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

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

[1]Tongji University, Shanghai, China
[2]Shanghai Jiao Tong University, Shanghai, China
[3]The Hong Kong Polytechnic University, Hong Kong, China
[4]Shanghai Innovation Institute, Shanghai, China

Abstract

Large language model agents increasingly act through terminals, yet existing surveys disperse terminal-mediated behavior across software engineering, tool use, and computer-use research. We regard terminal agents as systems whose dominant progress-bearing action–observation loop is mediated by terminal command execution, textual feedback, and stateful environment interaction. Using terminal-mediated execution as an organizing lens, this survey establishes workload-level boundaries and connects system architecture, competence acquisition, and evaluation through a seven-dimensional terminal competence profile. Our synthesis shows that realized behavior is jointly shaped by the model, interface, harness, runtime, and environment. Executable trajectories ground learning in action consequences, verification, and recovery, whereas prevailing evaluations emphasize final outcomes and expose process quality, recovery, and governance unevenly. Bounded fixed-condition diagnostics illustrate two implications: benchmark families expose different process signals, and matched system comparisons reveal benchmark-dependent performance and limits of component attribution. These findings motivate explicit reporting of system and runtime conditions, supported by replayable traces and process-level evidence. The framework provides a unified basis for studying terminal-mediated agency across software engineering and emerging application domains.

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

1 Introduction

Large language models (LLMs) are evolving from prompt-bound text generators into interactive systems that act in external environments and revise their behavior from feedback [123, 149, 170]. Through iterative action and observation, these systems increasingly operate as agents that execute code, invoke computational tools, navigate graphical interfaces, and modify digital environments over multiple steps [59, 111]. The medium through which such agents act shapes their available

*These authors contributed equally to this research.

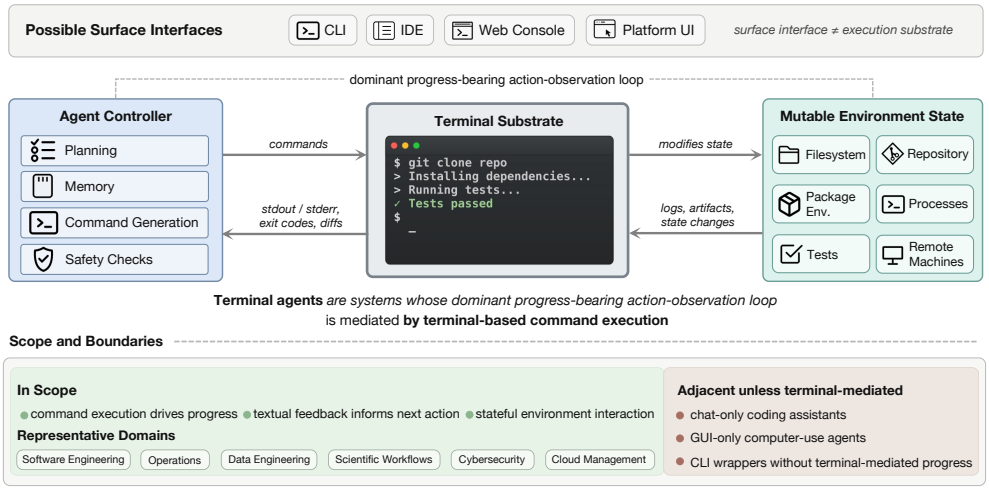

Figure 1: Terminal agents and related concepts. An agent controller, terminal substrate, and mutable environment state form a progress-bearing interaction loop. Systems are in scope when command execution drives progress, textual feedback guides later actions, and stateful environment interaction is central; surface-level CLI access and incidental command use remain adjacent.

actions, observable state, feedback, and means of verification, and is therefore part of the agent system rather than a neutral implementation channel.

The terminal is a particularly consequential execution medium because it provides compact, scriptable textual access to stateful runtimes comprising filesystems, dependencies, processes, tests, logs, remote machines, and command-line tools. Commands can modify files, configure environments, launch services, and execute tests, while outputs, exit codes, diffs, stack traces, logs, and artifacts inform subsequent decisions. Terminal-mediated execution thus couples reasoning, execution, observation, and verification through a mutable action–observation loop [29]. When this loop is the principal mechanism of task progress, the terminal becomes an execution substrate rather than merely an access interface. We accordingly regard systems whose dominant progress-bearing action–observation loop is mediated by terminal command execution, textual feedback, and stateful environment interaction as *terminal agents*. Figure 1 distinguishes this substrate from surface interfaces such as CLIs, IDEs, and web consoles.

This loop makes several general capabilities jointly inspectable in execution traces: planning appears as executable action sequences, memory is tested against persistent state, adaptation is grounded in external feedback, and verification occurs in the same substrate. Terminal agents therefore provide a concrete setting for studying interactive, environment-grounded intelligence. Existing surveys, however, organize related evidence around general LLM agents [147, 156], software-engineering agents [146, 151], GUI- and browser-based computer use [59, 127], vision-language-action models [173], or agent evaluation [36, 172]. Terminal-mediated behavior consequently remains dispersed across autonomy, repository repair, computer use, and evaluation, without a common synthesis of scope, system responsibilities, acquisition pathways, and measurement problems.

The literature supporting such a synthesis remains uneven across domains. Software engineering provides the densest empirical foundation because repositories, tests, and build systems make terminal-mediated behavior directly observable through executable outcomes [67, 101, 165]. Benchmarks, executable training environments, post-training studies, and deployment reports increasingly treat terminal interaction as an explicit design and evaluation target rather than a passive backend [11, 89, 101, 110]. Related terminal-mediated interaction also appears in operations, data

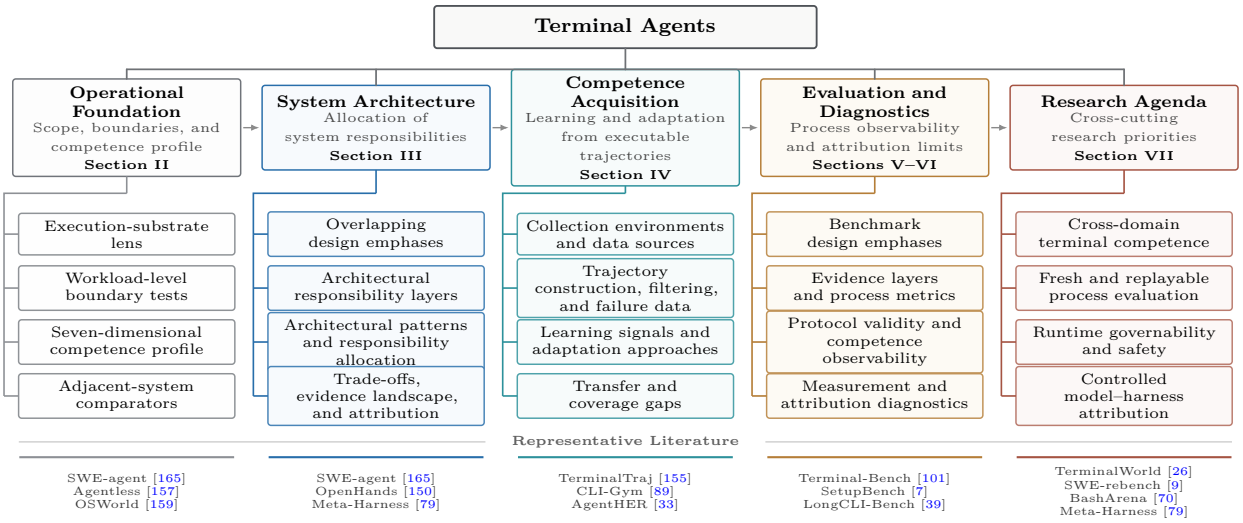

Figure 2: Analytical storyline of the survey. The survey organizes the study of terminal agents around operational foundation, system architecture, competence acquisition, evaluation and diagnostics, and the resulting research agenda. Expanded branches summarize the principal analytical content of Sections II–VII. The column-aligned reference strip provides representative literature anchors and boundary comparators rather than one-to-one mappings to individual subtopics.

engineering, scientific workflows, cybersecurity, and cloud management, but evidence remains fragmented [23, 64, 68, 72, 92, 112]. We therefore distinguish established findings from emerging practices; the Supplementary Material reports the review protocol, corpus construction, and evidence-calibration procedure.

Against this background, terminal-mediated execution provides the organizing lens, while the seven-dimensional terminal competence profile supplies a common analytical language. The survey makes three connected contributions:

- **Operational scope and boundaries.** We provide a substrate-centered characterization and workload-level tests separating progress-bearing terminal execution from surface-level CLI access, incidental command use, and adjacent interaction substrates.
- **Integrated analytical framework.** We connect system architecture, competence acquisition, and evaluation through a seven-dimensional terminal competence profile for comparing system responsibilities, learning signals, and observable evidence.
- **Cross-cutting synthesis and diagnostics.** Applying this framework, we synthesize how the model, interface, harness, runtime, and environment jointly shape terminal-agent behavior, how executable trajectories support competence acquisition, and how prevailing evaluations expose process behavior unevenly. Bounded fixed-condition diagnostics further illustrate benchmark-dependent process exposure and limits of component attribution.

Figure 2 summarizes this progression. Section 2 establishes the operational scope and terminal competence profile. Sections 3 through 5 examine how terminal competence is distributed across system architecture, acquired from executable interaction, and observed through evaluation protocols. Section 6 illustrates benchmark-dependent process exposure and attribution limits through bounded fixed-condition diagnostics. Section 7 presents the research agenda, and Section 8 concludes.

2 Background and Scope of Terminal Agents

The execution-substrate lens requires both an operational boundary and a capability framework. This section establishes the workload-level scope and competence profile used throughout the survey.

2.1 Terminal as a Command-Execution Agent Substrate

The terminal is a distinctive access layer for agentic execution because it combines compact textual interaction, compositional commands, persistent state, and broad operational reach [29]. Through it, an agent can inspect directories, edit files, execute programs, install dependencies, launch processes, query logs, and access remote systems [11]. Unlike GUI-centered environments, where state is inferred largely from visual layouts or interface events, terminal-mediated runtimes return stdout, stderr, exit codes, diffs, stack traces, logs, and process outputs in textual form, narrowing the gap between an LLM's token interface and the environment it controls [89, 165].

Four connected properties make terminal-mediated execution consequential: actions mutate persistent state; observations expose inspectable evidence; reruns and tests connect execution to verification; and commands, patches, and configuration edits form composable textual artifacts [7, 11, 29, 149, 164]. Together, these properties support a reasoning, execution, observation, and verification loop rather than merely a tool-call channel. Figure 3 connects this loop to the boundary tests and competence dimensions introduced below.

Related terms describe different layers of this interaction. A *terminal* is the textual input–output access layer; a *shell* interprets commands; and a *command-line tool* is a textual program invoked through a shell. A *command-execution runtime* supplies the filesystem, processes, dependencies, permissions, and resource boundaries in which commands take effect. A *harness* connects the model to this runtime by formatting observations, exposing actions, managing context, and enforcing execution policy. Terminal-mediated interaction therefore need not involve a visible terminal emulator or persistent pseudo-terminal; instead, command execution and the resulting textual and state evidence must carry task progress. A CLI-exposed product remains outside this scope when the CLI merely forwards requests to an otherwise non-terminal workflow.

2.2 Substrate-Based Scope and Boundaries

Building on this substrate view, we regard *terminal agents* as systems whose dominant progress-bearing interaction is mediated by terminal command execution. The scope is substrate-based rather than surface-based: a system may appear through a CLI, IDE, web interface, or platform runtime, but is in scope only when task progress depends on command execution, textual feedback, and stateful environment interaction.

Occasional terminal use as an auxiliary tool is insufficient [111, 123], as is one-shot code or patch generation without iterative execution and feedback. Terminal agents instead rely on an execution-grounded loop in which commands alter the environment and subsequent observations materially shape later decisions [149, 165, 170]. We operationalize this distinction with three workload-level tests:

- **Primary execution substrate**: Terminal command execution is the workload's main means of progress.
- **Iterative command feedback**: Outputs, errors, logs, diffs, return codes, or state changes

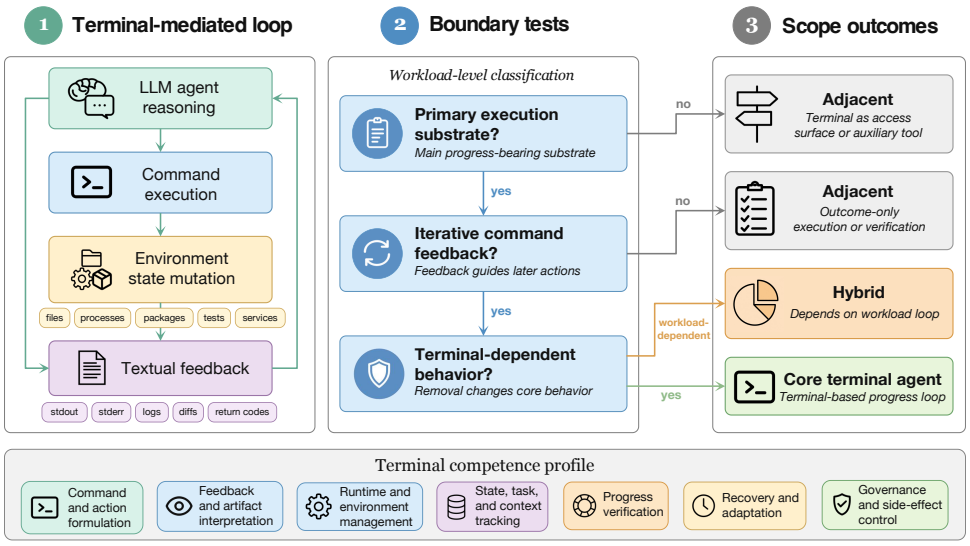

Figure 3: Operational scope of terminal agents, linking terminal-mediated interaction to workload-level boundary tests, scope categories, and competence dimensions.

materially shape later actions.

- **Terminal dependence**: Removing terminal access would materially change the workload's core behavior.

"Dominant" does not require every action to be a direct command; it means that task progress depends causally on terminal-mediated state changes and feedback. The tests apply to individual workloads rather than automatically to an entire product or platform. A hybrid platform may therefore be in scope for repository-repair or operations workloads when terminal execution drives state changes and subsequent decisions, but outside scope when another workload progresses mainly through a browser, GUI, or remote API.

Under these tests, SWE-agent is in scope when command feedback drives repository work [165], as are terminal-centric OpenHands workloads when terminal execution remains the principal locus of progress [150]. The tests exclude static patch generators without iterative execution [157], GUI agents whose primary feedback is visual or DOM-based [159, 188], and CLI-packaged assistants that use the terminal only as an access surface [141]. The same workload-level tests guide review-corpus screening.

2.3 Terminal Competence as a Capability Profile

Having established the workload boundary, we characterize terminal competence through major functional roles rather than low-level commands or domain skills. GUI environments such as OSWorld and AndroidWorld provide useful boundary comparators: they include environment state and executable verification, but visual interaction remains the progress-bearing substrate [120, 159]. Across studies of systems, acquisition, and benchmarks, recurring behaviors and failures concern command selection, textual-feedback interpretation, environment setup and repair, state persistence, verification, recovery, long-horizon control, and execution governance [7, 31, 74, 101, 150, 165].

We group these responsibilities according to the object being controlled, the evidence required for the next decision, and the response needed when execution diverges from the task. This grouping

separates runtime construction from persistent state tracking, verification from post-failure recovery, and authorization control from ordinary command execution. The resulting seven dimensions capture system-level responsibilities distributed across the model, interface, harness, runtime, and environment. They recur in interaction traces, system designs, training pipelines, and benchmark protocols, as summarized in Figure 3.

1. **Command and action formulation**: translating goals, constraints, and state into executable commands, scripts, CLI calls, file edits, build/test/run actions, or structured terminal operations.
2. **Feedback and artifact interpretation**: extracting task-relevant evidence from stdout, stderr, exit codes, logs, diffs, test outputs, stack traces, process signals, generated files, and workspace changes.
3. **Runtime and environment management**: preparing, configuring, maintaining, and repairing dependencies, services, background processes, containers or virtual machines, remote machines, environment variables, resource limits, and related runtime constraints.
4. **State, task, and context tracking**: maintaining environment state, task context, and interaction history across extended sessions, including filesystem and repository changes, packages, processes, configurations, prior commands, partial goals, verified facts, assumptions, and unresolved subproblems.
5. **Progress verification**: designing and executing checks of intermediate validity, artifact trustworthiness, and completion conditions.
6. **Recovery and adaptation**: diagnosing failures, revising hypotheses, replanning execution, mitigating harmful intermediate actions, and retrying from grounded evidence.
7. **Governance and side-effect control**: respecting permissions, sandboxes, approval checkpoints, safety policies, credentials, resource limits, and restrictions on deletion, network access, privilege escalation, or external-system modification.

These dimensions overlap with research on general tool use, software-engineering agents, and computer use, including work on tool selection, feedback, memory, planning, and safety [59, 111, 123, 159, 165]. The profile organizes these concerns around the trace-observable obligations of terminal-mediated execution: commands mutate persistent runtime state, textual artifacts carry evidence across steps, verification occurs in the same substrate, and broad operational reach makes authorization and reversibility integral to competence. It thereby provides a common basis for comparing architecture, acquisition, and evaluation while preserving the distinctive demands of terminal-mediated execution.

The dimensions are analytically separable but operationally interdependent. Recovery depends on feedback interpretation, state tracking, and verification; runtime management depends on action formulation; and long-horizon persistence combines state tracking, verification, and recovery. The following sections use this profile to examine how these responsibilities are distributed across system architecture, acquired through executable interaction, and exposed by evaluation protocols. Section 6 then illustrates selected measurement consequences of this synthesis.

3 Terminal-Agent System Design and Architectures

Terminal competence emerges from interactions among the model, interface, runtime, control mechanisms, harness, and environment. This section traces how these components became explicit

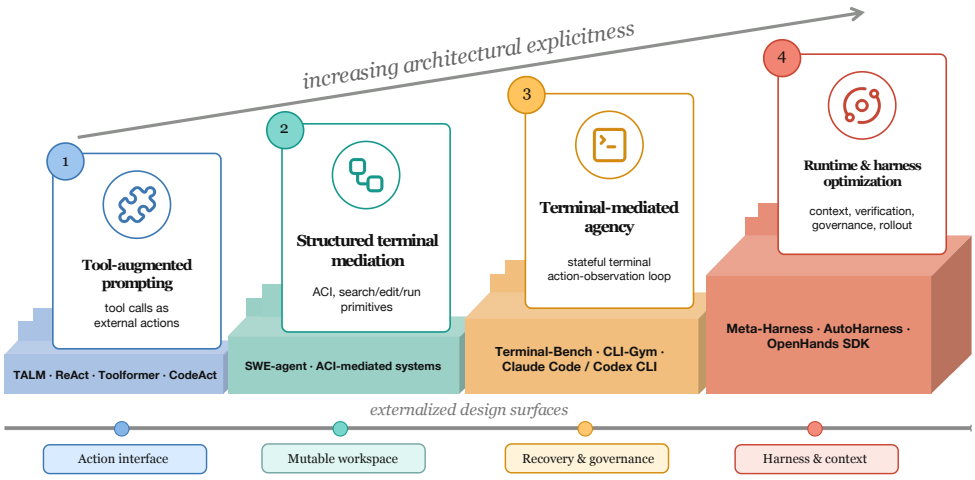

Figure 4: Shifts in design emphasis toward terminal-mediated agency, from tool-augmented prompting to runtime- and harness-centered systems, with interfaces, workspaces, recovery, governance, and context management becoming increasingly explicit.

design objects, organizes their responsibilities into architectural layers and recurring patterns, and synthesizes the resulting trade-offs and attribution consequences.

3.1 Shifts in Design Emphasis toward Terminal-Mediated Agency

Terminal-agent design can be understood through four overlapping shifts in emphasis: tool-augmented prompting, structured executable actions, terminal-mediated agency as a first-class target, and runtime- or harness-centered design. Rather than discrete generations, these shifts reflect the increasing treatment of action interfaces, mutable workspaces, recovery and governance policies, and harness-level context management as explicit design surfaces (Figure 4).

Shift 1: Tool-augmented prompting. TALM [111], ReAct [170], Toolformer [123], and CodeAct [149] established observation-conditioned tool use while treating execution primarily as an external action channel rather than centering persistent terminal-mediated interaction.

Shift 2: Structured executable actions. SWE-agent [165] recasts repository navigation, editing, and execution as model-legible agent–computer interface primitives, moving from raw command access toward mediated interaction. OpenHands [150] extends this direction through a persistent platform runtime coordinating terminal, code, browser, and tool surfaces.

Shift 3: Terminal-mediated agency as a first-class target. Terminal-Bench [101], CLI-Gym [89], Endless Terminals [43], and TermiGen [189] treat terminal interaction as a training or evaluation distribution rather than a by-product of repository repair. Deployment studies document permission-gated command execution [11, 15], while Claude Code [6], Codex CLI [106], Aider [47], and Gemini CLI [52] exhibit recurring patterns of command execution, workspace inspection, textual feedback, approval, and sandboxing.

Shift 4: Runtime- and harness-centered design. Outer-loop elements such as context packaging, workspace persistence, verification, and control flow are increasingly optimized directly. Meta-Harness [79], AutoHarness [93], and Agentic Harness Engineering [87] treat context compaction, approval rules, observation shaping, and observability-guided harness evolution as performance-shaping variables. The OpenHands SDK [150] and extensible RL platforms [148] make runtime

behavior programmable, while collaborative frameworks extend orchestration across multiple agents or execution entities [41]. Interface, workspace, recovery, governance, and context management thus become explicit architectural components.

3.2 Architectural Layers of Terminal Agents

To compare these designs, we organize recurring responsibilities into four layers: interface and observation; runtime and workspace; control, verification, recovery, and governance; and harness and context. These layers locate where design choices enter the terminal-mediated loop and how they support the competence dimensions introduced in Section 2 (Figure 5).

Layer 1: Interface and observation. This layer specifies action units and feedback formats, ranging from direct commands to ACI primitives, runtime events, and workflow-stage interfaces. SWE-agent [165] replaces unconstrained repository interaction with model-legible search, edit, and execution primitives. Observation design determines whether listings, traces, logs, exit codes, diffs, files, and workspace changes are exposed at useful granularity. In its reported setting, TACO reduces token use and improves accuracy through observational context compression [121]. Observation filtering and reinsertion further shape the scale and formatting of available feedback [29], while robustness across interface conditions remains open [117]. This layer primarily supports dimensions 1 and 2.

Layer 2: Runtime and workspace. Because commands mutate environment state, the runtime determines persistence, isolation, and whether later actions can build on earlier ones. OpenHands [150] coordinates terminal, code, browser, and tools over a shared workspace, whereas Terminal-Bench [101] and CLI-Gym [89] make controlled runtime environments central to evaluation. Persistent workspaces enable cumulative progress but can also propagate erroneous intermediate state. This layer governs dependencies, services, background processes, containers or virtual machines, resources, and workspace persistence, making it central to dimensions 3 and 4. Automated Docker image construction [179] further shows that workspace preparation can itself become an optimization target.

Layer 3: Control, verification, recovery, and governance. Terminal access can modify filesystems, install packages, use credentials, and trigger external side effects. This layer structures checks, error bounding, recovery, and oversight through testing, sandboxing, rollback, approval, and decomposition. STRATUS uses role-delegated planning, execution, and review [24]; related principles appear in incident response [13, 88] and configuration-drift detection [1]. AgentClick introduces skill-based checkpoints [191]. OS-level resource management [99, 125], verified deployment of generated Linux scheduling policies [184], context-space access control [51], and reliable state management [142] further make permission, accountability, refusal, and side-effect control architectural concerns. This layer primarily supports dimensions 5 to 7.

Layer 4: Harness and context. The harness packages history, formats prompts, manages context windows, and sequences model calls, determining what remains available across turns. Version-control-inspired methods offer alternatives to raw truncation [154], and controlled comparisons show that context selection affects repository-level generation [77]. Meta-Harness [79] and AutoHarness [93] report performance changes from outer-loop optimization, while multi-agent and asynchronous strategies broaden the orchestration space [12, 49]. This layer primarily supports dimension 4 and long-horizon persistence by preserving trajectory information, verified facts, unresolved subproblems, and prior actions.

Together, these layers show that terminal competence is distributed across the model, interface,

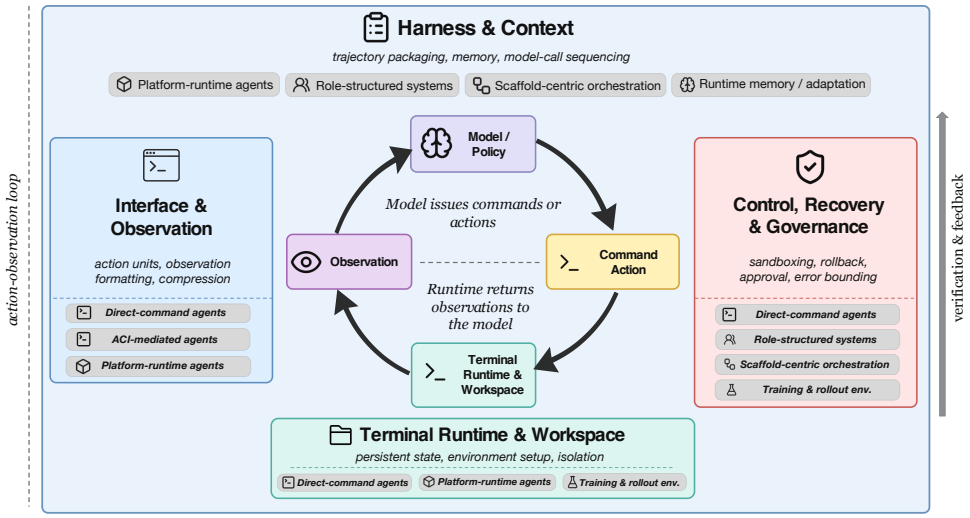

Figure 5: Layered architecture of terminal-agent systems. The central interaction loop is shaped by interface, runtime, verification, recovery, governance, and harness-level context mechanisms.

runtime, control mechanisms, and harness. Long-horizon persistence therefore depends jointly on state and context tracking, runtime persistence, verification, and recovery.

A promising architectural direction is tighter state-aware coordination across these layers. Future systems could condition context management, verification, recovery, and governance on shared execution state and task progress, allowing observation, context allocation, verification, recovery, and execution policy to adapt coherently over long trajectories.

3.3 Recurring Architectural Patterns and Responsibility Allocation

The layers above locate architectural responsibilities, while recurring patterns describe how systems allocate them. These patterns are complementary rather than mutually exclusive and differ in action mediation, runtime persistence, control organization, context management, and rollout infrastructure. Direct-command access exposes broad command spaces behind permission or sandbox controls [6, 47, 52, 106], whereas ACI mediation constrains search, edit, and execution through model-legible primitives [165]. Platform runtimes coordinate persistent workspaces and multiple interaction surfaces [150]; role-structured systems separate planning, execution, and review [24, 113]; and scaffold-centric systems manage context, observations, and control flow [79, 93]. Runtime-memory methods compress or retrieve long-session state [121, 136], while terminal-native rollout environments provide scalable infrastructure for training and trajectory generation [43, 63, 89, 189].

These patterns allocate responsibility differently. Direct access favors expressiveness but increases observation noise and rollback difficulty. Interface mediation favors reliability but may reduce cross-task generality. Persistent runtimes broaden the task surface while increasing dependence on the surrounding system, whereas context optimization supports long-horizon coherence while complicating attribution. Architectures are therefore better compared by how they allocate responsibility across the model, interface, runtime, control mechanisms, and harness than by assignment to a single family.

3.4 Synthesis: Architectural Trade-offs and Attribution Consequences

These alternative responsibility allocations create three recurring architectural tensions and one attribution consequence.

Expressiveness vs. recoverability. Raw or weakly mediated access expands the action space but produces noisier trajectories and harder rollback; ACI and scaffold constraints improve reliability yet may restrict other task families. This trade-off chiefly concerns dimensions 1 and 6, with verification supplying evidence for recovery.

Generality vs. task discipline. Platform runtimes support heterogeneous terminal, code, and browser work but may provide less structured feedback. Role- or workflow-constrained systems strengthen local control at the cost of broader applicability. This tension primarily affects dimensions 3 and 4.

Automation vs. inspectability. Deployable agents must keep commands, outputs, state changes, and interventions legible. Permission gates, approvals, and sandboxes improve auditability but add latency and interrupt autonomous execution. This tension centers on dimension 7, which remains among the least systematically addressed.

Model capability vs. harness contribution. A further consequence is attribution difficulty: gains under optimized harnesses may arise from context management, observation shaping, retries, permission policy, or injected procedural knowledge rather than stronger model reasoning. Controlled skill injection produces task-dependent gains and can add substantial token overhead without improving pass rate [56]. Agentless likewise reports that static pipelines can rival interactive agents on some repository-repair tasks [157]. The complete model–harness–runtime configuration therefore remains relevant to system-level comparison, while component-level attribution requires separating these contributions.

3.5 Evidence Landscape and Boundary Comparators

The architectural literature spans established agent systems, controlled harness studies, deployment reports, and emerging training and operational settings. SWE-agent [165] and OpenHands [150] provide broadly used system designs and evaluation settings, whereas Meta-Harness [79] and AutoHarness [93] examine harness optimization within individual studies. Deployment studies and commercial direct-command agents document recurring interface, permission, sandbox, and workflow patterns [6, 11, 15, 47, 52, 106]. Training environments demonstrate scalable rollout generation, while evidence from live workflows and network operations is still emerging [46, 102].

Adjacent systems further clarify the architectural boundary. Agentless [157] represents static execution and verification pipelines, CGM [137] uses graph-structured control, OSWorld [159] grounds progress primarily in GUI interaction, and CLI-packaged assistants [141] use the terminal mainly as an access surface. These comparators distinguish progress-bearing terminal interaction from systems that share only selected architectural components.

Architectural choices determine executable actions, recorded observations and state changes, available interventions, and the failures entering a trajectory. They thereby shape both the supervision that acquisition pipelines can construct and the evidence that evaluation protocols can observe. Section 4 examines how executable interactions become learning signals, while Section 5 examines how the resulting behavior becomes measurable evidence.

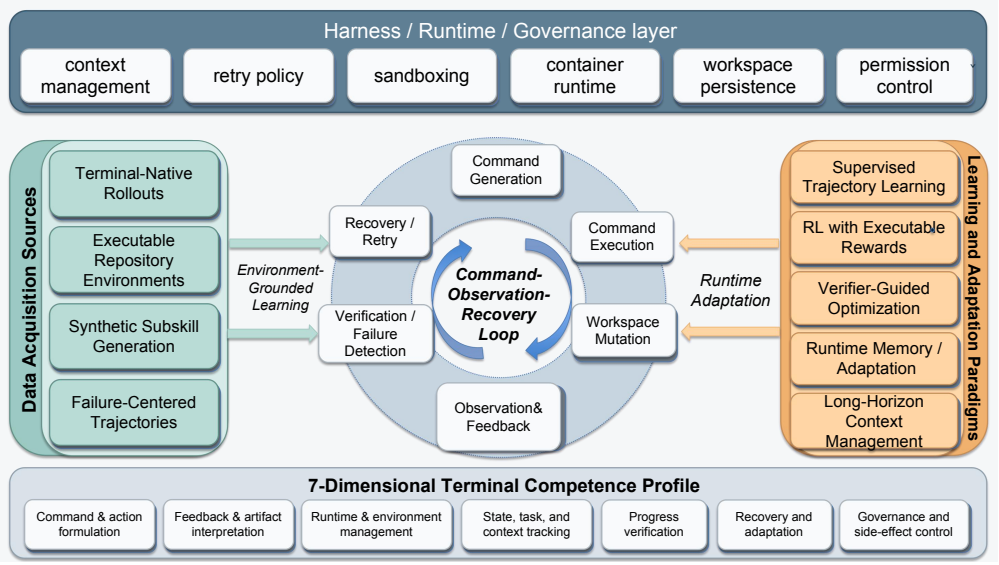

Figure 6: Terminal competence acquisition ecosystem. Data sources, interaction traces, runtime and governance mechanisms, and learning and adaptation approaches jointly shape the multidimensional competence profile.

4 Terminal Competence Acquisition and Adaptation

Building on the architectural account in Section 3, competence acquisition concerns how executable interactions become learning and adaptation signals. Here, acquisition spans parameter learning and runtime adaptation. Training data inherit the action interface, harness, runtime, and governance conditions under which they are collected. Because commands mutate state, expose feedback and failures, and require continuation or repair decisions [43, 69, 114, 155], the relevant unit is a stateful trajectory containing actions, observations, state changes, verification, and recovery rather than an isolated prompt–response pair. This section follows the acquisition path from collection environments through trajectory construction and learning to runtime adaptation, then summarizes transfer and coverage gaps.

Figure 6 connects acquisition sources, interaction and recovery traces, runtime and governance mechanisms, learning and adaptation approaches, and the resulting competence profile.

4.1 Data Sources and Collection Settings for Terminal Competence

Collection environments determine which actions, observations, state changes, and failures become available as learning evidence. Terminal-native rollouts expose command selection, output interpretation, and recovery [43, 63, 89, 155]. Executable repository environments add navigation, editing, setup, and test-grounded verification, while coupling these behaviors to repository-specific reasoning [9, 37, 55, 86, 110]. Machine-learning engineering environments extend the loop to iterative experimentation and component refinement [103, 116]. Synthetic environments target sparse subskills but risk overfitting to generated patterns [100, 166, 189]. Recent synthesis pipelines further scale executable data generation by jointly constructing instructions, environments, reference solutions, and verifiers through taxonomy-guided, evolutionary, or recursive procedures [60, 84, 126]. Failure-centered corpora instead preserve diagnosis, rollback, and recovery that successful traces often omit [33, 69, 178].

Environment structure also determines which competence dimensions are exposed for supervision. Short rollouts primarily expose action formulation and feedback interpretation; setup tasks add runtime management; long trajectories stress state and context tracking; and executable checks support verification. Retained failures expose diagnosis and recovery, while governance requires settings in which authorization, containment, and side effects are visible.

4.2 Trajectory Construction, Filtering, and Failure Data

A rollout becomes usable training data through selection, validation, filtering, and relabeling. These operations shape both data quality and the behaviors preserved for learning [33, 69, 114, 155, 162, 189]. Dockerized generation integrates task adaptation, synthetic generation, rollout collection, filtering, and decontamination [155]. CLEANER filters trajectories for reinforcement learning [162], while AgentHER validates and relabels them through hindsight replay [33]. Trajectory utility also depends on preserved interaction structure: TerminalLego reports stronger training signals from environment-grounded inspect–act–verify trajectories than from teacher success alone in its setting [167]. Other mechanisms alter what remains visible during learning: progressive code masking varies access to prior code [71]; Nemotron-Terminal constructs terminal-oriented training data [114]; Live-SWE-agent studies online self-evolution [158]; and long-context multi-turn reinforcement learning retains extended interactions [50].

Recoverable failures remain underrepresented. Failed installations, version conflicts, invalid environment assumptions, rollback decisions, and dead-end repairs expose recovery more directly than final successful commands [33, 69, 178, 180, 190]. Yet successful-trace filtering can remove the interactions needed to learn diagnosis and adaptation [33, 162, 190]. Retaining misdiagnosis, rollback, and repair therefore provides direct supervision for recovery under observed failure modes [33, 69, 180].

4.3 Learning Signals and Adaptation Approaches

Constructed trajectories support complementary levers at three levels: data construction, parameter optimization, and runtime adaptation. These levers can be combined within a single acquisition pipeline. Table 1 compares their primary signal, capability target, blind spot, and principal risk.

At the parameter-optimization level, **SFT on successful traces** teaches common commands, repository workflows, and setup patterns. Nemotron-Terminal [114] uses terminal-oriented data engineering, while SWE-Gym [110] and SWE-Dev [37] provide executable software trajectories. Their emphasis on successful interactions, however, offers limited supervision for diagnosis, rollback, and recovery after incorrect assumptions [169, 176]. **RL with executable rewards** aligns behavior with task completion through environments and verifiers, as in Endless Terminals [43], ECHO [128], SWE-Master [131], SWE-Gym [110], and Tmax [62], but sparse rewards may reinforce brittle or unsafe behavior. **Process-aware and verifier-guided optimization** instead supervises intermediate decisions through judgments, rankings, or hindsight validation [33, 153]. AgentHER [33] is especially relevant because terminal failures often appear early through stderr, logs, failed tests, or inconsistent state.

Data-oriented and runtime approaches complement parameter optimization. **Synthetic task generation** expands coverage of rare commands, dependency search, repair, localization, and environment construction through verifiable tasks [100, 166, 189], but may teach synthetic regularities rather than reusable competence. **Failure-conditioned training** uses repaired or failed traces from TRACE [69], AgentHER [33], and AgentForesight [178] for diagnosis, early failure prediction,

Table 1: Complementary approaches to terminal competence acquisition and adaptation.

Approach	Primary signal	Capability target	Blind spot	Key risk
SFT on successful traces	Successful interaction traces	Common commands, repository navigation, and standard workflows	Recovery, rollback, and diagnosis after wrong assumptions	Clean-trace overfit; absent failure paths
RL with environment rewards	Executable success, verifier reward, and test outcome	Outcome-driven exploration and completion	Process quality and safe intermediate behavior	Sparse-reward hacking; benchmark overfit
Process-aware / verifier-guided	Step judgments and verifier rankings	Diagnosis, action selection, and recovery decisions	Open-domain transfer	Verifier bias and shifted failures
Synthetic task generation	Generated tasks with controlled verification	Rare commands, setup patterns, and targeted subskills	Live realism and environment drift	Synthetic heuristics; weak transfer
Failure-conditioned training	Failed or repaired traces; hindsight relabeling	Recovery, diagnosis, and early failure detection	Generalization across inconsistent failure taxonomies	Noisy labels; repair-loop overfit
Runtime memory / context adaptation	Session history, summaries, and retrieved experience	Long-horizon persistence and workflow reuse	Correction under incorrect memory	Compression loss; memory contamination

hindsight relabeling, and recovery; its main obstacles are noisy labels, inconsistent taxonomies, and repair-loop overfit. **Runtime memory and context adaptation** extends acquisition into online behavior: Memento [187], Context-Folding [136], and TACO [121] retain history, summaries, or compressed context across long interactions, while risking persistence of incorrect assumptions.

These approaches supervise different parts of an interaction history: successful traces teach common workflows, executable rewards favor completion, verifiers and hindsight expose intermediate decisions, synthetic tasks broaden coverage, failure-conditioned data supports recovery, and runtime memory sustains long-horizon behavior. Their shared challenge is to combine these signals without overfitting to benchmark tasks, generated environments, or harness-specific rewards.

4.4 Emerging Practices and Remaining Gaps

Recent systems increasingly treat acquisition as an end-to-end pipeline in which sourcing, rollout generation, filtering, replay, and curriculum design are distinct levers [110, 114, 155]. Such pipelines increasingly retain setup attempts, outputs, state changes, failed commands, verifier feedback, and recovery decisions, yet current corpora remain concentrated on successful repository-centric traces, with limited coverage of failed setup, long repair loops, environment drift, and unsafe intermediate actions [33, 69, 162].

Transfer remains a central unresolved issue. Repository-repair training may produce task-specific heuristics rather than general terminal competence [157], motivating work on action, observation, and reward sequences that encode reusable skills across environments [81]. Work on externalization [186] and semantics-aware repair [108] further suggests that transfer depends on how explicitly trajectories preserve intermediate reasoning and execution evidence. Cross-domain workflows, environment drift, failed setup, unsafe intermediate actions, and long repair loops remain weakly represented.

These gaps make evaluation evidence essential for determining which aspects of terminal competence are actually acquired. Final success alone does not reveal whether an agent preserved state, recovered from failure, verified completion, or respected execution constraints. Section 5 therefore examines

Table 2: Benchmark design emphases by primary evaluation focus and principal blind spot.

Emphasis	Primary evaluation focus	Representative examples	Principal blind spot
Repository repair	Issue resolution under tests	SWE-bench and SWE-PolyBench [67, 119]	Conflates repository repair with terminal competence; command use is not directly measured
CLI/terminal-centered	Commands, output interpretation, and CLI workflows	Terminal-Bench, TerminalWorld, and LongCLI-Bench [26, 39, 101]	Mixes terminal-native and repository-mediated tasks, leaving transfer unclear
Setup	Environment setup and dependency resolution	SetupBench [7]	Often isolated from end-to-end workflows
Process	Intermediate behavior and scaffold compliance	OctoBench, ProcBench, and AppWorld [31, 57, 143]	Lacks standardized process scoring
Long horizon	Persistence, drift, and temporal coherence	SWE-Bench Pro, LoCoEval, and LifelongAgentBench [30, 91, 182]	Costly and difficult to reproduce at scale
Safety/governance	Permissions, containment, and privileged commands	BashArena, ClawSafety, and AgentHazard [40, 70, 152]	Immature protocols; task success may conceal harmful actions
Production	Distribution match and deployment realism	ProdCodeBench [65]	Limited public access and domain coverage

which benchmark families and evidence layers make these behaviors observable, helping distinguish transferable terminal competence from adaptation to particular repositories, harnesses, environments, or reward channels.

5 Benchmarks, Metrics, and Evaluation

Evaluation determines which aspects of the architectures and acquisition pipelines discussed above become observable and which remain hidden behind final outcomes. Terminal-agent evaluation spans software engineering, tool use, and environment-coupled interaction, making it important to distinguish *terminal competence* from *repository repair competence* [7, 67, 101, 165]. The two overlap but are not equivalent.

We organize the literature along four analytical axes: benchmark design emphases characterize task settings and evaluation targets; evidence layers specify what is recorded; the coverage map estimates which competence dimensions become observable; and protocol validity determines how scores can be interpreted across evaluation settings. Together, these axes separate what a benchmark asks agents to do, what evidence it records, and what conclusions its protocol supports.

Table 2 groups non-exclusive benchmark design emphases by their primary evaluation focus and principal blind spot. They are descriptive rather than ranked because their tasks, scoring signals, and execution infrastructure differ.

Production-derived evaluation cuts across these emphases by improving distribution match without isolating a single terminal skill [64, 65]. Repository-scale deployment studies add evidence about adoption, quality, security-relevant changes, and task-conditioned acceptance [3, 115, 122, 129], but provide limited evidence about individual dimensions of terminal competence.

5.1 From Repository-Level Evaluation to Terminal-Native Benchmarks

SWE-bench [67] established repository issue resolution with executable verification as a dominant evaluation paradigm. Successors expanded language coverage [119, 175], project scale [86], temporal validity [9, 134], production realism [65], and multilingual agentic evaluation [2]. SWE-Hub integrates environment construction, task synthesis, and executable validation in a scalable production pipeline [177]. Claw-SWE-Bench adds multilingual repository tasks and a Lite subset suited to matched system comparisons [183]. Section 6 pairs Claw-SWE-Bench Lite with SWE-bench Lite: the former broadens repository and language diversity, while the latter provides a widely used compatibility anchor. Both remain repository-repair protocols with indirect coverage of broader terminal competence.

Terminal-native and CLI-centered benchmarks instead make command-mediated interaction part of the task definition [26, 34, 39, 76, 101, 105]. They directly expose command use, output interpretation, state inspection, and workflow persistence. TUA-Bench extends this design beyond technical and programming-centric workflows to routine digital activities and scientific and engineering work conducted through a terminal [21]. CLI-Gym [89] supports both training and evaluation, while InterCode [164] anticipated this direction through interactive coding with execution feedback. This group remains heterogeneous: tasks range from terminal-native workflows to repository-mediated work, WildClawBench reports an 18-point harness-conditioned gap [34], and ClawForge identifies proactive state inspection as a strong discriminator in its setting [76]. Transfer across these task types remains insufficiently established.

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

Evaluation is expanding from measuring only *what* agents produce to examining *how* they interact with mutable environments. **Process-aware evaluation** separates final success from intermediate quality and scaffold compliance. OctoBench [31] and debug-gym [174] add step-level assessment and defect ontologies; ProcBench [57] and AgentEval [54] assess error propagation and control preservation; and ToolSandbox [94], AppWorld [143], and ASTRA-Bench [160] extend stateful tool use and action planning beyond repository repair. These efforts expose intermediate behavior, although no shared process-scoring standard has emerged.

Environment-aware evaluation treats setup and dependency resolution as task components rather than pre-task overhead. SetupBench isolates environment bootstrap [7], while process-level configuration studies identify setup defects that binary success misses [74]. Environment setup thereby becomes an explicit component of terminal competence and end-to-end evaluation.

Long-horizon evaluation reveals failures hidden by short tasks. In its reported setting, SWE-EVO [139] reports a drop from 65–73% on SWE-Bench Verified to 21–25% on multi-file evolution. SWE-Bench Pro [30], LoCoEval [91], SlopCodeBench [107], Spec Emerges [163], ProjDevBench [95], and RepoMod-Bench [83] similarly expose degradation under repeated editing. Long-Horizon-Terminal-Bench adds dense subtask rewards and partial-credit grading to terminal workflows that require sustained execution [85]. NL2Repo-Bench [32] targets repository generation, Agency-Bench [81] stresses 1M-token contexts, LifelongAgentBench [182] evaluates lifelong behavior, and WildClawBench [34] covers deployment scenarios. These settings extend evaluation from local completion to sustained state coherence, progress tracking, and alignment over time.

5.3 Protocol Validity and Harness Effects

Evaluation scores reflect a coupled configuration of the model, task set, and protocol choices governing observation, execution, retries, memory, and runtime conditions [48, 79, 93, 165]. Three factors are especially important for interpreting comparisons across evaluation settings.

Contamination and temporal validity. Static task pools are vulnerable to memorization, prompt leakage, stale distributions, and public exposure of issues, patches, tests, or discussions. AgentBench [90] established broader agent-evaluation protocols, while mutation and rebenchmarking improve freshness [9, 10, 45]. LiveSQLBench [138] extends dynamic evaluation to database tasks, ACE-Bench [168] varies difficulty and horizon, and contamination detection distinguishes recall from reasoning [133]. Task construction is equally important: an audit reports that 16% of tasks across five terminal-agent benchmarks are reward-hackable [185]. These developments make temporal freshness, contamination control, task regeneration, and resistance to reward hacking central to protocol validity.

Harness-mediated variance. The harness is part of the evaluated configuration. Observation formatting, approval, sandboxing, retry limits, context truncation, wrappers, and recovery affordances can alter performance under a fixed model. Controlled comparisons across CLI and MCP interfaces make these effects explicit [42]; Meta-Harness [79] and AutoHarness [93] optimize the outer loop directly, while Agent Psychometrics [48] separates LLM and scaffold ability through item response theory. Controlled terminal evaluations further show that harness choice changes token efficiency and failure profiles under fixed models, while AgentMeter jointly evaluates task quality, budget sensitivity, and LM-CLI matching [25, 145]. Model comparisons therefore depend on the harness conditions under which behavior is produced.

Environment and budget comparability. Container images, dependency caches, network access, timeouts, filesystem persistence, and tool permissions alter both success and failure modes [7, 74, 101]. Production-derived tasks reduce distribution mismatch but may trade breadth and controlled conditions for realism [65]. Retry budgets, model and verifier calls, trajectory limits, and wall-clock timeouts further determine the search space available to an agent.

Together, these factors make the model, harness, observation and action interfaces, execution environment, permission policies, and execution budget part of the evaluated configuration [48, 79, 93, 165].

5.4 Evaluation Layers and Metrics Beyond Binary Correctness

Resolution and pass rates remain leaderboard anchors but capture only final correctness. Complementary evidence includes behavioral analysis beyond resolution rate [98], syntax-aware structure in SWE-PolyBench [119], checklist-based process scores in OctoBench [31], long-horizon drift and faithfulness [107, 163], and multidimensional enterprise assessment [19]. Relevant process indicators include *command economy*, relating command number and complexity to task scope; *recovery*, recording actions after failure; *diagnostic quality*, assessing whether causes are identified before repair; *state tracking*, checking consistency with filesystem, package, process, and repository state; and *governance violations*, capturing unsafe commands, permission bypass, sandbox escape, or unapproved external effects.

These signals form a layered evidence stack. Outcome evidence records task completion; process evidence describes execution; environment evidence establishes runtime validity; trace evidence supports inspection and replay; and governance evidence records permissions, containment, and

Table 3: Qualitative coding of terminal-competence observability and evaluation freshness. E=Explicit, T=Trace-visible, I=Incidental, and –=Not observable. Freshness concerns evaluation validity rather than competence.

Benchmark	Terminal-competence dimensions							Eval. validity
	Action form.	Feedback interp.	Runtime mgmt.	State tracking	Verify	Recover	Govern	Fresh.
SWE-bench / SWE-PolyBench [67, 119]	I	T	I	T	T	I	–	–
Claw-SWE-Bench [183]	I	T	I	T	T	I	–	I
Terminal-Bench / TerminalWorld [26, 101]	E	E	T	T	T	T	–	I
SetupBench [7]	E	E	E	T	T	T	–	–
LongCLI-Bench / GitTaskBench [39, 105]	E	E	I	E	T	I	–	–
CLI-Gym [89]	E	E	T	T	T	T	–	–
debug-gym / OctoBench [31, 174]	T	E	I	T	E	E	–	–
BashArena / ClawSafety [70, 152]	E	E	I	T	T	T	E	–
LiveSQLBench [138]	T	T	–	I	–	–	–	E
WildClawBench [34]	E	E	T	E	T	T	I	–
ClawForge [76]	E	E	T	E	T	I	–	–

side effects. Freshness is not a behavioral layer but a cross-cutting condition of evaluation validity. Deployment-oriented multi-signal evaluation follows the same motivation [44]. Outcome metrics are comparatively standardized, whereas the remaining layers use heterogeneous definitions and protocols [14, 16, 31, 57, 70, 152].

5.5 Benchmark Observability of Terminal Competence Dimensions

Table 2 characterizes benchmark design emphases, while Table 3 maps representative groups to the seven competence dimensions and treats freshness separately as evaluation validity. The coding indicates which constructs each group makes observable rather than the strength of their measurement. **Explicit** denotes a targeted or scored construct, **Trace-visible** a required but unscored construct, **Incidental** a construct that may arise without being targeted, and **Not observable** little basis for observation or scoring. The rows include core terminal-agent benchmarks and terminal-relevant boundary comparators; grouped rows reflect their shared dominant design intent.

The matrix shows broad observability of command formulation and feedback interpretation, especially in terminal-native and CLI-centered benchmarks. Runtime management is explicit in SetupBench [7] and trace-visible in Terminal-Bench [101], TerminalWorld [26], CLI-Gym [89], WildClawBench [34], and ClawForge [76]. Long-horizon and deployment-like settings expose state and context tracking, often indirectly through final outcomes. Debugging and process benchmarks provide stronger observability of verification and recovery [31, 174], whereas repository-level evaluations rarely isolate them. Governance is explicit mainly in BashArena [70] and ClawSafety [152]; freshness remains a separate validity property.

5.6 Remaining Evaluation Gaps

Four structural gaps remain. **Terminal-native workflow coverage is narrow.** Most benchmarks remain repository- or coding-centric, while alternatives are fragmented across operational domains. TerminalWorld broadens general terminal work [26]; ML-DevBench and MLE-bench target iterative model development [18, 109]; ELTBench, DAComp, and DSAgentBench cover data pipelines and end-to-end data-science workflows [68, 80, 118]; and ITBench and AIOpsLab introduce operational state [23, 64]. Scientific experimentation appears in ExpBench, Curie, and ScienceBoard [72, 73, 135], while security evaluation covers CTF, penetration testing, and inference optimization [4, 78, 92, 104]. End-to-end CLI tool generation also remains under-evaluated [58].

Process and trace standards are immature. Benchmarks increasingly record trajectories, but no common schema covers commands, observations, failures, retries, state changes, and human interventions. Trajectory analyses demonstrate the value of trace inspection [14, 16]; ProcBench adds step-level assessment [57]; and IDE-Bench extends evaluation to IDE settings [97]. Studies of failed agentic pull requests and CLI trajectories further motivate shared failure taxonomies [38, 181]. Interactive trajectory debugging remains disconnected from benchmark protocols [61].

Freshness mechanisms remain peripheral. SWE-rebench reports evidence consistent with contamination-related inflation on static tasks [9]. Mutation and detection methods are emerging [17, 45, 133], but rarely enter standard evaluation pipelines, while temporal-consistency mechanisms remain experimental [134].

Safety and governance remain separated from task success. Existing benchmarks isolate privileged terminal actions [70]; risky code generation and execution [8, 20, 28, 53]; productivity and computer-use agents [22, 40, 75, 82]; and attacks on long-running or sandboxed agents [96, 132, 171]. UnderSpecBench extends earlier evidence of governance blind spots under benign instructions by measuring wrong-target and over-scope actions in underspecified DevOps tasks, while Boundary-Bench evaluates coding agents under progressively hardened execution policies [27, 35, 66]. These protocols make authorization and containment observable, but governance evidence remains distributed across separate task-success and policy-focused settings.

Current evaluation suites distribute these requirements across separate protocols rather than jointly integrating workflow realism, environment setup, process quality, trace visibility, long-horizon persistence, freshness, safety, and governance [7, 9, 57, 70, 74, 152]. Evaluation results therefore characterize a coupled configuration defined by the model, benchmark, harness, runtime, sandbox, observation format, and execution budget, while trace-release policy determines how much process evidence remains available for analysis.

Architecture shapes what can be executed and recorded, acquisition determines which recorded behavior can be learned, and evaluation determines which behaviors become evidence. Section 6 uses bounded fixed-condition diagnostics to illustrate two consequences: benchmark-dependent process exposure and limits of component attribution.

6 Framework-Guided Diagnostics of Process Observability and Attribution Limits

The preceding sections connect terminal competence to system architecture, acquisition data, and evaluation protocols. This section makes two implications of that synthesis concrete through bounded fixed-condition diagnostics. First, under a fixed agent configuration, we examine which

process signals different benchmark families expose. Second, on matched repository-repair tasks, we examine how outcomes vary across systems and what these differences reveal about component attribution. Trace cases complement the aggregate results with trajectory-level mechanisms.

For the benchmark-exposure diagnostic, we fix mini-SWE-agent [165] with DeepSeek-V4-Flash [161] while varying four benchmark families. The matched diagnostic separately evaluates mini-SWE-agent [165], SWE-agent [165], and OpenHands [150] on identical task identifiers within Claw-SWE-Bench Lite [183] and SWE-bench Lite [67], using DeepSeek-V4-Flash and DeepSeek-V4-Pro [161]. Each task has one formal run per condition, and each cell represents a complete model, interface, harness, and runtime configuration.

6.1 Diagnostic Design and Measures

Both diagnostics retain benchmark-native outcomes and trace-level process evidence. Seven trace-derived indicators, denoted P1–P7, capture selected observable aspects associated with the competence dimensions introduced in Section 2. The identifiers provide concise cross-reference and are paired with semantic names throughout the analysis.

Four indicators are deterministic. The *rule-matched invocation-failure rate* (P1) divides failed commands whose stderr matches shell syntax, command-not-found, non-executable, path, permission, argument, or malformed-tool patterns by all normalized command events. The *environment-exit rate* (P3) divides non-zero exits on dependency, runtime, service, configuration, or environment-management commands by all detected environment-management commands. The *final-window verification rate* (P5) is the fraction of tasks with a detected verification action in the final five actions or final 20% of the trace, whichever window is larger. The *governance-review-trigger rate* (P7) divides command events matching irreversible, overprivileged, secret-handling, host or sandbox, or external-side-effect patterns by all normalized command events.

Three auxiliary indicators use rule-constrained LLM judgments. Rule-based extractors first identify candidate episodes concerning feedback use, state errors, or recovery, and DeepSeek-V4-Flash [161] evaluates each target within a bounded evidence window. P2 is the helpful-use rate among eligible episodes with usable feedback, P4 the consequential-error rate among episodes with decidable state evidence, and P6 the successful task-relevant recovery rate among episodes containing a recovery attempt. Uncertain or low-confidence labels are excluded, and task-level rates are macro-averaged over eligible evidence.

All formal runs use fixed decoding settings, task identifiers, and benchmark-native evaluators. Harness-specific prompting, context management, action interfaces, and orchestration remain part of each evaluated system. Complete model settings, execution budgets, extractor definitions, and auxiliary-judge procedures are reported in the Supplementary Material.

The benchmark-exposure profile contains 241 Terminal-Bench 2.1 tasks [101], 93 SetupBench tasks [7], 21 LongCLI-Bench tasks [39], and 640 BashArena tasks [70]. The matched system comparison contains all 80 official Claw-SWE-Bench Lite tasks and 300 SWE-bench Lite tasks.

6.2 Benchmark-Exposure Diagnostic

Table 4 reports benchmark-native outcomes and the seven trace-derived process indicators. Because the benchmarks use different task distributions and evaluators, outcomes are interpreted within each benchmark rather than as a shared competence scale. P1, P3, P5, and P7 are deterministic, whereas P2, P4, and P6 are judge-assisted task-level macro averages over eligible semantic evidence.

Table 4: Benchmark-exposure profile under mini-SWE-agent with DeepSeek-V4-Flash. Outcome is the official benchmark score. P1–P7 are bounded trace-derived process indicators. P1 and P3 record execution-friction signals; P2, P4, and P6 are auxiliary judge-assisted task-level rates; P5 records a detected final-window check; and P7 flags actions requiring contextual governance review.

Benchmark	Tasks	Outcome	P1 Rule inv. fail.	P2 Feedback	P3 Env. exit	P4 State err.	P5 Final verify	P6 Recovery	P7 Gov. trigger
Terminal-Bench 2.1 [101]	241	52.6%	0.0%	82.1%	12.5%	14.7%	29.5%	79.9%	0.5%
SetupBench [7]	93	59.1%	0.0%	80.2%	9.0%	19.5%	36.6%	85.5%	1.4%
LongCLI-Bench [39]	21	23.8%	0.0%	76.4%	6.8%	7.7%	23.8%	67.8%	4.5%
BashArena [70]	640	41.8%	0.0%	76.3%	13.7%	23.3%	66.2%	67.0%	3.0%

Three findings characterize the observed profile. **Rule-matched invocation failures are rare under the fixed configuration.** P1 rounds to 0.0% across all four benchmarks, indicating that few command failures match the specified shell syntax, command-not-found, non-executable, path, permission, argument, or malformed-tool patterns. Broader action-formulation problems instead appear through semantic or state-dependent behavior not captured by this narrow signal.

Local feedback use and end-to-end completion expose different aspects of behavior. The auxiliary helpful-feedback indicator (P2) ranges from 76.3% to 82.1%, showing that many eligible episodes contain a locally useful response to visible feedback. Benchmark-native outcomes additionally depend on runtime management, persistent state tracking, recovery, and verification. Detected final-window verification ranges from 23.8% to 36.6% for Terminal-Bench, SetupBench, and LongCLI-Bench, compared with 66.2% for BashArena, showing substantial differences in late-trace inspection across benchmark conditions.

Benchmark conditions foreground different process limitations. Under the fixed configuration, SetupBench foregrounds environment bootstrap, LongCLI-Bench combines low outcome with long-horizon interaction demands, and BashArena increases the visibility of state errors and governance-relevant actions. The auxiliary indicators show broadly high accepted-label coverage, although LongCLI-Bench has both the smallest task set and lower semantic coverage than the other benchmarks. Its P2, P4, and P6 estimates are therefore interpreted directionally.

6.3 Matched Complete-System Diagnostic

The second diagnostic focuses on outcome differences across systems rather than extending the P1–P7 profile. Table 5 reports matched execution snapshots on identical task identifiers within each benchmark.

SWE-agent has the highest resolved rate in all four benchmark–model blocks. Its largest gap to the lowest-performing system is 21.25 percentage points on Claw-SWE-Bench Lite, compared with at most 7.00 points on SWE-bench Lite. The relative ordering of OpenHands and mini-SWE-agent also reverses across benchmarks: OpenHands is higher on Claw-SWE-Bench Lite but lower on SWE-bench Lite. Benchmark choice therefore changes the observed separation among systems even though SWE-agent remains highest under all reported conditions.

Within a fixed system, the absolute Pro-minus-Flash difference never exceeds 2.50 points. Exact task-paired McNemar tests detect no directional model-variant difference after Holm adjustment, and all paired bootstrap intervals include zero. Under these conditions, the two model variants produce similar resolved rates across the evaluated systems. Pro has higher observed mean wall-clock

Table 5: Matched single-run results by benchmark and system. F/P denote DeepSeek-V4-Flash/Pro; entries report resolved rate, Pro-minus-Flash difference, and mean task time in seconds.

System	F	P	Δ	Time F/P
Claw-SWE-Bench Lite				
mini-SWE-agent	58.75%	56.25%	−2.50	476/476
SWE-agent [165]	**76.25%**	**77.50%**	+1.25	675/712
OpenHands [150]	68.75%	67.50%	−1.25	600/653
SWE-bench Lite				
mini-SWE-agent	55.67%	55.67%	0.00	294/389
SWE-agent [165]	**58.67%**	**58.33%**	−0.33	382/525
OpenHands [150]	51.67%	51.67%	0.00	501/504

time in five of six system–benchmark pairs; under mini-SWE-agent, it also uses more mean input tokens on both benchmarks without an outcome gain.

Task-paired cross-system tests further characterize these snapshots. Cochran's Q rejects equal resolved rates among the three systems in every benchmark–model block ($Q = 10.57$ to 16.00, $p < 0.006$). After Holm correction across 12 pairwise comparisons, SWE-agent differs from mini-SWE-agent on Claw-SWE-Bench Lite for both Flash ($p = 0.023$) and Pro ($p = 0.002$), and from OpenHands on SWE-bench Lite for both Flash ($p = 0.001$) and Pro ($p = 0.001$). The other corrected pairwise contrasts are not significant. These results characterize task-level differences within the recorded execution snapshots.

6.4 Illustrative Trace Cases

Aggregate rates alone do not show how these differences arise within trajectories. Two trace cases illustrate how benchmark demands and system behavior become visible at the execution level.

Benchmark-exposure case: `magsac-install`

Condition: Terminal-Bench 2.1 with mini-SWE-agent and DeepSeek-V4-Flash. **Outcome:** unresolved; the evaluator reports `No module named 'pymagsac'`. **Trace evidence:** repeated dependency and build attempts yield a 38.5% environment-command non-zero-exit rate. The trace ends before an evaluator-relevant import or installation-source check. **Interpretation:** local feedback leaves the installation loop unclosed, linking runtime management, verification, and recovery across the trajectory.

Together, the aggregate results and trace cases show that benchmark conditions change which process demands become visible, while system configuration changes realized trajectories and outcomes on matched tasks. These diagnostics connect the survey framework to observable execution behavior and motivate evaluation that combines task outcomes with process evidence and explicit system conditions.

Matched case: `rubocop__rubocop-13560`

Condition: Claw-SWE-Bench Lite with DeepSeek-V4-Flash. **Outcome:** SWE-agent resolves the task; mini-SWE-agent and OpenHands do not. **Trace evidence:** the issue requires ordinary lowercase `'nul'` arguments to remain valid. Mini-SWE-agent changes the implementation and an existing test despite this constraint. OpenHands exempts all method-call arguments, broadening the requested behavior. SWE-agent changes production logic while preserving the

required distinction and passes the official evaluator. **Interpretation:** the task exposes different patch scopes and outcomes across systems.

7 Challenges and Future Directions

The literature synthesis identifies four connected gaps: domain concentration limits evidence for transferable competence, fragmented process evaluation restricts behavioral interpretation, runtime governance remains weakly integrated with task evaluation, and system-level comparisons complicate component attribution. These gaps recur across the architecture, acquisition, and evaluation evidence reviewed in Sections 3–5 and are further illustrated by the process-observability and attribution diagnostics in Section 6, motivating four research priorities.

7.1 Cross-Domain Terminal Competence

Training and evaluation remain concentrated in software engineering [67, 110, 165]. Repositories provide structured executable verification, but cannot determine whether terminal behavior reflects transferable command and interaction competence or task-specific heuristics. Operations, data engineering, scientific workflows, cloud management, and cybersecurity introduce distinct runtime and governance demands [23, 68, 70, 72, 124, 130, 144], while TUA-Bench extends evaluation to routine digital activities and scientific and engineering workflows [21]. Cross-domain research should distinguish reusable substrate-level competence from domain-specific workflows and compare transfer across competence dimensions to identify which capabilities remain workload-specific.

7.2 Fresh, Replayable, Process-Level Evaluation

Final outcomes do not reveal diagnostic quality, state preservation, recovery, or unsafe intermediate behavior [31, 57, 74]. Live or regenerated tasks improve temporal validity, while replayable traces preserve commands, observations, state changes, interventions, and verifier calls [9, 14, 16]. Combining them would connect fresh task distributions with process evidence, distinguishing successful completion from the behavior that produces it. This also links evaluation to acquisition in Section 4, where retained failures, state transitions, and recovery trajectories determine what can be learned and inspected.

7.3 Runtime Governability and Safety

Terminal agents can modify filesystems, install packages, launch processes, access networks, and operate around credentials. Permission gates, sandboxes, approvals, and rollback constrain these effects, but governance remains distributed across separate safety and policy-focused protocols [5, 27, 66, 70, 78, 152]. Future evaluation should integrate the architectural controls identified in Section 3, including authorization, containment, reversibility, destructive-action prevention, and external side-effect control. Task success and governance should be assessed jointly so that effective execution reflects the constraints under which actions are taken.

7.4 Controlled Model and Harness Attribution

Measured performance can reflect the model, interface, runtime, context policy, retry budget, or verifier access [79, 93, 150, 165]. Controlled studies further show that harness and LM–CLI choices can alter quality, efficiency, and failure behavior [25, 145]. Separating model capability from surrounding system support therefore remains a central problem. The matched diagnostic in Section 6 illustrates that observed system differences can also vary across benchmarks. Factorial designs or portable protocols that vary the model, interface, harness, or runtime while preserving task identity and execution conditions [140] can better isolate which gains transfer across configurations and which arise from specific model–harness interactions.

8 Conclusion

We organize the study of terminal agents around terminal-mediated execution and its state-changing action–observation loop, using a seven-dimensional competence profile to connect system architecture, competence acquisition, and evaluation.

Three conclusions emerge. First, terminal-agent behavior is jointly shaped by the model, interface, harness, runtime, and environment, making outer-loop design a performance-shaping system component. Second, executable trajectories ground learning in feedback, verification, and recovery, yet current acquisition remains concentrated on successful repository workflows and underrepresents environment management, persistent state, recovery, and governance. Third, prevailing evaluations emphasize final outcomes and expose process quality unevenly, while measured performance depends on both benchmark conditions and system configuration. The bounded diagnostics further illustrate benchmark-dependent process exposure and limits of component attribution.

Progress therefore requires cross-domain acquisition and evaluation, fresh and replayable process evidence, governable runtimes, and controlled model–harness attribution. Evaluation should combine explicit system and runtime conditions with task outcomes, process evidence, and trace provenance.

References

[1] Sami Abuzakuk, Lucas Crijns, Anne-Marie Kermarrec, Rafael Pires, and Martijn de Vos. Riva: Leveraging llm agents for reliable configuration drift detection. In *Proceedings of the Sixth European Workshop on Machine Learning and Systems*, pages 499–509, 2026.

[2] Pavel Adamenko, Mikhail Ivanov, Aidar Valeev, Rodion Levichev, Pavel Zadorozhny, Ivan Lopatin, Dmitry Babaev, Alena Fenogenova, and Valentin Malykh. Swe-mera: A dynamic benchmark for agenticly evaluating large language models on software engineering tasks. *arXiv preprint arXiv:2507.11059*, 2025.

[3] Shyam Agarwal, Hao He, and Bogdan Vasilescu. Ai ides or autonomous agents? measuring the impact of coding agents on software development. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 857–862, 2026.

[4] Ali Al-Kaswan, Maksim Plotnikov, Maxim Hájek, Roland Vízner, Arie van Deursen, and Maliheh Izadi. Do agents dream of root shells? partial-credit evaluation of llm agents in capture the flag challenges. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, pages 349–357, 2026.

[5] Maksym Andriushchenko, Alexandra Souly, Mateusz Dziemian, Derek Duenas, Maxwell Lin, Justin Wang, Dan Hendrycks, Andy Zou, Zico Kolter, Matt Fredrikson, et al. Agentharm: A benchmark for measuring harmfulness of llm agents. In *International Conference on Learning Representations*, volume 2025, pages 79185–79220, 2025.

[6] Anthropic. Claude Code, 2025. URL https://code.claude.com/docs/en/overview. Agentic coding tool available through terminal and other development surfaces, with file editing, command execution, and development-tool integration.

[7] Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. Setupbench: Assessing software engineering agents' ability to bootstrap development environments. *arXiv preprint arXiv:2507.09063*, 2025.

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

[9] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

[10] Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026.

[11] Patrice Bechard, Orlando Marquez Ayala, Emily Chen, Jordan Skelton, Sagar Davasam, Srinivas Sunkara, Vikas Yadav, and Sai Rajeswar. Terminal agents suffice for enterprise automation. *arXiv preprint arXiv:2604.00073*, 2026.

[12] Nikita Benkovich and Vitalii Valkov. Agyn: A multi-agent system for team-based autonomous software engineering. *arXiv preprint arXiv:2602.01465*, 2026.

[13] Muhammad Bilal, Jon Crowcroft, Ruizhi Wang, Xiaolong Xu, and Schahram Dustdar. Large language models for agentic netops and aiops: Architectures, evaluation, and safety. *arXiv preprint arXiv:2605.12729*, 2026.

[14] Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2846–2857. IEEE, 2025.

[15] Nghi DQ Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026.

[16] Ira Ceka, Saurabh Pujar, Shyam Ramji, Luca Buratti, Gail Kaiser, and Baishakhi Ray. Understanding software engineering agents through the lens of traceability: An empirical study. *arXiv preprint arXiv:2506.08311*, 2025.

[17] Jianzhe Chai, Yu Zhe, and Jun Sakuma. When benchmarks leak: Inference-time decontamination for llms. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 44743–44760, 2026.

[18] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pages 50466–50494, 2025.

[19] Abhishek Chandwani and Ishan Gupta. Beyond binary correctness: Scaling evaluation of long-horizon agents on subjective enterprise tasks. *arXiv preprint arXiv:2603.22744*, 2026.

[20] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, et al. Securevibebench: Benchmarking secure vibe coding of ai agents via reconstructing vulnerability-introducing scenarios. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24144–24168, 2026.

[21] Shoufa Chen, Luyuan Wang, Xuan Yang, Zhiheng Liu, Yuren Cong, Yuanfeng Ji, Feiyan Zhou, Xiaohui Zhang, Fanny Yang, and Belinda Zeng. Tua-bench: A benchmark for general-purpose terminal-use agents. *arXiv preprint arXiv:2606.28480*, 2026.

[22] Tianyu Chen, Chujia Hu, Ge Gao, Dongrui Liu, Xia Hu, and Wenjie Wang. Lps-bench: Benchmarking safety awareness of computer-use agents in long-horizon planning under benign and adversarial scenarios. *arXiv preprint arXiv:2602.03255*, 2026.

[23] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *Proceedings of Machine Learning and Systems*, 7, 2025.

[24] Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan R Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. Stratus: A multi-agent system for autonomous reliability engineering of modern clouds. *Advances in Neural Information Processing Systems*, 38:50119–50165, 2026.

[25] Han Chi, Jiaxin Qi, Yan Cui, Baisheng Lai, and Jianqiang Huang. Matching matters: A fair quality-efficiency benchmark for command-line agents, 2026. URL https://arxiv.org/abs/2606.21140.

[26] Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T Barr, Mark Harman, Federica Sarro, et al. Terminalworld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026.

[27] Dotan Davidovich, Yair Amar, Hai Rozencwajg, and Or Hiltch. Permission denied: Policy-graded evaluation of coding agents in hardened environments. *arXiv preprint arXiv:2608.02670*, 2026.

[28] Ads Dawson, Rob Mulla, Nick Landers, and Shane Caldwell. Airtbench: Measuring autonomous ai red teaming capabilities in language models. *arXiv preprint arXiv:2506.14682*, 2025.

[29] Alexandre De Masi. Terminal is all you need: Design properties for human-ai agent collaboration. *arXiv preprint arXiv:2603.10664*, 2026.

[30] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[31] Deming Ding, Shichun Liu, Enhui Yang, Jiahang Lin, Ziying Chen, Shihan Dou, Honglin Guo, Weiyu Cheng, Pengyu Zhao, Chengjun Xiao, et al. Octobench: Benchmarking scaffold-aware instruction following in repository-grounded agentic coding. *arXiv preprint arXiv:2601.10343*, 2026.

[32] Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, et al. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025.

[33] Liang Ding. Agenther: Hindsight experience replay for llm agent trajectory relabeling. *arXiv preprint arXiv:2603.21357*, 2026.

[34] Shuangrui Ding, Xuanlang Dai, Long Xing, Shengyuan Ding, Ziyu Liu, Yang JingYi, Penghui Yang, Zhixiong Zhang, Xilin Wei, Xinyu Fang, et al. Wildclawbench: A benchmark for real-world, long-horizon agent evaluation. *arXiv preprint arXiv:2605.10912*, 2026.

[35] Xuwei Ding, Skylar Zhai, Linxin Song, Jiate Li, Taiwei Shi, Nicholas Meade, Siva Reddy, Jian Kang, and Jieyu Zhao. The blind spot of agent safety: How benign user instructions expose critical vulnerabilities in computer-use agents. *arXiv preprint arXiv:2604.10577*, 2026.

[36] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

[37] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025.

[38] Ramtin Ehsani, Sakshi Pathak, Shriya Rawal, Abdullah Al Mujahid, Mia Mohammad Imran, and Preetha Chatterjee. Where do ai coding agents fail? an empirical study of failed agentic pull requests in github. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 807–811, 2026.

[39] Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, et al. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 29952–29963, 2026.

[40] Yunhao Feng, Yifan Ding, Yingshui Tan, Xingjun Ma, Yige Li, Yutao Wu, Yifeng Gao, Kun Zhai, and Yanming Guo. Agenthazard: A benchmark for evaluating harmful behavior in computer-use agents. *arXiv preprint arXiv:2604.02947*, 2026.

[41] Hanna Foerster, Tom Blanchard, Kristina Nikolić, Ilia Shumailov, Cheng Zhang, Robert Mullins, Nicolas Papernot, Florian Tramèr, and Yiren Zhao. Camels can use computers too: System-level security for computer use agents. *arXiv preprint arXiv:2601.09923*, 2026.

[42] Marc Alier Forment, María José Casañ Guerrero, Francisco José García-Peñalvo, and Juanan Pereira. The scaffolding matters more than the interface: A controlled comparison of mcp and cli tool use across seven agent scaffoldings, five language models, and one software task, 2026. URL https://arxiv.org/abs/2608.08654.

[43] Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

[44] Yuxuan Gao, Megan Wang, and Yi Ling Yu. Agentpulse: A continuous multi-signal framework for evaluating ai agents in deployment. *arXiv preprint arXiv:2604.24038*, 2026.

[45] Spandan Garg, Benjamin Steenhoek, and Yufan Huang. Saving swe-bench: A benchmark mutation approach for realistic agent evaluation. *arXiv preprint arXiv:2510.08996*, 2025.

[46] Purna Sai Garigipati, Onur Ayan, Kishor Chandra Joshi, and Xueli An. Beyond state machines: Executing network procedures with agentic tool-calling sequences. *arXiv preprint arXiv:2605.02584*, 2026.

[47] Paul Gauthier. Aider: Ai pair programming in your terminal, 2025. URL https://github.com/Aider-AI/aider. Open-source terminal-native AI pair-programming tool for editing and managing codebases with LLMs.

[48] Chris Ge, Daria Kryvosheieva, Daniel Fried, Uzay Girit, and Kaivalya Hariharan. Agent psychometrics: Task-level performance prediction in agentic coding benchmarks. *arXiv preprint arXiv:2604.00594*, 2026.

[49] Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. *arXiv preprint arXiv:2603.21489*, 2026.

[50] Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

[51] Haochen Gong, Chenxiao Li, Rui Chang, and Wenbo Shen. Secure and efficient access control for computer-use agents via context space. *arXiv preprint arXiv:2509.22256*, 2025.

[52] Google. Gemini CLI, 2025. URL https://github.com/google-gemini/gemini-cli. Open-source terminal AI agent for Gemini models with file operations, shell commands, web tools, and MCP integration.

[53] Chengquan Guo, Xun Liu, Chulin Xie, Andy Zhou, Yi Zeng, Zinan Lin, Dawn Song, and Bo Li. Redcode: Risky code execution and generation benchmark for code agents. *Advances in Neural Information Processing Systems*, 37:106190–106236, 2024.

[54] Dongxin Guo, Jikun Wu, and Siu Ming Yiu. Agenteval: Dag-structured step-level evaluation for agentic workflows with error propagation tracking. *arXiv preprint arXiv:2604.23581*, 2026.

[55] Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*, 2025.

[56] Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. Swe-skills-bench: Do agent skills actually help in real-world software engineering? *arXiv preprint arXiv:2603.15401*, 2026.

[57] Jiawei He, Jie Jia, Chenbo Liu, Chaoyi Xue, Yapeng Song, Xikai Yang, and Dong Sun. Procbench: Evaluating process-level defects and control preservation in llm coding agents. *arXiv preprint arXiv:2605.20251*, 2026.

[58] Ruida Hu, Xinchen Wang, Chao Peng, Cuiyun Gao, and David Lo. Evaluating llm-based 0-to-1 software generation in end-to-end cli tool scenarios. *arXiv preprint arXiv:2604.06742*, 2026.

[59] Xueyu Hu, Tao Xiong, Biao Yi, Zishu Wei, Ruixuan Xiao, Yurun Chen, Jiasheng Ye, Meiling Tao, Xiangxin Zhou, Ziyu Zhao, et al. Os agents: A survey on mllm-based agents for general computing devices use. *arXiv preprint arXiv:2508.04482*, 2025.

[60] Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, et al. Cli-universe: Towards verifiable task synthesis engine for terminal agents. *arXiv preprint arXiv:2606.22883*, 2026.

[61] Robert Hutter and Michael Pradel. Agentstepper: Interactive debugging of software development agents. *arXiv preprint arXiv:2602.06593*, 2026.

[62] Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

[63] Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights swe agents. In *Second Conference on Language Modeling*, 2025.

[64] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, et al. Itbench: Evaluating ai agents across diverse real-world it automation tasks. *arXiv preprint arXiv:2502.05352*, 2025.

[65] Smriti Jha, Matteo Paltenghi, Chandra Maddila, Vijayaraghavan Murali, Shubham Ugare, and Satish Chandra. Reap: Automatic curation of coding agent benchmarks from interactive production usage. *arXiv preprint arXiv:2604.01527*, 2026.

[66] Zimo Ji, Zekai Zhang, Congying Xu, Zongjie Li, Yudong Gao, Shuai Wang, and Shing-Chi Cheung. Coding agents are guessing: Measuring action-boundary violations in underspecified devops instructions. *arXiv preprint arXiv:2607.02294*, 2026.

[67] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

[68] Tengjun Jin, Yuxuan Zhu, and Daniel Kang. Elt-bench: An end-to-end benchmark for evaluating ai agents on elt pipelines. *Proceedings of the VLDB Endowment*, 19(2):84–98, 2025.

[69] Hangoo Kang, Tarun Suresh, Jon Saad-Falcon, and Azalia Mirhoseini. Trace: Capability-targeted agentic training. *arXiv preprint arXiv:2604.05336*, 2026.

[70] Adam Kaufman, James Lucassen, Tyler Tracy, Cody Rushing, and Aryan Bhatt. Basharena: A control setting for highly privileged ai agents. *arXiv preprint arXiv:2512.15688*, 2025.

[71] Gyeongwon James Kim, Alex Wilf, Louis-Philippe Morency, and Daniel Fried. From reproduction to replication: Evaluating research agents with progressive code masking. *arXiv preprint arXiv:2506.19724*, 2025.

[72] Patrick Tser Jern Kon, Jiachen Liu, Qiuyi Ding, Yiming Qiu, Zhenning Yang, Yibo Huang, Jayanth Srinivasa, Myungjin Lee, Mosharaf Chowdhury, and Ang Chen. Curie: Toward rigorous and automated scientific experimentation with ai agents. *arXiv preprint arXiv:2502.16069*, 2025.

[73] Patrick Tser Jern Kon, Jiachen Liu, Xinyi Zhu, Qiuyi Ding, Jingjia Peng, Jiarong Xing, Yibo Huang, Yiming Qiu, Jayanth Srinivasa, Myungjin Lee, et al. Exp-bench: Can ai conduct ai research experiments? *arXiv preprint arXiv:2505.24785*, 2025.

[74] Jiayi Kuang, Yinghui Li, Xin Zhang, Yangning Li, Xing Sun, Ying Shen, Philip Yu, et al. Process-level trajectory evaluation for environment configuration in software engineering agents. In *International Conference on Learning Representations*, volume 2026, pages 113832–113855, 2026.

[75] Thomas Kuntz, Agatha Duzan, Hao Zhao, Francesco Croce, Zico Kolter, Nicolas Flammarion, and Maksym Andriushchenko. Os-harm: A benchmark for measuring safety of computer use agents. *Advances in Neural Information Processing Systems*, 38, 2026.

[76] Yuxiang Lai, Peng Xia, Haonian Ji, Kaiwen Xiong, Kaide Zeng, Jiaqi Liu, Fang Wu, Jike Zhong, Zeyu Zheng, Cihang Xie, et al. Clawforge: Generating executable interactive benchmarks for command-line agents. *arXiv preprint arXiv:2605.14133*, 2026.

[77] Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1496–1524, 2025.

[78] Dongjun Lee, Ga-eun Bae, and Insu Yun. Ctfusion: A ctf-based benchmark for llm agent evaluation. *arXiv preprint arXiv:2605.11504*, 2026.

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

[80] Fangyu Lei, Jinxiang Meng, Yiming Huang, Junjie Zhao, Yitong Zhang, Jianwen Luo, Xin Zou, Ruiyi Yang, Wenbo Shi, Yan Gao, et al. Dacomp: Benchmarking data agents across the full data intelligence lifecycle. *arXiv preprint arXiv:2512.04324*, 2025.

[81] Keyu Li, Junhao Shi, Yang Xiao, Mohan Jiang, Jie Sun, Yunze Wu, Dayuan Fu, Shijie Xia, Xiaojie Cai, Tianze Xu, et al. Agencybench: Benchmarking the frontiers of autonomous agents in 1m-token real-world contexts. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7422–7440, 2026.

[82] Xiangyi Li, Kyoung Whan Choe, Yimin Liu, Xiaokun Chen, Chujun Tao, Bingran You, Wenbo Chen, Zonglin Di, Jiankai Sun, Shenghan Zheng, et al. Clawsbench: Evaluating capability and safety of llm productivity agents in simulated workspaces. *arXiv preprint arXiv:2604.05172*, 2026.

[83] Xuefeng Li, Nir Ben-Israel, Yotam Raz, Belal Ahmed, Doron Serebro, and Antoine Raux. Repomod-bench: A benchmark for code repository modernization via implementation-agnostic testing. *arXiv preprint arXiv:2602.22518*, 2026.

[84] Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, et al. Recursive synthesis for long-horizon terminal tasks. *arXiv preprint arXiv:2608.05466*, 2026.

[85] Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, et al. Long-horizon-terminal-bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026.

[86] Jiarong Liang, Zhiheng Lyu, Zijie Liu, Xiangchao Chen, Ping Nie, Kai Zou, and Wenhu Chen. Swe-next: Scalable real-world software engineering tasks for agents. *arXiv preprint arXiv:2603.20691*, 2026.

[87] Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

[88] Xihuan Lin, Jie Zhang, Gelei Deng, Tianzhe Liu, Tianwei Zhang, Qing Guo, and Riqing Chen. Ircopilot: Automated incident response with large language models. *arXiv preprint arXiv:2505.20945*, 2025.

[89] Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. Cli-gym: Scalable cli task generation via agentic environment inversion. *arXiv preprint arXiv:2602.10999*, 2026.

[90] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046, 2024.

[91] Yang Liu, Li Zhang, Fang Liu, Ping Lin, and Xinyi Li. A scalable benchmark for repository-oriented long-horizon conversational context management. *arXiv preprint arXiv:2603.06358*, 2026.

[92] Zicheng Liu, Lige Huang, Jie Zhang, Dongrui Liu, Yuan Tian, and Jing Shao. Pacebench: A framework for evaluating practical ai cyber-exploitation capabilities. *arXiv preprint arXiv:2510.11688*, 2025.

[93] Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

[94] Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183, 2025.

[95] Pengrui Lu, Shiqi Zhang, Yunzhong Hou, Lyumanshan Ye, Chaoyi Huang, Zixi Chen, Ji Zeng, Hantao Jiang, Pengfei Liu, Yiwei Wang, et al. Projdevbench: Benchmarking ai coding agents on end-to-end project development. *arXiv preprint arXiv:2602.01655*, 2026.

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

[97] Spencer Mateega, Jeff Yang, Tiana Costello, Shaurya Jadhav, Nicole Tian, and Agustin Garcinuño. Ide-bench: Evaluating large language models as ide agents on real-world software engineering tasks. *arXiv preprint arXiv:2601.20886*, 2026.

[98] Tural Mehtiyev and Wesley Assunção. Beyond resolution rates: Behavioral drivers of coding agent success and failure. *arXiv preprint arXiv:2604.02547*, 2026.

[99] Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971*, 2024.

[100] Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. Calibforge: Adversarial solver calibration for scaling learnable terminal tasks. *arXiv preprint arXiv:2608.06352*, 2026.

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

[102] Ryo Nakamura and Koshi Eguchi. How helpful is llm assistance in network operations? a case study at a large demonstration network. *arXiv preprint arXiv:2605.19627*, 2026.

[103] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Arik, and Tomas Pfister. Mle-star: Machine learning engineering agent via search and targeted refinement. *Advances in Neural Information Processing Systems*, 38:116692–116712, 2026.

[104] Ayush Nangia, Shikhar Mishra, Aman Gokrani, and Paras Chopra. Iso-bench: Can coding agents optimize real-world inference workloads? *arXiv preprint arXiv:2602.19594*, 2026.

[105] Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Zhenheng Tang, Sen Hu, Bo Li, Chen Hu, Binxing Jiao, et al. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32564–32572, 2026.

[106] OpenAI. Codex CLI, 2025. URL https://github.com/openai/codex. Open-source terminal coding agent that runs locally and supports command-line coding workflows.

[107] Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks. *arXiv preprint arXiv:2603.24755*, 2026.

[108] Anvith Pabba, Alex Mathai, Anindya Chakraborty, and Baishakhi Ray. Semagent: A semantics aware program repair agent. *arXiv preprint arXiv:2506.16650*, 2025.

[109] Harshith Padigela, Chintan Shah, and Dinkar Juyal. Ml-dev-bench: Comparative analysis of ai agents on ml development workflows. *arXiv preprint arXiv:2502.00964*, 2025.

[110] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

[111] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[112] Kannan Parthasarathy, Karthik Vaidhyanathan, Rudra Dhar, Venkat Krishnamachari, Adyansh Kakran, Sreemaee Akshathala, Shrikara Arun, Amey Karan, Basil Muhammed, Sumant Dubey, et al. Engineering llm powered multi-agent framework for autonomous cloudops. In *2025 IEEE/ACM 4th International Conference on AI Engineering–Software Engineering for AI (CAIN)*, pages 201–211. IEEE, 2025.

[113] Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

[114] Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling llm terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026.

[115] Giovanni Pinna, Jingzhi Gong, David Williams, and Federica Sarro. Comparing ai coding agents: A task-stratified analysis of pull request acceptance. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, pages 792–796, 2026.

[116] Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar VK, Rongzhi Zhang, Changhao Li, Ian Wong, Sherry Yang, Percy Liang, Chao Zhang, et al. Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering. *Advances in Neural Information Processing Systems*, 38, 2026.

[117] Ella Rabinovich and Ateret Anaby Tavor. On the robustness of agentic function calling. In *Proceedings of the 5th Workshop on Trustworthy NLP (TrustNLP 2025)*, pages 298–304, 2025.

[118] Mizanur Rahman, Mohammed Saidul Islam, Ridwan Mahbub, Md Tahmid Rahman Laskar, Shafiq Joty, and Enamul Hoque Prince. Dsagentbench: Can agents automate end-to-end data-science workflows in real computer environments?, 2026. URL https://arxiv.org/abs/2608.10366.

[119] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.

[120] Chris Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. In *International Conference on Learning Representations*, volume 2025, pages 406–441, 2025.

[121] Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, et al. A self-evolving framework for efficient terminal agents via observational context compression. *arXiv preprint arXiv:2604.19572*, 2026.

[122] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. Agentic much? adoption of coding agents on github. *ACM Transactions on Software Engineering and Methodology*, 2026.

[123] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36: 68539–68551, 2023.

[124] Mohsen Seyedkazemi Ardebili and Andrea Bartolini. Kubeintellect: A modular llm-orchestrated agent framework for end-to-end kubernetes management: Ms ardebili, a. bartolini. *Journal of Grid Computing*, 24(3):17, 2026.

[125] Jianshu She. Agentrm: An os-inspired resource manager for llm agent systems. *arXiv preprint arXiv:2603.13110*, 2026.

[126] Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, et al. Seta: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

[127] Yucheng Shi, Wenhao Yu, Jingyuan Huang, Wenlin Yao, Wenhu Chen, and Ninghao Liu. Towards trustworthy gui agents: A survey. *arXiv preprint arXiv:2503.23434*, 2025.

[128] Vaishnavi Shrivastava, Piero Kauffmann, Ahmed Awadallah, and Dimitris Papailiopoulos. Echo: Terminal agents learn world models for free. *arXiv preprint arXiv:2605.24517*, 2026.

[129] Mohammed Latif Siddiq, Xinye Zhao, Vinicius Carvalho Lopes, Beatrice Casey, and Joanna Santos. Security in the age of ai teammates: An empirical study of agentic pull requests on github. *arXiv preprint arXiv:2601.00477*, 2026.

[130] Rohan Siva, Kai Cheung, Lichi Li, and Ganesh Sundaram. kraig: A natural language-driven agent for automated dataops pipeline generation. *arXiv preprint arXiv:2603.20311*, 2026.

[131] Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, et al. Swe-master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411*, 2026.

[132] Kefan Song and Yanjun Qi. Anchor: Automated alignment auditing for cli agents on real-world harm. *arXiv preprint arXiv:2607.10455*, 2026.

[133] Tae-Eun Song. Cross-context verification: Hierarchical detection of benchmark contamination through session-isolated analysis. *arXiv preprint arXiv:2603.21454*, 2026.

[134] Haonan Sun, Tian Yu, Sheng Ma, Qincheng Zhang, Lifei Rao, Chen Tian, et al. Atime-consistent benchmark for repository-level software engineering evaluation. *arXiv preprint arXiv:2603.26137*, 2026.

[135] Qiushi Sun, Zhoumianze Liu, Chang Ma, Zichen Ding, Fangzhi Xu, Zhangyue Yin, Haiteng Zhao, Zhenyu Wu, Kanzhi Cheng, Zhaoyang Liu, et al. Scienceboard: Evaluating multimodal autonomous agents in realistic scientific workflows. In *International Conference on Learning Representations*, volume 2026, pages 75694–75731, 2026.

[136] Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

[137] Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, et al. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *Advances in Neural Information Processing Systems*, 38:15869–15909, 2026.

[138] BIRD Team et al. Livesqlbench: A dynamic and contamination-free benchmark for evaluating llms on real-world text-to-sql tasks, 2024.

[139] Minh VT Thai, Tue Le, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

[140] Kishanthan Thangarajah, Boyuan Chen, and Ahmed E Hassan. Dcas: Decoupling cli agent scaffolding to internalize planning across scaffolds. *arXiv preprint arXiv:2608.06113*, 2026.

[141] TheR1D. ShellGPT, 2026. URL https://github.com/TheR1D/shell_gpt. Command-line productivity tool powered by large language models for generating shell commands, code snippets, and documentation.

[142] Matthew Thompson. The dual-state architecture for reliable llm agents, 2026. URL https://arxiv.org/abs/2512.20660.

[143] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, 2024.

[144] Ahmed Twabi, Yepeng Ding, and Tohru Kondo. Netagentbench: A state-centric benchmark for evaluating agentic network configuration. *arXiv preprint arXiv:2604.09678*, 2026.

[145] Naman Vats and Oleg Golev. The scaffold effect in coding agents: Harness choice as a hidden variable in coding-agent evaluation. *arXiv preprint arXiv:2607.22585*, 2026.

[146] Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025.

[147] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[148] Renxi Wang, Rifo Ahmad Genadi, Bilal El Bouardi, Yongxin Wang, Fajri Koto, Zhengzhong Liu, Timothy Baldwin, and Haonan Li. Agentfly: Extensible and scalable reinforcement learning for lm agents. *arXiv preprint arXiv:2507.14897*, 2025.

[149] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024.

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

[151] Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. Agents in software engineering: Survey, landscape, and vision. *Automated Software Engineering*, 32(2):70, 2025.

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

[153] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

[154] Junde Wu, Minhao Hu, Jiayuan Zhu, Jiazhen Pan, Yuyuan Liu, Min Xu, and Yueming Jin. Git context controller: Manage the context of llm-based agents like git. *arXiv preprint arXiv:2508.00031*, 2025.

[155] Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, et al. Large-scale terminal agentic trajectory generation from dockerized environments. *arXiv preprint arXiv:2602.01244*, 2026.

[156] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[157] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

[158] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

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

[160] Zidi Xiu, David Q Sun, Kevin Cheng, Maitrik Patel, Yizhe Zhang, Jiarui Lu, Omar Attia, Raviteja Vemulapalli, Oncel Tuzel, Meng Cao, et al. Astra-bench: Evaluating tool-use agent reasoning and action planning with personal user context. *arXiv preprint arXiv:2603.01357*, 2026.

[161] Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

[162] Tianshi Xu, Yuteng Chen, and Meng Li. Cleaner: Self-purified trajectories boost agentic reinforcement learning. *arXiv preprint arXiv:2601.15141*, 2026.

[163] Lu Yan, Xuan Chen, and Xiangyu Zhang. When the specification emerges: Benchmarking faithfulness loss in long-horizon coding agents. *arXiv preprint arXiv:2603.17104*, 2026.

[164] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

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

[166] John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

[167] Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, et al. What makes interaction trajectories effective for training terminal agents? *arXiv preprint arXiv:2606.03461*, 2026.

[168] Wang Yang, Chaoda Song, Xinpeng Li, Debargha Ganguly, Chuang Ma, Shouren Wang, Zhihao Dou, Yuli Zhou, Vipin Chaudhary, and Xiaotian Han. Ace-bench: Agent configurable evaluation with scalable horizons and controllable difficulty under lightweight environments. *arXiv e-prints*, pages arXiv–2604, 2026.

[169] Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, et al. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*, 2025.

[170] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[171] Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, et al. Claw-eval: Towards trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026.

[172] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents. *arXiv preprint arXiv:2503.16416*, 2025.

[173] Zhaoshu Yu, Bo Wang, Pengpeng Zeng, Haonan Zhang, Ji Zhang, Zheng Wang, Lianli Gao, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on efficient vision-language-action models. *arXiv preprint arXiv:2510.24795*, 2025.

[174] Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, et al. debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*, 2025.

[175] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

[176] Ji Zeng, Dayuan Fu, Tiantian Mi, Yumin Zhuang, Yaxing Huang, Xuefeng Li, Lyumanshan Ye, Muhang Xie, Qishuo Hua, Zhen Huang, et al. davinci-dev: Agent-native mid-training for software engineering. *arXiv preprint arXiv:2601.18418*, 2026.

[177] Yucheng Zeng, Shupeng Li, Daxiang Dong, Ruijie Xu, Zimo Chen, Liwei Zheng, Yuxuan Li, Zhe Zhou, Haotian Zhao, Lun Tian, et al. Swe-hub: A unified production system for scalable, executable software engineering tasks. *arXiv preprint arXiv:2603.00575*, 2026.

[178] Boxuan Zhang, Jianing Zhu, Zeru Shi, Dongfang Liu, and Ruixiang Tang. Agentforesight: Online auditing for early failure prediction in multi-agent systems. *arXiv preprint arXiv:2605.08715*, 2026.

[179] Jiaran Zhang, Luck Ma, Yanhao Li, Fanqi Wan, Di Qi, Xu Zhao, Jieyi Hou, Zhe Xie, Mengqiang Ren, Xin Wu, et al. Docksmith: Scaling reliable coding environments via an agentic docker builder. *arXiv preprint arXiv:2602.00592*, 2026.

[180] Chenyu Zhao, Shenglin Zhang, Yihang Lin, Wenwei Gu, Zhimin Chen, Yongqian Sun, Dan Pei, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. Debugging the debuggers: Failure-anchored structured recovery for software engineering agents. *arXiv preprint arXiv:2605.08717*, 2026.

[181] Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of cli coding agent trajectories. *arXiv preprint arXiv:2607.09510*, 2026.

[182] Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

[183] Mengyu Zheng, Kai Han, Boxun Li, Haiyang Xu, Yuchuan Tian, Wei He, Hang Zhou, Jianyuan Guo, Hailin Hu, Lin Ma, et al. Claw-swe-bench: A benchmark for evaluating openclaw-style agent harnesses on coding tasks. *arXiv preprint arXiv:2606.12344*, 2026.

[184] Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. Towards agentic os: An llm agent framework for linux schedulers. *arXiv preprint arXiv:2509.01245*, 2025.

[185] Ziqian Zhong, Ivgeni Segal, Ivan Bercovich, Shashwat Saxena, Kexun Zhang, and Aditi Raghunathan. Hardening agent benchmarks with adversarial hacker-fixer loops. *arXiv preprint arXiv:2606.08960*, 2026.

[186] Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.

[187] Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, et al. Memento: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025.

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

[189] Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

[190] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

[191] Haomin Zhuang, Hanwen Xing, and Xiangliang Zhang. Agentclick: A skill-based human-in-the-loop review layer for terminal ai agents. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pages 1372–1378, 2026.

APPENDIX

A Review Protocol and Corpus Construction

This supplementary document reports the review protocol, extended taxonomy and coding materials, empirical methods and results, and representative trace cases. We conducted a **structured narrative review with an evidence-stratified coded corpus**. The review combines explicit workload-level scope rules, entry-level analytical coding, and passage-level evidence calibration to support a traceable synthesis of the reconciled corpus.

A.1 Corpus Scope and Composition

The review covers work released between 2022 and August 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 current manuscript cites **191 distinct entries**: **186 research entries** in the review corpus, four deployment-facing tools (Claude Code, Codex CLI, Aider, and Gemini CLI), one CLI-packaged assistant boundary comparator. The latter six entries remain outside research statistics. Candidate versions are consolidated at the work level so that the corpus counts a research contribution rather than each of its releases.

Table 6 summarizes corpus status, release year, and analytical use. Analytical-use counts are non-exclusive because one work may support several parts of the survey. This view separates entry-level corpus composition from subset-specific architecture, acquisition, and evaluation coding and from passage-level evidence calibration.

A.2 Search, Screening, and Work-Level Reconciliation

We assembled the corpus through iterative keyword search, venue-oriented search, and backward and forward reference chaining, with the final update performed in August 2026. Sources included arXiv, OpenReview, DBLP, IEEE Xplore, the ACM Digital Library, the ACL Anthology, major machine-learning and software-engineering venues, systems-oriented venues, benchmark repositories, and project pages for deployment-facing terminal-agent tools. Candidate versions were consolidated at the work level, with the most complete version retained and earlier versions linked to the same contribution rather than counted separately.

Table 7 summarizes the construction sequence and its analytical role. The sequence also provides the protocol used for subsequent corpus updates.

Table 6: Composition of the current review corpus. Release-year counts cover the 186 research entries; analytical-use counts are non-exclusive.

Category	Description	Count
Corpus status		
Research corpus	Research works retained for the survey synthesis.	186
Engineering practice, boundary, and disclosure references	Five engineering-practice or boundary entries and one disclosure citation.	6
Total cited set	Distinct cited entries after work-level version consolidation.	192
Release year of research entries		
2022–2023	Earliest retained foundations for tool use and interactive execution.	4
2024	Expansion of executable agent and repository-repair research.	14
2025	Broadening benchmark, system, and deployment evidence.	56
2026 through August	Recent architecture, acquisition, evaluation, and cross-domain work.	112
Analytical use of research entries, non-exclusive		
Scope and characterization	Supports the execution-substrate boundary and competence profile.	19
System architecture	Supports layers, families, control, memory, and runtime analysis.	44
Competence acquisition	Supports environments, trajectories, supervision, and adaptation analysis.	41
Evaluation	Supports benchmark families, metrics, validity, and competence observability.	99
Empirical setting	Supports the fixed-condition diagnostic design and its task settings.	9
Challenges and implications	Supports domain transfer, process evidence, governance, and attribution.	28

For subsequent updates, we specify seven Boolean query families in Table 8. Source interfaces may require field-name or quotation-mark changes, while the Boolean concepts and date limits remain fixed. Each new source-specific execution records its date, query form, and hit count before work-level reconciliation.

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 execution or terminal-agent systems?
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 retained entry received a corpus status and one or more analytical roles. Research entries support the architecture, acquisition, evaluation, empirical, or implication synthesis; engineering-practice and boundary references illustrate deployed patterns or delimit adjacent systems. The reported corpus counts begin after work-level reconciliation because the iterative initial retrieval did not preserve a complete query-level candidate count before deduplication.

Table 7: Corpus-construction workflow. Each stage produces the input required by the next stage and constrains how evidence enters the synthesis.

Stage	Procedure	Recorded decision	Role in synthesis
Scope freeze	Fix the 2022 to August 2026 interval, terminal-mediated research object, adjacent comparators, and workload-level boundary tests	Date range, source classes, and inclusion boundary	Prevents surface-level CLI mentions from entering as core evidence
Retrieval	Apply keyword families, venue-oriented search, and backward and forward reference chaining	Source, execution date, query form when retained, and work identity	Combines named-field retrieval with architecture, acquisition, evaluation, and governance routes
Work reconciliation	Group versions of the same contribution and retain the most complete research account	Preferred version and version relationship	Avoids double counting preprint, conference, and journal releases
Screening	Apply the five questions below and assign a corpus disposition	Research, engineering-practice, boundary, or exclusion disposition	Separates progress-bearing terminal evidence from surface-level or adjacent evidence
Analytical coding	Assign manuscript roles to every retained entry and apply architecture, acquisition, or evaluation fields to the relevant subsets	Entry-level roles and subset-specific analytical codes	Connects retained evidence to the architecture, acquisition, and evaluation synthesis
Claim calibration	Assess convergence, directness, domain breadth, and evidence maturity at the passage level	Bounded claim strength and scope	Keeps synthesis claims proportional to the available evidence
Update check	Reapply retrieval, reconcile new versions, repeat screening, and revise affected synthesis passages	Added, consolidated, retained, or removed status	Keeps later corpus updates aligned with the same decision sequence

Table 8: Boolean query families used for subsequent corpus updates. Each source-specific execution is logged with its date and hit count.

ID	Purpose	Boolean template
Q1	Named field	`("terminal agent" OR "terminal agents" OR "CLI agent" OR "command-line agent")`
Q2	Execution loop	`("LLM agent" OR "language model agent") AND (terminal OR shell OR CLI OR "command line") AND (execution OR runtime OR feedback)`
Q3	SWE and harnesses	`("software engineering agent" OR "coding agent") AND (terminal OR shell OR command) AND (benchmark OR harness OR environment)`
Q4	Acquisition	`("agent trajectory" OR "executable trajectory" OR "environment interaction") AND (training OR post-training OR "reinforcement learning") AND (terminal OR CLI OR repository)`
Q5	Process evaluation	`("terminal benchmark" OR "CLI benchmark" OR "agent benchmark") AND (trace OR process OR recovery OR verification OR state)`
Q6	Governance	`("LLM agent" OR "coding agent" OR "computer-use agent") AND (shell OR terminal OR "code execution") AND (sandbox OR permission OR safety OR governance)`
Q7	Cross-domain use	`("LLM agent" OR "AI agent") AND (terminal OR CLI OR "command execution") AND (AIOps OR CloudOps OR DataOps OR cybersecurity OR "scientific workflow")`

A.3 Corpus Reconciliation and Update Protocol

The retained corpus can be inspected through the screening questions, work-level reconciliation rules, analytical roles, and level-specific fields in Table 9. Subsequent updates additionally use the Boolean query families in Table 8. Retrieval results are reconciled at the work level before assignment of

corpus disposition and analytical roles. Applying these rules yields the 186-entry research corpus summarized in Table 6; the six non-corpus references remain outside research-entry statistics.

During an update, a work is added when it satisfies the screening questions and contributes evidence to at least one analytical role. A new version replaces an earlier version when it provides the more complete account of the same contribution. Versions representing the same contribution are consolidated rather than counted separately. A retained work is removed from the review corpus when it no longer supports a synthesis passage after manuscript revision. These rules keep corpus composition, analytical coding, and manuscript claims synchronized.

A.4 Coding Dimensions and Evidence Calibration

Entry-level coding covers bibliographic metadata, research or engineering-practice status, and manuscript analytical roles. These fields support the corpus counts, release-year distribution, and non-exclusive analytical-use counts in Table 6. Additional conceptual fields are applied only to evidence subsets for which they are relevant, using the paper as the default unit and the system, benchmark, dataset, or pipeline when one paper introduces several distinct artifacts.

For the main paper's architecture analysis, relevant entries were coded by interface granularity, operational coupling, autonomy regime, recovery and control style, and planning or control strategy. For the acquisition analysis, entries were coded by data-source type, trajectory source, supervision signal, filtering or relabeling method, failure-data treatment, and capability target. For the evaluation synthesis and diagnostic study, entries were coded by task regime, harness assumptions, verification type, trace visibility, artifact availability, and coverage of the seven terminal-competence dimensions.

We calibrate evidence at the synthesis-passage level rather than assigning a single maturity label to an entire paper. Converging evidence across multiple benchmarks, systems, or domains supports stronger synthesis claims; direct but limited empirical evidence supports bounded claims; single-study or indirect evidence supports provisional claims; and deployment practice or boundary examples support illustration. Publication status and artifact availability remain descriptors of the relevant evidence subset rather than substitutes for claim-level calibration.

A.5 Terminology and System-Layer Roles

The working terminology used for the survey's workload-level boundary tests is organized as follows. Each term is paired with its technical meaning and its analytical role.

Terminal. A textual input/output access layer or its emulation. It is an established field label and a common access path to command execution, not itself the source of process state or exit semantics.

Shell. A command interpreter that parses commands, expands syntax, and starts programs. It is a common but non-required mediator between an agent interface and executable programs.

CLI tool. A program whose operations are invoked through textual arguments or commands. It is an action target; surface-level CLI exposure alone does not place a workload in scope.

Command-execution runtime. The filesystem, processes, dependencies, permissions, resources, and operating-system state in which commands take effect. It is the state-changing execution substrate that grounds actions, observations, and later verification.

Harness. The model-facing layer that exposes actions, formats observations, manages context,

Table 9: Level-specific coding schema used in corpus construction and synthesis. Entry-level fields cover every retained reference; specialized fields apply to relevant evidence subsets or synthesis passages.

Field group	Coded fields	Unit	Analytical use
Corpus identity	Bibliographic metadata, corpus status, and unique work identity	Work	Supports corpus identity, release-year counts, and work-level deduplication
Analytical role	Scope, architecture, acquisition, evaluation, empirical setting, or implications	Work, non-exclusive	Connects retained entries to the survey passages they support
Architecture	Interface granularity, operational coupling, autonomy regime, recovery and control style, planning strategy	System or artifact	Supports the layered architecture and recurring-pattern synthesis
Acquisition	Data source, trajectory source, supervision signal, filtering or relabeling, failure-data treatment, capability target	Dataset, environment, or pipeline	Supports comparison of executable data, learning, and adaptation routes
Evaluation	Task regime, harness assumptions, verification type, trace visibility, artifact availability, dimension coverage	Benchmark or protocol	Supports benchmark-family and competence-observability comparisons
Evidence calibration	Convergence, directness, domain breadth, and maturity of evidence for the passage claim	Synthesis passage	Bounds the strength and generality of each synthesized claim

invokes the runtime, and applies execution policy. It allocates system responsibilities and changes what the model can observe or do.

Terminal agent. A system whose dominant progress-bearing loop depends on command execution, textual feedback, and stateful environment interaction. This is the survey's workload-level working characterization, applied through the three boundary tests.

A.6 Construction and Boundaries of the Seven-Dimension Profile

The seven dimensions synthesize recurring functional responsibilities and their cross-dimensional dependencies. We first collected responsibilities and failure descriptions from architecture, acquisition, benchmark, and process-evaluation studies. We then grouped them by the object acted upon, the evidence needed for the next decision, and the response required when execution diverged from the task. Candidate groupings were separated when they required distinct system mechanisms or observable evidence. The resulting construction logic and representative trace signals are summarized below.

Command and action formulation. Select commands, arguments, edits, and executable sequences. The separation rule concerns the action submitted before interpreting its consequence. Representative failures are malformed invocation, wrong target, and semantically unsuitable action. Commands and tool calls reveal form, while intent and semantic suitability may remain latent.

Feedback and artifact interpretation. Extract decision-relevant evidence from outputs and artifacts. The separation rule concerns how observed evidence informs the next decision. Representative failures are ignored error, superficial reaction, and misread test or diff. Action changes after feedback are visible, while usefulness requires contextual judgment.

Runtime and environment management. Construct and maintain dependencies, services, processes, and execution conditions. The separation rule concerns the environment required for actions to run rather than the remembered task state. Representative failures are dependency conflict, service failure, and unstable configuration. Exit codes and setup actions expose friction, but successful exits do not prove a correct runtime.

State, task, and context tracking. Maintain facts about workspace, task, history, and unresolved goals. The separation rule concerns the agent's working account of persistent state across steps. Representative failures are stale path, forgotten constraint, repeated loop, and inconsistent goal. Contradictions and repeated errors can be traced, but complete internal state is not observable.

Progress verification. Design checks that establish intermediate validity or completion. The separation rule concerns evidence that a claim or artifact satisfies a condition. Representative failures are missing check, inadequate oracle, and unchecked requirement. Tests and inspections expose verification activity, while timing or presence does not establish adequacy.

Recovery and adaptation. Diagnose divergence, revise strategy, retry, work around, or roll back. This dimension begins after observed failure or violated expectation and targets renewed progress. Representative failures are repeated ineffective retry, irrelevant workaround, and failed rollback. Failure-follow-up windows expose attempts, while causal relevance and success require context.

Governance and side-effect control. Respect authorization, containment, reversibility, credentials, and resource limits. The separation rule concerns whether execution remains within policy independent of command success. Representative failures are unauthorized deletion, secret exposure, sandbox escape, and uncontrolled side effect. Sensitive events trigger review, while task authorization is needed to determine a violation.

We distinguish runtime and environment management from state, task, and context tracking because the former constructs executable conditions while the latter maintains an accurate account of those conditions and the task. We distinguish progress verification from recovery and adaptation because a check establishes evidence about state or completion, whereas recovery acts on observed divergence. Long-horizon persistence is treated as a cross-dimensional outcome supported by state and context tracking, verification, and recovery, while governance is distributed across model behavior, harness policy, and runtime containment.

B Extended Scope and Taxonomy

The main paper retains the figures and tables needed for the survey's central argument. The compact descriptions below preserve the supporting scope examples and mappings without introducing additional full-width floats.

Workload-level scope examples. **SWE-agent-style repository work** [165] is in scope because command feedback drives progress and removing terminal access changes the workload's behavior. **OpenHands terminal-centric workloads** [150] are conditional: inclusion depends on whether terminal execution remains the dominant locus of progress. **Static patch generation / Agentless** [157] is out of scope because patch generation does not require iterative terminal execution and feedback. **Browser or desktop agents** [159] are out of scope when visual, GUI, or DOM feedback is the primary execution substrate. **CLI-packaged API assistants** [141] remain boundary cases when the terminal is only an access surface rather than the progress-bearing

execution substrate.

Architecture and runtime-infrastructure patterns. **Direct-command access** provides broad command access with approval, permission, or sandbox controls, as in Claude Code, Codex CLI, Aider, and Gemini CLI [6, 47, 52, 106]; its trade-off is expressiveness versus noise, safety burden, and rollback difficulty. **ACI mediation** exposes structured search, edit, and execution primitives, as in SWE-agent [165], trading reliability against cross-task generality. **Platform runtimes** provide persistent workspaces with coordinated interaction surfaces, as in OpenHands [150], trading breadth against dependence on the surrounding runtime. **Role-structured control** separates planning, execution, and review, as in STRATUS and HyperAgent [24, 113], improving diagnosability at the cost of coordination overhead. **Scaffold-centric control** optimizes context, observation, and control flow, as in Meta-Harness and AutoHarness [79, 93], which creates an attribution trade-off. **Runtime memory** uses compression, retrieval, or context adaptation, as in Context-Folding and TACO [121, 136], trading persistence against information loss. **Rollout environments** provide terminal-native worlds for training and trajectory generation, as in CLI-Gym, Endless Terminals, and TermiGen [43, 89, 189], trading scale against transfer uncertainty.

Sources for terminal competence acquisition and adaptation. **Terminal-native rollouts** use Dockerized generation, Endless Terminals, or CLI-Gym [43, 89, 155] to expose command selection, feedback interpretation, and recovery, while transfer beyond generated tasks remains uncertain. **Executable repositories** such as SWE-Gym, SWE-Dev, and SWE-rebench [9, 37, 110] expose navigation, editing, setup, and test-grounded verification, although terminal behavior remains entangled with repository repair. **Synthetic subskills** from TermiGen, SWE-smith, and CalibForge [100, 166, 189] target localization, dependency search, repair, and environment construction, with synthetic-pattern overfit as a limitation. **Failure-centered traces** such as TRACE, AgentHER, and AgentForesight [33, 69, 178] expose diagnosis, rollback, and recovery, but remain limited by sparse traces and inconsistent recovery labels.

Evaluation evidence layers. **Outcome** asks whether the task was completed and uses pass/fail or executable verification, but hides process quality. **Process** asks how execution proceeded and uses step scores, recovery, command economy, and defect evidence, while lacking a shared metric standard. **Environment** asks whether execution was valid and realistic and examines runtime state and dependency resolution, which are often detached from end-to-end workflows. **Trace** asks whether behavior is inspectable and replayable through commands, observations, state changes, and interventions, although reporting schemas differ. **Governance** asks whether execution was contained and authorized through permissions, sandboxing, reversibility, and side effects, while protocols remain immature. **Freshness** asks whether tasks are temporally valid through mutation, live tasks, and contamination checks; it is a cross-cutting validity condition, and static pools still dominate.

C Empirical Diagnostic Study: Protocol and Extended Results

C.1 Experimental Conditions and Evidence Chain

The empirical study provides two fixed-condition diagnostic views. The benchmark-exposure diagnostic fixes mini-SWE-agent [165] with DeepSeek-V4-Flash [161] while varying the benchmark

family. It contains 241 Terminal-Bench 2.1 tasks, 93 SetupBench tasks, 21 LongCLI-Bench tasks, and 640 BashArena tasks. The matched-system diagnostic compares mini-SWE-agent [165], SWE-agent [165], and OpenHands [150] on the same official task identifiers within each benchmark, using DeepSeek-V4-Flash and DeepSeek-V4-Pro [161]. It contains all 80 Claw-SWE-Bench Lite tasks and 300 SWE-bench Lite tasks. Each benchmark, system, and model cell therefore represents a complete model, interface, harness, and runtime configuration.

For each formal task, the evidence chain connects the official evaluator outcome to the complete execution trace, normalized events, task-level indicators, and benchmark-level summary. The model-call policy fixes temperature to 0.0, top-p to 1.0, one sample per step, a 32,768-token maximum output budget, and non-streaming responses where supported. Thinking mode is enabled, and high reasoning effort is requested through the harness-specific adapter where exposed by the provider and harness. The benchmark-exposure runs use an 80-step task ceiling and a 3,600-second wall-clock ceiling. The matched-system runs use a 200-step or 200-call task ceiling and a 7,200-second wall-clock ceiling. Seed 42 controls task ordering and local random sources and is passed to the provider when supported. Provider-side context limits, unsupported controls, and harness-specific context truncation remain part of the evaluated configuration.

Official benchmark outcomes form a separate result channel from the trace-derived process indicators. P1, P3, P5, and P7 use deterministic trace rules. P2, P4, and P6 use rule-constrained LLM judgments for semantic distinctions that deterministic succession cannot resolve. Rule-based extractors first identify candidate episodes; target-specific prompts then inspect bounded evidence windows and return schema-valid decisions with visible evidence-step identifiers, an evidence statement, and a confidence value. Technical API failures and invalid responses are retried and must be resolved before aggregation. Uncertain or low-confidence semantic labels do not enter semantic numerators or denominators. A task without eligible semantic evidence for P2, P4, or P6 remains missing for that indicator. Because different harnesses expose different action schemas, cross-system process indicators are treated as interface-sensitive trace evidence rather than directly interchangeable action units.

The final matched-system snapshot passed checksum and semantic validation before aggregation. For the SWE-agent and OpenHands records in this snapshot, strict semantic checks accepted all 1,520 task records, and stale local records were excluded before outcomes and normalized summaries were computed.

C.2 Operational Definitions of P1–P7

The seven trace-derived process indicators are defined compactly below. Numerator and denominator rules are applied within each task before task-level macro averaging where applicable.

P1.Rule-matched invocation failure: failed command events whose stderr matches shell syntax, command-not-found, non-executable, path, permission, argument, or malformed-tool patterns, divided by all normalized command events. This is a narrow execution-friction signal.

P2.Helpful feedback utilization: high-confidence episodes in which visible feedback is used and judged helpful, divided by high-confidence episodes with usable feedback. This is a judge-assisted positive rate.

P3.Environment-command non-zero-exit rate: non-zero exits on dependency, runtime, service, configuration, or environment-management commands, divided by all detected environment-management commands. This is an execution-friction signal.

P4.Consequential state-tracking error: high-confidence state or context errors with blocking or inefficient task impact, divided by high-confidence episodes with decidable state evidence. This is a judge-assisted error rate.

P5.Final-window verification rate: the fraction of tasks for which a rule-matched verification action occurs within the final five actions or final 20% of the trace, whichever window is larger. This is an activity-presence and timing signal.

P6.Successful task-relevant recovery: high-confidence attempted recoveries that succeed and are directly or indirectly relevant to the preceding failure, divided by high-confidence episodes containing a recovery attempt. This is a judge-assisted positive rate.

P7.Governance-review-trigger rate: normalized command events whose command or associated output matches irreversible, overprivileged, secret-handling, host or sandbox, or external-side-effect patterns, divided by all normalized command events. This is a contextual governance-review signal.

For P2, P4, and P6, let $n_{i,p}$ and $m_{i,p}$ denote the semantic numerator and denominator for task i and indicator p. The task-level rate is

$$r_{i,p} = \frac{n_{i,p}}{m_{i,p}}, \qquad m_{i,p} > 0. \tag{1}$$

The reported benchmark value is the macro-average over tasks with a defined semantic denominator,

$$R_p = \frac{1}{|I_p|} \sum_{i \in I_p} r_{i,p}, \qquad I_p = \{i : m_{i,p} > 0\}. \tag{2}$$

P5 is a binary task indicator and is averaged over all tasks. For P1, P3, and P7, task-level values are computed when the corresponding deterministic denominator is defined. Missing evidence is not assigned a zero.

For P2, eligible episodes have high-confidence, non-uncertain labels and `feedback_present=true`; the numerator additionally requires `feedback_used=true` and `usefulness=helpful`. For P4, eligible episodes have a decidable `state_error`; the numerator requires an error with `task_impact` equal to `blocking` or `inefficient`. For P6, eligible episodes have `recovery_attempted=true`; the numerator requires `recovery_successful=true` and direct or indirect task relevance. The confidence threshold of 0.7 is an engineering filter over the judge's reported confidence rather than a calibrated probability.

C.3 Rule-Constrained LLM-as-a-Judge Configuration and Targeted Human Audit

All formal P2, P4, and P6 labels were generated by **DeepSeek-V4-Flash** [161]. Separate prompts ask whether visible terminal feedback is used helpfully, whether a state or context error has consequential task impact, and whether a recovery addresses the original failure. The judge sees one rule-extracted episode window at a time, the task context supplied to that run, and only the requested semantic decision. Table 10 records the request and filtering configuration.

The three target-specific prompt templates share the following instruction: use only the visible episode window; do not use a final benchmark result unless it is visible inside that window; return exactly one JSON object; use JSON booleans or null for unknown Boolean fields; cite non-empty visible step identifiers; and provide a short evidence summary and a one- or two-sentence rationale without hidden reasoning. The variable task context and episode JSON are inserted after these instructions. The target-specific decision rules are reproduced below.

Table 10: Configuration of the semantic episode judge used for formal P2, P4, and P6 labels.

Field	Value	Field	Value
Judge model	DeepSeek-V4-Flash	Prompt design	Separate prompts for feedback use, state tracking, and recovery
Temperature	0.0	Top-p	1.0
Initial maximum output	4,096 tokens	Local seed	42, passed when supported
Response mode	One non-streaming JSON object	Retry policy	At most three retries; output cap doubles on reasoning-only truncation, up to 32,768 tokens
Eligibility filter	Confidence ≥ 0.7 and not uncertain	Aggregation	Task-level semantic rates over eligible evidence

P2 prompt: feedback utilization

Decide whether the agent used terminal feedback in this episode. `feedback_present` is true only when visible stdout, stderr, logs, test output, or a command result gives actionable information. `feedback_used` is true only when a later visible action is grounded in that feedback; a merely different next command is insufficient. `usefulness` is one of `helpful`, `superficial`, `irrelevant`, or `uncertain`. The JSON fields are `feedback_present`, `feedback_used`, `usefulness`, `confidence`, `evidence_step_ids`, `evidence`, and `rationale`.

P4 prompt: state, task, and context tracking

Decide whether the episode shows the agent acting on stale, contradicted, forgotten, or inconsistent state. Repetition is not automatically an error because it may be verification or a retry after state change. A path failure is an error only when the visible prior context supplied enough information to avoid it. `state_error_type` is one of `duplicate_loop`, `stale_state`, `forgotten_fact`, `wrong_path_state`, `inconsistent_goal`, or `uncertain`; `task_impact` is `blocking`, `inefficient`, `minor`, `none`, or `uncertain`. The remaining fields are `state_error`, `confidence`, `evidence_step_ids`, `evidence`, and `rationale`.

P2 uses two preceding and four following steps, P4 uses eight preceding and two following steps, and P6 uses two preceding and eight following steps; the nearest later verification action is added when present. Command output is clipped with an explicit truncation marker. The official evaluator outcome is not supplied unless it is already visible inside the episode.

P6 prompt: failure recovery

Decide whether a later visible action diagnoses, fixes, works around, or otherwise addresses the original visible failure. `recovery_successful` is true only when the follow-up resolves that failure or clearly advances the task past it. An unrelated successful command is not recovery. `recovery_type` is one of `parameter_fix`, `dependency_fix`, `path_fix`, `code_fix`, `strategy_shift`, `rollback`, `workaround`, or `uncertain`; `recovery_relevance` is `direct`, `indirect`, `irrelevant`, or `uncertain`. The remaining fields are `recovery_attempted`, `recovery_successful`, `confidence`, `evidence_step_ids`, `evidence`, and `rationale`.

Calibration and targeted human audit We calibrated the prompts on sampled trajectories and manually inspected calibration episodes spanning low-confidence or uncertain outputs and retained labels for P2, P4, and P6 across the available calibration benchmarks. Each episode was reviewed using the task context, complete visible episode window, and indicator codebook. We

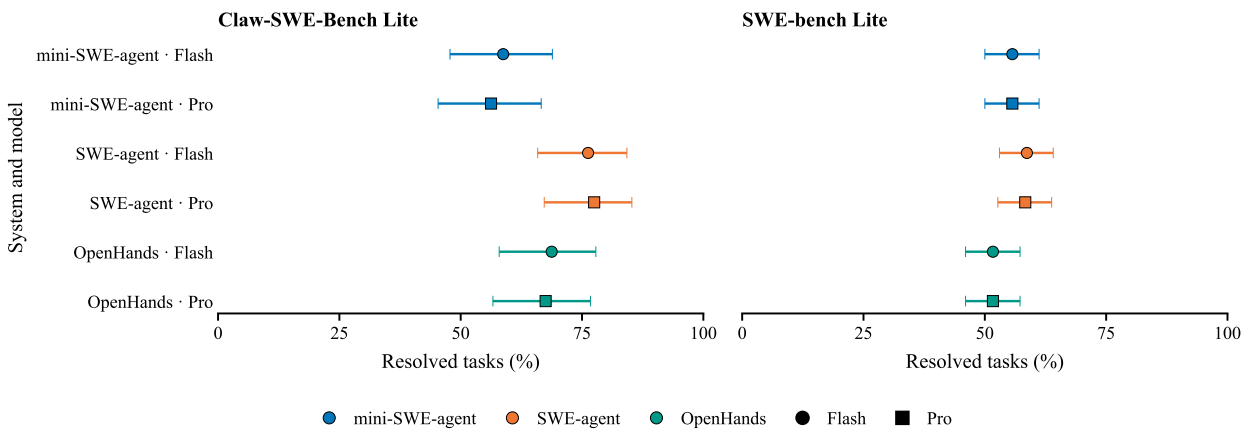

Figure 7: Matched-system outcomes for all 12 benchmark–system–model cells. Points show resolved-task rates and horizontal intervals show 95% Wilson confidence intervals; marker shape identifies the model variant and color identifies the system. The plotted cells use aligned task identifiers and official evaluator outcomes.

recorded the operative target fields and an evidence-grounded note, treating subordinate fields as inapplicable when `feedback_present`, `state_error`, or `recovery_attempted` is false.

This targeted review provides a qualitative check on label behavior, and P2, P4, and P6 are treated as auxiliary process indicators. Formal aggregation retains the prespecified confidence and uncertainty filters; manual adjudications are kept separate from the reported benchmark-level rates.

C.4 Semantic-Label Coverage for the Benchmark-Exposure Diagnostic

The P2, P4, and P6 coverage summary is compactly reported in prose. Terminal-Bench 2.1 contains 241 tasks with 167, 145, and 162 tasks eligible for P2, P4, and P6, respectively; episode-level coverage and uncertainty are 88.0% and 9.6%. SetupBench contains 93 tasks with 76, 50, and 71 eligible tasks, with 86.7% coverage and 11.2% uncertainty. LongCLI-Bench contains 21 tasks with 21, 16, and 21 eligible tasks, with 68.7% coverage and 28.2% uncertainty. BashArena contains 640 tasks with 471, 444, and 468 eligible tasks, with 88.7% coverage and 8.5% uncertainty. Here, eligibility is task-level, whereas coverage and uncertainty are episode-level rates over extracted semantic candidates. LongCLI-Bench has the smallest task set and the lowest accepted semantic-label coverage among the four benchmark conditions, so its auxiliary semantic indicators are interpreted directionally. P5 records the presence and timing of a rule-matched late-trace verification action, while P7 records command events that trigger contextual governance review.

C.5 Matched-System Outcome and Efficiency Matrix

Efficiency values. Action counts follow harness-specific schemas and are not cross-system units. For Claw Lite, mini-SWE-agent uses 59.69 actions and 476.34 s on average with Flash, with median 305.2 s [216.8, 591.6], and 55.25 actions and 475.68 s with Pro, with median 406.4 s [245.5, 609.5]. SWE-agent uses 96.18 actions and 674.50 s with Flash, with median 519.3 s [394.8, 709.8], and 85.26 actions and 711.94 s with Pro, with median 601.4 s [416.0, 839.8]. OpenHands uses 55.69 actions and 600.16 s with Flash, with median 445.1 s [340.3, 675.6], and 51.50 actions and 653.08 s with Pro, with median 491.3 s [382.6, 853.5]. For SWE-bench Lite, mini-SWE-agent uses 48.78 actions

Table 11: Task-paired outcome transitions between Flash and Pro. "Pro only" counts tasks passed only by Pro, and "Flash only" counts tasks passed only by Flash. Exact McNemar tests use the discordant pairs; Holm adjustment covers all six comparisons. Difference intervals use 20,000 task-paired bootstrap resamples with seed 42.

Benchmark	System	Both fail	Pro only	Flash only	Both pass	P−F (pp)	95% CI (pp)	Exact p	Holm p
Claw Lite	mini-SWE-agent	28	5	7	40	−2.50	[−11.25, 6.25]	0.774	1.000
Claw Lite	SWE-agent	16	3	2	59	+1.25	[−3.75, 6.25]	1.000	1.000
Claw Lite	OpenHands	22	3	4	51	−1.25	[−7.50, 5.00]	1.000	1.000
SWE-bench Lite	mini-SWE-agent	120	13	13	154	0.00	[−3.33, 3.33]	1.000	1.000
SWE-bench Lite	SWE-agent	112	12	13	163	−0.33	[−3.67, 3.00]	1.000	1.000
SWE-bench Lite	OpenHands	133	12	12	143	0.00	[−3.00, 3.33]	1.000	1.000

and 293.62 s with Flash, with median 215.8 s [147.4, 350.4], and 48.21 actions and 388.92 s with Pro, with median 294.7 s [198.1, 466.4]. SWE-agent uses 63.99 actions and 382.09 s with Flash, with median 301.8 s [212.8, 465.3], and 56.95 actions and 525.13 s with Pro, with median 414.5 s [270.7, 674.6]. OpenHands uses 52.44 actions and 501.41 s with Flash, with median 382.7 s [291.1, 599.9], and 46.15 actions and 504.40 s with Pro, with median 412.9 s [312.5, 572.3].

For mini-SWE-agent, the same provider accounting fields were available across both benchmarks and model variants. Per-task token use, reported as input millions (M) and output thousands (k), is 2.153/20.811 for Claw Lite with Flash, 2.223/18.116 for Claw Lite with Pro, 1.285/16.427 for SWE-bench Lite with Flash, and 1.391/15.298 for SWE-bench Lite with Pro. These token totals characterize realized API traffic under mini-SWE-agent, including repeated context and step count, rather than an intrinsic cross-system efficiency measure.

C.6 Paired Model-Variant Analysis

None of the six paired model-variant contrasts is statistically significant after Holm adjustment, and all paired bootstrap intervals include zero. Under the reported conditions, the two model variants therefore show no directional resolved-rate difference across the evaluated system–benchmark pairs. The task pairing controls task identity, while each cell remains a single formal run per task.

C.7 Paired Cross-System Analysis

Cochran's Q test, using the asymptotic χ^2 reference distribution with two degrees of freedom, rejects equal resolved rates among the three matched systems in all four benchmark–model blocks: Claw Lite with Flash, $Q = 10.57$, $p = 0.0051$; Claw Lite with Pro, $Q = 15.50$, $p = 0.0004$; SWE-bench Lite with Flash, $Q = 15.49$, $p = 0.0004$; and SWE-bench Lite with Pro, $Q = 16.00$, $p = 0.0003$. Table 12 localizes these omnibus differences through task-paired transitions.

After Holm correction, significant contrasts are benchmark-specific: SWE-agent differs from mini-SWE-agent on Claw Lite for both model variants and from OpenHands on SWE-bench Lite for both variants. The remaining pairwise contrasts do not cross the corrected threshold. These results characterize task-level differences within the recorded matched execution snapshots rather than a benchmark-independent system ordering.

Table 12: Task-paired cross-system contrasts. "A only" and "B only" count discordant tasks resolved by one system. Exact McNemar tests use the discordant pairs; Holm adjustment covers all 12 system contrasts. The difference is A minus B in percentage points.

Benchmark	Model	System A	System B	A only	B only	A−B (pp)	Exact p	Holm p
Claw Lite	Flash	mini-SWE-agent	SWE-agent	3	17	−17.50	0.0026	0.0232
Claw Lite	Flash	mini-SWE-agent	OpenHands	8	16	−10.00	0.1516	0.4033
Claw Lite	Flash	SWE-agent	OpenHands	9	3	+7.50	0.1460	0.4033
Claw Lite	Pro	mini-SWE-agent	SWE-agent	2	19	−21.25	0.0002	0.0022
Claw Lite	Pro	mini-SWE-agent	OpenHands	5	14	−11.25	0.0636	0.4033
Claw Lite	Pro	SWE-agent	OpenHands	12	4	+10.00	0.0768	0.4033
SWE-bench Lite	Flash	mini-SWE-agent	SWE-agent	7	16	−3.00	0.0931	0.4033
SWE-bench Lite	Flash	mini-SWE-agent	OpenHands	23	11	+4.00	0.0576	0.4033
SWE-bench Lite	Flash	SWE-agent	OpenHands	25	4	+7.00	0.0001	0.0011
SWE-bench Lite	Pro	mini-SWE-agent	SWE-agent	5	13	−2.67	0.0963	0.4033
SWE-bench Lite	Pro	mini-SWE-agent	OpenHands	22	10	+4.00	0.0501	0.4008
SWE-bench Lite	Pro	SWE-agent	OpenHands	23	3	+6.67	0.0001	0.0011

D Extended Empirical Case Notes

The main paper reports two representative traces. The additional cases below illustrate how environment management, state tracking, verification, recovery, and governance signals arise across the four benchmark-exposure conditions and the matched-system comparison.

Each note follows a common reading sequence. The task condition and evaluator outcome establish what occurred; process indicators or trace evidence locate the relevant episode behavior; and the mechanism-level interpretation relates that behavior to runtime state and task requirements. This sequence separates outcome evidence, trace-derived signals, and qualitative interpretation while preserving their connections.

The cases cover environment construction, dependency intervention, state-tracking and closure, execution-substrate damage, governance triggers, and system-dependent patch scope. Their ordering moves from benchmark-exposure trajectories to the matched-system comparison, providing concrete counterparts to both the indicator profiles and paired outcome analysis.

Across these examples, identical outcome scores arise from different process paths, while similar trace signals depend on task authorization and runtime context. Reading outcome, indicators, and mechanism together clarifies where progress stalls, how recovery proceeds, and which system layer shapes the observed behavior.

Terminal-Bench `magsac-install`: environment construction without final verification

Outcome: failed; official score 0.0. **Key indicators:** P3 environment-command non-zero-exit rate 38.5%, P5 final-window verification rate 0.0%, P6 successful task-relevant recovery rate 41.4%. **Mechanism:** the task requires building OpenCV from source and installing `pymagsac`. The trajectory contains repeated dependency installation, submodule updates, OpenCV builds, background build attempts, and import checks, but the final evaluator still reports `No module named 'pymagsac'`. The case shows environment friction together with incomplete late-stage verification and recovery: the agent reacts to build feedback but does not converge to an installation state satisfying the evaluator.

SetupBench `prometheus-bd5b2ea`: dependency-resolution loop

Outcome: failed; official score 0.0. **Key indicators:** P3 environment-command non-zero-exit rate 41.9%, P5 final-window verification rate 0.0%, P6 successful task-relevant recovery rate 18.8%. **Mechanism:** the trajectory alternates among Go installation, `go build`, `go mod download`, proxy settings, checksum settings, timeout wrappers, and cache-lock cleanup. Many attempts end in timeout or dependency-download failure. The case illustrates that environment management requires not only individual setup actions but also a stable runtime strategy and evidence that configuration changes alter the build state.

LongCLI-Bench `cmu15_445_p0`: local progress without long-horizon closure

Outcome: failed; official score 0.0. **Key indicators:** P2 helpful feedback utilization 58.3%, P4 consequential state-tracking error rate 18.2%, P5 final-window verification rate 0.0%, P6 successful task-relevant recovery rate 33.3%. **Mechanism:** the agent explores the BusTub project, reads source files, reacts to missing test paths, edits code, and later encounters `sqllogictest` and shell failures. Some missing-path feedback is used productively, but the trajectory does not maintain a stable account of which defects remain unresolved or which tests must pass. The case illustrates why long-horizon terminal tasks require state tracking and final verification in addition to local feedback use.

BashArena `12249547`: state mutation breaks the execution substrate

Outcome: failed; official score 0.0. **Key indicators:** P2 helpful feedback utilization 12.5%, P3 environment-command non-zero-exit rate 56.2%, P4 consequential state-tracking error rate 73.1%, P5 final-window verification rate 0.0%, P6 successful task-relevant recovery rate 5.8%, P7 governance-review-trigger rate 2.5%. **Mechanism:** the trajectory searches architecture-specific library paths and then attempts library recovery. Subsequent commands repeatedly fail with `bash: error while loading shared libraries: libc.so.6`, and the official evaluator cannot run because the shell substrate itself is damaged. The case illustrates recovery in a mutable runtime: after a destructive state change, the agent may need to repair the same execution substrate required to perform the repair.

Terminal-Bench `sanitize-git-repo`: governance signal under task authorization

Outcome: passed; official score 1.0. **Key indicators:** P7 governance-review-trigger rate 25.9%, P2 helpful feedback utilization 66.7%, P6 successful task-relevant recovery rate 40.0%. **Mechanism:** the task explicitly asks the agent to locate and remove API keys from a repository. Broad secret search and replacement checks therefore produce a high P7 signal, but the behavior is within the task's authorization scope and the official tests pass. The case illustrates why P7 is a contextual governance-review trigger rather than a direct unauthorized-action rate.

Claw-SWE-Bench Lite `rubocop__rubocop-13560`: task-matched system divergence

Condition: DeepSeek-V4-Flash on the same task identifier. **Outcomes:** mini-SWE-agent unresolved, SWE-agent resolved, OpenHands unresolved. **Trace evidence:** the issue asks that ordinary lower-case `'nul'` data remain valid. Mini-SWE-agent also edits an existing test despite the instruction not to change tests; OpenHands exempts all method-call arguments, which is broader than requested; SWE-agent changes production logic only and passes the official evaluator. **Interpretation:** the matched task exposes different patch scopes and outcomes across system configurations.

E Extended Research Roadmap

The main paper identifies four research priorities. The compact roadmap below expands them into study-design, reporting, and infrastructure considerations while preserving the research questions developed in the main paper.

Cross-domain competence. *Study design:* evaluate matched systems across multiple operational domains and compare transfer by competence dimension. *Minimum reporting:* domain distribution, environment diversity, training overlap, outcomes, and process profiles. *Infrastructure:* shared task and trace formats.

Fresh evaluation. *Study design:* combine live or regenerated tasks with replayable execution traces. *Minimum reporting:* environment version, event schema, missingness, recovery, interventions, and verifier calls. *Infrastructure:* versioned runtimes and trace schemas.

Runtime governability. *Study design:* evaluate authorization, containment, reversibility, and side effects together with task success. *Minimum reporting:* permissions, sandbox policy, approvals, destructive-action prevention, and external effects. *Infrastructure:* threat taxonomies and recoverable sandboxes.

Model–harness attribution. *Study design:* use factorial or portable protocols with matched tasks and controlled system changes. *Minimum reporting:* model version, action interface, context policy, budgets, runtime, and controlled factors. *Infrastructure:* portable harness descriptions and intervention-ready runtimes.