From Question Answering to Task Completion: A Survey on Agent System and Harness Design

Jianyuan Guo , Zhiwei Hao , Chengcheng Wang , Cheng Fan , Tingzhang Luo , Hongguang Li , Ying Gao , Hefei Mei , Jiankun Peng , Rongjian Xu , Minjing Dong , Han Wu , Mengyu Zheng , Kai Han , Shiqi Wang , Chang Xu * , Yunhe Wang *

Posted Date: 17 June 2026

doi: 10.20944/preprints202606.1312.v1

Keywords: LLM-based agents; harness engineering; prompt engineering; model-harness co-evolution; evaluation benchmarks

 doi:10.20944/preprints202606.1312.v1

Article

From Question Answering to Task Completion: A Survey on Agent System and Harness Design

Jianyuan Guo [1], Zhiwei Hao [1], Chengcheng Wang [2], Cheng Fan [1], Tingzhang Luo [1], Hongguang Li [1], Ying Gao [1], Hefei Mei [1], Jiankun Peng [1], Rongjian Xu [1], Minjing Dong [1], Han Wu [3], Mengyu Zheng [4], Kai Han [4], Shiqi Wang [1], Chang Xu [2,*] and Yunhe Wang [4,*]

[1] Department of Computer Science, City University of Hong Kong, HKSAR, China
[2] School of Computer Science, University of Sydney, Australia
[3] Peking University, China
[4] TokenRhythm Technologies, China
* Correspondence: c.xu@sydney.edu.au (C.X.); wangyunhe@pku.edu.cn (Y.W.)

Abstract

LLM-based agents mark a shift from passive question answering to active task completion: they perceive environments, invoke tools, maintain state, and act over extended horizons. As agent systems have evolved from prompt engineering to workflows and context engineering, harness engineering, and agent-native training with co-evolution, a central question has become increasingly important: where does the bottleneck in agent performance reside—in the foundation model, in the execution harness, or in the coupling between them? This survey examines LLM-based agents through a model–harness lens. We first clarify the functional definition of agents and the implementation view of an LLM-based agent as a foundation model coupled with an execution harness. We then analyze the limits of model-centric scaling, trace four paradigms of agent engineering, and decompose the execution harness into six coupled runtime responsibilities: observation, context, control, action, state, and verification/governance. Using this decomposition, we map task properties and domain pressures to harness configurations, review benchmark and evaluation practices, and synthesize model–harness evidence on how runtime design affects long-horizon task completion, efficiency, and reliability. Finally, we identify open challenges in value-aware evaluation, safety, harness generalization, and model–harness co-evolution. Rather than treating agents as models with auxiliary tools, this survey argues that agent quality—including success, efficiency, safety, and generalization—emerges from the interaction between model capability, runtime infrastructure, task structure, and evaluation design. A collection of papers discussed in this survey is provided in https://github.com/ggjy/Awesome-Agent-Engineering.

Keywords: LLM-based agents; harness engineering; prompt engineering; model-harness co-evolution; evaluation benchmarks

1. Introduction

"Nothing is particularly hard if you divide it into small jobs." — Henry Ford

Large language model (LLM)-based agents—autonomous systems that perceive environments, reason over goals, and execute multi-step actions—mark a transition from passive question answering [1,2] to active task completion [3,4]. Unlike early chat interfaces that optimized single-turn response quality, modern agent systems operate as closed loops that invoke tools, update state, and verify outcomes over extended horizons. Prominent examples span multiple domains: coding agents such as Devin [5], Claude Code [6], and Codex [7] independently diagnose and resolve software engineering tasks across entire repositories; general-purpose agents like Manus [8] orchestrate multi-step workflows from research to data analysis; open-source platforms including AutoGPT [9], OpenHands [10], and OpenClaw [11] provide extensible frameworks for building custom agent pipelines.

 doi:10.20944/preprints202606.1312.v1

This landscape illustrates a broader shift from conversational competence to operational competence, and it changes where the performance bottleneck lies. For question answering (QA), incremental improvements in model capability—*i.e.*, larger parameters, more training data, or better alignment—often yield direct and predictable gains. Yet traditional benchmarks that measure such capability, including MMLU [12], GPQA [13], and HumanEval [14], have become increasingly saturated at the frontier, with contamination risks further complicating interpretation; harder evaluations such as Humanity's Last Exam [15] have been proposed to restore discriminative power. More critically, when evaluation shifts from closed-form QA to interactive, multi-step task completion, even frontier models reveal substantial reliability gaps—SWE-bench [16], WebArena [17], OSWorld [18], TheAgentCompany [19], and Terminal-Bench [20] all demonstrate that agentic tasks retain significant headroom. This divergence between static benchmarks (nearing saturation) and agentic benchmarks (far from solved) raises a natural question: if model scaling alone does not close the gap on agentic tasks, what does?

1.1. Harness Design as a Performance Lever

A growing body of work suggests that agent performance is increasingly limited not only by the model's raw reasoning power, but also by the design of its *execution harness*: the runtime infrastructure that shapes what the model perceives, how it acts, and whether its errors are detected and recovered. The idea that interface design matters was first demonstrated empirically by SWE-agent [21], which showed that redesigning the agent–computer interface (ACI) can substantially improve SWE-bench performance under a fixed base model. The broader concept was subsequently crystallized under the term *harness engineering* by Hashimoto [22] and OpenAI [7], who framed an agent as *model plus harness* and identified observation shaping, action-space design, execution sandboxing, context management, and verification loops as its core components. More recently, NLAH [23] formalizes harness logic as an editable, portable natural-language artifact, and Meta-Harness [24] treats harness configuration as an optimizable search space.

Following this line of work, we adopt the harness-centric perspective as a unifying lens: we systematically examine how the design of the runtime infrastructure, rather than model capability alone, determines agent reliability, efficiency, and generalization across diverse tasks. We further argue that this perspective is now **extending beyond single-model scaffolds**: recent systems [25] treat the harness as a compositional runtime over multiple models, and increasingly as a learnable object whose routing, orchestration, and verification policies can themselves be optimized.

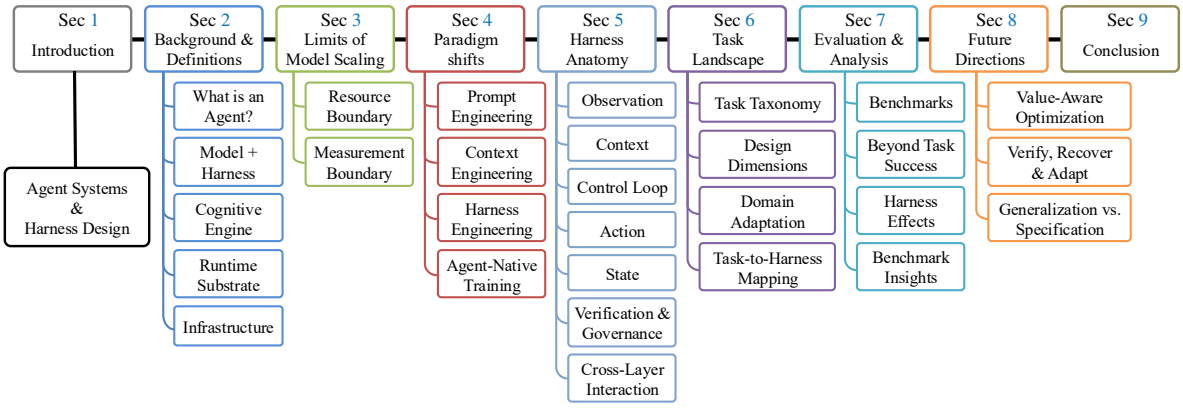

Figure 1. A diagram that summarizes the structure of this survey.

1.2. Four Paradigms of Agent Engineering

We organize the recent literature through an evolutionary lens of four paradigms. Each emerged to address limitations exposed by its predecessor; each foregrounds a different performance lever.

Phase 1: Prompt Engineering optimizes the single-turn instruction sent to the model. Techniques such as few-shot exemplars [1], chain-of-thought reasoning [26], self-consistency [27], and tree-of-thought search [28] can clarify tasks, constrain output format, and elicit the model's latent capabilities. Yet

 doi:10.20944/preprints202606.1312.v1

prompting fundamentally addresses an *expression* problem: how to ask. It does not solve the *information* problem: prompting alone cannot **supply missing knowledge, manage dynamically evolving state, or maintain coherence across long action sequences.**

Phase 2: Workflows and Context Engineering shifts the unit of optimization from a single prompt to the information lifecycle surrounding multi-step execution. Its core discipline is curating *what* information enters the model's context window, *when*, and *in what form* [29], encompassing retrieval-augmented generation [30], long-term memory management [31], tool and API definitions [32,33], and progressive skill disclosure [34,35]. The evaluation criterion changes accordingly: the question is no longer only whether a single answer is correct, but whether the assembled context enables the model to complete multi-step tasks. However, context engineering remains fundamentally feedforward: it optimizes the input to each reasoning step but **provides no structural mechanism to detect drift, verify intermediate outcomes, or recover from errors.**

Phase 3: Harness Engineering closes the loop. Beyond assembling the right context, the harness introduces feedback-driven execution: the model acts, observes environment responses, and reasons over observations to decide its next step [36]. More broadly, harness engineering treats the entire runtime infrastructure as the primary design object [7,22], governing execution sandboxing, state checkpointing, verification loops, error recovery, and sub-agent coordination [23,24]. The governing question shifts from *what to show the model* to *how to keep the whole system on track*: **prevent drift, maintain stable execution, and recover from errors.**

Within this phase, a further shift is already visible. Early harness design typically wraps a *fixed* foundation model with hand-crafted or searched runtime policies. More recent systems move toward a *multi-model harness*: the runtime routes, delegates, and composes heterogeneous models for planning, tool use, verification, coding, and domain-specific subtasks [37–39]. At the same time, the harness itself is becoming *learnable*: harness modules, orchestration logic, and runtime policies can be edited, searched, or optimized as first-class artifacts [23,24,40]. This changes what counts as an agent system. A single prompt wrapped around one model can still function as a lightweight agent, but reliable long-horizon task completion increasingly depends on **a compositional, optimizable runtime over multiple models**, not on prompt craft alone.

Phase 4: Agent-Native Training and Co-Evolution builds on the learnable multi-model harness view above. Its first direction is *internalization*: agentic behaviors such as planning, tool use, verification, and recovery are increasingly trained into model parameters through interactive environments [41–44]. Its second direction is *co-evolution*: over deployment, the model, harness, and improvement loop may all be updated from execution traces that indicate what to keep, change, or undo [40,44–46]. This does not eliminate the harness; it shifts the design question toward how much of agent behavior is learned in models, how much stays in the runtime, and how the full stack improves safely over time, opening a path toward **self-evolving agent systems**.

These four phases form a conceptual evolutionary lens rather than a strict temporal partition; all four coexist in practice today. Our goal is not to introduce another component taxonomy, but to use this progression and benchmark evidence (Sec. 7) to analyze how the dominant performance bottleneck moves across stages, and why harness design has become a central object of agent engineering.

doi:10.20944/preprints202606.1312.v1

Table 1. Comparison between our work and representative prior surveys. "Broad" denotes coverage of the general LLM-based agent landscape rather than a specific subfield; "Eval." denotes explicit coverage of benchmarks and the evaluation of methods; "App." denotes substantial discussion of application domains and use cases; and "Industry" denotes the extent to which a survey incorporates practitioner reports, production systems, or industrial engineering evidence as part of its main analysis.

Survey	Time	Organizing lens	Primary focus	Broad	Eval.	App.	Industry
Wang *et al*. [47]	2023.08	Module-based agent construction	How to construct an autonomous LLM agent through core modules, *e.g*., profile, memory, planning, and action.	✓	✗	✓	No
Xi *et al*. [4]	2023.09	Brain–perception–action framework	Agents as intelligent systems, from single-agent arch. to multi-agent society and human–agent interaction.	✓	✗	✓	No
Luo *et al*. [3]	2025.03	Build–collaborate–evolve taxonomy	Taxonomy of agents spanning methodological foundations, collaboration, applications, and evaluation.	✓	✓	✓	Limited
Guo *et al*. [48]	2024.02	Communication and collaboration	Overall progress, communication patterns, and open challenges in LLM-based multi-agent systems.	✗	✗	✓	No
Li *et al*. [49]	2024.10	Workflow-based taxonomy	How multi-agent systems are structured through workflow, infrastructure, core functional modules.	✗	✗	✗	Limited
Li *et al*. [50]	2025.01	Collaboration mechanism	Collaboration in multi-agent systems, categorized by actors, structures, strategies and coordination protocols.	✗	✗	✓	No
Shen *et al*. [51]	2025.03	Evaluation-based taxonomy	Benchmarks, metrics, and methodological issues in evaluating LLM-agents.	✗	✓	✗	Limited
Gu *et al*. [52]	2025.07	Domain-focused taxonomy	GUI/computer-use agents: benchmarks, architectures, and training methods.	✗	✓	✓	Limited
Ma *et al*. [53]	2024.05	Embodied-agent taxonomy	Vision-language-action models for embodied AI.	✗	✓	✓	No
Zhang *et al*. [54]	2025.03	Safety-oriented taxonomy	Threats, safety risks, evaluation, and countermeasures for trustworthy LLM-based agents.	✗	✓	✓	Limited
Meng *et al*. [55]	2026.04	Execution harness taxonomy	Six-component tuple for harness definition, historical tracing, and cross-cutting harness challenges.	✓	✗	✗	Strong
Li *et al*. [56]	2026.04	Seven-layer harness taxonomy	ETCLOVG, a seven-layer taxonomy and practitioner principles from deployed agent stacks.	✓	✗	✓	Strong
Ning *et al*. [57]	2026.05	Code-as-harness layers	Code as executable harness substrate: interface and multi-agent scaling across application domains.	✗	✓	✓	Limited
Ours	**2026.06**	**Engineering paradigm shifts**	**How agent engineering evolved from prompt optimization to runtime system design and future directions.**	✓	✓	✓	**Strong**

 doi:10.20944/preprints202606.1312.v1

1.3. Relation to Prior Surveys

Recent surveys have documented the rapid rise of LLM-based agents, but most organize the field through taxonomy-oriented lenses. Table 1 compares our survey with representative prior work. General-purpose surveys summarize agent architectures and components such as memory, planning, action, perception, applications, safety, and evaluation [3,4,47]. Multi-agent surveys focus on communication, coordination, collaboration structures, and workflow organization [48–50]. Other surveys examine narrower but important slices, including evaluation methodology [51], GUI/computer-use agents [52], embodied systems [53], and trustworthy agents [54]. Since early 2026, several works have narrowed the lens specifically to agent harnesses. Meng *et al*. [55] formalize the harness as a six-component tuple. Li *et al*. [56] further propose the seven-layer ETCLOVG taxonomy and map a large open-source corpus onto it to expose ecosystem coverage and production design principles. Ning *et al*. [**?**] organize the field from a code-centric perspective, treating executable programs as the substrate for reasoning, action, state, and verification. These surveys provide valuable harness taxonomies, catalogs, or substrate-specific roadmaps.

Relative to recent harness-focused surveys, our contribution is not primarily another layer taxonomy or project catalog. We instead ask how the dominant engineering bottleneck migrates across prompt optimization, context/workflow organization, compositional and learnable runtimes, and agent-native co-evolution, and how that migration should be evaluated empirically. Accordingly, we connect harness anatomy to task pressure profiles (Sec. 6), benchmark evidence (Sec. 7), and value-aware deployment objectives (Sec. 8), rather than centering the analysis on taxonomy completeness or repository coding alone.

Our survey makes three distinctions explicit. First, it is *evolution-first*: we organize the literature around engineering paradigm shifts rather than a static component taxonomy. Second, it is *harness-centric*: we treat the execution harness as a first-class technical object that governs observation, context, control, action, state, verification, recovery, and efficiency. Third, it connects *academic evidence with industrial practice*, using benchmark results, open-source systems, engineering reports, and controlled model–harness analyses to examine how runtime design choices affect agent reliability, cost, and latency.

In short, our goal is not only to catalog LLM-based agents, but to explain why harness engineering emerged as a central systems concern and how it may extend toward future agent-native training and co-evolution.

1.4. Scope Boundaries

This survey covers LLM-based agent systems from 2020 to 2026, including prompting methods, workflow frameworks, harness and runtime design, multi-model orchestration, agent-native training, model–harness co-evolution, domain deployments, and evaluation methodology. We focus on systems in which one or more LLMs serve as cognitive engines within an execution harness, and synthesize published papers, public engineering reports, benchmarks, and controlled model–harness comparisons. We prioritize high-impact and verifiable sources that directly inform agent system design, efficiency, or evaluation. Adjacent traditions such as neuro-symbolic planning, classical embodied control, and non-LLM systems are treated as complementary work rather than surveyed in depth, because they rely on different assumptions, architectures, and evaluation criteria.

1.5. Contributions and Survey Structure

The main contributions of this survey are:

- We provide an evolution-first synthesis of agent engineering, tracing shifts from prompt engineering to context engineering, harness engineering, and agent-native training.
- We analyze the limits of model-centric scaling for long-horizon task completion and argue that agent performance is a property of the model–harness pairing.

 doi:10.20944/preprints202606.1312.v1

- We formalize the execution harness as a runtime design object and decompose it into six coupled responsibilities.
- We map task properties, domain adaptations, and evaluation practices to harness pressure profiles rather than treating agent components as an independent checklist.
- We synthesize benchmark and empirical evidence to motivate value-aware evaluation beyond task success.

The remainder of this survey is organized as follows. Sec. 2 defines agents and harnesses and reviews core infrastructure. Sec. 3 analyzes the limits of model-centric scaling. Sec. 4 presents the four-paradigm evolution of agent engineering. Sec. 5 decomposes the execution harness into six runtime components. Sec. 6 maps task pressures and domain adaptations to harness configurations. Sec. 7 reviews benchmarks, evaluation methodology, and model–harness evidence. Sec. 8 discusses open challenges and future directions, and Sec. 9 concludes. Figure 1 summarizes the overall structure of this survey.

2. Background and Definitions

Two abstraction levels are often conflated in the agent literature. At the functional level, an agent is a goal-directed closed-loop system: it perceives an external environment, maintains task state, reasons and decides, executes actions, and adapts from feedback. At the implementation level, an LLM-based agent is not the foundation model alone, but a coupled system consisting of a foundation model and an execution harness. The model supplies flexible language understanding, reasoning, planning, and action proposal; the harness supplies the runtime machinery that exposes observations, constructs context, executes actions, persists state, and verifies or recovers from failures. This distinction reconciles classical agent definitions with recent harness-centered accounts of LLM agents: the former define *what* an agent must do, whereas the latter specify *how* those functions are realized in deployed systems.

2.1. Functional View: What Is an Agent?

The notion of an agent predates LLMs. Wooldridge and Jennings [58] characterize an intelligent agent as a system situated in an environment, able to perceive that environment and act upon it in pursuit of goals [59–61]. For this survey, the defining property is not whether the system is implemented by symbolic rules, reinforcement learning, or a language model, but whether it sustains a goal-conditioned loop with its environment. We therefore use a functional definition: an agent is a system that organizes five operations around a task objective: **perception**, **state maintenance**, **reasoning and decision-making**, **action**, and **feedback adaptation**. The goal and environment condition a particular run, but they are not themselves internal components of the agent. As illustrated in Figure 2, the agent receives observations, updates internal or external state, chooses the next action through reasoning and decision-making, acts on the environment, and incorporates feedback or outcomes into subsequent behavior. This closed-loop property separates agents from single-shot model calls, static retrieval systems, and fixed automation scripts that do not revise behavior as observations change.

 doi:10.20944/preprints202606.1312.v1

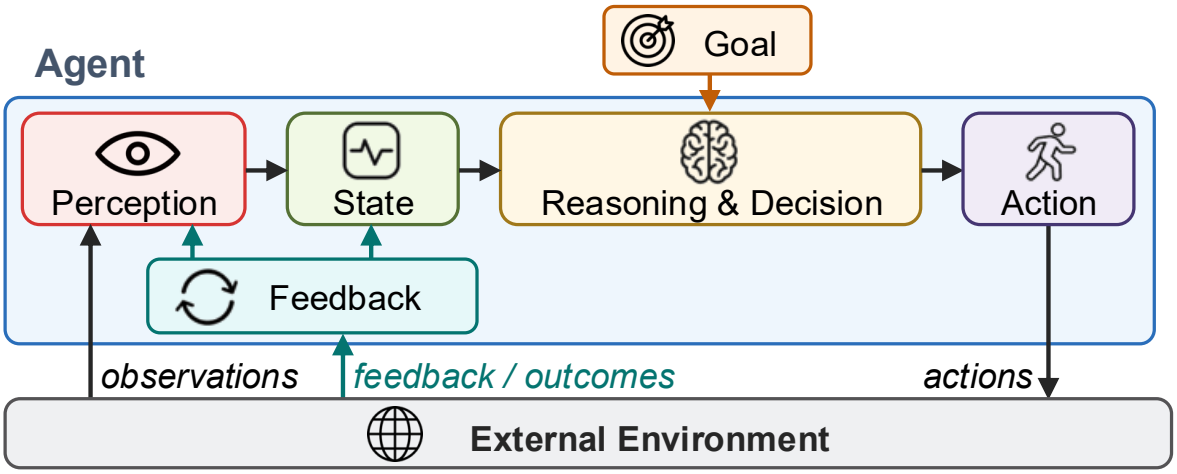

Figure 2. Functional view of an agent: a goal-directed closed-loop system that receives observations from external environments, maintains state, reasons and acts on the environment, and adapts from feedback or outcomes. This view defines *what* an agent must do, independent of any particular implementation.

2.2. Implementation View: Model Plus Harness

In LLM-based agents, the functional loop is implemented by more than an inference call to a model. The foundation model is necessary because it provides the general-purpose cognitive capabilities that make open-ended task completion possible. It is not sufficient, however, because the model does not by itself define what observations are available, which actions are permitted, where long-term state is stored, how execution is validated, or how failures are recovered. Following recent industrial and academic discussions of harness engineering [7,22], we write LLM-based agent as:

$$\mathcal{A}_{\text{LLM}} = \langle \mathcal{M}, \mathcal{H} \rangle = \langle \mathcal{M}, \mathcal{I}_{\text{obs}}, \mathcal{C}, \mathcal{L}, \mathcal{I}_{\text{act}}, \mathcal{S}, \mathcal{V} \rangle, \tag{1}$$

where $\mathcal{M}$ denotes the model layer of the agent. In the simplest case, $\mathcal{M}$ is a single foundation model. In deployed multi-model systems, $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_k\}$ denotes a set of backbone models with heterogeneous capabilities, costs, and context limits [37–39]. $\mathcal{H}$ denotes the execution harness surrounding $\mathcal{M}$. The second equality expands the harness into six runtime components used throughout this survey. This expression is an implementation-oriented decomposition, not a replacement for the functional definition above. When $|\mathcal{M}| = 1$, the agent reduces to the familiar single-backbone setting; when $|\mathcal{M}| > 1$, the harness must additionally decide *which* model acts at each step [25]. The model layer and harness jointly instantiate the functional loop: the active model reasons over a supplied context and proposes next steps, while the harness determines what it sees, what it can do, how execution state persists, and how errors are detected, constrained, or repaired.

2.3. LLM as the Cognitive Engine

LLMs became viable cognitive engines for agents because they combine capabilities that previously required separate modules or task-specific policies.

Reasoning and planning. Prompting methods such as chain-of-thought [26], Tree of Thoughts [28,62], self-consistency [27], and Reflexion [63] show that sufficiently capable models can support task decomposition, branching search, self-critique, and multi-step inference. These abilities make the model a plausible decision engine for tasks whose solution cannot be enumerated in advance.

In-context adaptation. The same frozen model can adapt its behavior through instructions, examples, retrieved documents, tool descriptions, and intermediate artifacts. This reduces the need to train a separate policy for each environment, while making the quality, ordering, and compression of the supplied context a primary determinant of behavior.

Action proposal and tool use. When models can emit structured tool calls [32,33], they are no longer limited to internal text generation. They can propose calls to code execution, retrieval systems,

 doi:10.20944/preprints202606.1312.v1

browsers, APIs, and external software. Yet a proposal is not an executed action: reliability depends on the harness to validate, dispatch, observe, and, when necessary, reject or repair the proposed action.

These strengths also expose the model's limitations. LLMs remain vulnerable to hallucination, finite context windows, weak persistent memory, prompt sensitivity, and limited intrinsic ability to verify long-horizon outcomes. The harness is therefore not an optional engineering wrapper; it is the runtime layer that turns model capability into sustained, inspectable interaction with an environment.

2.4. Harness as the Runtime Substrate

Following [7,22,64], we use *harness* to denote the runtime infrastructure that surrounds the model and realizes closed-loop agent execution. The harness is broader than an individual tool, memory module, prompt template, or workflow script. It is the coordinating layer that decides which observations reach the model, how context is assembled, how the agent loop advances, how actions are executed, how state and artifacts persist, and how failures are detected, governed, and recovered. This runtime substrate can be formalized as:

$$\mathcal{H} = \langle \mathcal{I}_{\text{obs}}, \mathcal{C}, \mathcal{L}, \mathcal{I}_{\text{act}}, \mathcal{S}, \mathcal{V} \rangle. \tag{2}$$

The six components are:

- **Observation interface** $\mathcal{I}_{\text{obs}}$: transforms raw environment signals into model-usable observations, including terminal output, file diffs, screenshots, DOM states, API responses, logs, retrieved passages, and event streams.
- **Context manager** $\mathcal{C}$: determines what information enters the model context, when it enters, and in what form, covering prompt construction, system instructions, retrieval, memory selection, compression, summarization, tool descriptions, and current task state.
- **Control loop** $\mathcal{L}$: orchestrates the observe-reason-act-feedback cycle, including step scheduling, stopping criteria, retries, reflection, delegation, handoffs, and multi-agent coordination. In multi-model settings, $\mathcal{L}$ additionally implements model routing and role assignment.
- **Action interface** $\mathcal{I}_{\text{act}}$: maps model outputs to executable operations, such as function calls, MCP tools, shell or code execution, browser actions, file operations, API calls, and sub-agent invocations.
- **State and artifact store** $\mathcal{S}$: persists execution state and products, including conversation history, plans, scratchpads, checkpoints, logs, traces, diffs, memory records, generated files, and task artifacts.
- **Verification and governance layer** $\mathcal{V}$: checks, constrains, and repairs execution through tests, assertions, verifier models, sandbox policies, permission gates, rollback, retry, budget control, safety constraints, and audit traces.

Figure 3 visualizes this implementation view. The figure should be read as an execution architecture rather than a static checklist: observations, context, control, actions, state, and verification form a coupled runtime around the model, and their interaction determines whether model capability becomes reliable task completion.

 doi:10.20944/preprints202606.1312.v1

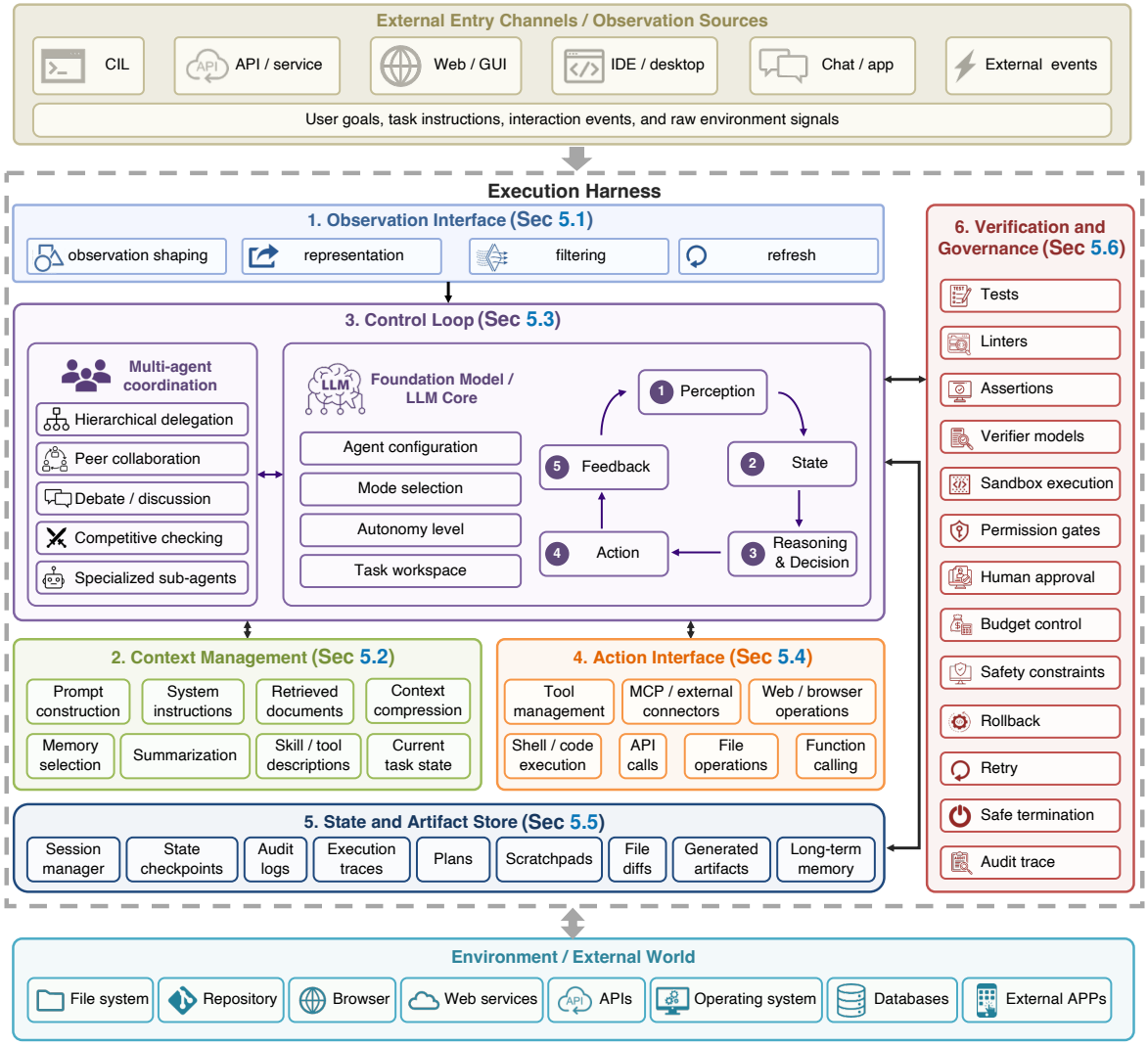

Figure 3. Implementation view of an LLM-based agent as a foundation model coupled with an execution harness. The harness mediates closed-loop interaction between the model and the external world through six runtime components: observation interface, context manager, control loop, action interface, state and artifact store, and verification and governance layer. The section labels inside the figure indicate where each component is analyzed in detail.

This decomposition differs from earlier component taxonomies because it is operational rather than purely functional. For example, memory appears in the functional loop as state, but in a deployed system it may be realized through context selection, artifact storage, retrieval indices, session managers, or checkpointing policies. Similarly, action is not merely an abstract action space; it is mediated by schemas, permissions, sandboxes, execution APIs, and side-effect controls. By separating these runtime responsibilities from the model itself, the decomposition explains why harness changes can improve agent performance even when the underlying model is unchanged [21,23,24].

2.5. Key Infrastructure Primitives

Several infrastructure primitives recur across modern LLM-based agents. They should not be treated as concepts parallel to the harness. Rather, they instantiate specific harness responsibilities in deployed systems and make perception, action, communication, and governance concrete.

Tool and function calling. Structured tool invocation converts model outputs from free-form suggestions into machine-executable calls [32,33]. Tool schemas are primarily part of the action interface $\mathcal{I}_{\text{act}}$, while tool descriptions, arguments, and returned results also shape the context manager $\mathcal{C}$ and observation interface $\mathcal{I}_{\text{obs}}$.

 doi:10.20944/preprints202606.1312.v1

Model Context Protocol (MCP). MCP [65] standardizes how LLM applications expose tools, data sources, and contextual resources to agents. In our notation, MCP primarily strengthens the boundary between the context manager and action interface by reducing connector fragmentation and making tool and data access more modular.

Agent-to-Agent communication. The Agent2Agent (A2A) protocol [66] targets interoperability among agents built by different vendors or frameworks. It is most relevant to the control loop $\mathcal{L}$ and action interface $\mathcal{I}_{\text{act}}$, especially when delegation, negotiation, debate, or multi-agent collaboration becomes part of the execution process.

Sandboxed execution and approval. When agents can write files, execute code, browse the web, or call APIs, isolation becomes both a safety mechanism and a reproducibility primitive. Sandboxes constrain filesystem access, network egress, process execution, and resource usage, while approval policies determine when human authorization is required before an action is dispatched. These mechanisms belong primarily to the verification and governance layer $\mathcal{V}$.

Agent SDKs and tracing. Frameworks such as the OpenAI Agents SDK [39] expose reusable abstractions for tools, handoffs, tracing, and loops. They package common harness patterns into developer-facing interfaces, making runtime behavior more reusable, inspectable, and debuggable.

Table 2 summarizes how the conceptual operations in the functional agent loop are realized by the implementation components defined above. The mapping is many-to-many rather than one-to-one: perception depends not only on the observation interface, but also on the context manager that selects and formats observations for the model; feedback involves verification, control-loop decisions, and state updates. This many-to-many mapping bridges the conceptual view in Figure 2 and the implementation view in Figure 3.

Table 2. Mapping from conceptual agent operations to harness realizations.

Functional operation	Harness components	Typical mechanisms
Perception	$\mathcal{I}_{\text{obs}}, \mathcal{C}$	Logs, DOMs, screenshots, retrieval, summaries
State Maintenance	$\mathcal{S}, \mathcal{C}$	Memory, checkpoints, artifacts, conversation history
Reasoning, Decision	$\mathcal{M}, \mathcal{C}, \mathcal{L}$	Prompted reasoning, plans, tool-choice context
Action	$\mathcal{I}_{\text{act}}, \mathcal{V}$	Function calls, shell commands, APIs, approval gates
Feedback Adaptation	$\mathcal{V}, \mathcal{L}, \mathcal{S}$	Tests, reflection, retries, rollback, trace updates

With these definitions in place, common application labels can be read as specializations of the same model-harness architecture. Coding, web/GUI, research, embodied, and domain-specific agents all rely on the same six runtime components, but they stress different parts of the harness because their observation channels, action spaces, feedback signals, and safety constraints differ. Representative examples are summarized in Table 3.

 doi:10.20944/preprints202606.1312.v1

Table 3. Representative types of LLM-based agents.

Type	Examples	Environment	Challenge
Coding	Claude Code, Codex	Repository, terminal	Long-horizon reliability
Web / GUI	Operator, VisualWebArena	Browser, desktop	Grounding, safe interaction
Research	Deep Research, Elicit	Web, literature	Synthesis, citation fidelity
Embodied	Voyager	Real world	Sim-to-real transfer, safety
Domain-specific	ChemCrow, Agent Hospital	Specialized tools	Compliance, domain expertise

3. The Limits of Model-Centric Scaling

Once an LLM-based agent is viewed as a model coupled with an execution harness, the role of foundation-model scaling can be stated more precisely. Scaling remains one of the main reasons why LLMs can serve as the cognitive engine of modern agents. Scaling laws first showed that language-modeling loss improves predictably with model size, data, and compute [67], while Chinchilla-style results refined this picture by emphasizing compute-optimal allocation between model parameters and training tokens [68]. The empirical impact is broad: larger and better-trained models have improved reasoning and problem solving [69], code generation [70,71], mathematical reasoning [72,73], and multimodal understanding [74,75]. These gains make stronger foundation models indispensable to agent systems, but they don't make model size a complete account of agent performance. Long-horizon task completion is a trajectory-level property: an agent must repeatedly observe, construct context, choose actions, preserve state, interpret feedback, and recover from errors. The relevant question is therefore where model-centric explanation stops and runtime design begins. Two boundaries are especially important: a *resource-performance boundary* and a *measurement boundary*.

3.1. Resource-Performance Boundary

The first limit concerns how much additional capability is obtained for additional resources. Increasing model capacity continues to improve frontier performance, but the gains are increasingly costly, uneven across capabilities, and constrained by inference latency and deployment complexity. LLaMA3.1 [76] provides a representative example: moving from the 70B model to the 405B model increased training compute from 7.0M to 30.84M H100 GPU hours, but yielded modest gains on several representative benchmarks, including 2.6 points on MMLU [12], 2.6 points on MBPP EvalPlus [77], and 1.7 points on GSM8K [78]. The larger model also remained far from near-perfect performance on harder reasoning benchmarks such as MATH [79] and MMLU-Pro [80]. Similar uneven returns are also visible in Qwen3 [81], where gains from a much larger base model vary substantially across benchmarks.

 doi:10.20944/preprints202606.1312.v1

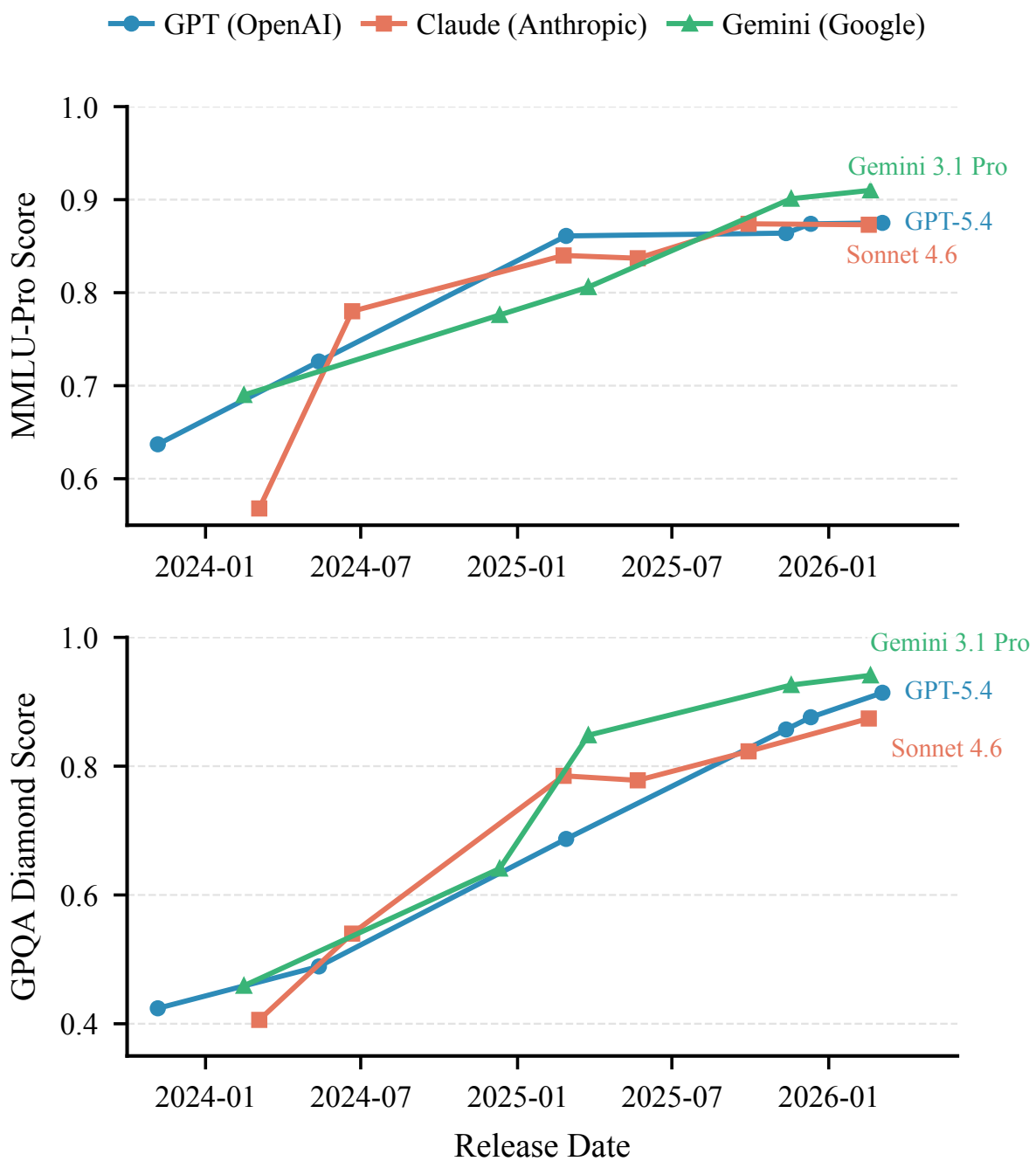

Figure 4. Evolution of frontier-model performance on MMLU-Pro and GPQA Diamond. Recent GPT, Claude, and Gemini releases increasingly occupy a narrow high-score range on both benchmarks, making later improvements less discriminative than earlier model-generation jumps.

Closed-source frontier models show the same pattern at the high-performance end. MMLU-Pro [80,82] and GPQA Diamond [13,83] remain challenging, but recent GPT, Claude, and Gemini releases increasingly cluster within a narrow score range, as shown in Figure 4. This does not mean that scaling has stopped working. Rather, once models enter a high-accuracy regime on common evaluations, additional scale often yields smaller and more capability-specific improvements while imposing higher cost, latency, and operational burden. For agents, this trade-off is amplified: a deployed agent invokes the model repeatedly across an execution trajectory, so per-call cost and latency accumulate, and small errors can compound over many steps.

 doi:10.20944/preprints202606.1312.v1

3.2. Measurement Boundary

The second limit concerns how scaling-driven progress is measured. Model-centric progress has often been validated through aggregate gains on static benchmarks. This was informative when benchmarks clearly separated model generations, but it becomes less discriminative when frontier systems cluster near the upper range of same metrics. The compression in Figure 4 illustrates the problem: small score differences on saturated benchmarks are difficult to interpret as meaningful differences in real-world agent capability.

The deeper issue is structural. Many traditional benchmarks are static, short-horizon, and self-contained: the input is fixed, the output is evaluated once, and the environment does not change in response to the model's actions. Agent tasks have a different form, they require long-context understanding, multi-step reasoning, environment interaction, tool use, adaptation to underspecified goals, and robustness to intermediate errors [84–91]. Recent evaluations make this mismatch explicit. For example, SWE-bench [16], BigCodeBench [92] and LiveClowBench [93] show that coding capability depends on repository-level context, executable environments, and realistic modification constraints, while MultiChallenge [94] shows that dialogue evaluation must capture inferential memory, revision, and consistency across turns. Together, these limits shift the central question from whether stronger models matter to how model competence is converted into dependable execution. Agent evaluation must therefore consider task duration, step count, environmental uncertainty, tool-use complexity, state persistence, and recovery demand. For example, a recent time-horizon study [95] evaluates agents by the duration of human tasks they can complete at a fixed success probability, rather than by single-shot accuracy alone. This framing makes long-horizon reliability central: progress depends not only on model competence, but also on the runtime that turns competence into sustained action, including what the model observes, how context is constructed, which actions are available, where state is preserved, and how errors are detected or repaired. This helps explain why agent engineering has moved from eliciting isolated model responses toward designing the surrounding execution environment.

4. Paradigm Shifts in Agent Engineering

The limits of model-centric scaling in Sec. 3 raise a more precise question for agent systems: where is reliable agentic behavior actually produced? Early LLM-based systems placed much of this burden on prompting, assuming that latent model capabilities could be elicited by effective instructions. As tasks required external knowledge, tool use, memory, and intermediate artifacts, the focus shifted to agentic workflows and context engineering. When these workflows became longer, more stateful, and more failure-prone, the bottleneck moved from organizing information for the model to controlling execution around the model, elevating the harness from an implementation detail to a first-class design object. More recently, verification and recovery have also become training targets, suggesting that some agentic behaviors may be internalized rather than externally scaffolded. These phases coexist in present systems, but together they reveal a migration of bottlenecks from prompt elicitation, to context and workflow organization, to harness-level execution control, to compositional and learnable multi-model runtimes, and finally toward agent-native training and model–harness co-evolution. Figure 5 summarizes this migration as a change in the locus of engineering effort, not as a claim that later paradigms replace earlier ones.

 doi:10.20944/preprints202606.1312.v1

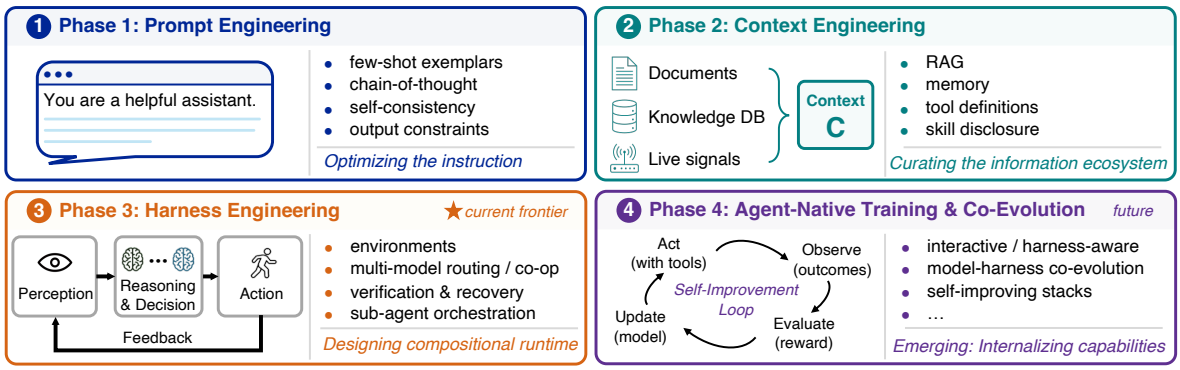

Figure 5. Four paradigms of agent engineering. The main locus of effort shifts from eliciting model behavior, to organizing context, stabilizing execution, composing and learning multi-model runtimes, and training or co-evolving agentic behavior.

4.1. Phase 1: Prompt Engineering

Phase 1 treated the prompt as the main interface through which latent model capabilities could be elicited and controlled. This view was established by in-context learning and then extended through zero-shot and few-shot prompting, chain-of-thought prompting, self-consistency, tree-style reasoning, self-refinement, ReAct-style reasoning-action traces, and automatic prompt optimization [1,26–28,36,96–98]. These methods made prompting a practical mechanism for task specification, reasoning elicitation, output-format control, and behavioral steering. More recent agent-oriented reasoning work broadens this phase from single-chain elicitation toward structured reasoning and planning. In software tasks, multi-agent optimization and question-driven self-QA extend prompting toward collaborative design reasoning [99], [100]. Self-evolving and graph-structured multi-agent methods further explore reasoning as an adaptive collaboration process rather than a linear trace alone [101], [102]. Other work makes reasoning traces operational for failure management [103], introduces reflection, branching, and rollback into web-agent reasoning[104], and uses reasoning gates to decide when web agents should continue or be constrained [105]. Heterogeneous-model assembly and intent-level skill abstraction also connect prompt-level reasoning with planning and computer-use skill organization [38], [106], while agentic software-architecture studies frame this progression as a shift from prompt-response interaction to goal-directed systems [107]. They are therefore essential to early agent systems, but their limitation is equally important for this survey's argument. Prompt engineering primarily addresses an *expression and elicitation* problem: it improves how a task is posed to the model and how the model's existing capabilities are invoked. It does not reliably provide knowledge absent from the model, maintain dynamically changing task state, validate external actions, or recover from failures over long execution trajectories. This motivates the next shift, from asking how to phrase the instruction to asking what information environment should surround each model call.

4.2. Phase 2: Workflows and Context Engineering

Phase 2 shifted the engineering focus from prompt design to agentic workflow orchestration and context management. This shift addressed two limitations left by prompting: the model may lack task-relevant knowledge, and the information needed during execution may change as the environment responds. Agentic workflows respond by sequencing model calls, retrieval, tool use, memory access, intermediate artifacts, and branching logic around the model [108–112]. Recent systems make this workflow view concrete in software and tool-use settings. SGAgent decomposes repository-level repair into suggestion-guided multi-agent collaboration [113], while studies of agentic coding-tool configuration show that performance depends not only on the base model, but also on workflow and tool settings [114]. Tool-use-oriented work further synthesizes tool-use trajectories through multi-agent role-playing [115] and studies extended tool-integrated reasoning as a way to scale agentic workflows beyond isolated tool calls [116]. Within these workflows, context engineering provides the central

 doi:10.20944/preprints202606.1312.v1

technical perspective: context is no longer a static prompt string, but a dynamically assembled runtime object. Formally, instead of assuming C=prompt, context is treated as:

$$C = A(c_1, c_2, \ldots, c_n), \tag{3}$$

where A denotes a high-level assembly function that combines contextual components c_i, including instructions, retrieved knowledge, tool descriptions and outputs, memory records, task state, intermediate artifacts, and the current query, into the final context C. Under this view [117], the engineering problem changes from optimizing the wording of a prompt to optimizing the functions that retrieve, select, compress, format, and refresh information during execution:

$$F^* = \arg\max_F \mathbb{E}_{\tau \sim T}[\text{Reward}(P_\theta(Y \mid C_F(\tau)), Y_\tau^*)], \tag{4}$$

where F denotes the set of context-construction functions, $C_F(\tau)$ is the context produced for task instance τ, and Y_τ^* denotes the desired or reference outcome. The objective is to maximize expected task quality under the constructed context, rather than to optimize a single instruction in isolation.

Agentic workflows therefore reframed the core engineering question from how to write better instructions to how to construct, organize, and update the information and tool-use environment available during execution. This development can be read through three closely related directions. The first focused on external information access. Retrieval-based methods such as RAG [30,118] introduced a practical mechanism for exposing non-parametric knowledge to the model, while retrieval-augmented architectures such as Fusion-in-Decoder [119], RETRO [120], and Atlas [121] further strengthened this paradigm. Later systems such as RAPTOR [122], GraphRAG [123], and HippoRAG [124] extended retrieval from flat passage lookup to richer pipelines based on hierarchical summarization, graph construction, and relation-aware memory organization.

The second direction focused on systematic context management. Here the question is not only what to retrieve, but also when to inject information, how to compress it, how to refresh it, and how to preserve task-relevant state over long-horizon execution. This shift is reflected in methods such as ACON [125], which formulates context compression as an optimization problem, ARC [126], which treats context as a dynamically managed internal state updated through reflection, and ContextBudget [127], which makes compression decisions under explicit context-window constraints. Related work further examines context maintenance in software and long-horizon settings, including CAT [128], which elevates context maintenance into a callable tool within the agent loop, and Compressing Code Context for LLM-based Issue Resolution [129], which studies how to distill and preserve task-relevant code context under limited budgets.

The third direction treated context itself as an explicit object of evaluation and optimization. Benchmarks such as ContextBench [130], SWE Context Bench [131], LoCoBench-Agent [132], and AgentLongBench [133] made context retrieval, retention, and utilization measurable research targets. More recent work such as ACE [134] and MCE [135] further treats contexts, and even context-engineering strategies, as adaptive optimization targets. Accordingly, Phase 2 can be read as a progression from external information access, to systematic context management, and finally to explicit context evaluation and adaptive context optimization.

Yet even well-engineered workflows and context do not by themselves guarantee reliable agency. They arrange information, tools, and intermediate steps around the model, but they do not fully specify how the overall process should remain stable, verifiable, and recoverable. As tasks became increasingly tool-augmented, stateful, and failure-prone, the bottleneck shifted from managing the information and workflow environment to designing the execution environment itself. Context did not disappear; it became one core component within a broader execution layer that must also manage action, state persistence, and verification.

 doi:10.20944/preprints202606.1312.v1

4.3. Phase 3: Harness Engineering

This phase begins when the central bottleneck is no longer only how a workflow assembles information and tool calls, but how the agent is controlled across a multi-step execution trajectory. Agentic workflows improve what enters the context window and which tools are invoked, but many long-horizon failures are not caused by missing information alone. They are execution failures: the agent loses track of progress, misuses tools, drifts from the original objective, repeats unproductive steps, or fails to recover after an error.

Harness engineering emerges from this execution-level bottleneck. As defined in Sec. 2.4, the harness is the structured runtime layer that organizes and stabilizes agent execution. It determines what the agent observes, what actions it may take, what state is carried forward, how control advances, and how failures are detected, constrained, or repaired. Workflow and context design remain important, but they become components within a broader execution layer that also manages observation, action, persistent state, verification, and governance.

Evidence that harness design matters. The case for harness design is no longer based only on engineering intuition or isolated examples. SWE-agent [21] showed that redesigning the agent-computer interface alone can substantially improve coding-agent performance under a fixed model. NLAH framed harness modules as portable and inspectable artifacts, and reported controlled ablations indicating that their contributions are measurable and additive [23]. Meta-Harness went one step further by treating harness optimization itself as a search problem, showing that automatically improved harnesses can outperform hand-designed baselines on Terminal-Bench [20,24]. Recent works extend this evidence from coding-agent interfaces to runtime orchestration and formal control mechanisms [136–138]. Other systems treat memory, protocol interoperability, contextual problem enhancement, and enterprise context lifecycles as harness-level design objects [139–144]. These results suggest that harness design has become a first-class optimization surface rather than a secondary implementation detail. More experimental results can be found in Sec. 7.

Industrial and ecosystem perspectives. The same transition is visible in industrial systems. Anthropic's public guidance emphasizes minimal, legible tools and disciplined runtime behavior [6,64,145]. OpenAI's guidance emphasizes environment design, structured artifacts, and reusable agent infrastructure [7,39,146]. Microsoft's Magentic-One highlights multi-agent orchestration for complex web and file tasks [37], while open-source systems [10,25], *e.g.*, OpenHands, expose harness itself as inspectable code.

At the ecosystem level, recent protocol-centered benchmarks reinforce the same shift by evaluating whether agents can invoke real services under realistic tool-routing conditions. MCPWorld [147], MCP-Atlas [148], MCPAgentBench [149], and OSWorld-MCP [150] move the discussion from abstract protocol design to measurable runtime behavior. Together, these systems and benchmarks suggest that harness engineering is becoming not only an engineering practice, but also a shared layer of infrastructure, evaluation, and design philosophy.

Design principles. Across papers and systems, several high-level principles recur:

- **Legibility:** the runtime should expose the right state at the right level of abstraction.
- **Mechanical enforcement:** constraints that matter for safety, correctness, or reproducibility should be enforced by the runtime when possible, rather than delegated entirely to prompt obedience.
- **Verification in the loop:** long-horizon autonomy without intermediate checks is structurally brittle.
- **Explicit artifacts:** plans, logs, diffs, summaries, and other intermediate products should exist as inspectable objects that can be reused, audited, or handed off.

These principles make the harness concrete rather than metaphorical. It must expose observations, assemble context, organize control, mediate actions, persist state, and enforce verification and governance as a coupled runtime system. Sec. 5 therefore turns the phase-level argument into an anatomy of the six harness components.

 doi:10.20944/preprints202606.1312.v1

Multi-model harnesses. Many recent systems no longer treat one model as the sole cognitive engine for every step. Instead, the harness composes heterogeneous models for planning, coding, tool use, verification, retrieval, and domain-specific subtasks [10,37–39,108]. This changes the control loop from "one model iterates until done" to "the runtime decides which model acts next, with what context, and under what constraints." Representative patterns include planner–executor–verifier decomposition, specialist routing, debate or committee-style validation, and handoffs among sub-agents [37,113,115,151]. From the harness-anatomy view in Sec. 5, multi-model design [65,66,147,150]. primarily stresses the control loop $\mathcal{L}$, but it also reshapes the context manager $\mathcal{C}$, action interface $\mathcal{I}_{\text{act}}$, and verification layer $\mathcal{V}$ because different models may observe, act, and judge under different scopes and permissions.

Learnable harnesses. In parallel, the harness itself is becoming an optimizable object. NLAH [23] treats harness logic as editable and portable runtime artifacts; Meta-Harness [24] searches over harness configurations; and Agentic Harness Engineering (AHE) [40] evolves harness components from observability-driven feedback while holding the base model fixed. These systems differ in mechanism—manual editing, search, or trace-driven adaptation—but they share one implication: runtime policies for routing, tool exposure, memory use, and verification can be improved as directly as prompts once were.

Together, multi-model composition and learnable runtime policies mark a qualitative shift in what counts as an agent system. A lightweight prompt-driven loop can still behave like an agent on short horizons, but dependable long-horizon task completion increasingly requires designing a compositional runtime over multiple models, with explicit orchestration, verification, and adaptation policies.

4.4. Phase 4: Agent-Native Training and Co-Evolution

Phase 4 begins once the harness is viewed not only as a hand-stabilized runtime, but as a compositional and increasingly learnable system over one or more models (Sec. 4.3). The central question is therefore twofold: which agentic behaviors should be *internalized* into model parameters, and how should the model and harness *co-evolve* over deployment without sacrificing safety or inspectability.

Internalization through Interactive Training. The first direction internalizes agentic behavior into the model itself. Rather than relying solely on prompts, workflows, or runtime orchestration, models are increasingly trained to plan, use tools, verify intermediate states, and recover from errors in interactive environments [42,152–156].

Recent work reflects two closely related tendencies. The first strengthens reasoning-to-action behavior through reinforcement learning. Examples include DeepSeekMath [73], DeepSeek-R1 [43], and DAPO [157], which treat multi-step reasoning, action selection, and verification as trainable behaviors rather than purely prompt-induced ones. The second reduces train-test mismatch by training agents in environments closer to deployment. Examples include ProRL [158], WebRL [41], ComputerRL [42], Environment Tuning [159], daVinci-Dev [160], and Kimi-Dev [161]. Together, these lines suggest that behaviors first implemented externally—planning, tool invocation, reflection, and recovery—may gradually become partially learned inside the model. Internalization shifts the division of labor rather than removing the harness: more short-horizon behavior may move into model parameters, while the runtime still supplies environment access, state, and safety control.

Co-Evolution and Self-Improvement. The second direction extends agent engineering from one-time training to ongoing improvement of the full stack. Here the model, harness, and update policy may all change over deployment, using execution feedback to decide which changes to keep, revise, or roll back [40,44–46,162]. The goal is not only to move behavior into parameters, but to improve *how* the combined system learns from experience.

Several recent lines make this distinction concrete. Experience-driven systems such as EvolveR [44] and AgentEvolver [45] treat interaction trajectories as reusable learning signals through self-questioning, navigation, and attribution. Continual Harness [162] and reward-free

 doi:10.20944/preprints202606.1312.v1

self-evolution [163] explore online adaptation without relying on dense external rewards at inference time. Harness-side adaptation, exemplified by AHE [40], shows that runtime components can evolve even when the base model remains fixed. More ambitiously, recursive self-improvement systems such as SICA [164], Darwin Gödel Machine [46], and Hyperagents [165] suggest that the improvement mechanism itself may become modifiable over time.

We separate three layers that are often conflated under "self-evolve". *Multi-model harnesses* define *who* performs each runtime role. *Learnable harnesses* define *how* runtime policies are optimized. *Co-evolution* defines *when and how* the model, harness, and improvement loop are jointly updated from deployment experience. These layers are complementary rather than interchangeable: compositional runtimes create the structure in which specialization and delegation become possible; learnable harnesses make runtime adaptation explicit; co-evolution governs long-horizon improvement under verification, safety and cost constraints.

5. Anatomy of the Execution Harness

Harness engineering shifts the optimization target from isolated prompts or workflows to the runtime that stabilizes agent execution. Following the formalization in Sec. 2.4, this runtime can be decomposed into six components: $\mathcal{H} = \langle \mathcal{I}_{\text{obs}}, \mathcal{C}, \mathcal{L}, \mathcal{I}_{\text{act}}, \mathcal{S}, \mathcal{V} \rangle$. The decomposition is not intended as a software package diagram. Rather, it identifies the runtime responsibilities that repeatedly determine whether model capability becomes reliable task completion: what the model observes, what enters context, how execution advances, which actions are available, what state persists, and how the run is checked or constrained.

5.1. Observation Interface

The observation interface $\mathcal{I}_{\text{obs}}$ determines which environment signals are exposed to the model and how those signals are rendered. It converts external state, such as terminal output, file diffs, screenshots, web DOMs, API responses, event streams, and logs, into observations that can be consumed by the current model call. Its design space includes three recurring questions: which state is relevant, at what abstraction level it should be represented, and when the observation should be refreshed. These choices matter because many long-horizon failures are not failures of reasoning alone. They are also failures of legibility: the needed state is absent, buried in noise, stale, or represented at a level that does not support the next decision.

Representative systems make this point concrete. SWE-agent showed that redesigning the agent-computer interface can substantially improve coding-agent performance under a fixed base model [21]. In web and desktop settings, benchmarks such as WebArena and OSWorld likewise reveal that success depends on whether visually and structurally complex interface state is converted into a form the model can actually use [17,18]. The general principle is therefore not to expose all available state, but to expose decision-relevant state in a faithful and usable representation. The dominant trade-off is richness versus tractability: rich observations improve grounding, but increase context cost and distractors; compressed observations are easier to process, but may discard task-critical evidence. An open problem is to design observation interfaces that remain faithful and decision-useful under partial observability, multimodal state, and long trajectories.

5.2. Context Manager

Context management first emerged as a central concern in agentic workflows. Within the harness, it becomes one runtime component among observation, control, action, persistence, and verification. The context manager $\mathcal{C}$ determines which available information enters the current model call and in what form. It selects, compresses, orders, and refreshes observations, tool outputs, retrieved evidence, memory records, summaries, instructions, and task artifacts before they become the working context for the next step [142,166–168]. Its main design choices concern inclusion, representation, refresh policy, and the amount of shared state exposed to active agents or sub-agents. As context engineering suggests,

 doi:10.20944/preprints202606.1312.v1

long-horizon performance depends less on simply increasing prompt length than on maintaining coherent task state over time [29,144,169–171].

Several implementation patterns recur. Retrieval-based systems bring in external documents or stored state on demand. Memory-oriented systems such as MemGPT separate the active context from a larger external memory [31]. Industrial harnesses increasingly externalize task state into explicit artifacts and selectively resurface those artifacts, rather than relying on a single ever-growing dialogue trace [146]. The dominant trade-off is fidelity versus manageability: raw histories preserve detail but scale poorly, whereas summaries and retrieved context are cheaper but can omit or distort important state. Thus, the crucial distinction is not between long and short prompts, but between monolithic and managed context. A key open problem is how to preserve summary faithfulness and state integrity while keeping context cost bounded, especially when context repair must interact with verification and recovery in long-horizon settings [172].

5.3. Control Loop

The control loop $\mathcal{L}$ organizes execution across steps, tools, and possible handoffs. It turns observation, reasoning, action, and feedback into a runnable process. This component determines whether the agent follows a simple perceive-act cycle, a ReAct-style loop, a plan-execute-verify routine, or a hierarchical and multi-agent workflow [38,108,136–138]. The main design questions are how control is divided between the model and the runtime, when plans are created or revised, when delegation is introduced, whether coordination is sequential or parallel, and when execution should stop. Long-horizon success depends not only on the model's reasoning quality, but also on whether the runtime keeps execution stable under uncertainty.

Existing systems occupy different points in this space. Some retain lightweight iterative loops, while others impose explicit planner-executor-verifier decomposition or multi-agent coordination. Recent harness-oriented work makes orchestration itself an optimization target: NLAH treats harness logic as an editable artifact, and Meta-Harness treats harness configuration as a searchable design space [23,24]. This layer is therefore not merely about adding steps. It is about choosing how much structure to impose on the trajectory. The dominant trade-off is adaptability versus stability: freer loops can respond to unexpected states, but are more prone to drift, repeated failure, and coordination overhead; stronger orchestration improves reliability, but can reduce efficiency or overconstrain exploration. A standing challenge is to design control policies that remain robust across horizons and domains without making delegation, verification, and recovery prohibitively expensive.

5.4. Action Interface

The action interface $\mathcal{I}_{\text{act}}$ maps model outputs to executable operations. It defines what the agent can do, how actions are specified, which permissions apply, and how action results are returned as subsequent observations. Recent tool-use studies further show that action-interface quality is a major source of agent reliability. Diagnostic work on tool invocation failures identifies cases where agents fail not because a tool is absent, but because the tool is poorly selected, invoked, or integrated into the execution trajectory [173]. ET-Agent studies behavior calibration for tool-integrated reasoning [174], and ToolTok represents tools as tokens to improve efficiency and generalization in GUI agents [175]. From a harness perspective, this layer shapes the agent's effective action space. Its design space spans tool granularity (low-level environments versus high-level APIs), tool specification (free-form commands versus structured schemas), routing and interoperability (local tools versus protocol-based ecosystems such as MCP), and governance (permissions, side-effect control, and invocation constraints). Existing work shows that performance depends not only on whether tools exist, but on how the action interface makes them usable, composable, and governable.

Representative implementations range from terminals and browsers to structured callable APIs. SWE-agent is a canonical example of observation-action co-design: redesigning the agent-computer interface changes both what the model sees and how it acts, producing gains under a fixed base model [21]. Protocol-oriented infrastructures such as MCP move tool access toward a standardized

 doi:10.20944/preprints202606.1312.v1

interface layer across heterogeneous services [65], while benchmarks such as MCPWorld and OSWorld-MCP test whether agents can invoke such services reliably in realistic environments [147,150]. The dominant trade-off is flexibility versus controllability: low-level tools are general but difficult to use robustly, whereas high-level tools improve reliability but may narrow behavior too aggressively. An open question is how to design action abstractions that remain expressive and governable when tool use is coupled with verification, sandboxing, and recovery over long horizons.

5.5. State and Artifact Store

The state and artifact store $\mathcal{S}$ persists execution state across steps, sessions, and subtasks. It provides continuity beyond the active context window by storing task progress, traces, plans, checkpoints, diffs, generated files, memory records, and other reusable artifacts. Its design space includes the granularity of stored state, the scope of persistence, the storage form, and the update policy. Long-horizon agents often fail not because no state is stored, but because the wrong state is preserved, the right state cannot be retrieved, or stale state is treated as current.

Several strategies recur in the literature. Some systems rely on session-level histories and checkpoint stores. Memory-oriented approaches, such as MemGPT, introduce explicit long-term memory beyond the current context [31]. Artifact-centered harnesses track state through logs, diffs, checkpoints, and inspectable runtime objects [146]. What these approaches share is a move from transient interaction traces toward reusable system state. The dominant trade-off is completeness versus usability: richer state improves continuity, auditability, and handoff, but increases retrieval burden, noise, and the risk of stale memory. The practical challenge is not to store more, but to decide what deserves persistence, what should be compressed, and what should be discarded. An important open problem is how to maintain state fidelity while supporting rollback, delegation, and memory reuse without accumulating drift or obsolete information [172].

5.6. Verification and Governance

The verification and governance layer $\mathcal{V}$ checks, constrains, and repairs execution during runtime. Verification includes tests, assertions, verifier models, judge signals, and other mechanisms for estimating whether execution is progressing correctly. Governance includes approval gates, sandboxing, budget control, rollback, retry, escalation, and safe termination. This view is reflected in recent multi-agent governance works. For example, AI Committee uses multiple agents for validation and remediation of web-sourced data [151], while act-or-refuse learning studies when agents should proceed, abstain, or stop during safe multi-step tool use [176]. These examples show that governance is not only a deployment constraint, but also an explicit decision layer within agent execution. These two roles are tightly coupled: verification produces evidence about the run, while governance determines what the harness is allowed or required to do with that evidence. Reliable agents therefore depend not only on strong reasoning and rich tools, but also on whether checks and constraints are mechanically enforced rather than left entirely to prompt obedience.

Representative systems make this dual role clear. In coding agents, tests, linters, and assertions provide relatively strong verification signals, while rollback and sandboxed execution contain side effects [16,20]. In web, desktop, and other open-ended environments, governance becomes more central because actions may be partially irreversible and clean oracles are often unavailable [17,18]. The dominant trade-off is autonomy versus robustness: looser constraints permit broader exploration, but increase the risk of harmful actions and unrecoverable drift; tighter governance improves safety and recoverability, but can slow execution or block useful behavior. An open problem is to design verification and governance mechanisms that are selective and cost-aware, so the harness can distinguish recoverable local errors from deeper task-level collapse without over-triggering interruption or rollback.

 doi:10.20944/preprints202606.1312.v1

5.7. Cross-Layer Interactions in the Harness

Although the six components are analytically separable, they do not operate independently. Design choices in one component often reshape the burden on others. The observation interface $\mathcal{I}_{\text{obs}}$ and context manager $\mathcal{C}$ are tightly coupled: richer observations can improve grounding, but they also increase the cost of selection, compression, and formatting before information enters the active context [125]. The action interface $\mathcal{I}_{\text{act}}$ interacts directly with verification and governance $\mathcal{V}$: more expressive actions expand capability, but require stronger permission control, sandboxing, rollback, and auditing. The state and artifact store $\mathcal{S}$ feeds back into context and verification because persistent plans, logs, checkpoints, and artifacts determine both what can be resurfaced to the model and what evidence is available for judging progress [31,172].

Harness design is therefore a coupled systems problem rather than the independent optimization of six modules. Improving one layer can shift risk elsewhere: stronger compression can reduce cost while weakening downstream verification; richer actions can improve task coverage while increasing governance pressure; more persistent state can improve continuity while also introducing stale or conflicting evidence. This coupling makes task structure part of harness design itself. Different domains, horizons, oracle strengths, and autonomy requirements place different pressure profiles over $\mathcal{I}_{\text{obs}}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{I}_{\text{act}}$, $\mathcal{S}$, and $\mathcal{V}$. The same anatomy therefore becomes a way to read the task landscape: tasks differ by which runtime responsibilities they stress and which configuration choices become decisive.

6. Task Landscape and Harness Configuration

Task structure determines which parts of the execution harness become performance-critical. Seen through the harness anatomy, a task is not merely an application label but a pressure profile over observation, context, control, action, state, and governance. Long horizons, partial observability, weak feedback, irreversible actions, and autonomy requirements shift pressure toward different configuration choices, including context selections, action abstractions, verifier loops, checkpoints, permission gates, and escalation rules. The central question is therefore which runtime responsibility becomes the limiting factor under a given task condition.

6.1. A Harness-Aware Task Taxonomy

Agent tasks differ not only by application label, but by structural properties that determine which harness components limit performance, reliability, cost, or safety. We use three dimensions to characterize these pressures: **task horizon, environment type, and autonomy level.** Together, they identify the primary bottleneck: the component or small set of components where failures most often concentrate.

Complexity and task horizon. Table 4 summarizes four coarse levels. The most consequential transition is usually from L2 to L3. Single-step and short multi-step tasks can often be handled with local reasoning, lightweight action access, and local checks. Long-horizon tasks create sustained pressure on the Context Manager, State and Artifact Store, and Control Loop because plans, intermediate artifacts, failed attempts, and partial results must survive beyond one prompt window. At L4, open-ended monitoring or exploration additionally requires budget control, stopping criteria, and escalation policies, shifting the bottleneck toward explicit Verification and Governance rather than generation quality alone.

 doi:10.20944/preprints202606.1312.v1

Table 4. Harness-aware task complexity levels and their primary bottlenecks.

Level	Description	Examples	Bottleneck
L1	Single-step	Search, translate, calculate	Context Mgr.; Verif. & Gov.
L2	Multi-step	Form filling, code generation	Act. Interface; State Store; Verif. & Gov.
L3	Long-horizon	Repo-scale coding, research	Context Mgr.; State Store; Ctrl. Loop
L4	Open-ended	Monitoring, auto exploration	Verif. & Gov.; Ctrl. Loop

Environment type. Environment type determines what the harness must observe and which actions it can expose [177–183]. Terminals and repositories stress the Action Interface, Observation Interface, and Verification and Governance components because commands, diffs, logs, and tests can often be wrapped as structured actions, observations, and verifier signals. Browser and desktop environments add visual or DOM grounding, session state, and persistent side effects, increasing pressure on observation construction, action abstraction, and permission boundaries. Knowledge environments, including web search, literature retrieval, and structured databases, shift the bottleneck toward the Context Manager and State and Artifact Store because the main challenge is managing evidence quality, provenance, and synthesis. Physical environments introduce real-time constraints and irreversibility, making the Control Loop and Verification and Governance more central than in purely digital settings. Social environments add norms, negotiation, and strategic responses, which raises the value of richer observation design and conservative escalation.

Autonomy level. Autonomy cuts across domains. Human-in-the-loop settings can route high-risk decisions through approval gates. Semi-autonomous systems delegate routine actions but escalate when uncertainty rises. Fully autonomous systems must absorb more of that burden inside the harness through verification, rollback, logging, and fail-safe termination. Autonomy is best understood as a multiplier on harness requirements: the more independently the system is expected to act, the more the Observation Interface and Verification and Governance move from optional guardrails to primary bottlenecks.

Taken together, these dimensions define a pressure profile over the six harness components. Task horizon mostly stresses context, state, and control; environment type mostly stresses observation, action, and safety boundaries; autonomy mostly stresses verification, logging, and recovery. This view turns task taxonomy into harness configuration: the key question is not simply which application domain a task belongs to, but which runtime responsibilities become bottlenecks under its structural pressures. The next subsection uses representative domains as case studies to illustrate this bottleneck migration in concrete settings.

6.2. Harness Adaptation by Domain

The following domains should be read as *instantiations* of the pressure-profile view above, not as a separate domain-only taxonomy. Each domain combines horizon, environment, oracle strength, irreversibility, and autonomy in a different way, thereby shifting the primary harness bottleneck. Software engineering is verification-dominant; web and GUI interaction is grounding-dominant; scientific discovery is synthesis-dominant; medical assistance is safety-dominant; and embodied agents are control-dominant. These cases show how the same harness anatomy leads to different configuration priorities once task pressures change. Table 5 later abstracts these examples into reusable configuration rules.

 doi:10.20944/preprints202606.1312.v1

Software engineering (verification-dominant). Software engineering places the primary bottleneck on Verification and Governance [184–187]. Builds, unit tests, linters, and execution logs provide mechanical feedback signals that most other domains lack [16]. These strong oracles enable closed-loop generate-test-repair cycles, where verifier output becomes the next iteration's evidence. Systems such as Claude Code, SWE-agent, and OpenHands further show that redesigning the Action Interface, including file and command interfaces, diff views, and patch inspection, can yield measurable gains at fixed model capability [6,10,21]. The State and Artifact Store is equally consequential: expressive action access must be paired with checkpoints, patch artifacts, safe rollback, and bounded side effects. As tasks scale from snippet-level generation (L2) to repo-scale issue resolution (L3-L4), the Context Manager and State and Artifact Store join the bottleneck set for tracking plans, failed hypotheses, and intermediate artifacts across long trajectories [188–191].

- **Configuration implication.** Verification-dominant settings turn runtime quality into a closed-loop optimization problem; the harness response is verifier loops, generate-test-repair cycles, and reversible execution.

Web and GUI interaction (grounding-dominant). Web and GUI agents shift the primary bottleneck from verification to grounding: the core difficulty is constructing observations the model can use and actions it can execute safely [17,18,150,192–196]. The Observation Interface must render screenshots, DOM state, interaction history, and session information into usable signals. The Action Interface must decide whether actions are exposed as brittle low-level selectors or as structured browser and desktop operations. The Context Manager then selects and compresses these signals for the current step. Because many web goals lack a single mechanical oracle, verification is structurally weaker than in coding. Navigation mistakes, form submissions, and account actions can produce persistent side effects, so Verification and Governance remains part of the bottleneck rather than a secondary concern. As tasks move from short form filling (L2) through multi-page workflows (L3) to long-lived monitoring (L4), the grounding burden compounds.

- **Configuration implication.** Grounding-dominant settings require co-design of observation and action interfaces; performance hinges on whether the harness exposes the right state in a form that also supports safe, robust action.

Scientific discovery and research (synthesis-dominant). Scientific agents shift the primary bottleneck to synthesis: the central runtime problem is trustworthy integration of evidence over long horizons [197–202]. The most stressed components are the Context Manager, State and Artifact Store, and Verification and Governance. Verification serves a different function here than in coding: rather than closing a test-based repair loop, it must assess provenance, source quality, and reasoning coherence. Tool-rich systems such as ChemCrow and BioDiscoveryAgent show that the Action Interface can expand capability, but without provenance tracking long-horizon reasoning can degrade into plausible but unsupported narrative. Research tasks are predominantly L3-L4: literature surveys, hypothesis generation, and experimental design require the harness to externalize evidence, intermediate claims, and artifacts more aggressively than in coding or web settings.

- **Configuration implication.** Synthesis-dominant settings lack closed-loop verification; provenance tracking, intermediate review stages, and artifact-centered memory must substitute for end-state oracles.

Medical applications (safety-dominant). Medical agents inherit the synthesis demands of research settings but add a qualitatively different constraint: the primary bottleneck shifts to safety [203–206]. Verification and Governance moves to the top of the harness stack, with the Observation Interface and Context Manager close behind [207]. Patient history, guidelines, and recent findings must be surfaced accurately; consequential actions must be permission-gated; and uncertainty must trigger conservative escalation rather than confident continuation. The objective is not maximal autonomy, but bounded

 doi:10.20944/preprints202606.1312.v1

and inspectable assistance. In this regime, Verification and Governance is a performance-critical harness component rather than a compliance add-on.

- **Configuration implication.** Safety-dominant settings optimize controlled delegation: approval gates, auditability, and conservative recovery are core harness components, not external overhead.

Embodied settings (control-dominant). Embodied agents shift the primary bottleneck to real-time control, elevating the Control Loop above other harness components [34,208–211]. High-level language reasoning is too slow for continuous interaction, so the harness typically becomes layered: deliberative planning at the top, reactive control below, and persistent skill or state representations connecting the two. The State and Artifact Store maintains goals, subgoals, maps, or reusable behaviors. Verification and Governance is critical because physical actions are often irreversible. The Action Interface exposes actuators, simulators, or perception modules rather than conventional software APIs. Embodied tasks span L3-L4 almost exclusively, making long-horizon state externalization a baseline requirement.

- **Configuration implication.** Control-dominant settings push part of the stack below the language loop, motivating tighter integration between harness design, lower-level controllers, and training-time adaptation.

Cross-domain synthesis. The five domains trace a single analytic thread: the primary bottleneck migrates across harness components as domain constraints change. It moves from Verification and Governance in coding, through Observation Interface and Action Interface in web/GUI, to Context Manager and State and Artifact Store in research, Verification and Governance in medicine, and Control Loop in embodied settings. This migration pattern shows that domain labels alone are insufficient descriptors. What matters for harness configuration is which component absorbs the failure budget.

6.3. From Task Properties to Harness Configurations

The domain case studies above illustrate bottleneck migration, but the reusable lesson lies below the domain level. Across domains, similar task properties induce similar harness responses: long horizons require externalized state, partial observability requires structured observation, strong oracles enable verifier loops, weak or delayed oracles require provenance and review, irreversible actions require governance, and high autonomy requires logging, budgets, and recovery. Table 5 summarizes these domain-independent configuration rules.

 doi:10.20944/preprints202606.1312.v1

Table 5. Mapping task properties to harness failure pressures and configuration responses.

Task property	Failure pressure	Harness response	Critical components
Long horizon	State drift	Checkpoints, summaries, artifacts	$\mathcal{C}, \mathcal{S}, \mathcal{L}$
Partial observability	Indirect state	Structured observations, grounding, abstraction	$\mathcal{I}_{\text{obs}}, \mathcal{C}, \mathcal{I}_{\text{act}}$
Strong oracle	Checkable outcomes	Verifier loops, repair cycles	$\mathcal{V}, \mathcal{L}$
Weak or delayed oracle	Uncertain success	Provenance tracking, review, approval	$\mathcal{V}, \mathcal{C}, \mathcal{S}$
Irreversible actions	Persistent side effects	Sandbox, gates, rollback	$\mathcal{V}, \mathcal{I}_{\text{act}}$
High autonomy or low latency	Limited human correction	Logging, budgets, controllers	$\mathcal{V}, \mathcal{L}, \mathcal{I}_{\text{obs}}$

 doi:10.20944/preprints202606.1312.v1

Verifier strength determines where configuration effort concentrates. Where strong automatic oracles exist, as in verification-dominant software engineering, the harness can invest heavily in closed-loop optimization through verifier loops and repair cycles. Where oracles are weak or delayed, as in synthesis-dominant research and safety-dominant medicine, the bottleneck migrates upstream toward provenance management, intermediate review, and conservative stopping criteria. Domains should therefore not be compared only by task success rates; they should also be compared by the quality and latency of feedback signals available to the harness.

Irreversibility and autonomy make constraints central. In read-mostly or reversible digital settings, recovery can often be handled through retries and checkpoints. In grounding-dominant web interaction, safety-dominant medical assistance, and control-dominant physical settings, actions can have persistent side effects. Verification and Governance therefore becomes part of the primary bottleneck rather than a peripheral add-on. Higher autonomy magnifies this pattern because the harness must absorb responsibilities that a human operator would otherwise carry.

Long-horizon performance depends on externalized state across all domains. Whether the task is repo-scale coding (L3), literature synthesis (L3-L4), or embodied exploration (L4), one prompt window is rarely the right unit of memory. Durable artifacts, summaries, checkpoints, plans, and logs keep trajectories coherent over time. The configuration consequence is that the Context Manager and State and Artifact Store must be designed jointly: summaries decide what is visible now, whereas artifacts and checkpoints decide what remains recoverable later.

Task pressure should be reported together with evaluation results. The mapping in Table 5 also constrains benchmark interpretation. Benchmarks are most informative when they stress the harness components that match the primary bottleneck of a target deployment setting. Benchmark reports should therefore describe not only the model, but also the task pressures and harness configuration under which results are obtained.

7. Evaluation and Empirical Analysis

Evaluation makes the model–harness interaction directly observable. Benchmark scores should therefore be interpreted as outcomes of a *model–harness pairing*: the same model may behave differently under different context policies, tool interfaces, control loops, verification procedures, and retry budgets. This section uses representative benchmarks to test this view across three interaction regimes: software-engineering tasks with strong test oracles, terminal tasks with command-line execution and environment manipulation, and web tasks with browser grounding and stateful interaction. Across these regimes, we try to answer three questions: *how much performance is explained by stronger backbone models, how much variation remains after conditioning on the model, and how runtime cost, latency, timeout behavior, and trace availability change the interpretation of task success*.

7.1. Benchmark Landscape and Evaluation Work

Existing evaluation work can be organized as a pipeline that turns an agent run into interpretable evidence: benchmarks specify the task, execution infrastructures standardize the run, judgment methods score and diagnose the outcome, and continuous evaluation practices feed these signals back into system improvement.

Benchmarks as task specifications. Table 6 summarizes representative benchmarks for LLM-based agents. Beyond application domains, these benchmarks stress different harness capabilities: SWE-bench [16] tests repository navigation, code editing, and hidden-test verification; WebArena [17], VisualWebArena [192], and OSWorld [18] test web/GUI grounding, state tracking, multimodal perception, and safe interface control; Terminal-Bench [20] tests command-line execution and environment manipulation; and LoCoMo [172] and OS-Harm [212] test memory persistence and harmful-action control. Together, they mark a shift toward ecologically realistic evaluation, where browsers, terminals, and operating systems reveal long-horizon failures that closed-form datasets often miss.

 doi:10.20944/preprints202606.1312.v1

Execution and trace infrastructure. For coding and terminal agents, systems such as SWE-agent [21], OpenHands [10], Repo2Run [213], R2E-Gym [214], and HAL [215] use controlled environments, sandboxed execution, standardized rollouts, and trace collection to make evaluation reproducible and diagnosable. This infra helps distinguish harness behavior from artifacts of dependency drift, invalid graders, changed tool interfaces, or inconsistent resets. Representative harness designs are summarized in Table 10.

Judgment and attribution methods. Executable tests and state-based checkers provide strong oracles for coding and terminal tasks, whereas open-ended outputs often require LLM-as-judge or human-audit protocols. G-Eval [216], MT-Bench [217], and surveys of LLM-as-a-judge [218] show the promise and risks of model-based evaluators, including bias, inconsistency, and evaluator drift. For harness engineering, judgment matters most when it attributes failures to model reasoning, context construction, tool exposure, execution control, safety constraints, or the evaluator itself.

Continuous evaluation practices. Frameworks such as LangChain's agent evaluation tooling [219], DeepEval [220], RAGAS [221], and lm-evaluation-harness [222] support recurring tests, trace inspection, judge-based metrics, and monitoring-style evaluation. Their role is not merely to report a leaderboard score, but to make evaluation reusable during prompt changes, tool updates, context-policy revisions, and deployment monitoring.

Table 6. Representative benchmarks for LLM-based agents.

Benchmark	Focus	Environment	Primary metric
AgentBench [223]	General	Interactive envs	Task completion
SWE-bench [16]	Coding	Real GitHub issues	Resolution rate
WebArena [17]	Web	Realistic websites	Task success
VisualWebArena [192]	Multimodal web	Visual web tasks	Task success
OSWorld [18]	Desktop	Real OS	Multi-app success
Terminal-Bench [20]	Terminal/Coding	Command-line	Task success
MCPWorld [147]	API+GUI	Hybrid tool envs	Task success/tool use
OS-Harm [212]	Safety	Desktop computer	Harmful action rate
LoCoMo [172]	Long-term mem.	Multi-session chat	QA/consistency
MMAU [224]	General	Cross-domain	Capability scores
MLE-Bench [225]	ML engineering	Kaggle-like tasks	Performance tier
MCPAgentBench [149]	MCP tool use	MCP sandbox	Task Compl./eff.
MCP-Atlas [148]	MCP tool use	Real MCP servers	Pass rate
GAIA [226]	General assistant	Web/files/tools	Answer accuracy
Claw-SWE-Bench [227]	Agent harnesses	Real GitHub issues	Resolution rate/cost
TheAgentCompany [19]	Enterprise-style	Simulated company	Task success

7.2. Evaluation Dimensions Beyond Task Success

Most public agent leaderboards still rank systems by a single outcome-centric score. SWE-bench [16] reports the percentage of resolved issues [228], while Terminal-Bench [20] primarily report task-level completion scores [229]. However, recent evaluation studies [21,23,24] increasingly argue that accuracy alone is insufficient for agent assessment. For example, Sayas *et al*. [230] calls for cost-controlled and reproducible evaluation. CLEAR [231] explicitly evaluates cost, latency, efficacy and assurance. ReliabilityBench [232] studies consistency, robustness, and fault tolerance under production-like stress. Procedure-aware evaluation [233] shows that apparent task completion can hide unsafe or invalid trajectories. Because an agent run couples model reasoning, harness design, environment setup, tool interfaces, and evaluator logic, a failure may originate from any part of this chain. Task success remains the primary outcome metric, but **meaningful harness comparison requires a richer reading of results along several additional dimensions:**

- **Task success:** whether the final objective is completed.

 doi:10.20944/preprints202606.1312.v1

- **Reliability:** whether performance remains stable across stochastic runs, retries, and environment variations.
- **Efficiency:** token usage, API cost, and compute cost.
- **Latency:** wall-clock time or number of interactions.
- **Safety:** whether actions remain within allowed boundaries and avoid harmful side effects.
- **Process quality:** whether the trajectory is inspectable, recoverable, and evidence-backed.

These dimensions explain why similar final scores can hide substantial harness differences. One harness may trade long trajectories, repeated retries, and heavy context accumulation for higher success, while another may deliver slightly lower success at much lower cost and latency. From a deployment perspective, the key is not raw success alone, but useful task completion under resource constraints.

In the following empirical analyses, we focus on the dimensions that are most consistently available across public reports and leaderboard logs: task success, runtime, timeout behavior, and token usage when available. Monetary cost is discussed only cautiously because public cost fields are sparse and often depend on harness-specific accounting, cache handling, and model-price assumptions.

7.3. Harness Effects on SWE-bench Verified

SWE-bench Verified [16] comprises 500 human-validated GitHub issues drawn from twelve Python repositories, each paired with a hidden test suite that provides a deterministic pass/fail oracle. Evaluations run inside sandboxed Docker containers with pinned dependencies, so observed differences reflect system capabilities rather than environmental artifacts. By filtering out ambiguous specifications and broken tests from the original 2,294-instance SWE-bench, the Verified split offers a cleaner signal for cross-system comparison. Solving an instance requires localizing the defect, editing source files, and passing hidden regression tests; because different harnesses partition these stages in distinct ways, the benchmark is well suited for studying how scaffold design affects measured performance.

Table 7 compares SWE-bench Verified results across several model–harness pairings, including open-source scaffolds, source-reported agent harnesses, lightweight mini-SWE-agent runs, and closed-source vendor reports. The table should be read as a compact synthesis of public evidence rather than a fully controlled factorial experiment. Model snapshots, reasoning settings, retry budgets, and proprietary scaffold details are not always aligned across sources, so the strongest comparisons are those within the same row family or under the same reported evaluation setting. Vendor-reported scores are included as upper-envelope references, but they should not be interpreted as controlled ablations against open-source harnesses. For example, Opus 4.5 with mini-SWE-agent is reported with extended thinking, whereas the same source reports 74.4% under medium reasoning and 67.6% for Opus 4; the OpenAI mini-SWE-agent row follows a GPT-5-2 extended-thinking style leaderboard setting, whereas GPT-5 (2025-08-07) under medium reasoning is reported at 65.0%, and the vendor 80.0% reference is a GPT-5.2 result reported in a Claude system card [228,234]. Similarly, DeepSeek rows distinguish V3 from V3.2 high-reasoning settings, and Gemini rows distinguish Gemini 3 Pro from later Gemini 3.1 Pro reports, including an 80.6% Gemini 3.1 Pro result under its own reported configuration [235]. These boundaries do not invalidate the table, but they mean the strongest claims should use within-row-family comparisons and treat vendor or reasoning-policy changes as upper-envelope evidence rather than controlled ablations.

 doi:10.20944/preprints202606.1312.v1

Table 7. Model–harness results on SWE-bench Verified. Resolved rates are percentages, resolved counts are out of 500 instances; cost is USD per instance when reported; vendor rows are proprietary references.

Primary model	Harness / scaffold	Res. (%)/ solved	Cost($)
GPT-4o	SWE-agent	23.2 / 116	-
	AutoCodeRover-v2	38.4 / 192	-
	Agentless	38.8 / 194	-
Claude 3.5 Sonnet	SWE-agent (20240620)	33.6 / 168	-
	SWE-agent + tools	49.0 / 245	-
	Agentless	50.8 / 254	1.19
	AutoCodeRover	51.8 / 259	4.50
	OpenHands + CodeAct 2.1	53.0 / 265	0.78
	PatchPilot	53.6 / 268	0.99
Claude 3.7 Sonnet	mini-SWE-agent	52.8 / 264	-
	SWE-agent + tools	63.2 / 316	-
	Vendor scaffold	*63.7* / -	-
Claude Sonnet 4	mini-SWE-agent	64.9 / 325	-
	OpenHands + CodeAct 2.1	70.4 / 352	-
	SWE-agent + tools	72.4 / 362	-
	Vendor scaffold	*72.7* / -	-
Claude Opus 4 / 4.5	SWE-agent + tools	73.2 / 366	-
	mini-SWE-agent	76.8 / 384	-
	OpenHands + CodeAct 2.1	77.6 / 388	-
	Vendor scaffold	*80.9* / -	-
o3 / o4-mini	PatchPilot v1.1	64.6 / 323	-
OpenAI GPT-5 family	OpenHands + CodeAct 2.1	71.8 / 359	-
	mini-SWE-agent	72.8 / 364	-
	Vendor scaffold	*80.0* / -	-
GPT-5.3 Codex	mini-SWE-agent	78.0 / 390	-
GPT-5.4	mini-SWE-agent	78.2 / 391	-
GPT-5.5	mini-SWE-agent	82.6 / 413	-
Gemini 3 Pro	mini-SWE-agent	74.2 / 371	-
	Vendor scaffold	*76.2* / -	-
Gemini 3.1 Pro Preview	mini-SWE-agent	78.8 / 394	-
DeepSeek V3 / V3.2	Agentless	42.0 / 210	-
	mini-SWE-agent	70.0 / 350	-
Claude Opus 4.6 Thinking	mini-SWE-agent	78.2 / 391	-
Claude Opus 4.7	mini-SWE-agent	82.0 / 410	-

Notes. Claude 3.5 Sonnet entries refer to the 2024-10-22 snapshot where specified; Claude Sonnet 4 and Opus 4 source-card entries use the 2025-05-14 generation. Some rows differ in inference policy, including extended-thinking or high-reasoning settings. Vendor rows use closed-source scaffolds.

The compared harnesses span a broad spectrum of scaffold complexity. Agentless [236] removes the interactive agent loop and uses a fixed localize–repair–validate pipeline. SWE-agent + tools [21,237] exposes shell and editing tools through a bash-oriented repair loop, while mini-SWE-agent [238] reduces this design to a minimal scaffold that leaves most orchestration to the model. OpenHands + CodeAct 2.1 [10] provides a richer software-engineering runtime with file editing, web browsing, and IPython execution. AutoCodeRover [239] and PatchPilot [240] represent more structured repair workflows, using repository search, localization, reproduction, validation, and refinement to constrain the repair process. Vendor scaffold lists the best vendor-reported scores on proprietary scaffolds [234, 241–244] as an upper-envelope reference. Together, these systems provide a useful, though not perfectly

 doi:10.20944/preprints202606.1312.v1

controlled, view of how scaffold design interacts with backbone model capability on repository-level coding tasks.

Specifically, **model capability and harness design both contribute to measured performance.** Within a single harness, backbone upgrades drive large gains: SWE-agent + tools improves from 49.0% with Claude 3.5 Sonnet to 73.2% with Opus 4, a 24% increase [237,245]; mini-SWE-agent shows a comparable trajectory from 52.8% (Claude 3.7 Sonnet) to 76.8% (Opus 4.5) [228]. Within a single model, harness choice also produces a consistent effect. For GPT-4o, source-reported harnesses range from 23.2% with SWE-agent to 38.8% with Agentless. For Claude 3.5 Sonnet, they range from 33.6% with SWE-agent to 53.6% with PatchPilot, with SWE-agent + tools, Agentless, AutoCodeRover, and OpenHands + CodeAct 2.1 occupying the middle of the range. For Claude Opus 4/4.5, the spread is narrower but still visible: 73.2% with SWE-agent + tools, 76.8% with mini-SWE-agent, and 77.6% with OpenHands + CodeAct 2.1. These ranges show that the same backbone can gain or lose tens of resolved instances depending on the scaffold.

Scaffold complexity does not predict effectiveness. Under Opus 4.5, mini-SWE-agent (roughly 100 lines of Python) reaches 76.8%, only slightly below the far richer OpenHands + CodeAct 2.1 sandbox at 77.6% [228,245]. These results suggest that scaffold effectiveness depends more on interface design than on feature count: a minimal scaffold with well-chosen primitives can extract nearly the same performance as a full-featured agent framework.

Vendor-reported scores, which reflect proprietary scaffold optimization, consistently exceed the best open-source results. OpenHands with Opus 4.5 at 77.6% [245] trails the corresponding vendor score of 80.9% [234] by about 3%; for Gemini 3 Pro, the gap narrows to 2% (74.2% *vs*.76.2%) [238,244]. For GPT-5 variants the margin is larger (72.8% *vs*.80.0%), though differences in model version and inference configuration complicate this comparison [228,234]. Across same-generation Claude and Gemini models, this 2-4% advantage is attributable to scaffold-level decisions such as prompt design, candidate selection, and compute scaling, not to differences in model capability.

7.4. Harness Effects on Terminal-Bench 2.0

Terminal-Bench provides a complementary perspective to SWE-bench because the agent must operate through an interactive command-line environment rather than only submit a final patch [20]. Each task specifies a natural-language instruction, a sandboxed terminal workspace, an executable test script, and a reference solution, so success is defined by whether the agent transforms the environment into a passing state. Tasks commonly require file inspection, tool installation or invocation, command execution, log interpretation, artifact editing, and explicit termination decisions. The benchmark is therefore well suited for studying execution harnesses, since terminal interaction jointly exercises observation design, context management, control-loop policy, action exposure, state persistence, and verification.

Our analysis uses the official Terminal-Bench 2.0 leaderboard and public submission logs as data sources [229,246]. Official submissions evaluate `terminal-bench@2.0` with five trials per task (`-k 5`), use each task's benchmark environment and default constraints, and must not override timeouts or CPU, memory, and storage limits [246]. Leaderboard-integrity rules further penalize reward-hacking trajectories, such as retrieving task solutions from the internet, which reduces the risk that reported scores reflect benchmark leakage rather than terminal task completion [247]. For the performance analysis, we use entries for which the backbone model and harness are identifiable, excluding entries whose model field is a mixture or "Multiple" so that each plotted point has a clear model identity. The resulting evidence is not a randomized ablation, since public submissions may differ in prompts, versions, budgets, and implementation details. It is nevertheless informative as an observational comparison: the same model appears under several harnesses, and the same harness often appears with several models. For resource analysis, we use the official HuggingFace public-submission repository as the primary source and align submissions to the currently visible leaderboard by strict metadata matching. The public repository contains 75 submissions with metadata and logs, covering 32,604 trial records. Among these, 48 submissions strictly match a currently visible leaderboard entry. Reward,

 doi:10.20944/preprints202606.1312.v1

agent-runtime, and full-runtime fields have high coverage (97.2%, 98.1%, and 100.0%, respectively), whereas input/output token fields cover 45.0% of trial records and dollar-cost fields cover only 15.2%. Accordingly, runtime and timeout statistics are the main resource-efficiency evidence, while token statistics are used as supplementary evidence and monetary cost is not used for cross-harness claims. These counts define the evidence boundary rather than a separate result table: 27 public submissions are excluded from the complete resource table because 24 are not visible or name-mismatched on the current leaderboard and 3 are ambiguous matches. A looser visible-entry check finds 55 matches among 142 visible leaderboard entries (38.7%) across 51 unique submissions, but it is used only as a coverage sanity check. The row-level comparisons below use the strict 48-row subset. Table 8 extracts representative same-model rows from this subset for the main-text comparison.

Table 8. Terminal-Bench 2.0 representative resource statistics for leaderboard-visible submissions. Input/Output report median token counts in thousands per trial, Agent reports median runtime in minutes, and TO reports timeout rate.

Model	Harness	textbfScore (%)	Trials	textbfInput (K)	textbfOutput (K)	textbfAgent (min)	textbfTO (%)
GPT-5.3 Codex	SageAgent	78.4	445	-	-	5.7	12.1
	Mux	74.6	445	238.7	5.7	5.5	8.1
	Terminus 2	64.7	445	58.4	20.5	8.9	20.7
Claude Opus 4.6	Meta-Harness	76.4	445	755.0	15.8	6.3	7.9
	Terminus-KIRA	74.7	445	618.6	16.9	9.6	18.2
	Mux	66.5	445	213.0	10.1	5.9	10.3
	Terminus 2	62.9	445	79.4	8.4	5.3	18.2
Gemini 3.1 Pro	TongAgents	80.2	445	-	-	11.1	21.1
	Terminus-KIRA	74.8	445	257.5	22.1	6.5	9.7

Figure 6 first indicates that **backbone capability remains a major determinant of success.** Within a fixed harness, stronger model generations often improve resolved rates by more than 10%. For example, under Terminus 2, GPT-5 improves from 35.2% to 54.0% with GPT-5.2 and to 64.7% with GPT-5.3-Codex. Under Codex CLI, the sequence from GPT-5 to GPT-5.2 and GPT-5.5 rises from 49.6% to 62.9% and 82.2%. The same pattern appears for Anthropic and Google models: newer Opus, Gemini, and Codex-specialized models generally occupy higher regions of the plot than earlier or smaller models. Thus, the benchmark does not support a harness-only interpretation; terminal agents still need strong planning, coding, debugging, and tool-use priors from the foundation model.

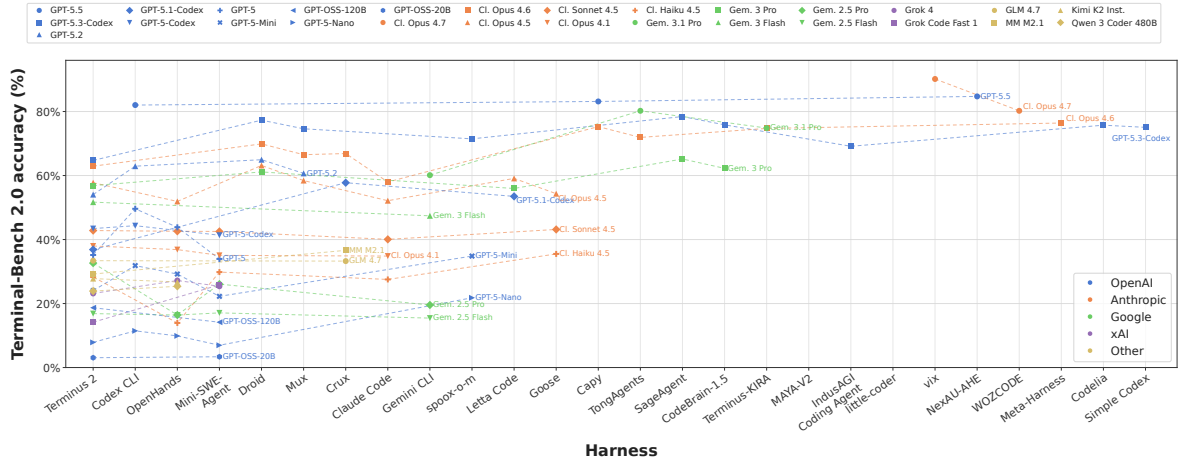

Figure 6. Terminal-Bench 2.0 accuracy across model–harness pairings. Each point is a single-backbone leaderboard entry, and dashed lines connect results that use the same model under different harnesses.

At the same time, conditioning on the model reveals large harness-induced spreads. GPT-5.3-Codex ranges from 64.7% with Terminus 2 to 78.4% with SageAgent, a 13.7% difference. Claude Opus 4.6 ranges from 58.0% with Claude Code to 76.4% with Meta-Harness, an 18.4% difference. Gemini 3.1 Pro ranges from 59.4% with Gemini CLI to 80.2% with TongAgents, a 20.8% difference. Even GPT-5,

 doi:10.20944/preprints202606.1312.v1

before the later Codex-specialized variants, ranges from 33.9% with Mini-SWE-Agent to 49.6% with Codex CLI. These gaps are substantially larger than the standard errors reported for most relevant leaderboard entries, and they correspond to different harness choices around terminal affordances, context packaging, command execution, stopping criteria, and recovery.

Figure 7 aggregates this fixed-model view. Among the 20 models that have at least three observed harness results, the median within-model range is 13.6% and 14 of the 20 models vary by at least 10% across harnesses. The largest spreads exceed 20%, as seen for Claude Haiku 4.5, GPT-5.1-Codex, and Gemini 3.1 Pro. This distribution shows that leaderboard accuracy cannot be attributed to the model alone. A model's measured terminal competence also depends on whether the harness exposes an effective command interface, preserves relevant execution state, routes observations back into context at an appropriate granularity, and uses verification signals for termination, retry, or repair.

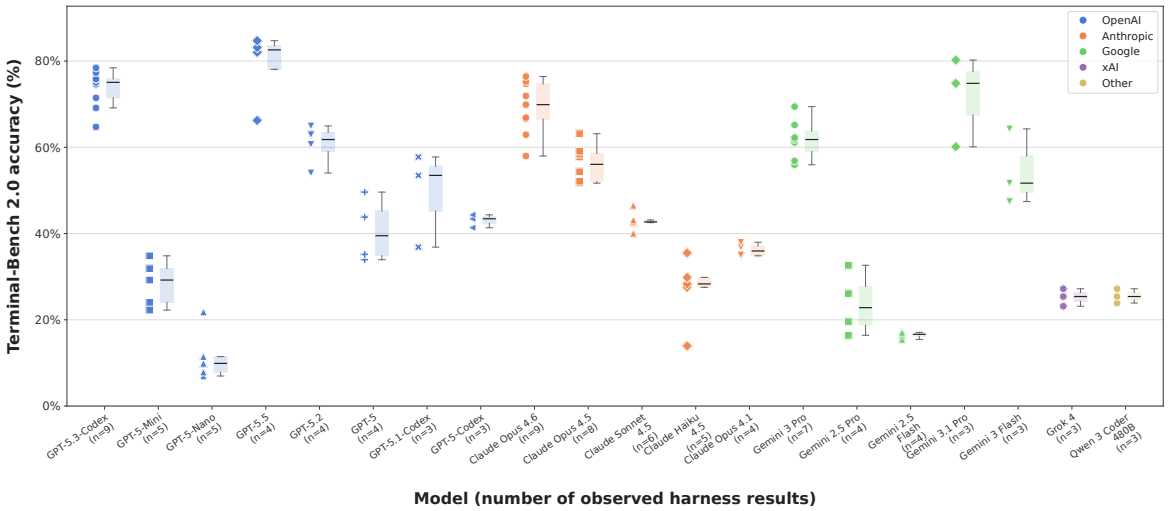

Figure 7. Within-model variation on Terminal-Bench 2.0. For each model with at least three observed harness results, the box and points summarize accuracy across harnesses.

Table 8 shows that accuracy differences are accompanied by distinct operational profiles. For GPT-5.3-Codex, SageAgent reaches the highest single-backbone score in Figure 6 with a median agent time of 5.7 minutes and a 12.1% timeout rate, whereas Terminus 2 takes 8.9 minutes and times out on 20.7% of trials. Mux uses a heavier median input context than Terminus 2 (238.7K versus 58.4K tokens) but reports shorter median agent time (5.5 versus 8.9 minutes) and a lower timeout rate (8.1% versus 20.7%). For Claude Opus 4.6, Meta-Harness uses a much larger median input context than Terminus 2 (755.0K versus 79.4K tokens) while reducing timeout rate from 18.2% to 7.9%. These examples show that the measured system behavior includes not only whether a task is solved, but also how much context is consumed, how long execution takes, and how often the harness fails to terminate successfully.

Taken together, Terminal-Bench results separate two effects without treating either as sufficient on its own. Model upgrades lift whole harness families, but harness choices can still shift the same model's score by more than 10% through terminal-state presentation, context management, command execution, and verifier feedback. Resource statistics add a deployment-facing caveat: higher accuracy may require longer trajectories, heavier context, or more robust timeout handling, and these costs are part of the practical capability being measured. On Terminal-Bench, reliable terminal task completion therefore depends on the fit between the model's interactive skills and the harness's runtime design under fixed benchmark constraints.

7.5. Harness Effects on WebArena

WebArena [17] is one of the most widely used reproducible benchmarks for web agents. It evaluates agents in self-hosted websites that cover realistic domains such as shopping, discussion forums,

 doi:10.20944/preprints202606.1312.v1

GitLab-style software collaboration, content management, maps, and knowledge resources. Unlike open-ended browsing benchmarks that often require human or LLM-based judgement, WebArena uses programmatic success checks over website state and task-specific answers. This makes it useful for studying what a web-agent harness contributes beyond a model-only baseline: browser agents must convert textual goals and visual or DOM observations into navigation, search, form filling, state tracking, and recovery actions, while the evaluator supplies a relatively concrete task-success signal.

Table 9 summarizes sparse WebArena results as backbone-level evidence slices, including both model-only references and harnessed agent systems. Rows containing the same backbone under multiple settings provide the strongest evidence for harness effects. Because most public WebArena reports describe complete agent systems rather than controlled factorial ablations, differences in prompts, browser actions, observation formats, retry policies, search budgets, training data, and implementation details should be treated as part of the reported system configuration [17,248,249].

Table 9. WebArena task-success evidence by backbone. Scores are percentages; Span reports the high–low difference in percentage points for each backbone.

Backbone	Model only	Harness low	Harness high	Span
GPT-3.5	8.9	22.0	29.1	20.2
GPT-4	14.9	20.2	33.0	18.1
GPT-4o	13.1	19.2	54.6	41.5
GPT-4 Turbo	16.5	33.3	45.7	29.2
GPT-4o-mini	-	13.6	13.6	-
GPT-5	-	71.2	71.2	-
DeepSeek R1-Llama 8B	8.5	43.6	43.6	35.1
DeepSeek V3.2	-	74.3	74.3	-
Gemini 3 Pro	-	51.2	71.6	20.4
Gemini 3.1 Flash-L	-	42.3	42.3	-
Claude Sonnet 3.5	-	36.2	52.1	15.9
Qwen3.5 family	-	3.1	41.5	38.4
Llama 3-70B	7.6	10.1	10.1	2.5
Llama 3.1-70B	-	18.4	18.4	-
Llama 3.2-1B	2.4	24.1	24.1	21.7
Llama 3.1-8B	5.6	48.5	48.5	42.9

The clearest harness effects come from backbones that have both model-only and harnessed results. GPT-4o ranges from 13.1% in the model-only baseline to 54.6% with WebOperator, a 41.5% span; even among named harnesses alone, it ranges from 19.2% with LM-TS to 54.6% with WebOperator [250,251]. GPT-4 improves from 14.9% to 33.0% with SteP, GPT-4-Turbo from 16.5% to 45.7% with AgentOccam, and GPT-3.5 from 8.9% to 29.1% under the stronger WebPilot entry [252–254]. The same phenomenon appears for open-weight or distilled models: DeepSeek-R1-Distill-Llama-8B moves from 8.5% to 43.6% with AgentSymbiotic, Llama-3.2-1B from 2.4% to 24.1%, and Llama-3.1-8B from 5.6% to 48.5% [255]. These gaps are too large to be explained by task noise alone; they reflect how observation design, search, workflow memory, action grounding, and stopping policies transform a language model into an effective web actor. Futhermore, WebTactix with DeepSeek V3.2 reaches 74.3%, corresponding to 594 solved tasks out of 812 in its public report [256]. OpAgent reaches 71.6% with Gemini 3 Pro, and ColorBrowserAgent reaches 71.2% with GPT-5 [257,258], showing that recent web-agent systems can exceed the 70% level on WebArena, but they combine strong backbones with specialized runtime structure, grounding, search, and adaptive memory; they should therefore be treated as model–harness system results.

 doi:10.20944/preprints202606.1312.v1

Fixed-harness comparisons show the complementary role of model capability. BrowserGym [249] provides the broadest same-harness slice: scores range from 51.2% for Gemini 3 Pro to 42.3% for Gemini 3.1 Flash-L, 41.5% for Qwen3.5-27B, 36.2% for Claude 3.5 Sonnet, 31.4% for GPT-4o, 23.5% for GPT-4, 18.4% for Llama-3.1-70B, and 13.6% for GPT-4o-mini [249]. Within Qwen3.5, performance falls monotonically from 27B to 9B, 4B, and 2B, indicating that stronger backbones generally improve planning, instruction following, and state tracking under a common browser interface. Yet model size does not fully determine outcomes: GPT-4o under BrowserGym trails Gemini 3.1 Flash-L and Qwen3.5-27B, while AgentSymbiotic with Llama-3.1-8B exceeds several larger-model BrowserGym results. The same backbone can be under-expressed by one harness and amplified by another.

Overall, WebArena reinforces the model–harness view from a web-interaction setting. In coding benchmarks, the harness shapes how tests, edits, and repository context are exposed; in WebArena, it shapes how a model sees the page, chooses browser actions, recovers from navigation errors, and verifies that a website state has changed as intended. Programmatic scoring reduces evaluator drift compared with LLM-as-judge protocols, but it does not remove all measurement risk: brittle checkers, ambiguous instructions, and environment leakage can still distort results. For high-confidence claims, audited variants such as WebArena Verified [259] are preferable when available because they retain the reproducible WebArena environment while repairing evaluator and instruction artifacts. Consequently, browser-agent success is best reported as a conditional property of a complete model–harness system, together with its observation mode, action interface, search or retry budget, memory policy, and source artifacts.

7.6. Benchmark Insights

Several lessons emerge from comparing harness behavior across benchmarks.

Harness design should match the benchmark oracle. When benchmark provides strong automatic feedback, as in coding tasks with tests, effective harnesses exploit verifier loops, patch refinement, and rollback. Terminal-Bench reinforces the same principle in command-line environments: useful harnesses turn command output, files, and completion checks into actionable feedback for termination, retry, and repair. When the oracle is weak or delayed, as in research or workplace tasks, the harness must rely more on provenance tracking, intermediate review, and conservative stopping.

Autonomy and complexity are not monotonic goods. Fully open-ended loops explore broadly but can accumulate context, drift, and cost. When objectives are narrow and success is mechanically checkable, structured pipelines such as localization–repair–validation can outperform more agentic loops, and compact scaffolds can match richer runtimes. The key design question is not how much autonomy the harness exposes, but which degrees of freedom help the model exploit the benchmark's feedback structure.

Model–harness compatibility matters. A strong model may perform poorly under a harness that exposes the wrong action space or overloads the context window. Conversely, a lightweight scaffold can be effective when it matches the model's preferred interaction pattern and the benchmark's feedback structure. On Terminal-Bench 2.0, the same model can vary by double-digit accuracy across harnesses; on WebArena, the gap between model-only baselines and browser-agent scaffolds can exceed 40% for GPT-4o. These differences make compatibility an empirical property of the model–harness pair rather than an implementation detail.

Scores are conditional on runtime configuration. Tool privileges, context policy, retry budget, sandbox restrictions, and completion criteria all shape measured performance. The Terminal-Bench public logs further show that runtime and timeout profiles vary substantially even among leaderboard-visible submissions. Thus, a benchmark score is interpretable only together with the runtime configuration that produced it. Reports should include at least model version, harness identity, tool privileges, retry and timeout policy, execution environment, token or API usage when available, and trace or verifier metadata.

Toward value-aware evaluation. The empirical results support a shift from score-centric ranking to value-aware agent evaluation. Stronger models raise the ceiling, but harness design determines how

 doi:10.20944/preprints202606.1312.v1

much of that capability becomes reliable, efficient, and auditable task completion. Future evaluation should therefore measure not only task success, but also resource use, latency, timeout behavior, recovery quality, safety constraints, and trace auditability. This observation motivates the value-aware objectives in Sec. 8, where success is evaluated jointly with cost, latency, risk, reliability, and process quality.

8. Outlook and Future Directions

Future agent progress will require more than stronger foundation models or richer benchmarks. We highlight three coupled directions: value-aware evaluation that accounts for success, cost, latency and safety; agent-native training that moves beyond planning and tool use toward verification and recovery; and harness design that adapts foundation models to task-specific tools, contexts, and constraints. Together, they connect near-term harness engineering with longer-term efforts to internalize reliable interaction, adaptation, and self-improvement into agent models.

8.1. From Score to Value-Aware Agent Optimization

Current agent leaderboards [228,229] are largely score-centric: systems are ranked by task success, while API cost, latency, safety, and trace quality are secondary or missing. This is useful for frontier comparison but incomplete for deployment, where cost-controlled, reliability-oriented, procedure-aware, and enterprise evaluation all point toward multi-dimensional agent quality [25,230–233,260].

A natural way to express this shift is to move from raw task success to value-aware agent optimization. Let $\tau \sim \mathcal{D}$ denote a task instance, and let $z = \mathrm{Run}(\tau; \mathcal{M}, \mathcal{H}, \omega)$ denote the execution trace produced by model $\mathcal{M}$ and harness $\mathcal{H}$ under stochasticity ω. For a trace z, let $S(z) \in \{0,1\}$ indicate task success, $C(z)$ execution cost, $L(z)$ latency, and $R(z)$ safety or compliance risk. Cost may include token/API usage, tool calls, compute, or infrastructure; latency may be wall-clock time or interaction steps; risk may come from policy violations, safety checkers, or human audits. Let $V(\tau) \geq 0$ denote task utility, such as user value, scientific value, priority, or risk-adjusted importance. The success probability of a model–harness pair can then be estimated from repeated runs as

$$P_{\mathrm{succ}}(\tau; \mathcal{M}, \mathcal{H}) = \mathbb{E}_{\omega}[S(\mathrm{Run}(\tau; \mathcal{M}, \mathcal{H}, \omega))]. \tag{5}$$

Let $Q_{\mathrm{proc}}(z) \in [0,1]$ summarize process quality, including trace inspectability, verifier use, recovery behavior, provenance quality, and policy compliance. Let $\mathrm{Rel}_k(\tau; \mathcal{M}, \mathcal{H})$ denote repeated-run reliability estimated from k runs, for example through consistency, pass@k, or stress-test reliability. Instead of maximizing success alone, value-aware optimization can be written as

$$\begin{aligned} \max_{\mathcal{M},\mathcal{H}} \quad & \mathbb{E}_{\tau\sim\mathcal{D}}\left[V(\tau)\, P_{\mathrm{succ}}(\tau; \mathcal{M}, \mathcal{H})\, \bar{Q}_{\mathrm{proc}}(\tau; \mathcal{M}, \mathcal{H})\right] \\ \text{s.t.} \quad & \mathbb{E}_{\tau,\omega}[C(z)] \leq B_C,\ \mathrm{Quantile}_p(L(z)) \leq B_L, \\ & \mathbb{E}_{\tau,\omega}[R(z)] \leq \epsilon,\ \mathbb{E}_{\tau}[\mathrm{Rel}_k(\tau; \mathcal{M}, \mathcal{H})] \geq \rho. \end{aligned} \tag{6}$$

where $\bar{Q}_{\mathrm{proc}}(\tau; \mathcal{M}, \mathcal{H}) = \mathbb{E}_{\omega}[Q_{\mathrm{proc}}(z)]$. This is not a universal leaderboard score; it makes the deployment target explicit by coupling task value with cost, latency, risk, and reliability constraints. The trade-off is task-dependent: high-value or high-risk tasks may justify stronger verification, while routine high-frequency tasks favor cheaper models, shorter trajectories, and stricter stopping policies.

Let $\tilde{C}(z) = C(z)/B_C$, $\tilde{L}(z) = L(z)/B_L$, and $\tilde{R}(z) = R(z)/\epsilon$ be normalized cost, latency, and risk. A complementary value-density objective is

$$\begin{aligned} \mathrm{VD}_{\alpha,\beta,\gamma} &= \mathbb{E}_{\tau,\omega}\left[V(\tau) S(z) Q_{\mathrm{proc}}(z) D(z)^{-1}\right], \\ D(z) &= (1+\tilde{C}(z))^{\alpha}(1+\tilde{L}(z))^{\beta}(1+\tilde{R}(z))^{\gamma}. \end{aligned} \tag{7}$$

 doi:10.20944/preprints202606.1312.v1

where α, β, and γ control the penalties on cost, latency, and risk. This is a family of deployment-specific utilities rather than a fixed metric. Different deployments can instantiate it differently: high-value tasks may tolerate stronger verification, high-frequency workflows may penalize latency and cost, and safety-critical settings may replace soft risk penalties with hard constraints. The same traces also support simpler reports, such as cost per effective success or latency per successful task, which distinguish systems with similar success rates but different runtime profiles.

From this perspective, harness engineering is a resource-allocation problem. The harness chooses models, context, memory access, tools, retries, verifiers, stopping rules, and human escalation. Model routing, context compression, cache reuse, verifier selection, recovery policy, and early stopping determine useful progress per unit cost, rather than being mere implementation details. Future benchmarks should therefore report success together with token/API cost, tool calls, retries, 95th-percentile (P95) latency, recovery behavior, policy violations, and trace auditability.

8.2. Learning to Verify, Recover, and Adapt

The value-aware view also suggests a path for agent learning. Execution traces are not only evaluation records; they contain outcomes, cost, tool calls, verifier signals, recovery attempts, policy violations, and feedback. Future agents should learn not only to plan and act, but also to verify intermediate states, diagnose failures, recover from local errors, and adapt across tasks.

A useful abstraction is a constrained self-evolution loop. Let θ_t denote the model parameters and ϕ_t denote the harness configuration at iteration t. Running the agent on tasks from $\mathcal{D}$ produces traces $\mathcal{Z}_t$, from which the system extracts an evidence set $\mathcal{E}_t$ containing outcomes, failure modes, verifier results, cost profiles, and safety events. An update operator U may then change the model, the harness, or both:

$$(\theta_{t+1}, \phi_{t+1}) = \text{VerifyRetain}(U(\theta_t, \phi_t, \mathcal{E}_t)). \tag{8}$$

Here VerifyRetain keeps an update only if it passes held-out tasks, regression tests, process checks, and safety constraints; otherwise it is rejected or rolled back. The expression is not a fixed algorithm; it makes the control structure explicit: reliable self-evolution must couple experience extraction, credit assignment, modification, and validation.

Existing agent-native training mostly advances the model side of this loop. Interactive RL and environment-based training reduce train–test mismatch in web, computer-use, and software-engineering agents [41,42,160,161]. Self-evolving systems further treat interaction experience as a reusable learning signal for self-questioning, attribution, online adaptation, and reward-free exploration [44,45,162,163]. Together, these works suggest that verification, recovery, and adaptation should become trainable behaviors, not only prompt-induced routines.

Parameter updates alone cannot absorb all runtime bottlenecks. Many failures arise from harness choices: observation format, action granularity, memory retrieval, or verifier timing. Agentic Harness Engineering (AHE) makes this harness-side path concrete by freezing the base model and evolving coding-agent harness components through observability-driven feedback [40]. Its key lesson is that self-evolution must be observable and falsifiable: components should be explicit and revertible, traces should be distilled into evidence, and proposed changes should make predictions that later outcomes can check.

The long-term direction is co-evolution of models and harnesses. Optimized harnesses produce better traces for training; trained models internalize recurring verification and recovery patterns; the resulting model changes which harness structure is optimal. This introduces risks such as benchmark overfitting, incorrect failure attribution, stale memory, and unsafe runtime modification. Agent-native training therefore does not eliminate the harness; it turns the harness into a training environment, evidence pipeline, verifier, and governance layer, with held-out evaluation, ablations, audit logs, rollback, and human approval for high-impact changes.

 doi:10.20944/preprints202606.1312.v1

Table 10. Representative harness designs behind agent benchmark performance. The table is not exhaustive; it highlights harness families that explain the empirical patterns in Sec. 7.

Harness	Control style	Key design	Strength	Typical limitation
SWE-agent [21]	ReAct-style loop	LM interacts with shell/editor tools and iteratively inspects, edits, and tests code.	Simple and general; strong baseline for real GitHub issues.	Can be unstable and token-expensive on long debugging trajectories.
mini-SWE-agent [238]	Minimal tool loop	Roughly 100-line scaffold exposing compact shell/edit actions and leaving most orchestration to the model.	High transparency; strong controlled comparisons across frontier backbones.	Relies on the model to manage planning, context, and recovery.
Agentless [236]	Fixed pipeline	Staged localization, repair generation, and patch selection without a fully autonomous interaction loop.	Stable, cheaper, and easier to reproduce.	Less adaptive when the issue requires exploratory debugging.
AutoCodeRover [239]	Search-guided repair	Repository-aware code search, AST-level localization, patch generation, and validation.	Strong at locating relevant files/functions before editing.	Depends heavily on localization quality and repo search signals.
OpenHands + CodeAct 2.1 [10]	General runtime agent	Full software-engineering runtime with shell, file editing, browser/tools, and iterative execution.	Flexible for broad coding tasks and long interactions.	Higher orchestration cost and larger action space.
PatchPilot [240]	Structured repair workflow	Reproduction, localization, generation, validation, and refinement are organized as a controlled pipeline.	Good cost-performance trade-off; validation-focused.	Less open-ended than fully interactive agents.
Codex / Claude Code [6,7]	Managed coding agent	Proprietary coding runtime tightly couples model, code execution, editing, and task management.	High end-to-end coding performance with productized recovery and state management.	System details are less transparent than open-source harnesses.
Meta-Harness [24]	Search-optimized harness	Treats prompts, tools, and runtime policies as a searchable harness design space.	Directly optimizes the harness rather than only the model.	Adds search cost and can overfit to benchmark-specific feedback.
SageAgent / OpenSage [229,261]	Generated agent scaffold	Uses a self-programming agent-generation engine to produce and refine executable agent scaffolds.	Strong Terminal-Bench results with relatively low observed runtime.	Public leaderboard evidence is observational rather than a controlled ablation.
Terminus 2 [229,246]	Reference terminal harness	Terminal-Bench native scaffold with shell execution, task state, verifier feedback, and benchmark constraints.	Useful anchor for cross-model and cross-harness terminal comparisons.	Can expose high timeout rates on difficult interactive tasks.
Terminus-KIRA [229,262]	Terminal-native agent	Tool-calling, terminal interaction, completion checks, and verification-oriented execution.	Strong for terminal-bench style tasks requiring environment manipulation.	Performance depends on robust task completion detection.
Mux [229,246]	Lightweight terminal harness	Minimal terminal-oriented scaffold for executing and verifying tasks.	Simple and relatively transparent.	Weaker planning and recovery compared with richer runtimes.
TongAgents [229,246]	Terminal agent system	Submission-level terminal harness combining command execution, state tracking, and completion control.	Strong observed Gemini 3.1 Pro Terminal-Bench result.	Design details are less documented than paper-backed harnesses.

8.3. Harness Generalization Versus Specialization

The previous two directions raise a systems question: should a harness be reusable across tasks or specialized for one environment? The answer depends on which harness layer is being considered. Tracing, sandboxing, permission control, artifact storage, budget management, model routing, and basic tool protocols can form a reusable substrate. Observation shaping, action abstraction, memory policy, verifier design, and recovery strategy are more often tied to the task pressure profile.

This distinction explains why realistic benchmarks are both specialized and compositional. Software-engineering benchmarks stress verification and reversible execution [16]; web and GUI benchmarks stress grounding, session state, and safe action selection [17,18]; workplace and cross-service benchmarks stress coordination across heterogeneous tools and failure modes [19,93]. These settings place their primary bottlenecks on different harness layers, as discussed in Sec. 6. A generic harness improves reuse and lowers engineering cost, but may provide weak inductive bias for the target bottleneck; a specialized harness can improve peak performance, but may reduce transferability and overfit to a benchmark-specific surface.

A practical direction is layered design: a general substrate plus domain-specific adapters. The substrate provides logging, isolation, permission control, persistence, cost accounting, standardized tool access, and auditability. Adapters define observations, actions, verifiers, memory policies, and retry, rollback, stopping, or escalation rules. Protocols such as MCP and A2A reduce connector fragmentation and improve interoperability [65,66], but protocol standardization is not the same as harness generalization. The harness must still decide what to expose, which actions to allow, how to verify outcomes and recover from failure.

Future evaluations should test whether harness improvements transfer across task distributions, model families, and adapter choices, not only whether they raise one benchmark score. Useful evidence includes same-model different-harness comparisons, component ablations, adapter replacement tests, held-out tasks, cross-domain transfer, and runtime profiles. For self-evolving harnesses, this separates benchmark-specific tuning from reusable gains in tracing, memory compression, tool abstraction, verifier selection, or recovery policy [40]. The long-term goal is modular and pressure-aware harness

 doi:10.20944/preprints202606.1312.v1

design: reusable substrates provide stability, observability, and governance, while domain adapters inject task-specific observation, action, verification, and recovery biases.

9. Conclusion

This survey argued that the development of LLM-based agents is best understood as an evolution across four paradigms: prompt engineering, agentic workflows, harness engineering, and agent-native training. The key systems insight is that agent performance is increasingly governed by the interaction between model and runtime rather than by model capability in isolation. The harness perspective helps explain why similar base models can behave so differently once deployed in different environments. It also clarifies why recent progress has depended so heavily on context management, verification, tool design, orchestration, and recovery. At the same time, the rise of RL for agentic behavior suggests that some of this external scaffolding will gradually be internalized into model parameters.

The field is still early. Reliability remains below deployment needs in many realistic settings, evaluation is still only partially aligned with real use, and the boundary between model design and system design is still being renegotiated. But the direction is clear: agent engineering has moved beyond prompt craft and into the study of full systems. Understanding that shift is essential for both building better agents and evaluating their progress responsibly.

References

1. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; et al.. Language Models are Few-Shot Learners. In Proceedings of the Advances in Neural Information Processing Systems, 2020.
2. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* **2022**.
3. Luo, J.; Zhang, W.; Yuan, Y.; Zhao, Y.; Yang, J.; Gu, Y.; Wu, B.; et al. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460* **2025**.
4. Xi, Z.; Chen, W.; Guo, X.; He, W.; Ding, Y.; Hong, B.; Zhang, M.; Wang, J.; et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences* **2025**.
5. Cognition Labs. Introducing Devin, the First AI Software Engineer. https://www.cognition.ai/blog/introducing-devin, 2024.
6. Anthropic. How Claude Code Works. https://docs.claude.com/en/docs/claude-code/how-claude-code-works, 2025.
7. OpenAI. Harness Engineering: Leveraging Codex in an Agent-First World. https://openai.com/index/harness-engineering/, 2026.
8. Shen, M.; Li, Y.; Chen, L.; Fan, Z.; Li, Y.; Yang, Q. From mind to machine: The rise of manus ai as a fully autonomous digital agent. *arXiv preprint arXiv:2505.02024* **2025**.
9. Richards, T.B. AutoGPT. https://github.com/Significant-Gravitas/AutoGPT, 2023.
10. Wang, X.; Li, B.; Song, Y.; Xu, F.F.; Tang, X.; Zhuge, M.; Pan, J.; Song, Y.; Li, B.; Singh, J.; et al. Openhands: An open platform for ai software developers as generalist agents. In Proceedings of the International Conference on Learning Representations, 2025.
11. OpenClaw Team. OpenClaw: Personal AI Assistant. https://github.com/openclaw/openclaw, 2025.
12. Hendrycks, D.; Burns, C.; Basart, S.; Zou, A.; Mazeika, M.; Song, D.; Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300* **2020**.
13. Rein, D.; Hou, B.L.; Stickland, A.C.; Petty, J.; Pang, R.Y.; Dirani, J.; et al. Gpqa: A graduate-level google-proof q&a benchmark. In Proceedings of the First conference on language modeling, 2024.
14. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* **2021**.
15. Phan, L.; Gatti, A.; Han, Z.; Li, N.; Hu, J.; Zhang, H.; et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249* **2025**.
16. Jimenez, C.E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? In Proceedings of the International Conference on Learning Representations, 2024.

 doi:10.20944/preprints202606.1312.v1

17. Zhou, S.; Xu, F.F.; Zhu, H.; Zhou, X.; Lo, R.; Sridhar, A.; Cheng, X.; Ou, T.; Bisk, Y.; Fried, D.; et al. Webarena: A realistic web environment for building autonomous agents. In Proceedings of the International Conference on Learning Representations, 2024.
18. Xie, T.; Zhang, D.; Chen, J.; Li, X.; Zhao, S.; Cao, R.; Hua, T.J.; Cheng, Z.; Shin, D.; Lei, F.; et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems* **2024**.
19. Xu, F.F.; Song, Y.; Li, B.; Tang, Y.; Jain, K.; Bao, M.; et al. Theagentcompany: benchmarking llm agents on consequential real world tasks. *Advances in Neural Information Processing Systems* **2026**.
20. Merrill, M.A.; Shaw, A.G.; Carlini, N.; Li, B.; Raj, H.; Bercovich, I.; Shi, L.; Shin, J.Y.; Walshe, T.; Buchanan, E.K.; et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868* **2026**.
21. Yang, J.; Jimenez, C.E.; Wettig, A.; Lieret, K.; Yao, S.; Narasimhan, K.; Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* **2024**.
22. Hashimoto, M. My AI Adoption Journey. https://mitchellh.com/writing/my-ai-adoption-journey, 2026.
23. Pan, L.; Zou, L.; Guo, S.; Ni, J.; Zheng, H.T. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723* **2026**.
24. Lee, Y.; Nair, R.; Zhang, Q.; Lee, K.; Khattab, O.; Finn, C. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052* **2026**.
25. OpenSquilla Team. OpenSquilla: Token-Efficient AI Agent with Same Budget, Higher Intelligence Density. https://github.com/opensquilla/opensquilla, 2026. Apache-2.0 License.
26. Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q.V.; Zhou, D.; et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* **2022**.
27. Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* **2022**.
28. Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* **2023**.
29. Anthropic. Effective Context Engineering for AI Agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025.
30. Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.t.; Rocktäschel, T.; et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* **2020**.
31. Packer, C.; Fang, V.; Patil, S.; Lin, K.; Wooders, S.; Gonzalez, J. MemGPT: towards LLMs as operating systems. **2023**.
32. Schick, T.; Dwivedi-Yu, J.; Dessì, R.; Raileanu, R.; Lomeli, M.; Hambro, E.; Zettlemoyer, L.; Cancedda, N.; Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems* **2023**.
33. Patil, S.G.; Zhang, T.; Wang, X.; Gonzalez, J.E. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems* **2024**.
34. Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* **2023**.
35. Xu, R.; Yan, Y. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430* **2026**.
36. Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* **2022**.
37. Fourney, A.; Bansal, G.; Mozannar, H.; Tan, C.; Salinas, E.; Niedtner, F.; Proebsting, G.; Bassman, G.; Gerrits, J.; Alber, J.; et al. Magentic-one: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468* **2024**.
38. Zhu, W.; Tang, Z.; Yue, K. SYMPHONY: Synergistic Multi-agent Planning with Heterogeneous Language Model Assembly. *arXiv preprint arXiv:2601.22623* **2026**.
39. OpenAI. OpenAI Agents SDK. https://github.com/openai/openai-agents-python, 2025.
40. Lin, J.; Liu, S.; Pan, C.; Lin, L.; Dou, S.; Huang, X.; Yan, H.; Han, Z.; Gui, T. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850* **2026**.
41. Qi, Z.; Liu, X.; Iong, I.L.; Lai, H.; Sun, X.; Sun, J.; Yang, X.; Yang, Y.; Yao, S.; Xu, W.; et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In Proceedings of the International Conference on Learning Representations, 2025, Vol. 2025.

 doi:10.20944/preprints202606.1312.v1

42. Lai, H.; Liu, X.; Zhao, Y.; Xu, H.; Zhang, H.; Jing, B.; Ren, Y.; Yao, S.; Dong, Y.; Tang, J. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. *arXiv preprint arXiv:2508.14040* **2025**.
43. Guo, D.; Yang, D.; Zhang, H.; Song, J.; Wang, P.; Zhu, Q.; et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* **2025**.
44. Wu, R.; Wang, X.; Mei, J.; Cai, P.; Fu, D.; et al. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079* **2025**.
45. Zhai, Y.; Tao, S.; Chen, C.; Zou, A.; Chen, Z.; Fu, Q.; Mai, S.; Yu, L.; Deng, J.; Cao, Z.; et al. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395* **2025**.
46. Zhang, J.; Hu, S.; Lu, C.; Lange, R.; Clune, J. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954* **2025**.
47. Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science* **2024**.
48. Guo, T.; Chen, X.; Wang, Y.; Chang, R.; Pei, S.; Chawla, N.V.; Wiest, O.; Zhang, X. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680* **2024**.
49. Li, X.; Wang, S.; Zeng, S.; Wu, Y.; Yang, Y. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* **2024**.
50. Tran, K.T.; Dao, D.; Nguyen, M.D.; Pham, Q.V.; O'Sullivan, B.; Nguyen, H.D. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322* **2025**.
51. Yehudai, A.; Eden, L.; Li, A.; Uziel, G.; Zhao, Y.; Bar-Haim, R.; Cohan, A.; Shmueli-Scheuer, M. Survey on evaluation of llm-based agents. *arXiv preprint arXiv:2503.16416* **2025**.
52. Nguyen, D.; Chen, J.; Wang, Y.; Wu, G.; Park, N.; Hu, Z.; Lyu, H.; Wu, J.; Aponte, R.; Xia, Y.; et al. Gui agents: A survey. In Proceedings of the Findings of the Association for Computational Linguistics: ACL 2025, 2025.
53. Ma, Y.; Song, Z.; Zhuang, Y.; Hao, J.; King, I. A Survey on Vision–Language–Action Models for Embodied AI. *IEEE Transactions on Neural Networks and Learning Systems* **2026**.
54. Yu, M.; Meng, F.; Zhou, X.; Wang, S.; Mao, J.; Pan, L.; Chen, T.; Wang, K.; et al. A survey on trustworthy llm agents: Threats and countermeasures. In Proceedings of the Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2, 2025.
55. Meng, Q.; Wang, Y.; Chen, L.; Wang, Q.; Lu, C.; Wu, W.; Gao, Y.; Wu, Y.; Hu, Y. Agent harness for large language model agents: A survey **2026**.
56. Li, J.; Xiao, X.; Zhang, Y.; Liu, C.; Zhao, L.; Liao, X.; Ji, Y.; Wang, J.; Gu, J.; Ge, Y.; et al. Agent Harness Engineering: A Survey. *OpenReview preprint* **2026**.
57. Ning, X.; Tieu, K.; Fu, D.; Wei, T.; Li, Z.; Bei, Y.; Zou, J.; Ai, M.; Liu, Z.; Li, T.W.; et al. Code as Agent Harness. *arXiv preprint arXiv:2605.18747* **2026**.
58. Wooldridge, M.; Jennings, N.R. Intelligent agents: Theory and practice. *The knowledge engineering review* **1995**.
59. Park, J.S.; O'Brien, J.; Cai, C.J.; Morris, M.R.; Liang, P.; Bernstein, M.S. Generative agents: Interactive simulacra of human behavior. In Proceedings of the Proceedings of the 36th annual acm symposium on user interface software and technology, 2023.
60. Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J.; et al. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In Proceedings of the First conference on language modeling, 2024.
61. Hong, S.; Zhuge, M.; Chen, J.; Zheng, X.; Cheng, Y.; Wang, J.; Zhang, C.; Yau, S.; Lin, Z.; Zhou, L.; et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In Proceedings of the International Conference on Learning Representations, 2024.
62. Long, J. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291* **2023**.
63. Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems* **2023**.
64. Anthropic. Effective Harnesses for Long-Running Agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents, 2025.
65. Anthropic. Model Context Protocol. https://modelcontextprotocol.io/introduction, 2025.
66. Cloud, G. Agent2Agent (A2A. https://github.com/a2aproject/A2A, 2025.
67. Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T.B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* **2020**.
68. Hoffmann, J.; Borgeaud, S.; Mensch, A.; Buchatskaya, E.; et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556* **2022**.

 doi:10.20944/preprints202606.1312.v1

69. Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research* **2023**.
70. Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* **2022**.
71. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level code generation with alphacode. *Science* **2022**.
72. Lewkowycz, A.; Andreassen, A.; Dohan, D.; Dyer, E.; Michalewski, H.; Ramasesh, V.; Slone, A.; Anil, C.; Schlag, I.; et al. Solving quantitative reasoning problems with language models. *Advances in neural information processing systems* **2022**.
73. Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Li, Y.; Wu, Y.; et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* **2024**.
74. Alayrac, J.B.; Donahue, J.; Luc, P.; Miech, A.; Barr, I.; Hasson, Y.; Lenc, K.; Mensch, A.; Millican, K.; Reynolds, M.; et al. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems* **2022**.
75. Chen, X.; Djolonga, J.; Padlewski, P.; Mustafa, B.; Changpinyo, S.; Wu, J.; Ruiz, C.R.; Goodman, S.; et al. On scaling up a multilingual vision and language model. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2024.
76. Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* **2024**.
77. Liu, J.; Xia, C.S.; Wang, Y.; Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems* **2023**, *36*, 21558–21572.
78. Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* **2021**.
79. Hendrycks, D.; Burns, C.; Kadavath, S.; Arora, A.; Basart, S.; Tang, E.; Song, D.; Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874* **2021**.
80. Wang, Y.; Ma, X.; Zhang, G.; Ni, Y.; Chandra, A.; et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems* **2024**.
81. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* **2025**.
82. TIGER-Lab. MMLU-Pro Leaderboard. https://huggingface.co/spaces/TIGER-Lab/MMLU-Pro, 2026.
83. Epoch AI. GPQA Diamond. https://epoch.ai/benchmarks/gpqa-diamond?view=graph&tab=leaderboard, 2026.
84. Zhang, Z.; Zhang, H.; Fei, H.; Bao, Z.; Chen, Y.; Lei, Z.; Liu, Z.; Sun, Y.; Xiao, M.; Ye, Z.; et al. SWE-AGI: Benchmarking Specification-Driven Software Construction with MoonBit in the Era of Autonomous Agents. *arXiv preprint arXiv:2602.09447* **2026**.
85. Tang, S.; Chen, R.; Lan, T. Agent Alpha: Tree Search Unifying Generation, Exploration and Evaluation for Computer-Use Agents. *arXiv preprint arXiv:2602.02995* **2026**.
86. Zhou, Q.; Zhang, J.; Wang, H.; Hao, R.; Wang, J.; Han, M.; Yang, Y.; Wu, S.; Pan, F.; Fan, L.; et al. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975* **2026**.
87. Xu, K.; Kordi, Y.; Nayak, T.; Asija, A.; Wang, Y.; Sanders, K.; Byerly, A.; Zhang, J.; Van Durme, B.; Khashabi, D. TurkingBench: A Challenge Benchmark for Web Agents. In Proceedings of the Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies, 2025.
88. Song, Y.; Thai, K.; Pham, C.M.; Chang, Y.; Nadaf, M.; Iyyer, M. Bearcubs: A benchmark for computer-using web agents. *arXiv preprint arXiv:2503.07919* **2025**.
89. Jia, H.; Qian, Y.; Tong, H.; Wu, X.; Chen, L.; Wei, F. Towards adaptive ml benchmarks: Web-agent-driven construction, domain expansion, and metric optimization. *arXiv preprint arXiv:2509.09321* **2025**.
90. Yoa, S.; Yoon, S.; Yoon, S.; Kim, D.; Sim, Y.S.; Lee, J.; Lim, W. From Static Benchmarks to Dynamic Protocol: Agent-Centric Text Anomaly Detection for Evaluating LLM Reasoning. *arXiv preprint arXiv:2602.23729* **2026**.
91. Wang, Y.; Yu, G.; Huang, H.; Wang, Z.; Huang, Y.; Chen, P.; Lyu, M.R. Cloud-OpsBench: A Reproducible Benchmark for Agentic Root Cause Analysis in Cloud Systems. *arXiv preprint arXiv:2603.00468* **2026**.
92. Zhuo, T.Y.; Vu, M.C.; Chim, J.; Hu, H.; et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In Proceedings of the International Conference on Learning Representations, 2025.

 doi:10.20944/preprints202606.1312.v1

93. Long, X.; Du, L.; Xu, Y.; Liu, F.; Wang, H.; Ding, N.; Li, Z.; Guo, J.; Tang, Y. LiveClawBench: Benchmarking LLM Agents on Complex, Real-World Assistant Tasks. *arXiv preprint arXiv:2604.13072* **2026**.
94. Deshpande, K.; Sirdeshmukh, V.; Mols, J.B.; Jin, L.; Hernandez-Cardona, E.Y.; Lee, D.; Kritz, J.; Primack, W.E.; Yue, S.; Xing, C. Multichallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier llms. In Proceedings of the Findings of the Association for Computational Linguistics: ACL 2025, 2025.
95. Kwa, T.; West, B.; Becker, J.; Deng, A.; Garcia, K.; Hasin, M.; Jawhar, S.; Kinniment, M.; Rush, N.; Von Arx, S.; et al. Measuring ai ability to complete long tasks. *arXiv preprint arXiv:2503.14499* **2025**.
96. Kojima, T.; Gu, S.S.; Reid, M.; Matsuo, Y.; Iwasawa, Y. Large language models are zero-shot reasoners. *Advances in neural information processing systems* **2022**.
97. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems* **2023**.
98. Sahoo, P.; Singh, A.K.; Saha, S.; Jain, V.; Mondal, S.; Chadha, A. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* **2024**.
99. Peng, H.; Patil, P.V.; Qiu, A.Z.; Thiruvathukal, G.K.; Davis, J.C. Beyond Local Code Optimization: Multi-Agent Reasoning for Software System Optimization. *arXiv preprint arXiv:2603.14703* **2026**.
100. Liu, Y.K.; Tsai, Y.C. Quality-Driven Agentic Reasoning for LLM-Assisted Software Design: Questions-of-Thoughts (QoT) as a Time-Series Self-QA Chain. *arXiv preprint arXiv:2603.11082* **2026**.
101. Peng, Y.; Zhu, X.; Wei, C.; Zeng, N.; Wang, L.; He, Y.T.; Yu, F.R. Sage: Multi-agent self-evolution for llm reasoning. *arXiv preprint arXiv:2603.15255* **2026**.
102. Hao, G.; Dai, Y.; Qin, X.; Yu, S. Brain-Inspired Graph Multi-Agent Systems for LLM Reasoning. *arXiv preprint arXiv:2603.15371* **2026**.
103. Zhang, L.; Jia, T.; Wang, M.; Hong, W.; Duan, C.; He, M.; Wang, R.; Peng, X.; Wang, M.; Zhang, G.; et al. Efficient failure management for multi-agent systems with reasoning trace representation. *arXiv preprint arXiv:2603.21522* **2026**.
104. Hu, M.; Fang, T.; Zhang, J.; Ma, J.; Zhang, Z.; Zhou, J.; Zhang, H.; Mi, H.; Yu, D.; King, I. Webcot: Enhancing web agent reasoning by reconstructing chain-of-thought in reflection, branching, and rollback. *arXiv preprint arXiv:2505.20013* **2025**.
105. Kumar, A.; Roh, J.; Naseh, A.; Houmansadr, A.; Bagdasarian, E. Throttling Web Agents Using Reasoning Gates. *arXiv preprint arXiv:2509.01619* **2025**.
106. Lee, S.; Yoon, S.; Lee, S.; Chun, Y.; Park, D.; Kim, D.; Sim, J.Y. IntentCUA: Learning Intent-level Representations for Skill Abstraction and Multi-Agent Planning in Computer-Use Agents. *arXiv preprint arXiv:2602.17049* **2026**.
107. Alenezi, M. From Prompt-Response to Goal-Directed Systems: The Evolution of Agentic AI Software Architecture. *arXiv preprint arXiv:2602.10479* **2026**.
108. Agashe, S.; Wong, K.; Tu, V.; Yang, J.; Li, A.; Wang, X.E. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906* **2025**.
109. Chen, Y.; Yan, L.; Yang, Z.; Zhang, E.; Zhao, J.; Wang, S.; Yin, D.; Mao, J. Beyond Monolithic Architectures: A Multi-Agent Search and Knowledge Optimization Framework for Agentic Search. *arXiv preprint arXiv:2601.04703* **2026**.
110. Chen, W.; Peng, Z.; Yin, X.; Ni, C.; Ying, C.; Xie, B.; Luo, Y. SolAgent: A Specialized Multi-Agent Framework for Solidity Code Generation. *arXiv preprint arXiv:2601.23009* **2026**.
111. Huang, J.; Ye, W.; Sun, W.; Zhang, J.; Zhang, M.; Liu, Y. TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code. *arXiv preprint arXiv:2602.06875* **2026**.
112. Chen, M.C.; Kao, Y.H.; Huang, P.H.; Ho, S.C.; Tsou, H.Y.; Wu, I.; Huang, E.M.; Hung, Y.K.; Hsin, W.P.; Liang, C.; et al. SiliconMind-V1: Multi-Agent Distillation and Debug-Reasoning Workflows for Verilog Code Generation. *arXiv preprint arXiv:2603.08719* **2026**.
113. Zhang, Q.; Gao, C.; Han, Y.; Shang, Y.; Fang, C.; Chen, Z.; Xiao, L. SGAgent: Suggestion-Guided LLM-Based Multi-Agent Framework for Repository-Level Software Repair. *ACM Transactions on Software Engineering and Methodology* **2026**.
114. Galster, M.; Mohsenimofidi, S.; Lulla, J.L.; Abubakar, M.A.; Treude, C.; Baltes, S. Configuring Agentic AI Coding Tools: An Exploratory Study. *arXiv preprint arXiv:2602.14690* **2026**.
115. Li, Y.; Zhang, W.; Huang, Z.; Yang, M.; Wu, J.; Guo, S.; Hu, H.; Sun, L.; Yang, J.; Tang, M.; et al. Close the Loop: Synthesizing Infinite Tool-Use Data via Multi-Agent Role-Playing. *arXiv preprint arXiv:2512.23611* **2025**.

 doi:10.20944/preprints202606.1312.v1

116. Zhang, X.; He, Q.; Zheng, Z.; Zhang, Z.; He, X.; Li, D. ASTER: Agentic Scaling with Tool-integrated Extended Reasoning. *arXiv preprint arXiv:2602.01204* **2026**.
117. Mei, L.; Yao, J.; Ge, Y.; Wang, Y.; Bi, B.; Cai, Y.; Liu, J.; Li, M.; Li, Z.Z.; Zhang, D.; et al. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334* **2025**.
118. Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J.; Bi, Y.; Dai, Y.; Sun, J.; Wang, H.; Wang, H.; et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* **2023**.
119. Izacard, G.; Grave, E. Leveraging passage retrieval with generative models for open domain question answering. In Proceedings of the Proceedings of the 16th conference of the european chapter of the association for computational linguistics: main volume, 2021.
120. Borgeaud, S.; Mensch, A.; Hoffmann, J.; Cai, T.; Rutherford, E.; Millican, K.; Van Den Driessche, G.B.; Lespiau, J.B.; et al. Improving language models by retrieving from trillions of tokens. In Proceedings of the International conference on machine learning. PMLR, 2022.
121. Izacard, G.; Lewis, P.; Lomeli, M.; Hosseini, L.; Petroni, F.; Schick, T.; et al. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research* **2023**.
122. Sarthi, P.; Abdullah, S.; Tuli, A.; Khanna, S.; Goldie, A.; Manning, C. Raptor: Recursive abstractive processing for tree-organized retrieval. In Proceedings of the International Conference on Learning Representations, 2024.
123. Edge, D.; Trinh, H.; Cheng, N.; Bradley, J.; Chao, A.; Mody, A.; Truitt, S.; Metropolitansky, D.; Ness, R.O.; Larson, J. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* **2024**.
124. Gutiérrez, B.J.; Shu, Y.; Gu, Y.; Yasunaga, M.; Su, Y. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in neural information processing systems* **2024**.
125. Kang, M.; Chen, W.N.; Han, D.; Inan, H.A.; Wutschitz, L.; Chen, Y.o. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615* **2025**.
126. Yao, Y.; Huang, S.; Dai, E.; Tan, Z.; Duan, Z.; Jia, S.; Jiang, Y.; Yang, T. ARC: Active and Reflection-driven Context Management for Long-Horizon Information Seeking Agents. *arXiv preprint arXiv:2601.12030* **2026**.
127. Wu, Y.; Zheng, Y.; Xu, T.; Zhang, Z.; Yu, Y.; Zhu, J.; Ma, C.; Lin, B.; Dong, B.; Zhu, H.; et al. ContextBudget: Budget-Aware Context Management for Long-Horizon Search Agents. *arXiv preprint arXiv:2604.01664* **2026**.
128. Liu, S.; Yang, J.; Jiang, B.; Li, Y.; Guo, J.; Liu, X.; Dai, B. Context as a tool: Context management for long-horizon swe-agents. *arXiv preprint arXiv:2512.22087* **2025**.
129. Jia, H.; Barr, E.T.; Mechtaev, S. Compressing code context for llm-based issue resolution. *arXiv preprint arXiv:2603.28119* **2026**.
130. Li, H.; Zhu, L.; Zhang, B.; Feng, R.; Wang, J.; Pan, Y.; Barr, E.T.; Sarro, F.; et al. ContextBench: A Benchmark for Context Retrieval in Coding Agents. *arXiv preprint arXiv:2602.05892* **2026**.
131. Zhu, J.; Wu, J.; Hu, M.; Zhu, S.; Pan, J.; Shen, W.; Yang, Y.; Liu, F.; Hao, J.; Jin, Y.; et al. Swe context bench: A benchmark for context learning in coding. *arXiv preprint arXiv:2602.08316* **2026**.
132. Qiu, J.; Liu, Z.; Liu, Z.; Murthy, R.; Zhang, J.; Chen, H.; Wang, S.; Zhu, M.; Yang, L.; Tan, J.; et al. LoCoBench-Agent: An Interactive Benchmark for LLM Agents in Long-Context Software Engineering. *arXiv preprint arXiv:2511.13998* **2025**.
133. Fang, S.; Wang, Y.; Liu, X.; Lu, J.; Tan, C.; Chen, X.; Zheng, Y.; Huang, X.; Qiu, X. Agentlongbench: A controllable long benchmark for long-contexts agents via environment rollouts. *arXiv preprint arXiv:2601.20730* **2026**.
134. Zhang, Q.; Hu, C.; Upasani, S.; Ma, B.; Hong, F.; Kamanuru, V.; Rainton, J.; Wu, C.; Ji, M.; Li, H.; et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618* **2025**.
135. Ye, H.; He, X.; Arak, V.; Dong, H.; Song, G. Meta Context Engineering via Agentic Skill Evolution. *arXiv preprint arXiv:2601.21557* **2026**.
136. Shen, A.; Shen, A. DOVA: Deliberation-First Multi-Agent Orchestration for Autonomous Research Automation. *arXiv preprint arXiv:2603.13327* **2026**.
137. Liu, B.; Zhao, G.; Xu, H. Utility-Guided Agent Orchestration for Efficient LLM Tool Use. *arXiv preprint arXiv:2603.19896* **2026**.
138. Sulc, A. Differentiable Modal Logic for Multi-Agent Diagnosis, Orchestration and Communication. *arXiv preprint arXiv:2602.12083* **2026**.
139. Lin, M.; Zhang, Z.; Lu, H.; Liu, H.; Tang, X.; He, Q.; Zhang, X.; Wang, S. MemMA: Coordinating the Memory Cycle through Multi-Agent Reasoning and In-Situ Self-Evolution. *arXiv preprint arXiv:2603.18718* **2026**.

 doi:10.20944/preprints202606.1312.v1

140. Shen, K.; Zhang, J.; Sun, C.; Zeng, W.; Yue, Y. Structurally Aligned Subtask-Level Memory for Software Engineering Agents. *arXiv preprint arXiv:2602.21611* **2026**.
141. Steiner, A.; Peeters, R.; Bizer, C. MCP vs RAG vs NLWeb vs HTML: A Comparison of the Effectiveness and Efficiency of Different Agent Interfaces to the Web. In Proceedings of the Proceedings of the ACM Web Conference 2026, 2026.
142. Suri, M.; Li, X.; Shojaie, M.; Han, S.; Hsu, C.C.; Garg, S.; Deshmukh, A.A.; Kumar, V. CodeScout: Contextual Problem Statement Enhancement for Software Agents. *arXiv preprint arXiv:2603.05744* **2026**.
143. Prakash, S. LDP: An identity-aware protocol for multi-agent LLM systems. *arXiv preprint arXiv:2603.08852* **2026**.
144. Vishnyakova, V.V. Context Engineering: From Prompts to Corporate Multi-Agent Architecture. *arXiv preprint arXiv:2603.09619* **2026**.
145. Anthropic. Building Effective Agents. https://www.anthropic.com/engineering/building-effective-agents, 2024.
146. OpenAI. A Practical Guide to Building Agents. https://openai.com/business/guides-and-resources/a-practical-guide-to-building-ai-agents, 2025.
147. Yan, Y.; Wang, S.; Du, J.; Yang, Y.; Shan, Y.; Qiu, Q.; Jia, X.; Wang, X.; Yuan, X.; Han, X.; et al. Mcpworld: A unified benchmarking testbed for api, gui, and hybrid computer use agents. *arXiv preprint arXiv:2506.07672* **2025**.
148. Bandi, C.; Hertzberg, B.; Boo, G.; et al. MCP-Atlas: A Large-Scale Benchmark for Tool-Use Competency with Real MCP Servers. *arXiv preprint arXiv:2602.00933* **2026**.
149. Liu, W.; Liu, Z.; Dai, E.; Yu, W.; Yu, L.; Yang, T.; Han, J.; Gao, H. Mcpagentbench: A real-world task benchmark for evaluating llm agent mcp tool use. *arXiv preprint arXiv:2512.24565* **2025**.
150. Jia, H.; Liao, J.; Zhang, X.; Xu, H.; Xie, T.; Jiang, C.; Yan, M.; Liu, S.; Ye, W.; Huang, F. Osworld-mcp: Benchmarking mcp tool invocation in computer-use agents. *arXiv preprint arXiv:2510.24563* **2025**.
151. Vallabhaneni, S.; Berkane, T.; Majumder, M.S. The AI committee: A multi-agent framework for automated validation and remediation of web-sourced data. In Proceedings of the Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 3: System Demonstrations), 2026.
152. Wei, Z.; Yao, W.; Liu, Y.; Zhang, W.; Lu, Q.; Qiu, L.; Yu, C.; Xu, P.; et al. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning. In Proceedings of the Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, 2025.
153. Zhuang, Y.; Jin, D.; Chen, J.; Shi, W.; Wang, H.; Zhang, C. WorkForceAgent-R1: Incentivizing reasoning capability in llm-based web agents via reinforcement learning. In Proceedings of the Findings of the Association for Computational Linguistics: EACL 2026, 2026.
154. Chen, Z.; Zhao, Z.; Han, Z.; Liu, M.; Ye, X.; Li, Y.; Min, H.; Ren, J.; Zhang, X.; Cao, G. TGPO: Tree-Guided Preference Optimization for Robust Web Agent Reinforcement Learning. In Proceedings of the ICASSP 2026-2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2026, pp. 2476–2480.
155. Zhou, W.; Xiong, X.; Tian, Y.; Yue, L.; Wu, X.; Li, W.; Zhao, C.; Dong, H.; Tang, M.; Wang, J.; et al. ESearch-R1: Learning Cost-Aware MLLM Agents for Interactive Embodied Search via Reinforcement Learning. *arXiv preprint arXiv:2512.18571* **2025**.
156. Ding, H.; Liu, P.; Wang, J.; Ji, Z.; Cao, M.; Zhang, R.; Ai, L.; Yang, E.; Shi, T.; Yu, L. DynaWeb: Model-Based Reinforcement Learning of Web Agents. *arXiv preprint arXiv:2601.22149* **2026**.
157. Yu, Q.; Zhang, Z.; Zhu, R.; Yuan, Y.; Zuo, X.; Yue, Y.; Dai, W.; Fan, T.; et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems* **2026**.
158. Zhang, H.; Liu, M.; Zhang, S.; Han, S.; Hu, J.; Jin, Z.; Zhang, Y.; Diao, S.; et al. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents. *arXiv preprint arXiv:2603.18815* **2026**.
159. Lu, S.; Wang, Z.; Zhang, H.; Wu, Q.; Gan, L.; Zhuang, C.; Gu, J.; Lin, T. Don't Just Fine-tune the Agent, Tune the Environment. *arXiv preprint arXiv:2510.10197* **2025**.
160. Zeng, J.; Fu, D.; Mi, T.; Zhuang, Y.; Huang, Y.; Li, X.; Ye, L.; Xie, M.; Hua, Q.; Huang, Z.; et al. davinci-dev: Agent-native mid-training for software engineering. *arXiv preprint arXiv:2601.18418* **2026**.
161. Yang, Z.; Wang, S.; Fu, K.; He, W.; Xiong, W.; Liu, Y.; Miao, Y.; Gao, B.; Wang, Y.; Ma, Y.; et al. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045* **2025**.
162. Karten, S.; Zhang, J.; Upaa Jr, T.; Feng, R.; Li, W.; et al. Continual Harness: Online Adaptation for Self-Improving Foundation Agents. *arXiv preprint arXiv:2605.09998* **2026**.

 doi:10.20944/preprints202606.1312.v1

163. Zhang, Q.; Ma, D.; Fang, T.; Li, J.; Tang, J.; Chen, N.; Mi, H.; Wang, Y. Training LLM Agents for Spontaneous, Reward-Free Self-Evolution via World Knowledge Exploration. *arXiv preprint arXiv:2604.18131* **2026**.
164. Robeyns, M.; Szummer, M.; Aitchison, L. A self-improving coding agent. *arXiv preprint arXiv:2504.15228* **2025**.
165. Zhang, J.; Zhao, B.; Yang, W.; Foerster, J.; Clune, J.; Jiang, M.; Devlin, S.; Shavrina, T. Hyperagents. *arXiv preprint arXiv:2603.19461* **2026**.
166. Enabe, P.A.F. Profile-Then-Reason: Bounded Semantic Complexity for Tool-Augmented Language Agents. *arXiv preprint arXiv:2604.04131* **2026**.
167. Shen, Y.; Li, K.; Zhou, W.; Hu, S. Mem2ActBench: A Benchmark for Evaluating Long-Term Memory Utilization in Task-Oriented Autonomous Agents. *arXiv preprint arXiv:2601.19935* **2026**.
168. Du, P. Memory for autonomous llm agents: Mechanisms, evaluation, and emerging frontiers. *arXiv preprint arXiv:2603.07670* **2026**.
169. Wang, S.; Liu, B.; Gao, Z.; Ma, L.; Wang, X.; Xie, Y.; Tan, X. Explore with Long-term Memory: A Benchmark and Multimodal LLM-based Reinforcement Learning Framework for Embodied Exploration. *arXiv preprint arXiv:2601.10744* **2026**.
170. Zhang, W.; Wei, X.; Huang, W.C.; Hui, Z.; Wang, C.; Gong, M.; Yu, P.S. Memorycd: Benchmarking long-context user memory of llm agents for lifelong cross-domain personalization. *arXiv preprint arXiv:2603.25973* **2026**.
171. Yang, Z.; Tian, S.; Hu, K.; Liu, S.; Nguyen, H.N.; Zhang, Y.; Guo, Z.; Yu, M.; et al. HippoCamp: Benchmarking Contextual Agents on Personal Computers. *arXiv preprint arXiv:2604.01221* **2026**.
172. Maharana, A.; Lee, D.H.; Tulyakov, S.; Bansal, M.; Barbieri, F.; Fang, Y. Evaluating very long-term conversational memory of llm agents. In Proceedings of the Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics, 2024.
173. Huang, D.; Malwe, G.; Wang, Z. When Agents Fail to Act: A Diagnostic Framework for Tool Invocation Reliability in Multi-Agent LLM Systems. *arXiv preprint arXiv:2601.16280* **2026**.
174. Chen, Y.; Dong, G.; Dou, Z. ET-Agent: Incentivizing Effective Tool-Integrated Reasoning Agent via Behavior Calibration. *arXiv preprint arXiv:2601.06860* **2026**.
175. Wang, X.; Zhang, G.; Li, J.; Tu, J.; Li, C.; Li, M. ToolTok: Tool Tokenization for Efficient and Generalizable GUI Agents. *arXiv preprint arXiv:2602.02548* **2026**.
176. Agarwal, A.; Siyan, G.; Pandya, Y.; Singh, J.; Nambi, A.; Awadallah, A. Learning When to Act or Refuse: Guarding Agentic Reasoning Models for Safe Multi-Step Tool Use. *arXiv preprint arXiv:2603.03205* **2026**.
177. Zhu, J.; Tian, Y.; Li, B.; Wu, K.; Liang, Z.; Li, J.; Zhang, X.; Guo, L.; Chen, F.; Liu, Y.; et al. FinMCP-Bench: Benchmarking LLM Agents for Real-World Financial Tool Use under the Model Context Protocol. In Proceedings of the ICASSP 2026-2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2026.
178. Ferrag, M.A.; Lakas, A.; Debbah, M. AgentDrive: An Open Benchmark Dataset for Agentic AI Reasoning with LLM-Generated Scenarios in Autonomous Systems. *arXiv preprint arXiv:2601.16964* **2026**.
179. Fu, M.; Yu, J.; El-Refai, K.; Kou, E.; Xue, H.; Huang, H.; Xiao, W.; Wang, G.; Li, F.F.; Shi, G.; et al. CaP-X: A Framework for Benchmarking and Improving Coding Agents for Robot Manipulation. *arXiv preprint arXiv:2603.22435* **2026**.
180. Ai, J.; Feng, Y.; Zhang, F.; Sun, J.; Li, Z.; et al. ProSoftArena: Benchmarking Hierarchical Capabilities of Multi-modal Agents in Professional Software Environments. In Proceedings of the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2026.
181. Yang, J.; Guo, H.; Ji, L.; Zhou, J.; Zheng, R.; Lei, Z.; Zhang, S.; Xi, Z.; et al. ABC-Bench: Benchmarking Agentic Backend Coding in Real-World Development. *arXiv preprint arXiv:2601.11077* **2026**.
182. Malay, S.K.R.; Nayak, S.; Nair, J.S.; Davasam, S.; Tiwari, A.; Madhusudhan, S.T.; Nemala, S.K.; Sunkara, S.; Rajeswar, S. Enterpriseops-gym: Environments and evaluations for stateful agentic planning and tool use in enterprise settings. *arXiv preprint arXiv:2603.13594* **2026**.
183. Das, A.; Patel, D. PHMForge: A Scenario-Driven Agentic Benchmark for Industrial Asset Lifecycle Maintenance. *arXiv e-prints* **2026**, pp. arXiv–2604.
184. Ding, Y.; Zhang, L. SWE-Replay: Efficient Test-Time Scaling for Software Engineering Agents. *arXiv preprint arXiv:2601.22129* **2026**.
185. Hutter, R.; Pradel, M. AgentStepper: Interactive Debugging of Software Development Agents. *arXiv preprint arXiv:2602.06593* **2026**.

 doi:10.20944/preprints202606.1312.v1

186. Nanda, R.; Maddila, C.; Jha, S.; Khan, E.M.; Paltenghi, M.; Chandra, S. Wink: Recovering from Misbehaviors in Coding Agents. *arXiv preprint arXiv:2602.17037* **2026**.
187. Joshi, A. XAI for Coding Agent Failures: Transforming Raw Execution Traces into Actionable Insights. *arXiv preprint arXiv:2603.05941* **2026**.
188. Guo, C.; Wu, J.; He, S.; Chen, Y.; Kuang, Z.; Fan, S.; Chen, B.; Bao, S.; Liu, J.; Wu, H.; et al. MEnvAgent: Scalable Polyglot Environment Construction for Verifiable Software Engineering. *arXiv preprint arXiv:2601.22859* **2026**.
189. Pohle, C. AgenticTyper: Automated Typing of Legacy Software Projects Using Agentic AI. *arXiv preprint arXiv:2602.21251* **2026**.
190. Kon, P.T.J.; Pradeep, A.; Chen, A.; Ellis, A.P.; Hunt, W.; Wang, Z.; et al. SWE-Protégé: Learning to Selectively Collaborate With an Expert Unlocks Small Language Models as Software Engineering Agents. *arXiv preprint arXiv:2602.22124* **2026**.
191. Shaji, S.; Huppertz, F.; Mitrevski, A.; Houben, S. From Language to Action: Can LLM-Based Agents Be Used for Embodied Robot Cognition? *arXiv preprint arXiv:2603.03148* **2026**.
192. Koh, J.Y.; Lo, R.; Jang, L.; Duvvur, V.; Lim, M.; Huang, P.Y.; Neubig, G.; Zhou, S.; Salakhutdinov, R.; Fried, D. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In Proceedings of the Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024.
193. Song, L.; Dai, Y.; Prabhu, V.; Zhang, J.; Shi, T.; Li, L.; et al. CoAct-1: Computer-using Multi-agent System with Coding Actions. In Proceedings of the The Fourteenth International Conference on Learning Representations.
194. Hu, S.; Ouyang, M.; Gao, D.; Shou, M.Z. The dawn of gui agent: A preliminary case study with claude 3.5 computer use. *arXiv preprint arXiv:2411.10323* **2024**.
195. Xu, Y.; Lu, D.; Shen, Z.; Wang, J.; Wang, Z.; Mao, Y.; Xiong, C.; Yu, T. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. In Proceedings of the International Conference on Learning Representations, 2025.
196. He, Y.; Jin, J.; Liu, P. Efficient agent training for computer use. *arXiv preprint arXiv:2505.13909* **2025**.
197. Bran, A.M.; Cox, S.; Schilter, O.; Baldassari, C.; White, A.D.; Schwaller, P. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376* **2023**.
198. Roohani, Y.; Lee, A.; Huang, Q.; Vora, J.; Steinhart, Z.; Huang, K.; Marson, A.; Liang, P.; Leskovec, J. Biodiscoveryagent: An ai agent for designing genetic perturbation experiments. In Proceedings of the International Conference on Learning Representations, 2025.
199. Liu, F.; Xu, J.; Cui, X.; Wang, X.; Guo, Z.; Wang, J.; Mousavi, S.M.; Gu, X.; Chen, H.; Fei, B.; et al. TRACE: A Multi-Agent System for Autonomous Physical Reasoning for Seismology. *arXiv preprint arXiv:2603.21152* **2026**.
200. Yu, J.; Yu, W.; Xiao, P.; Xing, F. Agent-Driven Corpus Linguistics: A Framework for Autonomous Linguistic Discovery. *arXiv preprint arXiv:2604.07189* **2026**.
201. Ai, K.; Miao, H.; Tang, K.; Gorski, N.; Sun, J.; Liu, G.; Ingolfsson, H.I.; Lenz, D.; Guo, H.; Yu, H.; et al. SciVisAgentBench: A benchmark for evaluating scientific data analysis and visualization agents. *arXiv preprint arXiv:2603.29139* **2026**.
202. Lyu, Y.; Zhang, X.; Yi, X.; Zhao, Y.; Guo, S.; Hu, W.; Piotrowski, J.; Kaliski, J.; Urbani, J.; Meng, Z.; et al. Evoscientist: Towards multi-agent evolving ai scientists for end-to-end scientific discovery. *arXiv preprint arXiv:2603.08127* **2026**.
203. You, Z.; Chen, X.; Vashishtha, A.; Du, S.; Erion-Barner, G.; Mei, H.; Peng, H.; Guo, Y. Improving Clinical Diagnosis with Counterfactual Multi-Agent Reasoning. *arXiv preprint arXiv:2603.27820* **2026**.
204. Liu, C.; Ma, C.; Tao, Y.; Hu, B.; Yang, M. CCD-CBT: Multi-Agent Therapeutic Interaction for CBT Guided by Cognitive Conceptualization Diagram. *arXiv preprint arXiv:2604.06551* **2026**.
205. Du, L.; Li, Y.; Long, Y.; Chen, S. EFT-CoT: A Multi-Agent Chain-of-Thought Framework for Emotion-Focused Therapy. *arXiv preprint arXiv:2601.17842* **2026**.
206. Chen, S.; Moreira, P.; Xiao, Y.; Schmidgall, S.; Warner, J.; Aerts, H.; Hartvigsen, T.; Gallifant, J.; Bitterman, D.S. Medbrowsecomp: Benchmarking medical deep research and computer use. *arXiv preprint arXiv:2505.14963* **2025**.
207. Li, J.; Lai, Y.; Li, W.; Ren, J.; Zhang, M.; Kang, X.; Wang, S.; Li, P.; et al. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957* **2024**.

 doi:10.20944/preprints202606.1312.v1

208. Wang, Y.; Xiang, Y.; Li, K.; Zhang, X.; Ye, B.; Fan, Z.; Wei, F.; Yang, T. Can a Robot Walk the Robotic Dog: Triple-Zero Collaborative Navigation for Heterogeneous Multi-Agent Systems. *arXiv preprint arXiv:2603.21723* **2026**.
209. Wang, L.; Ying, Z.; Yang, X.; Zou, Q.; Yin, Z.; Li, T.; Yang, J.; Yang, Y.; Liu, A.; Liu, X. RoboSafe: Safeguarding Embodied Agents via Executable Safety Logic. *arXiv preprint arXiv:2512.21220* **2025**.
210. Liu, J.; Zhao, P.; Kong, Z.; Shen, X.; Dong, P.; Yang, F.; Cui, L.; Tang, H.; Yuan, G.; Niu, W.; et al. When Should a Robot Think? Resource-Aware Reasoning via Reinforcement Learning for Embodied Robotic Decision-Making. *arXiv preprint arXiv:2603.16673* **2026**.
211. Zhang, Z.; Zhang, J.; Liu, H.; Lv, Q.; Yang, J.; Cai, K.; Wang, K. AgriWorld: A World Tools Protocol Framework for Verifiable Agricultural Reasoning with Code-Executing LLM Agents. *arXiv preprint arXiv:2602.15325* **2026**.
212. Kuntz, T.; Duzan, A.; Zhao, H.; Croce, F.; Kolter, Z.; Flammarion, N.; Andriushchenko, M. Os-harm: A benchmark for measuring safety of computer use agents. *Advances in Neural Information Processing Systems* **2026**, *38*.
213. Hu, R.; Peng, C.; Xu, J.; Gao, C. Repo2run: Automated building executable environment for code repository at scale. *Advances in Neural Information Processing Systems* **2026**.
214. Jain, N.; Singh, J.; Shetty, M.; Zheng, L.; Sen, K.; Stoica, I. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164* **2025**.
215. Kapoor, S.; Stroebl, B.; Kirgis, P.; et al. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation. *arXiv preprint arXiv:2510.11977* **2025**.
216. Liu, Y.; Iter, D.; Xu, Y.; Wang, S.; Xu, R.; Zhu, C. G-eval: NLG evaluation using gpt-4 with better human alignment. In Proceedings of the Proceedings of the 2023 conference on empirical methods in natural language processing, 2023.
217. Zheng, L.; Chiang, W.L.; Sheng, Y.; Zhuang, S.; Wu, Z.; Zhuang, Y.; Lin, Z.; Li, Z.; Li, D.; Xing, E.; et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* **2023**.
218. Gu, J.; Jiang, X.; Shi, Z.; Tan, H.; et al. A survey on llm-as-a-judge. *The Innovation* **2024**.
219. Trivedy, V.; Daugherty, M.; Yurtsev, E.; Chase, H. How We Build Evals for Deep Agents. https://www.langchain.com/blog/how-we-build-evals-for-deep-agents, 2026.
220. Confident AI. DeepEval: The LLM Evaluation Framework. https://github.com/confident-ai/deepeval, 2025.
221. Es, S.; James, J.; Anke, L.E.; Schockaert, S. Ragas: Automated evaluation of retrieval augmented generation. In Proceedings of the Proceedings of the 18th conference of the european chapter of the association for computational linguistics: system demonstrations, 2024.
222. Gao, L.; Tow, J.; Biderman, S.; Black, S.; et al. A framework for few-shot language model evaluation. *Zenodo* **2021**.
223. Liu, X.; Yu, H.; Zhang, H.; Xu, Y.; Lei, X.; Lai, H.; Gu, Y.; Ding, H.; Men, K.; Yang, K.; et al. Agentbench: Evaluating llms as agents. In Proceedings of the International Conference on Learning Representations, 2024.
224. Yin, G.; Bai, H.; Ma, S.; Nan, F.; Sun, Y.; Xu, Z.; Ma, S.; Lu, J.; Kong, X.; Zhang, A.; et al. Mmau: A holistic benchmark of agent capabilities across diverse domains. In Proceedings of the Findings of the Association for Computational Linguistics: NAACL 2025, 2025.
225. Chan, J.S.; Chowdhury, N.; Jaffe, O.; Aung, J.; Sherburn, D.; Mays, E.; Starace, G.; Liu, K.; et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In Proceedings of the International Conference on Learning Representations, 2025.
226. Mialon, G.; Fourrier, C.; Wolf, T.; LeCun, Y.; Scialom, T. Gaia: a benchmark for general ai assistants. In Proceedings of the International Conference on Learning Representations, 2024.
227. Zheng, M.; Han, K.; Li, B.; Xu, H.; Tian, Y.; He, W.; et al. Claw-SWE-Bench: A Benchmark for Evaluating OpenClaw-style Agent Harnesses on Coding Tasks. *arXiv preprint arXiv:2606.12344* **2026**.
228. SWE-bench Team. SWE-bench Leaderboards. https://www.swebench.com/, 2026.
229. Terminal-Bench Team. Terminal-Bench Leaderboard. https://www.tbench.ai/leaderboard/terminal-bench/2.0, 2026.
230. Kapoor, S.; Stroebl, B.; Siegel, Z.S.; Nadgir, N.; Narayanan, A. AI agents that matter. *arXiv preprint arXiv:2407.01502* **2024**.

 doi:10.20944/preprints202606.1312.v1

231. Mehta, S. Beyond Accuracy: A Multi-Dimensional Framework for Evaluating Enterprise Agentic AI Systems. *arXiv preprint arXiv:2511.14136* **2025**.
232. Gupta, A. ReliabilityBench: Evaluating LLM Agent Reliability Under Production-Like Stress Conditions. *arXiv preprint arXiv:2601.06112* **2026**.
233. Cao, H.; Driouich, I.; Thomas, E. Beyond task completion: Revealing corrupt success in LLM agents through procedure-aware evaluation. *arXiv preprint arXiv:2603.03116* **2026**.
234. Anthropic. Claude Sonnet 4.6 System Card. https://www.anthropic.com/claude-sonnet-4-6-system-card, 2026.
235. Google DeepMind. Gemini 3.1 Pro Model Card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, 2026.
236. Xia, C.S.; Deng, Y.; Dunn, S.; Zhang, L. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering* **2025**.
237. Engineering, A. Raising the Bar on SWE-bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/engineering/swe-bench-sonnet, 2025.
238. Yang, J.; Jimenez, C.E.; Press, O.; Narasimhan, K. mini-SWE-agent. https://github.com/SWE-agent/mini-swe-agent, 2025.
239. Zhang, Y.; Ruan, H.; Fan, Z.; Roychoudhury, A. AutoCodeRover: Autonomous Program Improvement. In Proceedings of the Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024. https://doi.org/10.1145/3650212.3680384.
240. Li, H.; Tang, Y.; Wang, S.; Guo, W. Patchpilot: A stable and cost-efficient agentic patching framework. *arXiv e-prints* **2025**, pp. arXiv–2502.
241. Anthropic. Claude 3.7 Sonnet. https://www.anthropic.com/news/claude-3-7-sonnet, 2025.
242. Anthropic. Introducing Claude 4. https://www.anthropic.com/news/claude-4, 2025.
243. OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/, 2025.
244. Google DeepMind. Gemini 3: Our Most Capable Model. https://blog.google/products/gemini/gemini-3, 2025.
245. SWE-bench Team. SWE-bench Experiments Repository. https://github.com/swe-bench/experiments, 2024.
246. Terminal-Bench Team. Terminal-Bench 2.0 Leaderboard Submissions. https://huggingface.co/datasets/harborframework/terminal-bench-2-leaderboard, 2026.
247. Terminal-Bench Team. Terminal-Bench Leaderboard Integrity Update. https://www.tbench.ai/news/leaderboard-integrity-update, 2026.
248. Steel.dev. WebArena Leaderboard 2026: Latest Browser Agent Scores. https://leaderboard.steel.dev/leaderboards/webarena/, 2026.
249. Le Sellier De Chezelles, T.; Gasse, M.; Drouin, A.; Caccia, M.; Boisvert, L.; Thakkar, M.; et al. The BrowserGym Ecosystem for Web Agent Research. *arXiv preprint arXiv:2412.05467* **2025**.
250. Koh, J.Y.; McAleer, S.; Fried, D.; Salakhutdinov, R. Tree Search for Language Model Agents. *arXiv preprint arXiv:2407.01476* **2024**.
251. Dihan, M.L.; Hashem, T.; Ali, M.E.; Parvez, M.R. WebOperator: Action-Aware Tree Search for Autonomous Agents in Web Environment. *arXiv preprint arXiv:2512.12692* **2025**.
252. Sodhi, P.; Branavan, S.R.K.; Artzi, Y.; McDonald, R. SteP: Stacked LLM Policies for Web Actions. *arXiv preprint arXiv:2310.03720* **2023**.
253. Yang, K.; Liu, Y.; Chaudhary, S.; Fakoor, R.; Chaudhari, P.; Karypis, G.; Rangwala, H. AgentOccam: A Simple Yet Strong Baseline for LLM-Based Web Agents. *arXiv preprint arXiv:2410.13825* **2024**.
254. Zhang, Y.; Ma, Z.; Ma, Y.; Han, Z.; Wu, Y.; Tresp, V. WebPilot: A Versatile and Autonomous Multi-Agent System for Web Task Execution with Strategic Exploration. *arXiv preprint arXiv:2408.15978* **2024**.
255. Zhang, R.; Qiu, M.; Tan, Z.; Zhang, M.; Lu, V.; Peng, J.; Xu, K.; Agudelo, L.Z.; Qian, P.; Chen, T. Symbiotic Cooperation for Web Agents: Harnessing Complementary Strengths of Large and Small LLMs. *arXiv preprint arXiv:2502.07942* **2025**.
256. WebTactix. WebTactix: Semantic Tree-Guided Parallel Multi-Agent Planning for Web Task. https://paper-submission-anoymous.github.io/webtactix_introduction/, 2026. Project page.
257. Guo, Y.; Yang, W.; Yang, S.; Liu, Z.; Chen, C.; Wei, Y.; Hu, Y.; Huang, Y.; Hao, G.; Yuan, D.; et al. OpAgent: Operator Agent for Web Navigation. *arXiv preprint arXiv:2602.13559* **2026**.
258. Wang, J.; Zhou, J.; Zhang, W.; Wang, T.; Liu, W.; Zhang, Z.; Lou, X.; Zhang, W.; Deng, H.; Wang, J. ColorBrowserAgent: Complex Long-Horizon Browser Agent with Adaptive Knowledge Evolution. *arXiv preprint arXiv:2601.07262* **2026**.

 doi:10.20944/preprints202606.1312.v1

259. Thakkar, M.; Chapados, N.; Pal, C.; et al. WebArena Verified: Reliable Evaluation for Web Agents. In Proceedings of the Workshop on Scaling Environments for Agents.
260. Liu, J.; Qian, C.; Su, Z.; Zong, Q.; Huang, S.; He, B.; Fung, Y.R. CostBench: Evaluating Multi-Turn Cost-Optimal Planning and Adaptation in Dynamic Environments for LLM Tool-Use Agents. *arXiv preprint arXiv:2511.02734* **2025**.
261. Li, H.; Wang, Z.; Dai, Q.; Nie, Y.; Peng, J.; Liu, R.; Zhang, J.; Zhu, K.; He, J.; Wang, L.; et al. OpenSage: Self-programming Agent Generation Engine. *arXiv preprint arXiv:2602.16891* **2026**.
262. KRAFTON AI.; Ludo Robotics. Terminus-KIRA: Boosting Frontier Model Performance on Terminal-Bench with Minimal Harness. https://github.com/krafton-ai/kira, 2026.