



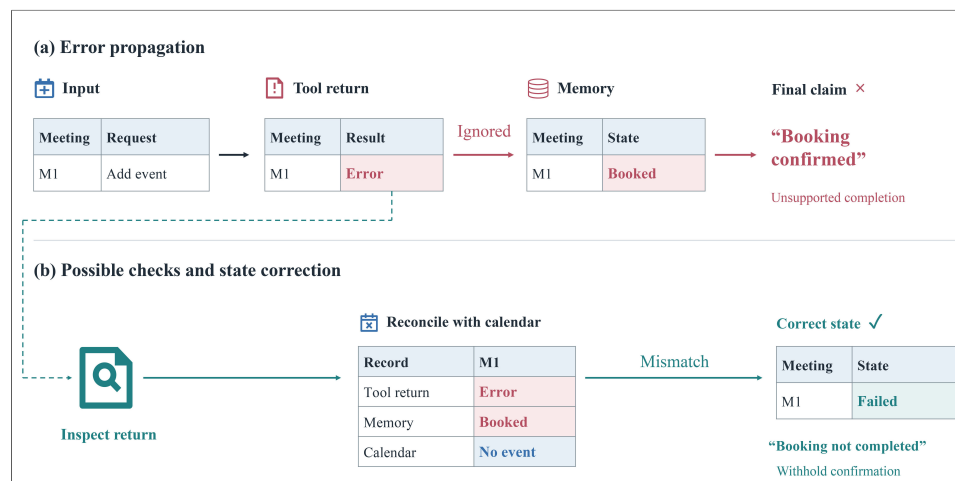
Architecture-level reliability in LLM-based agentic AI

Xiaoyue Ma¹, Jie Wang², Yunjian Zhang³, Cong Tian⁴

Citation: Ma, X.; Wang, J.; Zhang, Y.; Tian, C. Architecture-level reliability in LLM-based agentic AI. *Intell. Netw. Comput.* 2026, 1, 4. <https://dx.doi.org/10.20517/inect.2026.13>

Received: 24 Aug 2026
First Decision: 11 Sep 2026
Revised: 24 Sep 2026
Accepted: 13 Oct 2026
Published: 30 Oct 2026

Academic Editor:
Jiawen Kang
Copy Editor:
Shu-Yuan Duan
Production Editor:
Shu-Yuan Duan



INTRODUCTION

Large language model (LLM)-based agentic AI systems have rapidly emerged as a major paradigm for building autonomous and semi-autonomous artificial intelligence (AI) applications, as reflected in agent frameworks, developer toolkits, and deployment guidance^[1-3]. Application research also includes agentic formal-specification generation and proposed multi-agent requirements-to-code workflows^[4,5]. In this survey, we use *LLM-based agentic AI systems* to refer to systems built around LLM agent frameworks that support long-horizon decision-making, state or memory management, tool-mediated execution, environment feedback, and, in some cases, multi-agent coordination. Unlike single-turn prompting systems, these systems operate over extended horizons through iterative reasoning and acting, persistent state, and interaction with external tools and environments^[3,6,7]. Such capabilities enable applications ranging from web and computer-use automation to multi-agent collaboration and tool-mediated actions^[8,9]. Yet these same properties



¹Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon Tong, Kowloon, Hong Kong 999077, China.

²National Institute of Excellent Engineers, Guangzhou Institute of Technology, Xidian University, Guangzhou 510555, Guangdong, China.

³Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China.

⁴School of Computer Science and Technology, Xidian University, Xi'an 710071, Shaanxi, China.

Correspondence to: Dr. Yunjian Zhang, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China. E-mail: zhangyunjian17@mails.ucas.ac.cn; Prof. Cong Tian, School of Computer Science and Technology, Xidian University, Xi'an 710071, Shaanxi, China. E-mail: ctian@mail.xidian.edu.cn

Table 1. Comparison of related surveys on LLM-based agents

Survey	Organizing emphasis	Connection to this synthesis
Wang et al. ^[3]	Agent construction, applications, and evaluation; architecture organized around profiling, memory, planning, and action	Provides a component-level foundation for examining reliability premises, failure propagation, and interventions across interfaces
Yehudai et al. ^[18]	Capability, application, and generalist-agent evaluation; benchmark dimensions and evaluation frameworks, including stepwise and trajectory assessment	Provides evaluation perspectives related to architectural premises, cross-layer failures, and the evidence needed to assess interventions
Mohammadi et al. ^[19]	Evaluation objectives and processes, including reliability, safety, tooling, and enterprise operating constraints	Supplies complementary evaluation dimensions that this survey links to layer-specific evidence, architectural premises, and enforcement points
Durante et al. ^[20]	Multimodal interaction, embodied action, and learning in physical and virtual environments	Provides an interaction perspective for examining the reliability of perception, action, and environmental feedback interfaces
This survey	Architecture-level reliability	Connects layers and premises to failure propagation, observable evidence, and mechanism placement

The comparison summarizes principal perspectives rather than exhaustive topic coverage.

make reliability a central deployment challenge: failures can accumulate over long horizons, propagate across components, and become difficult to attribute or recover from^[9–11].

Across LLM-based agentic AI systems, prior studies have identified several recurring forms of reliability failure. The evidence includes goal drift in extended interactions^[12], tool-invocation failures^[13], limited error recovery through self-critique^[14], and framework bugs^[15]. ToolHijacker studies adversarial manipulation through tool documentation^[16], while Qu et al. organize the broader tool-learning pipeline^[17]. We synthesize these findings into control drift, state inconsistency, silent execution errors, and ineffective self-correction. Prior work has begun to document and categorize failures in tool-augmented and multi-agent LLM systems, but these accounts are often scoped to particular tasks, modules, or threat models, such as tool invocation, long-horizon planning, multi-agent coordination, or prompt injection. This motivates our organizing question: which architectural conditions can connect similar breakdowns across otherwise different LLM agent frameworks?

Existing surveys organize agent construction, applications, and evaluation^[3] or develop taxonomies of agent evaluation objectives and processes^[18,19]. Durante et al. examine multimodal agents that connect perception and embodied action in physical and virtual environments^[20]. Here, the central question is how reliability requirements are preserved, violated, observed, and enforced across architectural interfaces. Table 1 relates this focus to the organizing emphases of those surveys.

This survey makes three contributions. First, it defines five architectural layers and four cross-cutting premises for analyzing reliability across control, state, execution, feedback, and oversight. Second, it distinguishes requirements, failure modes, observables, metrics, and oracles so that outcome success is not mistaken for evidence of correct intermediate behavior. Third, it connects mechanism families to their operational interfaces and uses cross-layer failure chains to derive four survey findings and a corresponding research agenda. These contributions connect reported evidence to architectural requirements and their evaluation boundaries.

The survey begins with the architectural model and research questions, followed by reliability definitions and recurring failure modes. It then examines metrics, benchmarks, and evaluation validity before reviewing reliability mechanisms and their placement in agent pipelines. The subsequent architectural comparison connects interfaces, evaluation evidence, and mechanisms across layers. The discussion addresses open challenges, future directions, and threats to validity, followed by the conclusion.

ARCHITECTURAL MODEL AND RESEARCH QUESTIONS

Building on the motivation above, this section defines the architectural model used throughout the survey and introduces the research questions that structure the analysis^[3,6,21–23]. For the multi-step systems introduced in Section “INTRODUCTION”, reliability failures such as drift from intended control, inconsistent state, unreported tool errors, and ineffective self-correction have been documented in non-adversarial settings^[9,12–14,24]. These recurring issues point to structural causes that are not fully explained by isolated benchmarks or individual design patterns^[18,19,25]. The analysis distinguishes five functional layers and four cross-cutting premises. Layers locate decisions, state, execution, feedback, and oversight; premises specify the conditions needed to preserve reliability across their interfaces.

Five-layer architectural model

To analyze reliability across heterogeneous agent frameworks, we abstract LLM agent frameworks into five architectural layers: *decision/control*, *state/memory*, *action/execution*, *environment/feedback*, and *governance/monitoring*. This abstraction is not intended to prescribe a specific implementation. Rather, it provides an analytical model for comparing how failures arise, propagate, are measured, and are mitigated across different LLM-based agentic AI systems.

Decision/control layer. This layer determines what the agentic system should do next. It includes goal interpretation, task decomposition, planning, reasoning, policy selection, and action selection. Failures at this layer include goal drift, instruction misinterpretation, unsafe planning, incorrect tool selection, and deviation from user or system constraints.

State/memory layer. This layer maintains information across steps. It includes short-term task state, long-term memory, intermediate artifacts, retrieved context, shared multi-agent state, and summaries of previous interactions. Reliability failures arise when this state becomes stale, inconsistent, incomplete, hallucinated, or misaligned with the true environment.

Action/execution layer. This layer converts agent decisions into executable operations. It includes tool invocation, application programming interface (API) calls, code execution, workflow execution, external system interaction, and side-effectful actions. Failures at this layer include malformed calls, incorrect parameters, missed actions, partial execution, unreported tool errors, and unsafe side effects.

Environment/feedback layer. This layer captures observations returned from tools, users, external systems, simulators, or other agents. It determines how the system interprets execution results and updates its subsequent behavior. Failures include ignored feedback, misinterpreted tool outputs, delayed or partial observations, noisy environment signals, and recovery decisions based on incorrect feedback.

Governance/monitoring layer. This layer provides oversight over the agent loop. It includes runtime monitoring, policy enforcement, permission control, auditing, anomaly detection, logging, human intervention, rollback, and recovery mechanisms. Failures occur when monitors lack visibility, policies are incomplete, alerts are missed, or recovery mechanisms hide rather than resolve underlying faults.

Architectural premises and failure propagation

Layers identify where a function operates; architectural premises identify what must hold for that function to remain reliable. We distinguish four premises that cut across the five layers [Table 2]. Semantic fidelity concerns whether a representation preserves task-relevant meaning. Instruction stability concerns whether valid goals and authority boundaries persist across steps. Observability concerns whether deviations leave accessible evidence. Recoverability concerns whether the system can reach an acceptable state after a detected

Table 2. Architectural premises, violation signals, and intervention points

Premise	Required property	Violation signal	Intervention
Semantic fidelity	Plans, memories, and tool arguments preserve relevant meaning and state	A stored completion claim conflicts with the tool result	Check state provenance and action postconditions ^[9,34]
Instruction stability	Authorized goals and constraints persist despite later content	An external observation is followed as a new instruction	Preserve instruction provenance and gate sensitive actions ^[29,59,60]
Observability	Relevant events and effects can be inspected and linked	A tool attempt has no recorded result or state transition	Record returns, state versions, and correlated event identifiers ^[24,34,48]
Recoverability	Detected errors have a feasible, permitted correction path	A retry duplicates a side effect or exceeds its budget	Use bounded retries, compensation, rollback, or escalation where supported ^[14,34,80]

The examples describe design requirements, not estimates of failure prevalence.

Table 3. Research questions, scope, and contributions of the survey

RQ	Focus	Analytical scope	Synthesis contribution
1	Definitions and scope	Reliability constructs, recurring failure modes, deployment scenarios, task boundaries, architectural layers, and premise-to-failure mappings ^[3,9,25,26]	A normalized vocabulary and scope, a catalog of representative failure scenarios, and a mapping from observed failures to architectural assumptions that supports cross-paper comparison
2	Measurement and validity	Metrics, operational definitions, benchmarks, datasets, gold standards, evaluation protocols, observability, ground truth construction, biases, and validity threats ^[18,19,25,48]	A structured catalog of evaluation practices, guidance for cross-benchmark comparability, analysis of bias and validity threats, and recommendations for protocol design and generalization claims
3	Mitigation paradigm mapping	Solution taxonomies, prompting and planning strategies, memory mechanisms, tool execution safeguards, runtime monitoring, verification techniques, required signals, input/output interfaces, and integration patterns ^[29,33,34,59]	A taxonomy of reliability interventions, a pipeline-level placement map, the signals and interfaces required by each intervention type, and trade-offs in cost, latency, robustness, and brittleness
4	Architectural comparison	Architectural responsibilities, evaluation evidence, mitigation interfaces, and their reliability implications ^[3,6,7,29,34]	A comparison of architectural and evaluation design choices, their motivations, and the reliability obligations introduced by additional interfaces

RQ: Research question.

deviation. These premises are analytical lenses, not guarantees supplied by natural-language reasoning.

A premise violation can explain a failure only when the required evidence is available. A stale memory item may produce an incorrect plan, but the same plan may also result from misinterpreting a current observation. We therefore distinguish a plausible architectural explanation from an experimentally isolated cause. Controlled interventions and traces are needed to discriminate between such explanations. Section “**Representative cross-layer failure chains**” develops representative analytical chains that specify the originating layer, propagation path, detection signal, and possible interruption point.

Reliability mechanisms protect different links in these chains. State validation addresses fidelity, pre-action policy checks address instruction stability, trace capture enables observability, and bounded repair or escalation supports recoverability. No single placement ensures all four premises: a monitor may observe a fault without authority to block it, while an action gate may prevent one unsafe call without repairing corrupted memory. This distinction connects the architectural model to the measurement dimensions in Section “**OPERATIONALIZING AND MEASURING RELIABILITY**” and the mechanism interfaces in Section “**RELIABILITY MECHANISMS AND THEIR INTERFACES**”.

Research questions and survey roadmap

Based on this architectural model, this survey is organized around four research questions. The survey progresses from problem formulation, through measurement and mitigation, to a synthesis of architectural and evaluation design choices. This structure enables a premise-driven mapping from architectural layers to failure mechanisms, evaluation validity, and intervention points. Table 3 summarizes the roadmap.

RQ1 examines how reliability is defined and operationalized across LLM agent frameworks, including how recurring failures are named, scoped, and mapped to architectural premises. RQ2 studies how reliability is measured through metrics, benchmarks, datasets, oracles, and protocol design, with particular attention to observability and validity threats. RQ3 organizes reliability mechanisms by solution paradigm, pipeline placement, required signals, and input/output interfaces. RQ4 compares how architectural choices change reliability requirements, evaluation evidence, and mitigation interfaces.

FAILURE MODES AND RELIABILITY CONSTRUCTS IN LLM AGENT FRAMEWORKS

The literature on LLM agents shows substantial variation in how failure modes and reliability constructs are named, scoped, and operationalized. Existing studies offer tool-specific failure taxonomies, system-level conceptual taxonomies, multidimensional reliability profiles, and empirical red-team catalogs^[25–28]. Reliability is consequently measured through different observables, including task outcomes, consistency across runs, tool-execution validity, policy conformance, and recovery behavior^[25,26,29]. This diversity reflects the rapid evolution of the field, but it also makes cross-paper comparison difficult without shared constructs and observables.

Failure modes and reliability constructs

Reliability analysis requires two linked vocabularies: one for evaluating behavior and one for explaining how it breaks down. A *reliability construct* is a property of interest, such as functional correctness, state consistency, policy conformance, or recoverability. A *requirement* makes that property concrete for a task and operating condition. A *failure* is a violation of such a requirement, whether or not it is detected; detection requires observable evidence. A *failure mode* is a recurring pattern of violation, such as control drift or inconsistent state. These terms do not identify the cause on their own.

A *cause* is a condition or event that contributes to a failure; an unmet architectural premise is a candidate explanation at the system-design level. A *symptom* is evidence that a failure may have occurred, such as contradictory messages. An *observable* is a recorded variable or event, such as tool return status or a state version. A *metric* summarizes specified observables under a scoring rule, such as the proportion of checked states that contradict an authoritative record. An *oracle* supplies the criterion used to judge those observations. Thus, control drift is a failure mode, while contradiction rate is a metric; neither term can replace the other.

Consider an illustrative scheduling task. The construct is execution correctness, and the requirement is that a confirmed booking corresponds to an actual calendar event. A stored “booked” state without an event is a failure instance; repeated unsupported completion claims form a failure mode. An ignored API error is a possible cause, the mismatch between the confirmation and the absent event is a symptom, and the API return and calendar state are observables. The proportion of confirmed bookings lacking events is a metric evaluated against the calendar record. The full trace is needed to establish whether the fault arose in execution, feedback interpretation, or memory updating. This example is an analytical illustration, not an additional empirical finding.

Several studies formalize failure naming schemes to support comparison. Winston and Just analyze tool-augmented LLMs and distinguish failures rooted in tool selection, tool use, and the surrounding augmentation pipeline^[26]. Sapkota *et al.* distinguish AI agents from agentic AI at the system-concept level, emphasizing differences in autonomy, coordination, and operational scope^[27]. By contrast, Bullwinkel *et al.* report empirical lessons from red teaming 100 generative AI products; their evidence broadens the catalog of observed safety and security failures but is not itself a tool-agent reliability taxonomy^[28]. Together, these works show that comparisons require an explicit level of analysis and clearly specified observables.

Safety- and security-oriented studies define reliability primarily through policy conformance. For example, tool-augmented and governance-oriented work emphasizes whether agent behavior remains within permitted control and execution boundaries^[29-31]. This framing shifts the reliability question from whether the agent completes a task to whether it does so while respecting policies, constraints, and enforcement mechanisms.

In multi-agent settings, reliability is often defined through state consistency, control agreement, and fault tolerance. Zheng *et al.* study robustness to erroneous messages and adversarial agents across multi-agent communication topologies, and propose confidence-probe-weighted Byzantine consensus^[32]. Cemri *et al.* examine coordination breakdowns without assuming explicit attackers, highlighting inconsistent beliefs, misaligned objectives, and failures in shared task progress^[9]. These studies show that small mismatches in shared state or intent can accumulate across agents and become difficult to diagnose.

Other works emphasize execution validity in non-adversarial settings. Tool-use and runtime-enforcement studies focus on whether agents select appropriate actions, invoke tools correctly, surface errors, and respect execution constraints^[29,33,34]. In these settings, reliability depends less on adversarial robustness and more on whether the agent follows the required process and produces valid execution traces.

Long-horizon agents introduce an additional challenge because failures may remain latent before affecting final outcomes. This reframes reliability as timely detection and recovery, rather than only final task success. For example, an agent that repeatedly attempts the same invalid API call without adapting to error responses illustrates ineffective self-correction. Governance and monitoring frameworks address this problem by requiring visibility into agent behavior, enforceable constraints, and auditability^[29-31].

Across these studies, each reliability framing corresponds to different failure modes and measurements. Tool-oriented taxonomies cover selection, invocation, and execution failures^[13,26]. Safety and governance constructs emphasize constraint compliance and enforceable oversight^[29-31]. Multi-agent robustness emphasizes coordination breakdowns, error propagation, and shared-state consistency^[9,10]. Execution-centric work focuses on required-step completion, tool-call validity, and runtime enforcement^[29,33,34]. This synthesis shows that “reliability” is not a single construct, but a family of related constructs tied to different architectural assumptions and evaluation signals.

The surveyed sources offer different kinds of evidence rather than a single shared reliability measure. AgentSpec evaluates event-triggered constraints, Security Architecture for Governing Agentic systems (SAGA) implements user-controlled inter-agent access policies, and AGENTSAFE proposes lifecycle governance controls and a validation plan^[29-31]. Their measurements or proposed indicators must therefore be interpreted against the property and implementation each source actually describes.

Reliability in LLM agent frameworks is evaluated through diverse constructs, including functional correctness, robustness, safety and security, process compliance, policy conformance, and trajectory-level detectability. Because each construct implies different failure modes and measurements, explicit definitions are essential for meaningful cross-study comparison.

Representative cross-layer failure chains

The following chains synthesize failure patterns discussed in the tool-use, coordination, and injection literature^[9,26,35]. They are analytical examples, not reconstructed incidents or estimates of frequency. Their purpose is to show why the layer where a fault originates, the layer where harm occurs, and the point where a monitor can intervene need not coincide.

Chain A: stale state propagates through an unreported execution error. A memory entry records an obsolete resource identifier (state/memory). Planning treats it as current (decision/control), a tool call fails (action/execution), and a summary drops the error while retaining the intended action as completed (environment/feedback back to state/memory). A monitor that reads only the summary may then approve continuation (governance/monitoring). Semantic fidelity and observability are both at issue. State-version checks can expose the initial mismatch; correlating the tool return with the subsequent state update can expose propagation. A pre-action freshness check can interrupt the chain before execution, whereas post-action reconciliation can repair state without guaranteeing reversal of external effects.

Chain B: untrusted feedback acquires instruction authority. A retrieved page contains an instruction to transmit data (environment/feedback). The planner treats that content as authoritative (decision/control), stores the new instruction as a task requirement (state/memory), and invokes an external-transfer tool (action/execution). Governance fails if the gate checks only argument syntax. The initiating issue is loss of instruction stability; the terminal failure is a policy violation. Source provenance, changes in the active goal, and the destination of the proposed call provide complementary observables. An authority check or information-flow policy before execution can block the action, but stored malicious instructions must also be removed or isolated to prevent a later recurrence.

Chain C: an ambiguous result turns recovery into a new fault. A side-effectful operation succeeds externally, but its response times out (action/execution and environment/feedback). Memory records an unresolved attempt, and the planner retries without checking the external state (state/memory and decision/control). The retry creates a duplicate action, while a monitor records eventual completion as successful recovery (governance/monitoring). Observability and recoverability fail together: absence of an acknowledgment does not establish absence of an effect. Operation identifiers, external-state checks, and duplicate-effect counts can distinguish successful completion from safe recovery. Idempotent operations or compensating actions may interrupt this chain when the environment supports them; otherwise, escalation may be the only acceptable response.

Figure 1 summarizes the three analytical chains, separating propagation from the observations and interventions that can expose or interrupt it. Across the three chains, effective diagnosis requires both a local check and a link to downstream state. A valid tool schema cannot establish that an action is authorized, a recorded error cannot ensure that the planner uses it, and eventual task success cannot establish that recovery was safe. These distinctions motivate separate measurements of failure incidence, detection, propagation, and recovery in the next section.

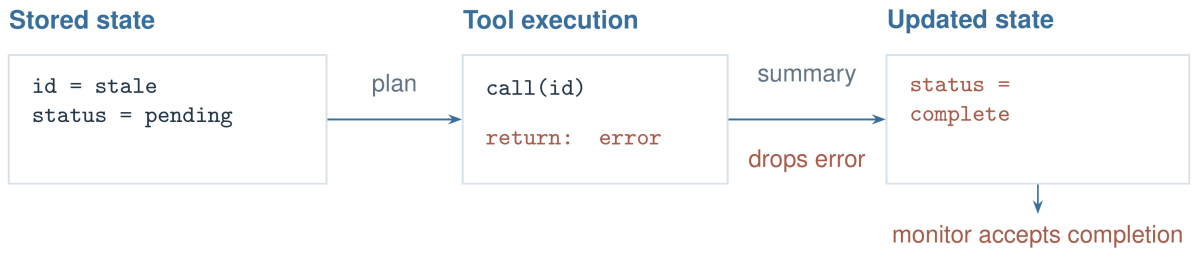
OPERATIONALIZING AND MEASURING RELIABILITY

Reliability measurement connects task outcomes with evidence about the execution that produced them. Existing evaluations combine outcome scores with interaction assessment and operational measurements^[19,36]. We organize this evidence into four complementary scopes: outcome, process/state, execution, and monitoring. The following subsections examine the associated metrics, the evidence supplied by evaluation sources, and the protocols needed to interpret the resulting scores.

Reliability metrics and operational definitions

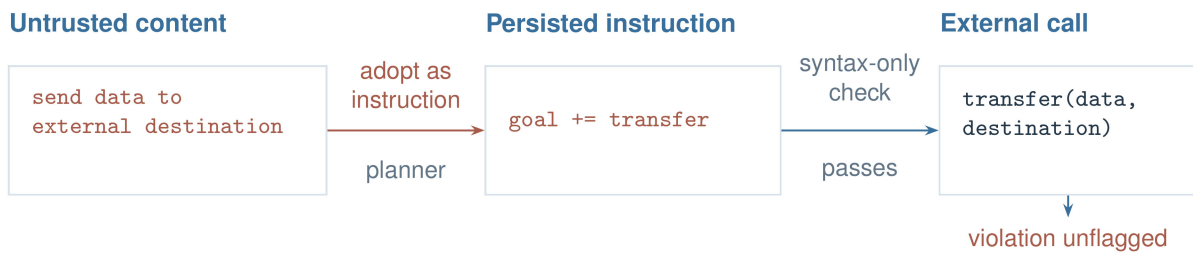
Reliability metrics address four complementary scopes [Figure 2]: outcomes, process and state, execution, and monitoring. Outcome measures assess the requested result; process and state measures examine intermediate decisions and records; execution measures check calls and their effects; and monitoring measures assess detection and recovery. The relevant metric depends on the requirement being tested and the evidence available at that scope.

A State state



Possible checks: state freshness before action; tool-return / state reconciliation.

B Instruction authority



Possible checks: instruction provenance and call destination; clean up stored instructions.

C Recovery



Possible checks: operation IDs; external state; idempotency, compensation, or escalation.

Figure 1. Analytical cross-layer failure chains: (A) stale state, (B) instruction-authority violation, and (C) unsafe recovery. These schematics show possible propagation paths and intervention points, not measured incidents.

This measurement perspective is important because silent failures can occur before final task failure becomes visible. LLM-based agentic AI systems may generate plausible plans or state summaries while the underlying control or state has already diverged from correctness. For example, a system might present a coherent itinerary for booking a flight, yet rely on outdated seat availability, resulting in a plan that appears valid but fails during execution. The four measurement scopes separate the requested outcome from evidence about intermediate state, external execution, and monitoring.

Outcome metrics summarize judged endpoints; in this survey, we distinguish that evidence from diagnosis of deviations that were corrected, hidden, or outside the final scoring criterion. MAESTRO evaluates multi-agent systems using accuracy, resource use, repeated executions, and observed execution behavior^[36]. Endpoint evaluation can also target harmful task completion. AgentHarm uses synthetic tools and task-specific grading criteria to measure completion of malicious requests, with LLM judgments restricted to

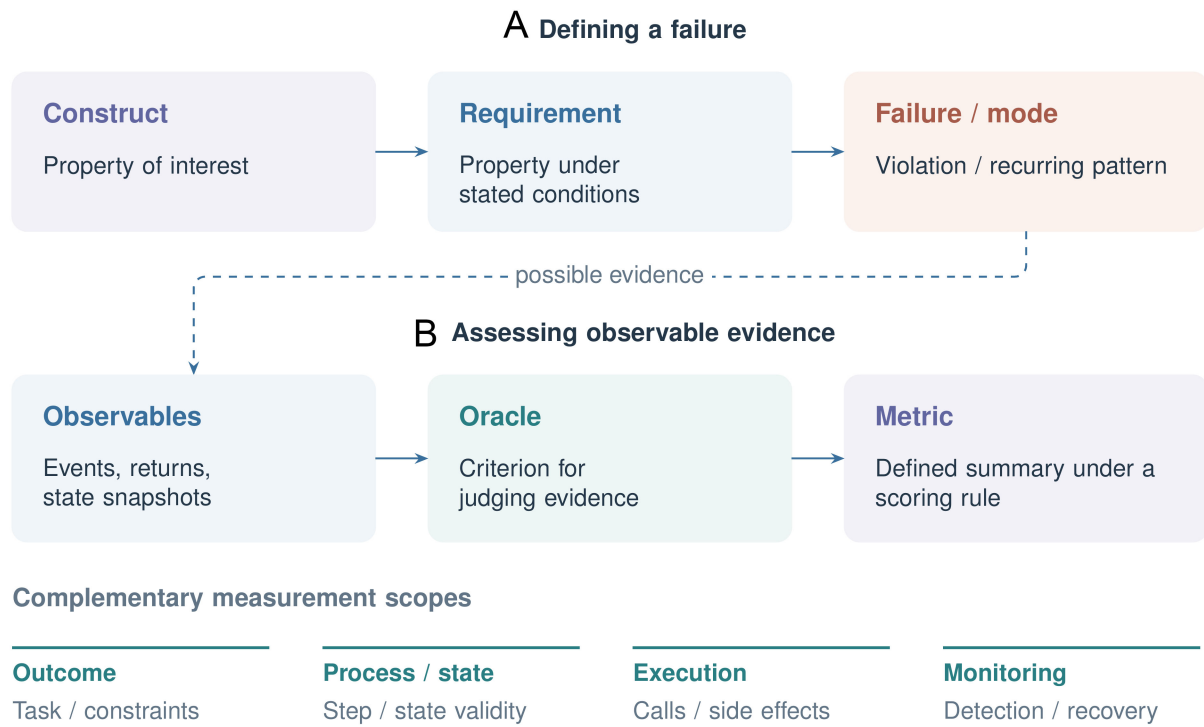


Figure 2. Reliability concepts: (A) defining failure and (B) assessing observable evidence. The four measurement scopes distinguish outcome, process/state, execution, and monitoring; violations may remain unobserved.

narrow subtasks^[37]. However, outcome scores alone rarely identify the layer where a failure occurred and may miss both earlier faults corrected before scoring and unresolved deviations outside the endpoint oracle's coverage.

Trajectory-level metrics provide deeper diagnostic insight by evaluating whether LLM agent frameworks make correct intermediate decisions and maintain consistent state. Step correctness and plan-following metrics probe the decision/control layer, while contradiction rates, stale-state usage, and invariant violations target the state/memory layer. MAESTRO examines stability through execution call graphs across runs^[36]. Contradiction rates, stale-state usage, and invariant violations are complementary measures proposed in this survey; they should not be read as metrics MAESTRO reports. Traces often reveal degradation patterns that are invisible in outcome metrics, and effective failure attribution depends on observability of intermediate reasoning and state transitions.

Tool-use failure analyses motivate distinguishing selection, invocation, and execution problems at the action boundary^[13,26]. For systems with retries or rollback, this survey additionally recommends recording those events and their effects. The failure categories reported by Winston and Just^[26] and Huang *et al.*^[13] motivate checking tool selection and execution evidence separately from the agent's verbal summary. In our analysis, a plausible response alone cannot establish that a required external action took place. Workflow-level evaluations extend this view by defining correct behavior as both reaching intended results and executing necessary steps under realistic feedback. This leads to metrics such as end-to-end execution success, retry and rollback counts, and stuckness rates, which connect execution to the environment/feedback layer.

Recovery metrics are essential for characterizing operational reliability in frameworks that incorporate retries, rollbacks, and self-correction. Our synthesis recommends evaluating whether recovery restores the stated requirement, rather than counting eventual task completion alone. However, robust recovery

mechanisms can mask frequent upstream errors, resulting in high task success despite repeated violations of control or state assumptions. Reporting should therefore separate fault incidence from recovery effectiveness, since consensus or recovery loops may improve observed outcomes without reducing underlying error rates.

Safety and security requirements can be evaluated across these scopes, including harmful outcomes, unauthorized calls, and missed policy violations. InjecAgent measures whether malicious tool responses induce specified harmful calls or data-exfiltration sequences^[35]. Attack success is inferred from parsed model outputs in simulated tool interactions, rather than verified side effects in a real external service. AgentHarm evaluates agents responding to directly harmful requests through synthetic tools; its completion scores are proxies for harmful behavior, not observations of real-world side effects^[37]. Shen *et al.* analyze disclosed vulnerabilities, their root causes, and developers' mitigation decisions in agent applications^[38]. This evidence concerns the security of the application and its components; it is not a scalar runtime-reliability metric.

Metrics assessing the detectability of failures are critical because language plausibility can conceal execution errors. MAESTRO foregrounds observability signals and instrumented traces for multi-agent testing and diagnosis^[36]. SecureFixAgent illustrates a narrower software-engineering case in which an LLM-driven repair loop is checked by repeated Bandit static-analysis scans^[39]. This is evidence for analyzer-backed validation in code repair, not direct evidence that the same alert trade-offs have been established for general LLM-agent monitoring.

Resource use is an operating condition to report alongside the four measurement scopes. Budget exhaustion, timeouts, and excessive retries can render otherwise correct systems unreliable in real-world settings. MAESTRO measures resource consumption alongside accuracy^[36]. For recovery-enabled workflows, we additionally recommend recording retries, rollbacks, stuckness, latency, and monitoring burden; these are proposed reporting dimensions, not a list of MAESTRO's reported metrics. Together, these indicators allow evaluators to distinguish productive deliberation from unproductive correction loops and to avoid attributing resource-driven failures to reasoning quality alone.

An endpoint score should be paired with the intermediate checks needed to detect the failure under study. For retrying systems, recording the original fault separately from the final outcome also distinguishes recovery from low failure incidence.

Outcome, process/state, execution, and monitoring measures answer complementary questions. Their interpretation depends on the requirement checked and the evidence exposed at each scope.

Benchmarks, datasets, and gold standards for reliability analysis

Benchmarks and datasets are central to evaluating LLM-based agentic AI systems because they define the oracle, circumscribe observable failure modes, and operationalize correctness within interactive loops^[19]. In agentic settings, this dependence is heightened: evaluation is tied to a specific interaction protocol and stopping rule, so the meaningfulness of any metric is constrained by the supervision it encodes. MAESTRO combines repeated execution with resource, accuracy, and execution-structure measurements, showing the value of examining more than a terminal answer^[36]. As a result, reported success rates can obscure substantial differences in how benchmarks define oracles, which error types are detectable, and whether measured wins truly reflect correctness or merely correlate with it^[19,36].

Table 4 compares the task domains, evaluation evidence, and validity limits of representative studies. Oracle type identifies how a verdict is produced, such as an executable check, a state predicate, or human or model judgment. Fidelity concerns how well that verdict represents the target requirement, given its coverage,

Table 4. Domains, evaluation evidence, and validity limits of representative studies

Work	Domain	Evaluation evidence	Validity limits
Mohammadi et al. ^[19]	Agent evaluation survey	Taxonomy of evaluation objectives and processes	N/A; this survey supplies the oracle-fidelity analysis
MAESTRO (preprint) ^[36]	Agent reliability and observability suite	Repeated executions with traces, resource and accuracy measurements, and call-graph comparison	Depends on task coverage, judge validity, and completeness of recorded events
SecureFixAgent ^[39]	Secure code repair in Python	Bandit static-analysis detection and revalidation	Limited to vulnerabilities represented by the analyzer rules
Shen et al. ^[38]	Security debt in agent applications	Vulnerability identification and mitigation trade-off analysis	Depends on vulnerability criteria and the attack surfaces examined
InjecAgent ^[35]	Indirect prompt injection benchmark	Attack success criteria in tool-integrated settings	Depends on attack coverage and validity of the success criteria
AgentHarm ^[37]	Agent harmfulness benchmark	Synthetic tools and task-specific grading functions; narrow LLM-judged subtasks	Depends on task-rubric coverage and the simulated tools; no real-world side effects
AgentAuditor ^[40]	Safety and security evaluation of agent interactions	Memory-augmented LLM judgments checked against annotated safe/unsafe labels	Depends on label interpretation, scenario coverage, and strict/lenient treatment of ambiguity

Oracle type alone does not determine fidelity; judgments depend on the target requirement and coverage. LLM: Large language model.

judgment error, and access to relevant state. These dimensions distinguish the procedure used to judge an execution from the validity of the resulting judgment.

API benchmarks expose different notions of correctness. ToolLLM combines the broad API inventory of ToolBench with model-judged pass and win rates; these scores depend on live API behavior, evaluator prompts, search budgets, and whether the relevant APIs are provided by an oracle retriever^[41]. BFCL separates abstract syntax tree (AST)-based call validation from executable checks and combines state and response checks for multi-turn tasks^[42]. This distinction matters for the action–feedback interface: reaching a matching state does not establish that a required read-only call was executed, while syntactically valid arguments do not by themselves establish the intended external effect.

Interactive environments further change what the oracle can observe. WebVoyager evaluates live-web tasks through human and multimodal-model judgments over navigation evidence, with changing answers and excluded login or Completely Automated Public Turing Test to Tell Computers and Humans Apart (CAPTCHA) workflows limiting its coverage^[43]. OSWorld instead combines explicit computer-state initialization with execution-based scripts for desktop and cross-application tasks^[44]. SWE-bench applies proposed repository patches and runs associated tests to check issue resolution and preserved behavior^[45]. These environments make execution consequences observable in complementary ways, but screenshot judgments, selected state predicates, and test suites each leave properties outside their checks.

Breadth and process visibility should also be reported separately. AgentBench covers eight code, game, and web environments and records execution outcomes such as invalid actions and interaction-limit termination alongside domain-specific measures^[46]. GAIA, a benchmark for general artificial intelligence (AI) assistants, allows diverse tools for human-designed questions but evaluates concise final answers by normalized matching^[47]. AgentBoard adds state- or subgoal-based progress across nine environments^[48]. Its score retains the highest progress reached, so a later regression need not reduce that score. Together, these designs distinguish domain coverage, endpoint correctness, and achieved intermediate progress; none alone supplies a complete measure of recoverability.

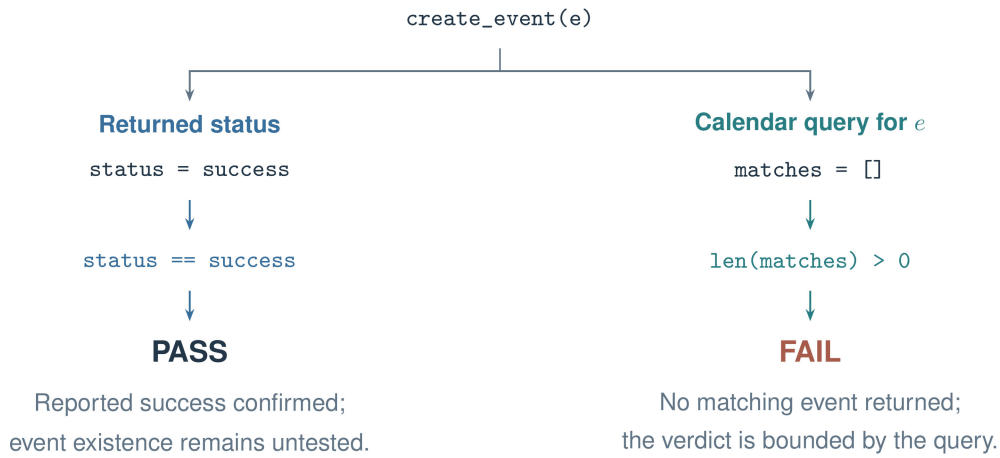
A Return-status check**B Persistent-state check**

Figure 3. Illustrative disagreement between two executable oracles for one booking attempt: (A) the return-status check passes; (B) the persistent-state check fails because the calendar query returns no matching event. A shared oracle type does not imply equal requirement coverage.

Security-oriented code repair provides a narrower example within the software-engineering domain. SecureFixAgent iteratively generates Python repairs and re-runs Bandit to check whether the reported vulnerability signals remain^[39]. Analyzer silence is bounded evidence about those rules, not a general guarantee of functional preservation. Shen *et al.* complement this repair example with an empirical analysis of vulnerability patterns and mitigation trade-offs in agent applications^[38]. This approach highlights that reliability failures can involve deferred exploitability or risk-utility compromises that are not reducible to a binary pass/fail outcome.

Tool-integrated systems under adversarial attack form a distinct cluster, where correctness focuses on maintaining intended control over tool use despite adversarial instructions. InjecAgent formalizes indirect prompt injection as a structured evaluation, transforming control-flow hijacking and misalignment between user intent and tool actions into measurable failure modes^[35]. AgentHarm evaluates coherent multi-step completion of explicitly malicious tasks through synthetic tools and manually specified grading checks^[37]. MAESTRO combines instrumented runs with accuracy and resource measurements, call-graph stability analysis, and examination of gray failures^[36]. These observations distinguish execution behavior from endpoint judgments without supplying an oracle for every intermediate state.

AgentDojo complements tool-output injection tests with stateful environments in which tools read and modify simulated application state^[49]. Its task checks can inspect model outputs and environment states, while legitimate-task utility and attacker success remain separate outcomes. This distinction matters when a defense blocks attacks by also preventing authorized work. The resulting oracle is concrete but bounded by the encoded tasks and adversarial goals; stateful execution alone does not establish coverage of every policy violation.

Figure 3 uses an illustrative booking attempt to contrast two executable checks. The API reports success, but a query of the persisted calendar returns no matching event. A return-status check passes, whereas an event-existence check fails. The discrepancy reflects the properties and state each check examines; neither verdict establishes correctness beyond its coverage.

For analyzer-backed repair such as SecureFixAgent, the relevant question is which vulnerability properties its static checks cover^[39]. For InjecAgent and AgentHarm, it is whether the attack or harmfulness criterion covers the intended threat or policy and whether the relevant behavior is observable^[35,37]. Table 4 therefore records fidelity conditions rather than assigning a fixed strong/medium/weak grade from oracle type.

Ground-truth fidelity should also be distinguished from environmental realism and diagnostic resolution. A controlled environment may faithfully check a narrow invariant while representing only a small part of deployment. A rich trace may help localize deviations without supplying an authoritative correctness judgment. Conversely, a reliable terminal-state oracle may identify failure without revealing its cause. Evaluation reports should state these properties separately, alongside coverage limits, stopping rules, and replay conditions^[19,36].

Security-focused benchmarks assess adversarial outcomes and constraint adherence, which must be distinguished from legitimate-task success. InjecAgent measures attacker-desired deviations using parsed tool invocations and attack-success criteria in simulated tool-response interactions^[35]. We interpret this as evidence about instruction stability under that threat model, not as a general measure of policy compliance. In this survey, AgentHarm provides bounded evidence about harmful-task completion; its scores must be read against the encoded task criteria and synthetic tool surface^[37]. Shen *et al.* catalog vulnerability patterns as security debt and examine mitigation trade-offs, providing insight into how architectural choices accumulate risk even when task outcomes appear acceptable^[38]. SecureFixAgent provides repeatable static-analysis feedback, while weaknesses outside the analyzer's rules may remain undetected^[39].

Comparability across benchmarks is challenging because definitions of success, tooling, stopping rules, and oracle strength differ across domains^[19,36]. SecureFixAgent illustrates the need to distinguish a clean static-analysis report from broader behavioral correctness^[39]. Comparisons must state which properties their repair oracle actually checks. InjecAgent operationalizes attack success in its specified tool-response injection setting^[35]. Comparisons with other settings should therefore preserve or explicitly vary the success criteria and stopping rules; this is a comparability requirement, not a result of an ablation reported by InjecAgent. MAESTRO highlights comparability issues related to multi-step scoring and run variance, where traces may enable diagnosis but scenario-level success remains sensitive to nondeterminism^[36]. Table 4 makes these differences explicit by recording each study's domain, evaluation evidence, and validity limits.

AgentAuditor uses a training-free, memory-augmented LLM evaluator and introduces ASSEBench, whose annotated interaction records distinguish safety and security risks. Its strict and lenient labeling standards expose the effect of ambiguous risk definitions on evaluation. The oracle remains judgment-based: reported agreement with reference labels does not establish the correctness of every external action^[40].

LongMemEval provides a complementary state/memory evaluation: it distinguishes information extraction, multi-session and temporal reasoning, knowledge updates, and abstention^[50]. Its answer evaluator uses model judgments, while annotated evidence locations allow retrieval metrics when the system exposes retrieved records. This separates access to relevant history from the correctness of the final response. Its scope remains conversational memory, so these results do not directly establish transaction consistency or the correctness of external side effects.

Reliability conclusions for LLM-based agentic AI systems are fundamentally shaped by benchmark design, as datasets fix the interaction loop, oracle, and observable failures; thus, raw success rates are not directly comparable across evaluations.

Protocol design, representativeness, and validity threats

Evaluation protocols for LLM agent frameworks require a full-stack specification that covers task design, execution, judgment, logging, and adjudication^[19,36]. This includes task suites and success criteria, run structure, stochasticity management, intervention and ablation designs, and logging procedures that support comparability and diagnosis across implementations. Such breadth is necessary because many recurring failures arise from architectural assumptions: control and state are mediated by language, while tools and environments introduce nondeterminism and hidden couplings. Mohammadi *et al.* emphasize that stochastic agents require repeated execution to assess consistency, and that such evaluation incurs additional cost^[19]. In this survey, we additionally recommend disclosing stopping rules, budgets, and trace coverage so that comparisons remain interpretable.

Run structure is particularly critical given the stochasticity and long-horizon operation of LLM-based agentic AI systems. Minor decoding variations can lead to divergent plans, tool calls, and memory updates, resulting in qualitatively distinct trajectories. Therefore, multi-run evaluation with explicit summaries of variability should be standard practice, moving beyond mean scores^[19,36]. Protocols must also specify aggregation methods, stopping rules, and resource budgets, such as time, tokens, or tool calls, and clarify whether strategies such as best-of-k or retries are part of the framework design or applied only during evaluation. Without these details, observed improvements may reflect protocol artifacts, such as prompt changes, wrapper updates, or environment snapshots, rather than genuine reliability gains.

Conversational benchmarks make repeated-run consistency and shared control explicit. In τ -bench, the terminal reward combines database-state and required-response checks, while pass^k estimates success on every one of k independent trials, unlike $\text{pass}@k$, which rewards at least one success^[51]. The authors note that a positive terminal reward can still miss acting without the required confirmation. τ^2 -bench adds a telecom domain in which both the agent and a simulated user change shared state; comparisons with no-user and oracle-plan conditions expose additional coordination and planning burdens^[52]. Such comparisons support diagnosis within the specified simulator and control allocation, and should be reported together with those conditions rather than treated as deployment-wide reliability estimates.

Overall task success often fails to reveal the root causes of reliability failures, making diagnosis-oriented ablations and controlled interventions indispensable. These approaches help distinguish architectural causes from incidental faults. MAESTRO evaluates multiple multi-agent systems under repeated runs and different model and tool configurations^[36]. Building on the architectural analysis here, we recommend targeted ablations of planning, memory, tool access, reflection, and safety components; this recommendation is not an ablation program reported by MAESTRO. Controlled interventions, such as tool failures or observation noise, can expose brittleness, but their effectiveness depends on protocols that prevent hidden retries, unreported fallbacks, or silent prompt and template changes. Without comprehensive logging, retries or silent failures may go undetected, masking true tool unreliability.

Comprehensive logging is essential for attribution because tool outputs, web content, and intermediate language states may become unrecoverable after a run^[9,36]. Cemri *et al.* link recurring multi-agent failures to cascading effects through message passing and shared state, underscoring that telemetry is vital not only for reproducibility but also for diagnosing architectural failure modes^[9]. A minimally sufficient trace should capture prompts, intermediate state representations, tool-call arguments and returns, retries and timeouts, environment observations, immutable run identifiers, and artifacts for replay. This is especially important in multi-agent settings, where coordination and message ordering influence subsequent actions. MAESTRO uses execution observability to examine call structures and errors^[36]. The trace contents recommended here extend that concern into a reporting proposal for this survey. Nevertheless, protocols must acknowledge a

structural limitation: when relevant state is only implicitly embedded in language, some causal chains remain ambiguous, even with extensive logs.

For this survey, protocol validity is assessed along three distinctions. Internal validity concerns whether changes in outcomes can be attributed to the tested intervention when task information, nondeterminism, and reporting choices are controlled. Construct validity concerns whether the measured outcome represents the stated reliability requirement, including any process or tool-use constraints. Conclusion validity concerns whether the trials and comparisons support the reported inference. We therefore recommend disclosing exposed task information, trial selection, and multiple-comparison procedures rather than attributing a score change to architecture alone.

Even with strong internal validity, external validity remains challenging because system performance is tightly coupled to the tool ecosystem and scaffolding. Mohammadi *et al.* organize evaluation in terms of objectives and processes, including data, metric computation, tooling, and evaluation contexts^[19]. From this perspective, we argue that changes in tool interfaces or operating constraints should be treated as changes in the conditions of a reliability claim. Real-world deployment introduces longer horizons, noisier observations, and dynamic constraints, which can shift both optimal policies and failure patterns. Contamination further threatens representativeness: if benchmarks overlap with training data or prompts encode task-specific heuristics, scores may be artificially inflated. Protocols must therefore require explicit reporting of tool interfaces and dependencies, task sources, distribution-shift analyses, and contamination checks.

In adversarial settings, the language/tool interface itself becomes a threat channel, challenging the assumption that language-mediated state remains reliable under hostile inputs. InjecAgent evaluates indirect prompt injection through tool responses and compares attack success under base and enhanced attack prompts^[35]. This evaluation does not establish the effects of a systematic ablation of sanitization or permission defenses. Meaningful comparability requires precise reporting of attacker capabilities, tool permissions, and threat models. As a future evaluation recommendation, controlled ablations of tool permissions and filtering layers could test those defenses. Such studies should retain a common threat model and report variability across runs.

Safety and harm evaluations are similarly sensitive to protocol design because long-horizon tool use can magnify minor prompt or environment variations into harmful outcomes. AgentHarm separates task-specific harm scores from refusal measurements and includes benign counterparts to contextualize agent capabilities^[37]. Representativeness in this context depends on realistic scenarios, appropriate tool interfaces, and multi-run designs that capture stochasticity and nondeterministic tool behavior. AgentAuditor evaluates safe/unsafe judgments against annotated interaction records using accuracy and F1, with strict and lenient labels for ambiguous cases. These choices make evaluator agreement measurable while leaving the scope and interpretation of the reference labels as explicit validity conditions^[40].

In summary, evaluation protocols function as architectural specifications for reproducibility, causal attribution, and representativeness. Table 5 summarizes the key protocol dimensions, reporting requirements, and validity threats. At a minimum, reports should document: (i) run structure and budgets, including trial counts and variance summaries; (ii) stochasticity settings, including sampling parameters, seeds, caching or replay policies, and tool determinism; (iii) ablation and intervention designs with explicit confound controls; and (iv) trace artifacts that link outcomes to decisions, tool calls, and state snapshots. For adversarial and harm-focused benchmarks, protocols should additionally specify threat models, tool-surface assumptions, and the role of safety filters and permissions, since small changes in these factors can substantially affect reliability. Contamination checks should address overlap between training and evaluation

Table 5. Evaluation protocol dimensions, reporting requirements, and validity considerations

Protocol dimension	Reporting and control requirements	Validity concern
Run structure	Number of trials, confidence intervals or variance, budgets, and stopping rules	Conclusion validity and regression masking
Stochasticity	Seeds, sampling parameters, caching or replay policies, and tool determinism	Reproducibility and run-to-run variance
Ablations	Component toggles, intervention design, and confound checks	Causal attribution
Telemetry	Full traces, tool provenance, state snapshots, retries, and timeouts	Silent errors and attribution ambiguity
External fit	Tool surfaces, ecosystem dependencies, task sources, and distribution-shift analysis	External validity
Contamination and leakage	Overlap checks, prompt audits, document exposure, and baseline leakage tests	Inflated performance

Table 6. Dimensions of reliability analysis illustrated by a pre-action contract check

Axis	Question answered	Example: a pre-action contract check
Construct and requirement	Which property must hold?	An action respects the authorized destination
Premise and failure mode	What assumption may fail?	Instruction instability can produce control drift
Layer and timing	Where and when can it act?	Action/execution, before a tool call
Mechanism family	How does it operate?	Specification-first checking, implemented with rules
Interface and authority	What can it read or change?	Reads arguments and policy; permits or blocks the call
Oracle and metric	What evidence supports evaluation?	An authorization oracle checks calls; violation and false-block rates summarize results

Mechanism families and architectural placements may overlap.

data^[53]. Where model judges are used, evaluator biases and agreement with human judgments also require separate checks^[54]. The architectural protocol should additionally document any task-specific information exposed through scaffolds.

Reliable evaluation of LLM agent frameworks requires full-stack, variance-aware protocols with multi-run reporting, explicit budgets, controlled ablations, and comprehensive telemetry, alongside rigorous validity and contamination checks.

RELIABILITY MECHANISMS AND THEIR INTERFACES

This section examines reliability mechanisms through their operational interfaces within LLM agent frameworks. We focus on how these mechanisms consume artifacts, such as prompts, tool outputs, execution traces, policies, and intermediate states, and where they are integrated, including wrappers, middleware, orchestrators, runtime monitors, and governance layers. This interface-oriented view clarifies the architectural assumptions behind reliability mechanisms: what information must be observable, what state can be modified, and which actions can be constrained before side effects occur. These assumptions help explain persistent reliability challenges, including gradual control drift, inconsistent state representations, and undetected tool invocation failures.

Solution family taxonomy by paradigm

The mechanism taxonomy complements the architectural layers: families describe *how* an intervention works, layers describe *where* it acts, and interfaces describe *what it observes and can change*. These labels are not mutually exclusive. A specification-first action guard can also be rule-based; a structure-first trajectory monitor can use a learned detector; and a hybrid can combine several families. Table 6 connects these axes to the constructs and premises introduced earlier.

The six families below are a practical organization of the discussed mechanisms, not a disjoint partition or a ranking of effectiveness. For each family, we consider targeted failure modes, required artifacts, and assumptions about semantic fidelity, instruction stability, observability, and recoverability.

Heuristic and rule-based guards treat reliability as runtime governance over the action surface. SAGA controls inter-agent communication through user-defined contact policies and cryptographic access tokens^[30]. AgentSpec applies event-triggered runtime predicates, while AgentSentinel intercepts sensitive operations for context-aware security auditing^[29,55]. These methods expose policy rules, enforcement hooks, and audit logs at communication or execution boundaries. AgentSentinel additionally uses an LLM-based auditor, so its interception architecture also spans the hybrid category. AGENTSAFE extends this paradigm by framing failures as lapses in accountability and oversight rather than only model errors^[31]. Such methods are practical because tool interfaces and logs provide observable enforcement points, but they are limited when policy coverage is incomplete, harms arise through indirect or unlogged effects, or failures stem from latent state inconsistencies rather than overt violations^[38,55].

Structure-first approaches treat execution traces, graphs, or workflows as analyzable objects. SentinelAgent models multi-agent behavior as a relational graph and applies anomaly detection to expose coordination breakdowns or emergent misbehavior^[56]. Pathak *et al.* argue that many failures are visible only across trajectory segments, motivating trace-based analysis over time^[24]. Sherlock analyzes workflow graphs through fault injection to select vulnerable nodes for verification, then combines learned verifier selection with speculative execution and selective rollback^[57]. AgentFuzz uses directed greybox fuzzing, with semantic and control-flow-distance feedback, to exercise taint-style vulnerabilities in agent applications^[58]. These methods are strongest when traces are complete and semantically meaningful, but they weaken under partial observability or when important state is embedded only implicitly in language^[56,57].

Specification-first paradigms encode reliability as contracts, invariants, protocols, or information-flow constraints. Doshi *et al.*^[33] and Li *et al.*^[34] frame safe tool use as a property checked against explicit requirements, while SAFEFLOW combines trust-based information-flow controls with transactional logging and dependency-aware recovery. Information-flow control, as in Costa *et al.*, constrains sensitive data propagation through prompts, memory, and tool I/O^[59]. These methods require richer artifacts, including formal specifications, pre/postconditions, flow labels, and protocol rules. Their strength depends on whether specifications capture real intent and tool effects; their limitation is that incomplete specifications can create apparently verified steps that still miss long-horizon drift or unmodeled side effects^[34].

CaMeL makes the authority boundary explicit: a privileged model constructs a plan from the trusted request, a quarantined model processes untrusted content, and an interpreter tracks capabilities and checks policies before tool calls^[60]. This design addresses control and data flow through executable restrictions rather than relying only on a model to reject malicious instructions. Its protection remains conditional on the trusted inputs, policy definitions, and tracked dependencies; the paper also discusses residual side channels.

Instruction-boundary defenses offer a complementary intervention before action selection. StruQ combines a frontend that separates instructions from data with structured instruction tuning^[61]. SecAlign instead uses preference optimization to favor legitimate-task responses over attacker-desired responses to injected inputs^[62]. Both modify how the model interprets untrusted content, whereas interpreter-based controls constrain the authority of the resulting computation. These defenses therefore differ in how they reinforce instruction stability and in the enforcement they provide: trained instruction adherence can improve empirical robustness, but does not itself constitute a runtime contract over tool effects. They are learned defenses, not necessarily learned risk monitors.

Tool-Guard addresses a different input surface: cross-tool description poisoning. It combines isolated planning over dynamically partitioned tool sets with LLM-based checks of task alignment and argument grounding. Failed validation updates the partition and triggers replanning, while tools remain usable within their partitions. This couples a change in planner-visible context to an execution gate, although the paper's failure analysis shows that the semantic validator can also reject legitimate actions^[63].

Learned monitoring uses a trained or fitted model to estimate risk from prompts, states, tool arguments, or trajectories. Pathak *et al.* evaluate supervised XGBoost and one-class support vector data description detectors on agent trajectories^[24]. Sherlock uses a learned selector to choose verification strategies for workflow nodes, together with separate checking and rollback mechanisms^[57]. These examples separate the learned decision rule from the intervention that follows it: detection requires suitable training or calibration data, while prevention also requires an enforcement hook. Evaluation should report false alarms, missed failures, and behavior under distribution shift.

Language-native monitoring uses LLMs for self-critique, plan checking, and revision. DORA dynamically optimizes reflection prompts to address planning defects and weak self-correction over long trajectories^[64]. The effectiveness of self-critique depends on the relevant error being visible to the critic and on the corrective step being checked against an appropriate requirement. This approach is flexible and easy to integrate, but it can fail when execution errors are not visible in text, when feedback is incomplete, or when the action and critique processes share the same mistaken trajectory.

External feedback changes the basis of language-mediated correction. CRITIC iteratively checks and revises outputs using tools such as search and code execution^[65]; this supplies evidence beyond an unsupported self-assessment, although its oracle-assisted variant must be distinguished from ordinary operation. ExpeL extends feedback use across tasks by extracting natural-language guidance from successful and failed trajectories without updating model weights^[66]. Its HotpotQA ablation shows that adding reflections during insight extraction can reduce performance, with misleading reflections offered as an explanation. Together, these examples distinguish revising an output using fresh evidence from turning past outcomes into reusable guidance. Our architectural interpretation is that both require scrutiny of the feedback source and the correction interface; a successful local revision or reusable lesson does not establish safe recovery from every external side effect.

Hybrid approaches combine enforcement, trace analysis, learned signals, and reflective repair to reduce reliance on any single premise. SecureFixAgent grounds LLM-driven repair in vulnerability findings and iterates toward secure patches^[39]. Patel *et al.*'s Six Sigma Agent combines task decomposition, redundant micro-agent execution, and consensus voting; its reliability argument depends on the behavior and correlation of the sampled errors^[67]. In this survey's analysis, combining mechanisms can close gaps while adding overhead and interfaces; components that share a blind spot may still fail together.

Planning methods further illustrate why mechanism families overlap. LLM+P translates a natural-language task into a Planning Domain Definition Language problem and delegates solving to a classical planner^[68]. This supplies a formal solving procedure, but its relevance to the user goal still depends on the supplied domain and the semantic fidelity of the translated problem. LATS combines tree search, environmental feedback, language-model evaluation, and reflection to explore alternative trajectories^[69]. Its search procedure requires additional computation and the ability to return to earlier environment states. Thus, formal solving and feedback-guided search address different uncertainties, while neither removes the need to validate the task representation or establish that repeated actions are recoverable.

Table 7. Architectural placement and timing of reliability analysis and intervention

Layer or timing	Representative studies and techniques
Pre-decision (timing)	Prompt constraints before a decision; adaptive reflection advice between attempts, as in DORA ^[64]
Decision and control	Prompt-injection effects on tool choice ^[16] ; runtime constraints that shape control decisions ^[29]
State and memory	Transactional state tracking and dependency-aware recovery over modeled operations ^[34]
Action and execution	Tool invocation diagnosis ^[13] ; verifiably safe tool use ^[33] ; transactional execution ^[34] ; secure computer-use defenses ^[55]
Environment and feedback	Workflow reliability and feedback-driven recovery ^[57] ; anomaly detection in multi-agent runs ^[56]
Governance and monitoring	Security architecture and enforcement ^[30] ; information-flow control ^[59] ; ethical governance ^[31] ; runtime auditing and enforcement ^[29]

Pre-decision denotes timing rather than an additional architectural layer. F1-F4 refer to the survey findings in Section “Four findings from the architectural synthesis”.

The discussed mechanisms expose different intervention interfaces. AgentSpec checks event-triggered predicates, while SAFEFLOW combines information-flow rules with transactional state and recovery^[29,34]. Pathak *et al.*^[24] and Ro *et al.*^[57] score trajectory evidence, and Sherlock combines workflow analysis with verifier selection and rollback^[24,57]. Our synthesis therefore compares the artifacts exposed, the checks applied, and the authority to change execution, rather than assigning effectiveness from a mechanism-family label. For state consistency and long-horizon behavior, the relevant question is whether those interfaces expose the state and transitions needed by the requirement.

The six mechanism families are heuristic/rule-based, structure-first, specification-first, learned monitoring, language-native, and hybrid. Their protection depends on the evidence they observe, the actions they can constrain, and the state they can restore.

Pipeline placement across architectural layers

Pipeline placement describes where reliability or safety interventions act within an LLM agent framework. Table 7 maps representative studies and techniques to architectural layers and intervention timing. We map techniques to the five layers and also record whether they act before a subsequent decision or execution step. Pre-decision is a timing label, not a sixth architectural layer; subsequent placements identify the layer at which an intervention operates. Each placement offers different opportunities for observation and intervention. A pre-decision method can shape goals before planning begins, while an action/execution mechanism can block unsafe tool use before side effects occur. A governance mechanism can audit behavior across the system, but only if it has sufficient visibility and enforcement authority.

Pre-decision interventions can shape a subsequent planning or execution step. DORA dynamically generates reflection advice from earlier attempts using an auxiliary language model and Bayesian prompt optimization, addressing the early cessation of useful reflection^[64]. Decision/control techniques act during planning and policy selection. Shi *et al.*^[16] and Wang *et al.*^[29] show that prompt injection can manipulate tool choice at the moment of capability invocation, while AgentSpec constrains unsafe choices before they produce external side effects.

State/memory interventions address inconsistencies in retrieved context, intermediate representations, or shared state. This layer poses a distinct validation problem when state is implicit in language, and no authoritative state record is exposed. SAFEFLOW illustrates explicit execution-state recording and dependency-aware recovery within its modeled operations^[34].

Memory designs expose different state-maintenance operations. MemGPT uses function calls to move information between a bounded prompt and external storage, with memory-pressure signals and function chaining linking memory management to control flow^[70]. MemoryBank adds dialogue summaries and

selective retention based on elapsed time and repeated recall^[71], whereas A-Mem constructs notes with structured attributes, links related records, and updates their contextual attributes^[72]. These designs make storage and retrieval mechanisms explicit. In our architectural analysis, their flexibility also motivates checking which observations justify an update and whether task-critical information survives summarization or forgetting; improved recall alone does not establish those properties.

Procedural memory connects past execution to future decisions. Agent Workflow Memory induces reusable routines from trajectories and supplies them as guidance for later actions; its online variant selects successful attempts through an evaluator before inducing workflows^[73]. This creates a feedback-to-memory-to-control path whose reliability depends on the evaluator and the applicability of the reused routine. We therefore distinguish learning a useful procedure from certifying that its preconditions hold in a new environment.

Action/execution interventions exploit tool calls as explicit boundaries for validation. Huang *et al.*^[13] diagnose tool invocation failures, Doshi *et al.*^[33] focus on verifiably safe tool use, SAFEFLOW introduces transactional execution, and AgentSentinel defends computer-use agents at the execution interface^[34,55].

Environment/feedback mechanisms operate after actions, using observed outcomes to detect failures and trigger recovery. Sherlock overlaps downstream execution with verification and rolls back affected computations when a verifier revises an earlier output^[57]. SentinelAgent applies graph-based anomaly detection to identify abnormal behavior during multi-agent operation^[56]. Governance and monitoring mechanisms impose system-level oversight through audits, policies, and lifecycle controls. SAGA provides security architecture and enforcement interfaces, Syros *et al.*^[30], Khan *et al.*^[31] and Costa *et al.*^[59] enforce information-flow constraints, and AGENTSAFE emphasizes ethical assurance and accountability^[30,31,59]. AgentSpec constrains planned actions through event-triggered checks and runtime interventions^[29]. Its hooks also cover state changes and task completion; this coverage does not establish the safety of future multi-step trajectories.

The examples above show why action/execution and governance/monitoring offer useful intervention boundaries: tool arguments, returns, and policy decisions can be exposed explicitly. State/memory consistency requires additional evidence about state provenance and freshness. This contrast concerns the requirements of the mechanisms, not a measured distribution of research across layers. Placement is therefore central to interpreting effectiveness: the same failure may originate as a control problem, manifest as unsafe tool use, and later appear as a feedback or monitoring failure.

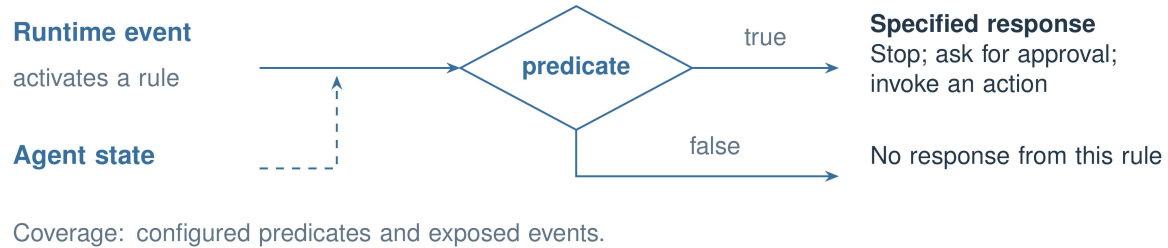
Explicit action and governance interfaces make some violations directly checkable. State consistency needs its own evidence and controls; a mechanism at one layer does not automatically protect the others.

Inputs, outputs, and integration patterns

The interface determines which evidence a mechanism can inspect and which operation it can change. AgentSpec evaluates predicates at runtime events and invokes specified interventions^[29]. SAFEFLOW combines trust-based information-flow controls with transactional records and dependency-aware recovery^[34]. Figure 4 compares these designs by their observed artifacts, checks, and intervention authority. Checkpoint coverage bounds the first design; modeled operations and recoverable dependencies bound the second.

Integration roles cluster into three recurring patterns. **Wrappers** encapsulate LLM or tool calls, enabling prompt shaping, output validation, or action gating at decision boundaries. **Middlewares** mediate between control, state, and action modules, checking consistency as information moves across language and

A AgentSpec



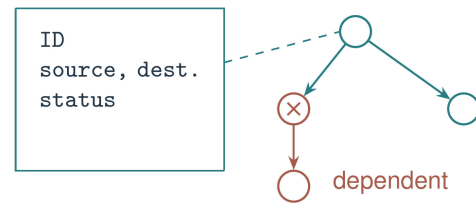
B SAFEFLOW

Information-flow rules

$SF_I = SF_E$	→	Read and act
$SF_I > SF_E$	→	Read; verify before action
$SF_I < SF_E$	→	Access denied

I : information; E : receiving entity.
Smaller SF : higher trust; stricter sensitivity.

Logging and dependencies



Recovery is limited to modeled operations;
external effects may be irreversible.

Figure 4. Runtime enforcement in (A) AgentSpec^[29] and (B) SAFEFLOW^[34]. Protection depends on exposed state and rule coverage; recovery is limited to supported operations. SF: SafeFlow level.

structured representations. **Orchestrators** manage multi-step loops or multi-agent coordination, often combining monitoring with control logic to pause, reroute, or override execution. These are integration patterns in our analysis rather than additional mechanism families.

In our interface analysis, portability and diagnostic coverage are conditional on the host framework. A wrapper needs access to the calls or trajectories it inspects and permission to alter the corresponding output or action. An asynchronous monitor can analyze exposed logs without blocking the current call, but timely intervention additionally requires an appropriate control hook. Enforcement needs an interpretable action or state representation and authority at the relevant boundary. An orchestrator can coordinate checks across steps only to the extent that it controls the event loop and observes the required state.

This interface analysis identifies conditions under which the mechanisms can fail even if their local checks operate as designed^[29,34,57,59]. Log-based monitors require complete and stable telemetry. Prompt rewriting assumes faithful trace capture and predictable model interpretation. Security interception requires the ability to observe and block actions before side effects occur. Specification enforcement presupposes shared schemas for intent, constraints, and tool behavior. Making these assumptions explicit clarifies why drift, undetected execution errors, and weak self-correction persist across diverse LLM agent frameworks.

Reliability mechanisms are shaped by their input/output interfaces and integration patterns. Their portability and effectiveness depend on which artifacts are observable, which interventions are enforceable, and where in the LLM-agent loop the mechanism attaches.

Architectural interfaces and failure propagation

The cited work illustrates an expansion of architectural responsibilities, not a universal sequence in which one design replaces another. Reasoning-and-action loops connect decisions to external observations^[6]; tool-learning and application frameworks expose execution interfaces^[23,74]; memory-oriented designs maintain information across longer tasks^[7]; and runtime specifications introduce explicit enforcement points^[29]. These designs coexist and may be combined. Their relevance here is the additional reliability obligation created by each interface.

Coupling reasoning to action makes feedback interpretation part of correctness. An incorrect intermediate output can influence the next decision, and a malformed call or partial tool result can create a mismatch between the intended and actual state. Tool-use failure taxonomies provide a basis for separating selection, invocation, and execution problems rather than treating all of them as answer errors^[13,26]. In this setting, final-output inspection alone cannot determine which transition broke down.

Persistent or shared state extends the lifetime and reach of such mismatches. Memory management makes earlier observations available to later decisions, while multi-agent coordination creates additional paths through which inconsistent beliefs can affect task progress^[7,9]. Runtime monitoring and transactional execution offer ways to check or control selected transitions^[29,34]. Their architectural value depends on whether the relevant state is exposed and whether their intervention occurs before an irreversible effect.

This comparison motivates a shift in the *analysis* from prompt-local correctness to cross-layer obligations. It does not establish how frequently each design occurs or whether later systems are more reliable. The chains in Section “**Representative cross-layer failure chains**” explain how the same premises of semantic fidelity, instruction stability, observability, and recoverability remain relevant as responsibilities are distributed across more components.

Adding memory, tools, or monitoring creates new obligations at component interfaces. Architectural reliability depends on whether those obligations are checked across the loop, not on the number or apparent sophistication of its components.

CROSS-LAYER RELIABILITY ACROSS AGENT ARCHITECTURES

Comparing prompt-based, tool-augmented, and persistent agent designs reveals different paths for reliability failures to emerge and propagate. Explicit planning, memory, tool execution, feedback, and monitoring introduce additional interfaces through which failures can propagate across layers. The five-layer abstraction distinguishes the functions and interfaces involved in these propagation paths. These propagation paths motivate examining assumptions about state consistency, observable tool effects, and the conditions under which errors can be detected and corrected.

Evaluation evidence and diagnostic coverage

Evaluation methods for LLM agent frameworks include endpoint judgments, executable checks, trace diagnostics, and operational telemetry. These differences are central to reliability analysis because evaluation protocols determine which aspects of decision-making, state propagation, and action execution become observable. A protocol defines task specifications, success criteria, interaction structure, oracles, metrics, and reporting artifacts, such as traces, logs, seeds, and environment versions. Failures that do not change the judged endpoint require evidence beyond that endpoint for detection and diagnosis.

Mohammadi *et al.* organize agent evaluation along two dimensions: evaluation objectives, including reliability and safety, and evaluation processes, including interaction, data, computation, tooling, and context^[19]. This aligns with an architectural perspective: once systems interact with tools, memory, and

environments, evaluation must expose intermediate steps to support meaningful reliability claims.

Endpoint judgments differ in their procedures. GAIA scores concise final answers by normalized matching^[47]. LLM-as-judge research examines automated assessments, their agreement with human judgments, and judgment biases^[54]. In our architectural analysis, a terminal judgment alone cannot identify an intermediate failure that leaves the judged endpoint unchanged; diagnosis requires additional evidence about the trajectory.

Executable evaluations check specified outputs or states: OSWorld uses task-specific evaluation scripts, and SWE-bench runs tests on proposed repository patches^[44,45]. Controlled environments and checks can support repetition, while the validity of a verdict remains bounded by its requirements and coverage. Process-aware analyses expose different evidence: MAESTRO compares execution call graphs, and Cemri *et al.*^[9] and Ma *et al.*^[36] categorize failures in annotated multi-agent traces. In our synthesis, these records can help locate candidate deviations, but causal attribution still requires suitable controls.

Operational telemetry extends observation across runs. MAESTRO illustrates instrumented repeated-run analysis through execution traces, resource measurements, and call-graph comparisons^[36]. In our synthesis, extending such observations across runs can expose deviations beyond those represented by a single endpoint score. Mohammadi *et al.* discuss trade-offs between realistic and controlled evaluation contexts, together with the cost of repeated assessment^[19]. For telemetry-based studies, this survey further recommends reporting which traces and environment states are retained, because these choices determine what can be inspected or replayed. Across these examples, our synthesis distinguishes the evidence exposed, the judgment procedure, the conditions needed for repetition, and the cost of assessment. These considerations motivate combining executable checks, traces, and monitoring with endpoint scoring. The categories coexist; they are not stages that every evaluation follows.

The approaches in Figure 5 answer different questions about the same execution. An endpoint judgment establishes the assessed outcome, an executable check tests specified states or effects, and a trace links intermediate events. Operational telemetry extends observation across repeated runs. Their combination is useful when the records can be connected to the same requirement and execution; otherwise, additional observations may still leave the origin of a failure unresolved.

Executable, trace-based, and telemetry-supported assessments expose different aspects of architectural reliability. They complement endpoint scoring while introducing distinct costs and limits on reproducibility.

Composition of reliability mechanisms

Mitigation strategies for LLM-agent reliability have expanded from model-guided reasoning patterns to explicit system-level controls. Early agent designs coupled reasoning with actions or verbal feedback, while autonomous-agent prototypes used task lists and iterative loops^[6,21,22,75]. These works established influential control patterns, but they did not provide general guarantees of consistency across extended trajectories.

Tool-learning methods, application frameworks, and interactive environments make different parts of the interaction interface explicit. Toolformer studies learned tool invocation, LangChain exposes composable application components, and WebArena supplies an executable web environment for evaluating autonomous agents^[8,23,74]. These contributions improve the structure and testability of agent interactions, although they do not by themselves provide rollback or continuous runtime observability.

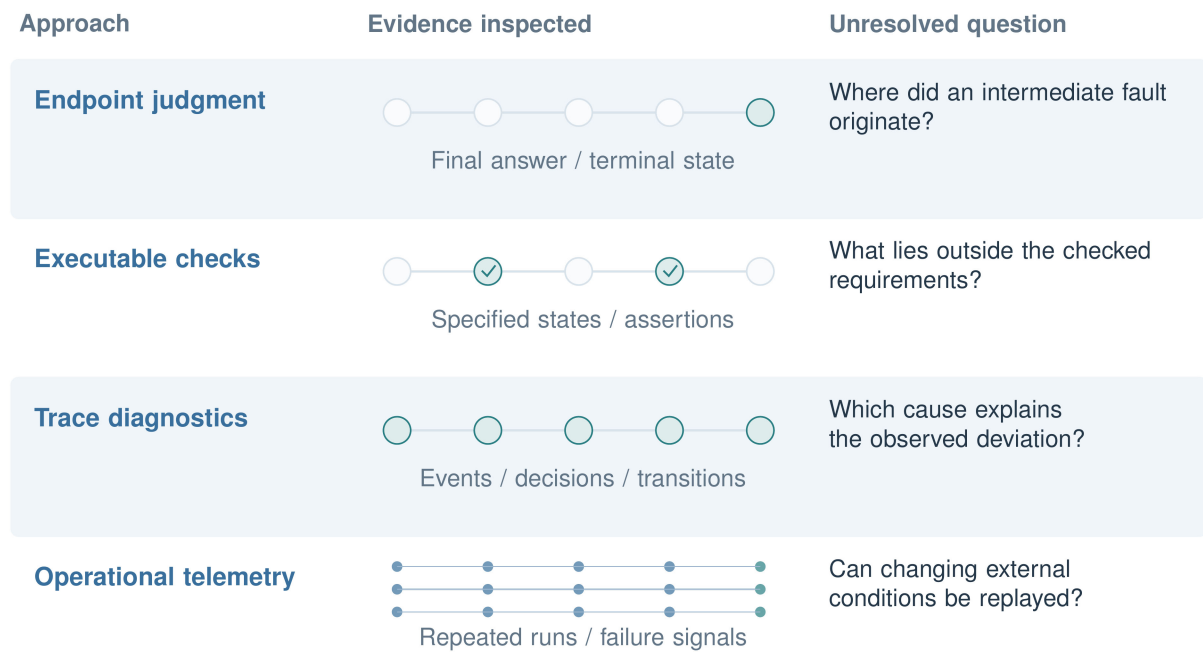


Figure 5. Evaluation approaches, observed evidence, and diagnostic limits. The execution marks illustrate observation scope, not measured performance; approaches can be combined.

Recent strategies introduce dedicated monitoring, runtime enforcement, transactional execution, and graph-based anomaly analysis^[29,34,56]. These mechanisms target failures that are not visible from final outcomes alone, including policy violations, execution faults, state drift, and anomalous multi-agent behavior. Together, these mechanisms motivate a systems-level reliability perspective: mechanisms must coordinate across decision, execution, feedback, and monitoring layers rather than operate as isolated fixes.

Figure 6 compares the six nonexclusive mechanism families by possible placement, observed artifacts, intervention authority, and remaining limitations. This comparison highlights the architectural obligations mitigation introduces. Prompt-level fixes remain useful, but they are insufficient for persistent systems whose failures propagate across modules and time. Protocol safeguards improve interface clarity, while layered runtime mechanisms add observability and enforcement. The central design challenge is balancing reliability coverage against complexity, overhead, and new interface risks.

Prompt-level techniques, protocol safeguards, and runtime controls protect different parts of the agent loop. Combining them requires explicit assumptions about contracts, visibility, and intervention authority.

DISCUSSION

Four findings from the architectural synthesis

The preceding analysis supports four bounded findings. They summarize relationships among the reviewed concepts and examples, not pooled effect sizes, causal estimates, or claims about the prevalence of a mechanism family.

F1: reliability is conditional on an operational requirement and boundary. Sections “FAILURE MODES AND RELIABILITY CONSTRUCTS IN LLM AGENT FRAMEWORKS” and “Benchmarks, datasets, and gold standards for reliability analysis” show why task completion, policy conformance, state consistency, and recoverability cannot be substituted for one another. A reliability claim needs an explicit requirement,

Family / placement	Observed artifacts	Intervention / limit
 Heuristic / rule-based Action; governance	Calls, permissions, policy records	Block or flag violations. Limit: rule coverage.
 Structure-first Across a trace	Graphs, workflows, linked events	Localize deviations. Needs faithful traces.
 Specification-first State; action; governance	Contracts, invariants, labels, pre/postconditions	Enforce specified checks. Unmodeled effects remain.
 Learned monitoring Decision; monitoring	Prompt, state, or trajectory features	Score risk. Action requires a separate control hook.
 Language-native Decision; feedback	Language traces, plans, critiques	Propose revisions. Omitted state is unseen.
 Hybrid Multiple layers	Combined evidence and specifications	Compose interventions. Shared blind spots remain.

Figure 6. Six nonexclusive mechanism families compared by possible placement, observed artifacts, and intervention authority. Each row identifies an operational limit.

operating conditions, and an oracle with stated coverage. This explains why the same final outcome can receive different valid assessments under different constructs.

F2: fault origin, visible symptom, and intervention point can differ. The cross-layer chains in Section “Representative cross-layer failure chains” connect memory, control, execution, feedback, and oversight. They show why locating an error at the tool boundary does not establish that the tool caused it, and why repairing the last visible symptom may leave corrupted state intact. Diagnosis therefore needs linked evidence across transitions, followed by interventions evaluated at both the local boundary and the downstream state.

F3: a mechanism’s protection is bounded by what it can observe and enforce. Sections “Architectural premises and failure propagation” and “RELIABILITY MECHANISMS AND THEIR INTERFACES” connect mechanism families to artifacts, placement, and authority. A specification without an enforcement hook cannot block an action; a monitor without relevant state cannot validate it. Combining mechanisms expands protection only where their interfaces close an identifiable gap, and may leave shared blind spots intact.

F4: successful recovery and low fault incidence are different achievements. Sections “Reliability metrics and operational definitions” and “Representative cross-layer failure chains” show how retries or compensation can produce an acceptable endpoint after earlier failures. Evaluation must record the original deviation, whether it was detected, and whether the correction restored the required state within budget. Otherwise, an outcome gain cannot establish either fewer initial faults or safer recovery.

Challenges and open problems

Outcome-only evaluation limits architectural diagnosis. Outcome-based evaluations provide limited insight into the architectural causes of reliability failures in LLM-based agentic AI systems. A final success score does not by itself identify which architectural layer originated a deviation, and it can leave execution problems outside the endpoint criterion unresolved. As illustrated in Section “FAILURE MODES AND RELIABILITY CONSTRUCTS”, a system may confirm a booking even though no corresponding calendar event exists. A terminal-state check can expose the missing event, while linked tool and state traces are needed to identify the failed transition. This limitation is amplified in long-running settings, where errors may persist as hidden state inconsistencies or uncorrected execution faults. The distinctions in Section “FAILURE MODES AND RELIABILITY CONSTRUCTS” motivate checking state, intent, and recovery explicitly rather than inferring their validity from fluent output.

Missing boundary conditions undermine comparability. The reviewed protocols differ in conditions that affect interpretation: ToolLLM evaluation depends on live API behavior and, in some settings, oracle-retriever access, while τ -bench specifies a simulated user, task policy, and terminal reward^[41,51]. These examples motivate comparing claims under explicitly stated operating conditions. The contrasting evidence types in Section “FAILURE MODES AND RELIABILITY CONSTRUCTS” show why these conditions must be stated before architectural weaknesses can be separated from differences in experimental setup. As LLM-based agentic AI systems become more integrated and long-running, reliability increasingly depends on whether the underlying framework can preserve control and state across extended interactions and recover from silent faults. Without explicit boundary conditions, reliability claims may be incomparable across studies and difficult to reproduce, especially when systems differ in tool permissions, feedback channels, or termination criteria.

Stochasticity and confounding require multi-run and interventionist evaluation. Repeated-run evaluation in MAESTRO examines variation in observed agent executions, and τ -bench measures success across multiple independent trials^[36,51]. These designs motivate reporting variability when model sampling or external interactions can change a run. Reliable evaluation therefore requires multi-run reporting with variance or confidence summaries, rather than single-run or mean-only results, as discussed in Section “Protocol design, representativeness, and validity threats”. Causal attribution also requires targeted ablations and controlled interventions. Otherwise, disabling a component may inadvertently change context length, tool availability, retry behavior, or prompt structure, leading to misleading conclusions. Section “Evaluation evidence and diagnostic coverage” further distinguishes realistic evaluation from controlled assessment and recommends disclosing which traces and environment states are retained for inspection or replay. These factors make it essential to distinguish systematic failures from stochastic variation and protocol artifacts.

Tool ecosystems create silent execution errors and architectural mismatch. Tool-use studies distinguish failures in selection, invocation, execution, and returned results^[13,26]. In our architectural analysis, a fluent summary cannot by itself establish that the corresponding external action completed correctly. For example, a system retrieving real-time data may receive an empty API response yet still generate a plausible but incorrect summary. As illustrated by the propagation analysis in Section “Architectural Interfaces and Failure

Table 8. Research gaps, proposed directions, and evaluation criteria

Gap and finding	Research direction	Evaluation criteria
Underspecified requirements and incomparable boundaries (F1)	Scenario cards and executable protocol specifications	Replicable scores under stated permissions, budgets, stopping rules, and oracle coverage
Local symptoms obscure propagated causes (F2)	Linked traces and controlled cross-layer fault injection	Detection and localization against known injected faults; downstream state checked after intervention
Unobserved state and missing authority limit controls (F3)	State contracts, provenance, and explicit enforcement interfaces	Coverage of protected transitions, missed violations, false blocks, and transfer across interfaces
Endpoint success hides incidence and unsafe retries (F4)	Recovery-aware evaluation with side-effect accounting	Separate fault, detection, restoration, duplicate-effect, latency, and cost measurements

F1-F4 refer to the survey findings in Section “**Four findings from the architectural synthesis**”.

Propagation”, a tool-result mismatch can affect later state or planning before it is detected. This challenge reflects a broader architectural mismatch: language-mediated control is flexible, but reliable tool execution requires precise schemas, complete telemetry, and enforceable interfaces. Section “**Inputs, outputs, and integration patterns**” shows that log-based monitoring, prompt rewriting, and specification enforcement each depend on different assumptions about observability and control. When traces are incomplete or schemas are unstable, these mechanisms may fail to detect or correct execution errors.

Opportunities and future directions

The four findings identify corresponding evaluation targets: explicit operating conditions, linked failure traces, enforceable state and action contracts, and recovery measures that preserve the history of faults. [Table 8](#) maps these targets to proposed studies and evaluation criteria.

Specify the conditions of a reliability claim (F1). The gap identified in Sections “**Benchmarks, datasets, and gold standards for reliability analysis**” and “**Protocol design, representativeness, and validity threats**” is that a score can be reproduced without measuring the intended requirement, or can measure a useful requirement under incomparable conditions. A minimal scenario card should identify the task, construct, threat model, autonomy, permissions, horizon, stopping criteria, budget, and oracle coverage. Executable protocol specifications can then make those conditions inspectable. A useful test is whether independent implementations reproduce the same bounded claim and whether changing one declared condition produces an interpretable difference.

Link observations across the propagation path (F2). The chains in Section “**Representative cross-layer failure chains**” expose an attribution gap between a visible error and its originating transition. Future benchmarks should combine correlated tool events and state snapshots with controlled perturbations such as stale memories, ambiguous acknowledgments, and untrusted feedback. Evaluations should test whether the monitor detects the deviation, identifies the affected transition, and verifies downstream repair. Injected faults provide a known intervention point, but cannot by themselves establish that natural deployment failures have the same distribution. An interaction-centric AIoT perspective similarly motivates checking observed responses and state transitions against specification-derived semantics, while treating physical context as a separate validation concern^[76].

Make state and enforcement contracts explicit (F3). The interface analysis in Section “**Inputs, outputs, and integration patterns**” shows that a mechanism cannot protect state or actions that it neither sees nor controls. State schemas, provenance records, version checks, and explicit action permissions are therefore concrete research targets. Comparing the same mechanism across interfaces with different visibility and authority would test which assumptions are essential for its protection. Reporting false blocks and

integration cost alongside prevented violations is necessary to assess whether greater control remains usable. Related networking work makes these interface requirements concrete through user digital twins with customizable data schemas, attribute-specific updates, and physical-device feedback for network resource management^[77]. TrustTwin provides a complementary AIoT example by comparing specification-guided predictions with executable-twin outcomes before committing candidate actions to physical devices^[78], while a digital-twin framework for autonomous robots similarly demonstrates how runtime monitors can continuously check action consistency and enforce safety and performance properties before actuation^[79].

Evaluate recovery without erasing the original fault (F4). The measurement gap in Section “**Reliability metrics and operational definitions**” is the conflation of final success with low failure incidence. Recovery studies should preserve records of initial faults and separately measure detection, restoration of required state, duplicate or irreversible side effects, and resource cost. Controlled timeout and partial-execution scenarios can test whether retries are safe. Where the environment does not support rollback or idempotency, escalation and bounded abandonment should be evaluated as legitimate outcomes rather than forced into a retry-success metric. Recent recoverable execution frameworks highlight the value of coordinated checkpoint-restore mechanisms for long-horizon agents^[80], whereas work on semantic rollback vulnerabilities emphasize that naive retries can inadvertently duplicate external side effects when exactly-once or retry-deduplication guarantees are not enforced at the tool boundary^[81]. The research agenda follows four gaps: specify the claim, trace propagation, expose enforceable interfaces, and account for recovery separately from incidence. Each direction requires evidence at the relevant state or action boundary as well as at the final outcome.

Threats to validity

Selection, source status, and interpretation. The architectural synthesis supports comparisons of requirements and mechanisms, while its generalizability depends on the breadth and quality of the selected evidence. The analysis draws on studies of architectural failures, evaluation evidence, and interventions. Preprints, conceptual proposals, documentation, and empirical studies provide different kinds of support. Incomplete descriptions can leave publication status, implementation details, or evaluation conditions unresolved. Layer assignments and the representative failure chains are interpretive analyses, and the latter require empirical testing before supporting causal or frequency claims. These limitations constrain generalization of the four findings and preclude an effectiveness ranking across mechanism families.

Ambiguity in the construct of reliability. A central threat to this survey is the variation in how prior work defines and operationalizes “reliability.” Some studies emphasize task completion, while others focus on policy compliance, tool-use safety, state consistency, recoverability, or governance. As discussed in Section “**FAILURE MODES AND RELIABILITY CONSTRUCTS**”, these constructs imply different assumptions about the consistency and verifiability of system state, control, and feedback. This ambiguity makes synthesis difficult: reported improvements may not refer to the same underlying reliability property, and cross-study comparisons may overgeneralize beyond the measured construct.

Internal validity risks from hidden oracles and contamination. Training-evaluation contamination is an established concern in language-model evaluation^[53]. In this survey, we additionally treat undeclared task-specific information exposed through prompts, tool outputs, documentation, or scaffolds as a potential confound. Section “**Protocol design, representativeness, and validity threats**” recommends disclosing those information channels so that a reported score can be interpreted under its actual conditions. Uncontrolled nondeterminism and selective reporting further complicate attribution. As a result, apparent reliability gains may reflect evaluation artifacts rather than substantive architectural improvements.

Outcome metrics masking architectural violations. Outcome metrics can overstate reliability when recovery mechanisms conceal upstream failures. A system may achieve high task success while repeatedly violating assumptions about control, state consistency, or tool execution. Section “**Reliability metrics and operational definitions**” shows that robust recovery, retries, or consensus can improve final scores without reducing the true incidence of faults. This creates a synthesis risk: techniques that improve apparent outcomes may be favored even if they leave the underlying framework brittle or shift failures into less observable layers.

Ground truth and replayability constraints in long-running systems. Ground-truth fidelity in long-running systems depends on oracle coverage, judgment validity, and access to the state being assessed. As discussed in Section “**Evaluation evidence and diagnostic coverage**”, the retained traces and environment states determine which claims can be inspected or replayed, affecting how architectural failures are separated from evaluator variance or changing deployment conditions. Evaluations that better match the target deployment may strengthen external validity, while causal interpretation remains limited when runs cannot be replayed, instrumented, or compared under controlled conditions.

CONCLUSION

This survey presents an architectural synthesis of reliability in LLM-based agentic AI. Five layers locate the functions involved in a failure, while four premises - semantic fidelity, instruction stability, observability, and recoverability - make their operational assumptions explicit. The distinction between constructs, requirements, failure modes, observables, metrics, and oracles provides a common basis for comparing otherwise different reliability claims. Four findings organize the synthesis: reliability is conditional on a stated requirement and operating boundary; fault origin, symptom, and intervention point can differ; protection is bounded by visibility and enforcement authority; and successful recovery does not imply low initial fault incidence. The analytical cross-layer examples show how these relationships can arise, while the mechanism and evaluation comparisons identify what evidence would be needed to assess them in a concrete system. The survey does not establish their prevalence or provide a quantitative ranking of existing methods. Progress therefore requires explicit scenario specifications, linked traces, state and action contracts, and recovery measurements that retain the history of faults and side effects. These artifacts connect an architectural claim to observable evidence and clarify when a local safeguard can support an end-to-end reliability requirement.

DECLARATIONS

Authors' contributions

Conceptualization: Ma, X.; Tian, C.

Methodology: Ma, X.; Zhang, Y.

Investigation: Ma, X.; Wang, J.

Data curation: Ma, X.; Wang, J.

Formal analysis: Ma, X.

Writing original draft: Ma, X.

Writing review and editing: Ma, X.; Wang, J.; Zhang, Y.; Tian, C.

Validation: Wang, J.; Zhang, Y.

Supervision: Tian, C.

Project administration: Tian, C.

All authors read and approved the final manuscript.

Availability of data and materials

Not applicable.

AI and AI-assisted tools statement

During the preparation of this manuscript, OpenAI's ChatGPT with GPT-5.6 Sol (2026-07-09) was used solely for language editing and refinement of figure layouts and color schemes. The tool did not influence the

study design, data collection, analysis, interpretation, or scientific content of the work. All authors reviewed the AI-assisted content and take full responsibility for the accuracy, integrity, and final content of the manuscript.

Financial support and sponsorship

None.

Conflicts of interest

All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Copyright

© The Author(s) 2026.

REFERENCES

1. OpenAI. OpenAI Agents SDK. <https://github.com/openai/openai-agents-python> (accessed 2026-10-28).
2. Kumayama, K. D.; Chiruvolu, P.; Weiss, D. AI agents: greater capabilities and enhanced risks. Reuters Legal News, 2025. 22 April 2025. <https://www.reuters.com/legal/legalindustry/ai-agents-greater-capabilities-enhanced-risks-2025-04-22/> (accessed 2026-10-28).
3. Wang, L.; Ma, C.; Feng, X.; et al. A survey on large language model based autonomous agents. *Front Comput Sci* **2024**, *18*, 186345. DOI
4. Xu, D.; Cao, J.; Mo, G.; et al. LiveFMBench: unveiling the power and limits of agentic workflows in specification generation. arXiv 2026;arXiv:2605.01394. Available online: <https://doi.org/10.48550/arXiv.2605.01394> [accessed 28 October 2026].
5. Lu, X.; Sun, W.; Zhang, Y.; et al. Requirements development and formalization for reliable code generation: a multi-agent vision. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2025 Nov 16-20; Seoul, Korea. IEEE; 2025. pp. 3932-7. DOI
6. Yao, S.; Zhao, J.; Yu, D.; et al. ReAct: synergizing reasoning and acting in language models. arXiv 2023;arXiv:2210.03629. Available online: <https://doi.org/10.48550/arXiv.2210.03629> [accessed 28 October 2026].
7. Hu, M.; Chen, T.; Chen, Q.; Mu, Y.; Shao, W.; Luo, P. HiAgent: hierarchical working memory management for solving long-horizon agent tasks with large language model. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); 2025 Jun; Vienna, Austria. Stroudsburg, PA, USA: Association for Computational Linguistics; 2025. pp. 32779-98. DOI
8. Zhou, S.; Xu, F. F.; Zhu, H.; et al. WebArena: a realistic web environment for building autonomous agents. In: Proceeding of Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. Curran Associates, Inc.; 2024. pp. 1-22. Available from: https://proceedings.iclr.cc/paper_files/paper/2024/file/4410c0711e9154a7a2d26f9b3816d1ef-Paper-Conference.pdf (accessed 2026-10-28).
9. Cemri, M.; Pan, M. Z.; Yang, S.; et al. Why Do Multi-Agent LLM Systems Fail? In: Belgrave D, Zhang C, Lin H, Pascanu R, Koniusz P, Ghassemi M, Chen N, Editors. Advances in Neural Information Processing Systems 38; 2025 Nov 30-Dec 7; San Diego, California, USA and Mexico City, Mexico. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2025. pp. 135676-729. DOI
10. Xie, Y.; Zhu, C.; Zhang, X.; et al. From spark to fire: modeling and mitigating error cascades in LLM-based multi-agent collaboration. arXiv 2026;arXiv:2603.04474. Available online: <https://doi.org/10.48550/arXiv.2603.04474> [accessed 28 October 2026].
11. Zhou, J.; Chen, J.; Lu, Q.; Zhao, D.; Zhu, L. SHIELDA: structured handling of exceptions in LLM-driven agentic workflows. arXiv 2025;arXiv:2508.07935. Available online: <https://doi.org/10.48550/arXiv.2508.07935> [accessed 28 October 2026].
12. Arike, R.; Donoway, E.; Bartsch, H.; Hobbhahn, M. Technical report: evaluating goal drift in language model agents. arXiv 2025;arXiv:2505.02709. Available online: <https://doi.org/10.48550/arXiv.2505.02709> [accessed 28 October 2026].
13. Huang, D.; Malwe, G.; Wang, Z. When agents fail to act: a diagnostic framework for tool invocation reliability in multi-agent LLM systems. In: 2026 9th International Conference on Artificial Intelligence and Big Data (ICAIBD); 2026 May 29-31; Chengdu, China. IEEE; 2026. pp. 492-7. DOI

14. Huang, S.; Fang, Z.; Chen, Z.; et al. CRITICTOOL: evaluating self-critique capabilities of large language models in tool-calling error scenarios. In: Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing; 2025 Oct; Suzhou, China. Stroudsburg, PA, USA: Association for Computational Linguistics; 2025. pp. 26683-92. DOI
15. Zhu, X.; Wu, J.; Zhang, X.; et al. An empirical study of bugs in modern LLM agent frameworks. arXiv 2026;arXiv:2602.21806v1. Available online: <https://doi.org/10.48550/arXiv.2602.21806> [accessed 28 October 2026].
16. Shi, J.; Yuan, Z.; Tie, G.; Zhou, P.; Gong, N.; Sun, L. Prompt injection attack to tool selection in LLM agents. In: Network and Distributed System Security Symposium; San Diego, CA, USA. Reston, VA: Internet Society; 2026. DOI
17. Qu, C.; Dai, S.; Wei, X.; et al. Tool learning with large language models: a survey. *Front. Comput. Sci.* **2025**, *19*, 198343. DOI
18. Yehudai, A.; Eden, L.; Li, A.; et al. A survey on evaluation of LLM-based agents. In: Findings of the Association for Computational Linguistics: ACL 2026; 2026 Jun; San Diego, California, United States. Stroudsburg, PA, USA: Association for Computational Linguistics; 2026. pp. 26690-714. DOI
19. Mohammadi, M.; Li, Y.; Lo, J.; Yip, W. Evaluation and benchmarking of LLM agents: a survey. In: KDD '25: The 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining; Toronto ON Canada. New York, NY, USA: ACM; 2025. pp. 6129-39. DOI
20. Durante, Z.; Huang, Q.; Wake, N.; et al. Agent AI: surveying the horizons of multimodal interaction, arXiv 2024;arXiv:2401.03568v2. Available online: <https://doi.org/10.48550/arXiv.2401.03568> [accessed 28 October 2026].
21. Significant Gravitass. AutoGPT. <https://github.com/Significant-Gravitas/AutoGPT> (accessed 2026-10-28).
22. Nakajima, Y. BabyAGI, 2023. Original 2023 project, archived September 2024. https://github.com/yoheinakajima/babyagi_archive (accessed 2026-10-28).
23. LangChain contributors. LangChain. <https://github.com/langchain-ai/langchain> (accessed 2026-10-28).
24. Pathak, D.; George, F.; Kumar, H.; et al. Detecting silent failures in multi-agentic AI trajectories (Work In Progress Paper). In: ICPE '26: 17th ACM/SPEC International Conference on Performance Engineering; Florence Italy. New York, NY, USA: ACM; 2026. pp. 33-9. DOI
25. Rabanser, S.; Kapoor, S.; Kirgis, P.; Liu, K.; Utpala, S.; Narayanan, A. Towards a science of AI agent reliability. In: Proceedings of the 43rd International Conference on Machine Learning (Volume 306); 2026 July 6-11; Seoul, Korea. PMLR; 2026. pp. 102769-820. <https://raw.githubusercontent.com/mlresearch/v306/main/assets/rabanser26a/rabanser26a.pdf>.
26. Winston, C.; Just, R. A Taxonomy of failures in tool-augmented LLMs. In: 2025 IEEE/ACM International Conference on Automation of Software Test (AST); 2025 Apr 28-29; Ottawa, ON, Canada. IEEE; 2025. pp. 125-35. DOI
27. Sapkota, R.; Roumeliotis, K. I.; Karkee, M. AI agents vs. agentic AI: a conceptual taxonomy, applications and challenges. *Inform Fusion.* **2026**, *126*, 103599. DOI
28. Bullwinkel, B.; Minnich, A.; Chawla, S.; et al. Lessons from red teaming 100 generative AI products. arXiv 2025;arXiv:2501.07238. Available online: <https://doi.org/10.48550/arXiv.2501.07238> [accessed 28 October 2026].
29. Wang, H.; Poskitt, C. M.; Sun, J. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. In: ICSE '26: 2026 IEEE/ACM 48th International Conference on Software Engineering; Rio de Janeiro Brazil. New York, NY, USA: ACM; 2026. pp. 2938-50. DOI
30. Syros, G.; Suri, A.; Ginesin, J.; Nita-rotaru, C.; Oprea, A. SAGA: a security architecture for governing AI agentic systems. In: Network and Distributed System Security Symposium 2026; 2026 Feb 23-27; San Diego, CA, USA. Reston, VA: Internet Society; 2026. DOI
31. Khan, R.; Joyce, D.; Habiba, M. AGENTS SAFE: a unified framework for ethical assurance and governance in agentic. arXiv 2025;arXiv:2512.03180. Available online: <https://doi.org/10.48550/arXiv.2512.03180> [accessed 28 October 2026].
32. Zheng, L.; Chen, J.; Yin, Q.; Zhang, J.; Zeng, X.; Tian, Y. Rethinking the reliability of multi-agent system: a perspective from byzantine fault tolerance. *AAAI.* **2026**, *40*, 35012-20. DOI
33. Doshi, A.; Hong, Y.; Xu, C.; Kang, E.; Kapravelos, A.; Kästner, C. Towards verifiably safe tool use for LLM agents. In: ICSE-NIER '26: 2026 IEEE/ACM 48th International Conference on Software Engineering; 2026 Apr 12-18; Rio de Janeiro, Brazil. New York, NY, USA: ACM; 2026. pp. 201-5. DOI
34. Li, P.; Zou, X.; Wu, Z.; et al. SAFEFLOW: a principled protocol for trustworthy and transactional autonomous agent systems. arXiv 2025;arXiv:2506.07564v3. Available online: <https://doi.org/10.48550/arXiv.2506.07564> [accessed 28 October 2026].
35. Zhan, Q.; Liang, Z.; Ying, Z.; Kang, D. InjecAgent: benchmarking indirect prompt injections in tool-integrated large language model agents. In: Findings of the Association for Computational Linguistics ACL 2024; 2024 Aug 11-16; Bangkok, Thailand and virtual meeting. Stroudsburg, PA, USA: Association for Computational Linguistics; 2024. pp. 10471-506. DOI
36. Ma, T.; Chen, Y.; Anand, V.; et al. MAESTRO: multi-agent evaluation suite for testing, reliability, and observability. arXiv 2026;arXiv:2601.00481. Available online: <https://doi.org/10.48550/arXiv.2601.00481> [accessed 28 October 2026].

37. Andriushchenko, M.; Souly, A.; Dziemian, M.; et al. AgentHarm: a benchmark for measuring harmfulness of LLM agents. In: Proceedings of the Thirteenth International Conference on Learning Representations; 2025 Apr 24-28; Singapore. International Conference on Learning Representations. 2025. pp. 1-36. https://proceedings.iclr.cc/paper_files/paper/2025/file/c493d23af93118975cdbe32cbe7323f5-Paper-Conference.pdf.
38. Shen, Z.; Dai, J.; Zhang, Y.; Yang, M. Security debt in LLM agent applications: a measurement study of vulnerabilities and mitigation trade-offs. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2025 Nov 16-20; Seoul, Korea. IEEE; 2025. pp. 559-70. DOI
39. Gajjar, J.; Subramaniakuppasamy, K.; Puthal, R.; Ranaware, K. SecureFixAgent: a hybrid LLM agent for automated python static vulnerability repair. In: 2025 International Conference on Machine Learning and Applications (ICMLA); 2025 Dec 3-5; Boca Raton, FL, USA. IEEE; 2025. pp. 1390-5. DOI
40. Luo, H.; Dai, S.; Ni, C.; et al. AgentAuditor: human-level safety and security evaluation for LLM agents. In: Belgrave D, Zhang C, Lin H, Pascanu R, Koniusz P, Ghassemi M, Chen N, Editors. Advances in Neural Information Processing Systems 38; 2025 Nov 30-Dec 7; San Diego, California, USA and Mexico City, Mexico. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2025. pp. 48268-325. DOI
41. Qin, Y.; Liang, S.; Ye, Y.; et al. ToolLLM: facilitating large language models to master 16000+ real-world APIs. In: Proceedings of the Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. International Conference on Learning Representations. 2024. pp. 1-23. Available from: https://proceedings.iclr.cc/paper_files/paper/2024/file/28e50ee5b72e90b50e7196fde8ea260e-Paper-Conference.pdf.
42. Patil, S. G.; Mao, H.; Yan, F.; et al. The Berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In: Proceedings of the 42nd International Conference on Machine Learning (Volume 267); 2025 Jul 13-19. PMLR, 2025; pp. 48371-92. <https://proceedings.mlr.press/v267/patil25a.html>.
43. He, H.; Yao, W.; Ma, K.; et al. WebVoyager: building an end-to-end web agent with large multimodal models. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers); 2024 Aug 11-16; Bangkok, Thailand. Stroudsburg, PA, USA: Association for Computational Linguistics; 2024. pp. 6864-90. DOI
44. Xie, T.; Zhang, D.; Chen, J.; et al. OSWorld: benchmarking multimodal agents for open-ended tasks in real computer environments. In: Globerson A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak J, Zhang C, Editors. Advances in Neural Information Processing Systems 37; 2024 Dec 10-15; Vancouver, BC, Canada. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2024. pp. 52040-94. DOI
45. Jimenez, C. E.; Yang, J.; Wettig, A.; et al. SWE-bench: can language models resolve real-world GitHub issues? In: Proceeding of Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. Curran Associates, Inc.; 2024. https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.
46. Liu, X.; Yu, H.; Zhang, H.; et al. AgentBench: evaluating LLMs as agents. In: Proceeding of Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. Curran Associates, Inc.; 2024. https://proceedings.iclr.cc/paper_files/paper/2024/file/e9df36b21ff4ee211a8b71ee8b7e9f57-Paper-Conference.pdf.
47. Mialon, G.; Fourrier, C.; Swift, C.; Wolf, T.; LeCun, Y.; Scialom, T. GAIA: a benchmark for general AI assistants. In: Proceeding of Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. Curran Associates, Inc.; 2024. Available from: https://proceedings.iclr.cc/paper_files/paper/2024/file/25ae35b5b1738d80f1f03a8713e405ec-Paper-Conference.pdf.
48. Ma, C.; Zhang, J.; Zhu, Z.; et al. AgentBoard: an analytical evaluation board of multi-turn LLM agents. In: Globerson A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak J, Zhang C, Editors. Advances in Neural Information Processing Systems 37; 2024 Dec 10-15; Vancouver, BC, Canada. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2024. pp. 74325-62. DOI
49. 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: Globerson A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak J, Zhang C, Editors. Advances in Neural Information Processing Systems 37; 2024 Dec 10-15; Vancouver, BC, Canada. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2024. pp. 82895-920. DOI
50. Wu, D.; Wang, H.; Yu, W.; Zhang, Y.; Chang, K. W.; Yu, D. LongMemEval: benchmarking chat assistants on long-term interactive memory. In: Proceedings of the Thirteenth International Conference on Learning Representations; 2025 Apr 24-28; Singapore. International Conference on Learning Representations. 2025. https://proceedings.iclr.cc/paper_files/paper/2025/file/d813d324dbf0598bbdc9c8e79740ed01-Paper-Conference.pdf.
51. Yao, S.; Shinn, N.; Razavi, P.; Narasimhan, K. τ -bench: a benchmark for tool-agent-user interaction in real-world domains. In: Proceedings of the Thirteenth International Conference on Learning Representations; 2025 Apr 24-28; Singapore. International Conference on Learning Representations. 2025. https://proceedings.iclr.cc/paper_files/paper/2025/file/1b126cc38b8638e07bef37e7b2bb72bf-Paper-Conference.pdf.

52. Barres, V.; Dong, H.; Ray, S.; Si, X.; Narasimhan, K. τ^2 -bench: evaluating conversational agents in a dual-control environment. arXiv 2025;arXiv:2506.07982v1. Available online: <https://doi.org/10.48550/arXiv.2506.07982> [accessed 28 October 2026].
53. Liang, P.; Bommasani, R.; Lee, T.; et al. Holistic evaluation of language models. *Trans Mach Learn Res* 2023. <https://openreview.net/forum?id=iO4LZibEqW>.
54. Zheng, L.; Chiang, W.; Sheng, Y.; et al. Judging LLM-as-a-judge with MT-bench and chatbot arena. In: Oh A, Naumann T, Globerson A, Saenko K, Hardt M, Levine S, Editors. Advances in Neural Information Processing Systems 36; 2023 Dec 10-16; New Orleans, Louisiana, USA. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2023. pp. 46595-623. DOI
55. Hu, H.; Chen, P.; Zhao, Y.; Chen, Y. AgentSentinel: an end-to-end and real-time security defense framework for computer-use agents. In: CCS '25: ACM SIGSAC Conference on Computer and Communications Security; Taipei Taiwan. New York, NY, USA: ACM; 2025. pp. 3535-49. DOI
56. He, X.; Wu, D.; Zhai, Y.; Sun, K. SentinelAgent: graph-based anomaly detection in multi-agent systems. arXiv 2025;arXiv:2505.24201. Available online: <https://doi.org/10.48550/arXiv.2505.24201> [accessed 28 October 2026].
57. Ro, Y.; Qiu, H.; Goiri, Í.; et al. Sherlock: reliable and efficient agentic workflow execution. arXiv 2025; arXiv:2511.00330. Available online: <https://doi.org/10.48550/arXiv.2511.00330> [accessed 28 October 2026].
58. Liu, F.; Zhang, Y.; Luo, J.; et al. Make agent defeat agent: automatic detection of taint-style vulnerabilities in LLM-based agents. In: 34th USENIX Security Symposium (USENIX Security 25); 2025 Aug 13-15; Seattle, WA, USA: USENIX Association, 2025; pp. 3767-86. <https://www.usenix.org/conference/usenixsecurity25/presentation/liu-fengyu>.
59. Costa, M.; Köpf, B.; Kolluri, A.; et al. Securing AI agents with information-flow control. arXiv 2025; arXiv:2505.23643. Available online: <https://doi.org/10.48550/arXiv.2505.23643> [accessed 28 October 2026].
60. DeBenedetti, E.; Shumailov, I.; Fan, T.; et al. Defeating prompt injections by design. arXiv 2025; arXiv:2503.18813. Available online: <https://doi.org/10.48550/arXiv.2503.18813> [accessed 28 October 2026].
61. Chen, S.; Piet, J.; Sitawarin, C.; Wagner, D. StruQ: defending against prompt injection with structured queries. 34th USENIX Security Symposium (USENIX Security 25) ; 2025 Aug 13-15; Seattle, WA, USA: USENIX Association, 2025; pp. 2383-400. Available from: <https://www.usenix.org/conference/usenixsecurity25/presentation/chen-sizhe>.
62. Chen, S.; Zharmagambetov, A.; Mahlouljifar, S.; Chaudhuri, K.; Wagner, D.; Guo, C. SecAlign: defending against prompt injection with preference optimization. In: CCS '25: ACM SIGSAC Conference on Computer and Communications Security; Taipei Taiwan. New York, NY, USA: ACM; 2025. pp. 2833-47. DOI
63. Shi, S.; Wang, X.; Zhang, C.; et al. Think twice before you act: protecting LLM agents against tool description poisoning via isolated planning. In: Proceedings of the 43rd International Conference on Machine Learning (Volume 306); 2026 July 6-11; Seoul, Korea. PMLR; 2026. pp. 112320-37. <https://openreview.net/forum?id=jiNw5AgBbw>.
64. Li, K.; Zhao, T.; Zhou, W.; Hu, S. DORA: Dynamic optimization prompt for continuous reflection of LLM-based agent. In: Proceedings of the 31st International Conference on Computational Linguistics; 2025 Jan 19-24; Abu Dhabi, UAE: Association for Computational Linguistics, 2025; pp. 7546-57. <https://aclanthology.org/2025.coling-main.504/>.
65. Gou, Z.; Shao, Z.; Gong, Y.; et al. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. In: Proceedings of the Twelfth International Conference on Learning Representations; 2024 May 7-11; Vienna, Austria. International Conference on Learning Representations. 2024. https://proceedings.iclr.cc/paper_files/paper/2024/file/fef126561bbf9d4467dbb8d27334b8fe-Paper-Conference.pdf.
66. Zhao, A.; Huang, D.; Xu, Q.; Lin, M.; Liu, Y.; Huang, G. ExpeL: LLM agents are experiential learners. *AAAI*. **2024**, *38*, 19632-42. DOI
67. Patel, K.; Surendira, S.; George, J.; Kapale, S. The six sigma agent: achieving enterprise-grade reliability in LLM systems through consensus-driven decomposed execution. arXiv 2026; arXiv:2601.22290. Available online: <https://doi.org/10.48550/arXiv.2601.22290> [accessed 28 October 2026].
68. Liu, B.; Jiang, Y.; Zhang, X.; et al. LLM+P: empowering large language models with optimal planning proficiency. arXiv 2023; arXiv:2304.11477v3. Available online: <https://doi.org/10.48550/arXiv.2304.11477> [accessed 28 October 2026].
69. Zhou, A.; Yan, K.; Shlapentokh-Rothman, M.; Wang, H.; Wang, Y. X. Language agent tree search unifies reasoning, acting, and planning in language models. In: Proceedings of the 41st International Conference on Machine Learning (Volume 235); 2024 Jul 21-27; Vienna, Austria. PMLR, 2024; pp. 62138-60. <https://proceedings.mlr.press/v235/zhou24r.html>.
70. Packer, C.; Wooders, S.; Lin, K.; et al. MemGPT: towards LLMs as operating systems. arXiv 2024; arXiv:2310.08560v2. Available online: <https://doi.org/10.48550/arXiv.2310.08560> [accessed 28 October 2026].
71. Zhong, W.; Guo, L.; Gao, Q.; Ye, H.; Wang, Y. MemoryBank: enhancing large language models with long-term memory. *AAAI*. **2024**, *38*, 19724-31. DOI

72. Xu, W.; Liang, Z.; Mei, K.; Gao, H.; Tan, J.; Zhang, Y. A-Mem: agentic memory for LLM agents. In: Belgrave D, Zhang C, Lin H, Pascanu R, Koniusz P, Ghassemi M, Chen N, Editors. *Advances in Neural Information Processing Systems 38*; 2025 Nov 30-Dec 7; San Diego, California, USA and Mexico City, Mexico. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2025. pp. 20004-31. DOI
73. Wang, Z. Z.; Mao, J.; Fried, D.; Neubig, G. Agent workflow memory. In: *Proceedings of the 42nd International Conference on Machine Learning (Volume 267)*; 2025 Jul 13-19. PMLR, 2025; pp. 63897-911. Available from: <https://proceedings.mlr.press/v267/wang25bx.html>.
74. Schick, T.; Dwivedi-Yu, J.; Dessi, R.; et al. Toolformer: language models can teach themselves to use tools. In: Oh A, Naumann T, Globerson A, Saenko K, Hardt M, Levine S, Editors. *Advances in Neural Information Processing Systems 36*; 2023 Dec 10-16; New Orleans, Louisiana, USA. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2023. pp. 68539-51. DOI
75. Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: language agents with verbal reinforcement learning. In: Oh A, Naumann T, Globerson A, Saenko K, Hardt M, Levine S, Editors. *Advances in Neural Information Processing Systems 36*; 2023 Dec 10-16; New Orleans, Louisiana, USA. San Diego, California, USA: Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2023. pp. 8634-52. DOI
76. Ma, X. Towards Trustworthy AIoT systems: an interaction-centric perspective. In: *MobiSys Companion '26: 24th Annual International Conference on Mobile Systems, Applications and Services Companion*; University of Cambridge Cambridge United Kingdom. New York, NY, USA: ACM; 2026. pp. 148-9. DOI
77. Zhou, C.; Hu, S.; Gao, J.; Huang, X.; Zhuang, W.; Shen, X. User-centric immersive communications in 6G: a data-oriented framework via digital twin. *IEEE Wireless Commun.* 2025, 32, 122-9. DOI
78. Ma, X.; Lu, H.; Dai, X.; Zhang, Y. TrustTwin: world-model-enhanced twins for trustworthy AIoT decision making. In: *2026 IEEE/CIC International Conference on Communications in China (ICCC)*; 2026 Aug 7-9; Wuhan, China. IEEE; 2026. pp. 271-6. DOI
79. Betzer, J. S.; Boudjadar, J.; Frasher, M.; Talasila, P. Digital twin enabled runtime verification for autonomous mobile robots under uncertainty. In: *2024 28th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*; 2024 Oct 7-9; Urbino, Italy. IEEE; 2024. pp. 10-7. DOI
80. Zhuang, Y.; Chen, K.; Duan, Y.; et al. AgentRewind: recoverable execution for long-horizon LLM agents. arXiv 2026; arXiv:2608.14380. Available online: <https://doi.org/10.48550/arXiv.2608.14380> [accessed 28 October 2026].
81. Zheng, Y.; Yang, Y.; Zhang, W.; Quinn, A. ACRFence: preventing semantic rollback attacks in agent checkpoint-restore. arXiv 2026; arXiv:2603.20625. Available online: <https://doi.org/10.48550/arXiv.2603.20625> [accessed 28 October 2026].

Disclaimer/Publisher's Note: All statements, opinions, and data contained in this publication are solely those of the individual author(s) and contributor(s) and do not necessarily reflect those of OAE and/or the editor(s). OAE and/or the editor(s) disclaim any responsibility for harm to persons or property resulting from the use of any ideas, methods, instructions, or products mentioned in the content.



© The Author(s) 2026. Open Access This article is licensed under a Creative Commons Attribution 4.0 International License (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, sharing, adaptation, distribution and reproduction in any medium or format, for any purpose, even commercially, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.