simulates workplace-style digital labor (Xu et al., 2024). WorkArena and WorkArena++ further emphasize enterprise workflows, where success depends on navigating complex software rather than answering isolated questions (Drouin et al., 2024; Boisvert et al., 2024). The main contribution of this benchmark family is coverage: it tests whether a harness can generalize across task types, tool interfaces, state representations, workflows, and success conditions.

8.3 Stage 2: Pre-execution Readiness Validation

The second stage asks whether the evaluation can be run fairly and reproducibly. This stage is often invisible in benchmark leaderboards, but it is central to harness evaluation. Before an agent begins, the harness must validate that the environment, tools, context state, permission boundaries, budget constraints, and evaluators are correctly initialized. Without this readiness layer, downstream failures become difficult to attribute: a failed run may be caused by the agent, but it may also be caused by the evaluation setup.

8.3.1 Environment and Sandbox Readiness

Environment readiness checks whether the sandbox, repository, browser, terminal, or VM starts from a known baseline. In software engineering, this includes the repository snapshot, dependencies, task setup, and test executability; in browser and computer-use settings, it includes website resets, browser profiles, desktop state, and service availability.

Several systems make this problem explicit. SWE-bench relies on executable repository states and test-based verification (Jimenez et al., 2024); Terminal-Bench packages terminal tasks with controlled environments and

verification logic (Merrill et al., 2026); and OSWorld uses real computer environments with execution-based evaluation scripts (Xie et al., 2024). Repo2Run targets dependency and environment setup by automating executable Docker environments for repositories (Hu et al., 2025a), while HAL emphasizes standardized, cost-aware evaluation infrastructure for agents (Kapoor et al., 2025). Together, these systems show that environment reset and dependency validation are part of the measurement instrument.

8.3.2 Tool, Context, and Permission Readiness

Readiness validation is cross-layer. Tool readiness checks whether APIs, MCP servers, browser controls, shell commands, and file operations are available and consistently described. Context readiness checks whether histories, memory stores, scratchpads, and retrieved documents are reset or intentionally initialized. Permission readiness checks whether file access, credentials, network access, and approval gates match the benchmark specification.

This stage connects evaluation to the tool, context, and governance layers. If tool descriptions change across runs, the benchmark no longer measures the same action space. If memory or context is not reset, the agent may benefit from leaked state. If permissions are too restrictive, a capable agent may fail for governance reasons; if they are too permissive, unsafe or benchmark-exploiting behavior may go undetected. Systems such as SWE-agent, SWE-ReX, and OpenHands illustrate that tool exposure, shell access, browser access, execution backends, and runtime interfaces are not separable from evaluation design: the agent-computer interface determines what actions are possible and therefore what a benchmark actually measures (Yang et al., 2024; SWE-agent Team, 2024; Wang et al., 2025b). A rigorous evaluation harness should therefore record these configurations, including the tool registry, context policy, permission policy, budget, timeout, model configuration, and runtime interface, as evaluation metadata.

8.3.3 Evaluator and Grader Readiness

The evaluator itself must also be validated before execution. Deterministic graders should be checked for flakiness, missing dependencies, and compatibility with environment state; LLM-as-Judge prompts, rubrics, and judge models should be versioned; and human-audit protocols should be defined in advance.

In benchmarks such as SWE-bench, executable tests provide scalable outcome verification, but score validity still depends on correct environment setup, task specification, and test reliability (Jimenez et al., 2024). For open-ended tasks, LLM-as-Judge systems require calibration, bias mitigation, and consistency checks rather than blind reuse of a strong model as a grader (Zheng et al., 2023; Gu et al., 2024). Evaluator readiness is therefore a prerequisite for meaningful failure attribution.

8.4 Stage 3: Controlled Execution and Trace Capture

The third stage asks what happened during execution. In conventional LLM evaluation, the evidence may consist of a single input-output pair. In agent evaluation, the trace is a multi-step trajectory: the model observes state, reasons, calls tools, receives tool outputs, modifies the environment, handles errors, and eventually terminates. Controlled execution and trace capture are therefore necessary for turning benchmark runs into diagnosable evidence.

8.4.1 Controlled Rollouts and Reproducible Execution

A rollout is the basic unit of agent evaluation. It includes the task, model configuration, harness configuration, action sequence, intermediate observations, final state, and grading result. A controlled rollout fixes key sources of accidental variation, including environment state, tool availability, timeout, budget, permission policy, and evaluator version. When nondeterminism remains, repeated rollouts can expose variance rather than hiding it behind a single score.

Several systems illustrate this need for reproducible execution. SWE-agent exposes agents to repository navigation, file editing, and test execution as part of the interaction loop (Yang et al., 2024). OpenHands combines code editing, command-line execution, browser interaction, sandboxed execution, and benchmark integration (Wang et al., 2025b). SWE-ReX abstracts over local and remote sandboxed shell environments

so that execution backends can change without changing agent logic (SWE-agent Team, 2024). LangChain further decomposes deep-agent evaluation into single-step validation, full-turn evaluation, and multi-turn simulation (LangChain, 2025b). Together, these systems show that reproducibility requires replaying the full model–tool–environment interaction, not merely rerunning a model call.

8.4.2 Trace-Native Evaluation

Trace capture converts binary scoring into causal diagnosis. A trace-native harness should record model outputs, tool calls, tool results, environment state changes, context snapshots, errors, retries, recovery actions, token usage, latency, and cost. These traces distinguish superficially similar failures: one agent may never find the relevant file, while another may find it but produce an invalid patch. They can also reveal undesirable success, such as benchmark exploitation, excessive tool calls, or permission violations.

HAL treats logs and traces as first-class artifacts for standardized agent evaluation, using large-scale agent logs to inspect behaviors that final scores may hide (Kapoor et al., 2025). R2E-Gym similarly links executable trajectories and hybrid verifiers to both test-time evaluation and training-time feedback (Jain et al., 2025). For harness engineering, traces are therefore not auxiliary debugging artifacts; they are primary evaluation data.

8.4.3 Cost, Latency, and Resource Tracking

Agent evaluation should report not only quality, but also the cost of achieving it. Long-horizon agents may improve success rates by spending more tokens, calling more tools, retrying more often, or using stronger models. A harness-level evaluation should therefore track token usage, model cost, wall-clock latency, tool-call count, retry count, and resource consumption.

Rather than ranking agents only by success rate, evaluations should expose a success-cost-latency frontier. HAL explicitly frames agent evaluation as standardized and cost-aware (Kapoor et al., 2025), while LangChain's deep-agent evaluation guidance emphasizes reproducible environments and multi-granularity evaluation for comparing cost-quality trade-offs (LangChain, 2025b). This is especially important for deployed agent services, where the unit of optimization is a recurring workload under budget and latency constraints.

8.5 Stage 4: Multi-level Judgement and Failure Attribution

After controlled execution and trace capture, the central question is how the run should be judged and, if it fails, how the failure should be localized. We define Stage 4 as a combined process of *multi-level judgement* and *failure attribution*. Multi-level judgement evaluates the run at three levels: whether the final outcome is correct, whether the trajectory is efficient and policy-compliant, and whether the evaluator itself is reliable. Failure attribution then uses these signals to diagnose where the failure most likely originated, such as the model, tool interface, context manager, execution environment, orchestration loop, benchmark specification, or evaluator. This stage is what distinguishes harness evaluation from ordinary benchmark evaluation: the goal is not merely to assign a score, but to convert execution evidence into actionable engineering diagnosis.

8.5.1 Outcome-level Evaluation

Outcome-level evaluation measures whether the final task objective was achieved. In software engineering, this is often operationalized through unit tests, regression tests, or issue-specific verification, as in SWE-bench (Jimenez et al., 2024). In terminal tasks, outcome evaluation may depend on final files, command outputs, or task-specific checkers (Merrill et al., 2026). In browser and enterprise workflow benchmarks, it often depends on final environment state, such as whether a form was submitted, a ticket was updated, or a configuration was changed (Zhou et al., 2024; Drouin et al., 2024). In general assistant benchmarks such as GAIA, the outcome may be an answer string or structured response (Mialon et al., 2023).

The strength of outcome-level evaluation is scalability and interpretability: it provides a clear task-level success signal and supports leaderboard comparison. Its limitation is compression. A full execution episode is reduced to one value, hiding whether the agent succeeded robustly, safely, or efficiently. Outcome evaluation

is therefore necessary but not sufficient. It answers whether the task was completed, but not whether the harness produced a trustworthy execution.

8.5.2 Trajectory-level Evaluation

Trajectory-level evaluation measures the quality of the path taken by the agent. It asks whether the agent selected appropriate tools, ordered actions sensibly, avoided redundant calls, respected permission boundaries, recovered from errors, and maintained context consistency over time. This level is where agent evaluation diverges most clearly from conventional output evaluation: a final answer can be correct while the trajectory is still unacceptable, for example if the agent wastes excessive tokens, repeatedly calls irrelevant tools, violates permissions, or relies on brittle benchmark-specific behavior.

ReAct provides a conceptual foundation for trajectory-level analysis by representing agent behavior as interleaved reasoning, acting, and observing steps (Yao et al., 2023). Modern agent benchmarks extend this view into richer environments where each action changes the state available to future steps. WebArena, WorkArena, OSWorld, and Terminal-Bench all produce trajectories whose intermediate actions are meaningful evaluation objects rather than hidden implementation details (Zhou et al., 2024; Drouin et al., 2024; Xie et al., 2024; Merrill et al., 2026). LangChain's evaluation patterns likewise emphasize single-step and full-turn evaluations because they allow developers to inspect decision quality before an entire run succeeds or fails (LangChain, 2025b).

Trajectory-level judgement is especially valuable for harness engineering because it provides a route from failure to repair. If an agent fails because it chooses the wrong tool, the tool interface may need improvement. If it forgets prior constraints, the context layer may need adjustment. If it loops without recovery, the orchestration layer may need lifecycle controls. Thus, trajectory evaluation turns benchmark results into layer-specific engineering feedback.

8.5.3 Evaluator-level Evaluation

Evaluator-level evaluation asks whether the evaluator itself can be trusted. Deterministic verifiers, such as unit tests or state-based checkers, are reproducible, cheap, and easy to compare across systems, but they are narrow and difficult to design for open-ended tasks. LLM-as-Judge systems are flexible for natural-language outputs and trajectory assessment, but they introduce bias, nondeterminism, and additional cost. Human audits remain important for ambiguous or safety-critical cases, but they do not scale to continuous evaluation.

G-Eval showed that LLM-based evaluators can better align with human judgements on NLG evaluation tasks, while also highlighting bias toward LLM-generated text (Liu et al., 2023b). MT-Bench and Chatbot Arena systematically studied LLM-as-Judge and identified position bias, verbosity bias, self-enhancement bias, and limited reasoning ability (Zheng et al., 2023). Recent surveys further argue that reliable LLM-as-Judge systems require standardization, bias mitigation, consistency checks, and meta-evaluation (Gu et al., 2024).

For agent harnesses, evaluator-level evaluation is not optional. If a grader is flaky, a test is nondeterministic, or an LLM judge is biased, then evaluation noise can be misattributed to the agent or model. A robust harness should therefore prefer layered graders: deterministic checks for objective state changes, LLM judges for semantic or trajectory-level assessment, and human audits for ambiguous or high-risk cases. The evaluator should be treated as a component under test, not as an oracle outside the system.

8.5.4 Failure Attribution Across Harness Layers

Failure attribution identifies what should be fixed after a run has been judged. A failed rollout may originate from model reasoning, tool-interface design, stale or compressed context, sandbox instability, orchestration loops, benchmark ambiguity, or evaluator unreliability (Kapoor et al., 2025; Hu et al., 2025a; Jimenez et al., 2024; LangChain, 2025b). Outcome-level signals indicate whether the objective was achieved; trajectory-level signals reveal whether the action sequence was coherent, efficient, and policy-compliant; evaluator-level

signals indicate whether the score itself is trustworthy. Together, these signals support diagnosis rather than mere scoring (Kapoor et al., 2025; LangChain, 2025b).

In practice, attribution is rarely a single-label classification problem. A vague task specification may cause the agent to choose the wrong tool; an overly compressed context may cause it to forget a constraint; a sandbox dependency issue may trigger recovery behavior that increases cost; or a flaky judge may obscure a valid solution. For this reason, failure attribution should be understood as a diagnostic process over the full trace rather than as a post-hoc label attached to the final score. The output of Stage 4 is therefore not only a judgement, but a structured diagnosis that Stage 5 can feed back into regression testing and harness improvement.

8.6 Stage 5: Continuous Regression and Deployment Feedback

The final stage asks how evaluation results drive the next harness revision. Agent harnesses evolve continuously: prompts, tool schemas, MCP servers, context policies, sandbox images, orchestration loops, governance rules, and judge prompts may all change over time. Continuous evaluation turns benchmark results and production failures into a regression system for the harness itself.

8.6.1 Regression Evaluation for Harness Changes

Regression evaluation should be triggered by harness changes as well as model changes. Changes to tool descriptions, context-compaction policies, sandbox images, permission rules, or judge prompts can alter agent behavior or evaluation scores. Because harness components interact, local improvements can create global regressions.

The practical pattern is to maintain a layered evaluation suite: unit-like tests for tool schemas and deterministic validators, single-step tests for local decisions, full-rollout tests for end-to-end completion, and multi-turn simulations for long-horizon coherence. LangChain's deep-agent evaluation guidance reflects this layered view (LangChain, 2025b), while Anthropic frames evals as automated tests with explicit grading logic that can run during development (Anthropic, 2026b).

8.6.2 Evaluation Frameworks and LLMOps Integration

General evaluation frameworks provide reusable infrastructure for continuous testing. Promptfoo supports prompt and LLM-application regression testing; DeepEval provides unit-test-like abstractions; RAGAS focuses on retrieval-augmented generation evaluation; and lm-evaluation-harness supports standardized language-model evaluation (promptfoo contributors, 2026; Confident AI, 2026; Es et al., 2024; Biderman et al., 2024; Gao et al., 2021). Although these tools were not all designed for long-running agents, they provide building blocks for regression-oriented harness evaluation.

Evaluation should also connect with observability. Production traces can become regression tests, and evaluation failures can become observability signals. This link closes the loop between monitoring what happened and constructing controlled tests that reproduce and diagnose it.

8.6.3 Verifier-Based and Training-Time Evaluation

A newer direction uses evaluators not only as offline metrics, but also as training signals and optimization targets. Verifier-based environments and RL-style agent gyms, including R2E-Gym and verifiers, use environment feedback as reward or validation signals for improving agent policies and scaffolds (Jain et al., 2025; Prime Intellect, 2026). This shifts evaluation from a post-hoc measurement step to an active component of agent training, test-time adaptation, and scaffold selection.

Meta-Harness extends this direction by treating harness design itself as an object of automated search (Lee et al., 2026). Instead of evaluating only which model performs best under a fixed harness, this view asks which harness structure, prompting strategy, tool interface, or control loop produces more reliable agent behavior. In this framing, evaluation is not the end of the pipeline; it is the signal that allows the harness to improve.

8.7 Summary

Verification and evaluation is the layer that turns agent behavior into engineering evidence. We have organized this layer as a task-to-feedback lifecycle. Stage 1 grounds tasks in realistic environments and success criteria. Stage 2 validates that the evaluation setup is ready, fair, and reproducible. Stage 3 runs controlled rollouts and captures traces. Stage 4 judges each run at the outcome, trajectory, and evaluator levels. Stage 5 feeds the results into continuous regression testing and harness improvement.

This lifecycle reframes evaluation from a leaderboard mechanism into a quality-control loop for agent harnesses. A final success rate remains useful, but it is no longer sufficient. For long-running agents, the central evaluation question is not only whether the agent succeeded, but why it succeeded or failed, whether the path was acceptable, whether the evaluator was trustworthy, and which harness component should be improved next.

9 Governance and Security (G)

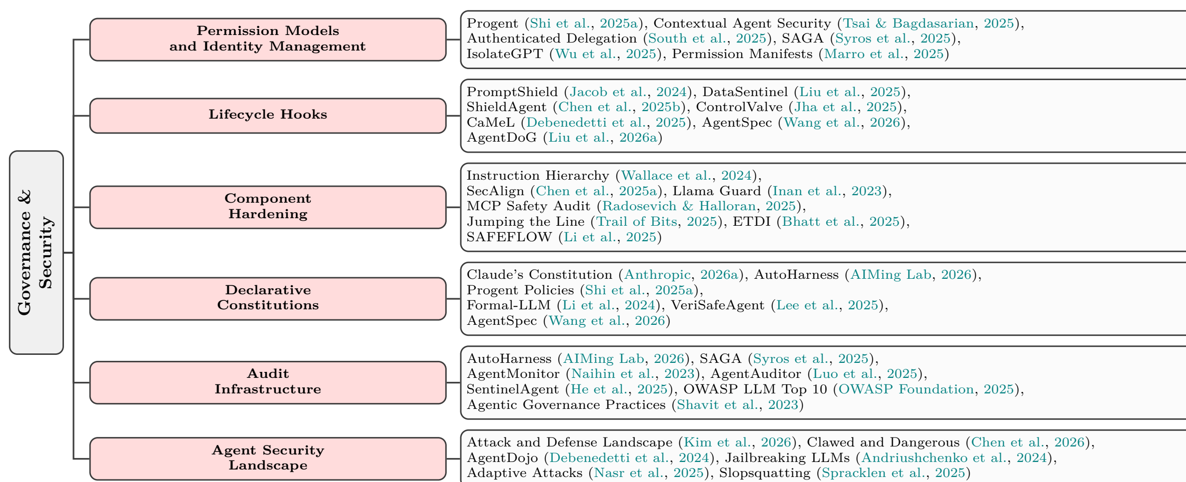

Figure 13: Representative work on governance and security for LLM agents, organized by the chapter's security categories.

This layer addresses how agent behavior is constrained, made safe, and held accountable. LLM agents now execute shell commands, send emails, commit code, and invoke third-party APIs; for production deployments, a central question is under what constraints they should act and who bears accountability when those constraints fail. In production systems, governance has its own tooling ecosystem, including permission engines, policy languages, audit pipelines, and gateway controls. This ecosystem is distinct from the lifecycle hooks and observability infrastructure covered in Sections 6 and 7, warranting treatment as an independent layer in the ETCLOVG taxonomy. We organize the discussion around five mechanisms: permission models and identity management (§9.1), lifecycle hooks (§9.2), component hardening (§9.3), declarative constitutions (§9.4), and audit infrastructure (§9.5). We then situate these mechanisms within the broader agent security landscape (§9.6) and close with open research directions (§9.7).

9.1 Permission Models and Identity Management

The first governance question for any agent harness is access control: which tools, files, network endpoints, and system resources can the agent reach? Agent workloads make this harder than conventional RBAC or ABAC because the required tool set often depends on a natural-language task that is unknown at deployment time. The literature therefore moves along a granularity axis: static boundaries fixed at deployment, contextual policies evaluated per tool call, and cross-boundary permissions for interactions outside the local harness.

Static permission boundaries. Production coding agents commonly predefine a permission scope and require escalation for actions outside it. Codex (OpenAI, 2025) runs shell commands in a sandbox with restricted filesystem and network access, while Gemini CLI (Mullen & Salva, 2025) combines workspace-scoped file access with command allow and deny lists. These boundaries are easy to inspect, but they cannot express task-specific intent.

Moving beyond fixed rules requires expressing constraints that depend on the runtime context of each tool call.

Context-dependent privilege control. Progent (Shi et al., 2025a) introduces a domain-specific language (DSL) whose predicates range over tool names, arguments, and environment state. The runtime evaluates these predicates before each invocation, allowing least-privilege policies to vary by role, user, or session. Conseca (Tsai & Bagdasarian, 2025) pushes this idea further by generating a task-specific policy from trusted context and enforcing it with a separate deterministic checker. The separation between policy generation and enforcement is important: the generative component can adapt to the task, while the enforcer remains auditable.

The preceding approaches operate within a single agent boundary. Multi-agent settings introduce additional challenges around both access control and agent identity.

Identity management and inter-agent access control. Before granting access, a system must establish *who* is requesting it. South et al. (2025) argue that agents need authenticated delegation: a framework extending OAuth 2.0 and OpenID Connect with User ID Tokens, Agent ID Tokens, and scoped Delegation Tokens that would bind user intent to agent-specific permissions. This token chain would create a verifiable accountability trail from human principal to agent action. SAGA (Syros et al., 2025) builds on this principle for multi-agent settings through a provider-mediated architecture. Agents register with cryptographic one-time keys, and short-lived access tokens enforce user-defined contact policies between agents. IsolateGPT (Wu et al., 2025) takes an architectural approach, adopting a hub-and-spoke model where each third-party application executes in an isolated spoke with its own LLM, memory, and tool access. Inter-spoke communication passes through a trusted hub, so cross-application data exchange becomes an explicit governance decision rather than an incidental side effect of a shared context window.

Credential management. Agents routinely handle API keys, session tokens, and one-time passwords. Exposing these credentials to the LLM context creates exfiltration risk. Skyvern (Skyvern-AI, 2025) illustrates a common pattern: store secrets in a vault, expose only placeholders to the LLM, and substitute raw values only at the automation layer that executes the authenticated action. The unresolved problem is secret lifecycle management for long-horizon sessions, where tokens may expire or be revoked mid-trajectory and renewed credentials must still remain outside the model context.

Web-level permission coordination. The mechanisms above operate within a single harness's trust boundary and govern one deployment. Cross-organizational interactions, where agents traverse third-party websites or call services owned by other parties, require coordination that no single harness can enforce in isolation. Marro et al. (2025) propose `agent-permissions.json`, a lightweight manifest file analogous to `robots.txt`. Website owners can declare which UI elements agents may use, rate limits, concurrency constraints, and actions that require human confirmation. Such manifests do not solve authentication, but they show how governance begins to cross organizational boundaries.

Open challenges. Designing permission models that are both expressive and usable is an unsolved problem. Overly restrictive policies degrade agent utility; overly permissive ones negate the purpose of governance. The community has not converged on a standard permission specification language that is portable across harness implementations. Open questions also persist about whether actions should be attributed to the human operator, the agent instance, or transient task identities (South et al., 2025).

9.2 Lifecycle Hooks

Permission models define what is allowed. Lifecycle hooks define when policy checks fire. Many harnesses expose hook points at each stage of the agent loop: before planning, before tool invocation, after tool execution, and before response delivery. These hooks allow governance logic to be injected without modifying the agent's core reasoning. Figure 14 situates the four hook points along a single tool-use cycle.

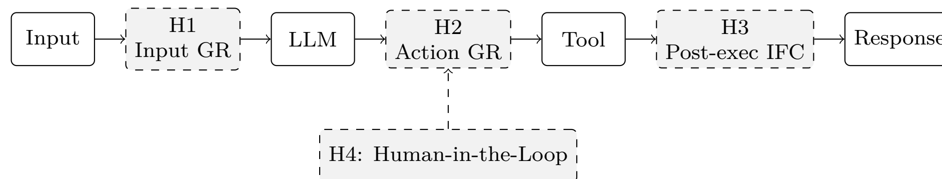

Figure 14: Four hook points along one tool-use cycle. Solid boxes are agent components; dashed boxes are governance hooks. H1 validates input before it reaches the LLM, H2 validates the proposed action before tool execution, H3 mediates information flow from tool output back into context, and H4 gates consequential actions on user approval.

Pre-execution hooks: input guardrails. Input guardrails validate data before it reaches the LLM. PromptShield (Jacob et al., 2024) and DataSentinel (Liu et al., 2025) train dedicated classifiers to detect prompt injection payloads in user inputs and retrieved content. Rule-based detectors are strict but brittle. Model-based detectors generalize better but are slower and susceptible to adaptive attacks (Andriushchenko et al., 2024; Nasr et al., 2025).

The next enforcement point occurs between the LLM's proposed action and the harness's tool execution.

Pre-invocation hooks: output guardrails and action validation. Before executing a tool call, output guardrails inspect the LLM's proposed action. ShieldAgent (Chen et al., 2025b) formalizes safety constraints as verifiable predicates and checks each action against them. In multi-agent systems, ControlValve (Jha et al., 2025) constrains the control-flow graph between agents and prevents unauthorized agent transitions, addressing control-flow hijacking attacks where one agent redirects execution to another.

After a tool executes, the returned data must also be inspected before re-entering the LLM context.

Post-execution hooks: information flow control and taint tracking. CaMeL (Debenedetti et al., 2025) implements capability-based information flow control. Every data value carries metadata tags that track its provenance, distinguishing trusted user input from untrusted web retrieval. A custom interpreter enforces that untrusted data cannot influence control-flow decisions. CaMeL separates planning over trusted context from processing over untrusted content, trading some flexibility for a stronger control/data-flow boundary on AgentDojo (Debenedetti et al., 2024).

A complementary enforcement mechanism places the human user in the loop.

Human-in-the-loop hooks. Several widely used coding agents, including Codex, Gemini CLI, Cursor, and OpenHands (Wang et al., 2025a), gate destructive or out-of-scope actions on explicit user approval. Three design dimensions shape these hooks: validation scope (which actions require approval), alert richness (how much context the user sees), and recurrence policy (allow-once versus allow-always). Felt et al. (2012) showed that only 17% of Android users paid attention to permission dialogs during app installation, and only 3% correctly answered comprehension questions about the permissions they had granted; agent approval dialogs may face analogous habituation and comprehension risks. Frequent requests lead users to approve dangerous actions reflexively, while infrequent requests leave coverage gaps.

Some systems treat hooks as a programmable enforcement substrate rather than as isolated classifiers. AgentSpec (Wang et al., 2026) defines rules with triggers, predicates, and enforcement actions at plan, tool-call, and response stages. AgentDoG (Liu et al., 2026a) shifts from single-action classification to trajectory-level diagnosis, organizing risks by source, failure mode, and consequence.

Open challenges. Across current systems, hook APIs remain heterogeneous. We are not aware of a widely adopted standard that defines a universal hook interface for third-party governance modules. Hooks also introduce latency, and the interaction effects between stacked hooks remain largely unstudied. A plausible failure mode is that an upstream sanitizer alters the signals that a downstream detector relies on, leaving the combined pipeline weaker than either component considered in isolation.

9.3 Component Hardening

Lifecycle hooks enforce governance at the harness level. Component hardening strengthens individual agent components, specifically the LLM and the tools it invokes, against their specific vulnerabilities. A harness benefits from hardened components because fewer malicious inputs survive to trigger governance hooks in the first place.

Model hardening. A root cause of prompt injection is that LLMs treat all input text with equal priority, making them unable to distinguish system instructions from untrusted data. Wallace et al. (2024) propose an instruction hierarchy that trains models to prioritize privileged instructions (system prompts) over lower-privileged ones (user messages, tool outputs). Models learn to align with lower-level instructions when they are consistent with higher-level constraints, and to ignore or refuse them when they conflict. SecAlign (Chen et al., 2025a) formulates the same defense goal as preference optimization over prompt-injected inputs, desirable responses, and undesirable responses.

These model-level defenses complement harness-level hooks. A hardened model rejects more injections at inference time, reducing the load on downstream guardrails. However, model hardening alone is insufficient: it addresses wrong instruction following in Kim et al.'s taxonomy but does not prevent unconstrained data flow or unauthorized actions, which require system-level enforcement (Kim et al., 2026; Wei et al., 2026).

Classifier-based runtime hardening. A complementary line of work hardens the model boundary without modifying the agent LLM itself, by deploying small auxiliary classifiers that screen inputs and outputs at runtime. Llama Guard (Inan et al., 2023) fine-tunes a separate model to classify both user prompts and assistant responses against a configurable safety taxonomy, returning per-category labels that the harness can route to allow, block, or escalate decisions. Such classifiers offer two practical properties that training-time hardening lacks: the taxonomy can be edited without retraining the agent model, and the same detector can be reused across heterogeneous LLM backends. The cost is the latency budget discussed in §9.2: each additional classifier adds a forward pass per tool-use cycle.

Model and classifier defenses harden the LLM boundary; the tool boundary requires its own protocol-level treatment.

Tool hardening and MCP security. The Model Context Protocol (MCP) (Anthropic, 2024c) has emerged as a common interface between agents and tools, but its initial specification lacks native security primitives. This gap has attracted both attacks and defenses, and an official specification response. Radosevich & Halloran (2025) demonstrate that widely used commercial LLMs can be coerced into using MCP tools to execute malicious code, establish remote access, and steal credentials. Their McpSafetyScanner uses a three-agent architecture (hacker, auditor, supervisor) to systematically discover such vulnerabilities. Trail of Bits (Trail of Bits, 2025) show that MCP servers can attack clients before a user ever invokes a tool, through poisoned tool descriptions that influence the LLM's behavior at registration time.

On the defense side, ETDI (Bhatt et al., 2025) extends MCP with cryptographically signed and versioned tool definitions, ensuring that any modification to a tool's code, schema, or permissions requires a new signed version. This prevents rug-pull attacks where a previously approved tool is silently updated to perform malicious actions.

Protocol-level hardening. Beyond individual components, SAFEFLOW (Li et al., 2025) introduces protocol-level enforcement for multi-agent systems. It combines fine-grained information flow control with transactional execution semantics, so a violating tool call or inter-agent message can be rolled back rather than propagated through shared state.

Supply chain risks. Agent governance must also address the tools and packages that agents invoke. Spracklen et al. (2025) analyze 576,000 code samples across 16 LLMs and find that open-source models hallucinate nonexistent package names at a rate of 21.7%. Attackers can register these hallucinated names on public repositories and inject malicious code, a technique termed "slopsquatting." ETDI's cryptographic signatures address part of this problem for MCP tools, but package managers and retrieval sources remain outside most agent governance stacks.

Open challenges. Model hardening and tool hardening address complementary threat surfaces, but no unified framework connects them. A hardened model may still invoke a compromised tool, and a signed tool definition does not prevent misuse by a jailbroken model. Composing these defenses into a coherent stack with quantifiable residual risk remains an open problem. As MCP adoption grows, its security extensions become natural targets for standardization, yet interoperability and certification remain largely unsettled.

9.4 Declarative Constitutions

Embedding governance logic directly in application code makes policies opaque, hard to audit, and difficult to update. A growing number of harnesses externalize governance rules into declarative configuration files, most commonly in YAML format. These files serve as a machine-readable constitution for the agent.

Training-time constitutions: Anthropic's Constitutional AI. Anthropic (Anthropic, 2026a) publishes a constitution organized as a four-tier priority hierarchy: safety (preserving human oversight), ethics (honesty, harm avoidance), compliance (Anthropic's guidelines), and helpfulness (user requests). The constitution distinguishes hard constraints, such as absolute prohibitions on chemical, biological, radiological, and nuclear (CBRN) assistance, from soft-coded defaults that operators may adjust within defined bounds. In agentic settings, this structure suggests a principal hierarchy in which Anthropic's policies override operator configurations, which in turn override end-user preferences.

Training-time constitutions operate by shaping the model's behavior during alignment and post-training. They shape model behavior at training time but are difficult to modify post-deployment and cannot be independently audited by the harness. A distinct approach externalizes governance rules into deployment-time configuration that the harness can read, validate, and enforce.

Deployment-time constitutions: YAML-based governance. The AutoHarness repository (AIMing Lab, 2026) encodes governance rules in a YAML file that specifies the pipeline mode (core, standard, or enhanced), risk classification patterns, allowed and denied tool patterns, token budget limits, and audit log destinations. This declarative style separates governance intent from execution logic. Non-developer stakeholders such as security teams and compliance officers can review and modify policies without touching agent code. Unlike training-time constitutions, YAML files can be updated, versioned, and diffed with standard tooling, enabling policy updates without retraining or redeploying the model.

The two layers differ operationally. Training-time constitutions are costly to modify and must be inferred through behavioral probes, while deployment-time YAML is directly readable and diff-reviewable. Training-time alignment shapes default behavior probabilistically and can be overridden by adversarial prompts (Andriushchenko et al., 2024); deployment-time rules instead execute as harness checks.

Pattern-based YAML rules suffice when policies reduce to literal triggers, but real deployments often require conditional logic over runtime state. Current YAML schemas do not standardize predicates over privilege escalation, host-based egress, session-level rate limits, or future actions. Between free-form YAML and hard-coded rules lies a third design point: structured policy DSLs.

Programmable policy languages. Progent's policy language (Shi et al., 2025a) supports boolean predicates, quantifiers, and environment references. It provides formal expressiveness while remaining readable. Formal-LLM (Li et al., 2024) goes further by encoding plan constraints as pushdown automata. This formalism can express sequential ordering constraints, prohibited tool combinations, and required approval gates at specific steps. VeriSafeAgent (Lee et al., 2025) formalizes user intent into a DSL over UI state transitions,

verifying that proposed GUI actions align with the user's task before execution. YAML constitutions admit ambiguity; formal specifications demand expertise.

Open challenges. No widely adopted standard schema exists for agent constitutions. Each harness tends to define its own YAML structure, making policies non-portable. Tooling to validate that a constitution is internally consistent (e.g., no contradictory allow/deny rules) and complete (e.g., no unhandled tool categories) appears limited in the current ecosystem. The interaction between training-time and deployment-time constitutions is particularly underexplored. Whether a YAML deny-rule can reliably override a behavior that RLHF has reinforced, and under what conditions the two layers conflict, remains an open question with practical security implications.

9.5 Audit Infrastructure

Governance requires accountability. Audit infrastructure records what the agent did, why it did it, and whether governance policies were respected. These records support post-hoc investigation, regulatory compliance, and continuous policy refinement.

Structured audit trails. AutoHarness (AIMing Lab, 2026) emits per-tool-call JSONL records that include the tool name, arguments, risk classification, permission decision, execution result, token cost, and wall-clock latency. These structured logs support both real-time dashboards and offline forensic analysis. SAGA (Syros et al., 2025) extends audit trails to multi-agent interactions by recording cryptographic tokens exchanged between agents, enabling provenance tracking across trust boundaries. Across these systems, a replayable audit record needs at least a trace identifier, principal identity, tool invocation, policy decision and version, execution result, resource cost, and an integrity hash over relevant inputs and outputs. Most current systems record only a subset of these fields, and few sign or hash records, leaving audit trails susceptible to log tampering by a compromised agent process.

Beyond recording, detecting governance violations in long-running agents requires automated analysis.

Anomaly detection: per-action versus trajectory level. Detection mechanisms split along the granularity at which they evaluate evidence. *Per-action* detectors classify each tool call in isolation against a learned or specified notion of risk: input and output guardrails (§9.2) operate in this regime, and AgentMonitor (Naihin et al., 2023) provides a lightweight log-and-flag implementation. This regime is cheap and easy to audit, but it cannot recognize attacks that distribute themselves across many individually benign actions, such as a slow exfiltration that reads one file per minute. *Trajectory-level* detectors evaluate sequences of actions jointly. AgentAuditor (Luo et al., 2025) combines behavioral pattern matching with LLM-based reasoning over full traces. SentinelAgent (He et al., 2025) models multi-agent communication as a temporal graph and flags anomalous interaction patterns. Trajectory-level detection catches multi-step attacks at the cost of higher latency and a less localized notion of which action triggered the alert, complicating both real-time intervention and post-hoc explanation. In the systems surveyed here, per-action checks are typically deployed inline, while trajectory-level analysis runs asynchronously over the audit log.

Beyond security, long-running agents also require cost and resource governance.

Cost and resource auditing. A runaway agent loop can exhaust API budgets quickly in loop-heavy workloads. The 2025 OWASP Top 10 for LLMs (OWASP Foundation, 2025) lists resource drain as a distinct risk category. AutoHarness (AIMing Lab, 2026) tracks per-call token consumption and enforces session-level budgets declaratively through its constitution. Cost-aware governance matters especially in multi-agent architectures where fan-out patterns can multiply API calls exponentially.

Tiered governance pipelines. An emerging design pattern integrates the preceding mechanisms (permissions, hooks, constitutions, and audit) into a single configurable pipeline. AutoHarness (AIMing Lab, 2026) exemplifies this pattern with three tiers. The tiers add progressively deeper checks, from parse–risk–permission–execute–audit loops to context enrichment, output validation, anomaly scoring, human escalation, and formal constraint verification. The tier is selected declaratively via the YAML constitution, so

governance overhead scales with deployment risk. Other production guidance exposes similar responsibilities as feature-level controls rather than named tiers (Shavit et al., 2023). Which abstraction is more usable remains an open empirical question.

Open challenges. Audit logs accumulate rapidly in long-running agents, raising challenges around storage, privacy, and signal-to-noise ratio. Logs may contain sensitive user data, API keys, or conversation content, creating tension between audit completeness and data protection requirements. The community also lacks standardized audit schemas, making cross-system analysis and regulatory reporting difficult.

9.6 Situating Governance in the Agent Security Landscape

Governance mechanisms respond to a concrete threat landscape. Recent evidence (Zhang et al., 2026b; Kim et al., 2026; Chen et al., 2026; Wei et al., 2026) from open agent-to-agent platforms further suggests that agent security is not limited to prompt injection or single-agent jailbreaks: social engineering, coordinated agent clusters, credential leakage, and platform-level engagement amplification can jointly shape the threat landscape. Kim et al. (2026) survey 128 papers and catalogue 51 attack methods and 60 defense methods across the agent stack. Chen et al. (2026) complement this with a software-engineering perspective, synthesizing 50 papers through a six-dimensional taxonomy and proposing a reference doctrine for secure-by-construction agent platforms. We use these frameworks to connect governance mechanisms to specific security risks and identify coverage gaps.

From design dimensions to governance mechanisms. Kim et al. identify seven design dimensions (Input Trust, Access Sensitivity, Workflow, Action, Memory, Tool, User Interface), each representing a flexibility spectrum. Greater flexibility expands the attack surface, and governance constrains that flexibility at runtime. Permission models bound tools and actions; lifecycle hooks inject checkpoints into workflows; constitutions externalize constraints; and audit infrastructure provides the feedback loop that reveals when constraints are too loose or too tight. Table 3 maps governance mechanisms to the seven risk categories (R1–R7) in Kim et al.'s taxonomy.

Table 3: Mapping governance mechanisms to the risk taxonomy of Kim et al. (2026).

Governance Mechanism	Risks Mitigated or Detected
Permission models and identity mgmt.	R1 (untrusted interfaces), R5 (data leakage), R6 (unauthorized actions)
Input guardrails	R1, R2 (wrong instruction following)
Output guardrails	R2, R4 (hallucination), R5, R6
Information flow control	R3 (unconstrained data flow), R5, R6
Component hardening	R1, R2, R4
Monitoring and audit	R5, R6, R7 (resource drain)
Human-in-the-loop	R2, R6
Privilege separation	R2, R3
Formal verification	R2, R6
Declarative constitution	Cross-cutting: configures all of the above

Defense gaps in real-world agents. Kim et al.'s case studies on six agent systems (Codex, Gemini CLI, OpenHands, Browser Use, Nanobrowser, Skyvern) reveal that no agent fully implements all defense categories. Information flow control, identity management, and formal verification are absent from every surveyed system. In all six cases, monitoring is partial: agents log tool calls but lack automated anomaly detection. The AutoGPT case study illustrates the consequence: downstream patches can address individual symptoms while leaving upstream input validation, information flow control, and policy composition underspecified.

Defense-in-depth and its limits. Kim et al. argue that agent security requires layered defenses that complement each other: input guardrails as first-line protection, output guardrails as last-line defense, information flow control and monitoring as continuous runtime protection, access control for authentication and authorization, and human-in-the-loop for critical decisions. However, layering is not free; as discussed in §9.2, poorly coordinated layers may interfere with each other, and the composability of independent governance hooks remains largely untested. In this framing, governance can serve as the orchestration layer that helps defense mechanisms cooperate rather than conflict.

Contextual security as a proposed extension. Kim et al. (2026) propose contextual security as a candidate fourth security goal for agentic systems, alongside the classical confidentiality–integrity–availability (CIA) triad. The proposal is recent and specific to their survey; it has not been adopted by standards bodies such as the National Institute of Standards and Technology (NIST) or by mainstream security textbooks, and we treat it here as a useful framing rather than as a settled definition. Conseca (Tsai & Bagdasarian, 2025) and CaMeL (Debenedetti et al., 2025) illustrate the engineering direction: context must be treated as governed state rather than as passive prompt material. Whether this warrants elevation to a standalone security goal remains unsettled.

9.7 Research Directions

Table 4 summarizes the state of governance across the systems discussed in this section.

Table 4: Governance feature coverage. ● = full support, ◐ = partial, ○ = absent.

System	Perm.	Hooks	Harden.	Constit.	Audit	Multi-Ag.
Codex	●	◐	○	○	◐	○
Gemini CLI	●	◐	○	○	◐	○
OpenHands	◐	●	○	○	◐	◐
AutoHarness	●	●	◐	●	●	◐
Progent	●	●	○	◐	○	○
CaMeL	◐	●	○	○	○	○
SAGA	●	◐	○	○	●	●
IsolateGPT	●	◐	○	○	○	●
AgentSpec	◐	●	○	●	○	○
SAFEFLOW	◐	●	◐	○	●	●

The sparsity visible in Table 4 suggests that governance infrastructure is often a practical bottleneck for safe deployment alongside model limitations. We identify eight open research directions, restricted here to governance-specific gaps and cross-referenced with the broader open problems in §12.

(1) Standardized policy and audit languages. Each harness defines its own YAML schema and proprietary DSL, fragmenting the governance ecosystem. A community-driven specification, analogous to what MCP (Anthropic, 2024c) is pursuing for tool interfaces, would enable portable policies, composable governance modules, and cross-harness audit interoperability.

(2) Formal governance guarantees. Current governance mechanisms remain limited in formal guarantees. Formal verification of agent behavior remains at an early stage (Li et al., 2024; Chen et al., 2025b; Lee et al., 2025). Extending these techniques to verify governance pipeline correctness, for instance proving that a constitution is internally consistent and covers all tool categories, remains underexplored.

(3) Adaptive governance. Static policies cannot anticipate every task context. Conseca (Tsai & Bagdasarian, 2025) demonstrates that LLM-generated policies can bridge this gap. The trustworthiness of machine-generated governance rules, and the question of how such policy generators should themselves be governed, remain open.

(4) Governance for long-horizon agents. Agents that execute multi-hour or multi-day tasks introduce context drift, session-spanning state, and evolving trust assumptions. Many current governance pipelines are designed for single-turn or short-session agents. Scaling them to long-horizon autonomy requires governance rules whose validity can be renewed, revoked, and audited over evolving sessions.

(5) Usable governance interfaces. Usable governance interfaces are necessary for deployment. The field needs human–computer interaction (HCI) research on permission UIs, audit dashboards, and constitution editors that are accessible to non-specialist users. Prior work on mobile permissions suggests a risk that approval dialogs will transfer poorly to the agentic setting (Felt et al., 2012), but agent-specific studies are still needed.

(6) Cross-layer governance coherence. Training-time alignment (constitutional AI), deployment-time configuration (YAML constitutions), and runtime enforcement (lifecycle hooks) operate at different layers with different fidelities. How these layers compose, and whether runtime governance can reliably override training-time dispositions, remains an open question.

(7) End-to-end supply chain governance. MCP security extensions such as ETDI (Bhatt et al., 2025) address tool integrity within a single harness. However, agents also depend on external packages, datasets, and retrieval sources whose provenance is difficult to verify. Package hallucination attacks (Spracklen et al., 2025) demonstrate that LLMs can inadvertently introduce supply chain compromises. Governance frameworks that span the full tool and data supply chain, from package repository to agent execution, remain limited.

(8) Unified adversarial benchmarks for governance stacks. Existing benchmarks target narrow threat classes, such as prompt injection, trajectory diagnosis, or MCP server vulnerabilities (Debenedetti et al., 2024; Liu et al., 2026a; Radosevich & Halloran, 2025). None evaluates a complete governance stack under a common adversary model with reproducible attack traces. A community benchmark would need to report defense effectiveness, false-positive rate, and overhead jointly rather than in isolation.

10 Cross-Cutting Concerns

The concerns collected in this section cut across multiple ETCLOVG layers and develop Claim 1 (the agent harness as an independent system layer, §1) by showing where layer-specific reasoning breaks down. The layer chapters isolate execution, tools, context, lifecycle, observability, evaluation, and governance so that each design surface can be described precisely. Production harnesses, however, fail most often at the interfaces among these surfaces.

Inter-layer interaction patterns. The seven layers are not independent. The execution environment (E) constrains which lifecycle and orchestration strategies (L) are practical; context management (C) affects evaluation reproducibility (V); and governance (G) imposes identity, permission, and audit constraints that span every other layer. A harness design should therefore be read as a dependency structure, not as a checklist of separable components.

The cost–quality–speed trilemma. Stronger evaluation (V), stricter governance (G), richer observability (O), and more faithful execution environments (E) generally increase cost and latency. Optimizing for speed often reduces diagnostic depth or safety margin, while optimizing for quality can make iteration too expensive for routine use. Real systems must choose which checks are synchronous, which run offline, and which failures justify costly recovery paths.

Standardization and ecosystem dynamics. Protocols such as MCP, ACP, and A2A reflect a pressure toward shared interfaces for tools, agents, and orchestration state. This pressure favors agent-agnostic infrastructure over single-agent integrations, but it also shifts responsibility to governance and observability: a standardized tool call is useful only if the surrounding harness can preserve provenance, permissions, cost, and failure evidence across systems.

Persistent ecosystem gaps. Across the corpus, five gaps recur at layer boundaries: cross-tool interoperability, cost attribution, failure recovery, multi-repository orchestration, and human-agent handoff. These

gaps motivate the synthesis in §11: the central question is no longer whether each layer has available tools, but whether the composed harness behaves as a reliable control system.

11 Cross-Layer Synthesis

This section consolidates the cross-cutting concerns of §10 into five system-level effects and is the central support for Claim 1 (§1). The purpose is to shift the discussion from component coverage to harness behavior: once execution, tools, context, orchestration, observability, evaluation, and governance are composed, their interactions create constraints that no single layer can solve alone.

11.1 Cost-Quality-Speed Trilemma

Harness reliability is constrained by a three-way tradeoff between cost, quality, and speed. Stronger sandboxes and more faithful environments improve safety and reproducibility but increase startup latency and infrastructure cost; richer context and memory policies can improve task continuity but consume tokens and introduce retrieval overhead; deeper evaluation and observability improve diagnosis but slow iteration and add storage, labeling, and trace-processing costs. Production systems therefore cannot treat quality as a scalar objective. They must decide which risks justify expensive controls, which checks can run asynchronously or in regression suites, and which telemetry is worth capturing at each stage of the agent lifecycle.

11.2 Capability-Control Tradeoff

More capable harnesses expose more authority to the agent, but every increase in authority expands the control problem. Larger tool menus broaden task coverage while increasing selection error and prompt-injection surface; persistent memory helps long-running tasks but creates provenance, staleness, and privacy risks; permissive sandboxes make autonomous execution useful but also enlarge the blast radius of misaligned or compromised actions. The capability-control tradeoff is therefore not a security add-on to an otherwise functional system. It is a design axis that links tool schemas, context policy, runtime permissions, identity, auditability, and human approval.

11.3 Harness Coupling Problem

Harness layers are coupled in ways that make local optimization fragile. The execution environment changes evaluation results by affecting package availability, reset semantics, latency, and failure modes; tool descriptions consume context budget and shape model behavior; observability traces become governance evidence only if identity and permission state are captured at the same granularity; and evaluation design feeds back into orchestration by rewarding some recovery loops and penalizing others. These couplings imply that harness changes should be tested as system changes. A prompt, tool, memory, sandbox, verifier, or monitor may look beneficial in isolation while degrading the whole rollout when combined with the rest of the control loop. The coupling problem also explains why agent scores cannot be cleanly attributed to the model without specifying the surrounding controller. Under the closed-loop framing, a change to context policy, tool schema, verifier, or recovery loop changes the controller C_H and therefore the measured behavior of the same model (Bölük, 2026b).

11.4 From Agent Frameworks to Agent Platforms

The ecosystem is moving from agent frameworks toward agent platforms. Frameworks package local abstractions such as agents, tools, memory stores, and execution loops; platforms add durable workspaces, managed sandboxes, identity, billing, observability, evaluation, governance, and human handoff across many runs and many users. This transition matters because long-running agents are no longer just programs that call models. They are operational systems that need tenancy, compliance, fault recovery, trace retention, and organizational ownership. As a result, the central design question shifts from "how do I build an agent?" to "how do I operate a fleet of agents whose actions remain inspectable and reversible over time?"

11.5 Open Research Agenda

The synthesis points to a research agenda centered on harnesses as adaptive control systems. The field needs benchmarks that vary harness interventions rather than only model weights, trace-native methods for attributing failures across layers, protocols for transferring state and responsibility among agents, tools, sandboxes, evaluators, and humans, and optimization methods that simplify harnesses as models improve. The next section turns these cross-layer effects into five open questions: how to harden and scale execution environments, maintain reliable state, diagnose failures from traces, standardize handoffs, and keep harnesses useful under changing model capability.

12 Open Problems and Future Directions

The open problems collected here follow from the binding-constraint thesis and the cross-layer synthesis, and they form the forward-looking part of the evidence for Claim 1 (§1). Rather than treating the seven ETCLOVG layers as independent component lists, this section asks where the whole harness remains scientifically under-specified. The central pattern is that agent harnesses are becoming long-running control systems, but the field still lacks mature answers for hardening the execution substrate, preserving state, diagnosing failures, transferring responsibility, and updating the harness as model capabilities change. We organize these gaps into five questions that cut across the taxonomy.

12.1 Hardening and Scaling Execution Environments

Execution environments are becoming the control boundary where security, scalability, and portability meet. SandboxEscapeBench documents that frontier models can exploit sandbox weaknesses under realistic configurations (Marchand et al., 2026), but defense work remains fragmented across systems with different threat models and evaluation protocols (Wu et al., 2025; Yan, 2025). At the same time, the one-container-per-task pattern strains large-scale training and evaluation, where tens of thousands of parallel trajectories need cheap reset and replay; SWE-World points toward Docker-free surrogate environments, but the fidelity of learned transitions relative to real execution remains unresolved (Sun et al., 2026). Even deployment portability is not a solved engineering detail: Docker-based sandboxes inherit Linux-kernel assumptions, while macOS, Windows, browser, desktop, and hybrid-cloud settings expose different isolation and reproducibility constraints.

The open problem is to make the runtime substrate both measurable and composable. Future harnesses need common security evaluations for prompt injection, goal misalignment, and compositional amplification; cost models that decide when to use containers, microVMs, OS-level permission boundaries, full desktop VMs, browser environments, or learned surrogates; and portability layers that preserve semantics across self-hosted, cloud, and hybrid deployments. The bundle-versus-compose split between framework-integrated runtimes (§3.2.4) and sandbox abstractions (§3.2.7) should be treated as an empirical design question rather than a product preference. Standards such as MCP may reduce composition cost, but only if the tool, governance, and observability layers expose enough state to keep runtime choices auditable, recoverable, and safe.

12.2 Maintaining Reliable State in Long-Running Agents

The deepest context problem is not simply how to fit more tokens into a prompt, but how to keep an agent's working state aligned with the true task state over long horizons. Long-running coding, research, and operations agents repeatedly summarize, retrieve, compact, and externalize information; every such operation can delete constraints, distort priorities, or preserve stale assumptions. Recent context-engineering work treats compaction, tool-result clearing, retrieval, and prompt-cache-aware ordering as practical mechanisms for managing limited context windows (Anthropic Applied AI Team, 2025; Anthropic, 2025c; OpenAI, 2026b). However, context rot and memory benchmarks show that longer inputs and richer memory stores do not automatically imply better task-state tracking (Hong et al., 2025; Tan et al., 2025; He et al., 2026).

A principled research agenda should therefore recast context management as *state estimation*. The open question is whether we can characterize how much task-relevant information is lost at each compression, retrieval, or forgetting step, and whether we can bound the divergence between the agent's internal state and the real state of the task. Recent surveys formalize agent memory as a write–manage–read loop and identify policy-learned management as an emerging mechanism family (Zhang et al., 2025; Du, 2026). Future systems need uncertainty-aware summaries, provenance for remembered facts, contradiction handling, explicit staleness markers, and recovery procedures that let an agent reconstruct missing state from durable artifacts rather than trusting its own compressed history. This also suggests a tighter link between memory and evaluation: memory policies should be judged not only by recall accuracy, but by whether they prevent downstream action errors in multi-session tasks.

12.3 Diagnosing Failures from Agent Traces

Agent evaluation is still too often final-score-centric: a run passes or fails, and the final number is treated as evidence about model quality. For harness engineering, this is insufficient because a failed rollout may originate from model reasoning, a misleading tool schema, sandbox misconfiguration, stale context, flaky tests, benchmark ambiguity, judge instability, or orchestration loops. Anthropic's analysis of agentic coding evaluations shows that infrastructure settings can measurably shift benchmark scores (Anthropic, 2026a), and recent work on randomness in agentic evals argues that single-run pass rates can hide substantial variance (Bjarnason et al., 2026). The evaluation layer must therefore be studied as a measurement instrument, not merely used as a leaderboard generator.

The next step is trace-native evaluation: traces should become the primary object from which systems compute outcome scores, trajectory quality, failure attribution, and regression tests. Observability systems already capture spans, tool calls, costs, retries, exceptions, and intermediate messages (OpenTelemetry, 2026; AlSayyad et al., 2026; Koc et al., 2025), but these traces are often disconnected from evaluation pipelines. LangChain's 2026 survey reports that 89% of teams use observability while only 52.4% run offline evaluations (LangChain, 2026a); this gap means teams can see what agents did without systematically judging whether the behavior was correct. Future work should close this loop by converting anomalous production traces into regression cases, computing trajectory metrics directly over spans, and feeding diagnostic signals back into prompt, tool, context, and orchestration changes. Reflexion showed that agents can learn from their own traces in short-horizon settings (Shinn et al., 2023); extending this idea to long-running, multi-session harnesses remains open.

12.4 Standard Handoffs Across Agents, Tools, and Humans

Modern harnesses increasingly distribute work across planners, subagents, tools, sandboxes, evaluators, and humans, but the interfaces between these actors remain ad hoc. There are emerging local standards: MCP standardizes tool access, A2A targets inter-agent communication, and OpenTelemetry provides a general substrate for traces (Model Context Protocol, 2025b; A2A Project, 2025; OpenTelemetry, 2026; Ehtesham et al., 2025). What is missing is a cross-layer handoff contract. When a planner hands work to an executor, an agent calls a tool, a subagent returns control, or a system escalates to a human, the handoff should transfer not only a text summary but also intent, constraints, permissions, artifacts, provenance, budget state, risk level, trace history, and unresolved decisions.

This problem is partly technical and partly institutional. OpenAI's Symphony frames the issue tracker and repository as a control plane for agent work, while Anthropic's long-running-agent harnesses emphasize durable progress artifacts and clean handoff state (Kotliarskyi et al., 2026; Anthropic, 2025d; 2026b). Governance work reaches the same conclusion from the opposite direction: agent identity, delegation, permission manifests, and auditability are needed before agents can safely act on behalf of users across systems (South et al., 2025; Marro et al., 2025; Syros et al., 2025). The open problem is to define handoff protocols that are rich enough for safety and recovery, but simple enough for broad adoption. Such protocols should make responsibility explicit: who authorized the action, which state was transferred, which evidence supports the current plan, what the receiver is allowed to do, and when control must return to another agent or a human.

12.5 Keeping Harnesses Useful as Models Improve

Harness design should not be assumed to move monotonically toward more scaffolding. Every wrapper, reset, verifier, planner, memory rule, and permission gate encodes an assumption about what the model cannot do reliably on its own. As model capabilities change, harness interventions should be re-estimated rather than assumed to remain beneficial. A factorial model-by-harness evaluation can reveal when an intervention improves all models, helps only specific model families, or reverses model rankings (Bölük, 2026b). Anthropic reports a concrete version of this pattern in long-running application development: context resets that were useful for one model became dispensable for a stronger model, and removing them reduced cost without degrading quality (Anthropic, 2026c). OpenAI similarly frames harness engineering as a discipline of keeping human attention, repository state, and agent execution aligned rather than merely adding more scaffolding (OpenAI, 2026a;b).

This creates a meta-engineering agenda: harnesses need mechanisms for optimizing and simplifying themselves. Meta-Harness shows that prompts, tools, and control loops can be searched as part of the optimization target rather than fixed by hand (Lee et al., 2026), while Natural-Language Agent Harnesses make harness modules explicit and ablatable (Pan et al., 2026). Production observability and cost systems such as TensorZero, Axon, and AgentOps point toward budget-aware harness operation (TensorZero, 2026; harshkedia177, 2026; AgentOps AI, 2026), but the research problem is broader than cost minimization. Future systems should identify which interventions are causally responsible for quality, safety, or reliability; run shadow-mode or A/B tests across harness variants; and optimize under joint quality–latency–cost–risk constraints. A central risk is benchmark overfitting: a harness that optimizes itself only against a narrow suite may become brittle. The more durable goal is adaptive simplification, where the harness continuously asks which controls are still necessary as tasks, tools, and model capabilities change.

13 Conclusion

This survey treats the agent harness as an independent engineering surface and argues that infrastructure quality, not model capability alone, sets the ceiling on real-world agent reliability. Around this binding-constraint thesis we develop three claims. The seven-layer ETCLOVG taxonomy separates Observability and Governance from Lifecycle Hooks and reflects how production teams already organize their tooling and ownership. A mapping of 170+ open-source projects onto the taxonomy gives the most extensive ecosystem snapshot to date and surfaces adoption patterns, coverage gaps, and emerging design principles. A three-phase engineering evolution from prompt to context to harness engineering, together with a cross-layer synthesis covering the cost–quality–speed trilemma, the capability–control tradeoff, and the harness coupling problem, situates the harness within a broader engineering trajectory. Our analysis has limits. The corpus is biased toward English-language, GitHub-visible, open-source projects, and toward the coding-agent ecosystem; extending it to closed-source production systems and to non-coding agent ecosystems would sharpen the empirical picture. The taxonomy itself is descriptive: turning ETCLOVG into a normative framework that can guide harness design decisions, rather than only classify them, is the natural next step we hope this survey will encourage.

References

A2A Project. Agent2agent (A2A) protocol. GitHub repository, 2025. URL https://github.com/a2aproject/A2A. Accessed May 14, 2026.

General Action. Emdash. https://github.com/generalaction/emdash, 2026. GitHub repository.

Aden. Openhive: Multi-agent swarms deployed in minutes for long-session business process. https://github.com/aden-hive/hive, 2026. GitHub repository.

affaan-m. everything-claude-code. https://github.com/affaan-m/everything-claude-code, 2025. GitHub repository.

Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pp. 419–434, 2020.

Agent Infra. agent-infra sandbox: All-in-one sandbox for AI agents. https://github.com/agent-infra/sandbox, 2024. Open-source software.

agentmd. AGENT.md: The universal agent configuration file. GitHub repository, 2025. URL https://github.com/agentmd/agent.md. Accessed May 14, 2026.

AgentOps AI. AgentOps: Python SDK for AI agent monitoring, LLM cost tracking, benchmarking, and observability. https://github.com/AgentOps-AI/agentops, 2026.

agentsmd. AGENTS.md. GitHub repository, 2025. URL https://github.com/agentsmd/agents.md. Accessed May 14, 2026.

Aider-AI. aider: Ai pair programming in your terminal. https://github.com/aider-ai/aider, 2025. GitHub repository.

AIMing Lab. AutoHarness: Automated harness engineering for AI agents. https://github.com/aiming-lab/AutoHarness, 2026. Version 0.1.0, released April 1, 2026.

Ken Aizawa. Writing effective tools for agents – with agents. Anthropic Engineering Blog, September 2025. URL https://www.anthropic.com/engineering/writing-tools-for-agents.

Alibaba. OpenSandbox: A general-purpose sandbox platform for AI applications. https://github.com/alibaba/OpenSandbox, 2026. Open-source software.

Adam AlSayyad, Kelvin Yuxiang Huang, and Richik Pal. Agenttrace: A structured logging framework for agent system observability. In *LLM-based Multi-Agent Systems: Towards Responsible, Reliable, and Scalable Agentic Systems*, 2026.

Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Jailbreaking leading safety-aligned llms with simple adaptive attacks. *arXiv preprint arXiv:2404.02151*, 2024.

Anomaly. Opencode: The open source ai coding agent. https://github.com/anomalyco/opencode, 2025. GitHub repository.

Anthropic. Building effective agents. Anthropic Engineering Blog, December 2024a. URL https://www.anthropic.com/engineering/building-effective-agents.

Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku. https://www.anthropic.com/news/3-5-models-and-computer-use, 2024b. Blog post.

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, November 2024c.

Anthropic. Claude code. https://github.com/anthropics/claude-code, 2025. GitHub repository.

Anthropic. Claude Code documentation. https://docs.claude.com/en/docs/claude-code/overview, 2025a. Documentation.

Anthropic. Beyond permission prompts: Making Claude Code more secure and autonomous. https://www.anthropic.com/engineering/claude-code-sandboxing, October 2025b. Blog post.

Anthropic. Context management: Tool result clearing and compaction. https://www.anthropic.com/news/context-management, 2025c. News article.

Anthropic. Effective harnesses for long-running agents. Anthropic Engineering Blog, 2025d. URL https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents. Accessed May 14, 2026.

Anthropic. How we built our multi-agent research system. https://www.anthropic.com/engineering/multi-agent-research-system, 2025e. Blog post.

Anthropic. Code execution with MCP: Building more efficient AI agents. Anthropic Engineering Blog, November 2025f. URL https://www.anthropic.com/engineering/code-execution-with-mcp. Accessed May 14, 2026.

Anthropic. Anthropic sandbox runtime: A lightweight sandboxing tool for enforcing filesystem and network restrictions at the OS level. https://github.com/anthropic-experimental/sandbox-runtime, 2025g. Research preview and npm package anthropic-ai/sandbox-runtime.

Anthropic. Claude's constitution. https://www.anthropic.com/constitution, January 2026a. Published January 22, 2026; released under CC0 1.0.

Anthropic. Demystifying evals for AI agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents, 2026b. Engineering blog.

Anthropic. Harness design for long-running application development. https://www.anthropic.com/engineering/harness-design-long-running-apps, 2026c. Blog post.

Anthropic. Quantifying infrastructure noise in agentic coding evals. Anthropic Engineering Blog, 2026a. URL https://www.anthropic.com/engineering/infrastructure-noise.

Anthropic. Scaling managed agents: Decoupling the brain from the hands. Anthropic Engineering Blog, 2026b. URL https://www.anthropic.com/engineering/managed-agents.

Anthropic Applied AI Team. Effective context engineering for AI agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025. Blog post.

Arize AI. OpenInference: OpenTelemetry-based auto-instrumentation and semantic conventions for AI applications. https://github.com/Arize-ai/openinference, 2026a.

Arize AI. Phoenix: AI observability and evaluation. https://github.com/Arize-ai/phoenix, 2026b.

Hammad Atta, Muhammad Zeeshan Baig, Yasir Mehmood, Nadeem Shahzad, Ken Huang, Muhammad Aziz Ul Haq, Muhammad Awais, and Kamal Ahmed. Qsaf: A novel mitigation framework for cognitive degradation in agentic ai. *arXiv preprint arXiv:2507.15330*, 2025.

Lei Ba, Qinbin Li, and Songze Li. Ciber: A comprehensive benchmark for security evaluation of code interpreter agents. *arXiv preprint arXiv:2602.19547*, 2026.

Fu Bang. Gptcache: An open-source semantic cache for llm applications enabling faster answers and cost savings. In *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, pp. 212–218, 2023.

Manish Bhatt, Vineeth Sai Narajala, and Idan Habler. Etdi: Mitigating tool squatting and rug pull attacks in model context protocol (mcp) by using oauth-enhanced tool definitions and policy-based access control. In *2025 Cyber Awareness and Research Symposium (CARS)*, pp. 1–6. IEEE, 2025.

Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, Leo Gao, Jonathan Tow, Baber Abbasi, Alham Fikri Aji, Pawan Sasanka Ammanamanchi, Sidney Black, Jordan Clive, et al. Lessons from the trenches on reproducible evaluation of language models. *arXiv preprint arXiv:2405.14782*, 2024.

Bjarni Haukur Bjarnason, André Silva, and Martin Monperrus. On randomness in agentic evals. *arXiv preprint arXiv:2602.07150*, 2026. URL https://arxiv.org/abs/2602.07150.

Bloop-AI. Vibe-kanban. https://github.com/BloopAI/vibe-kanban, 2026. GitHub repository.

Birgitta Böckeler. Harness engineering for coding agent users. Martin Fowler website, April 2026. URL https://martinfowler.com/articles/harness-engineering.html.

Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault L De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. Workarena++: Towards compositional planning and reasoning-based common knowledge work tasks. *Advances in Neural Information Processing Systems*, 37:5996–6051, 2024.

Can Bölük. I improved 15 LLMs at coding in one afternoon. Only the harness changed., February 2026a. URL https://blog.can.ac/2026/02/12/the-harness-problem/.

Can Bölük. I improved 15 LLMs at coding in one afternoon. Only the harness changed. https://blog.can.ac/2026/02/12/the-harness-problem/, 2026b. Blog post.

Hugo Bowne-Anderson and Jeff Huber. Harness engineering: Why agent context isn't enough. https://hugobowne.substack.com/p/harness-engineering-why-agent-context, 2026. Blog post.

Christoph Bühler, Matteo Biagiola, Luca Di Grazia, and Guido Salvaneschi. Securing ai agent execution. *arXiv preprint arXiv:2510.21236*, 2025.

Bytedance. Deerflow - 2.0. https://github.com/bytedance/deer-flow, 2026. GitHub repository.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Shiping Chen, Qin Wang, Guangsheng Yu, Xu Wang, and Liming Zhu. Clawed and dangerous: Can we trust open agentic systems? *arXiv preprint arXiv:2603.26221*, 2026.

Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, David Wagner, and Chuan Guo. Secalign: Defending against prompt injection with preference optimization. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2833–2847, 2025a.

Zhaorun Chen, Mintong Kang, and Bo Li. Shieldagent: Shielding agents via verifiable safety policy reasoning. *arXiv preprint arXiv:2503.22738*, 2025b.

Daixuan Cheng, Shaohan Huang, Yuxian Gu, Huatong Song, Guoxin Chen, Li Dong, Wayne Xin Zhao, Ji-Rong Wen, and Furu Wei. Llm-in-sandbox elicits general agentic intelligence. *arXiv preprint arXiv:2601.16206*, 2026.

De Chezelles, Thibault Le Sellier, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F Xu, Siva Reddy, Quentin Cappart, et al. The browsergym ecosystem for web agent research. *arXiv preprint arXiv:2412.05467*, 2024.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Hongcheol Cho, Ryangkyung Kang, and Youngeun Kim. SkillRet: A large-scale benchmark for skill retrieval in LLM agents. *arXiv preprint arXiv:2605.05726*, 2026. URL https://arxiv.org/abs/2605.05726.

Comet ML. Opik: Open-source LLM evaluation, observability, and optimization platform. https://github.com/comet-ml/opik, 2026.

Confident AI. DeepEval: The LLM evaluation framework. https://github.com/confident-ai/deepeval, 2026. Open-source software.

context-space. context-space: Context engineering infrastructure. https://github.com/context-space/context-space, 2025. GitHub repository.

Cua. Cua: Open-source infrastructure for computer-use agents. https://github.com/trycua/cua, 2024. Open-source software.

Daytona Platforms, Inc. Daytona: Secure and elastic infrastructure for running AI-generated code. https://github.com/daytonaio/daytona, 2024. Open-source software.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in Neural Information Processing Systems*, 37:82895–82920, 2024.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025.

Peng Ding. ToolRegistry: A protocol-agnostic tool management library for function-calling LLMs. *arXiv preprint arXiv:2507.10593*, 2025. URL https://arxiv.org/abs/2507.10593.

Docker, Inc. Docker sandboxes: A new approach for coding agent safety. https://www.docker.com/blog/docker-sandboxes-a-new-approach-for-coding-agent-safety/, 2025. Blog post.

Herman Zvonimir Došilović. Judge0: Robust, fast, scalable, and sandboxed open-source online code execution system. https://github.com/judge0/judge0, 2024. Open-source software.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, et al. Workarena: How capable are web agents at solving common knowledge work tasks? *arXiv preprint arXiv:2403.07718*, 2024.

Pengfei Du. Memory for autonomous llm agents: Mechanisms, evaluation, and emerging frontiers. *arXiv preprint arXiv:2603.07670*, 2026.

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale API calls. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11812–11829. PMLR, 2024. URL https://proceedings.mlr.press/v235/du24h.html.

E2B. E2B: Code interpreting for AI apps. GitHub repository, 2024. URL https://github.com/e2b-dev/E2B. Accessed May 14, 2026.

Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. A survey of agent interoperability protocols: Model context protocol (MCP), agent communication protocol (ACP), agent-to-agent protocol (A2A), and agent network protocol (ANP). *arXiv preprint arXiv:2505.02279*, 2025. URL https://arxiv.org/abs/2505.02279.

Shahul Es, Jithin James, Luis Espinosa Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation. In *Proceedings of the 18th conference of the european chapter of the association for computational linguistics: system demonstrations*, pp. 150–158, 2024.

Xiang Fei, Xiawu Zheng, and Hao Feng. MCP-zero: Active tool discovery for autonomous LLM agents. *arXiv preprint arXiv:2506.01056*, 2025. URL https://arxiv.org/abs/2506.01056.

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pp. 1–14, 2012.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, et al. A framework for few-shot language model evaluation. *Zenodo*, 2021.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1):32, 2023.

GitHub. Github agentic workflows. https://github.com/github/gh-aw, 2026. GitHub repository.

GitHub. Searching for repositories. https://docs.github.com/en/search-github/searching-on-github/searching-for-repositories, 2026. GitHub Docs, accessed April 29, 2026.

Google. gVisor: The container security platform. https://github.com/google/gvisor, 2018. Open-source software.

Google. Gemini cli: An open-source ai agent that brings the power of gemini directly into your terminal. https://github.com/google-gemini/gemini-cli, 2025. GitHub repository.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pp. 79–90, 2023.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *The Innovation*, 2024.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11143–11156, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.664. URL https://aclanthology.org/2024.findings-acl.664/.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware llm reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24842–24855, 2025.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. *Advances in Neural Information Processing Systems*, 36:45870–45894, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/8fd1a81c882cd45f64958da6284f4a3f-Abstract-Conference.html.

harshkedia177. Axon. https://github.com/harshkedia177/axon, 2026. GitHub repository.

Xu He, Di Wu, Yan Zhai, and Kun Sun. Sentinelagent: Graph-based anomaly detection in multi-agent systems. *arXiv preprint arXiv:2505.24201*, 2025.

Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

Helicone. Helicone: Open-source LLM observability platform for developers. https://github.com/Helicone/helicone, 2026.

Yeachan Heo. oh-my-claudecode. https://github.com/yeachan-heo/oh-my-claudecode, 2026. GitHub repository.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale. *arXiv preprint arXiv:2502.13681*, 2025a.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. *arXiv preprint arXiv:2507.05257*, 2025b.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. MetaTool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023. URL https://arxiv.org/abs/2310.03128.

Hugging Face. huggingface/smolagents. GitHub repository, 2026. URL https://github.com/huggingface/smolagents. Accessed May 14, 2026.

Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, et al. Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674*, 2023.

Dennis Jacob, Hend Alzahrani, Zhanhao Hu, Basel Alomair, and David Wagner. Promptshield: Deployable detection for prompt injection attacks. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy*, pp. 341–352, 2024.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Rishi Jha, Harold Triedman, Justin Wagle, and Vitaly Shmatikov. Breaking and fixing defenses against control-flow hijacking in multi-agent systems. *arXiv preprint arXiv:2510.17276*, 2025.

Menglin Jia, Luming Tang, Bor-Chun Chen, Claire Cardie, Serge Belongie, Bharath Hariharan, and Ser-Nam Lim. Visual prompt tuning. In *European conference on computer vision*, pp. 709–727. Springer, 2022.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *12th International Conference on Learning Representations, ICLR 2024*, 2024.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, et al. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation. *arXiv preprint arXiv:2510.11977*, 2025.

Andrej Karpathy. Context engineering. https://x.com/karpathy/status/1937902205765607626, 2025. X post.

Kata Containers Project. Kata Containers: The speed of containers, the security of VMs. https://katacontainers.io/, 2017. Open-source OCI-compatible runtime using lightweight VMs.

Juhee Kim, Xiaoyuan Liu, Zhun Wang, Shi Qiu, Bo Li, Wenbo Guo, and Dawn Song. The attack and defense landscape of agentic ai: A comprehensive survey. *arXiv preprint arXiv:2603.11088*, 2026.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 2024. URL https://proceedings.mlr.press/v235/kim24y.html.

David Kimai. Context-engineering: A first-principles handbook. https://github.com/davidkimai/Context-Engineering, 2025. GitHub repository.

Vincent Koc, Jacques Verre, Douglas Blank, and Abigail Morgan. Mind the metrics: Patterns for telemetry-aware in-IDE AI application development using the model context protocol (MCP). *arXiv preprint arXiv:2506.11019*, 2025. URL https://arxiv.org/abs/2506.11019. arXiv:2506.11019.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Alex Kotliarskyi, Victor Zhu, and Zach Brock. An open-source spec for Codex orchestration: Symphony. https://openai.com/index/open-source-codex-orchestration-symphony/, April 2026. OpenAI engineering blog.

Muratcan Koylan. Agent skills for context engineering. https://github.com/muratcankoylan/Agent-Skills-for-Context-Engineering, 2025. GitHub repository.

Kubernetes SIG Apps. agent-sandbox: A Kubernetes CRD and controller for isolated, stateful, singleton workloads. https://github.com/kubernetes-sigs/agent-sandbox, 2025. Open-source software.

Laminar AI. Laminar (lmnr): Open-source observability platform purpose-built for AI agents. https://github.com/lmnr-ai/lmnr, 2026.

LangChain. Evaluating deep agents: Our learnings. LangChain Blog, December 2025a. URL https://www.langchain.com/blog/evaluating-deep-agents-our-learnings.

LangChain. Evaluating deep agents: Our learnings. https://www.langchain.com/blog/evaluating-deep-agents-our-learnings, December 2025b. Engineering blog.

LangChain. langchain-sandbox: Safely run untrusted Python code using Pyodide and Deno. https://github.com/langchain-ai/langchain-sandbox, 2025c. Open-source software.

LangChain. Langgraph: Low-level orchestration framework for building stateful agents. https://github.com/langchain-ai/langgraph, 2026a. GitHub repository.

LangChain. deepagents. https://github.com/langchain-ai/deepagents, 2026b. GitHub repository.

LangChain. State of agent engineering. https://www.langchain.com/state-of-agent-engineering, 2026a. Survey report.

LangChain. The anatomy of an agent harness. https://www.langchain.com/blog/the-anatomy-of-an-agent-harness, 2026b. Blog post.

LangChain. langchain-ai/langchain. GitHub repository, 2026c. URL https://github.com/langchain-ai/langchain. Accessed May 14, 2026.

Langfuse. Langfuse: Open source LLM engineering platform. https://github.com/langfuse/langfuse, 2026.

Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. Verisafe agent: Safeguarding mobile gui agent via logic-based action verification. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, pp. 817–831, 2025.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023a.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, Singapore, 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

Peiran Li, Xinkai Zou, Zhuohang Wu, Ruifeng Li, Shuo Xing, Hanwen Zheng, Zhikai Hu, Yuping Wang, Haoxi Li, Qin Yuan, et al. Safeflow: A principled protocol for trustworthy and transactional autonomous agent systems. *arXiv preprint arXiv:2506.07564*, 2025.

Zelong Li, Wenyue Hua, Hao Wang, He Zhu, and Yongfeng Zhang. Formal-llm: Integrating formal language and natural language for controllable llm-based agents. *arXiv preprint arXiv:2402.00798*, 2024.

Dongrui Liu, Qihan Ren, Chen Qian, Shuai Shao, Yuejin Xie, Yu Li, Zhonghao Yang, Haoyu Luo, Peng Wang, Qingyu Liu, et al. Agentdog: A diagnostic guardrail framework for ai agent safety and security. *arXiv preprint arXiv:2601.18491*, 2026a.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023a.

Xunzhuo Liu, Bowei He, Xue Liu, Andy Luo, Haichen Zhang, and Huamin Chen. Dual-pool token-budget routing for cost-efficient and reliable llm serving. *arXiv preprint arXiv:2604.08075*, 2026b.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 2511–2522, 2023b.

Yupei Liu, Yuqi Jia, Jinyuan Jia, Dawn Song, and Neil Zhenqiang Gong. Datasentinel: A game-theoretic detection of prompt injection attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 2190–2208. IEEE, 2025.

Jiaying Lu, Bo Pan, Jieyi Chen, Yingchaojie Feng, Jingyuan Hu, Yuchen Peng, and Wei Chen. Agentlens: Visual analysis for agent behaviors in llm-based autonomous systems. *IEEE Transactions on Visualization and Computer Graphics*, 2024.

Hanjun Luo, Shenyu Dai, Chiming Ni, Xinfeng Li, Guibin Zhang, Kun Wang, Tongliang Liu, and Hanan Salam. Agentauditor: Human-level safety and security evaluation for llm agents. *arXiv preprint arXiv:2506.00641*, 2025.

Manus Team. Context engineering for AI agents: Lessons from building manus. https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus, 2025. Blog post.

Rahul Marchand, Art O Cathain, Jerome Wynne, Philippos Maximos Giavridis, Sam Deverett, John Wilkinson, Jason Gwartz, and Harry Coppock. Quantifying frontier llm capabilities for container sandbox escape. *arXiv preprint arXiv:2603.02277*, 2026.

Samuele Marro, Alan Chan, Xinxing Ren, Lewis Hammond, Jesse Wright, Gurjyot Wanga, Tiziano Piccardi, Nuno Campos, Tobin South, Jialin Yu, et al. Permission manifests for web agents. *arXiv preprint arXiv:2601.02371*, 2025.

Meirtz. Awesome context engineering. https://github.com/Meirtz/Awesome-Context-Engineering, 2025. GitHub repository.

Qianyu Meng, Yanan Wang, Liyi Chen, Qimeng Wang, Chengqiang Lu, Wei Wu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. *Preprints*, 2026.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Microsoft. Autogen. https://github.com/microsoft/autogen, 2025. GitHub repository.

Microsoft. Semantic kernel. https://github.com/microsoft/semantic-kernel, 2026. GitHub repository.

Microsoft. microsoft/semantic-kernel. GitHub repository, 2026. URL https://github.com/microsoft/semantic-kernel. Accessed May 14, 2026.

mindfold-ai. Trellis: Multi-platform coding-agent workflow framework. https://github.com/mindfold-ai/Trellis, 2025. GitHub repository.

MLflow. MLflow: An open-source platform for the machine learning lifecycle. https://github.com/mlflow/mlflow, 2026.

Modal Labs. Modal: Serverless cloud platform for AI and data workloads. https://modal.com, 2024. Cloud platform.

Model Context Protocol. modelcontextprotocol/servers. GitHub repository, 2025a. URL https://github.com/modelcontextprotocol/servers. Accessed May 14, 2026.

Model Context Protocol. Model context protocol specification (2025-06-18). Specification, 2025b. URL https://modelcontextprotocol.io/specification/2025-06-18. Version 2025-06-18; accessed May 14, 2026.

Model Context Protocol. modelcontextprotocol/modelcontextprotocol. GitHub repository, 2025c. URL https://github.com/modelcontextprotocol/modelcontextprotocol. Accessed May 14, 2026.

Dany Moshkovich and Sergey Zeltyn. Taming uncertainty via automation: Observing, analyzing, and optimizing agentic ai systems. *arXiv preprint arXiv:2507.11277*, 2025.

Dany Moshkovich, Hadar Mulian, Sergey Zeltyn, Natti Eder, Inna Skarbovsky, and Roy Abitbol. Beyond black-box benchmarking: Observability, analytics, and optimization of agentic systems. *arXiv preprint arXiv:2503.06745*, 2025.

Mozilla AI. cq: An open standard for shared agent learning. https://github.com/mozilla-ai/cq, 2025. GitHub repository.

Hadar Mulian, Sergey Zeltyn, Ido Levy, Liane Galanti, Avi Yaeli, and Segev Shlomov. Agentfixer: From failure detection to fix recommendations in llm agentic systems. *arXiv preprint arXiv:2603.29848*, 2026.

Taylor Mullen and Ryan J. Salva. Gemini CLI: An open-source AI agent that brings Gemini into your terminal. Google Blog, June 2025. URL https://blog.google/technology/developers/introducing-gemini-cli-open-source-ai-agent/. Software: https://github.com/google-gemini/gemini-cli.

Silen Naihin, David Atkinson, Marc Green, Merwane Hamadi, Craig Swift, Douglas Schonholtz, Adam Tauman Kalai, and David Bau. Testing language model agents safely in the wild. *arXiv preprint arXiv:2311.10538*, 2023.

Yohei Nakajima. BabyAGI. GitHub repository, 2023. URL https://github.com/yoheinakajima/babyagi.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, et al. The attacker moves second: Stronger adaptive attacks bypass defenses against llm jailbreaks and prompt injections. *arXiv preprint arXiv:2510.09023*, 2025.

Northflank. Northflank: Deploy microservices, jobs, and databases. https://northflank.com, 2024. Cloud platform.

NVIDIA. Sandboxing agentic AI workflows with WebAssembly. https://developer.nvidia.com/blog/sandboxing-agentic-ai-workflows-with-webassembly/, December 2024. Blog post.

OpenAI. Code interpreter: Allow models to write and run Python in a sandboxed environment. https://platform.openai.com/docs/guides/tools-code-interpreter, 2023. API documentation.

OpenAI. Codex: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025. GitHub repository.

OpenAI. Introducing codex. OpenAI Blog, May 2025. URL https://openai.com/index/introducing-codex/.

OpenAI. Openai agents sdk. https://github.com/openai/openai-agents-python, 2026a. GitHub repository.

OpenAI. Symphony. https://github.com/openai/symphony, 2026b. GitHub repository.

OpenAI. Harness engineering: leveraging Codex in an agent-first world. https://openai.com/index/harness-engineering, 2026a. Blog post.

OpenAI. Unrolling the codex agent loop. https://openai.com/index/unrolling-the-codex-agent-loop, 2026b. Blog post.

OpenAI. Function calling. OpenAI API Documentation, 2026c. URL https://developers.openai.com/api/docs/guides/function-calling. Accessed May 14, 2026.

OpenAPI Initiative. OAI/OpenAPI-specification. GitHub repository, 2025. URL https://github.com/OAI/OpenAPI-Specification. Accessed May 14, 2026.

OpenTelemetry. OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://github.com/open-telemetry, 2026.

OthmanAdi. planning-with-files: Persistent file-based planning. https://github.com/OthmanAdi/planning-with-files, 2025. GitHub repository.

OWASP Foundation. OWASP top 10 for large language model applications 2025. https://owasp.org/www-project-top-10-for-large-language-model-applications/, 2025.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. *arXiv*, 2023.

Matthew J Page, Joanne E McKenzie, Patrick M Bossuyt, Isabelle Boutron, Tammy C Hoffmann, Cynthia D Mulrow, Larissa Shamseer, Jennifer M Tetzlaff, Elie A Akl, Sue E Brennan, et al. The prisma 2020 statement: an updated guideline for reporting systematic reviews. *bmj*, 372, 2021.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Shishir G Patil, Tianjun Zhang, Vivian Fang, Roy Huang, Aaron Hao, Martin Casado, Joseph E Gonzalez, Raluca Ada Popa, Ion Stoica, et al. Goex: Perspectives and designs towards a runtime for autonomous llm applications. *arXiv preprint arXiv:2404.06921*, 2024a.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 2024b. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Plastic Labs. Honcho: Memory library for building stateful agents. https://github.com/plastic-labs/honcho, 2025. GitHub repository.

Prime Intellect. verifiers: RL environments, evaluations, and agent harnesses for language models. https://github.com/PrimeIntellect-ai/verifiers, 2026. Open-source software. Accessed: 2026-05-06.

promptfoo contributors. Promptfoo: LLM evals and red teaming. https://github.com/promptfoo/promptfoo, 2026. Open-source software.

Pyodide Contributors. Pyodide: Python with the scientific stack, compiled to WebAssembly. https://pyodide.org/, 2018. Open-source software.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023a. URL https://arxiv.org/abs/2307.07924.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, Singapore, 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.462. URL https://aclanthology.org/2023.findings-emnlp.462/.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Brandon Radosevich and John Halloran. Mcp safety audit: Llms with the model context protocol allow major security exploits. *arXiv preprint arXiv:2504.03767*, 2025.

RagaAI. RagaAI-Catalyst: Python SDK for agent AI observability, monitoring, and evaluation. https://github.com/raga-ai-hub/RagaAI-Catalyst, 2026.

Benjamin Rombaut, Sogol Masoumzadeh, Kirill Vasilevski, Dayi Lin, and Ahmed E Hassan. Watson: A cognitive observability framework for the reasoning of llm-powered agents. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 739–751. IEEE, 2025.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025. Open-source software.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Yonadav Shavit, Sandhini Agarwal, Miles Brundage, Steven Adler, Cullen O'Keefe, Rosie Campbell, Teddy Lee, Pamela Mishkin, Tyna Eloundou, Alan Hickey, Katarina Slama, Lama Ahmad, Paul McMillan, Andrea Vallone, Alexandre Passos, and David G. Robinson. Practices for governing agentic AI systems. https://openai.com/index/practices-for-governing-agentic-ai-systems/, 2023. OpenAI publication.

Shivanshu Shekhar, Tanishq Dubey, Koyel Mukherjee, Apoorv Saxena, Atharv Tyagi, and Nishanth Kotla. Towards optimizing the costs of llm usage. *arXiv preprint arXiv:2402.01742*, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. TaskBench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574, 2024. doi: 10.52202/079017-0148. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/085185ea97db31ae6dcac7497616fd3e-Abstract-Datasets_and_Benchmarks_Track.html.

Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for llm agents. *arXiv preprint arXiv:2504.11703*, 2025a.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524, Vienna, Austria, 2025b. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.1258. URL https://aclanthology.org/2025.findings-acl.1258/.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2303.11366.

Significant Gravitas. AutoGPT: Build, deploy, and run AI agents. GitHub repository, 2023. URL https://github.com/Significant-Gravitas/AutoGPT.

Skyvern-AI. Skyvern: Automate browser-based workflows with AI. GitHub repository, 2025. URL https://github.com/Skyvern-AI/skyvern.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808*, 2026.

Tobin South, Samuele Marro, Thomas Hardjono, Robert Mahari, Cedric Deslandes Whitney, Alan Chan, and Alex Pentland. Position: Ai agents need authenticated delegation. In *Forty-second International Conference on Machine Learning Position Paper Track*, 2025.

Joseph Spracklen, Raveen Wijewickrama, AHM Nazmus Sakib, Anindya Maiti, and Bimal Viswanath. We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 3687–3706, 2025.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, Yang Song, Hongteng Xu, Tao Zhang, and Ji-Rong Wen. SWE-World: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

Rao Surapaneni, Miku Jha, Michael Vakoc, and Todd Segal. Announcing the Agent2Agent protocol (A2A). Google Developers Blog, April 2025. URL https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/.

SWE-agent. Swe-agent. https://github.com/swe-agent/swe-agent, 2025. GitHub repository.

SWE-agent Team. SWE-ReX: SWE-agent remote execution framework. https://github.com/SWE-agent/swe-rex, 2024. Open-source software.

Georgios Syros, Anshuman Suri, Jacob Ginesin, Cristina Nita-Rotaru, and Alina Oprea. Saga: A security architecture for governing ai agentic systems. *arXiv preprint arXiv:2504.21034*, 2025.

Haoran Tan, Zeyu Zhang, Chen Ma, Xu Chen, Quanyu Dai, and Zhenhua Dong. Membench: Towards more comprehensive evaluation on the memory of llm-based agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19336–19352, 2025.

TensorZero. TensorZero: An open-source stack for industrial-grade LLM applications. https://github.com/tensorzero/tensorzero, 2026.

Th0rgal. sandboxed.sh: Self-hosted orchestrator for AI autonomous agents. https://github.com/Th0rgal/sandboxed.sh, 2026. Open-source software.

thedotmack. claude-mem: Plugin-style memory layer for coding agents. https://github.com/thedotmack/claude-mem, 2025. GitHub repository.

Traceloop. OpenLLMetry: Open-source observability for LLM applications based on OpenTelemetry. https://github.com/traceloop/openllmetry, 2026.

Trail of Bits. Jumping the line: How MCP servers can attack you before you ever use them. Trail of Bits Blog, April 2025. URL https://blog.trailofbits.com/2025/04/21/jumping-the-line-how-mcp-servers-can-attack-you-before-you-ever-use-them/.

Vivek Trivedy. Improving Deep Agents with harness engineering. https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering, 2026. Blog post.

Lillian Tsai and Eugene Bagdasarian. Contextual agent security: A policy for every purpose. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pp. 8–17, 2025.

Shubham Ugare and Satish Chandra. Agentic code reasoning. *arXiv preprint arXiv:2603.01896*, 2026. URL https://arxiv.org/abs/2603.01896.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vectorize.io. Hindsight: Agent memory that learns. https://github.com/vectorize-io/hindsight, 2025. GitHub repository.

Vaishali Vinay. Failure modes in llm systems: A system-level taxonomy for reliable ai applications. *arXiv preprint arXiv:2511.19933*, 2025.

Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. The instruction hierarchy: Training llms to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208*, 2024.

Haoyu Wang, Christopher M Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents.(2026). In *Proceedings of the IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 12–18, 2026.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. *arXiv preprint arXiv:2406.04692*, 2024. URL https://arxiv.org/abs/2406.04692.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, et al. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690*, 2025c.

Bowen Wei, Yunbei Zhang, Jinhao Pan, Kai Mei, Xiao Wang, Jihun Hamm, Ziwei Zhu, and Yingqiang Ge. Clawsafety:" safe" llms, unsafe agents. *arXiv preprint arXiv:2604.01438*, 2026.

Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. IsolateGPT: An execution isolation architecture for LLM-based agentic systems. In *Network and Distributed System Security Symposium (NDSS)*, 2025. URL https://www.ndss-symposium.org/ndss-paper/isolategpt-an-execution-isolation-architecture-for-llm-based-agentic-systems/.

Xi Xiao, Yunbei Zhang, Xingjian Li, Tianyang Wang, Xiao Wang, Yuxiang Wei, Jihun Hamm, and Min Xu. Visual instance-aware prompt tuning. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 2880–2889, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025.

Boyang Yan. Fault-tolerant sandboxing for ai coding agents: A transactional approach to safe autonomous execution. *arXiv preprint arXiv:2512.12806*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Yuzhe Yu. Claude code reverse engineering: A tool to visualize Claude Code's LLM interactions. https://github.com/Yuyz0112/claude-code-reverse, 2026.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023. URL https://arxiv.org/abs/2309.17428.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 951–972, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.44. URL https://aclanthology.org/2025.naacl-long.44/.

Yunbei Zhang, Yingqiang Ge, Weijie Xu, Yuhui Xu, Jihun Hamm, and Chandan K Reddy. Visual exclusivity attacks: Automatic multimodal red teaming via agentic planning. *arXiv preprint arXiv:2603.20198*, 2026a.

Yunbei Zhang, Kai Mei, Ming Liu, Janet Wang, Dimitris N Metaxas, Xiao Wang, Jihun Hamm, and Yingqiang Ge. Agents in the wild: Safety, society, and the illusion of sociality on moltbook. *arXiv preprint arXiv:2602.13284*, 2026b.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

YanZhao Zheng, ZhenTao Zhang, Chao Ma, YuanQiang Yu, JiHuai Zhu, Yong Wu, Tianze Xu, Baohua Dong, Hangcheng Zhu, Ruohui Huang, and Gang Yu. SkillRouter: Skill routing for LLM agents at scale. *arXiv preprint arXiv:2603.22455*, 2026. URL https://arxiv.org/abs/2603.22455.

Yusheng Zheng, Yanpeng Hu, Tong Yu, and Andi Quinn. Agentsight: System-level observability for ai agents using ebpf. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, pp. 110–115, 2025.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yuhang Zhou, Lizhu Zhang, Yifan Wu, Jiayi Liu, Xiangjun Fan, Zhuokai Zhao, and Hong Yan. Synthetic sandbox for training machine learning engineering agents. *arXiv preprint arXiv:2604.04872*, 2026.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025. URL https://arxiv.org/abs/2509.25370. arXiv:2509.25370.