Article

Not peer-reviewed version

Terminal Agents: A Survey of AI Agents in Command-Line Environments

Xiaoyang Yuan †, Haoxi Zeng †, Wencheng Ye †, Yi Bin *, Wenqi Shao, Chen Qian, Wei Ye, Yujuan Ding, Zheng Wang, Pengpeng Zeng, Jingkuan Song, Heng Tao Shen

Posted Date: 18 June 2026

doi: 10.20944/preprints202606.1409.v1

Keywords: terminal agents; LLM-based agents; terminal command execution; command-observation trajectories; outer-loop design; executable environments; terminal competence acquisition; agent evaluation

 doi:10.20944/preprints202606.1409.v1

Article

Terminal Agents: A Survey of AI Agents in Command-Line Environments

Xiaoyang Yuan [1,2,†], Haoxi Zeng [1,†], Wencheng Ye [1,†], Yi Bin [1,*], Wenqi Shao [2], Chen Qian [3], Wei Ye [1], Yujuan Ding [4], Zheng Wang [1], Pengpeng Zeng [1], Jingkuan Song [1] and Heng Tao Shen [1]

[1] Tongji University, Shanghai, China
[2] Shanghai Innovation Institute, Shanghai, China
[3] Shanghai Jiao Tong University, Shanghai, China
[4] The Hong Kong Polytechnic University, Hong Kong, China
* Correspondence: yi.bin@hotmail.com
† These authors contributed equally to this research.

Abstract

LLM-based agents increasingly interact with external environments through terminal command execution, yet existing surveys have rarely treated the terminal itself as a primary analytical object. This survey examines terminal agents, namely systems whose task progress depends on iterative command execution, textual feedback, and stateful terminal command interaction, and clarifies their boundaries with adjacent categories such as software-engineering agents, GUI- or browser-based computer-use agents, and CLI-packaged assistants. Through a substrate-centered lens, we systematize the literature around architectures and outer-loop design patterns, competence acquisition through executable environments, command–observation trajectories, and post-training, and evaluation protocols for terminal-mediated capabilities. Across systems, acquisition pipelines, and benchmarks, the synthesis shows that outer-loop design is not an implementation detail but a first-class variable that materially shapes measured performance. The evidence further indicates that terminal competence is multi-dimensional, spanning how agents formulate actions, interpret feedback, manage runtimes, track state and context, verify progress, recover from failures, and control side effects, rather than reducible to a single capability ranking. Current evidence remains concentrated in software engineering, while cross-domain transfer, model-versus-scaffold attribution, reliable recovery in mutable environments, and process-level evaluation remain underdeveloped. The survey provides an evidence-calibrated map of established findings, emerging practices, and unresolved challenges for terminal agents.

Keywords: terminal agents; LLM-based agents; terminal command execution; command-observation trajectories; outer-loop design; executable environments; terminal competence acquisition; agent evaluation

---

GitHub: https://github.com/EnigmaYYYY/awesome-terminal-agents

1. Introduction

Large language models (LLMs) are moving from prompt-bound text generation toward interactive systems that can act on external environments and revise their behavior from feedback [1–3]. When such systems are embedded in iterative action-observation loops, they become agentic systems. These agents execute code, invoke computational tools, navigate graphical interfaces, and operate diverse digital environments over multiple steps [4,5]. This development spans general agent autonomy, software engineering, computer use, and scientific computing. As agents increasingly operate through different execution media, the properties of these media shape what actions are possible, what feedback is available, how state is exposed, and how progress can be verified. This motivates a substrate-level view of agent behavior, in which the execution medium is treated as part of the agent system rather than as a neutral implementation channel.

 doi:10.20944/preprints202606.1409.v1

The terminal is one such execution medium, but it is especially consequential for agentic work. It is a compact, scriptable, and stateful medium that exposes filesystems, package managers, processes, logs, compilers, test runners, remote machines, and command-line tools through a shared textual interface. Commands directly modify the environment: they create files, edit repositories, install dependencies, launch services, execute tests, and alter runtime state. Their results appear as stdout and stderr streams, exit codes, diffs, stack traces, logs, and generated artifacts. For agents operating through this medium, reasoning, execution, observation, and verification are coupled through a mutable command-observation loop [6]. We regard such systems whose dominant progress-bearing action-observation loop depends on this form of terminal-based command execution, textual feedback, and stateful environment interaction as *terminal agents*.

Existing surveys overlap with terminal agents from several adjacent directions. Surveys of general LLM agents discuss tool use, planning, and autonomous interaction [7,8]; surveys of software-engineering agents cover repository repair, code editing, and test-driven workflows [9,10]; surveys of GUI and browser-based computer use examine visual or interface-mediated interaction [5,11,12]; and evaluation-oriented surveys analyze benchmark design and measurement practice [13,14]. These works provide important context, and some of the systems they discuss involve terminal execution. However, terminal agents themselves have not yet been treated as the central object of survey-level analysis. Terminal-mediated behavior is therefore often examined as part of broader agent autonomy, software engineering, computer use, or evaluation, rather than as a substrate-centered form of agency with its own boundaries, capability requirements, system designs, acquisition pipelines, and measurement issues.

The evidence base for terminal agents is also uneven. Software engineering currently provides the densest empirical foundation because repositories, tests, and build systems make terminal-mediated behavior observable and reproducible [15–17]. Recent benchmarks, environment-construction frameworks, post-training studies, and deployment reports further show that terminal-mediated interaction is increasingly being treated as an object of training, evaluation, and deployment rather than as a passive execution backend [17–20]. At the same time, terminal agents are not limited to repository repair or coding assistance. Similar command-observation loops appear in operations, data engineering, scientific workflows, cybersecurity, cloud management, and other terminal-centered domains [21–26]. This uneven landscape calls for an evidence-calibrated synthesis that distinguishes established findings from emerging practices and from claims that remain underdetermined by current evidence. The review follows a structured scoping protocol, with the evidence-stratified corpus and screening procedure documented in Appendix A.

In this survey, we use the terminal-mediated action-observation loop in Figure 1 as the organizing unit for synthesizing the literature. Rather than grouping prior work only by application domain or surface interface, we examine how terminal execution shapes four linked aspects of agent behavior: the operational boundary between terminal-grounded agency and adjacent systems, the architectural evolution and design mechanisms that make terminal interaction an explicit design surface, the acquisition pipelines that turn executable trajectories into competence, and the evaluation protocols that measure or under-measure these capabilities. This perspective allows us to relate the relatively mature evidence from software engineering to emerging work in operations, data engineering, scientific workflows, cybersecurity, and cloud management, while keeping clear which claims are empirically established and which remain early or underdetermined.

 doi:10.20944/preprints202606.1409.v1

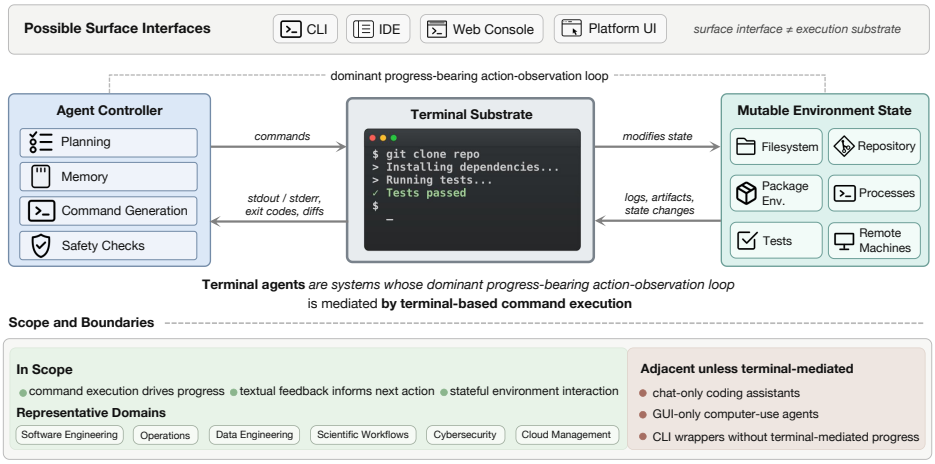

Figure 1. Terminal agents and related basic concepts. The figure separates surface interfaces from the execution substrate and shows how an agent controller, terminal substrate, and mutable environment state form a progress-bearing action-observation loop. Systems are in scope when command execution drives task progress, textual feedback informs subsequent actions, and stateful environment interaction is central to the workload; systems with only surface-level CLI access or incidental command use are treated as adjacent.

Across this substrate-centered perspective, this survey makes three contributions. First, it provides a substrate-centered definition of terminal agents and a boundary framework that separates progress-bearing terminal execution from surface modality. Second, it synthesizes terminal competence as a seven-dimensional capability profile, covering how agents formulate actions, interpret feedback, manage runtimes, track state and context, verify progress, recover from failures, and control side effects. Third, it provides an evidence-calibrated map of terminal-agent architectures, competence-acquisition pipelines, evaluation practices, and unresolved challenges, distinguishing findings with repeated empirical support from emerging or deployment-facing claims.

Figure 2 summarizes the organization of the survey. Section 2 establishes the scope of terminal agents, distinguishes them from adjacent agent categories, and introduces the competence dimensions used throughout the paper. Section 3 examines terminal-agent system design and architectures, focusing on architectural evolution, interface design, runtime organization, control mechanisms, governance policies, and harness-level context management. Section 4 examines how terminal agents acquire terminal competence through executable environments, command-observation trajectories, filtering, relabeling, replay, and post-training. Section 5 analyzes benchmark families, evaluation layers, protocol validity, and the coverage of current metrics. Section 6 consolidates evidence across architectures, acquisition pipelines, and benchmarks. Section 7 discusses open challenges and future directions, and Section 8 concludes.

 doi:10.20944/preprints202606.1409.v1

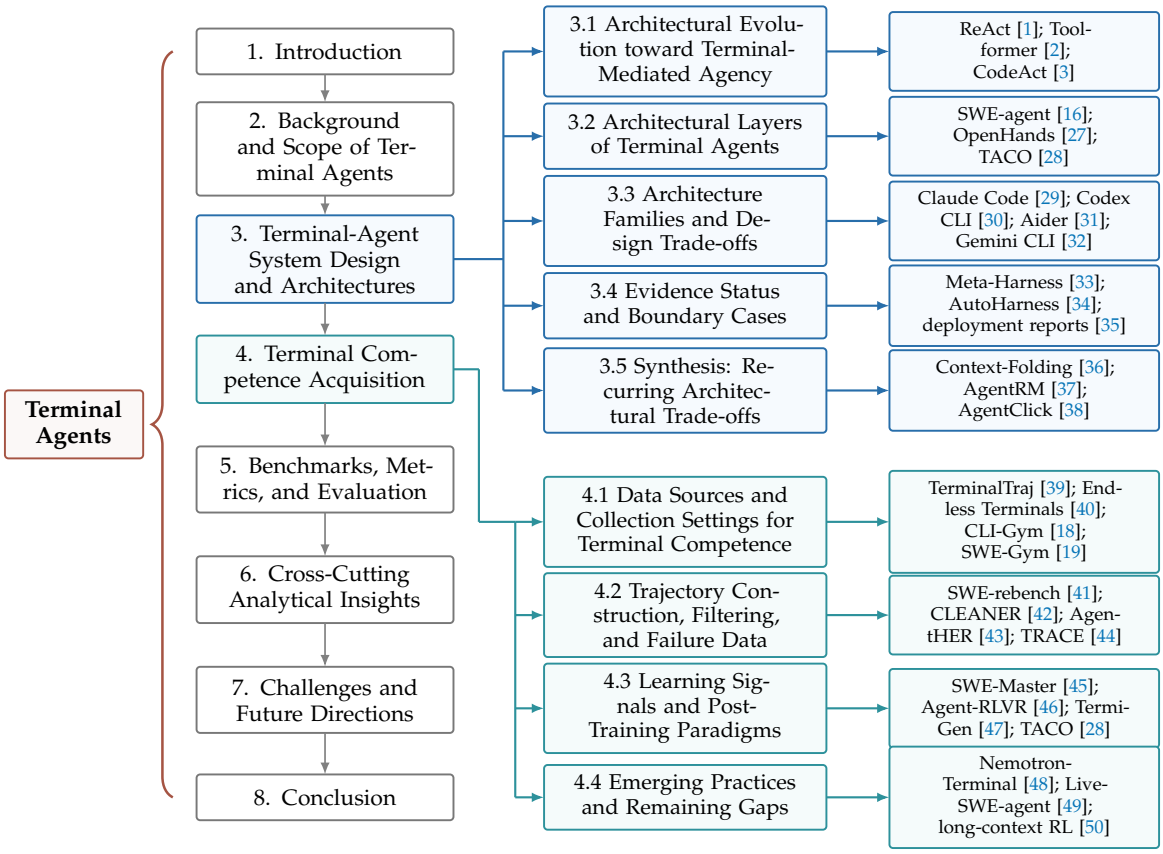

Figure 2. Organization of the survey. The left brace groups the full article under the topic of terminal agents. The main column shows the eight-section structure, while the expanded branches detail the two central analytical sections: terminal-agent system design and architectures, and terminal competence acquisition. The rightmost column lists representative citation anchors for the corresponding subsections.

2. Background and Scope of Terminal Agents

This section establishes the analytical scope used throughout the survey. It clarifies what kinds of systems are treated as terminal agents, why terminal-based command execution matters as an execution substrate, how terminal agents differ from adjacent agent paradigms, and what capabilities terminal-mediated interaction requires.

2.1. Terminal as a Command-Execution Agent Substrate

The terminal is a distinctive agent substrate because it combines compactness, compositionality, statefulness, and operational reach [6]. A relatively small vocabulary of commands and shell primitives through the terminal can inspect directories, edit files, execute programs, install dependencies, launch processes, query logs, and connect to remote systems through a unified textual interface [20]. Unlike GUI-centered environments, where state must often be inferred from visual layouts or interface events, the terminal exposes much of its interaction state in text. Observations arrive as stdout and stderr streams, return codes, diffs, stack traces, logs, and process outputs. This textual observability narrows the gap between a language model's token-level interface and the environment it must control [16,18].

Four properties make terminal-based command execution especially consequential for agent design. First, commands executed through a shell or equivalent terminal runtime are state-mutating: they can create, delete, edit, install, configure, and launch artifacts that persist beyond a single model turn [6,20]. Second, terminal observations are textually inspectable: errors, logs, test outputs, and runtime traces can be incorporated directly into subsequent reasoning [16,18,51]. Third, command-line interaction couples execution with verification: commands can be rerun, outputs compared, tests executed, and failures localized to concrete messages or artifacts [3,16,52]. Fourth, terminal actions are often code-like textual artifacts: command strings, scripts, patches, and configuration edits align

 doi:10.20944/preprints202606.1409.v1

with the code-centric capabilities and training distributions that make contemporary LLMs especially effective in software-engineering workflows [3,53–55]. Commands can also be chained, redirected, parameterized, and embedded in scripts, which allows local actions to be composed into longer workflows [6].

The terminal is therefore not merely an input-output channel for tool calls. It shapes the action-observation loop itself by turning state-changing commands and textual runtime feedback into a coupled process of reasoning, execution, observation, and verification. Figure 3 summarizes this terminal-mediated loop and connects it to the boundary tests, scope outcomes, and competence dimensions used throughout the survey.

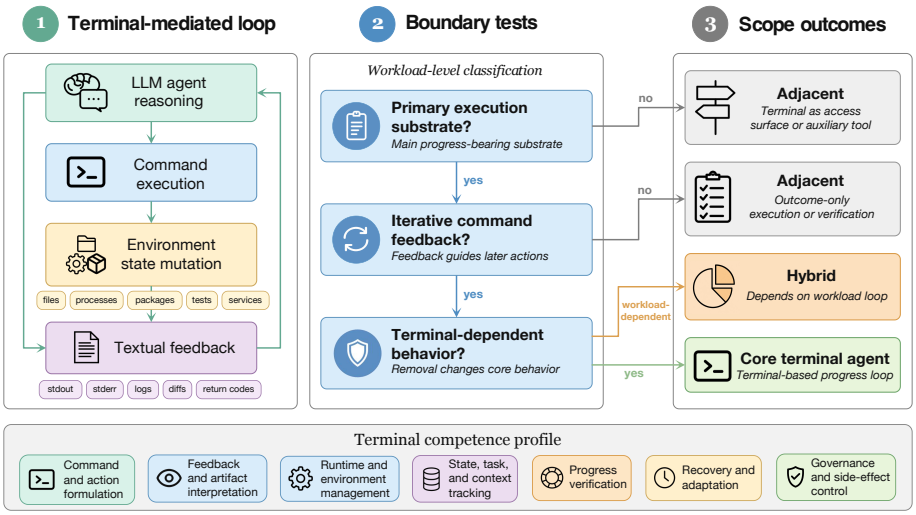

Figure 3. Operational scope of terminal agents. The figure connects the terminal-mediated action–observation loop, the workload-level boundary tests, the resulting scope categories, and the competence dimensions used throughout the survey.

2.2. Substrate-Based Scope and Boundaries

In this survey, we regard terminal agents as systems whose dominant progress-bearing action-observation loop is mediated by terminal-based command execution. This characterization is substrate-based rather than surface-based. A system may be exposed through a CLI, IDE, web interface, or platform runtime, but it falls within our scope only when task progress depends on command execution, textual feedback, and stateful environment interaction. Relevant actions include command execution, file and process manipulation, environment inspection, dependency installation, test execution, and invocation of terminal-accessible tools. Relevant observations include command outputs, logs, errors, diffs, process state, test results, and other environment feedback that informs later actions. The left panel of Figure 3 illustrates this progress-bearing terminal loop.

This scope excludes two common but insufficient criteria for identifying terminal agents. First, a system is not a terminal agent merely because it can invoke a terminal command as an auxiliary tool. Generic tool-using agents may invoke terminal commands occasionally while remaining primarily grounded in other environments or APIs [2,4]. Second, a one-shot code or patch generator is not a terminal agent if it does not rely on iterative execution and feedback. What distinguishes terminal agents is an execution-grounded loop in which commands alter the environment and subsequent observations materially shape future decisions [1,3,16].

To make this scope operational for classification and later synthesis, we use three rules, which correspond to the workload-level boundary tests in the middle panel of Figure 3:

1. **Primary execution substrate**: Terminal-based command execution must be the primary execution substrate for the workload under discussion, not a marginal auxiliary call.

 doi:10.20944/preprints202606.1409.v1

2. **Iterative command feedback**: Command outputs, errors, logs, diffs, return codes, or environment state changes must materially shape later actions.
3. **System character**: The system must change character if terminal access is removed. If the core behavior remains essentially the same without terminal interaction, the system is treated as adjacent rather than core.

"Dominant" does not mean that every action must be a direct terminal command. It means that task progress causally depends on terminal-mediated state changes and feedback. The unit of classification is also workload-sensitive: a hybrid platform may be terminal-mediated for repository repair or operations tasks, but not for workflows where the main progress loop occurs through a browser, GUI, or remote API. These rules do not eliminate all boundary judgment, but they make inclusion and exclusion decisions more auditable by localizing disputes to specific criteria rather than holistic impressions. The right panel of Figure 3 shows the corresponding scope outcomes.

Table 1 illustrates how the three rules separate core terminal agents from common adjacent and hybrid cases. The table is not intended to exhaust all possible systems. Rather, it shows how the rules should be applied when a system contains, exposes, or invokes a terminal but terminal interaction may or may not be the progress-bearing substrate. The corpus-level screening procedure is reported in Appendix A.

Table 1. Decision framework for separating terminal agents from adjacent and hybrid systems.

Case	Rules	In scope?	Reason
SWE-agent-style agents [16]	1, 2, 3	Yes	Command feedback drives progress; removing terminal access changes system character.
OpenHands terminal-centric mode [27]	1, 2; 3 conditional	Conditional	Depends on whether terminal execution is the dominant locus of progress for the workload.
Static patch generators / Agentless [56]	Fails Rules 1, 2, and 3	No	No iterative terminal-mediated loop; command execution is delivery or verification infrastructure rather than the reasoning substrate.
Browser/desktop computer-use agents [57]	Fails Rule 1	No	GUI, DOM, or visual feedback is the primary substrate; terminal interaction is weak, absent, or incidental.
CLI-packaged API assistants [58]	Fails Rule 1	No	Terminal is the access surface, not the grounding substrate; the tool primarily routes prompts, command suggestions, or explanations to model APIs.

The examples in Table 1 clarify why terminal-agent status should not be inferred from surface modality alone. A system exposed through a terminal is not necessarily a terminal agent, and a system exposed through an IDE or platform runtime may still qualify if terminal execution underneath drives task progress. Conversely, a system that invokes commands only for patch application, testing, or final verification may remain outside our core scope if those commands do not form the iterative reasoning substrate.

Hybrid platforms require particular care. The relevant question is whether terminal execution drives task progress for the workload under discussion, not whether the system *contains* a terminal. An Agentless-style pipeline [56] that performs static analysis and delegates patch application or testing to shell commands is a structured reasoning pipeline with command output, not a terminal agent. Conversely, an IDE-surfaced system [35] can qualify when terminal execution underneath drives task progress.

The boundary tests also clarify three common conflations. First, coding agents are an important subpopulation but do not exhaust terminal agents. Repository tasks provide reproducible evidence, whereas operations [59,60], diagnostics [61], data tasks [62], scientific workflows [24], OS-level automation [63], and cloud operations [26] require terminal-mediated behavior beyond repository repair [15,16,64]. Second, computer-use agents may include a terminal application, but their primary feedback substrate is visual or DOM-based, not stdout, stderr, logs, diffs, and exit codes [57,65,66].

 doi:10.20944/preprints202606.1409.v1

Third, CLI-packaged assistants are defined by access modality rather than grounding. A system surfaced in a terminal is not a terminal agent unless terminal execution drives task progress, while an IDE-surfaced system can qualify if terminal execution underneath is progress-bearing [20,35].

2.3. Terminal Competence as a Capability Profile

The preceding scope and boundary tests establish what kinds of systems count as terminal agents in this survey. A related question is what capabilities terminal-mediated interaction demands and develops. We synthesize terminal competence as a functionally complete capability profile at the level of major roles in terminal-mediated agency. The profile does not enumerate every low-level command, tool, or domain-specific skill. Instead, it identifies the recurring capability requirements imposed by a progress-bearing terminal action–observation loop: formulating executable terminal-mediated actions, interpreting feedback and artifacts, managing mutable runtime environments, tracking state and task context, verifying progress, recovering from failures, and controlling side effects. These dimensions are used as an analytical reference frame rather than as a claim that the underlying skills are mutually independent.

Across system, acquisition, and benchmark papers, prior work repeatedly reports behaviors and failure modes involving command selection, textual feedback interpretation, environment setup and repair, state persistence, progress verification, recovery, long-horizon control, and execution governance [16,17,27,52,67,68]. We use these recurring patterns to define dimensions that can be observed in terminal interaction traces, system designs, training pipelines, or benchmark protocols. The bottom strip of Figure 3 provides a compact visual summary of this competence profile.

We use seven dimensions to characterize terminal competence:

1. **Command and action formulation**: translating user goals, task constraints, and the current task state into executable terminal-mediated actions, including shell commands, command sequences, scripts, CLI calls, file edits, build/test/run actions, and structured action primitives or tool-mediated terminal operations.
2. **Feedback and artifact interpretation**: extracting task-relevant evidence from terminal-exposed feedback and artifacts, including stdout, stderr, exit codes, logs, diffs, test outputs, stack traces, process signals, generated files, workspace changes, and other runtime artifacts.
3. **Runtime and environment management**: preparing, configuring, maintaining, and repairing executable environments, including dependency installation, version resolution, service configuration, background processes, containers or virtual machines, remote machines, environment variables, resource limits, and runtime constraints.
4. **State, task, and context tracking**: maintaining awareness of external environment state, task-specific context, and accumulated interaction history across extended terminal sessions, including filesystem changes, repository state, installed packages, running processes, configuration state, prior commands, partial goals, verified facts, unverified assumptions, and unresolved subproblems.
5. **Progress verification**: designing and executing checks that determine whether intermediate progress is valid, whether outputs or artifacts are trustworthy, and whether task completion conditions have been satisfied.
6. **Recovery and adaptation**: diagnosing failures, revising hypotheses, changing strategies, replanning execution paths, rolling back or mitigating harmful intermediate actions, and retrying based on grounded execution evidence.
7. **Governance and side-effect control**: acting within permission boundaries, sandbox constraints, approval checkpoints, safety policies, credential-handling requirements, resource constraints, and limits on external side effects such as file deletion, network access, privilege escalation, or modification of external systems.

These dimensions are not independent in execution. Recovery and adaptation depend on feedback interpretation, state tracking, and progress verification; runtime and environment management

 doi:10.20944/preprints202606.1409.v1

requires appropriate action formulation; and long-horizon persistence is expressed through the agent's ability to track state, task context, verified facts, and unresolved subproblems across extended terminal sessions. The value of the profile is therefore analytical rather than ontological: it provides a stable vocabulary for comparing what terminal-agent systems require, what architectures expose or scaffold, what acquisition pipelines train or omit, and what benchmarks measure or under-measure. Sections 3–5 use this capability profile to analyze how terminal competence is distributed across models, interfaces, runtimes, recovery mechanisms, governance policies, and evaluation protocols.

3. Terminal-Agent System Design and Architectures

Section 2 established terminal agents through a terminal-mediated action-observation loop and introduced seven competence dimensions required by this loop. This section examines how such agents are realized as systems. We first trace the architectural evolution through which terminal-mediated interaction became an explicit design target. We then analyze the main architectural layers, recurring architecture and runtime-infrastructure families, evidence status, and trade-offs that shape terminal-agent system design.

3.1. Architectural Evolution toward Terminal-Mediated Agency

Terminal-agent system design did not emerge fully formed. We summarize the literature through four overlapping architectural shifts rather than a strict chronology: from tool-augmented prompting, to executable code actions with structured interfaces, to terminal-mediated agency as a first-class design target, and then to runtime- and harness-centered architectures. These shifts identify how terminal-agent systems gradually externalize action interfaces, mutable workspaces, recovery policies, governance constraints, and harness-level context management as explicit design surfaces.

Figure 4 summarizes this progression as an architectural evolution toward terminal-mediated agency. The figure emphasizes how increasingly explicit design surfaces make terminal interaction less a passive execution backend and more a system-level object of design.

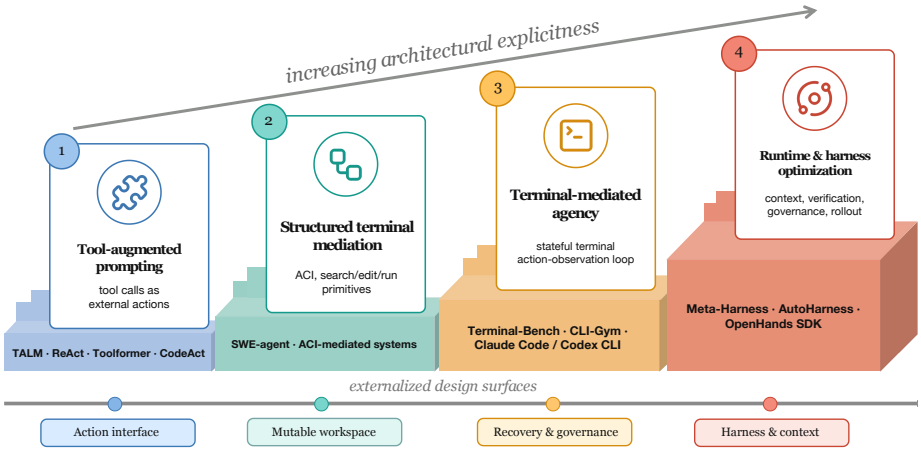

Figure 4. Architectural evolution toward terminal-mediated agency. The stages summarize overlapping shifts from tool-augmented prompting to runtime- and harness-centered terminal-agent systems. The bottom axis highlights how action interfaces, mutable workspaces, recovery and governance mechanisms, and harness-level context become explicit design surfaces.

Shift 1: Tool-augmented prompting. Early work by TALM [4], ReAct [1], Toolformer [2], and CodeAct [3] established that language models could call external tools and fold observations back into subsequent decisions. Execution was treated as one tool among many; the action space was relatively flat, and the environment was not yet treated as a persistent mutable terminal workspace. These systems demonstrated the feasibility of model-mediated execution but did not directly address the challenges of operating in a mutable, stateful terminal environment.

 doi:10.20944/preprints202606.1409.v1

Shift 2: Executable code actions with structured interfaces. SWE-agent [16] marks a pivotal transition: it redesigned terminal interaction through an agent-computer interface (ACI) that recasts repository navigation, editing, and execution as model-legible operations. The insight is that raw command access offers expressive freedom, but much of that freedom is difficult to exploit when the search space is too wide. It treats terminal access not as a passive backend but as a systems design problem, shifting from raw command *access* to designed terminal *mediation*. OpenHands [27] broadened this into a platform-runtime perspective where the terminal coexists with code execution, browser interaction, and tool orchestration inside a persistent system.

Shift 3: Terminal-mediated agency as a first-class design target. The next shift moves the terminal from infrastructure to design object. Terminal-native benchmarks such as Terminal-Bench [17] and CLI-Gym [18], together with training environments such as Endless Terminals [40] and TermiGen [47], treat terminal-mediated interaction as a primary distribution for training and evaluation rather than as a side effect of repository repair. Commercial terminal coding agents provide deployment-practice evidence that direct command execution with permission gating can be packaged as a usable product surface, although these reports should not be treated as controlled empirical evidence [20,35]. Deployment-facing terminal harnesses make this shift concrete. Systems such as Claude Code [29], Codex CLI [30], Aider [31], and Gemini CLI [32] package terminal-mediated agency as interactive developer-facing runtimes: the agent can inspect and modify local workspaces, run commands or tests, and continue from textual feedback while operating under permission, approval, and sandbox policies. These systems are important because they instantiate terminal agency with relatively direct command execution in deployed workflows, but their evidence status remains engineering-practice rather than controlled empirical evidence. At this point, the question is no longer only whether models can use tools, but how to design systems whose primary progress loop is terminal-mediated.

Shift 4: Runtime- and harness-centered architectures. A more recent line of work treats the outer-loop, including context packaging, workspace persistence, verification policy, and control flow, as a first-class optimization target. Meta-Harness [33] and AutoHarness [34] show that outer-loop design can be optimized or synthesized, turning context compaction, approval rules, and observation shaping into levers that alter measured capability. Training-oriented frameworks such as the OpenHands SDK [27] and extensible RL platforms [69] treat the agent runtime as programmable infrastructure; collaborative multi-agent frameworks extend this view to multi-entity orchestration [70]. In this line of work, architecture becomes an object of study rather than merely the vessel through which a model is deployed.

Across these shifts, a common design logic emerges: functions that were previously implicit in the model, the environment, or the evaluation harness become explicit architectural components. The action interface determines how the model acts; the workspace determines what state persists; recovery and governance policies determine how errors and side effects are bounded; and the harness determines what context is retained across turns. This evolution motivates the layered architectural view developed next.

3.2. Architectural Layers of Terminal Agents

The architectural evolution above suggests that terminal-agent systems, regardless of family, must address four layers: interface and observation; runtime and workspace; control, verification, recovery, and governance; harness and context. These layers are not a pipeline with fixed boundaries; they identify where design decisions enter the terminal-mediated action-observation loop. Each layer maps to distinct competence dimensions from Section 2, although concrete systems often couple adjacent layers and support multiple dimensions at once.

Figure 5 summarizes the four analytical layers around the terminal-mediated action-observation loop. The diagram is not meant to impose a fixed execution order. In concrete systems, harness and context management often wrap the entire loop rather than appearing as a terminal stage. Family tags indicate layer emphasis under the taxonomy in Table 2; repeated tags reflect cross-layer coupling rather than duplicate categories.

 doi:10.20944/preprints202606.1409.v1

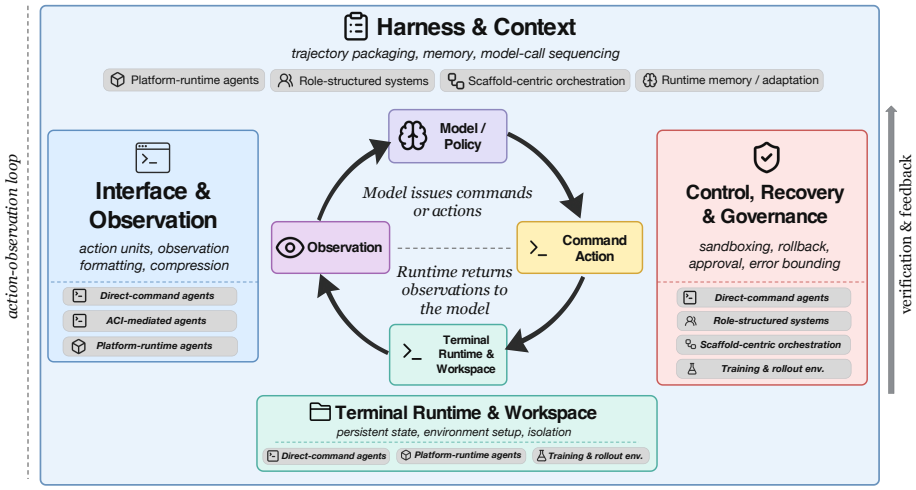

Figure 5. Layered architecture of terminal-agent systems. The central loop shows action generation, command execution, workspace mutation, and observation return. The surrounding layers show interface shaping, runtime and workspace management, verification, recovery, governance, and harness-level context management.

Layer 1: Interface and observation. This layer determines what action units are available to the model and how environment feedback is represented. Design choices span direct terminal commands, ACI-mediated primitives such as structured search, editing, and execution, runtime event layers, and workflow-stage interfaces. SWE-agent [16] is a representative ACI-mediated design: it replaces unconstrained repository interaction with model-legible search, edit, and execution primitives, thereby making terminal actions easier for the model to select and recover from. The observation side is equally consequential: directory listings, stack traces, logs, exit codes, diffs, generated files, and workspace changes are useful only when exposed at the right granularity. TACO [28] reports that principled context compression can simultaneously reduce token overhead and improve accuracy. The terminal's textual observations are comparatively easy to filter, compress, and reincorporate into inference, although this advantage depends on observation scale and formatting [6]. Robustness of agentic function calling under varying interface conditions remains an open concern [71]. This layer most directly shapes **command and action formulation** and **feedback and artifact interpretation** (dimensions 1 and 2).

Layer 2: Runtime and workspace. Commands mutate environment state; the runtime layer determines what state persists, how it is isolated, and how later actions build on earlier ones. Systems range from stateless execution to repository-coupled workspaces to persistent multi-agent environments. OpenHands [27] provides a representative platform-runtime example: terminal execution, code editing, browser interaction, and tool orchestration operate over a shared persistent workspace rather than as isolated tool calls. Terminal-native environments such as Terminal-Bench [17] and CLI-Gym [18] make the same runtime issue explicit for evaluation by packaging tasks inside reproducible command-line workspaces. Workspace-coupled runtimes enable cumulative progress but introduce drift: incorrect intermediate actions silently shape later observations. The runtime layer also governs dependency installation, service configuration, version resolution, background processes, containers, virtual machines, resource limits, and workspace persistence, making it the primary architectural determinant of **runtime and environment management** and an important support for **state, task, and context tracking** (dimensions 3 and 4). Agentic approaches to constructing these environments, such as automated Docker image building [72], point toward treating the workspace layer itself as an optimizable component.

Layer 3: Control, verification, recovery, and governance. Terminal agents can install packages, modify filesystems, access credentials, and trigger external side effects. This layer addresses how progress is checked, how errors are bounded, how recovery is structured, and how oversight enters the loop. Design mechanisms include test-feedback loops, sandboxing, rollback, human approval

 doi:10.20944/preprints202606.1409.v1

checkpoints, and workflow decomposition. STRATUS illustrates role-delegated recovery through specialized planning, execution, and review agents [73]; analogous role-delegation principles extend to automated incident response [59,74] and configuration drift detection [75]. AgentClick structures human oversight around skill-based checkpoints instead of per-command text inspection [38]. Broader architectural work on OS-level agent resource management [37,76], context-space access control [77], and reliable agent state management [78] reinforces the view that the terminal is a medium through which permission, accountability, refusal, and side-effect constraints are operationalized. This layer maps most directly to **progress verification**, **recovery and adaptation**, and **governance and side-effect control** (dimensions 5, 6, and 7).

Layer 4: Harness and context. The harness packages trajectory history, formats prompts, manages context windows, and sequences model calls. It determines what the model remembers across turns and how different models or strategies are invoked at different steps. Context management mechanisms inspired by version-control primitives offer structured alternatives to raw truncation [79]. Meta-Harness [33] and AutoHarness [34] elevate the harness from fixed infrastructure to an optimizable component, showing that outer-loop design choices shift measured performance. Multi-agent and asynchronous execution strategies further expand this design space [80,81]. The harness is a major architectural determinant of **state, task, and context tracking** (dimension 4), because it controls which trajectory information, verified facts, unresolved subproblems, and prior actions remain available across turns. It also supports long-horizon persistence as a cross-cutting outcome rather than as a separate competence dimension.

These four layers structure the comparison of architecture and runtime-infrastructure families that follows. Each family emphasizes different layers and couples them differently. These differences are not merely implementation details; they reflect distinct hypotheses about where terminal competence resides and how much competence should be supplied by the model, the interface, the runtime, the governance policy, or the harness. This layered view also clarifies that long-horizon persistence should be analyzed as an emergent property of state, task and context tracking, runtime persistence, verification, and recovery, rather than as a replacement for any one of the seven competence dimensions defined in Section 2.

3.3. Architecture and Runtime-Infrastructure Families

We organize the surveyed systems into seven architecture and runtime-infrastructure families, each defined by a distinctive mechanism through which terminal-mediated progress is achieved or supported. Table 2 summarizes these families by defining mechanism, representative examples, main trade-off, and the competence dimensions each most directly addresses. The families are design patterns rather than mutually exclusive partitions; a system may instantiate more than one pattern when, for example, a deployed terminal tool also introduces structured editing or command primitives. Because these families differ substantially in evidence maturity, Table 2 should be read as a design-space taxonomy rather than as a ranking of empirical support. Section 3.4 discusses evidence status and boundary cases after the taxonomy.

The family labels are based on architectural mechanisms rather than on competence dimensions; the competence column only indicates which dimensions are most directly shaped by each mechanism. A work-centered reading of the taxonomy shows that these families differ less by surface interface than by how they distribute responsibility for terminal competence. SWE-agent [16] emphasizes interface design by replacing unconstrained shell interaction with model-legible search, edit, and execution primitives. OpenHands [27] shifts attention to the runtime layer by coordinating terminal execution with code, browser, and tool surfaces inside a persistent workspace. Deployment-facing agents such as Claude Code [29], Codex CLI [30], Aider [31], and Gemini CLI [32] preserve more direct command-level interaction, but place greater weight on permission gating, sandboxing, user-visible feedback, and workspace-level safety controls. Meta-Harness [33] and AutoHarness [34], in contrast, make the outer loop itself the object of optimization, showing that measured terminal-agent behavior depends not only on the base model but also on prompt packaging, context management, model-call sequencing,

 doi:10.20944/preprints202606.1409.v1

and control policy. Training and rollout environments such as CLI-Gym [18], Endless Terminals [40], TermiGen [47], R2E-Gym [82], and SWE-Factory [83] further shift the focus from deployed interaction to the construction of terminal-native trajectories used for learning and evaluation.

Table 2. Terminal-agent architecture and runtime-infrastructure families, representative mechanisms, design trade-offs, and primary competence dimensions. Dimension numbers 1–7 follow the seven-dimension profile in Section 2: 1=command and action formulation, 2=feedback and artifact interpretation, 3=runtime and environment management, 4=state, task, and context tracking, 5=progress verification, 6=recovery and adaptation, and 7=governance and side-effect control. *Training and rollout environments are not deployed agents; they are included because they define terminal-native runtime infrastructures for rollout and learning.

Family	Defining mechanism	Examples	Main trade-off	Comp.
Deployment-facing direct-command agents	Direct terminal access with permission, approval, or sandbox gating	Claude Code [29]; Codex CLI [30]; Aider [31]; Gemini CLI [32]; commercial terminal coding agents [20,35]	Expressive reach vs. noisy trajectories, safety burden, and harder rollback	1,2,5,7
ACI-mediated agents	Structured search, edit, and execution primitives	SWE-agent [16]; Aider [31]	Reliability vs. generality across task types	1,2,3
Platform-runtime agents	Persistent runtime coordinating terminal and auxiliary surfaces	OpenHands [27]	Breadth vs. harness dependence	3,4,6
Role-structured systems	Role-delegated planning, execution, and review	STRATUS [73]; HyperAgent [84]	Structured recovery vs. coordination overhead	5,6,7
Scaffold-centric orchestration	Harness-mediated command execution, context, and permissions	Meta-Harness [33]; AutoHarness [34]	Optimizable performance vs. attribution difficulty	2,4,6
Runtime memory / adaptation	Online context adaptation for long-horizon sessions	Context-Folding [36]; TACO [28]	Persistence vs. compression loss	4,6
Training & rollout env.*	Terminal-native worlds for rollout, RL, and trajectory generation	CLI-Gym [18]; Endless Terminals [40]; TermiGen [47]; R2E-Gym [82]; SWE-Factory [83]	Scalable learning vs. transfer uncertainty	1,2,3,5

Each family foregrounds a distinct terminal-specific problem. **Deployment-facing direct-command agents** foreground the expressiveness problem: any terminal-accessible program can become part of the working environment, but the action space is wide and noisy, placing heavy demands on command formulation, observation interpretation, and recovery. **ACI-mediated agents** address the reliability problem: by restricting the action space to structured primitives, they reduce planning burden at the cost of generality. **Platform-runtime agents** foreground the multi-surface coordination problem: the terminal is central but must interoperate with code execution sandboxes, browsers, and external tools. **Role-structured systems** address recovery at scale: decomposing planning, execution, and review into specialized roles makes failures more diagnosable and repairable within a structured workflow. **Scaffold-centric orchestration** makes the attribution problem explicit: by treating the harness as an optimization target, it separates what the model contributes from what the outer-loop contributes, although it does not fully resolve attribution. **Runtime memory architectures** address long-horizon coherence: context compression and retrieval determine whether the agent maintains coherence across extended sessions. **Training and rollout environments** address the data-generation problem: terminal-native rollouts produce the terminal-mediated trajectories that other families consume.

3.4. Evidence Status and Boundary Cases

The evidence base for these architecture and runtime-infrastructure families is unevenly distributed. SWE-agent [16] and OpenHands [27] have broader community use and more repeated evaluation than most newer families; Meta-Harness [33] and AutoHarness [34] rest on controlled but single-study evidence; commercial direct-command terminal agents are documented primarily through engineering reports rather than controlled experiments [20,35]. Deployment-facing terminal

 doi:10.20944/preprints202606.1409.v1

harnesses such as Claude Code [29], Codex CLI [30], Aider [31], and Gemini CLI [32] are useful for identifying real-world interface, permission, sandbox, and workflow patterns, but they should not be treated as controlled evidence of architectural superiority without reproducible benchmark comparisons. Training environments show promising scalability results, but transfer to live terminal workflows remains largely untested. Domain-specific architectural evidence is emerging in network operations [85,86], but it has not yet been integrated into the broader architectural frameworks discussed here. This unevenness is not merely a bibliographic detail. It affects how strongly architectural claims can be generalized.

Considering this section focuses on architecture and runtime-infrastructure families, Agentless [56], CGM [87], OSWorld [57], and CLI-packaged assistants [58] are retained only as boundary comparators. They clarify what is lost when terminal execution is static, graph-based, GUI/DOM-grounded, or merely a product surface.

3.5. Synthesis: Recurring Architectural Trade-offs

The taxonomy is useful only to the extent that it exposes recurring architectural trade-offs. Across the four layers and seven families, several trade-offs recur in how terminal-agent systems allocate responsibility among the model, action interface, runtime, governance policy, and harness. These trade-offs are not strict oppositions: many systems try to improve both sides through better scaffolding, observation design, sandboxing, or context management. They are nevertheless useful analytical axes because different architecture families prioritize and combine these design goals differently.

Balancing expressiveness and recoverability. Raw or weakly mediated terminal access maximizes the space of possible actions but produces noisier trajectories and harder rollback. ACI-mediated and scaffold-centric architectures improve reliability by constraining the action space, but constraints that help on one task family may block necessary actions on another. The trade-off maps primarily to dimensions 1 (command and action formulation) and 6 (recovery and adaptation), with progress verification providing the checks that make recovery actionable.

Balancing generality and task discipline. Platform-runtime agents support heterogeneous work across terminal runtimes, code execution, and browsers, but their generality can dilute the structured feedback that drives reliable progress. Role-structured and workflow-constrained systems benefit from stronger local structure at the cost of narrower applicability. This trade-off surfaces in dimensions 3 (runtime and environment management) and 4 (state, task, and context tracking): general platforms struggle to maintain coherent state across diverse tasks, while disciplined systems may fail on tasks outside their designed workflow.

Balancing automation and inspectability. Terminal agents can automate substantial work, yet their most deployable forms depend on keeping commands, outputs, and control interventions legible to users. Permission gating, approval checkpoints, and sandboxing make automation auditable but introduce latency and interrupt autonomy. The trade-off is concentrated in dimension 7 (governance and side-effect control), which is among the least systematically addressed competence dimensions across all architecture families.

Disentangling model capability and harness contribution. When harness design is treated as an optimization target, measured gains become difficult to attribute: a benchmark improvement may reflect better context management, more forgiving observation shaping, or smarter retry logic, not stronger model reasoning. Agentless underscores this trade-off by showing that, on some repository-repair tasks, static reasoning pipelines can rival interactive agents [56]. The trade-off cuts across all seven competence dimensions and makes the case that terminal-agent architecture should be evaluated at the level of the *system loop*, not only at the level of the model.

These trade-offs connect the architectural analysis back to the competence profile in Section 2. Each trade-off reflects a different way of prioritizing, exposing, or scaffolding the seven competence dimensions through system design. Taken together, the architectural evidence suggests that terminal competence is distributed across the model, the action interface, the runtime workspace, the recovery and governance policy, and the harness that packages context across turns. Architecture therefore

 doi:10.20944/preprints202606.1409.v1

does not merely implement a terminal agent after the fact; it determines which capabilities the model must supply, which capabilities are scaffolded by the system, and which failures become observable to training or evaluation. Sections 4 and 5 next examine how acquisition pipelines and evaluation protocols respectively train, expose, or obscure these capabilities.

4. Terminal Competence Acquisition

For terminal agents, competence acquisition is not only a matter of scaling executable tasks, but also depends on coverage over command-observation-verification-recovery patterns, namely the recurring ways in which terminal-executed commands change state, produce textual feedback, expose failures, and require subsequent repair or continuation decisions [39,40,44,48]. This section focuses on work that constructs terminal-relevant training data, supervision signals, or optimization procedures. Harnesses and runtime scaffolds appear only when they shape what traces can be collected, replayed, filtered, or learned from.

Terminal competence is not reducible to static code generation. Recent work frames it as a problem of *environment-grounded supervision*: the agent learns from trajectories collected through live interaction [19,39,40,48], where each action, including terminal commands, edits, or tool invocations, produces observations such as stdout, stderr, diffs, exit codes, and test outputs that shape subsequent decisions. Unlike ordinary instruction-tuning data, the central unit is not an isolated prompt-response pair but a stateful command-observation trace with executable feedback and, when available, failure and recovery signals. Throughout this section, we interpret acquisition mechanisms in relation to the seven competence dimensions introduced in Section 2: command and action formulation; feedback and artifact interpretation; runtime and environment management; state, task, and context tracking; progress verification; recovery and adaptation; governance and side-effect control. The central question is therefore not only how to collect more terminal trajectories, but how to preserve the parts of interaction history that teach executable action selection, feedback interpretation, runtime setup, state and context tracking, verification, failure diagnosis, recovery, and side-effect-aware execution.

Figure 6 gives the chapter-level view of terminal competence acquisition. It connects acquisition sources, command-observation-recovery traces, runtime and governance mechanisms, learning paradigms, and the resulting competence profile.

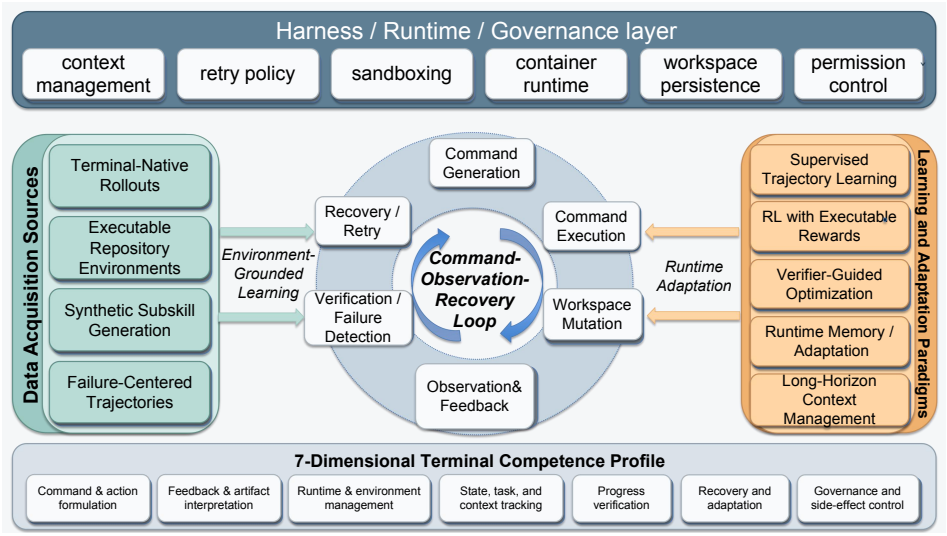

Figure 6. Terminal competence acquisition ecosystem for terminal agents. Data sources, command-observation-recovery traces, runtime and governance mechanisms, and learning paradigms jointly shape the multi-dimensional terminal competence profile.

4.1. Data Sources and Collection Settings for Terminal Competence

The first question is what kinds of environments can expose the behaviors we want the agent to learn. In POMDP terms, the environment defines which states are reachable, which observations

 doi:10.20944/preprints202606.1409.v1

are produced, and which action sequences lead to reward. Early repository-level benchmarks such as SWE-bench [15] were crucial precursors because they made software tasks executable and therefore observable beyond final text output. However, such benchmarks were designed primarily for evaluation, not for terminal-centric data construction. More recent work shifts the emphasis from tasks that merely *allow* execution to settings explicitly designed to *produce* terminal-mediated interaction data [18,39,40,47]. The key distinction is not whether an environment is executable, but whether it exposes recoverable command-observation trajectories that can support command selection, state tracking, diagnosis, and failure recovery.

Table 3 summarizes four data-source categories. We discuss each category below, focusing on what behavior it exposes and what limitation it introduces.

Table 3. Data source categories for terminal competence acquisition.

Data source type	Representative works	Exposed capability	Main limitation
Terminal-native rollout collection	Dockerized traj. gen. [39]; Endless Terminals [40]; CLI-Gym [18]	Command selection, command-output interpretation, and error recovery in Dockerized environments	Realism and transfer beyond generated terminal tasks remain uncertain
Executable repository environments	SWE-Gym [19]; SWE-Dev [88]; SWE-rebench [41]; SWE-Next [89]	Repository navigation, multi-file editing, test-grounded verification, and environment setup	Strongly SWE-biased; terminal behavior may be incidental to repository repair
Synthetic subskill generation	TermiGen [47]; SWE-smith [90]; Hybrid-Gym [91]	Targeted subskills such as localization, dependency search, repair, and environment construction	May overfit synthetic task patterns; transfer to live terminal workflows remains unclear
Failure-centered trajectories / diagnostics	TRACE [44]; AgentHER [43]; AgentForesight [92]	Diagnosis, rollback reasoning, early failure prediction, and recovery from dead ends	Under-collected; failure labels and recovery taxonomies remain inconsistent

Terminal-native rollout collection treats the terminal interaction as the primary training distribution. Large-scale Dockerized trajectory generation [39], Endless Terminals [40], CLI-Gym [18], and R2E-Gym [82] each treat terminal rollouts as a data-engineering problem rather than treating the terminal as a passive backend. Whether the resulting behaviors transfer beyond the generated terminal tasks remains uncertain.

Executable repository environments convert real repositories into runnable training environments with executable verification. SWE-Gym [19], SWE-Dev [88], SWE-rebench [41], SWE-Next [89], SWE-Factory [83], and SWE-Hub [93] provide strong executable realism, but terminal behavior in these settings is often entangled with repository-specific reasoning rather than learned as a separable skill. MLE-STAR extends executable training to ML engineering workflows [94], while community-driven approaches explore decentralized training data collection [95].

Synthetic generation broadens coverage of sparse subskills through controlled task construction. TermiGen [47], SWE-smith [90], Hybrid-Gym [91], and MLE-Dojo [96] each produce tasks with verifiable success conditions, but a central question remains: whether agents trained on such data learn reusable terminal habits or synthetic-environment heuristics.

Failure-centered trajectory data is the fourth and least-developed category. Misdiagnoses, failed installations, rollback attempts, and recovery decisions are behaviors that clean success-only traces cannot expose. TRACE [44], AgentHER [43], and AgentForesight [92] represent initial collection efforts, but failure labels and recovery taxonomies remain inconsistent.

4.2. Trajectory Construction, Filtering, and Failure Data

Raw interaction is rarely usable as supervision without further processing. Terminal trajectories are noisy, path-dependent, partially successful, and often expensive to collect. As a result, a major contribution of recent work lies not only in building environments, but in defining how rollouts become reusable corpora [39,42,43,48]. This is one of the clearest respects in which terminal-agent acquisition

 doi:10.20944/preprints202606.1409.v1

differs from ordinary instruction tuning. The literature pursues this through two complementary strategies: *pipeline engineering*, which treats the full collection-to-training chain as a design object, and *skill-coverage expansion*, which broadens the range of behaviors that traces encode [39,44,47,48].

Dockerized terminal-trajectory generation work [39] provides the clearest end-to-end example: its pipeline combines task adaptation, synthetic generation, rollout collection, filtering, and decontamination to produce a terminal-oriented corpus. The acquisition lesson is that trajectory *composition* matters: which traces are retained and how they are staged in a curriculum can change what the model learns. CLEANER [42] shows self-purified trajectories yield stronger RL than raw rollouts; AgentHER [43] validates trajectories through hindsight experience replay; and progressive code masking [97] evaluates research agents by varying how much prior code is visible. Nemotron-Terminal [48] illustrates the potential impact of terminal-oriented data engineering on Terminal-Bench performance; Live-SWE-agent [49] explores whether agents can self-evolve through online interaction; and long-context multi-turn RL [50] shows promise for training software engineering agents with extended trajectories.

Explicit treatment of failed interaction remains under-developed. The most important missing data are recoverable failures: trajectories where the agent misdiagnoses, repairs, or adapts after environment drift [43,44,92,98]. Current pipelines excel at collecting successful traces but remain weak at preserving dead ends, long repair loops, and messy environment drift. Yet failed package installation attempts, version-conflict diagnoses, broken environment assumptions, and rollback decisions often expose recovery behavior more directly than the final successful command [44,98,99]. A corpus built only from clean trajectories may produce agents that look competent on curated tasks but remain brittle under real recovery challenges [42,43,99]. We therefore argue that failure data should be treated as a primary acquisition target for terminal agents rather than as a supplement to successful trajectories [43,44,98]. Without observing misdiagnosis, rollback, and repair, models have little direct supervision for the behaviors that distinguish robust terminal-mediated interaction from clean command imitation.

4.3. Learning Signals and Post-Training Paradigms

Sections 4.1 and 4.2 discussed where terminal-relevant trajectories come from and how raw interaction traces are filtered, repaired, or retained. The remaining question is how these trajectories become learning signals. For terminal agents, the signal is not only whether a final answer is correct, but also whether command choices, observations, state changes, failures, and recovery attempts provide usable supervision for later behavior. Table 4 organizes six paradigms by their dominant supervision signal, main capability target, and major blind spots or transfer risks.

Table 4. Learning and post-training paradigms for terminal competence acquisition. The table summarizes dominant supervision signals, main capability targets, and major blind spots or transfer risks.

Paradigm	Learning signal	Main capability target	Main blind spot	Key risk
SFT on successful traces	Successful cmd-obs trajectories	Common commands, repository navigation, standard workflows	Recovery, rollback, diagnosis after wrong assumptions	Clean-trace overfit; no failure paths
RL with env. rewards	Executable success, verifier reward, test pass/fail	Outcome-oriented exploration, task completion	Process quality, safe intermediate behavior	Sparse reward hacking; benchmark overfit
Process-aware / verifier-guided	Step-level judgments, verifier ranking	Diagnosis, action selection, recovery decision quality	Open-domain transfer	Verifier bias; shifted failures
Synthetic terminal task gen.	Generated tasks with controlled verification	Rare commands, setup patterns, targeted subskills	Live realism, messy environment drift	Synthetic heuristics; weak transfer
Failure-conditioned training	Failed or repaired traces, hindsight relabeling	Recovery, diagnosis, early failure detection	Stable generalization under inconsistent failure taxonomies	Noisy labels; repair-loop overfit
Runtime memory / context adapt.	Session history, summarized traces, retrieved past experience	Long-horizon persistence, recurring workflow reuse	Robust correction under wrong memory	Compression loss; memory contamination

SFT on successful traces. The most direct paradigm is supervised fine-tuning on successful command-observation trajectories. Nemotron-Terminal [48] illustrates this direction by emphasizing terminal-oriented data engineering and showing that curated terminal interaction data can affect

 doi:10.20944/preprints202606.1409.v1

Terminal-Bench performance. Repository-centered environments such as SWE-Gym [19] and SWE-Dev [88] provide another source of successful trajectories, where navigation, editing, testing, and environment setup can be learned from executable software tasks. This paradigm is useful for teaching common commands, repository workflows, and standard setup patterns, but it under-represents diagnosis, rollback, and recovery because unsuccessful attempts are often filtered away [100,101].

RL with executable rewards. A second paradigm uses executable feedback as reward. Endless Terminals [40] treats terminal environments as interactive worlds for reinforcement learning, while Agent-RLVR [46] and SWE-Master [45] use executable verification, test outcomes, or reward models to optimize agent behavior over software tasks. SWE-Gym [19] also connects executable environments to training rather than only evaluation. The strength of this paradigm is that it aligns learning with task completion under real execution. Its limitation is that reward signals may be sparse or benchmark-specific, so agents can improve final success while still learning brittle command habits or unsafe intermediate behavior.

Process-aware and verifier-guided optimization. A third paradigm adds supervision over intermediate decisions rather than only final success. Verifier-guided methods [43,102] use step-level judgments, rankings, or hindsight validation to distinguish useful actions from misleading ones. AgentHER [43], for example, uses hindsight experience replay to convert trajectories into more informative learning signals for recovery and decision quality. These methods are especially relevant for terminal agents because many failures become visible before the final outcome, through stderr, logs, failing tests, or inconsistent environment state. The open problem is whether verifiers trained or calibrated on one task distribution transfer to open-ended terminal workflows.

Synthetic task generation for subskill coverage. Synthetic generation broadens coverage when real traces are sparse. TermiGen [47] targets terminal-use tasks, while SWE-smith [90] and Hybrid-Gym [91] construct verifiable tasks that expose specific subskills such as dependency search, repair, localization, and environment construction. This paradigm is valuable because many terminal behaviors, including rare commands and setup patterns, appear too infrequently in natural trajectories. Its risk is that agents may learn synthetic-environment regularities rather than reusable terminal competence, especially when generated tasks lack messy environment drift or realistic failure paths.

Failure-conditioned training. Failure-centered approaches treat unsuccessful or partially repaired trajectories as supervision rather than noise. TRACE [44], AgentHER [43], and AgentForesight [92] focus on diagnosis, early failure prediction, hindsight relabeling, or recovery from dead ends. This paradigm directly targets the recovery dimension of terminal competence: the agent must interpret failed commands, revise assumptions, and decide whether to retry, rollback, or change strategy. However, the field still lacks stable taxonomies for failure types and recovery outcomes, making failure-conditioned learning vulnerable to noisy labels and repair-loop overfitting.

Runtime memory and context adaptation. A final paradigm treats acquisition as an online adaptation problem. Systems such as Memento [103], Context-Folding [36], and TACO [28] use session history, summarized traces, or compressed context to preserve information across long interactions. These methods are important because terminal tasks often require remembering prior commands, changed files, installed packages, failed attempts, and partial hypotheses. Their main risk is that compressed or retrieved memory can preserve wrong assumptions, causing the agent to repeat earlier mistakes under distribution drift.

Across these paradigms, the central pattern is that terminal-agent post-training increasingly depends on interaction-history construction rather than isolated instruction-response construction. Successful traces teach common workflows; executable rewards align behavior with task completion; verifiers and hindsight relabeling expose process quality; synthetic tasks expand subskill coverage; failure-conditioned data teaches recovery; and runtime memory supports long-horizon persistence. The unresolved question is how to combine these signals without overfitting to particular benchmarks, generated environments, or harness-specific reward channels.

 doi:10.20944/preprints202606.1409.v1

Because these paradigms depend on live execution and interaction-history construction, terminal-agent training papers should report the number of executed rollouts, average trajectory length, container or runtime cost, retry budget, verifier calls, total model-inference tokens, failure-retention policy, decontamination procedure, and train-test environment overlap. Without these quantities, acquisition results are difficult to compare because live terminal execution makes data generation cost, data quality, and environment overlap central experimental variables.

4.4. Emerging Practices and Remaining Gaps

Several patterns cut across the acquisition literature. First, acquisition is becoming pipeline-centric rather than dataset-centric: sourcing, rollout generation, filtering, replay, and curriculum are treated as separate design levers rather than monolithic data dumps [19,39,48]. Second, terminal competence depends on preserving more than successful final trajectories. Setup attempts, command outputs, state changes, failed commands, verifier feedback, and recovery decisions all carry supervision signals, yet current corpora still represent successful repository-centric traces more consistently than failed setup attempts, long repair loops, environment drift, or unsafe intermediate actions [42–44]. Third, transfer remains the main unresolved validity question. It is not yet clear whether agents trained on repository-repair trajectories acquire general terminal competence or merely task-specific heuristics [56]. The core question is which command-observation-reward sequences encode transferable terminal skill rather than benchmark-specific behavior [104]. Complementary work on externalization in LLM agents [105] and semantics-aware program repair [106] suggests that skill acquisition is mediated by how explicitly reasoning is encoded in trajectories.

The acquisition literature therefore points to a shift from dataset construction to interaction-history construction. Terminal competence is learned through commands that mutate state, observations that provide evidence, failures that expose diagnostic demands, and recovery attempts that reveal whether an agent can revise its behavior. Current pipelines increasingly treat environment construction, rollout collection, filtering, relabeling, and post-training as coupled design problems, but the evidence remains uneven: successful repository-centric traces are better represented than cross-domain workflows, environment drift, failed setup attempts, unsafe intermediate actions, and long repair loops.

This gap leads directly to the evaluation problem in Section 5. If acquisition targets command-observation-recovery behavior, then evaluation should measure not only final success but also the seven terminal-competence dimensions: action formulation, feedback interpretation, runtime management, state and context tracking, verification, recovery, and side-effect control. Without such process-level measurement, reported gains may reflect adaptation to particular repositories, harnesses, or reward channels rather than transferable terminal competence.

5. Benchmarks, Metrics, and Evaluation

Section 4 explains how terminal competence is acquired; this section asks how it should be measured. Terminal agents sit at the intersection of software engineering, tool use, and environment-coupled interaction, so final task success is only one layer of evidence. A recurring problem is that *terminal competence*, understood as the ability to act effectively through terminal environments, is often conflated with *repository repair competence* [15–17,52]. These overlap but are not identical: an agent may repair a repository through mostly static reasoning with limited terminal-mediated interaction, while another may exhibit strong terminal competence on tasks unrelated to repositories. We therefore organize the evaluation literature along three axes: benchmark families, evaluation layers, and coverage of the terminal-competence dimensions introduced in Section 2.

We organize benchmarks descriptively by what they are designed to expose rather than ranking them ordinally, because task formats, scoring protocols, and infrastructure assumptions vary widely across studies. Table 5 summarizes the benchmark families used in this section. To support this synthesis, we conduct a qualitative coverage analysis of representative benchmarks according to their task targets, scoring signals, observable interaction traces, and coverage of the terminal-competence dimensions introduced in Section 2.

 doi:10.20944/preprints202606.1409.v1

Table 5. Benchmark families organized by measured skill and main blind spot. The table groups representative benchmarks according to the primary capability or evaluation concern they are designed to expose.

Family	Measured skill	Representative examples	Blind spot
Repo repair	Issue resolution under tests	SWE-bench [15], SWE-PolyBench [107]	Conflates repository repair with terminal skill; terminal-mediated command use is not directly measured
CLI / terminal-centered	Command use, output interpretation, CLI-mediated workflows	Terminal-Bench [17], TerminalWorld [108], LongCLI-Bench [109]	Mixes terminal-native and repository-mediated workflows; transfer remains unclear
Setup	Environment bootstrap, dependency resolution	SetupBench [52]	Often detached from full workflow evaluation
Process	Intermediate behavior quality, scaffold compliance	OctoBench [68], ProcBench [110], AppWorld [111]	Scoring not standardized across studies
Long horizon	Persistence, drift, coherence over time	SWE-Bench Pro [112], LoCoEval [113], LifelongAgentBench [114]	Expensive to run; hard to reproduce at scale
Safety / governance	Permission use, sandbox containment, privileged-command behavior	BashArena [115], ClawSafety [116], AgentHazard [117]	Safety protocols remain immature; success can hide harmful intermediate actions
Production	Deployment realism, distribution match	ProdCodeBench [118]	Often not fully public; narrow domain coverage

Production-derived benchmarks are included as a benchmark-realism family rather than a separate terminal-competence dimension; their value lies in distribution match and deployment realism rather than in isolating a single terminal skill [21,118].

5.1. From Repository-Level Evaluation to Terminal-Native Benchmarks

SWE-bench [15] established repository-level issue resolution with executable verification as the dominant evaluation paradigm. Later benchmarks expanded language coverage [107,119], project scale [89], temporal validity [41,120], task production realism [118], and multi-language agentic evaluation [121], but most still evaluate terminal interaction indirectly: command execution appears as an implementation channel for repository repair rather than as the measured capability.

Terminal-native and CLI-centered benchmarks increasingly make terminal-mediated interaction part of the task definition [17,108,109,122–124]. They measure command use, output interpretation, state inspection, and workflow persistence, while CLI-Gym [18] provides terminal environments for training and evaluation. Earlier work such as InterCode [51] established the principle of interactive coding with execution feedback. However, this family remains heterogeneous: some tasks are genuinely terminal-native, whereas others remain repository-mediated. Harness choice alone can produce large performance swings, with one study reporting an 18-point difference [123], and proactive state inspection before acting has been identified as the strongest performance discriminator in ClawForge [124]. Transfer across these subtypes is still unclear.

5.2. Process-Aware, Environment-Aware, and Long-Horizon Evaluation

One major shift is from asking *what* the agent produced to *how* it behaved. **Process-aware evaluation** separates task success from scaffold compliance and intermediate behavior quality. OctoBench [68] and debug-gym [125] introduce step-level assessment and defect ontologies; ProcBench [110] and AgentEval [126] add error-propagation tracking and control-preservation evaluation. ToolSandbox [127] contributes stateful, conversational tool-use evaluation, while AppWorld [111] and ASTRA-Bench [128] broaden stateful tool-use and action-planning evaluation beyond repository repair. No shared process-scoring standard exists across these benchmarks.

Environment-aware evaluation treats setup and dependency resolution as measured capabilities rather than pre-task overhead. SetupBench isolates environment bootstrap [52], and process-level configuration studies show setup quality cannot be reduced to a binary flag [67]. For terminal agents, setup is part of the measured capability, not a nuisance phase.

 doi:10.20944/preprints202606.1409.v1

Long-horizon evaluation exposes gaps invisible to short-horizon benchmarks. SWE-EVO [129] reports that, under its evaluation setting, agents scoring 65–73% on SWE-Bench Verified fall to 21–25% on multi-file evolution tasks. SWE-Bench Pro [112], LoCoEval [113], SlopCodeBench [130], Spec Emerges [131], ProjDevBench [132], and RepoMod-Bench [133] show that agent quality erodes under repeated editing even when short-horizon performance appears adequate. NL2Repo-Bench [134] targets repository generation over long horizons, AgencyBench [104] stresses 1M-token real-world contexts, LifelongAgentBench [114] evaluates lifelong-learning behavior, and WildClawBench [123] evaluates real-world deployment scenarios. Terminal competence includes staying aligned and coherent over time, not merely reaching one locally correct endpoint.

5.3. Protocol Validity and Harness Effects

Benchmark scores for terminal agents depend not only on the model and task set, but also on protocol choices that determine what the agent can observe, execute, retry, and remember [16,33,34,135]. Three threats recur across the evaluation literature.

Contamination and temporal validity. Static benchmark pools are vulnerable to memorization, prompt leakage, and stale task distributions. Repository-level benchmarks are particularly exposed because issue descriptions, patches, tests, and discussions may appear in public data. Earlier generalist agent benchmarks such as AgentBench [136] established broader agent-evaluation protocols, but static task pools remain vulnerable to memorization and stale distributions. Mutation-based and freshness-oriented protocols partly address this risk, as in benchmark mutation and rebenchmarking work [41,137,138]; LiveSQLBench [139] extends freshness concerns to dynamic database tasks, and configurable-horizon evaluation such as ACE-Bench [140] offers another route to varying task difficulty and horizon length. Contamination-detection work further suggests that some effects may appear sharply, with models either recalling benchmark artifacts or genuinely reasoning under the tested conditions [141]. Beyond contamination, task design quality directly affects measurement validity: one benchmark-design study [142] reports that more than 15% of tasks in popular terminal-agent benchmarks are reward-hackable, and benchmark design guidelines emphasize adversarial, difficult, and legible task construction. The evaluation question therefore shifts from whether a task is held out to whether contamination, temporal validity, and task regeneration are inspectable.

Harness-mediated variance. For terminal agents, the harness is part of the evaluated system. Observation formatting, command approval, sandbox policy, retry limits, context truncation, tool wrappers, and recovery affordances can change measured performance even when the base model is fixed. Meta-Harness [33] and AutoHarness [34] make this explicit by showing that measurement infrastructure and outer-loop design can themselves be optimized. Agent Psychometrics [135] further decomposes benchmark performance into separable LLM and scaffold ability components using item response theory, enabling task difficulty prediction even for unseen model-scaffold combinations. Direct model-to-model comparison is therefore fragile unless the harness is standardized or fully reported. A system may appear more capable because the interface exposes better diffs, compresses logs more effectively, or prevents destructive actions before they occur.

Environment and budget comparability. Terminal-agent evaluation is also sensitive to execution infrastructure. Container images, dependency caches, network access, timeout policies, filesystem persistence, and allowed external tools can alter both success rates and failure modes [17,52,67]. Production-derived benchmarks reduce distribution mismatch but may sacrifice full public reproducibility or domain breadth [118]. Compute and interaction budgets matter as well: retry budget, number of model calls, maximum trajectory length, verifier calls, and wall-clock timeout all shape the reachable search space.

Reporting checklist. Terminal-agent evaluation papers should report the model, harness, observation format, action interface, permission policy, sandbox configuration, execution environment, network policy, retry budget, timeout, maximum trajectory length, number of model calls, verifier calls, and whether traces are released [16,33,34,135]. These details are not implementation trivia; they are part of the experimental condition.

 doi:10.20944/preprints202606.1409.v1

5.4. Metric Design Beyond Binary Correctness

Outcome metrics such as resolve rate and pass rate anchor most leaderboards but are insufficient for terminal competence. Several additional metric families have emerged. SWE-PolyBench [107] uses syntax-aware structural metrics, and OctoBench [68] uses checklist-based process scoring. Long-horizon studies [130,131] add metrics for drift, faithfulness, and qualitative erosion, while Beyond Binary Correctness argues for multi-dimensional enterprise task assessment [143].

For consistency, we use process-level terms in a narrow sense. *Command economy* refers to the number and complexity of commands required relative to task scope. *Recovery* refers to actions taken after an observable failure signal, such as a nonzero exit status, test failure, exception trace, or dependency conflict. *Diagnostic quality* concerns whether the agent identifies the likely cause of failure before attempting repair. *State tracking* refers to whether subsequent actions remain consistent with filesystem, package, process, and repository state. *Governance violations* include unsafe commands, permission bypasses, sandbox escapes, or unapproved external side effects.

A more complete evaluation stack requires layered metrics. Table 6 organizes six layers, from basic outcome checking through safety governance, with each defined by its evaluation question, measured signal, representative examples, and main weakness.

Table 6. Layered evaluation stack for terminal agents. Each layer is defined by its evaluation question, measured signal, representative examples, and main weakness.

Layer	Question	Signal	Examples	Main weakness
Outcome	Did it finish?	Pass/fail; executable verification	SWE-bench [15], Terminal-Bench [17], TerminalWorld [108]	Binary; hides process quality
Process	How did it proceed?	Step score, command economy, recovery patterns, defect classification	OctoBench [68], ProcBench [110], AppWorld [111]	No shared metric across benchmarks
Environment	Was execution realistic?	Docker/VM state, dependency resolution	SetupBench [52], CLI-Gym [18]	Often detached from full workflows
Trace	Can we inspect the trajectory?	Trajectory logs, command history, replayable traces	Trajectory analyses [144,145], Terminal-Bench [17]	Trace reporting not standardized
Freshness	Is it fresh?	Mutation, live tasks	SWE-rebench [41], LiveSQLBench [139], ACE-Bench [140]	Static pools dominate
Safety	Can execution be contained?	Permission gating; sandbox failures	BashArena [115], ClawSafety [116], AgentHazard [117]	Safety protocols remain immature

The stack reveals an asymmetry: outcome metrics are comparatively standardized, whereas process, trace, freshness, and safety metrics remain weakly specified and inconsistently reported [68, 110,115,116,144,145].

5.5. Benchmark Coverage of Terminal Competence Dimensions

The evaluation literature is organized by benchmark family and metric layer in Tables 5 and 6. A complementary question is which *capability dimensions* representative benchmark families actually measure. Based on our qualitative coding of benchmark task targets, scoring signals, and observable interaction traces, Table 7 maps representative benchmark groups over the seven terminal-competence dimensions from Section 2, with freshness added separately as an evaluation-validity dimension.

We use four qualitative coverage labels. **Full** coverage means that the dimension is an explicit task target or scoring signal. **Partial** coverage means that the dimension is required by many tasks and observable in traces, but is not separately scored. **Weak** coverage means that the dimension may appear incidentally in some tasks but is not a central benchmark target. **Absent** means that the benchmark provides little basis for observing or scoring the dimension. For rows that group related benchmarks, the label reflects the dominant design intent and scoring structure of the group rather

 doi:10.20944/preprints202606.1409.v1

than every individual task instance. The coding reflects benchmark design rather than benchmark quality or empirical difficulty.

Table 7. Qualitative benchmark coverage over the seven terminal-competence dimensions and one evaluation-validity dimension. Cell values summarize coverage coding rather than measured benchmark scores: F=Full, P=Partial, W=Weak, –=Absent. D1=command and action formulation, D2=feedback and artifact interpretation, D3=runtime and environment management, D4=state, task, and context tracking, D5=progress verification, D6=recovery and adaptation, and D7=governance and side-effect control. Evaluation freshness is included separately because it concerns benchmark validity rather than terminal competence itself.

Benchmark	Terminal-competence dimensions							Eval. validity
	D1	D2	D3	D4	D5	D6	D7	Fresh.
SWE-bench / SWE-PolyBench [15,107]	W	P	W	P	P	W	–	–
Terminal-Bench / TerminalWorld [17,108]	F	F	P	P	P	P	–	W
SetupBench [52]	F	F	F	P	P	P	–	–
LongCLI-Bench / GitTaskBench [109,122]	F	F	W	F	P	W	–	–
CLI-Gym [18]	F	F	P	P	P	P	–	–
debug-gym / OctoBench [68,125]	P	F	W	P	F	F	–	–
BashArena / ClawSafety [115,116]	F	F	W	P	P	P	F	–
LiveSQLBench [139]	P	P	–	W	–	–	–	F
WildClawBench [123]	F	F	P	F	P	P	W	–
ClawForge [124]	F	F	P	F	P	W	–	–

This matrix reveals the gap between benchmark coverage and the full capability profile without relying on rhetorical claims about what is missing. Command and action formulation and feedback and artifact interpretation are comparatively well covered, especially in terminal-native and CLI-centered benchmarks. Coverage does not imply high-quality measurement; it only indicates that these dimensions are more directly targeted by benchmark design, scoring signals, or observable interaction traces. Runtime and environment management is measured most explicitly by SetupBench [52], and appears partially in terminal-native environments such as Terminal-Bench [17], TerminalWorld [108], CLI-Gym [18], WildClawBench [123], and ClawForge [124]. State, task, and context tracking becomes more visible in long-horizon and deployment-like settings, but is still often inferred from final outcomes rather than scored as a separate process-level capability. Progress verification and recovery and adaptation are partially covered by process-aware and debugging-oriented benchmarks such as debug-gym and OctoBench [68,125], but remain weakly isolated in many repository-level evaluations. Governance and side-effect control is systematically under-measured outside safety-oriented benchmarks such as BashArena [115] and ClawSafety [116]. Evaluation freshness remains an orthogonal validity concern rather than a competence dimension. This is not a criticism of individual benchmarks, each of which has a focused design purpose, but a structural observation about what the aggregate evaluation landscape measures and omits.

5.6. Remaining Evaluation Gaps

Despite rapid progress, four structural gaps persist.

Terminal-native workflow coverage remains narrow. Most benchmarks remain repository-centric or coding-centric. Broader terminal workflows such as system inspection, data manipulation, ML/scientific workflows, database operations, and cloud operations are only partially represented. Domain-specific efforts provide partial coverage: TerminalWorld [108] broadens coverage toward real-world terminal task recordings, ML-DevBench [146] and MLE-bench [147] target ML workflows, ELTBench [23] and DAComp [148] address data engineering, dynamic Text-to-SQL work [149] covers interactive database exploration, ITBench [21] covers IT operations, ExpBench [150], Curie [24], and

 doi:10.20944/preprints202606.1409.v1

ScienceBoard [151] evaluate scientific or experiment-management workflows, AIOpsLab [22] tests cloud operations, CTF-based evaluations [152,153] and penetration-testing benchmarks [25] target cybersecurity, and ISO-Bench [154] examines inference optimization. These efforts remain fragmented, with no unified benchmark spanning the full range of terminal workflows. Even end-to-end CLI tool generation scenarios remain under-evaluated relative to their prevalence in practice [155].

Process and trace standards remain immature. Benchmarks increasingly log trajectories, but there is no common schema for commands, observations, failures, retries, state changes, and human interventions. Recent trajectory analyses demonstrate the value of trace-level inspection [144,145], and ProcBench [110] proposes step-level assessment. IDE-Bench [156] evaluates agents on real-world IDE tasks and Agent-Diff [157] benchmarks enterprise API tasks with state-diff-based scoring, but no shared trace schema or process-scoring protocol has been adopted across benchmark families. Studies of where and how agents fail underscore the need for standardized failure taxonomies in evaluation [98,158], and interactive debugging tools for agent trajectories remain disconnected from benchmark evaluation protocols [159].

Freshness and contamination control are unresolved. Static task pools become stale quickly, as SWE-rebench demonstrates [41], while live or mutated benchmarks are harder to reproduce and compare. Contamination detection methods are improving [137,141,160] but not yet integrated into most benchmark workflows, and temporal-consistency mechanisms remain experimental [120].

Safety/governance is not integrated with success measurement. Terminal agents can succeed while taking unsafe intermediate actions. Current benchmarks rarely combine task success with permission use, sandbox behavior, reversibility, and external side-effect control. Safety-oriented benchmarks [115,117,161–170] extend evaluation into risk-aware territory; the blind spot of agent safety under benign user instructions remains a structural concern [171].

The field still lacks an integrated evaluation suite that assesses workflow realism, setup ability, process quality, trace visibility, long-horizon persistence, freshness, and safety governance within a single reproducible protocol [41,52,67,110,115,116]. The evaluation evidence therefore shifts attention from leaderboard scores alone to the experimental conditions under which those scores are produced. For terminal agents, the benchmark, harness, runtime, sandbox, observation format, retry budget, and trace-release policy jointly determine what competence is visible and what failure modes remain hidden.

This perspective sets up the cross-cutting synthesis in Section 6. Architectures shape what terminal agents can do, acquisition pipelines shape what they can learn, and evaluation protocols shape what the field can reliably observe. The next section connects these three views and distinguishes supported findings from open or under-measured claims.

6. Cross-Cutting Analytical Insights

Sections 3-5 described architectures, acquisition pipelines, and evaluation protocols. This section synthesizes what these literatures jointly support. We focus on claims that cut across system design, training, and benchmarking: whether terminal agents are analytically distinct, when measured gains can be attributed to models rather than harnesses, which failure modes recur, and where evidence remains too immature for strong conclusions. Because the field is preprint-heavy and methodologically heterogeneous, we treat claims as evidence-calibrated rather than settled. Table 8 provides a compact evidence map.

 doi:10.20944/preprints202606.1409.v1

Table 8. Compact evidence map for cross-cutting claims. The claims have varying evidence strength; the table summarizes the main evidence basis and residual uncertainty for each claim. Detailed discussion appears in the corresponding subsections.

Cross-cutting claim	Evidence basis	Residual uncertainty
Terminal agents are analytically distinct from repository-level coding agents.	Terminal-native and setup-focused benchmarks expose action formulation, feedback interpretation, runtime management, and recovery capabilities beyond repository repair [15,17,52].	How broad and stable this category becomes beyond software engineering and terminal-native benchmarks.
Harness design is a first-class empirical variable.	Action abstraction, observation shaping, context handling, permission policy, and recovery affordances can shift measured performance [16,33,34].	Which harness improvements transfer across models and benchmarks rather than overfitting to a specific task regime.
Benchmarks increasingly measure terminal competence beyond repository repair.	Setup, CLI-centered, process-level, and long-horizon benchmarks broaden measured capabilities [52,68,109,113].	A unified benchmark protocol covering terminal-native setup, recovery, trace quality, safety, and long-horizon persistence.
Diagnosis-to-recovery transfer remains weak.	Process-level studies show recurring environment misunderstanding, dependency failure, ineffective repair, and diagnosis-recovery gaps [67,98,125].	Shared metrics for partial success, recovery quality, and trajectory-level repair behavior.
Cross-domain transfer remains unproven.	Enterprise automation, network operations, data engineering, ML/scientific workflows, and cybersecurity show measurable terminal-relevant tasks [20,23,115,146,172].	Whether SWE-trained terminal skills transfer to non-SWE terminal workflows under fixed harnesses and comparable trace metrics.
Model gains and scaffold gains are not cleanly separable.	Static pipelines can rival interactive agents in some settings, while harness optimization and outer-loop design change scores [33,34,56,173].	Factorial protocols that control model, harness, task regime, execution policy, and scoring.

6.1. Conditions for Comparable Evidence

Before comparing results across terminal-agent papers, three conditions must be made explicit: **task regime**, **execution policy**, and **trace visibility**. These conditions determine whether a reported result is evidence about model capability, harness design, benchmark-specific task structure, or some mixture of all three.

Task regime. The first condition is what kind of workload the benchmark actually measures. Repository-grounded issue resolution remains central for executable verification and cross-file reasoning [15], but a synthesis focused only on repository repair under-observes setup ability, dependency handling, command-execution reliability, and recovery. Setup-focused and CLI-native benchmarks [17,52] are therefore important complements: SetupBench isolates environment bootstrap, while Terminal-Bench centers terminal-mediated interaction itself.

Execution policy. The second condition is how the agent is allowed to act. A clean comparison should state the allowed tool set, sandbox assumptions, command budget, retry policy, context policy, and stopping criteria. The harness determines what the model can see, how actions are serialized, which observations persist, and whether recovery is cheap or brittle [16,35,174]. Evidence about model capability is straightforward to interpret only when the surrounding harness is fixed or sufficiently reported. Evidence about harness quality is clean only when the model family is controlled.

Trace visibility. The third condition is whether the interaction process is observable. Terminal agents act through long, stateful trajectories, so empirical claims are interpretable only when command histories, intermediate observations, error messages, and state changes are preserved [17,44,144,145]. Without traces, one can report leaderboards but cannot distinguish early localization failures from setup breakdowns, misdiagnosis, policy drift, or low-quality recovery.

Task regime specifies what capability is being tested, execution policy specifies what actions and scaffolds are available, and trace visibility specifies whether the resulting behavior can be inspected. Without all three, terminal-agent scores are difficult to attribute to the model, the harness, the benchmark, or the execution environment.

 doi:10.20944/preprints202606.1409.v1

6.2. Model-Side Capability Patterns

When comparisons hold the harness fixed, or at least report it sufficiently, available evidence suggests that model differences in terminal settings are real but highly uneven across capability dimensions. Stronger models often perform better on repository-scale and long-horizon benchmarks, but the improvements are uneven across repository navigation, long-context integration, setup handling, and high-level task persistence [113,122,131]. Yet terminal competence does not reduce to stronger planning or stronger code generation. SetupBench reports low success even for strong agents on environment bootstrap tasks, with especially large gaps in package installation, service initialization, and constraint interpretation [52]. Terminal-Bench likewise suggests that hard terminal tasks remain challenging even for strong contemporary systems, pointing to interaction reliability as a key bottleneck beyond single-step reasoning [17].

A model comparison for terminal agents should not be summarized by one scalar. The capability profile should separate repository localization, command execution correctness, dependency and setup handling, build/test diagnosis, recovery quality, and long-horizon faithfulness. SetupBench isolates bootstrap skill; process-level configuration evaluation isolates diagnostic and repair subskills; long-horizon benchmarks expose specification drift and context erosion [67,130,131]. The most defensible claim is not that one model is universally better, but that different models occupy different points in a structured capability space.

6.3. Harness and Attribution Effects

If model-side comparisons reveal uneven capability profiles, harness-side comparisons reveal a complementary truth: terminal-agent capability is not reducible to the base model alone. SWE-agent [16] provided early evidence that agent-computer interfaces materially affect whether a model can navigate repositories, issue edits, and use execution feedback. OpenHands [27] and terminal-first engineering reports [20,35] further suggest that coding-agent performance is mediated by tool abstraction, file editing primitives, execution guards, and context management. The impact of context on repository-level code generation is now directly measured [175]. Meta-Harness [33] and AutoHarness [34] elevate the harness into an optimization target, showing that outer-loop design can be tuned or synthesized rather than treated as fixed infrastructure.

For terminal agents, the harness shapes performance through coupled design choices. It structures the action space, such as direct terminal commands, higher-level commands, and specialized tools; the observation space, such as outputs, diffs, and metadata; risk controls, such as sandboxing, rollback, and permissions; and trajectory continuity, including what memories or checkpoints persist across turns. SWE-Skills-Bench [176] provides direct evidence that agent skills vary markedly in effectiveness across tasks and that their benefit depends on how well they match the specific harness and workflow.

Automated Harness Evolution (AHE) [177] takes this logic further by making the harness itself an optimization target. In its reported setting, an evolution agent improves system prompts, tools, middleware, and long-term memory through a closed loop with component, experience, and decision observability. The study reports pass@1 gains from 69.7% to 77.0% over 10 iterations and cross-model transfer gains of 5-10 percentage points without repeating the harness-evolution process.

Evidence pattern: Harness-mediated attribution

Pattern. Reported terminal-agent performance is often better understood as a property of the coupled model-harness system rather than the base model alone.
Evidence anchors. ACI-style systems show that action abstraction and observation shaping can make repository navigation, editing, and execution feedback more usable for models. Harness-optimization work further treats context handling, command mediation, and recovery policy as tunable components rather than fixed infrastructure [16,33,34].

 doi:10.20944/preprints202606.1409.v1

Interpretation. Scores may change because the model reasons better, because observations are shaped better, because risky actions are blocked, or because retry and context policies changed.
Residual uncertainty. The field still lacks factorial protocols that separately vary model, harness, task regime, execution policy, and scoring.

Harnesses should therefore be treated as first-class experimental components: they shape measured outcomes and must be reported alongside model details [16,33,34,135]. Without harness information, benchmark improvements may reflect interface shaping, not model improvement.

6.4. Failure, Recovery, and Long-Horizon Drift

Final failure often hides the stage at which a trajectory became unrecoverable. Process-level setup evaluation [67] shows that agents may localize configuration problems yet fail to convert diagnosis into correct repair. SetupBench [52] reveals recurring breakdowns: incomplete tooling installation, hallucinated task constraints, and non-persistent environment modifications. Long-horizon benchmarks [130,131] add specification drift, qualitative degradation under repeated edits, and gradual erosion of coherence. These studies jointly support a failure taxonomy in which environment misunderstanding, dependency failure, command misuse, ineffective recovery, and long-horizon drift are at least as important as incorrect final patches.

Trace-oriented analyses of software engineering agents sharpen this view. Studies of thought-action-result trajectories [144,145] report that successful agents tend to have more stable decision pathways, better use of intermediate evidence, and fewer low-value actions. Related empirical work on agentic pull requests identifies recurring failure categories, including environment misconfiguration, dependency misresolution, and incomplete test coverage, consistent with the trajectory-level failure modes discussed here [158]. Task-stratified analyses [178] report substantial variation in acceptance rates across task types, suggesting aggregate success rates can hide large differences in task difficulty. Behavioral analyses [179] further indicate that drivers of coding-agent success and failure extend beyond resolution rates to interaction patterns and environmental factors.

In one process-level recovery study, PROBE [98] approaches recovery as a structured diagnosis-to-action pipeline: a telemetry layer preserves fine-grained runtime signals from failed runs, a diagnosis layer fuses cross-signal evidence into grounded diagnoses, and a guidance gate produces recovery instructions only when evidence-grounded and actionable. In its reported setting, PROBE reaches 65.4% top-1 diagnosis accuracy, a 43.6 percentage-point improvement over baselines, while recovery reaches only 21.8%. This exposes a diagnosis-recovery gap: knowing what went wrong does not guarantee that the agent can fix it within its behavioral scope.

Evidence pattern: Diagnosis-recovery gap

Pattern. Identifying a plausible cause of failure is often easier than converting that diagnosis into a successful repair in a mutable terminal environment.
Evidence anchors. Setup, debugging, process-level, and long-horizon studies repeatedly surface environment misunderstanding, dependency conflicts, non-persistent modifications, incorrect repair loops, and partial recovery as recurring failure modes [52,67,125].
Interpretation. Terminal agents recover from environments their previous commands may already have changed. This makes rollback, state tracking, and grounded retry behavior central to terminal competence.
Residual uncertainty. Benchmarks rarely standardize partial success, recovery quality, or trajectory-level repair metrics.

 doi:10.20944/preprints202606.1409.v1

Deployment-adjacent evidence should be interpreted cautiously. Studies of agentic refactoring, coding-agent adoption, agent-authored pull requests, open-source contributions, and deployed agent evaluation report behavioral changes, front-loaded productivity gains, rising static-analysis warnings, churn, merge outcomes, modification patterns, and quality degradation patterns that are not visible in offline benchmarks [180–188]. Additional analyses identify maintainability smells, security risks, and vulnerability concerns in AI-generated or agent-authored code [189,190]. These studies identify risks that terminal-grounded agents may amplify, but they usually do not isolate the effect of terminal-mediated interaction from broader agentic coding workflows.

6.5. Boundaries of Current Evidence

The current evidence does not yet support several stronger claims sometimes implied by the literature.

First, terminal agents are best treated as an emerging substrate-defined research area rather than a fully consolidated standalone field. Terminal-native benchmarks and setup-focused tasks expose distinct capabilities, but the field still lacks independent replication across diverse terminal domains [17,20,52,172].

Second, harness improvements should not be interpreted as model-agnostic progress unless they transfer across models, benchmarks, and task families. Outer-loop optimization shows that scaffold design matters, but it also makes attribution harder: a gain may come from observation shaping, command abstraction, permission policy, retry budget, or memory management rather than from improved model reasoning [33,34,173,177].

Third, cross-domain transfer remains unproven. Non-SWE terminal-relevant tasks are measurable in enterprise automation [20], network configuration [172], data engineering [23], ML/scientific workflows [146], and cybersecurity [115], but the field lacks controlled studies showing that SWE-trained terminal skills transfer to these settings under fixed harness and comparable trace metrics.

Fourth, safety and governance remain under-integrated with success measurement. Permission gating, sandboxing, and approval mechanisms appear in deployment-facing systems [20,35], but terminal agents can succeed while taking unsafe or irreversible intermediate actions [115,116,171]. Future evaluation must combine task success with containment, reversibility, external side-effect control, and trace visibility.

The evidence therefore supports a cautious conclusion: terminal agents are a useful and increasingly visible substrate-defined research area, but many stronger claims about maturity, transfer, safety, and model-level progress remain underdetermined.

7. Challenges and Future Directions

Section 6 identified four residual uncertainties: cross-domain transfer beyond software engineering, process quality beyond final outcomes, runtime safety under state-changing execution, and attribution between model gains and harness gains. This section turns these uncertainties into a research agenda. We focus on four directions whose resolution would most improve the validity, transferability, and real-world reliability of terminal-agent research: cross-domain competence, fresh and replayable process-level evaluation, runtime governability, and controlled model-harness attribution.

7.1. Cross-Domain Terminal Competence

Terminal-agent training and evaluation remain concentrated in software engineering [15,16,19]. This concentration is useful because repositories provide executable tests, rich state, and reproducible feedback, but it leaves unresolved whether terminal behaviors learned from software-engineering environments transfer through shared command primitives or merely reflect repository-specific heuristics. Terminal-mediated interaction also appears in enterprise automation, network operations, data engineering, scientific workflows, cloud operations, and cybersecurity [20,22–24,115,172].

A stronger evaluation design would test the same model-harness pair across multiple terminal domains while controlling action interface, observation format, retry budget, and scoring. Such proto-

 doi:10.20944/preprints202606.1409.v1

cols should report both aggregate success and per-domain capability profiles, because a model that transfers command formulation may still fail on environment management, feedback interpretation, or recovery and adaptation in unfamiliar infrastructures. [52,67,98]. The missing piece is therefore not simply more domains, but controlled cross-domain protocols with explicit reporting of domain distribution, environment diversity, training-data overlap, and trace-level behavior.

7.2. Fresh, Replayable, Process-Level Evaluation

Terminal-agent evaluation remains disproportionately outcome-centered. Binary task success is necessary but insufficient: it hides whether an agent used commands efficiently, diagnosed failures correctly, preserved environment state, or recovered from its own mistakes [67,68,110]. This limitation is terminal-specific because commands can mutate filesystem, package, process, and network state. Two agents may both pass a final test while exhibiting very different command economy, recovery behavior, and state-tracking quality.

Future benchmarks should move toward fresh, replayable, process-level protocols. Freshness reduces contamination and staleness through live, regenerated, or temporally updated tasks [41]; replayability preserves command histories, observations, state changes, retries, human interventions, and verifier calls for later inspection [110,144,145,159]. Benchmarks should report command economy, recovery attempts, diagnostic accuracy, failure-retention policy, trace availability, and harness configuration. Without these process-level signals, terminal-agent evaluation will continue to reward final success while obscuring inefficient, brittle, or unsafe paths to that success.

7.3. Runtime Governability and Safety

Terminal agents may install packages, modify repositories, launch processes, interact with networked systems, or operate near credentials and sensitive resources. Safety mechanisms such as permission gating, sandboxing, approval checkpoints, and rollback policies appear in deployed systems, while adjacent safety work studies harmful behavior, code-execution risks, containment, sandboxing, CTF-style evaluation, and penetration-testing frameworks [25,115,116,152,191,192]. Yet systematic safety evaluation for terminal-mediated autonomous execution remains nascent.

The terminal makes safety both harder and more inspectable. It is harder because a single command can cause irreversible filesystem, permission, or network side effects; it is more inspectable because commands and outputs are textual, replayable, and auditable. Future safety protocols should therefore move from generic refusal testing toward runtime governability: command-level permissions, sandbox containment, approval quality, rollback behavior, destructive-command prevention, external side-effect detection, and trace-level accountability [115,116,171]. Without coupling task success with governance and side-effect-control measurement, an agent can appear successful while taking unsafe intermediate actions that would be unacceptable in deployed environments.

7.4. Controlled Model-Harness Attribution

Reported gains in terminal-agent systems often conflate model improvement, harness improvement, and evaluation-protocol differences. This problem is especially severe for terminal agents because the harness controls the action space, observation format, context persistence, permission policy, retry budget, verifier access, and recovery affordances [16,27,33,34]. The same model may behave differently when given direct terminal command access, ACI-mediated primitives, structured diffs, compressed logs, or different sandbox policies.

Future work should treat harness design as an experimental variable rather than as fixed infrastructure. A minimal attribution protocol should include fixed-model comparisons across alternative harnesses, fixed-harness comparisons across models, and fixed model-harness pairs across task regimes. Agentless-style static pipelines [56] show that interactive scaffolding is not always necessary for strong repository-repair performance, while harness-optimization work [33,34,177] shows that outer-loop redesign can shift measured performance. Recent factorial studies [173] provide a starting point, but the field still lacks shared decomposition protocols that control model, harness, action space, task

 doi:10.20944/preprints202606.1409.v1

regime, execution policy, and scoring. Without such protocols, reported improvements will remain difficult to attribute.

The preceding discussion motivates four paired challenges and research paths:

1. **From SWE concentration to cross-domain terminal competence.** Current evidence remains concentrated in software-engineering tasks, so future work should test whether terminal skills transfer to operations, data engineering, scientific workflows, cloud management, and cybersecurity under fixed model-harness settings.
2. **From outcome-only scores to fresh and replayable process evaluation.** Current benchmarks often emphasize final success, so future evaluation should preserve command histories, failure paths, recovery attempts, and trace-level evidence through fresh and replayable protocols.
3. **From generic safety checks to runtime governability.** Current safety evaluation rarely measures state-changing terminal execution together with task success, so future protocols should evaluate command-level permission, sandbox containment, approval quality, rollback behavior, and external side-effect control.
4. **From leaderboard gains to controlled model-harness attribution.** Current improvements often conflate model progress with harness design, so future studies should use factorial protocols to separate model-side gains from scaffold-side gains across task regimes and execution policies.

Table 9 summarizes these directions in terms of near-term targets, minimum reporting requirements, and prerequisites. These directions are mutually reinforcing: cross-domain claims require process-aware evaluation, process-aware evaluation requires governable runtime conditions, and both require controlled model-harness attribution. The near-term priority is to make terminal-agent experiments comparable, auditable, and attributable, rather than only larger in scale.

Table 9. Research agenda for terminal-agent studies, including near-term targets, minimum reporting requirements, and prerequisites.

Research direction	Near-term target	Minimum reporting	Prerequisites
Cross-domain competence	Controlled multi-domain evaluation under fixed model-harness pairs	Domain distribution, environment diversity, per-domain success, training-data overlap	Process-level metrics and shared task formats
Fresh and replayable process-level evaluation	Benchmarks with trace replay, recovery metrics, and freshness-preserving tasks	Command economy, recovery attempts, diagnostic accuracy, failure-retention policy, trace availability, harness configuration	Instrumented runtimes and trace schemas
Runtime governability	Command-level permission, sandbox, approval, rollback, and side-effect evaluation	Containment failures, permission bypasses, destructive-command prevention, approval trace quality, side-effect detection	Standardized threat taxonomy
Model-harness attribution	Factorial studies varying model, harness, task regime, execution policy, and scoring	Model version, harness configuration, action space, observation policy, context budget, retry budget, fixed-harness and fixed-model ablations	Portable harness descriptions

8. Conclusion

This survey examined terminal agents as systems whose progress depends on terminal-mediated action-observation loops. By treating the terminal as an interaction substrate rather than passive infrastructure, we organized the literature around architectures, competence acquisition, and evaluation protocols for terminal-mediated agency. This substrate-centered view separates terminal agents from adjacent categories such as repository-level coding agents, GUI-centered computer-use agents, agentless pipelines, and CLI-packaged assistants whose terminal-facing interface does not necessarily ground task progress.

 doi:10.20944/preprints202606.1409.v1

Across the literature, three conclusions stand out. First, terminal competence is multi-dimensional. Command and action formulation and feedback and artifact interpretation matter, but robust terminal-mediated agency also depends on runtime and environment management, state, task, and context tracking, progress verification, recovery and adaptation, and governance and side-effect control. Long-horizon persistence is best understood as a cross-cutting outcome supported by these dimensions rather than as a separate capability. Second, outer-loop design is a performance-shaping variable rather than an implementation detail. Action abstraction, observation formatting, runtime organization, context management, permission policy, verification mechanisms, and recovery affordances can materially change measured behavior, making the coupled model and harness the relevant unit of analysis in many studies. Third, current acquisition and evaluation pipelines remain better at capturing successful execution than recoverable failure. Clean trajectories, binary pass/fail metrics, and repository-centered tasks under-measure diagnosis-recovery gaps, process quality, trace visibility, progress verification, safety constraints, and transfer beyond software engineering.

The evidence therefore supports a cautious but useful conclusion. Terminal agents are an increasingly visible substrate-defined research area, but the field has not yet established mature cross-domain transfer, clean attribution between model and harness effects, or standardized evaluation of verification, recovery, and side-effect control. Progress will depend less on simply scaling models or enlarging benchmarks than on making terminal-agent experiments comparable, auditable, and attributable. Fixed or fully reported harnesses, replayable traces, process-level metrics, failure-aware training data, and governance-sensitive evaluation protocols should become standard. The terminal offers a powerful substrate for autonomous agents precisely because it is textual, compositional, stateful, and executable. Understanding how agents learn, verify progress, fail, recover, and remain governable in this substrate remains the central research challenge.

Appendix

A. Review Protocol and Corpus Construction

This appendix summarizes the review protocol and corpus construction. We conducted a **structured scoping review with an evidence-stratified coded corpus**. The goal was not to provide an exhaustive systematic review or formal meta-analysis, but to map an emerging and methodologically heterogeneous research area through transparent inclusion rules, coding dimensions, and evidence calibration.

A.1. Corpus Scope and Composition

The review covers work released between 2022 and mid-2026 on terminal agents, terminal-mediated architectures, executable training environments, terminal-native or repository-based evaluation, process-level agent assessment, runtime governance, and adjacent agent paradigms used for boundary comparison. The coded corpus contains **187 unique research entries**. The manuscript bibliography contains **192 distinct cited entries**; the difference consists of four deployment-facing terminal-agent tools, Claude Code, Codex CLI, Aider, and Gemini CLI, and one CLI-packaged assistant boundary comparator, which are cited as engineering-practice references but are not counted as coded academic papers.

Table 10 summarizes the corpus composition. Inclusion tiers indicate how directly a work supports terminal-agent claims. Evidence labels calibrate claim strength rather than paper quality. Placement records whether a work is used in the main text, table taxonomies, or background discussion.

 doi:10.20944/preprints202606.1409.v1

Table 10. Summary of the coded corpus. The coded research corpus contains 187 entries; the manuscript bibliography contains 192 cited entries after adding five engineering-practice tool references that are not coded as academic papers.

Category	Description	Count
Inclusion tier		
Core-Terminal-Primary	Terminal-mediated execution is the dominant locus of task progress.	59
Core-Hybrid-Terminal	Command execution is materially present alongside non-terminal modalities.	70
SWE-Executable-Adjacent	Execution is present, but terminal interaction is not the primary object.	17
Adjacent-Comparator	GUI/browser agents, agentless pipelines, and other boundary cases.	31
Background-Theory	Conceptual or framing sources without direct terminal-agent evidence.	10
Evidence label		
Established	Widely used benchmark, repeated evaluation, peer-reviewed work, or mature baseline.	22
Emerging	Recent preprint, new benchmark, single-system study, or developing empirical setting.	163
Engineering Practice	Deployment-facing or product-engineering reference.	2
Manuscript placement		
Main text	Entry is discussed in the main narrative.	166
Tables only	Entry appears in taxonomy or comparison tables only.	1
Background only	Entry is used for framing or background discussion only.	20

A.2. Search and Screening Procedure

The corpus was assembled through iterative keyword search, venue-oriented search, and backward and forward reference chaining. Sources included arXiv, OpenReview, ACL-family venues, major machine-learning and software-engineering venues, systems-oriented venues, benchmark repositories, and project pages for deployment-facing terminal-agent tools. Query families covered terminal-agent terms, software-engineering agent terms, tool-use and computer-use terms, acquisition and training terms, and evaluation, safety, and runtime-governance terms.

Each candidate work was screened using five questions:

1. Does the agent execute terminal commands, operate CLI tools, or interact with a terminal-mediated environment?
2. Does stdout, stderr, logs, diffs, return codes, or execution feedback materially shape subsequent actions?
3. Does the system produce real or simulated environment state changes through execution?
4. Does the work provide executable verification, trajectory data, a benchmark, an acquisition pipeline, or a runtime architecture relevant to terminal-mediated agency?
5. What claim scope does the work support: core terminal-agent evidence, terminal-hybrid evidence, executable SWE-adjacent evidence, boundary comparison, or background framing?

Based on these questions, each coded entry was assigned to one inclusion tier. The two core tiers, Core-Terminal-Primary and Core-Hybrid-Terminal, form the 129-entry core corpus. The remaining 58 entries support adjacent comparison, SWE-executable context, or theoretical framing. These tiers are not quality labels; they indicate how directly a work supports the terminal-agent synthesis.

A.3. Coding Dimensions and Evidence Calibration

Entries were coded for bibliographic metadata, application domain, inclusion tier, evidence label, claim scope, terminal centrality, primary role, architecture family, verification type, artifact availability, trace visibility, and manuscript placement. Coding was performed primarily at the paper level for corpus statistics and at the artifact level when a paper introduced multiple relevant systems, benchmarks, or datasets.

For architecture analysis in Section 3, relevant entries were coded by interface granularity, operational coupling, autonomy regime, recovery/control style, and planning/control strategy. For acquisition analysis in Section 4, entries were coded by data-source type, trajectory source, supervision signal, filtering or relabeling method, failure-data treatment, and capability target. For evaluation

 doi:10.20944/preprints202606.1409.v1

analysis in Sections 5 and 6, entries were coded by task regime, harness assumptions, verification type, trace visibility, artifact availability, and coverage of the terminal-competence dimensions introduced in Section 2.

Evidence labels are used to calibrate claim strength rather than to rank paper quality. Claims are treated as stronger when they are supported by converging evidence across multiple benchmarks, systems, or domains; as moderate when they are supported by direct but limited empirical evidence; as weak when they rely on single-study or indirect evidence; and as illustrative when they are based mainly on deployment practice or boundary-comparison examples. This calibration supports the evidence maps and cautious claims in the main text.

References

1. 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**.
2. 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**, *36*, 68539–68551.
3. Wang, X.; Chen, Y.; Yuan, L.; Zhang, Y.; Li, Y.; Peng, H.; Ji, H. Executable code actions elicit better llm agents, 2024. *URL https://arxiv. org/abs/2402.01030* **2024**, *6*, 6–2.
4. Parisi, A.; Zhao, Y.; Fiedel, N. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255* **2022**.
5. Hu, X.; Xiong, T.; Yi, B.; Wei, Z.; Xiao, R.; Chen, Y.; Ye, J.; Tao, M.; Zhou, X.; Zhao, Z.; et al. Os agents: A survey on mllm-based agents for general computing devices use. *arXiv preprint arXiv:2508.04482* **2025**.
6. De Masi, A. Terminal Is All You Need: Design Properties for Human-AI Agent Collaboration. *arXiv preprint arXiv:2603.10664* **2026**.
7. 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**, *18*, 186345.
8. Xi, Z.; Chen, W.; Guo, X.; He, W.; Ding, Y.; Hong, B.; Zhang, M.; Wang, J.; Jin, S.; Zhou, E.; et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences* **2025**, *68*, 121101.
9. Wang, Y.; Zhong, W.; Huang, Y.; Shi, E.; Yang, M.; Chen, J.; Li, H.; Ma, Y.; Wang, Q.; Zheng, Z. Agents in software engineering: Survey, landscape, and vision. *Automated Software Engineering* **2025**, *32*, 70.
10. Wang, H.; Gong, J.; Zhang, H.; Xu, J.; Wang, Z. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126* **2025**.
11. Shi, Y.; Yu, W.; Huang, J.; Yao, W.; Chen, W.; Liu, N. Towards trustworthy gui agents: A survey. *arXiv preprint arXiv:2503.23434* **2025**.
12. Ning, L.; Liang, Z.; Jiang, Z.; Qu, H.; Ding, Y.; Fan, W.; Wei, X.y.; Lin, S.; Liu, H.; Yu, P.S.; et al. A survey of webagents: Towards next-generation ai agents for web automation with large foundation models. In Proceedings of the Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2, 2025, pp. 6140–6150.
13. 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**.
14. Dong, Y.; Jiang, X.; Qian, J.; Wang, T.; Zhang, K.; Jin, Z.; Li, G. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083* **2025**.
15. 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, Vol. 2024, pp. 54107–54157.
16. 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**, *37*, 50528–50652.
17. 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**.
18. Lin, Y.; Wang, H.; Wu, S.; Fan, L.; Pan, F.; Zhao, S.; Tu, D. CLI-Gym: Scalable CLI Task Generation via Agentic Environment Inversion. *arXiv preprint arXiv:2602.10999* **2026**.
19. Pan, J.; Wang, X.; Neubig, G.; Jaitly, N.; Ji, H.; Suhr, A.; Zhang, Y. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139* **2024**.

 doi:10.20944/preprints202606.1409.v1

20. Bechard, P.; Ayala, O.M.; Chen, E.; Skelton, J.; Davasam, S.; Sunkara, S.; Yadav, V.; Rajeswar, S. Terminal Agents Suffice for Enterprise Automation. *arXiv preprint arXiv:2604.00073* **2026**.
21. Jha, S.; Arora, R.; Watanabe, Y.; Yanagawa, T.; Chen, Y.; Clark, J.; Bhavya, B.; Verma, M.; Kumar, H.; Kitahara, H.; et al. Itbench: Evaluating ai agents across diverse real-world it automation tasks. *arXiv preprint arXiv:2502.05352* **2025**.
22. Chen, Y.; Shetty, M.; Somashekar, G.; Ma, M.; Simmhan, Y.; Mace, J.; Bansal, C.; Wang, R.; Rajmohan, S. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *Proceedings of Machine Learning and Systems* **2025**, *7*.
23. Jin, T.; Zhu, Y.; Kang, D. Elt-bench: An end-to-end benchmark for evaluating ai agents on elt pipelines. *Proceedings of the VLDB Endowment* **2025**, *19*, 84–98.
24. Kon, P.T.J.; Liu, J.; Ding, Q.; Qiu, Y.; Yang, Z.; Huang, Y.; Srinivasa, J.; Lee, M.; Chowdhury, M.; Chen, A. Curie: Toward rigorous and automated scientific experimentation with ai agents. *arXiv preprint arXiv:2502.16069* **2025**.
25. Liu, Z.; Huang, L.; Zhang, J.; Liu, D.; Tian, Y.; Shao, J. PACEbench: A Framework for Evaluating Practical AI Cyber-Exploitation Capabilities. *arXiv preprint arXiv:2510.11688* **2025**.
26. Parthasarathy, K.; Vaidhyanathan, K.; Dhar, R.; Krishnamachari, V.; Kakran, A.; Akshathala, S.; Arun, S.; Karan, A.; Muhammed, B.; Dubey, S.; et al. Engineering llm powered multi-agent framework for autonomous cloudops. In Proceedings of the 2025 IEEE/ACM 4th International Conference on AI Engineering–Software Engineering for AI (CAIN). IEEE, 2025, pp. 201–211.
27. Wang, X.; Rosenberg, S.; Michelini, J.; Smith, C.; Tran, H.; Nyst, E.; Malhotra, R.; Zhou, X.; Chen, V.; Brennan, R.; et al. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690* **2025**.
28. Ren, J.; Wu, S.; Li, Y.; Zhu, K.; Xu, S.; Feng, B.; Yuan, R.; Zhang, W.; Batista-Navarro, R.; Yang, J.; et al. A Self-Evolving Framework for Efficient Terminal Agents via Observational Context Compression. *arXiv preprint arXiv:2604.19572* **2026**.
29. Anthropic. Claude Code, 2025. Agentic coding tool available through terminal and other development surfaces, with file editing, command execution, and development-tool integration.
30. OpenAI. Codex CLI, 2025. Open-source terminal coding agent that runs locally and supports command-line coding workflows.
31. Gauthier, P. Aider: AI Pair Programming in Your Terminal, 2025. Open-source terminal-native AI pair-programming tool for editing and managing codebases with LLMs.
32. Google. Gemini CLI, 2025. Open-source terminal AI agent for Gemini models with file operations, shell commands, web tools, and MCP integration.
33. 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**.
34. Lou, X.; Lázaro-Gredilla, M.; Dedieu, A.; Wendelken, C.; Lehrach, W.; Murphy, K.P. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329* **2026**.
35. Bui, N.D. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344* **2026**.
36. Sun, W.; Lu, M.; Ling, Z.; Liu, K.; Yao, X.; Yang, Y.; Chen, J. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967* **2025**.
37. She, J. AgentRM: An OS-Inspired Resource Manager for LLM Agent Systems. *arXiv preprint arXiv:2603.13110* **2026**.
38. Zhuang, H.; Xing, H.; Zhang, X. AgentClick: A Skill-Based Human-in-the-Loop Review Layer for Terminal AI Agents. *arXiv preprint arXiv:2604.16520* **2026**.
39. Wu, S.; Li, Y.; Song, Y.; Zhang, W.; Wang, Y.; Batista-Navarro, R.; Yang, X.; Tang, M.; Dai, B.; Yang, J.; et al. Large-Scale Terminal Agentic Trajectory Generation from Dockerized Environments. *arXiv preprint arXiv:2602.01244* **2026**.
40. Gandhi, K.; Garg, S.; Goodman, N.D.; Papailiopoulos, D. Endless Terminals: Scaling RL Environments for Terminal Agents. *arXiv preprint arXiv:2601.16443* **2026**.
41. Badertdinov, I.; Golubev, A.; Nekrashevich, M.; Shevtsov, A.; Karasik, S.; Andriushchenko, A.; Trofimova, M.; Litvintseva, D.; Yangel, B. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems* **2026**, *38*.
42. Xu, T.; Chen, Y.; Li, M. CLEANER: Self-Purified Trajectories Boost Agentic Reinforcement Learning. *arXiv preprint arXiv:2601.15141* **2026**.

 doi:10.20944/preprints202606.1409.v1

43. Ding, L. AgentHER: Hindsight Experience Replay for LLM Agent Trajectory Relabeling. *arXiv preprint arXiv:2603.21357* **2026**.
44. Kang, H.; Suresh, T.; Saad-Falcon, J.; Mirhoseini, A. TRACE: Capability-Targeted Agentic Training. *arXiv preprint arXiv:2604.05336* **2026**.
45. Song, H.; Huang, L.; Sun, S.; Jiang, J.; Le, R.; Cheng, D.; Chen, G.; Hu, Y.; Chen, Z.; Jia, Y.; et al. Swe-master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411* **2026**.
46. Da, J.; Wang, C.; Deng, X.; Ma, Y.; Barhate, N.; Hendryx, S. Agent-rlvr: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425* **2025**.
47. Zhu, K.; Nie, Y.; Li, Y.; Huang, Y.; Wu, J.; Liu, J.; Sun, X.; Yin, Z.; Wang, L.; Liu, Z.; et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274* **2026**.
48. Pi, R.; Lam, G.; Shoeybi, M.; Jannaty, P.; Catanzaro, B.; Ping, W. On data engineering for scaling llm terminal capabilities. *arXiv preprint arXiv:2602.21193* **2026**.
49. Xia, C.S.; Wang, Z.; Yang, Y.; Wei, Y.; Zhang, L. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? *arXiv preprint arXiv:2511.13646* **2025**.
50. Golubev, A.; Trofimova, M.; Polezhaev, S.; Badertdinov, I.; Nekrashevich, M.; Shevtsov, A.; Karasik, S.; Abramov, S.; Andriushchenko, A.; Fisin, F.; et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501* **2025**.
51. Yang, J.; Prabhakar, A.; Narasimhan, K.; Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems* **2023**, *36*, 23826–23854.
52. Arora, A.; Jang, J.; Moghaddam, R.Z. SetupBench: Assessing Software Engineering Agents' Ability to Bootstrap Development Environments. *arXiv preprint arXiv:2507.09063* **2025**.
53. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.D.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* **2021**.
54. 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**, *378*, 1092–1097.
55. Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; Neubig, G. Pal: Program-aided language models. In Proceedings of the International conference on machine learning. PMLR, 2023, pp. 10764–10799.
56. Xia, C.S.; Deng, Y.; Dunn, S.; Zhang, L. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* **2024**.
57. 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**, *37*, 52040–52094.
58. TheR1D. ShellGPT, 2026. Command-line productivity tool powered by large language models for generating shell commands, code snippets, and documentation.
59. Bilal, M.; Crowcroft, J.; Wang, R.; Xu, X.; Dustdar, S. Large Language Models for Agentic NetOps and AIOps: Architectures, Evaluation, and Safety. *arXiv preprint arXiv:2605.12729* **2026**.
60. Vo, N.P.A.; Kesarwani, M.; Mahindru, R.; Narayanaswami, C. FinOps Agent–A Use-Case for IT Infrastructure and Cost Optimization. *arXiv preprint arXiv:2510.25914* **2025**.
61. Ardebili, M.S.; Bartolini, A. Kubeintellect: A modular llm-orchestrated agent framework for end-to-end kubernetes management. *arXiv preprint arXiv:2509.02449* **2025**.
62. Siva, R.; Cheung, K.; Li, L.; Sundaram, G. kRAIG: A Natural Language-Driven Agent for Automated DataOps Pipeline Generation. *arXiv preprint arXiv:2603.20311* **2026**.
63. Zheng, Y.; Hu, Y.; Zhang, W.; Quinn, A. Towards Agentic OS: An LLM Agent Framework for Linux Schedulers. *arXiv preprint arXiv:2509.01245* **2025**.
64. Legrand, M.; Jiang, T.; Feraud, M.; Navet, B.; Taghzouti, Y.; Gandon, F.; Dumont, E.; Nothias, L.F. Mimosa Framework: Toward Evolving Multi-Agent Systems for Scientific Research. *arXiv preprint arXiv:2603.28986* **2026**.
65. 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, Vol. 2024, pp. 15585–15606.
66. Rawles, C.; Clinckemaillie, S.; Chang, Y.; Waltz, J.; Lau, G.; Fair, M.; Li, A.; Bishop, W.; Li, W.; Campbell-Ajala, F.; et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In Proceedings of the International Conference on Learning Representations, 2025, Vol. 2025, pp. 406–441.

 doi:10.20944/preprints202606.1409.v1

67. Kuang, J.; Li, Y.; Zhang, X.; Li, Y.; Yin, D.; Sun, X.; Shen, Y.; Yu, P.S. Process-level trajectory evaluation for environment configuration in software engineering agents. *arXiv preprint arXiv:2510.25694* **2025**.
68. Ding, D.; Liu, S.; Yang, E.; Lin, J.; Chen, Z.; Dou, S.; Guo, H.; Cheng, W.; Zhao, P.; Xiao, C.; et al. OctoBench: Benchmarking Scaffold-Aware Instruction Following in Repository-Grounded Agentic Coding. *arXiv preprint arXiv:2601.10343* **2026**.
69. Wang, R.; Genadi, R.A.; Bouardi, B.E.; Wang, Y.; Koto, F.; Liu, Z.; Baldwin, T.; Li, H. Agentfly: Extensible and scalable reinforcement learning for lm agents. *arXiv preprint arXiv:2507.14897* **2025**.
70. Foerster, H.; Blanchard, T.; Nikolić, K.; Shumailov, I.; Zhang, C.; Mullins, R.; Papernot, N.; Tramèr, F.; Zhao, Y. Camels can use computers too: System-level security for computer use agents. *arXiv preprint arXiv:2601.09923* **2026**.
71. Rabinovich, E.; Tavor, A.A. On the robustness of agentic function calling. In Proceedings of the Proceedings of the 5th Workshop on Trustworthy NLP (TrustNLP 2025), 2025, pp. 298–304.
72. Zhang, J.; Ma, L.; Li, Y.; Wan, F.; Qi, D.; Zhao, X.; Hou, J.; Xie, Z.; Ren, M.; Wu, X.; et al. DockSmith: Scaling Reliable Coding Environments via an Agentic Docker Builder. *arXiv preprint arXiv:2602.00592* **2026**.
73. Chen, Y.; Pan, J.; Clark, J.; Su, Y.; Zheutlin, N.; Bhavya, B.; Arora, R.R.; Deng, Y.; Jha, S.; Xu, T. Stratus: A multi-agent system for autonomous reliability engineering of modern clouds. *Advances in Neural Information Processing Systems* **2026**, *38*, 50119–50165.
74. Lin, X.; Zhang, J.; Deng, G.; Liu, T.; Zhang, T.; Guo, Q.; Chen, R. Ircopilot: Automated incident response with large language models. *arXiv preprint arXiv:2505.20945* **2025**.
75. Abuzakuk, S.; Crijns, L.; Kermarrec, A.M.; Pires, R.; de Vos, M. RIVA: Leveraging LLM Agents for Reliable Configuration Drift Detection. In Proceedings of the Proceedings of the Sixth European Workshop on Machine Learning and Systems, 2026, pp. 499–509.
76. Mei, K.; Zhu, X.; Xu, W.; Hua, W.; Jin, M.; Li, Z.; Xu, S.; Ye, R.; Ge, Y.; Zhang, Y. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971* **2024**.
77. Gong, H.; Li, C.; Chang, R.; Shen, W. Secure and Efficient Access Control for Computer-Use Agents via Context Space. *arXiv preprint arXiv:2509.22256* **2025**.
78. Thompson, M. The Dual-State Architecture for Reliable LLM Agents, 2026, [arXiv:cs.LG/2512.20660].
79. Wu, J.; Hu, M.; Zhu, J.; Pan, J.; Liu, Y.; Xu, M.; Jin, Y. Git context controller: Manage the context of llm-based agents like git. *arXiv preprint arXiv:2508.00031* **2025**.
80. Benkovich, N.; Valkov, V. Agyn: A Multi-Agent System for Team-Based Autonomous Software Engineering. *arXiv preprint arXiv:2602.01465* **2026**.
81. Geng, J.; Neubig, G. Effective Strategies for Asynchronous Software Engineering Agents. *arXiv preprint arXiv:2603.21489* **2026**.
82. 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**.
83. Guo, L.; Wang, Y.; Li, C.; Tao, W.; Yang, P.; Chen, J.; Song, H.; Tang, D.; Zheng, Z. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954* **2025**.
84. Phan, H.N.; Nguyen, T.N.; Nguyen, P.X.; Bui, N.D. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299* **2024**.
85. Garigipati, P.S.; Ayan, O.; Joshi, K.C.; An, X. Beyond State Machines: Executing Network Procedures with Agentic Tool-Calling Sequences. *arXiv preprint arXiv:2605.02584* **2026**.
86. Nakamura, R.; Eguchi, K. How Helpful is LLM Assistance in Network Operations? A Case Study at a Large Demonstration Network. *arXiv preprint arXiv:2605.19627* **2026**.
87. Tao, H.; Zhang, Y.; Tang, Z.; Peng, H.; Zhu, X.; Liu, B.; Yang, Y.; Zhang, Z.; Xu, Z.; Zhang, H.; et al. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *Advances in Neural Information Processing Systems* **2026**, *38*, 15869–15909.
88. Du, Y.; Cai, Y.; Zhou, Y.; Wang, C.; Qian, Y.; Pang, X.; Liu, Q.; Hu, Y.; Chen, S. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975* **2025**.
89. Liang, J.; Lyu, Z.; Liu, Z.; Chen, X.; Nie, P.; Zou, K.; Chen, W. SWE-Next: Scalable Real-World Software Engineering Tasks for Agents. *arXiv preprint arXiv:2603.20691* **2026**.
90. Yang, J.; Lieret, K.; Jimenez, C.; Wettig, A.; Khandpur, K.; Zhang, Y.; Hui, B.; Press, O.; Schmidt, L.; Yang, D. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems* **2026**, *38*.
91. Xie, Y.; Liu, E.; Zhang, G.; Kotalwar, N.; Gandhi, S.; Acharya, S.; Wang, X.; Rose, C.; Neubig, G.; Fried, D. Hybrid-Gym: Training Coding Agents to Generalize Across Tasks. *arXiv preprint arXiv:2602.16819* **2026**.

 doi:10.20944/preprints202606.1409.v1

92. Zhang, B.; Zhu, J.; Shi, Z.; Liu, D.; Tang, R. AgentForesight: Online Auditing for Early Failure Prediction in Multi-Agent Systems. *arXiv preprint arXiv:2605.08715* **2026**.
93. Zeng, Y.; Li, S.; Dong, D.; Xu, R.; Chen, Z.; Zheng, L.; Li, Y.; Zhou, Z.; Zhao, H.; Tian, L.; et al. SWE-Hub: A Unified Production System for Scalable, Executable Software Engineering Tasks. *arXiv preprint arXiv:2603.00575* **2026**.
94. Nam, J.; Yoon, J.; Chen, J.; Shin, J.; Arik, S.; Pfister, T. Mle-star: Machine learning engineering agent via search and targeted refinement. *Advances in Neural Information Processing Systems* **2026**, *38*, 116692–116712.
95. Li, S.; Sun, W.; Li, S.; Talwalkar, A.; Yang, Y. CoMind: Towards Community-Driven Agents for Machine Learning Engineering. *arXiv preprint arXiv:2506.20640* **2025**.
96. Qiang, R.; Zhuang, Y.; Li, Y.; Sagar VK, D.; Zhang, R.; Li, C.; Wong, I.; Yang, S.; Liang, P.; Zhang, C.; et al. Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering. *Advances in Neural Information Processing Systems* **2026**, *38*.
97. Kim, G.J.; Wilf, A.; Morency, L.P.; Fried, D. From Reproduction to Replication: Evaluating Research Agents with Progressive Code Masking. *arXiv preprint arXiv:2506.19724* **2025**.
98. Zhao, C.; Zhang, S.; Lin, Y.; Gu, W.; Chen, Z.; Sun, Y.; Pei, D.; Bansal, C.; Rajmohan, S.; Ma, M. Debugging the Debuggers: Failure-Anchored Structured Recovery for Software Engineering Agents. *arXiv preprint arXiv:2605.08717* **2026**.
99. Zhu, K.; Liu, Z.; Li, B.; Tian, M.; Yang, Y.; Zhang, J.; Han, P.; Xie, Q.; Cui, F.; Zhang, W.; et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370* **2025**.
100. 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**.
101. 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**.
102. Wei, Y.; Duchenne, O.; Copet, J.; Carbonneaux, Q.; Zhang, L.; Fried, D.; Synnaeve, G.; Singh, R.; Wang, S. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems* **2026**, *38*, 78500–78525.
103. Zhou, H.; Chen, Y.; Guo, S.; Yan, X.; Lee, K.H.; Wang, Z.; Lee, K.Y.; Zhang, G.; Shao, K.; Yang, L.; et al. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153* **2025**.
104. Li, K.; Shi, J.; Xiao, Y.; Jiang, M.; Sun, J.; Wu, Y.; Fu, D.; Xia, S.; Cai, X.; Xu, T.; et al. Agencybench: Benchmarking the frontiers of autonomous agents in 1m-token real-world contexts. *arXiv preprint arXiv:2601.11044* **2026**.
105. Zhou, C.; Chai, H.; Chen, W.; Guo, Z.; Shan, R.; Song, Y.; Xu, T.; Yang, Y.; Yu, A.; Zhang, W.; et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224* **2026**.
106. Pabba, A.; Mathai, A.; Chakraborty, A.; Ray, B. Semagent: A semantics aware program repair agent. *arXiv preprint arXiv:2506.16650* **2025**.
107. Rashid, M.S.; Bock, C.; Zhuang, Y.; Buchholz, A.; Esler, T.; Valentin, S.; Franceschi, L.; Wistuba, M.; Sivaprasad, P.T.; Kim, W.J.; et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703* **2025**.
108. Chu, Z.; Hu, J.; Jiang, X.; Zou, P.; Li, H.; Peng, C.; O'Hearn, P.; Barr, E.T.; Harman, M.; Sarro, F.; et al. TerminalWorld: Benchmarking Agents on Real-World Terminal Tasks. *arXiv preprint arXiv:2605.22535* **2026**.
109. Feng, Y.; Sun, J.; Yang, Z.; Ai, J.; Li, C.; Li, Z.; Zhang, F.; He, K.; Ma, R.; Lin, J.; et al. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337* **2026**.
110. He, J.; Jia, J.; Liu, C.; Xue, C.; Song, Y.; Yang, X.; Sun, D. ProcBench: Evaluating Process-Level Defects and Control Preservation in LLM Coding Agents. *arXiv preprint arXiv:2605.20251* **2026**.
111. Trivedi, H.; Khot, T.; Hartmann, M.; Manku, R.; Dong, V.; Li, E.; Gupta, S.; Sabharwal, A.; Balasubramanian, N. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In Proceedings of the Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024, pp. 16022–16076.
112. Deng, X.; Da, J.; Pan, E.; He, Y.Y.; Ide, C.; Garg, K.; Lauffer, N.; Park, A.; Pasari, N.; Rane, C.; et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941* **2025**.
113. Liu, Y.; Zhang, L.; Liu, F.; Lin, P.; Li, X. A Scalable Benchmark for Repository-Oriented Long-Horizon Conversational Context Management. *arXiv preprint arXiv:2603.06358* **2026**.

 doi:10.20944/preprints202606.1409.v1

114. Zheng, J.; Cai, X.; Li, Q.; Zhang, D.; Li, Z.; Zhang, Y.; Song, L.; Ma, Q. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942* **2025**.
115. Kaufman, A.; Lucassen, J.; Tracy, T.; Rushing, C.; Bhatt, A. BashArena: A Control Setting for Highly Privileged AI Agents. *arXiv preprint arXiv:2512.15688* **2025**.
116. Wei, B.; Zhang, Y.; Pan, J.; Mei, K.; Wang, X.; Hamm, J.; Zhu, Z.; Ge, Y. ClawSafety:" Safe" LLMs, Unsafe Agents. *arXiv preprint arXiv:2604.01438* **2026**.
117. Feng, Y.; Ding, Y.; Tan, Y.; Ma, X.; Li, Y.; Wu, Y.; Gao, Y.; Zhai, K.; Guo, Y. Agenthazard: A benchmark for evaluating harmful behavior in computer-use agents. *arXiv preprint arXiv:2604.02947* **2026**.
118. Jha, S.; Paltenghi, M.; Maddila, C.; Murali, V.; Ugare, S.; Chandra, S. REAP: Automatic Curation of Coding Agent Benchmarks from Interactive Production Usage. *arXiv preprint arXiv:2604.01527* **2026**.
119. Zan, D.; Huang, Z.; Liu, W.; Chen, H.; Xin, S.; Zhang, L.; Liu, Q.; Aoyan, L.; Chen, L.; Zhong, X.; et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems* **2026**, *38*.
120. Sun, H.; Yu, T.; Ma, S.; Zhang, Q.; Rao, L.; Tian, C.; et al. ATime-Consistent Benchmark for Repository-Level Software Engineering Evaluation. *arXiv preprint arXiv:2603.26137* **2026**.
121. Adamenko, P.; Ivanov, M.; Valeev, A.; Levichev, R.; Zadorozhny, P.; Lopatin, I.; Babayev, D.; Fenogenova, A.; Malykh, V. SWE-MERA: A Dynamic Benchmark for Agenticly Evaluating Large Language Models on Software Engineering Tasks. *arXiv preprint arXiv:2507.11059* **2025**.
122. Ni, Z.; Wang, H.; Zhang, S.; Lu, S.; He, Z.; Tang, Z.; Hu, S.; Li, B.; Hu, C.; Jiao, B.; et al. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. In Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence, 2026, Vol. 40, pp. 32564–32572.
123. Ding, S.; Dai, X.; Xing, L.; Ding, S.; Liu, Z.; JingYi, Y.; Yang, P.; Zhang, Z.; Wei, X.; Fang, X.; et al. Wild-ClawBench: A Benchmark for Real-World, Long-Horizon Agent Evaluation. *arXiv preprint arXiv:2605.10912* **2026**.
124. Lai, Y.; Xia, P.; Ji, H.; Xiong, K.; Zeng, K.; Liu, J.; Wu, F.; Zhong, J.; Zheng, Z.; Xie, C.; et al. ClawForge: Generating Executable Interactive Benchmarks for Command-Line Agents. *arXiv preprint arXiv:2605.14133* **2026**.
125. Yuan, X.; Moss, M.M.; Feghali, C.E.; Singh, C.; Moldavskaya, D.; MacPhee, D.; Caccia, L.; Pereira, M.; Kim, M.; Sordoni, A.; et al. debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557* **2025**.
126. Guo, D.; Wu, J.; Yiu, S.M. AgentEval: DAG-Structured Step-Level Evaluation for Agentic Workflows with Error Propagation Tracking. *arXiv preprint arXiv:2604.23581* **2026**.
127. Lu, J.; Holleis, T.; Zhang, Y.; Aumayer, B.; Nan, F.; Bai, H.; Ma, S.; Ma, S.; Li, M.; Yin, G.; et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In Proceedings of the Findings of the Association for Computational Linguistics: NAACL 2025, 2025, pp. 1160–1183.
128. Xiu, Z.; Sun, D.Q.; Cheng, K.; Patel, M.; Zhang, Y.; Lu, J.; Attia, O.; Vemulapalli, R.; Tuzel, O.; Cao, M.; et al. ASTRA-bench: Evaluating Tool-Use Agent Reasoning and Action Planning with Personal User Context. *arXiv preprint arXiv:2603.01357* **2026**.
129. Thai, M.V.; Le, T.; Manh, D.N.; Nhat, H.P.; Bui, N.D. SWE-EVO: Benchmarking Coding Agents in Long-Horizon Software Evolution Scenarios. *arXiv preprint arXiv:2512.18470* **2025**.
130. Orlanski, G.; Roy, D.; Yun, A.; Shin, C.; Gu, A.; Ge, A.; Adila, D.; Roberts, N.; Sala, F.; Albarghouthi, A. SlopCodeBench: Benchmarking How Coding Agents Degrade Over Long-Horizon Iterative Tasks. *arXiv preprint arXiv:2603.24755* **2026**.
131. Yan, L.; Chen, X.; Zhang, X. When the Specification Emerges: Benchmarking Faithfulness Loss in Long-Horizon Coding Agents. *arXiv preprint arXiv:2603.17104* **2026**.
132. Lu, P.; Zhang, S.; Hou, Y.; Ye, L.; Huang, C.; Chen, Z.; Zeng, J.; Jiang, H.; Liu, P.; Wang, Y.; et al. ProjDevBench: Benchmarking AI Coding Agents on End-to-End Project Development. *arXiv preprint arXiv:2602.01655* **2026**.
133. Li, X.; Ben-Israel, N.; Raz, Y.; Ahmed, B.; Serebro, D.; Raux, A. RepoMod-Bench: A Benchmark for Code Repository Modernization via Implementation-Agnostic Testing. *arXiv preprint arXiv:2602.22518* **2026**.
134. Ding, J.; Long, S.; Pu, C.; Zhou, H.; Gao, H.; Gao, X.; He, C.; Hou, Y.; Hu, F.; Li, Z.; et al. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents. *arXiv preprint arXiv:2512.12730* **2025**.
135. Ge, C.; Kryvosheieva, D.; Fried, D.; Girit, U.; Hariharan, K. Agent psychometrics: Task-level performance prediction in agentic coding benchmarks. *arXiv preprint arXiv:2604.00594* **2026**.

 doi:10.20944/preprints202606.1409.v1

136. 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, Vol. 2024, pp. 52989–53046.
137. Garg, S.; Steenhoek, B.; Huang, Y. Saving SWE-Bench: A Benchmark Mutation Approach for Realistic Agent Evaluation. *arXiv preprint arXiv:2510.08996* **2025**.
138. Badertdinov, I.; Nekrashevich, M.; Shevtsov, A.; Golubev, A. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866* **2026**.
139. Team, B.; et al. LiveSQLBench: A Dynamic and Contamination-Free Benchmark for Evaluating LLMs on Real-World Text-to-SQL Tasks, 2024.
140. Yang, W.; Song, C.; Li, X.; Ganguly, D.; Ma, C.; Wang, S.; Dou, Z.; Zhou, Y.; Chaudhary, V.; Han, X. ACE-Bench: Agent Configurable Evaluation with Scalable Horizons and Controllable Difficulty under Lightweight Environments. *arXiv e-prints* **2026**, pp. arXiv–2604.
141. Song, T.E. Cross-Context Verification: Hierarchical Detection of Benchmark Contamination through Session-Isolated Analysis. *arXiv preprint arXiv:2603.21454* **2026**.
142. Bercovich, I. What Makes a Good Terminal-Agent Benchmark Task: A Guideline for Adversarial, Difficult, and Legible Evaluation Design. *arXiv preprint arXiv:2604.28093* **2026**.
143. Chandwani, A.; Gupta, I. Beyond Binary Correctness: Scaling Evaluation of Long-Horizon Agents on Subjective Enterprise Tasks. *arXiv preprint arXiv:2603.22744* **2026**.
144. Bouzenia, I.; Pradel, M. Understanding software engineering agents: A study of thought-action-result trajectories. *arXiv preprint arXiv:2506.18824* **2025**.
145. Ceka, I.; Pujar, S.; Ramji, S.; Buratti, L.; Kaiser, G.; Ray, B. Understanding Software Engineering Agents Through the Lens of Traceability: An Empirical Study. *arXiv preprint arXiv:2506.08311* **2025**.
146. Padigela, H.; Shah, C.; Juyal, D. Ml-dev-bench: Comparative analysis of ai agents on ml development workflows. *arXiv preprint arXiv:2502.00964* **2025**.
147. Chan, J.S.; Chowdhury, N.; Jaffe, O.; Aung, J.; Sherburn, D.; Mays, E.; Starace, G.; Liu, K.; Maksin, L.; Patwardhan, T.; et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In Proceedings of the International Conference on Learning Representations, 2025, Vol. 2025, pp. 50466–50494.
148. Lei, F.; Meng, J.; Huang, Y.; Zhao, J.; Zhang, Y.; Luo, J.; Zou, X.; Yang, R.; Shi, W.; Gao, Y.; et al. DAComp: Benchmarking Data Agents across the Full Data Intelligence Lifecycle. *arXiv preprint arXiv:2512.04324* **2025**.
149. Sun, L.; Guo, T.; Liang, H.; Li, Y.; Cai, Q.; Wei, J.; Yu, B.; Zhang, W.; Cui, B. Rethinking Text-to-SQL: Dynamic Multi-turn SQL Interaction for Real-world Database Exploration. *arXiv preprint arXiv:2510.26495* **2025**.
150. Kon, P.T.J.; Liu, J.; Zhu, X.; Ding, Q.; Peng, J.; Xing, J.; Huang, Y.; Qiu, Y.; Srinivasa, J.; Lee, M.; et al. Exp-bench: Can ai conduct ai research experiments? *arXiv preprint arXiv:2505.24785* **2025**.
151. Sun, Q.; Liu, Z.; Ma, C.; Ding, Z.; Xu, F.; Yin, Z.; Zhao, H.; Wu, Z.; Cheng, K.; Liu, Z.; et al. Scienceboard: Evaluating multimodal autonomous agents in realistic scientific workflows. *arXiv preprint arXiv:2505.19897* **2025**.
152. Lee, D.; Bae, G.e.; Yun, I. CTFusion: A CTF-based Benchmark for LLM Agent Evaluation. *arXiv preprint arXiv:2605.11504* **2026**.
153. Al-Kaswan, A.; Plotnikov, M.; Hájek, M.; Vízner, R.; van Deursen, A.; Izadi, M. Do Agents Dream of Root Shells? Partial-Credit Evaluation of LLM Agents in Capture The Flag Challenges. *arXiv preprint arXiv:2604.19354* **2026**.
154. Nangia, A.; Mishra, S.; Gokrani, A.; Chopra, P. ISO-Bench: Can Coding Agents Optimize Real-World Inference Workloads? *arXiv preprint arXiv:2602.19594* **2026**.
155. Hu, R.; Wang, X.; Peng, C.; Gao, C.; Lo, D. Evaluating LLM-Based 0-to-1 Software Generation in End-to-End CLI Tool Scenarios. *arXiv preprint arXiv:2604.06742* **2026**.
156. Mateega, S.; Yang, J.; Costello, T.; Jadhav, S.; Tian, N.; Garcinuño, A. IDE-Bench: Evaluating Large Language Models as IDE Agents on Real-World Software Engineering Tasks. *arXiv preprint arXiv:2601.20886* **2026**.
157. Pysklo, H.M.; Zhuravel, A.; Watson, P.D. Agent-Diff: Benchmarking LLM Agents on Enterprise API Tasks via Code Execution with State-Diff-Based Evaluation. *arXiv preprint arXiv:2602.11224* **2026**.
158. Ehsani, R.; Pathak, S.; Rawal, S.; Mujahid, A.A.; Imran, M.M.; Chatterjee, P. Where Do AI Coding Agents Fail? An Empirical Study of Failed Agentic Pull Requests in GitHub. *arXiv preprint arXiv:2601.15195* **2026**.
159. Hutter, R.; Pradel, M. AgentStepper: Interactive Debugging of Software Development Agents. *arXiv preprint arXiv:2602.06593* **2026**.
160. Chai, J.; Zhe, Y.; Sakuma, J. When Benchmarks Leak: Inference-Time Decontamination for LLMs. *arXiv preprint arXiv:2601.19334* **2026**.

 doi:10.20944/preprints202606.1409.v1

161. Guo, C.; Liu, X.; Xie, C.; Zhou, A.; Zeng, Y.; Lin, Z.; Song, D.; Li, B. Redcode: Risky code execution and generation benchmark for code agents. *Advances in Neural Information Processing Systems* **2024**, *37*, 106190–106236.
162. Li, X.; Choe, K.W.; Liu, Y.; Chen, X.; Tao, C.; You, B.; Chen, W.; Di, Z.; Sun, J.; Zheng, S.; et al. Clawsbench: Evaluating capability and safety of llm productivity agents in simulated workspaces. *arXiv preprint arXiv:2604.05172* **2026**.
163. Chen, J.; Huang, H.; Lyu, Y.; An, J.; Shi, J.; Yang, C.; Zhang, T.; Tian, H.; Li, Y.; Li, Z.; et al. SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. *arXiv preprint arXiv:2509.22097* **2025**.
164. Chen, J.; Huang, H.; Lyu, Y.; An, J.; Shi, J.; Yang, C.; Zhang, T.; Tian, H.; Li, Y.; Li, Z.; et al. SecureVibeBench: Benchmarking Secure Vibe Coding of AI Agents via Reconstructing Vulnerability-Introducing Scenarios, 2026, [arXiv:cs.SE/2509.22097].
165. Dawson, A.; Mulla, R.; Landers, N.; Caldwell, S. Airtbench: Measuring autonomous ai red teaming capabilities in language models. *arXiv preprint arXiv:2506.14682* **2025**.
166. 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*.
167. Chen, T.; Hu, C.; Gao, G.; Liu, D.; Hu, X.; Wang, W. LPS-Bench: Benchmarking Safety Awareness of Computer-Use Agents in Long-Horizon Planning under Benign and Adversarial Scenarios. *arXiv preprint arXiv:2602.03255* **2026**.
168. Marchand, R.; Cathain, A.O.; Wynne, J.; Giavridis, P.M.; Deverett, S.; Wilkinson, J.; Gwartz, J.; Coppock, H. Quantifying frontier llm capabilities for container sandbox escape. *arXiv preprint arXiv:2603.02277* **2026**.
169. Ye, B.; Li, R.; Yang, Q.; Liu, Y.; Yao, L.; Lv, H.; Xie, Z.; An, C.; Li, L.; Kong, L.; et al. Claw-Eval: Towards Trustworthy Evaluation of Autonomous Agents. *arXiv preprint arXiv:2604.06132* **2026**.
170. Jiang, T.; Wang, Y.; Liang, J.; Wang, T. Agentlab: Benchmarking llm agents against long-horizon attacks. *arXiv preprint arXiv:2602.16901* **2026**.
171. Ding, X.; Zhai, S.; Song, L.; Li, J.; Shi, T.; Meade, N.; Reddy, S.; Kang, J.; Zhao, J. The blind spot of agent safety: How benign user instructions expose critical vulnerabilities in computer-use agents. *arXiv preprint arXiv:2604.10577* **2026**.
172. Twabi, A.; Ding, Y.; Kondo, T. NetAgentBench: A State-Centric Benchmark for Evaluating Agentic Network Configuration. *arXiv preprint arXiv:2604.09678* **2026**.
173. Bandel, E.; Yehudai, A.; Eden, L.; Sagron, Y.; Perlitz, Y.; Venezian, E.; Razinkov, N.; Ergas, N.; Ifergan, S.S.; Shlomov, S.; et al. General agent evaluation. *arXiv preprint arXiv:2602.22953* **2026**.
174. Agarwal, A.; Chan, A.; Chandel, S.; Jang, J.; Miller, S.; Moghaddam, R.Z.; Mohylevskyy, Y.; Sundaresan, N.; Tufano, M. Copilot evaluation harness: Evaluating llm-guided software programming. *arXiv preprint arXiv:2402.14261* **2024**.
175. Le Hai, N.; Nguyen, D.M.; Bui, N.D. On the impacts of contexts on repository-level code generation. In Proceedings of the Findings of the Association for Computational Linguistics: NAACL 2025, 2025, pp. 1496–1524.
176. Han, T.; Zhang, Y.; Song, W.; Fang, C.; Chen, Z.; Sun, Y.; Hu, L. SWE-Skills-Bench: Do Agent Skills Actually Help in Real-World Software Engineering? *arXiv preprint arXiv:2603.15401* **2026**.
177. 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**.
178. Pinna, G.; Gong, J.; Williams, D.; Sarro, F. Comparing ai coding agents: A task-stratified analysis of pull request acceptance. *arXiv preprint arXiv:2602.08915* **2026**.
179. Mehtiyev, T.; Assunção, W. Beyond Resolution Rates: Behavioral Drivers of Coding Agent Success and Failure. *arXiv preprint arXiv:2604.02547* **2026**.
180. Horikawa, K.; Li, H.; Kashiwa, Y.; Adams, B.; Iida, H.; Hassan, A.E. Agentic Refactoring: An Empirical Study of AI Coding Agents. *arXiv preprint arXiv:2511.04824* **2025**.
181. Agarwal, S.; He, H.; Vasilescu, B. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. *arXiv preprint arXiv:2601.13597* **2026**.
182. Mihai Popescu, R.; Gros, D.; Botocan, A.; Pandita, R.; Devanbu, P.; Izadi, M. Investigating Autonomous Agent Contributions in the Wild: Activity Patterns and Code Change over Time. *arXiv e-prints* **2026**, pp. arXiv–2604.

 doi:10.20944/preprints202606.1409.v1

183. Li, H.; Zhang, H.; Hassan, A.E. Aidev: studying ai coding agents on github. *arXiv preprint arXiv:2602.09185* **2026**.
184. Rahman, S.; Rabbi, M.F.; Zibran, M. A Task-Level Evaluation of AI Agents in Open-Source Projects. *arXiv preprint arXiv:2602.02345* **2026**.
185. Gao, Y.; Wang, M.; Yu, Y.L. AgentPulse: A Continuous Multi-Signal Framework for Evaluating AI Agents in Deployment. *arXiv preprint arXiv:2604.24038* **2026**.
186. Robbes, R.; Matricon, T.; Degueule, T.; Hora, A.; Zacchiroli, S. Agentic Much? Adoption of Coding Agents on GitHub. *arXiv preprint arXiv:2601.18341* **2026**.
187. Watanabe, M.; Li, H.; Kashiwa, Y.; Reid, B.; Iida, H.; Hassan, A.E. On the use of agentic coding: An empirical study of pull requests on github. *ACM Transactions on Software Engineering and Methodology* **2025**.
188. Ogenrwot, D.; Businge, J. How AI Coding Agents Modify Code: A Large-Scale Study of GitHub Pull Requests. *arXiv preprint arXiv:2601.17581* **2026**.
189. Ghammam, A.; Almukhtar, M. AI builds, We Analyze: An Empirical Study of AI-Generated Build Code Quality. *arXiv preprint arXiv:2601.16839* **2026**.
190. Siddiq, M.L.; Zhao, X.; Lopes, V.C.; Casey, B.; Santos, J. Security in the Age of AI Teammates: An Empirical Study of Agentic Pull Requests on GitHub. *arXiv preprint arXiv:2601.00477* **2026**.
191. Andriushchenko, M.; Souly, A.; Dziemian, M.; Duenas, D.; Lin, M.; Wang, J.; Hendrycks, D.; Zou, A.; Kolter, Z.; Fredrikson, M.; et al. Agentharm: A benchmark for measuring harmfulness of llm agents. In Proceedings of the International Conference on Learning Representations, 2025, Vol. 2025, pp. 79185–79220.
192. Ba, L.; Li, Q.; Li, S. CIBER: A Comprehensive Benchmark for Security Evaluation of Code Interpreter Agents. *arXiv preprint arXiv:2602.19547* **2026**.