

Shell-Based Agents as Execution Systems: A Survey of Command Generation, Verification, Security, State, and Recovery

Jyoti Dabass^{1,*}, Bhupender Singh Dabass²

Abstract

Shell environments are increasingly becoming interfaces through which AI agents interact with software and operating systems. Unlike conventional command generation systems, shell-based agents can interpret task objectives, plan multi-step actions, invoke terminal tools, observe system state, and adapt subsequent actions. These capabilities support software development, system administration, experimentation, and automated operations, but also introduce reliability and security challenges that cannot be assessed through command generation alone. This survey reviews 57 studies published or publicly available between 2024 and 2026 on shell scripting, terminal-based agents, agentic workflows, execution environments, security, developer tools, and related evaluation methods. The reviewed studies were analyzed using structured evidence records covering research objectives, technical approaches, execution environments, evaluation strategies, findings, and limitations. The review examines shell-based agents as end-to-end execution systems, considering task interpretation, planning, command construction, execution, observation, state management, verification, and recovery. Findings indicate that syntax-aware generation, structured task representations, additional training signals, and multi-agent approaches can improve specific aspects of shell automation. However, reliable and secure execution also depends on tool interfaces, permission models, isolation mechanisms, state representations, approval processes, and recovery strategies. Security studies further indicate that unsafe behavior can originate from contextual sources such as logs, repository files, tool descriptions, external instructions, and agent skills. The review identifies a lack of standardized evaluation across functional correctness, security, efficiency, state consistency, and recovery. Future research should therefore emphasize trajectory-level evaluation, session-wide security analysis, fine-grained permissions, state-aware recovery, broader operating system coverage, and reproducible execution environments to enable reliable and controllable shell-based agents.

Keywords: Shell-based agents; terminal agents; agentic workflows; shell scripting; command generation; command verification; tool use; execution security; state management; agent evaluation; recovery; large language models

*Author for Correspondence

Jyoti Dabass
E-mail: jyotidabas91@gmail.com

¹Global Professor of Practice, Golden Gate University, San Francisco, California, USA

²Researcher and Advocate, District Bar Association, Palwal, Haryana, India

Received Date: September 23, 2026

Accepted Date: September 26, 2026

Published Date: September 28, 2026

Citation: Jyoti Dabass, Bhupender Singh Dabass. Shell-Based Agents as Execution Systems: A Survey of Command Generation, Verification, Security, State, and Recovery. Journal of Advances in Shell Programming. 2026; 13(2): 30–54p.

INTRODUCTION

Shell Environments provide direct access to files, processes, operating system services, network utilities, build systems, and containers. Bash is particularly useful for these tasks because pipes, redirection, variables, quoting, expansion, and process control allow complex operations to be expressed through compact command sequences and scripts. With the emergence of large

language models (LLMs), these capabilities are increasingly being incorporated into agentic systems that can translate natural-language goals into shell actions, execute those actions, inspect the resulting output, and determine subsequent steps. This setting differs from conventional code generation because generated shell commands can immediately modify the execution environment. An agent may create or delete files, change permissions, install packages, modify configurations, control processes, access sensitive information, or communicate with external services. A command can therefore be syntactically valid while producing an incorrect result or violating constraints imposed by the environment. The challenge becomes more pronounced when a task consists of dependent operations, as later actions may rely on changes produced by earlier ones. Reliable execution consequently requires more than producing an appropriate command. Existing research addresses several aspects of this problem. Studies on shell command and script generation have explored syntax-aware generation, structured representations, robustness, and local language models [1–8]. Other work has examined mechanisms for checking and controlling execution. CARE investigates pre-execution verification for shell-executing LLM agents, while shell type systems provide additional information for identifying potentially problematic operations before execution [9, 10]. Fractal and incremental re-execution (INCR) investigate execution-related issues including fault tolerance and efficient re-execution [11, 12]. Together, these studies indicate that generation, verification, and execution represent related but distinct parts of the shell-agent problem. The interface connecting an agent to the execution environment also influences system behavior. Research has examined tool representation, selection, and invocation in multi-step agent workflows [13–19]. Differences in tool granularity, available operations, returned information, and execution control can affect how an agent plans and acts. Consequently, agent behavior cannot always be attributed to the underlying language model alone. Security presents a related systems-level concern. During execution, an agent may process repository files, logs, command output, tool descriptions, skills, workflow instructions, and other external information. Studies of workflow injection, skill security, and related threats show that these inputs can influence subsequent agent decisions [20–30]. Security must therefore be considered across the interaction between the agent, its tools, its context, and the execution environment rather than being limited to inspection of the final command. The literature remains distributed across command generation, runtime systems, security, tool use, and agent architecture.

These research areas often employ different tasks, environments, models, and evaluation criteria, which limit direct comparison of their findings. In particular, command-level measures do not capture whether an operation succeeds in the target environment, preserves the intended state, respects security constraints, or recovers appropriately after failure. This separation motivates an execution-oriented view of shell agents. This survey examines 57 studies covering shell and terminal agents, command and script generation, verification, execution, tool interfaces, agent architectures, security, stateful interaction, sandboxing, and environment-level evaluation. The studies are analyzed through a common execution workflow comprising goal interpretation, planning and tool selection, command generation, verification, execution, observation and state management, and recovery. Security is treated as a cross-cutting dimension because relevant threats and controls may occur at multiple stages. The analysis focuses on how existing methods affect not only command quality but also execution correctness, security, state consistency, recovery, efficiency, and reproducibility. This perspective allows the survey to distinguish improvements in the language model from those arising from the surrounding agent architecture and execution environment. The survey makes four contributions. First, it provides an execution-oriented framework for organizing shell-agent research from task interpretation through recovery. Second, it connects findings from command generation, verification, runtime execution, tool interfaces, security, and state management. Third, it examines the limitations of current evaluation practices and the factors that prevent consistent comparison across studies. Fourth, it identifies research directions for evaluating shell agents under realistic conditions involving changing system state, execution failures, security constraints, and long-horizon tasks. The remainder of the paper is organized as follows. Section 2 defines the review scope, research questions, evidence organization procedure, and analytical framework. Section 3 examines command generation,

pre-execution verification, execution, monitoring, and recovery. Section 4 reviews tool interfaces, agent architectures, permissions, authorization, privacy, and execution security. Section 5 presents the evaluation practices used across the reviewed studies, covering generation, execution, security, state, recovery, efficiency, and reproducibility. Section 6 synthesizes the cross-study findings and identifies the major research gaps in shell-agent research. Section 7 discusses future research directions, including realistic evaluation, session-level security, state-aware recovery, broader execution environments, controlled system design, and reproducible execution. Section 8 describes the limitations of the survey and the scope of its conclusions. Section 9 concludes the paper.

REVIEW SCOPE, RESEARCH QUESTIONS, AND EVIDENCE SYNTHESIS

This section describes the boundaries of the review, the questions used to organize the literature, and the procedure followed to synthesize findings across different types of shell-agent research. The analysis is designed to connect command generation with the broader execution, security, and evaluation processes examined throughout the survey.

Review Scope

The review is restricted to research in which shell commands, scripts, command-line tools, or shell execution environments play a substantive role in an automated or agentic workflow. This scope establishes which studies are included and separates shell-focused evidence from broader work on LLM-based agents.

Research Questions

The research questions define the dimensions along which the selected studies are examined. They cover generation, execution, system interaction, security, and evaluation so that findings from different research areas can be considered within a common framework. The review is structured around five research questions.

RQ1. How are natural-language requests translated into shell commands or scripts, and which techniques improve their correctness?

This question covers syntax-aware generation, structured representations, model adaptation, training methods, robust techniques, and related approaches for improving generated shell commands and scripts.

RQ2. How are commands generated, verified, executed, optimized, and recovered when execution fails?

This question considers the mechanisms between generation and task completion, including pre-execution verification, static analysis, execution control, fault tolerance, incremental execution, monitoring, checkpointing, and recovery.

RQ3. How do tool interfaces, permissions, state, and execution environments shape agent behavior?

The focus is on the components surrounding the language model, including tool design, action representation, permission boundaries, isolation, environmental state, execution feedback, and interaction between the agent and the system in which its actions are performed.

RQ4. Which security threats can arise before, during, and after shell command execution?

This question examines security across the execution process, including unsafe command generation, indirect prompt injection, misleading or malicious tool information, skill and workflow manipulation, excessive permissions, information leakage, and unsafe execution.

RQ5. How are shell-based agents evaluated, and which factors make results difficult to compare?

This question addresses the metrics, benchmarks, models, task definitions, execution environments, security configurations, and reporting practices used across existing studies. Particular attention is given to the difference between evaluating an individual command and the completion of a broader task. These questions are interdependent. A change in command generation can affect subsequent verification, whereas tool design and environmental conditions can determine whether the generated action succeeds. Security restrictions can also modify the actions available to an agent, whereas execution failures can alter the state encountered by later steps. The synthesis therefore treats these aspects as interacting elements of one execution process.

Evidence Organization and Study Coding

The selected studies were coded using a common evidence structure so that technically different contributions could be examined using consistent categories. Coding captures both the proposed method and the conditions under which its reported findings were obtained. Each study was represented using a structured evidence record containing its research problem, proposed approach, shell or execution environment, model or agent architecture, input and output representation, tools, evaluation setting, metrics, findings, limitations, and future research directions. This coding scheme enables studies with different experimental designs to be examined according to their technical role without assuming a common protocol. The evidence was organized according to the primary contribution of each study. The resulting categories include command and script generation [1, 3, 5, 8], verification and secure execution, agent and tool interaction [15, 16, 19], execution optimization and evaluation environments [31, 32], security and deception [23, 33–38], developer tooling [39–41], and educational or training environments [42–47]. Some studies contribute to more than one area. Rather than assigning such work repeatedly, the primary category was selected according to the research question under discussion, while secondary contributions were retained during the cross-study analysis. The coding also separates direct evidence from studies that address related components of the execution workflow. This distinction is important because work on general agent security or tool use does not necessarily evaluate shell execution itself. Such evidence is therefore used only where the technical setting provides a meaningful basis for understanding shell-agent behavior.

Interpretation of Quantitative Results

Because the studies used different experimental settings, the numerical results were interpreted in relation to their original evaluation conditions rather than combined into a single overall measure. This approach avoids treating measurements that answer different research questions as directly equivalent. The survey does not combine reported accuracy, task-success rates, latency, speedup, or security measurements into a single aggregate score. The studies differ in datasets, tasks, models, hardware, operating systems, execution environments, baselines, and definitions of success. Consequently, an improvement reported under one experimental configuration cannot automatically be interpreted as equivalent to an improvement reported under another. Quantitative findings are therefore considered together with the conditions under which they were obtained. Command generation metrics describe the characteristics of generated commands or scripts, whereas execution metrics capture behavior after those commands interact with an environment. Security measures may instead concern attack detection, prevention of unsafe actions, policy violations, or attack success. These measures represent different properties and are retained as separate dimensions. The same principle applies to efficiency. Lower latency or reduced token usage does not necessarily correspond to more effective execution if the system requires additional retries or produces more failures. Conversely, an additional verification step may increase processing cost while reducing unsafe actions. The synthesis therefore reports such measurements independently and relates them only when the original evidence supports the comparison.

Evidence Synthesis Across the Execution Workflow

The evidence from different research areas is connected through a common execution model. This model provides the structure used throughout the remaining analysis while allowing individual studies to address only the stages relevant to their contributions. The survey uses an analytical workflow with

seven stages:

1. Goal interpretation
2. Planning and tool selection
3. Command generation
4. Verification
5. Execution
6. Observation and state management
7. Recovery

These stages provide a common analytical structure and do not imply that every reviewed system explicitly implements all seven components. Goal interpretation concerns identifying the intended outcome of the user's request. Planning and tool selection determine the actions and interfaces through which that outcome can be pursued. Command generation then produces the executable shell action or script. Verification forms a control point between proposing an operation and executing it. For example, CARE investigates shell-specific pre-execution analysis using canonicalization, multiple forms of static evidence, and selective escalation for uncertain cases rather than depending entirely on unrestricted execution or an LLM-based judge [9]. Execution moves the proposed action into the target environment, where its behavior depends on conditions such as files, permissions, processes, installed software, environment variables, and network access. The resulting output and state changes provide information for subsequent decisions. Observation and state management determine how this information is retained and used, which becomes particularly important for tasks involving dependent commands. Recovery addresses incomplete, failed, or unexpected execution. Depending on the system, recovery may involve retrying an operation, changing the command, selecting another tool, restoring the state, or revising the plan. Including recovery in the analytical workflow makes it possible to distinguish systems optimized for individual command generation from systems designed to complete multi-step tasks under changing execution conditions.

Cross-Cutting Security and System Controls

Security is considered across the execution workflow rather than assigned to a single stage. This treatment reflects the fact that both threats and protective mechanisms can enter or operate at different points in an agent's interaction with its environment. An external document or repository can introduce malicious instructions before generation; a tool description or skill can influence action selection; and permission settings can determine whether an operation can reach a sensitive resource. Execution isolation can reduce the consequences of unsafe actions [48–53], while monitoring can provide evidence of their effects. The synthesis distinguishes controls according to the point at which they operate. Verification attempts to identify or block an operation before execution. Permission boundaries and sandboxing instead restrict what an executed operation can affect. State validation and recovery address the consequences that become visible after execution. These mechanisms therefore serve different purposes and are not treated as interchangeable forms of protection. Human approval represents another control point. It can introduce a decision before a high-impact operation is executed, but its usefulness depends on whether the proposed action and its possible consequences are presented clearly enough for the reviewer to assess them. The presence of an approval mechanism alone is therefore not treated as evidence of effective security.

Scope of the Synthesis

The final synthesis focuses on recurring technical patterns rather than ranking individual systems. This allows studies with different architectures, tasks, and evaluation settings to be compared according to their assumptions, mechanisms, and evidence. Studies are compared according to the execution stage they address, their operating assumptions, the mechanisms they introduce, and the evidence used to evaluate those mechanisms. This approach is important because shell-agent behavior emerges from interactions among several components rather than from the language model alone. A

stronger model may improve command generation, while the outcome can still depend on the tool interface, verification policy, permissions, execution environment, state representation, and recovery strategy. The survey therefore uses the complete execution workflow as its common analytical structure. The central unit of analysis is consequently the sequence of agent decisions and environment changes involved in completing a task, rather than an individual shell command considered in isolation. This framework provides the basis for connecting command generation findings in Section 3 with the verification, execution, state, security, and recovery mechanisms examined later in the survey.

COMMAND GENERATION, VERIFICATION, EXECUTION, AND RECOVERY

Shell-based agents differ from ordinary language-generation systems because their outputs can directly change a computing environment. A syntactically valid command may still fail because of an incorrect path, unavailable utility, quoting error, unsuitable pipeline, or inaccurate assumption about system state. Bash also introduces expansion, redirection, process, environment, and exit status semantics that make command generation a constrained program-generation problem rather than a simple text-generation task. The reviewed studies address different stages of this process. Some improve how commands are represented or generated, while others introduce checks before execution or mechanisms for execution, monitoring, optimization, and recovery. These capabilities should be considered separately because success at one stage does not establish reliability at the others.

Syntax-Aware and Structured Command Generation

The first challenge is producing an action that reflects both the user’s intent and the rules of the target shell. Several studies have provided models with structural or semantic information that is not available when commands are treated only as token sequences. Bash-Commenter incorporates Bash syntax into preference optimization by constructing preference pairs through controlled modifications of abstract syntax trees. It reports BLEU-4 scores of 33.40% for single-line commands and 22.15% for multi-line scripts [2]. Because the task concerns Bash comment generation rather than execution, these results should not be interpreted as evidence of complete task success. Instead, they show the value of representing shell-specific structure during model optimization. MABash follows a more explicitly decomposed design. Its parsing process separates lexical, syntactic, and semantic analysis and maintains intermediate representations through a shared blackboard before producing the final comment. On 10,592 Bash command-comment pairs, it reports improvements of 1.56% in BLEU-4, 2.85% in ROUGE-L, and 2.73% in metric for evaluation of translation with explicit ordering (METEOR) over the comparison baseline [4]. The relevance to shell agents is the preservation of intermediate structure, which can make relationships between command components more explicit than a flat token representation. BashCoder-R1 extends this idea toward robustness. It combines continued pretraining, long-form supervised fine-tuning, and robustness-aware group relative policy optimization, with rewards based on syntax correctness, robustness checks, and formatting constraints [6]. On BashBench’s 952 tasks, it reports FullRate values of 90.04% for single-line tasks and 73.18% for multi-line tasks. The difference indicates that longer scripts introduce dependencies between actions: an individually valid command can still create a state that prevents subsequent operations from succeeding. Other work applies structural constraints at the target language level. Text2DSL reports an increase in syntactic validity from 80.5% to 99.4% in its GigaChat experiment when grammar, API information, and vocabulary constraints are supplied [54]. Since its target is a policy-oriented domain-specific language rather than Bash, the result supports evidence rather than direct shell-agent evidence. It nevertheless demonstrates how explicit constraints can reduce invalid outputs when the target language has a well-defined structure. Execution-based assessment provides a complementary view. Local NL2Bash executes generated commands inside Docker-isolated environments instead of relying solely on textual similarity to reference commands [7]. Its evaluation covers 300 tasks and reports a success rate of 0.82 for Qwen3-14B-FP8. This approach is important because different commands can achieve the same result, whereas a command that resembles a reference answer may fail under different environmental conditions. Overall, the evidence supports

the use of syntax, intermediate representations, semantic constraints, robust objectives, and execution testing in command generation. However, improvements in language-generation metrics alone cannot establish that an output is safe, executable in the intended environment, or reliable over a multi-step task.

Pre-Execution Verification

Generation produces a proposed action, and verification determines whether that action should proceed. This distinction is particularly important for shell operations because execution can alter files, processes, permissions, configurations, credentials, and external resources. CARE introduces a shell-specific pre-execution verifier based on canonicalization, attribution, and resolution. Instead of sending every command to an LLM-based judge, it first derives deterministic evidence and escalates only uncertain cases. Its reported balanced evaluation achieves 85.64% F1 with a 0.91% false-positive rate and 2.32 ms mean latency, while its static enforcement profile reports 84.99% F1 at 0.34 ms [9]. The design separates command generation from execution authorization and therefore avoids making the generative model the sole judge of its own output. A regular-type system for Unix and Linux shells provides another form of independent analysis. It uses data invariants to reason about shell operations and reports 90% accuracy while identifying 84 bugs missed by the state-of-the-art [10]. Its objective differs from that of CARE, but both approaches demonstrate the usefulness of placing part of the correctness or safety decision outside the generative model. Verification also needs to address more than shell grammar. A command can be syntactically valid while being inconsistent with the user's objective, current permissions, available resources, or execution policy. Consequently, the relevant checks may involve structure, semantics, paths, provenance, permissions, policy, and environmental conditions. The required context also expands with workflow length: a command of a multi-step task may depend on changes introduced by earlier actions. Thus, command-level verification provides an important execution boundary, but it does not by itself establish safety across an entire agent trajectory.

Execution, Monitoring, and Runtime Behavior

Once an action is authorized, its behavior is determined by the actual runtime environment. Files, processes, installed software, permissions, network conditions, environment variables, and prior operations can all produce outcomes that differ from the model's assumptions. Several systems address runtime efficiency and fault handling from different perspectives. Fractal represents shell scripts as data-flow graphs and isolates side effects to support fault-tolerant execution, reporting speedups of up to $9.6\times$ over Bash together with faster fault recovery [11]. INCR avoids recomputing unchanged work by reusing previous results and reports a mean speedup of $34.2\times$, a maximum of $373.3\times$, and 99.9% equivalence in its experiments [12]. hS investigates speculative reordering at subprocess granularity and reports a geometric-mean speedup of $2.6\times$ [55]. These figures should not be treated as a common performance scale because the systems use different workloads, objectives, baselines, and execution models. Their shared relevance is that shell agents may execute long sequences of operations, repeat work, or revisit earlier steps after failure. Runtime optimization can therefore reduce both execution cost and the amount of work that must be repeated, provided that required semantics and side effects are preserved. Monitoring adds information that cannot always be obtained from the command string or exit status. The reviewed evidence includes mechanisms for observing processes and resource behavior, including child processes and continuing activity. Such information becomes important when an agent launches processes or invokes tools whose effects extend beyond the immediate command. For agentic systems, these runtime observations also influence subsequent decisions. A non-zero exit status may indicate a transient failure, a partially completed operation, or an incorrect assumption about the environment. Likewise, missing output does not necessarily identify the cause of failure. The agent must therefore interpret execution state before selecting its next action rather than simply repeating the previous command.

Recovery as an Independent Capability

Recovery deserves separate treatment because successful execution is not guaranteed even when command generation and pre-execution checks are correct. Multi-step shell workflows can encounter missing files, permission problems, unavailable utilities, process conflicts, network failures, malformed output, or state changes caused by earlier operations. The appropriate response depends on the failure. Repeating an action may resolve a transient problem, but it can also amplify an error when the first attempt has already changed the environment. Repeating a file modification after partial completion, for example, can duplicate changes, while repeating a process-management command can affect a different process than intended. Recovery therefore requires knowledge of the attempted action, its observed effect, and the validity of the original plan. Execution infrastructure can support this process. Side-effect isolation, retained intermediate results, and partial re-execution provide mechanisms for restoring or continuing execution [11, 12]. However, these mechanisms should not be confused with the agent's decision-making capability. The infrastructure determines what can be safely repeated or restored; the agent must still determine which recovery action is appropriate. This distinction suggests that recovery should be measured directly rather than being hidden within final task completion scores. Relevant measures include recovery success, recovery time, additional actions, repeated unsafe actions, state divergence, and successful return to a valid execution state. Such measurements can distinguish systems that achieve the same outcome through substantially different execution trajectories.

Cross-Stage Synthesis

The evidence indicates that command generation, verification, execution, monitoring, and recovery represent connected but non-interchangeable capabilities. Generation determines the proposed action, verification provides a decision point before execution, runtime behavior determines its actual effects, monitoring exposes those effects, and recovery addresses deviations from the expected state. This separation has direct implications for system design and evaluation. A more accurate generator does not eliminate the need for execution checks. A verifier cannot correct an incorrect model of the environment, and faster execution does not compensate for poor planning. Similarly, rollback or re-execution infrastructure does not determine whether the agent chooses an appropriate next action. The same distinction should be retained in the evaluation. Generation metrics describe the proposed command or script, execution metrics describe its observed behavior, security metrics capture whether actions satisfy relevant safety constraints, and recovery metrics describe the response to failure. A meaningful evaluation of shell-based agents therefore needs to examine the complete transition from the proposed action to the resulting state and subsequent response. The broader finding is that shell-agent reliability is a system property rather than a property of generated text alone. The language model proposes actions, while verification, runtime controls, state representation, monitoring, and recovery determine how those actions affect the environment and how the system responds to unexpected outcomes. This execution-oriented view provides the foundation for examining the role of tool interfaces and agent architecture in the following section.

TOOL INTERFACES, AGENT ARCHITECTURE, AND SECURITY

A shell-based agent is shaped not only by the language model that generates its actions but also by the interface through which those actions reach the operating system. The interface determines the available operations, argument structure, returned information, and handling of execution errors. These choices affect how an agent decomposes tasks, how often it can inspect intermediate results, and how much authority is exercised by each action. This section examines these system-level factors together with the security mechanisms that constrain them. The discussion distinguishes interface design, agent coordination, information trust, permissions, human authorization, privacy, and runtime containment because each addresses a different part of the execution path.

Tool Interface Design

The design of the tool interface can change agent behavior even when the underlying model remains unchanged. An interface can expose individual operations as separate actions or allow several operations to be expressed through a larger executable unit. The choice affects the interaction cost,

observability, and amount of control available between actions. *The Devil Is in the Interface* reports a 4.7× improvement in consistency with an atomic interface, while a Python-oriented interface reduces the number of required steps by 41% in its reported experiments [13]. These results come from a particular experimental setting and should not be treated as general performance estimates. They nevertheless demonstrate that action representation can influence task decomposition. With an atomic interface, each operation becomes a distinct decision point. This makes individual actions easier to inspect and can provide more opportunities for intervention, but complex tasks may require many model-tool interactions. A scripting-oriented interface has the opposite property: several operations can be expressed in one action, reducing the interaction overhead while increasing the consequences of an incorrect script because fewer intermediate decision points are introduced. Related evidence comes from *Is Bash All You Need?*, which reports approximately 24% higher task scores and 19–72% reductions in token usage for Bash-based interaction in its evaluated enterprise digital-worker tasks [14]. The result indicates that shell-based interaction can compactly represent multi-step procedures, although lower interaction cost does not necessarily imply stronger control. For shell agents, interface design therefore involves a balance among action granularity, interaction overhead, observability, and execution authority. Low-level interfaces provide finer control, whereas higher-level scripting interfaces support more compact workflows. The latter consequently require stronger safeguards around generated scripts, permissions, and execution.

Agent Architecture and Action Decomposition

The next design choice concerns how responsibilities are distributed around the model. An agent may perform planning, generation, verification, and execution within a single loop, or these functions may be separated across multiple components or agents. The latter can introduce additional control points but also create communication paths through which errors can propagate. CoAct-1 combines computer interaction and coding through multiple agent roles and reports a 60.76% success rate on OSWorld with an average of 10.15 steps in its evaluation [17]. PagePilot similarly applies a multi-agent architecture to web automation and reports 76.7% on WebVoyager and 73.1% on general AI assistants (GAIA) Level 1 [18]. Neither system is shell-specific; therefore, these values should not be interpreted as direct Bash-agent results. Their relevance is architectural: task performance can depend on how responsibilities and intermediate decisions are distributed. For shell workflows, role separation can distinguish planning from execution, command generation from verification, or reasoning from permission decisions. Such separation can reduce the direct path between one model output and a system-level action. However, simply increasing the number of agents does not establish better reliability or security. Additional components introduce more communication channels, and an erroneous decision can be transferred from one role to another. The relevant architectural properties are therefore the scope of each role, tool access permissions, information shared between components, and locations of independent checks. An architecture in which every component can invoke the same unrestricted shell provides few meaningful control boundaries. More narrowly scoped roles can instead restrict which components are able to perform high-impact actions.

Security Across the Shell-Agent Workflow

Security risks can enter before a command is generated. Shell agents may process repositories, documents, logs, tool descriptions, skills, command output, environment variables, and other external content. These sources can contain text that resembles an instruction, even when it should be treated as data. If the agent assigns excessive authority to such content, an attacker may influence subsequent decisions without directly modifying the generated command. The reviewed workflow injection research demonstrates this broader attack surface. A GitHub Actions study represents workflows using an intermediate representation and constructs action-workflow dependency graphs for taint analysis. It reports 496 confirmed exploitable cases with 95.6% precision, including 343 reported zero-days [20]. Although this study concerns GitHub Actions rather than shell agents specifically, the underlying problem is relevant to automated workflows in which shell commands form part of the execution pipeline. LogJack examines indirect prompt injections through cloud logs. Its reported experiments

produced an 86.2% hijack rate for Llama 3.3 70B, while the tested Azure Shield configuration defended against one case in 32 trials [26]. This result illustrates that execution feedback cannot automatically be treated as trusted control information. Log contents originate from the environment, yet their textual form can influence an LLM's subsequent reasoning. Tool metadata presents a similar problem. The model context protocol (MCP) poisoning benchmark reports average attack success above 82%, with nearly 100% attack success for GPT-4o in the reported benchmark [22]. Although this evaluates tool description poisoning rather than shell commands directly, the mechanism matters for shell agents because tool descriptions can affect both action selection and request construction. These findings motivate a basic distinction between information used by the agent and instructions that the agent is authorized to follow. Repository contents, logs, command output, and tool descriptions may be essential for completing a task without necessarily having authority to change the agent's control policy. Preserving this distinction is therefore a fundamental part of secure agent design.

Permissions and Skill-Level Controls

Permissions provide an enforcement boundary at the resource level. In shell environments, they can restrict access to files, processes, network connections, package installation, environment variables, credentials, and privileged operating system operations. SkillGuard treats an agent skill as a security principle and separates permissions for contextual information from permissions for performing actions [21]. Its reported evaluation reduces attack success from 35.3% to 20.7%, together with a 6.5% average reduction in task success. This result illustrates the operational trade-off of access control: restricting an agent can prevent harmful actions while also blocking operations required for legitimate tasks. SKILLSHIELD applies security controls at the prompt level and reports a reduction in malware severity from 3.37 to 0.58 after introducing security skills [24]. Prompt level mechanisms can influence behavior before execution, but they do not constitute the same enforcement boundary as operating system permissions. Their effectiveness depends partly on the model following the instructions supplied. For shell agents, this distinction is important because permissions can constrain an action even when the model has selected it incorrectly. A command without authorization to access a resource can be blocked by the execution environment independently of the model's reasoning. This is particularly relevant to credentials, system files, processes, network resources, and persistent configuration.

Human Approval and Action Authorization

Human review introduces another decision point before high-impact operations are executed. However, its effectiveness depends on whether the reviewer can understand the actual operation and its potential consequences. Consent Integrity addresses this problem through a mediator intended to decode action semantics and present them through a trusted path. The reported evaluation detected 90% of abuses with a mean overhead of 0.056 ms [25]. Although the result is specific to the study's evaluation conditions, it highlights an important property of approval mechanisms: the reviewer should authorize the operation that will actually execute rather than an incomplete or misleading description of it. This is particularly difficult for shell commands containing multiple commands, pipelines, substitutions, redirects, scripts, environment-variable expansion, or external tools. An approval interface may therefore need to expose command arguments, affected resources, privilege requirements, and relevant side effects when these can be determined. Human approval should also be viewed as an additional control rather than the sole security mechanism. A reviewer may approve an action without understanding every consequence, especially when the command is complex or the execution environment is unfamiliar. Automated verification, permission boundaries, and isolation can provide protection when human inspection is incomplete.

Privacy and Sensitive Context

Shell execution can expose information that was absent from the original task. Commands may return credentials, configuration values, environment variables, file contents, database results, or

responses from external services. Once incorporated into the agent's context, this information may subsequently appear in another tool call, command, log, or generated response. SlotGuard addresses this problem by creating a local boundary around agent transcripts and replacing sensitive values with typed synthetic slots. Its evaluation reports the removal of 20,814 sensitive characters and 0.0% literal leakage while maintaining a 98% task-success rate, compared with 2.5% for generic redaction [29]. These results are specific to the evaluated setting, but they demonstrate that privacy protection can be applied during context handling rather than only after the interaction has finished. For shell agents, information protection must cover the entire data path. Sensitive content can enter through files, environment variables, command output, logs, or external services and can subsequently leave through generated commands, tool arguments, logs, responses, or later API calls. Protecting only the original prompt, therefore, leaves other parts of the execution workflow exposed.

Execution Isolation and Containment

Isolation addresses a different question from verification or authorization: what can happen if an action reaches execution? A shell agent can be placed inside an environment that restricts its access to host filesystems, networks, processes, credentials, devices, or operating system privileges. The reviewed evidence includes sandboxing, transactional execution, checkpointing, rollback, human intervention, and dynamic defenses as mechanisms for limiting the effects of unsafe or erroneous behavior. The OpenClaw security study evaluated 47 adversarial scenarios and reported human-in-the-loop defense rates of up to 91.5% for Claude in its tested configuration [28]. This finding is specific to the evaluated system and attack set but illustrates how containment and intervention can reduce the consequences of attacks that reach the decision process. Isolation does not answer whether an action was appropriate. A sandbox can prevent an unsafe command from reaching production resources without identifying the command as unsafe, while a permission policy can block one resource but leave another accessible. These mechanisms serve different purposes and are most useful when combined with appropriate verification and authorization controls.

Cross-Layer Security Synthesis

The evidence points to a distributed security boundary rather than a single protective mechanism. Tool interfaces determine what actions are exposed and how they are represented. Agent architecture determines which components make decisions and which can invoke tools. Permission controls restrict resource access, verification examines proposed actions, human approval can provide an additional authorization point, privacy mechanisms regulate sensitive information, and isolation limits the consequences of actions that reach execution. These mechanisms address different failure modes and therefore cannot substitute for one another. A permission boundary does not determine whether malicious instructions entered through a repository or tool description. A prompt level defense does not enforce operating system access restrictions. A verifier may identify a dangerous command without preventing an already authorized process from exposing sensitive information. Similarly, sandboxing can contain an operation without explaining why the agent selected it. For shell-based agents, security is consequently better understood as a sequence of trust decisions distributed across the workflow. The system must determine which information can influence reasoning, which information has instruction-level authority, which tools may be invoked, what resources those tools can access, which proposed operations are permitted, what the runtime can affect, and what information may return to the agent after execution. This view also connects security with the execution mechanisms discussed earlier. An injected instruction can alter command generation; an unsafe command can modify persistent state; and misleading execution feedback can influence subsequent actions. Security evaluation should therefore consider the complete interaction between information, decisions, tools, permissions, runtime effects, and subsequent actions rather than inspecting only the final command.

EVALUATION OF SHELL-BASED AGENTS

Evaluating shell-based agents requires more than checking whether a generated command is valid. The same command can behave differently across environments, and a successful execution may still

violate a security policy, alter an unintended resource, or leave the system in an inconsistent state. For this reason, the reviewed studies evaluate shell-based systems at different levels, including generation quality, executable task completion, runtime behavior, security, recovery, and efficiency. These evaluation settings answer different questions and should not be collapsed into a single measure. A generation model evaluated against a fixed command dataset and an autonomous agent operating inside an isolated environment are exposed to different tasks, interfaces, environments, and failure conditions. Reported scores must therefore be interpreted with their experimental conditions rather than as universal measures of shell-agent capability.

Command and Generation-Level Evaluation

Generation-level evaluation examines the properties of the command or script before it is considered a complete execution outcome. Typical measures include syntax validity, functional correctness, and similarity to reference outputs. BashCoder-R1 evaluates Bash generation on BashBench, which contains 952 single-line and multi-line tasks [6]. It reports SyntaxPass values of 100% for single-line tasks and 94.97% for multi-line tasks. The functional rate is reported as 93.01% for single-line and 93.85% for multi-line generation, while the full rate is 90.04% and 73.18%, respectively. The separation between functional and full success is useful because a generated command may perform part of the requested operation without satisfying the complete task. Text2DSL provides another syntax-oriented evaluation. After incorporating grammar, API, and vocabulary constraints, its reported syntactic validity increased from 80.5% to 99.4% [54]. This type of measurement determines whether an output conforms to a formal target structure, but it does not establish that the resulting system state is correct. MABash evaluates Bash command understanding and generation using 10,592 command-comment pairs and reports improvements in BLEU-4, ROUGE-L, and METEOR over its evaluated baselines [4]. These metrics measure the similarity between generated and reference texts. Such similarity is useful for assessing language-generation behavior, but it does not establish functional equivalence. Multiple shell commands can achieve the same objective, while a command that resembles a reference may behave differently under another environment. Generation metrics are therefore most informative when tied to the capability they were designed to measure. They provide evidence about the quality or validity of proposed commands, but they do not by themselves establish successful or safe agent execution.

Execution-Based Task Evaluation

Execution-based evaluation tests generate commands in an actual or isolated runtime and measure whether the intended operation is completed. This moves the assessment from the properties of the output text to the behavior produced by executing it. The Local LLMs for NL2Bash study evaluated 22 open models on 300 Docker-isolated tasks [7]. Qwen3-14B-FP8 achieved a reported success rate of 0.82 in this setting. The use of isolated environments allows the evaluation to capture failures caused by unavailable packages, directory assumptions, incompatible command options, permissions, and other runtime conditions that cannot be reliably inferred from textual similarity. Execution systems also introduce performance measures that are distinct from language model generation quality. Fractal reports speedups of up to $9.6\times$ for its fault-tolerant shell-script distribution system and evaluates recovery behavior within its execution environment [11]. INCR reports a mean speedup of $34.2\times$, a maximum of $373.3\times$, and 99.9% equivalence in its evaluated workloads [12]. These results concern execution system behavior rather than command generation quality and should consequently be interpreted as a separate evaluation category. An execution score is therefore meaningful only in conjunction with the environment in which it was obtained. Shell type, operating system, installed software, filesystem state, permissions, network access, and runtime configuration can all influence the outcome.

Security Evaluation

Functional success and security are separate evaluation dimensions. An agent can complete a requested task while simultaneously executing an unauthorized operation, exposing sensitive information, or violating a security policy. CARE evaluates pre-execution verification for shell-

executing LLM agents [9]. On its main balanced split, it reports 85.64% F1, a 0.91% false-positive rate, and 2.32 ms latency. Its static profile reports 84.99% F1 with 0.34 ms latency. The study also reports a realized-harm rate of 37.33% on RedCode-gen before verification is applied. These measurements assess the ability to identify potentially harmful commands rather than whether an agent completes a task. Security risks can also originate outside the final command. The reviewed studies examine prompt injection, tool description poisoning, malicious logs, skill-level attacks, and workflow manipulation [20, 22, 26]. Adversarial input may influence the agent’s reasoning and tool selection before an unsafe command is produced. Inspecting only the final command can therefore miss the mechanism through which the attack entered the workflow. Security evaluation should consequently distinguish between action-level security and workflow-level security. Action-level evaluation examines whether an individual operation violates a defined policy. Workflow-level evaluation examines whether untrusted content can alter the agent’s trajectory and eventually lead to an unauthorized or harmful outcome.

State and Long-Horizon Evaluation

Multi-step shell tasks introduce another evaluation problem: the environment changes as execution progresses. A command that is appropriate at one point may become invalid after an earlier operation changes a file, process, dependency, or configuration. State-oriented evaluation should therefore examine whether the agent maintains an accurate view of the environment throughout the trajectory. Relevant failures include stale observations, incorrect file or directory assumptions, missing dependencies, inconsistent intermediate states, and continued execution after a preceding operation has failed. The reviewed evidence includes mechanisms for maintaining, isolating, or restoring execution state. Fractal studies fault-tolerant shell-script distribution and recovery [11], while related execution systems use mechanisms such as checkpointing, rollback, and transactional behavior. However, the 57-study evidence set does not provide a common state consistency metric that allows these systems to be compared directly. This is a significant limitation of final task completion measures. A trajectory may eventually reach the requested state even after several intermediate errors or unsafe operations. Evaluation of long-horizon shell agents should therefore preserve information about intermediate states and transitions rather than recording only the final result.

Recovery Evaluation

Recovery measures what happens after the initial execution path fails. A capable recovery mechanism must identify the nature of the failure, determine whether the existing state remains usable, and select an appropriate next action without repeatedly producing the same error. Final task completion does not reveal how recovery occurred. Two systems can reach the same final result even though one required several retries, consumed substantially more tokens, or temporarily performed unsafe operations. A more informative recovery evaluation should report recovery success, additional actions, recovery time or token cost, and whether the recovery itself introduces a functional or security failure. Repeated execution of an unsafe action should be distinguished from a safe retry of a transient failure. The reviewed literature does not yet provide a common recovery benchmark for shell-based agents. This makes comparison difficult for systems using different combinations of retries, rollbacks, checkpointing, partial re-execution, or replanning.

Efficiency Evaluation

Efficiency can be measured through execution time, model latency, token consumption, number of tool calls, resource usage, and related system costs. These measures become important when an agent can solve the same task through either a compact script or a long sequence of individual interactions. Interface studies report reductions in interaction steps and token consumption under particular interface designs [13, 14]. Fractal and INCR provide execution-level efficiency measurements through reported speedups [11, 12]. These results illustrate that efficiency can be improved at different points in the system, from the model-tool interaction layer to the underlying execution engine. However, efficiency should be reported alongside the conditions under which it is obtained. Fewer

tool calls may reduce latency and token consumption while also reducing opportunities to inspect intermediate results or apply verification. Similarly, faster execution does not establish that the resulting actions are correct or safe. Therefore, efficiency is best interpreted together with functional, state, and security outcomes.

Reproducibility and Evaluation Conditions

Differences in experimental configuration are a major source of variation across shell-agent evaluations. Relevant conditions include the language model and version, prompts or system instructions, shell implementation, operating system version, installed utilities and package versions, filesystem state, network access, tool interface, permissions, sandbox configuration, task set, and definition of success. The reviewed studies used substantially different evaluation environments and protocols. Some relied on Docker-isolated execution, while others evaluated complete execution systems or specialized benchmarks [9, 7]. Therefore, a numerical result from one setting cannot be interpreted independently of the conditions that produced it. Reproducible evaluation should provide sufficient information to recreate these conditions. Fixed task specifications, containerized environments, explicit model versions, documented randomness settings, and publicly available evaluation scripts can reduce ambiguity. Where proprietary models or execution environments are required, their limitations should be reported explicitly.

A Unified Evaluation Framework

The evidence can be organized into six complementary evaluation dimensions: generation, execution, security, state, recovery, and efficiency. Generation measures whether the agent produces syntactically valid and functionally appropriate commands. Execution measures whether those commands achieve the intended task in an actual runtime. Security examines unauthorized actions, policy violations, attacks, harmful behavior, and information leakage. State evaluates whether the agent correctly tracks changes in the execution environment. Recovery measures the response to failures and deviations from the expected execution path. Efficiency captures the time, tokens, tool calls, and computational resources required to achieve the outcome. Reproducibility applies across all six dimensions. Each result should identify the model, task set, environment, interface, permissions, evaluation protocol, and other conditions that materially influence the measurement. These dimensions should remain separate rather than being combined into a single score. Syntax validity, task completion, security detection, state consistency, recovery behavior, and execution efficiency represent different properties of an agent. Aggregating them into one value would conceal important differences in system behavior and would not resolve the underlying variation between evaluation settings.

Implications for Future Benchmarks

Future shell-agent benchmarks should evaluate executable tasks rather than relying primarily on command-text comparison. Each benchmark task should specify the intended objective, available tools, execution environment, permissions, expected state, and relevant security constraints. The evaluation should retain the complete trajectory, including generated commands, tool calls, execution outputs, state changes, failures, recovery attempts, and outcomes. This would allow researchers to determine not only whether the final objective was achieved but also how the system reached that result. Adversarial cases should be evaluated alongside ordinary tasks. Functional tasks can measure normal execution capability, while adversarial scenarios can test prompt injections, malicious repository content, misleading logs, poisoned tool descriptions, excessive permissions, and sensitive-data exposure. Reporting both categories would make it possible to distinguish functional capability from behavior under hostile conditions. The reviewed evidence indicates that current evaluation is distributed across different research objectives. Generation studies primarily measure command quality, systems research emphasizes execution and efficiency, and security research focuses on attacks and defensive mechanisms. A stronger benchmark should connect these perspectives without collapsing them into a single score. The resulting evaluation would characterize shell-agent behavior

through executable correctness, controlled resource use, security, state consistency, recovery, efficiency, and reproducibility.

CROSS-STUDY FINDINGS AND RESEARCH GAPS

The 57 studies reviewed in this survey address different aspects of shell-based agents, from command generation and verification to execution, security, recovery, and tool interaction. Despite this diversity, the evidence points to several recurring patterns. In particular, agent behavior depends on the interaction between the language model and the surrounding execution system, including interfaces, runtime state, verification, permissions, and recovery mechanisms. The same model can therefore exhibit different behaviors when these surrounding conditions change. The following findings consolidate the evidence across the reviewed studies and identify the main gaps that remain open.

Shell-Agent Capability Is a System Property

Generation-focused studies demonstrate substantial improvements in producing syntactically valid and functionally useful shell commands. Structured representations, additional training signals, and robustness-oriented methods improve the generation quality on their respective benchmarks [4, 6, 39]. However, these results measure only one stage of the overall execution process. Once a command is executed, its outcome depends on the state of the environment. Files may have changed, packages may be unavailable, processes may already be running, permissions may differ, or an earlier action may have failed. Consequently, the same command can produce different results under different runtime conditions. Evidence from execution and systems research supports this broader view. Changes to execution strategies, fault tolerance, and script distribution can improve task-level behavior without modifying the underlying language model [11, 12]. Tool interface studies report similar effects when the available action primitives are changed [13, 14]. These results suggest that model capability and shell-agent capability should be analyzed as related but distinct factors. Improvements in command generation do not by themselves characterize the behavior of the complete execution system.

Tool Interfaces Shape Agent Behavior

The tool interface determines how an agent interacts with the environment and where the control points occur during execution. Studies comparing different interfaces show changes in interaction length, consistency, and task execution behavior [13, 14]. An atomic interface can expose individual operations to the agent and create additional opportunities for observation or intervention. A higher-level interface can reduce the number of interactions by grouping several operations into a larger action. This creates a practical trade-off between interaction efficiency and control granularity. More granular interfaces may provide better visibility into intermediate actions but increase the interaction overhead. Larger action abstractions can shorten workflows while reducing the number of intermediate control points. Therefore, interface design affects not only efficiency but also verification, monitoring, and security. This distinction is important when comparing agents that use the same underlying model. An observed improvement may result from the model itself, the action representation, the execution harness, or their combination. Experimental reports should therefore identify these components explicitly rather than attributing the entire improvement to model capability.

Security Extends Across the Agent Workflow

The reviewed security studies show that shell-agent attacks can originate outside the generated command itself. Untrusted repositories, logs, tool descriptions, skills, external documents, and execution outputs can all provide information that later affects agent decisions [20, 22, 24, 26]. Therefore, the relevant security path extends across multiple stages:

Untrusted input → interpretation → planning → command generation → verification → execution
→ observation → subsequent action

A control applied at one stage does not necessarily protect the remaining stages. Command verification can prevent a harmful command from reaching the runtime [9], but it may not detect an earlier manipulation of the agent's planning context. Permission controls can limit the consequences of an action while leaving the underlying decision process unchanged. This distinction supports a broader evaluation model in which security is examined at both the action and workflow levels. Such an evaluation should consider whether untrusted information changes the agent's trajectory, whether alternative tools provide a route around a blocked action, and whether the agent repeatedly attempts an unsafe operation after an initial rejection.

Verification Requires Runtime Context

Pre-execution verification provides an important control point because unsafe actions can be rejected before they affect the environment. CARE illustrates this approach through static-first analysis followed by selective neural adjudication [9]. However, the safety of a command cannot always be determined from its text alone. A command may be acceptable in one state and problematic in another because its behavior depends on files, processes, permissions, external services, or outputs generated by earlier actions. Verification is therefore more useful when connected to execution state and policy rather than treated as an isolated classifier. The limitations of fixed command lists provide a related example. CmdNeedle reports that 69.0–98.6% of the 1,709 evaluated real-world command denylists were vulnerable to bypasses in its tested setting. This result does not establish that all command-gating mechanisms fail, but it demonstrates that exact command matching can leave semantic and compositional gaps. Security evaluation should consequently test whether unsafe behavior can be expressed through alternative commands or action sequences rather than relying only on known blocked strings.

State Tracking and Recovery Are Underdeveloped

The literature contains substantially more standardized evidence for generation and final task completion than for state consistency and recovery. During a long-running shell task, an agent may need to track modified files, installed packages, running processes, failed commands, permission changes, and other environmental effects. Its original plan may become invalid as the environment changes. Recovery therefore requires more than retrying a failed command. The agent must identify the likely cause of failure, determine how the environment has changed, update its state or plan, and continue without introducing additional errors. A failure caused by syntax requires a different response from one caused by missing resources, insufficient permissions, or an inconsistent runtime state. The reviewed studies provide examples of checkpointing, rollback, and fault-tolerant execution [11], but there is no common recovery metric across the 57-study evidence set. This limits direct comparison of systems that claim recovery capabilities and leaves open the question of how recovery quality should be measured consistently.

Evaluation Remains Fragmented

Evaluation practices differ substantially across research objectives. Generation studies commonly use syntax validity and reference-based measures. Execution studies emphasize task success, functional correctness, latency, or resource use. Security studies measure attack success, harmful actions, leakage, or policy violations, while systems studies may emphasize throughput, speedup, or fault tolerance. These metrics answer different questions and should not be collapsed into a single quality measure. High command similarity does not establish safe execution, while high task completion does not demonstrate that all intermediate actions were authorized. Similarly, faster execution may reduce opportunities for verification or intervention. A common evaluation structure can therefore improve comparability without forcing these properties into one score. Process-level evidence is particularly important because the final task outcome does not reveal which commands were executed, which failures occurred, how the agent recovered, or whether unsafe intermediate actions were attempted.

Model and Harness Effects Need Controlled Attribution

Another recurring limitation is the difficulty in identifying the source of an observed improvement. Changes to a shell agent may involve the language model, prompting strategy, tool descriptions, execution loop, verification module, permissions, or recovery mechanism simultaneously. This problem is more pronounced in multi-agent systems, where adding agents can change planning, context, communication, tool access, and verification simultaneously. Without matched configurations, an improvement cannot be reliably attributed to a particular component. Future experiments should therefore separate model and system effects through controlled comparisons. One useful design is to keep the language model fixed while varying the execution harness, followed by experiments that keep the harness fixed while changing the model. The same principle can be applied to tool interfaces, verification, and recovery modules. Such controlled studies would provide clearer evidence about which components contribute to observed gains.

Evidence Is Concentrated in Bash and Linux Environments

The reviewed evidence is concentrated around Bash, Linux, software development, coding tasks, and related command-line workflows. This provides detailed coverage of an important class of shell-agent applications but limits the extent to which the findings can be generalized. PowerShell and Windows administration receive substantially less attention, while heterogeneous enterprise infrastructure, cloud environments, container orchestration, and domain-specific command languages are also less consistently represented. Similar limitations appear for non-development applications such as scientific experimentation, data engineering, system administration, and operational automation. This environmental concentration matters because shell semantics, available utilities, permission models, error behavior, and state management differ across platforms. A method evaluated successfully in a controlled Linux environment may therefore require additional validation before its behavior can be assumed to transfer to heterogeneous infrastructure.

Safety, Correctness, and Recovery Interact During Execution

Functional correctness, security, and recovery are often evaluated as separate properties, even though they can interact during the same task. An agent may recover from a failed command by selecting an alternative action that completes the task but requires broader permissions than originally allowed. The task may therefore be functionally successful while violating the execution policy. The opposite trade-off can also occur. An agent that refuses a potentially unsafe action may satisfy a security requirement while failing to complete the requested task. Treating either outcome as a single success or failure hides the underlying behavior. Future evaluations should consequently report functional outcomes, security compliance, and recovery behavior together while keeping them as separate dimensions. The objective is not to construct a universal score, but to expose the conditions under which an agent succeeds, fails, refuses, or recovers.

Research Gaps

The cross-study evidence identifies six major gaps. First, there is no widely standardized benchmark for evaluating shell agents as complete execution systems. Existing benchmarks generally emphasize specific components such as command generation, software engineering, security testing, or task completion. A broader benchmark should combine executable multi-step tasks with realistic state changes, normal and adversarial inputs, explicit permission policies, and recovery requirements. Second, session-level security remains insufficiently evaluated. Command-level verification is useful, but it does not capture whether untrusted information can influence planning, tool selection, later command generation, or recovery. Future security benchmarks should therefore track information flow across the full interaction rather than evaluating only the final action. Third, state-aware recovery lacks a common evaluation methodology. A useful recovery benchmark should measure whether the agent identifies the cause of failure, updates its representation of the environment, restores or adapts the execution state, and continues safely. Recovery success should be reported together with additional actions, execution cost, state consistency, repeated failures, and unsafe recovery behavior.

Fourth, broader environment coverage is needed. Future studies should extend beyond Bash and Linux to PowerShell, Windows, heterogeneous Linux distributions, containers, cloud infrastructure, enterprise environments, and domain-specific command languages. Such coverage would help determine which findings reflect general properties of shell agents and which depend on a particular operating environment. Fifth, model-harness attribution requires controlled experiments. Future evaluations should vary language models, prompts, tool interfaces, verification modules, multi-agent coordination, runtime controls, and recovery mechanisms. Matched configurations would make comparisons more informative than experiments in which several components change simultaneously. Sixth, process-level reproducibility remains limited. A final task-success value provides little information about how an agent reached its outcome. Future benchmarks should preserve commands, observations, failures, retries, state changes, verifier decisions, permission events, and human interventions in replayable execution traces. This would allow researchers to analyze not only whether a task succeeded, but also the sequence of decisions and environmental changes that produced the result. Overall, the reviewed evidence points toward a shift from component-level testing to controlled, executable, and traceable evaluation of shell-based agents. Generation, tool interfaces, verification, permissions, runtime state, security, execution, and recovery should be studied as interacting parts of the same system while retaining separate measurements for each property. This approach would provide a more complete view of shell-agent behavior than command-level evaluation alone.

FUTURE DIRECTIONS

The reviewed evidence suggests that shell-based agents are entering a stage where reliable execution matters as much as command generation. Future progress will depend on making these systems robust under changing states, constrained permissions, untrusted inputs, and long-running workflows. This requires improvements not only in models but also in execution environments, control mechanisms, recovery strategies, and evaluation infrastructure.

Unified Evaluation in Realistic Execution Environments

Future benchmarks should evaluate shell agents through complete executable workflows rather than isolated command generation tasks. A representative task may require the agent to interpret a goal, select tools, generate commands, inspect results, modify its plan, and recover from intermediate failures. The environment should be controlled sufficiently to permit reproduction while retaining realistic sources of variation, such as changing files, unavailable resources, permission constraints, and service failures. Both routine and adversarial tasks should be included. Routine tasks can measure the practical completion of legitimate workflows, whereas adversarial cases can introduce unsafe instructions, malicious repository content, misleading logs, poisoned tool descriptions, permission misuse, and sensitive-data exposure. Evaluation should preserve the trajectory rather than only the outcome, including commands, tool calls, observations, failures, retries, verification decisions, and final environment state. Recent terminal benchmarks provide useful foundations for this direction. TUA-Bench evaluates general-purpose terminal use across 120 real-world tasks using deterministic setup procedures and execution-based scoring [56]. TerminalWorld derives tasks from 80,870 real terminal recordings and provides a verified set of 200 tasks across 18 categories [57]. Future benchmarks can build on these approaches by adding explicit measurements for state consistency, security, recovery, and execution cost.

Session-Level Security and Authorization

Security evaluation should follow information and actions across the complete agent session. An attack may begin with a repository file, log entry, tool description, skill, external document, or command output and influence an action several steps later. Measuring only whether the final command is classified as unsafe can therefore miss the earlier stages through which the attack propagated. Future evaluations should distinguish at least three outcomes: whether untrusted information reached the agent's decision process, whether it changed the resulting behavior, and

whether that change produced an actual security consequence. This distinction would help separate attempted manipulation from successful exploitation. Authorization should be treated as a complementary runtime control. Instead of relying on the model to enforce its own restrictions, the execution environment should enforce explicit permissions over concrete resources and operations. An agent could, for example, read files within a designated directory while being prevented from accessing credentials, modifying system configuration, terminating unrelated processes, or creating unauthorized network connections. The model can determine whether an operation is relevant to the task, while the runtime determines whether that operation is permitted.

State-Aware Recovery for Long-Horizon Tasks

Recovery deserves more explicit treatment because shell environments change as actions are executed. Future systems should maintain a representation of the relevant environment state and update it after successful actions, failures, and externally caused changes. Benchmarks should deliberately introduce failures such as missing files, changed configurations, failed installations, unavailable services, permission errors, and incorrect assumptions. The evaluation should then determine whether the agent can identify the cause, revise its state representation, alter its plan, and continue without creating additional problems. Recovery should be measured independently of final task completion. Useful measures include recovery success rate, additional actions, latency, token consumption, state consistency, repeated failure rate, and unsafe actions during recovery. These measures would distinguish deliberate recovery from repeated retries that eventually happen to produce a successful outcome.

Human Oversight and Risk-Sensitive Approval

Human intervention is likely to remain relevant for operations with substantial system impact. Future interfaces should provide enough context for a reviewer to understand what an action will modify before approval. This may include the target resources, required permissions, expected side effects, relevant dependencies, and uncertainty associated with the proposed operation. Approval itself should become an evaluation variable. Studies can measure whether users correctly identify risky actions, how much information is required for accurate decisions, and whether complex shell commands make review more difficult. Requiring approval for every operation is unlikely to be practical. A more scalable design is risk-sensitive approval, where routine low-impact actions can proceed automatically, while operations involving sensitive resources, elevated permissions, destructive changes, or external communication receive stronger reviews.

Cross-Platform and Heterogeneous Execution

The current evidence base is concentrated around Bash and Linux; therefore, future work should test whether the observed capabilities transfer across different execution environments. Relevant settings include PowerShell and Windows administration, heterogeneous Linux distributions, macOS, containers, cloud infrastructure, and domain-specific command languages. Cross-platform studies should preserve the conceptual objective of a task while adapting commands and tools to the target environment. This would make it possible to examine which capabilities remain stable when shell syntax, available utilities, permission models, error messages, and execution semantics change. The same principle can be extended to enterprise environments, where a workflow may span local shells, containers, remote systems, cloud services, and internal tools. Such settings provide a stronger test of whether an agent can maintain a consistent execution state across heterogeneous resources.

Controlled System Design and Component Attribution

Future experiments should make the relationship between the model and the surrounding execution system explicit. A shell agent may contain a language model, planning strategy, memory mechanism, tool interface, verifier, permission layer, runtime, retry policy, and recovery module. Changing several of these components simultaneously makes it difficult to determine which change produced an observed result. Controlled experiments can address this problem by holding the model constant while changing the execution harness, followed by experiments that hold the harness constant while

changing the model. Similar controls can be applied to verification, tool interfaces, memory, and recovery. Reporting these configurations in detail would make experimental comparisons more interpretable and reduce the risk of attributing system-level improvements solely to model changes.

Replayable Execution and Reproducibility

Shell-agent experiments are particularly sensitive to environmental conditions. Differences in operating system versions, package installations, filesystem contents, network availability, shell versions, and resource limits can change the result of the same command sequence. Future research should therefore provide reproducible execution environments through containers or virtual machines whenever feasible. Experimental descriptions should specify the operating system, shell, and package versions, model version, prompts, tool definitions, filesystem state, network conditions, resource limits, and execution timeouts. Evaluation scripts and task initialization procedures should also be made available where possible. Reproducibility should extend to the execution trajectory. A replayable record should preserve the initial environment, user instruction, model outputs, tool calls, commands, execution results, state changes, verifier decisions, permission events, failures, recovery actions, and final state. Such traces would allow researchers to reproduce failures and investigate why two systems reached different outcomes even when their final task scores were similar.

Integrated Safety, Correctness, and Recovery

Future benchmarks should evaluate safety, functional correctness, and recovery within the same execution trajectory while retaining them as separate measurements. These properties can produce different outcomes for the same task. An agent may complete the requested operation by violating a permission policy, refuse an operation that could have been completed safely, or recover from an error by introducing an unintended state change. A useful evaluation record should therefore capture functional outcome, authorization compliance, state consistency, recovery behavior, and execution cost. Keeping these dimensions separate makes the trade-offs visible without reducing agent behavior to a single score.

Toward Reliable Shell Execution Systems

The broader direction emerging from the reviewed studies is a change in the unit of analysis. For simple tasks, command generation may be a sufficient research question. However, for current shell agents, the command is one step within a longer process involving planning, tool interaction, verification, execution, observation, and recovery. Future systems should consequently be designed around the complete execution loop. A capable agent should be able to interpret a goal, select appropriate tools, generate valid actions, verify operations that require control, execute them within defined permissions, observe the resulting environment, detect deviations, and recover when the original plan no longer applies. The central challenge is to maintain this behavior as workflows become longer, environments become more heterogeneous, and inputs become less trustworthy. Progress will therefore depend on combining language model research with systems engineering, security controls, execution-aware benchmarks, and reproducible evaluation. The resulting systems should be assessed not only by whether they eventually complete a task, but also by whether they reach the intended state through a controlled, observable, and reproducible execution process.

LIMITATIONS OF THE SURVEY

This survey should be interpreted in light of several methodological and evidentiary limitations. First, the analysis covers a defined set of 57 studies spanning shell scripting, terminal-based agents, agentic workflows, security, execution systems, developer tools, and related evaluation settings. Although this provides broad coverage of the topic, it does not constitute an exhaustive record of every publication, system, or implementation related to shell-based agents. The findings therefore represent a synthesis of the selected evidence rather than a complete account of the field. A second limitation arises from the substantial variation among the reviewed studies. Second, the studies differ in their research objectives, datasets, models, tools, execution environments, and evaluation protocols.

Some measure command generation, whereas others investigate execution, security, tool interfaces, or developer workflows. As a result, numerical results reported by different studies are not directly interchangeable. The survey consequently preserves study-specific measurements rather than combining them into a single aggregate score. The survey also relies on results reported by the original studies rather than independently reproducing their experiments. Differences in hardware, software, and package versions, model versions, prompts, runtime configurations, and implementation details may influence reported performance. When these conditions are not fully specified in the source studies, their individual effects cannot be separated by the survey. The synthesis should therefore be understood as an analysis of the reported experimental evidence rather than an independent replication of each result. Another limitation concerns the distribution of the available evidence across execution environments. A substantial portion of the reviewed work focuses on Bash and Linux-oriented settings. Although the survey includes studies involving broader agentic workflows and other environments, equivalent evidence is not available across all shells, operating systems, and infrastructure configurations. Conclusions drawn from the reviewed studies should therefore not be assumed to transfer directly to PowerShell, Windows, macOS, cloud infrastructure, or other command environments without additional evaluation. The evidence base also includes some studies that are not exclusively focused on shell agents but address problems directly relevant to shell-agent workflows. These studies were considered when their findings informed areas such as tool use, workflow injection, permissions, state management, security, and multi-agent execution. Their results must nevertheless be interpreted within their original experimental context. A finding obtained in a broader agentic or software engineering setting should not automatically be treated as a direct measurement of Bash-agent performance. The organization of literature introduces further limitations. The taxonomy used in this survey provides a structured way to examine the execution workflow, but the boundaries between categories are not always discrete. A single study may simultaneously address generation, verification, permissions, execution, security, and recovery. Assigning a primary category simplifies these relationships and is useful for synthesis, but it cannot fully represent the dependencies between research areas. Finally, the field is developing rapidly. New terminal benchmarks, agent architectures, execution frameworks, security techniques, and evaluation protocols continue to emerge. The results presented here should therefore be viewed as a structured snapshot of the evidence available within the defined review scope rather than as a permanent characterization of shell-based agents. Despite these limitations, the reviewed studies provide a useful basis for examining shell agents as execution systems rather than only as command generation models. The limitations also indicate where stronger evidence is needed, particularly through broader environmental coverage, controlled experimentation, reproducible execution settings, and evaluation methods that capture the complete execution process.

CONCLUSION

Shell-based agents are increasingly being developed as complete execution systems rather than as tools for producing isolated commands. A typical workflow may begin with interpreting a task, followed by planning, tool interaction, execution, observation, verification, and, when necessary, recovery. The 57 studies examined in this survey indicate that this broader workflow changes the way terminal agents need to be designed and assessed. Command generation remains central, but the quality of an agent cannot be determined from the generated command alone. Several approaches have improved command generation performance. Syntax-aware methods, structured representations, additional training signals, and multi-stage generation have each been used to improve different aspects of shell command construction. However, their benefits have clear limitations. A command may satisfy syntactic requirements or resemble a reference command and still fail when executed in a real environment. Shell operations interact with files, processes, dependencies, permissions, environment variables, and other mutable resources. Consequently, the surrounding execution state can be as important as the command itself when determining whether an operation succeeds. Reliability also depends on what happens after a command has been generated. Pre-execution verification can screen potentially harmful operations before they reach the runtime, while fault-tolerant execution and system-level mechanisms can support continued operation and recovery

following failures. These mechanisms serve different purposes and should not be viewed as substitutes. A verification layer may prevent a particular unsafe action, but it cannot account for every runtime condition. Similarly, an operation that completes successfully does not necessarily indicate that the agent maintained a correct system state or responded appropriately when an earlier step failed. The interface through which an agent interacts with the shell is another important design factor. Fine-grained tool operations and higher-level scripting interfaces provide different trade-offs in interaction cost, observability, and control. Agent architecture introduces similar considerations. Multi-agent systems can assign planning, execution, verification, and interaction to separate components, potentially distributing complex responsibilities across specialized roles. This division alone does not establish greater reliability or security. The resulting behavior depends on how information is shared, which agent can access particular tools, how permissions are assigned, and where responsibility for verification is placed. Security concerns extend beyond the commands produced by the model. The execution trajectory can contain several other sources of risk. Repository files, logs, tool descriptions, agent skills, task inputs, contextual information, and external instructions may affect subsequent decisions. An apparently harmless command can therefore become part of an unsafe workflow when its surrounding context has been manipulated or contains untrusted information.

For this reason, shell-agent security needs to cover the complete interaction sequence rather than only the final command. Permission boundaries, isolation, verification, privacy mechanisms, and human approval address different stages or failure modes within that sequence. Treating them as complementary controls provides a more realistic security model than assuming that any single mechanism is sufficient. Evaluation is another area where literature remains difficult to compare. Command generation studies often focus on syntax or similarity to reference commands, whereas execution-oriented work may report task completion, runtime behavior, or system performance. Security research typically examines attacks, harmful actions, or policy violations. These metrics describe different properties of an agent and should not be collapsed into one overall performance measure. A system may produce commands that closely match references while failing during execution or complete a task while violating a security constraint. A useful evaluation therefore needs to examine several dimensions together, including command quality, functional correctness, security, state consistency, recovery behavior, efficiency, and reproducibility. Recent work on terminal-oriented agents further points toward evaluation under realistic runtime conditions. Final task success provides only a partial view of agent behavior because important failures may occur during intermediate steps and may not be visible from the final output. Replayable execution traces and process-level evidence can expose these intermediate decisions, state changes, tool interactions, and recovery actions. Such evidence is particularly useful when investigating failures that arise from changing environments or from interactions between multiple components. Taken together, the findings address the central research questions of this survey. Improvements in shell command generation are evident, but reliable terminal execution requires capabilities beyond the language model itself. Verification and runtime safeguards can reduce specific risks, while recovery and state-aware execution remain less consistently addressed. Tool design, permission boundaries, and environmental conditions can substantially alter system behavior. Security must also be considered throughout the workflow because potentially harmful information can enter at several points before an action is executed. At the same time, differences in benchmarks, environments, metrics, and experimental conditions continue to limit direct comparison across studies. These observations suggest several priorities for future research. Benchmarks should evaluate executable tasks under realistic and changing environments rather than relying only on static command generation datasets. Execution traces should be retained so that intermediate decisions and failures can be examined and reproduced. Evaluation should distinguish model-level capabilities from the safeguards and infrastructure provided by the surrounding agent system. More systematic approaches are also needed for maintaining state consistency, handling partial failures, recovering from unexpected changes, and enforcing permissions throughout an execution session. Broader coverage of shells, operating systems, tools, and deployment conditions would further improve the external validity of reported results. The resulting

perspective changes the basic unit of analysis. Instead of asking whether an agent generated the right command, evaluation should ask whether the complete workflow produced the intended outcome under the permitted conditions. A dependable shell agent must interpret the task correctly, choose suitable actions, operate within defined permissions, verify important steps, respond to environmental feedback, maintain an accurate representation of system state, and recover appropriately when execution does not proceed as expected. It must also limit unintended side effects and provide sufficient evidence for its behavior to be inspected and reproduced. Considering these capabilities together provides a more meaningful basis for studying shell-based agents as practical execution systems.

REFERENCES

1. Bin Y, Yuan X, Zeng H, Ye W, Shao W, Qian C, et al. Terminal agents: a survey of AI agents in command-line environments [Preprint]. 2026. arXiv:2608.20485.
2. Jiang L, Huang S, Wu X, Li Y, Zhang D, Wei F. VisCodex: unified multimodal code generation via merging vision and coding models. International Conference on Learning Representations (ICLR). 2026.
3. Liu Y, Zhang W, Yang Z, Zhang Z, Feng H, Wang X, et al. CARE: pre-execution command verification for shell-executing LLM agents [Preprint]. 2026. arXiv:2607.21642.
4. Huang Z, Dundar R, Xie Y, Kallas K, Vasilakis N. Fractal: fault-tolerant shell-script distribution. 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26), Renton, WA, USA. 2026. p.2339-54.
5. Wang S, Hou X, Liu Z, Zhao Y, Cheng X, Zou Q, et al. Demystifying and detecting agentic workflow injection vulnerabilities in GitHub Actions [Preprint]. 2026. arXiv:2605.07135.
6. Yu L, Zhang J, Wang X, Yang L, Zhang F, Wang P, et al. Bash-commenter: leveraging syntax-aware preference optimization to reinforce large language model for bash code comment generation. Proc ACM Softw Eng. 2026;3:2117-40. doi:10.1145/3808102.
7. Jiang W, Zhang X, Zhai J, Ma S, Shen C, Liu Y. False friends in the Shell: unveiling the emoticon semantic confusion in large language models. Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), San Diego, CA, USA. 2026. p.26783-803. doi:10.18653/v1/2026.acl-long.1233.
8. Pan S, Sun X, Zhang T, Liao D, Yang K, Xing Z. SkillGuard: a permission-centric framework for agent skill security [Preprint]. 2026. arXiv:2606.03024.
9. Xu X, Saghir H, Wu Q, Côté MA, Wang T, Lakkaraju K, et al. The devil is in the interface: evaluating how tool architecture shapes coding agent behavior [Preprint]. 2026. arXiv:2608.11386.
10. Mak H, Suresh S, Bhatnagar S, Wang B, Methani C, Munoz AG. Is Bash all you need? An empirical study of tool interfaces for enterprise digital worker agents [Preprint]. 2026. arXiv:2609.11999.
11. Wang F, Shen G, Fu Y, Guo C, Cui Y, Chen Y. MABash: Multi-agent collaborative parsing for Bash command comment generation [Preprint]. 2026 Jun 1. doi:10.21203/rs.3.rs-9709428/v1.
12. Zhang K, Chen Z, Feng X, Fang D, Zheng Y, Li Z, et al. Lightweight yet secure: secure scripting language generation via lightweight LLMs [Preprint]. 2026. arXiv:2601.06419.
13. Du M, Luo Y, Banerjee S, Wojcik M, Popovic J, Cherukara MJ. Experiment automation agents: automating materials characterization with vision language model agents. NPJ Comput Mater. 2026. doi:10.1038/s41524-026-02213-8.
14. Liu S, Tang X, Yang X, Lin L, Zhou B, Xiao W, et al. When the manual lies: a realistic benchmark to evaluate MCP poisoning attacks for LLM agents. 2026 29th International Conference on Computer Supported Cooperative Work in Design (CSCWD). 2026. p.2090-5.
15. Yu L, Wang P, Xu J, Zhang J, Wang X, Ma J, et al. Towards robust and explainable Bash code generation with robustness-aware group relative policy optimization [Preprint]. 2026. arXiv:2606.27733.

16. Rashidi M. The Balkanization of execution-security research for AI coding agents: isolation, access control, and time-of-check-to-time-of-use vulnerabilities [Preprint]. 2026. arXiv:2607.05743.
17. Wu X, Zhao Z, Li Q, Li X, Shi Y, Adams B, et al. SkillShield: prompt-space security skills for LLM coding agents [Preprint]. 2026. arXiv:2608.25817.
18. Alonso-Carracedo M, Fernandez-Bouillon R, Celard P, Rodriguez-Martinez FJ, Otero-Cerdeira L. Automated grading of Linux/Bash examinations using large language models: a four-level cognitive taxonomy approach [Preprint]. 2026. arXiv:2607.02432.
19. Li Z. Engineering a regular-type system for Unix and Linux shell [thesis]. Providence (RI): Brown University. 2026. Available from: <https://atlas.cs.brown.edu/pdf/rt:brown:2026.pdf>
20. Hong J, Ascoli BG, Choi JD. ReCUBE: evaluating repository-level context utilization in code generation [Preprint]. 2026. arXiv:2603.25770.
21. Xie Y, Lamprou E, Xia J, Vasilakis N. Incr: faster re-execution via bolt-on incrementalization. 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26), Seattle, WA, USA. 2026. p.683-99.
22. Liargkovas G, Jin D, Zhu TE, Liu D, Thompson AB, Narsipur A, et al. hS: speculative script reordering at subprocess granularity. 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26), Seattle, WA, USA. 2026. p.665-81.
23. Song L, Dai Y, Prabhu V, Zhang J, Shi T, Li L, et al. Coact-1: computer-using multi-agent system with coding actions. International Conference on Learning Representations (ICLR). 2026.
24. Chen M, Zhang L, Feng Y, Wang X, Zhao W, Cao R, et al. SWE-Universe: scale real-world verifiable environments to millions [Preprint]. 2026. arXiv:2602.02361.
25. Sladić M, Alibalić E, Valeros V, Catania C, Garcia S. AdvancedShellLM: a stateful multi-agent LLM honeypot for SSH deception [Preprint]. 2026. arXiv:2606.27990.
26. Yeh CJ, Chang CH. PagePilot for web automation based on a multi-agent architecture. Proc Int AAAI Conf Web Soc Media. 2026;20(1):2623-36. doi:10.1609/icwsm.v20i1.42771.
27. Khan HA, Khan I, Usmani KA, Ali M. Architecting next-generation shells: AI-driven command assistance and proactive safety mechanisms in the command-line. Int J Comput Appl. 2026;187:10-16. doi:10.5120/ijca4f98e99c1f94.
28. Sladić M, Valeros V, Alibalić E, Garcia S. Improving LLM-based SSH honeypots through prompting and fine-tuning [Preprint]. 2026. arXiv:2608.18686.
29. Jacobs J, Lapon J, Naessens V. Local LLMs for NL2Bash: a large-scale open-source model evaluation for Bash command generation. Workshop on LLM Assisted Security and Trust Exploration (LAST-X). 2026 Jan 20.
30. Salviati U, De Gaspari F, Conti M, Mancini LV. ShellGames: speculative LLM-driven SSH deception [Preprint]. 2026. arXiv:2606.17986.
31. Apostu AM, Preda A, Damir AD, Bolocan D, Ionescu RT, Croitoru I, et al. AutoMalDesc: large-scale script analysis for cyber threat research. Proc AAAI Conf Artif Intell. 2026;40(1):12-20. doi:10.1609/aaai.v40i1.36959.
32. Kampourakis V, Chatzoglou E, Gkioulos V, Katsikas S. From conceptual scaffold to prototype: a standardized zonal architecture for Wi-Fi security training [Preprint]. 2026. arXiv:2605.07400.
33. Curi de Miranda H, Lopes Roth Ferraz Á, Comin Sonaglio W, Alves Pereira Júnior L. A systematic security testing approach for InterUSS-based environments [Preprint]. 2026. arXiv:2605.
34. Weng X. What you approve is what executes: consent integrity for black-box LLM agents [Preprint]. 2026. arXiv:2606.02668.
35. Mazur K. Hack in a box: portable Docker labs for teaching web application security. In: International Conference on Computational Science. Cham: Springer Nature Switzerland; 2026. p.516-30.
36. Miguel CJ, Izal M. CTF as a service: a reproducible and scalable infrastructure for cybersecurity training [Preprint]. 2026. arXiv:2603.22511.
37. Canedo A. paper.json: a coordination convention for LLM-agent-actionable papers [Preprint]. 2026. arXiv:2605.16194.

38. LogJack SH. Indirect prompt injection through cloud logs against LLM debugging agents [Preprint]. 2026. arXiv:2604.15368.
39. Kozachok AV, Nazimov AM, Magomedov SG. 2DSL: LLM-based code generation for domain-specific languages [Preprint]. 2026. arXiv:2606.22586.
40. Zhang Z, Xu Y, Liang J, Li W, Chen X, Qian L, et al. RepoZero: can LLMs generate a code repository from scratch? [Preprint]. 2026. arXiv:2605.07122.
41. Madatha P. A deterministic control plane for LLM coding agents [Preprint]. 2026. arXiv:2606.26924.
42. Shan Z, Xin J, Zhang Y, Xu M. Don't let the claw grip your hand: a security analysis and defense framework for OpenClaw [Preprint]. 2026. arXiv:2603.10387.
43. Xia H, SlotGuard PY. Stop oversharing private local context in LLM agent transcri [Preprint]. 2026. arXiv:2607.17147.
44. Apostu AM, Preda A, Damir AD, Bolocan D, Ionescu RT, Croitoru I, et al. AutoMalDesc: large-scale script analysis for cyber threat research. Proc AAAI Conf Artif Intell. 2026;40(1):12-20. doi:10.1609/aaai.v40i1.36959.
45. Paolillo A. Pythainer: composable and reusable Docker builders and runners for reproducible research. J Open Source Softw. 2026;11(117):9059. doi:10.21105/joss.09059.
46. Akbar S, El-Hamalawy AF. From Bash to boards: integrating project-based learning in a foundational embedded systems course. 2026 ASEE Annual Conference & Exposition, Charlotte, NC, USA. 2026. doi:10.18260/1-2--59550.
47. Carrillo B, Mallona I. Denet, a lightweight command-line tool for process monitoring in benchmarking and beyond [Preprint]. 2025. arXiv:2510.13818.
48. Mudduluru S, Kittur J, Mackay SN. Evaluating the role of research-inspired projects in lower-division programming course. 2026 ASEE Annual Conference & Exposition, Charlotte, NC, USA. 2026. doi:10.18260/1-2--59581.
49. Chen S, Wang L, Yang X, Liu Z, Cong Y, Ji Y, et al. TUA-Bench: a benchmark for general-purpose terminal-use agents [Preprint]. 2026. arXiv:2606.28480.
50. Chu Z, Hu J, Jiang X, Zou P, Li H, Peng C, et al. Terminal world: benchmarking agents on real-world terminal tasks [Preprint]. 2026. arXiv:2605.22535.
51. Meng L, Feng H, Shumailov I, Fernandes E. Cellmate: sandboxing browser AI agents [Preprint]. 2025. arXiv:2512.12594.
52. Piao Y, Min H, Su H, Zhang L, Wang L, Yin Y, et al. Agent Bay: a hybrid interaction sandbox for seamless human-AI intervention in agentic systems [Preprint]. 2025. arXiv:2512.04367.
53. Wu Y, Roesner F, Kohno T, Zhang N, Iqbal U. IsolateGPT: an execution isolation architecture for LLM-based agentic systems [Preprint]. 2024. arXiv:2403.04960.
54. Yan B. Fault-tolerant sandboxing for AI coding agents: a transactional approach to safe autonomous execution [Preprint]. 2025. arXiv:2512.12806.
55. Dong Y, He J, Liu S, Hou Y, Du D, Xu Z, et al. DeltaBox: scaling stateful AI agents with millisecond-level sandbox checkpoint rollback [Preprint]. 2026. arXiv:2605.22781.
56. Ruan Y, Dong H, Wang A, Pitis S, Zhou Y, Ba J, et al. Identifying the risks of LM agents with an LM-emulated sandbox [Preprint]. 2023. arXiv:2309.15817. doi:10.48550/arXiv.2309.15817.
57. Debenedetti E, Zhang J, Balunovic M, Beurer-Kellner L, Fischer M, Tramèr F. AgentDojo: a dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In: Advances in Neural Information Processing Systems 37. 2024. p.82895-82920. doi:10.52202/079017-2636.