# agent survey (LLM agents) LaTeX survey

January 19, 2026

## Contents

**Abstract**

Large language model (LLM) agents couple a foundation model with a control loop that iteratively observes, decides, and acts through tools or environments. This looped setting makes agents qualitatively different from single-shot prompting: success depends on interface contracts, orchestration policies, and evaluation protocols as much as on model capability. We survey recent work on tool-using LLM agents with an evidence-first lens, organizing the design space from foundations and interfaces to planning, memory, adaptation, multi-agent coordination, and evaluation and risk. Throughout, we emphasize what claims are comparable across papers (and which are not) by explicitly calling out benchmark/protocol choices and security threat models. Our goal is to help practitioners reason about agent design decisions and help researchers identify evaluation gaps that currently limit reproducible progress [91, 62, 58, 53, 100].

# 1 Introduction

LLM agents are increasingly deployed as closed-loop systems that must decide what to do next, call tools, recover from failures, and complete multi-step tasks under cost and latency constraints. Compared to pure prompting, agent performance depends on the *contract* between the model and its environment: what constitutes an action, what feedback is observed, and how errors are surfaced and handled. Early designs such as ReAct make this loop explicit by interleaving reasoning traces with tool-grounded actions [91], while tool-use training methods aim to expand the action space beyond natural language by teaching explicit tool calls [62]. Reflection-style loops further highlight that "capability" is not a single number in agentic settings: it can be improved (or destabilized) by how the system stores state, critiques itself, and selects actions over time [68].

We adopt a systems-oriented definition: an LLM agent is an LLM embedded in a control loop that (i) maintains an internal state, (ii) selects actions from a defined action space (tool calls, code edits, web interactions, environment actions), and (iii) receives observations that influence subsequent decisions. This definition separates agent work from adjacent areas such as one-off tool invocation or retrieval-only augmentation, and it aligns with how recent surveys frame agentic LLM systems [58, 73]. Under this lens, the central question becomes less "which model is best?" and more "which loop and interface choices make behavior reliable, debuggable, and comparable under a stated protocol?"

This framing matters because agent results are frequently protocol-sensitive. Cost-aware evaluations show that planning depth, retries, and verification can move a system along a sharp success–cost frontier, so headline success rates can be misleading if budgets are not normalized [46]. Similarly, evaluation suites that model realistic tool ecosystems emphasize that tool access, tool noise, and logging policies can dominate outcomes, which makes "agent progress" inseparable from benchmark design [35, 99].

Accordingly, this survey emphasizes two objects that are often under-specified in agent papers: evaluation protocols and threat models. Protocol taxonomies make clear why comparisons drift across benchmarks and why success-only reporting is insufficient for synthesis [53]. Security-oriented benchmarks and monitor red-teaming methods further show that tool use introduces new attack surfaces (prompt/tool injection, tool misuse, data exfiltration) and that defenses are best evaluated as end-to-end system properties rather than as model-only behaviors [18, 100, 33].

Evidence policy: we primarily rely on arXiv metadata/abstracts and structured notes to build the taxonomy and evidence packs. This is sufficient to map design axes and identify protocol details, but some quantitative claims and threat-model specifics can remain unclear without fulltext verification. We therefore (i) treat benchmark/protocol descriptions as first-class evidence, (ii) prefer conservative phrasing when protocol details are missing, and (iii) avoid over-interpreting isolated numeric claims unless the evaluation setup is explicit [53, 18, 33].

The paper proceeds from interface contracts to system components, then to evaluation and risk. We first define agent-loop semantics and tool interfaces (Foundations & Interfaces), then examine core components that dominate behavior in practice—planning/reasoning loops and memory/retrieval mechanisms. Next, we review adaptation and coordination (self-improvement and multi-agent protocols). We conclude by discussing evaluation methodology and governance constraints that reshape what it means to deploy agents safely and reproducibly [35, 100].

## 2 Related Work

Our scope sits at the intersection of LLM reasoning methods, tool use, and interactive evaluation. On the reasoning side, agent loops are often built on structured deliberation patterns that extend beyond single-pass chain-of-thought. For example, ReAct combines reasoning with grounded action to make intermediate tool calls explicit [91], while tree/search-style approaches explore multiple candidates or trajectories before committing to an action sequence [92]. These methods motivate protocol-aware comparisons: changing the deliberation policy can change both performance and cost.

Planning-specific work sharpens this point by treating "better reasoning" as a budgeted systems choice. Cost-focused benchmarks measure success alongside queries/tokens and highlight that many planning gains come from spending more interaction budget (more branches, more verification, more retries) rather than from algorithmic improvements at a fixed budget [46]. API-call oriented evaluation further stresses that agent performance is bounded by tool-call validity and repair behavior, not only by the quality of intermediate thoughts [54, 55].

On the tool-use side, training and interface work expands what counts as an action. Toolformer demonstrates that a model can self-supervise tool use by inserting tool calls into text and learning when and how to invoke external APIs [62]. More recent work focuses on tool selection, routing, and robustness under noisy tool outputs or tool availability changes, which becomes central once agents must orchestrate multiple tools over long horizons [13, 47]. These lines of work inform an "interface contract" framing: a tool schema and orchestration policy jointly determine what failures are observable and what claims are verifiable.

Several surveys summarize the emerging landscape of agentic LLM systems and provide high-level taxonomies [58, 73]. We align with their definition of agents as looped systems, but we put more weight on two aspects that repeatedly determine whether synthesis is meaningful: (i) evaluation protocols (task suites, budgets, tool access, logging), and (ii) threat models for deployed agents. This yields an organization closer to a systems paper: interfaces and protocols first, followed by component-level design levers, then evaluation and risk.

Related threads also study how agents change over time and how multiple agents interact. Self-improvement methods revise prompts or policies using self-generated feedback, while multi-agent protocols use debate, aggregation, or role specialization as verification layers; both lines of work stress that results are sensitive to evaluator design, message budgets, and diversity assumptions [82, 10]. We treat these mechanisms as part of the broader agent loop, because they affect not only outcomes but also failure modes and governance requirements.

Evaluation and safety are increasingly treated as their own research threads. Dedicated evaluation overviews propose taxonomies of objectives (capability, reliability, safety) and emphasize protocol heterogeneity as a core obstacle to synthesis [53]. Security-oriented benchmarks and red-teaming methods show that agent loops introduce new attack surfaces (prompt/tool injection, tool misuse, data exfiltration) that are not well captured by classic NLP benchmarks [18, 100, 33]. We therefore integrate evaluation and threat-model discussions throughout rather than treating them as a final afterthought.

Finally, adjacent work on retrieval augmentation and memory systems provides building blocks for agent state, but not all RAG systems are agents. We treat RAG as an agent component when it participates in a closed-loop policy (retrieval decisions affect downstream actions

and the environment responds). This distinction matters for interpreting empirical results and for deciding when benchmarks truly measure agentic behavior rather than retrieval quality [30, 51].

# 3 Foundations & Interfaces

This chapter fixes the boundary conditions of an LLM agent: what the loop looks like, what counts as an action, and how tools are represented and orchestrated. These interface choices are not "implementation details"; they constrain what evaluation claims are meaningful and what failure modes are even observable. A key theme is the trade-off between expressivity and control: richer action spaces and looser interfaces can unlock capability, whereas stricter contracts can improve verifiability and reduce costly failure cascades [91, 12].

We first examine the agent loop and action spaces (what the policy can do and what feedback it receives), then focus on tool interfaces and orchestration (how actions are grounded into executable APIs, how tools are routed, and how sandboxing/permissions change risk). These two pieces interact tightly: the loop defines *when* the agent can act, while the interface defines *how* those actions map to real-world side effects and evaluation protocols [101, 51].

## 3.1 Agent loop and action spaces

Agentic systems become analyzable once the control loop and action space are made explicit. A core tension is that richer action spaces (more tools, broader environment affordances, multi-step plans) can expand what an agent can do, yet the same richness makes behavior harder to constrain, debug, and evaluate under a stable protocol. In practice, agent-loop designs differ in how they represent state, how they decide when to act versus deliberate, and what constitutes a recoverable failure. The result is that "agent performance" is inseparable from loop semantics and evaluation setup, not only from the underlying model [91, 36].

One axis is how the loop interleaves reasoning with actions. Some systems treat reasoning traces as part of the policy interface, enabling tool-grounded intermediate steps and explicit observation updates [91]. Others emphasize externalized state and environment scaling, where the loop is designed to cope with broader action sets and longer horizons, often requiring stronger assumptions about logging and observability to support reproducible comparisons [69, 84]. In contrast to "single prompt" settings, these loop decisions directly affect what failure modes are likely (e.g., compounding errors, tool misuse, or brittle recovery) and what evaluation signals are available.

A second axis is what the action space *means* for evaluation. Benchmarks that expose many tools or realistic environments can stress test exploration and recovery, but they also introduce hidden degrees of freedom: success may hinge on tool availability, latency/cost budgets, or environment stochasticity. For example, benchmark suites and domain-specific settings shape what "success" operationalizes—task completion, correctness, cost efficiency, or robustness to noise—making protocol details central to synthesis [50, 99]. This motivates treating the agent loop and action representation as an interface contract rather than as incidental implementation.

Moreover, optimization and orchestration methods increasingly target *execution efficiency* as a first-class objective. Experiments on challenging agentic benchmarks such as GAIA and BrowseComp+ report that integrating EvoRoute into off-the-shelf agentic systems can sustain or enhance performance while reducing execution cost by up to 80% and latency by over 70% [101]. Such results illustrate a broader point: loop-level choices (when to replan, how to route actions, how to avoid redundant tool calls) can dominate system-level trade-offs even when the underlying model is fixed.

Across these studies, a useful comparison is between approaches that "scale the loop" versus those that "scale the environment." Environment-centric work expands the breadth of feasi-

ble actions and tasks, whereas loop-centric work constrains decision-making to keep behavior verifiable under a fixed protocol; in contrast, some domain agents prioritize specialized action primitives and evaluation metrics that are hard to compare against generic benchmarks [19, 16]. This suggests that future benchmarks should explicitly encode what the action space is and what assumptions are made about observability and tool reliability.

Evaluation protocols implicitly encode an action-space contract. When benchmarks expose an API/tool catalog with typed calls, success can be decomposed into tool selection, argument correctness, and recovery behavior; whereas free-form action spaces collapse these into a single end-to-end score that hides where errors occur. Benchmarks oriented around tool-use ecosystems and long-horizon interactions make these distinctions visible by tracking both outcome and operational signals (cost, latency, or tool-call counts), which is critical for debugging agent loops at scale [50, 84, 101]. Conversely, task suites that span heterogeneous domains can inflate apparent progress if protocol details (tool access, budgets, logging) are not aligned, leaving cross-paper comparisons unclear even when models look similar on headline metrics [99, 36].

Two concrete reporting practices would improve comparability for agent-loop papers. First, release action traces (tool calls, observations, retries) so that success can be attributed to specific loop decisions rather than to an opaque end score. Second, report calibration-style metrics that describe loop behavior, such as how often the agent chooses to act versus replan and how often recovery succeeds under tool or environment noise. Benchmark and methodology work begins to move in this direction by defining standardized harnesses and exemplar-driven evaluation, but the evidence is still limited until common logging and budget conventions are adopted broadly [36, 106, 86].

One practical consequence is that papers should expose the loop contract as an artifact: a state schema, an action schema, and an observation schema. When the action space is typed (tool calls with structured arguments), evaluation can attribute failures to selection versus argument filling versus recovery; when actions are free-form, the same end score hides where the loop breaks. Benchmarks that standardize the interface—by fixing tool catalogs and logging policies—make loop-level comparisons possible, whereas ad hoc environments often require reverse-engineering what actions were actually available [50, 84, 36].

Loop specification also interacts with the scale of the environment. ToolGym-style environments increase breadth by exposing diverse action primitives, whereas environment-scalers emphasize throughput and reproducibility by controlling observation streams and execution conditions [84, 69]. In this setting, efficiency-focused routers can change the apparent capability frontier: EvoRoute reports up to 80% cost and 70% latency reductions on GAIA and BrowseComp+ without sacrificing performance, and AgentSwift reports an average 8.34% gain over automated search baselines across seven benchmarks [101, 40]. These results underline that "better agents" may come from better loop budgeting and routing policies, not only from stronger base models.

A complementary line of work argues for exemplar-driven evaluation and reporting: provide canonical tasks and traceable exemplars so that loop semantics and recovery behaviors are comparable across implementations [86, 106]. This matters because agent loops embed hidden policies (when to ask for clarification, when to retry, when to stop), and those policies can dominate outcomes on suites that span heterogeneous domains [99, 91]. Viewed this way, the loop is the substrate that later components—tool interfaces, memory, and coordination—plug into; making it explicit is what lets system-level claims survive beyond a single environment.

Domain agents illustrate how action-space design reshapes what "reliability" means. Cybersleuth-style systems operate over investigative action primitives (search, triage, evidence linking), while ORFS-agent targets domain-specific optimization actions where the outcome metric is not task completion but engineering objectives such as routed wirelength [19, 21]. Even within nominally "tool-use" settings, ProAgent-like frameworks emphasize alignment between action abstractions and downstream evaluation: if the action space exposes high-level macros, apparent success may

come from the macro designer rather than from planning; whereas low-level action spaces stress exploration and recovery under tool noise [89, 16]. These contrasts explain why cross-domain benchmark suites are both valuable and risky: they encourage generality, yet they can mix incomparable action contracts unless each task specifies what observations and side effects are possible [99]. A practical synthesis is to treat the action space as part of the problem definition: report the tool catalog (or environment affordances), the admissible action formats, and the termination/recovery rules, then interpret performance only within that declared contract [106, 36].

From a survey perspective, this suggests a simple loop card for action-space papers: specify the action schema, the observation schema, and the budget and logging rules under which the loop runs. MCPAgentBench-style harnesses and exemplar-driven evaluation suggest that this metadata is often enough to explain why two systems with similar models diverge in performance—one may be operating under stricter tools, noisier observations, or tighter budgets [50, 86]. Reporting this card alongside benchmark results would make it easier to compare environment-scaled systems with loop-scaled systems and to separate algorithmic improvements from protocol engineering [106].

Additionally, making the loop contract explicit makes it easier to align protocol assumptions across benchmark suites; therefore, agent-loop claims should always be interpreted together with the action/observation contract and the budget model. This makes it easier to attribute failures to loop design rather than to environment idiosyncrasies.

As a result, a recurring limitation is that many papers under-specify protocol details that matter for interpretation: budgets (time/cost/token), tool permissioning, and logging granularity. When these constraints differ, cross-paper comparisons remain unclear, and improvements may reflect protocol engineering rather than more general agent competence. A practical takeaway is to treat action-space and loop definitions as part of the evaluation artifact: if the protocol cannot be reproduced (or the action space is ambiguous), strong claims should be taken as limited until verified under matched conditions [106, 21].

To keep the chapter's contrasts coherent, we next focus on function calling / tool schema / routing as the comparison lens.

### 3.2 Tool interfaces and orchestration

Tool interfaces determine how an agent's intentions become executable actions, and orchestration policies determine how those actions compose over a long horizon. The central tension is that expressive interfaces (many tools, flexible schemas, implicit routing) can increase capability, whereas stricter interfaces (typed arguments, explicit permissions, structured logging) can improve control and verifiability. Because tool calls create real side effects, interface contracts also act as a security boundary: they shape what an agent *can* do and what failures can be detected or prevented [12, 7].

A first comparison is between "free-form" tool use and schema-constrained tool use. Schema constraints reduce ambiguity in tool invocation and can support more reliable parsing and validation, but they also restrict the action space and can make generalization to new tools harder. Work on tool-use robustness and tool selection accuracy highlights that interface details—tool descriptions, argument formats, and error surfaces—often dominate downstream performance, especially when many tools overlap semantically [13, 47]. In contrast, smaller or simplified interface designs can reduce routing complexity at the cost of expressivity, which shifts the burden to prompting and post-hoc validation [65].

In addition, orchestration policies introduce additional degrees of freedom: how tools are selected, whether tool calls are retried, and how failures are handled. Evaluations that explicitly target end-to-end orchestration can make these decisions visible; for example, MSC-Bench targets multi-hop tool orchestration in an MCP ecosystem, emphasizing hierarchical tool contexts

rather than isolated calls [12]. Yet protocol details remain critical: if tool availability or budgets differ across runs, reported gains may be difficult to compare.

Interface design also shapes what "memory" means in tool use. Evaluating each MemTool mode across 13+ LLMs on the ScaleMCP benchmark reports experiments over 100 consecutive user interactions, measuring tool removal ratios (short-term memory efficiency) and task completion accuracy [51]. This illustrates that interface/orchestration choices can be evaluated not only by success rates, but also by operational metrics (cost, latency, tool usage) that matter for real deployments.

Across the literature, an emerging pattern is that reliability improvements often come from coupling interface constraints with training or confidence mechanisms. For instance, reported tool-selection accuracy gains and confidence-aware routing can improve robustness when tools are noisy or partially overlapping, whereas prompt-only routing policies can be brittle under distribution shift [47, 88]. In contrast, self-training or self-challenging approaches can improve tool use on multi-turn benchmarks, but the evidence can be limited when protocols and tool inventories vary across studies [110, 53].

A practical way to compare tool orchestration systems is to separate three layers: (i) the interface surface (schemas, error contracts, permissions), (ii) the routing policy (which tool, when, and with what arguments), and (iii) the runtime control policy (retries, fallback tools, and validation). Systems that formalize these layers can report intermediate metrics (tool selection accuracy, failure-recovery rates, or sandbox violation counts) that are otherwise hidden; in contrast, end-to-end task success alone can obscure whether improvements come from better reasoning, better routing heuristics, or more permissive tool access [47, 12]. Work that explicitly targets tool-use robustness and interface generalization highlights that routing errors often stem from ambiguous tool descriptions or overlapping affordances, whereas schema-constrained interfaces shift the difficulty toward correct argument filling and validation [13, 17, 65]. This decomposition also clarifies what ablations are needed: hold the tool catalog fixed while varying routing, or hold routing fixed while varying schema strictness.

Tool interfaces also define a security boundary. Permissioned tool surfaces and explicit runtime policies can prevent classes of misuse (e.g., disallowed parameters or unsafe side effects), whereas permissive interfaces shift the burden to downstream monitors and post-hoc auditing. Work that studies failures under realistic tool ecosystems highlights that observability (what is logged and validated) is as important as the schema itself, because many interface-level vulnerabilities are only detectable when tool I/O and decision rationales are retained [7, 24, 8, 41].

In practice, the tool catalog itself becomes a moving target. Tool descriptions can be ambiguous, incomplete, or inconsistent with tool behavior, creating specification gaps that force routing policies to learn undocumented conventions. Analyses of tool-use dissonances highlight that these gaps directly translate into tool selection and argument errors, especially when many tools overlap semantically [44, 47]. Systems that generate or curate tools automatically can widen coverage, but they also amplify the need for versioned schemas and validation because the catalog changes over time [32, 41].

Multi-turn tool use makes these interface choices measurable. Confidence-aware routers can trade off exploration and commitment by deferring uncertain calls, whereas self-challenging or self-repair loops attempt to detect and correct tool errors after the fact [88, 110]. However, these gains are hard to interpret if evaluations report only end success: interface health is better captured by intermediate metrics such as invalid-call rate, repair success, and tool-removal or tool-usage ratios in long interactions [51, 17].

Orchestration benchmarks start to encode these intermediate observables. MSC-Bench, for instance, stresses multi-hop orchestration in an MCP ecosystem and makes hierarchical tool contexts explicit, pushing evaluations away from "single-call accuracy" toward end-to-end tool pipelines [12]. At the same time, agent-guard style defenses emphasize that interface

contracts are also enforcement points: if tool I/O and routing decisions are not logged and checked, many failures become indistinguishable from adversarial manipulation or user error [7, 24, 8]. A useful synthesis is to treat interface design, routing policy, and runtime controls as co-designed layers; without that decomposition, cross-paper comparisons collapse into opaque "agent success" scores that hide where reliability is won or lost [53].

Another under-discussed variable is the human-facing layer. Many deployed agents must interpret ambiguous natural-language requests, choose among tools whose descriptions are partial, and decide when to ask clarifying questions. Avatar and personal-assistant settings suggest that interface contracts extend beyond schemas: the system must decide how to summarize tool outputs, what to expose to users, and what to log for auditing [83, 41]. This interacts with tool catalog evolution: AutoTool-style creation can add new capabilities quickly, whereas it also increases the need for schema versioning and regression tests because routing policies can silently change as tools are added or renamed [32, 47]. Tool-use dissonances further imply that evaluations should fix the tool catalog snapshot (descriptions and behavior) as part of the benchmark artifact; otherwise, improvements may reflect a cleaner tool description rather than a better agent policy [44, 12]. A pragmatic reporting move is to attach a compact "tool card" to results—schema, expected failure modes, and validation policy—so that interface comparisons remain interpretable as ecosystems evolve [53, 8].

Concretely, interface studies become more reusable when they isolate what is being held fixed. When a paper claims an orchestration improvement, readers need to know whether the tool catalog is fixed (same descriptions, same failure behaviors) or whether the catalog is being co-designed alongside the router. Benchmarks that ship the catalog and harness as part of the artifact support this kind of isolation, whereas ad hoc evaluations can conflate better routing with better tool curation [12, 44]. In practice, adding even a small ablation—same router with perturbed tool descriptions, or same catalog with a different schema strictness—can reveal whether robustness comes from the interface contract or from model-side heuristics [17, 47].

Finally, interface benchmarks should version their tool descriptions and failure behaviors, because routing policies can overfit to surface forms. Treating tool catalogs as datasets—with changelogs and regression tests—would make orchestration research more cumulative rather than anecdotal [32].

Additionally, treating the tool catalog as a versioned artifact reduces ambiguity; therefore, robustness claims should be paired with ablations that perturb tool descriptions and schema strictness. This suggests that a large fraction of orchestration failures are interface failures, not reasoning failures.

A key limitation is that the threat model is frequently implicit: papers may not specify what the agent is allowed to do, what inputs are adversarial, or how tool outputs can be manipulated. Without explicit sandboxing, permissioning, and observability assumptions, it remains unclear whether a proposed interface/orchestration policy is robust in real settings. As a result, interface comparisons should be read as protocol-scoped: strong claims should be treated as limited until reproduced under matched tool access and security constraints [24, 8].

# 4 Core Components (Planning + Memory)

Planning and memory are the two levers that most visibly change agent behavior over long horizons. Planning mechanisms decide *how* the system searches over action sequences (single-pass deliberation versus explicit search and verification), while memory mechanisms decide *what* information the policy can condition on (and how reliably that information reflects the environment) [91, 46].

We therefore split this chapter into planning/reasoning loops and memory/retrieval (RAG). The first subsection compares control-loop designs (planner/executor splits, search policies, verification) and how they trade accuracy for cost. The second contrasts memory types (ephemeral

scratchpads versus persistent stores) and retrieval/write policies that shape robustness, leakage, and reproducibility in agent evaluations [55, 51].

### 4.1 Planning and reasoning loops

Planning mechanisms decide how an agent selects action sequences under uncertainty, and reasoning-loop designs decide how deliberation interleaves with tool calls and observations. The practical tension is that deeper deliberation and explicit search can improve correctness on multi-step tasks, whereas it can also increase cost and latency and introduce new failure modes (e.g., overthinking, inconsistent intermediate states). Consequently, planning results are only interpretable when paired with evaluation constraints such as budget, tool access, and logging [91, 46]. The key point is that planning gains should be read as protocol-scoped budget trade-offs, a pattern emphasized in cost-aware evaluations [46].

One design axis is the control-loop decomposition: planner/executor splits versus monolithic policies. Planner/executor designs can isolate long-horizon reasoning from low-level tool invocation, enabling verification and recovery policies; in contrast, monolithic designs can be simpler but may entangle reasoning errors with tool misuse. Empirical studies of planning-oriented agents often highlight that the same base model can behave very differently depending on how plans are represented and validated [64, 9].

Moreover, another axis is search versus single-trajectory deliberation. Search-style methods explore multiple candidates or branches and can hedge against early commitment errors, whereas single-trajectory methods rely on strong priors encoded in prompts or learned policies. In practice, this trade-off is often a cost–reliability curve rather than a binary choice: more search can help, but only until budget constraints or tool costs dominate [54, 46].

Quantitative evidence underscores how planning choices interact with protocol constraints. For example, a state-of-the-art LLM penetration testing tool using self-guided reasoning reports completing only 13.5%, 16.5%, and 75.7% of subtasks and requiring 86.2%, 118.7%, and 205.9% more model queries across settings, illustrating that "stronger reasoning" can still be fragile and costly under realistic tool-use tasks [55]. This kind of result is most informative when the task type, success metric, and query/cost model are made explicit.

Across papers, a robust synthesis is that planning mechanisms should be evaluated with *both* success and cost metrics. Benchmarks that only measure end success can mask pathological behaviors (excessive tool calls, unsafe retries), whereas cost-aware benchmarks make explicit the decision-relevant trade-off between reliability and efficiency [46, 11]. In contrast, some planning results remain limited when evaluation protocols do not specify tool failure models or when intermediate states are not logged, making replication and comparison unclear.

Cost-aware evaluation reveals an additional contrast between planning as search and planning as policy. Search-heavy planners can improve success under uncertainty, but they also amplify the risk of budget blow-ups when tool calls are expensive or when verification is imperfect; policy-like planners reduce overhead, whereas they may fail catastrophically when an early mistake commits the trajectory. Benchmarks that model budget explicitly make this trade-off measurable by reporting success together with query/token counts and latency proxies, enabling comparisons between algorithms that would look identical under success-only reporting [46, 54]. In practice, many agent papers report qualitative improvements without enough protocol detail to tell whether gains come from search depth, better heuristics for branching/pruning, or simply more generous budgets; this leaves an open question about how planning methods generalize once budgets are normalized [64, 9].

Planning quality is often bottlenecked by verification. When tools are unreliable or observations are partial, planners that include explicit checks can avoid compounding errors, whereas unchecked deliberation can confidently pursue invalid trajectories. This makes ablations on verification and error-handling essential: a planner that appears stronger under idealized tools may fail under realistic noise models. Several planning-oriented evaluations therefore empha-

size protocol specification (failure modes, retries, budget) as part of the planning method itself, not a downstream implementation choice [55, 29, 36]. Reporting the full cost–success curve (rather than a single operating point) can expose when an approach is dominated across budgets, which is common once prompt and tool costs are significant. Without such curves, it remains unclear whether a method is practically useful or only competitive under a narrow, underreported budget setting [46].

Recent planning work increasingly treats the agent as an architecture problem: how to decompose deliberation, execution, and verification into components with explicit responsibilities. Architectural overviews and planning taxonomies distinguish between policy-like planners (single-pass, learned heuristics) and deliberative planners (iterative planning with validation hooks), and argue that the choice should follow the observability and budget constraints of the environment [59, 26]. In contrast, "planning" framed purely as a prompting pattern can look strong on small tasks yet degrade under long-horizon tool use when intermediate states are not validated or cached [28].

Reasoning-centric benchmarks further show that the failure mode is often not a lack of ideas but a mismatch between search depth and verification. Methods that explore branches can reduce early commitment errors, whereas they can also amplify inconsistency when branches are not grounded in stable intermediate representations [9, 108]. Novel planning objectives and learned planners can improve task planning under constrained compute, but their reported gains are most convincing when accompanied by ablations that separate planning quality from tool/interface changes [37, 29].

A practical reporting upgrade is to treat planning as a cost–success curve rather than a single operating point. CostBench-style evaluations and API-call focused benchmarks already motivate this view by measuring success alongside queries/tokens, tool-call counts, and latency proxies [46, 54]. Extending that idea to planning requires exposing plan representations (tree depth, branch factor, pruning rules) and failure recovery statistics, so that a reader can tell whether improved success comes from better search, better verification, or simply larger budgets. Without these details, even strong results remain protocol-scoped and hard to transfer to deployments where tool access and budget constraints differ [55, 28].

Domain-specific agents highlight another confound: planners are often evaluated in environments where domain knowledge is embedded in the tool layer. Medical or enterprise agents can appear to "reason" well because tools encapsulate workflows; in contrast, open-ended planning benchmarks force the planner to construct the workflow from primitive actions, which shifts the error distribution from knowledge gaps to search and verification mistakes [67, 64]. This difference matters because planning loops frequently rely on intermediate representations—task graphs, action templates, or structured plans—that are not shared across papers, making it hard to distinguish algorithmic improvements from representation engineering [28, 59]. Even when the same reasoning pattern is used, execution-time constraints can invert rankings: a method that is superior under generous budgets can be dominated under tight budgets once tool latency and verification overhead are accounted for [46, 54]. Therefore, planning results are most comparable when papers publish plan traces and branching statistics, tool-call logs with failure codes, and ablations that hold tool inventories fixed while varying planning depth and verification policies [29, 108, 80].

Because planning failures are often systematic (not random), reporting which subskills fail under budget is as important as reporting aggregate success. Reviews of planning-oriented agents emphasize recurring failure modes—hallucinated intermediate states, brittle recovery, and verification that does not match the task constraints—and argue that these should be surfaced in evaluation reports alongside the cost–success curve [26, 55]. CostBench-style reporting provides a natural place to attach this diagnosis: for each budget regime, report what fraction of failures are due to wrong tool calls, wrong plan structure, or premature termination [46].

Additionally, publishing plan traces enables failure analysis beyond single scores. This sug-

gests that planning progress will be easier to compare once benchmarks standardize the cost model and the verification hooks that planners rely on.

A limitation for the field is that planning improvements are often confounded with interface and memory choices: a "better planner" may simply have access to a different tool set or better retrieval. Without protocol alignment and ablations, it remains unclear which gains reflect planning algorithms versus environment engineering. For practitioners, a practical takeaway is to treat planning as part of a system contract: choose a planning loop that fits your budget and observability constraints, and validate claims under matched tool access and logging settings [29, 36].

The remaining uncertainty is retrieval / index / write policy, and resolving it makes the next trade-offs easier to interpret.

### 4.2 Memory and retrieval (RAG)

Memory mechanisms determine what information an agent can reliably condition on across steps, and retrieval mechanisms determine how that information is selected from a potentially large corpus. The central tension is persistence versus freshness: retaining more context can help long-horizon tasks, whereas persistent state can introduce staleness, contamination, and evaluation leakage that make results harder to reproduce. Because memory participates in the control loop, memory design is an agent-level decision, not merely a retrieval subsystem [30, 71].

A first comparison is between ephemeral scratchpads and persistent stores. Scratchpad-style memory is tightly coupled to a single episode and supports interpretability and debugging, whereas persistent memory can accumulate long-term traces that change agent behavior across episodes. In contrast, retrieval-centric designs emphasize selecting a small working set of relevant context at each step, which can improve efficiency but may be sensitive to indexing and query generation policies [98, 74].

Another comparison concerns the *source* of memory and the write policy. Some agents retrieve from static documents or curated knowledge bases; others retrieve from interaction logs, tool traces, or self-generated notes. These choices change what failure modes are likely: document retrieval can suffer from outdated or irrelevant content, whereas log-based memory can amplify early mistakes or introduce privacy and security concerns if writes are not constrained [66, 96]. In practice, evaluation protocols rarely standardize these memory sources, which makes cross-paper comparison fragile.

Quantitative results often bundle memory with broader agent design choices. For example, AgentSwift is evaluated across seven benchmarks spanning embodied, math, web, tool, and game domains and reports an average performance gain of 8.34% over existing automated agent search methods and manually designed agents [40]. Such numbers are informative only when the protocol makes clear what memory/retrieval resources are available and how they interact with planning and tool orchestration.

A useful synthesis is to treat memory as a reliability mechanism: the question is not only "does it help?", but "under what protocol assumptions does it remain trustworthy?" Retrieval from the web can improve coverage, whereas it can also increase the risk of prompt injection and unverifiable claims; persistent self-notes can stabilize behavior, whereas they can also lock in incorrect beliefs. Systems that explicitly measure memory efficiency and tool usage, rather than only end success, can make these trade-offs visible [51, 70].

Memory also interacts with planning in a way that many evaluations do not disentangle. When retrieval is treated as free context, planners can appear stronger simply because they see more relevant information; whereas when retrieval has an explicit cost (latency, token budget, or tool-call limits), the agent must trade recall against efficiency and robustness. This makes write policies and index choices decision-relevant: persistent memories can stabilize behavior across episodes, but they can also accumulate stale or incorrect traces that are hard to detect without explicit validation [71, 70]. Similarly, active retrieval and provenance-aware designs can

reduce hallucinated grounding, whereas they may increase complexity and create new failure modes when the retrieved context is adversarial or mismatched to the tool protocol [74, 111, 66].

Finally, memory is inseparable from safety and governance in deployed settings. Web retrieval and persistent notes can expose agents to retrieval injection and privacy risks, and these risks depend on what gets written, what gets retrieved, and how content is validated before action. A memory system that improves success in benign benchmarks may be risky under adversarial inputs unless the protocol includes sanitization, provenance, and permission checks. This reinforces the need to treat memory policies as part of the interface contract and to evaluate them under explicit threat models [70, 111, 103]. Even simple ablations (turning off long-term writes, freezing retrieval sources, or measuring sensitivity to stale documents) can clarify whether gains come from retrieval quality or from policy changes. These checks are still uncommon, which makes many reported memory improvements difficult to interpret across benchmarks [71]. This remains an open question for memory-centric agents evaluated under shifting web/tool distributions.

Agentic retrieval work emphasizes that retrieval is itself a policy: what to query, when to query, and how to validate what comes back. Active retrieval policies can improve usefulness under tight budgets, whereas they also introduce additional failure modes when the agent's queries are mis-specified or when retrieved context is adversarial [74, 87]. Task-oriented evaluations that stress long-horizon decision making make this explicit by measuring not just final success but the trajectory of retrieval and tool actions over time [93, 94].

For long-term memory, the key question becomes what is written and what is retrieved. Systems such as MemBox and MemTool motivate reporting memory-write events, retrieval frequency, and removal/retention behavior in addition to task success, because persistent stores can silently accumulate stale or sensitive content [70, 51]. This also clarifies contamination and leakage checks: without logging memory writes and retrieval sources, it is difficult to distinguish genuine agent improvement from evaluation artifacts—especially in web- or code-based settings where corpora evolve over time [30, 103].

Finally, memory design varies by domain. Code agents often benefit from structured summaries, dependency graphs, or repository-level indexes, whereas naive text dumps can overwhelm the context window and degrade tool-use reasoning. Meta-RAG style approaches and retrieval-centric code agents illustrate this spectrum: summarization and structured indexes can reduce token cost, but they can also mask provenance and make debugging harder when summaries drift from source [79, 38]. In these settings, provenance-aware retrieval and adversarially robust memory policies become essential, because incorrect or injected context can directly alter downstream edits and tool calls [111, 66, 96].

Memory also creates subtle evaluation leakage channels. When an agent writes persistent notes or caches tool outputs, later episodes may benefit from previously seen answers, inflating apparent generalization—especially on benchmarks with repeated templates or limited task diversity. Large-scale retrieval systems for agents therefore need contamination checks and provenance tracking: what document or trace supported a decision, and whether that support would be available in a clean evaluation setting [98, 30]. Long-horizon memory boxes highlight that even benign optimizations (keeping successful plans, storing user preferences) can become liabilities if stale entries are retrieved without validation, or if memory is updated by untrusted inputs [70, 111]. This is why memory evaluation should include drift and adversarial settings: freeze the corpus to test reproducibility, perturb documents to test robustness, and measure how often retrieval changes downstream tool actions [103, 74]. Operational metrics such as long-run completion and tool usage ratios can expose whether a memory system is accumulating shortcuts, whereas success-only scores can hide that the agent is simply caching answers [51, 96].

Similarly, memory contributions are most informative when they specify what part of the loop they improve: better retrieval queries, better indexing, or better write policies. Task-centric evaluations already encourage this decomposition by tracking long-horizon trajectories;

a memory system that helps early steps but harms later steps (via staleness) should be visible in per-step analyses rather than being averaged away [93, 94]. In agent settings, it is also useful to report memory as a resource: how many tokens are retrieved, how often retrieval is invoked, and how the agent trades retrieval against direct tool calls [51, 74].

A simple but underused diagnostic is to replay the same task under frozen memory: disable writes, or freeze retrieval sources, and measure how sensitive success is to cached context. This kind of ablation helps distinguish retrieval quality from policy changes and makes long-term memory claims easier to interpret [70, 30].

More generally, memory papers should report write/read provenance and publish replayable traces, because without them it is difficult to audit whether retrieval improves reasoning or merely caches answers [30].

This suggests that memory mechanisms should be evaluated under explicit drift and contamination tests, not only on static leaderboards.

As a result, memory results are often under-specified: papers may omit logging of memory writes, do not report contamination checks, or evaluate on benchmarks where retrieval artifacts are hard to detect. This makes it unclear whether improvements are due to better memory algorithms or to evaluation leakage. A pragmatic recommendation is to include memory-write transparency and protocol constraints (what can be written, when it is retrieved, and how it is validated) as part of the benchmark contract for agent evaluations [71, 103].

# 5 Learning, Adaptation & Coordination

Beyond fixed policies, modern agent systems increasingly adapt: they update prompts, revise plans, synthesize training data, or coordinate across multiple agents to improve outcomes. These adaptations can produce rapid gains, but they also raise stability and comparability challenges because the policy may change over the course of evaluation [110, 82].

We first review self-improvement and adaptation mechanisms (reflection, self-training, preference/RL signals, regression control), then multi-agent coordination patterns (roles, communication protocols, debate/refereeing, aggregation). The throughline is decision-relevance: which adaptation or coordination choice changes reliability/cost/safety under a fixed evaluation protocol, and which claims remain unclear without tighter logging and threat-model assumptions [10, 33].

## 5.1 Self-improvement and adaptation

Self-improvement mechanisms aim to make an agent better over time by revising prompts, synthesizing training data, or optimizing policies based on feedback. The central tension is that stronger training signals and more aggressive adaptation can yield rapid gains, whereas they also risk instability, regression, and reward hacking when the feedback channel is imperfect. Because adaptation changes the policy, meaningful evaluation depends on whether protocols control for what data, tools, and budgets the agent uses during improvement [73, 56].

One cluster of approaches uses iterative reflection and revision: the agent critiques its own outputs, proposes fixes, and re-executes under a loop that resembles debugging. This can improve outcomes without external labels, but it can also amplify bias if the critique model is miscalibrated. In contrast, self-training approaches synthesize trajectories or tool-use demonstrations and then fine-tune the model, which can improve tool proficiency but may overfit to benchmark idiosyncrasies or collapse diversity [82, 39].

Another cluster focuses on training-signal design: supervised fine-tuning versus preference optimization versus RL-style signals. While these signals can improve specific behaviors, comparisons remain unclear when protocols differ in tool access, budget, or evaluator strength. For

tool-use agents, training-signal choices interact with orchestration policies: an agent trained on one tool inventory may not generalize when tools are replaced or failure modes change [13, 61].

Quantitative evidence illustrates both promise and fragility. On two multi-turn tool-use benchmarks (M3ToolEval and TauBench), the Self-Challenging framework reports over a two-fold improvement in Llama-3.1-8B-Instruct despite using only self-generated training data [110]. Such results suggest that self-generated feedback can materially improve agent behavior, yet they also highlight the importance of protocol clarity (benchmark details, evaluator assumptions, and budget accounting) for interpreting "two-fold" gains.

Across studies, a synthesis is that adaptation should be evaluated as a controlled process with explicit regression checks. Techniques that optimize for a narrow reward can degrade robustness on out-of-distribution tasks, whereas conservative adaptation strategies can yield smaller gains but better stability. In contrast, routing and efficiency mechanisms that reduce wasted actions can provide "free" improvements under fixed budgets, even without changing the base model [101, 85].

Additionally, a core evaluation challenge for self-improvement is separating genuine generalization from overfitting to the evaluator. When the same model (or prompt) both generates training data and judges it, improvements can be brittle or misleading; in contrast, using held-out tasks, stronger external evaluators, or explicit regression checks can make gains more trustworthy but also more expensive. Recent work emphasizes that adaptation should be treated as a controlled loop with audit logs: what data was synthesized, under what tool access, and how the final policy was validated under the same protocol [56, 39]. This perspective suggests a practical reporting standard: publish the adaptation budget (queries/tokens), the acceptance criteria used for self-training, and the distribution of failures that remain, rather than only a final benchmark score [82, 110].

Self-improvement also raises governance questions: should an agent be allowed to change its own policy in production, under what budgets, and with what rollback guarantees? Conservative adaptation schemes can reduce regression risk by constraining update frequency or requiring external validation, whereas aggressive self-training can accumulate silent failures that only surface in deployment. As a result, safe self-improvement is best framed as a monitored control loop with explicit stopping criteria and auditability, rather than as a one-off training trick [106, 85, 91]. Because adaptation consumes additional queries/tokens, evaluation should normalize for improvement budget and report whether improvements persist under tighter constraints. Otherwise, a method may appear superior simply because it spends more compute during self-improvement, which is a different claim than improved sample efficiency [56]. This remains an open question for deployment-time learning loops where feedback is noisy and incentives are imperfect.

Beyond reflection, some approaches focus on grounding the improvement loop in external constraints: retrieving evidence, generating verifiable intermediate artifacts, or constraining updates to avoid compounding hallucinations. Grounded self-training and feedback design work argues that the improvement signal should be tied to checkable traces (tool calls, retrieved evidence) rather than to free-form self-judgment, because otherwise the loop can reinforce fluent but incorrect behaviors [5, 27]. In contrast, broad "agent evolution" frameworks that search over prompts or policies can discover effective behaviors, but they risk producing brittle solutions if the search is optimized against a narrow benchmark distribution [82, 2].

This makes evaluation design central. Agent benchmarks that include robustness and failure-mode coverage encourage reporting not only average gains but also regression profiles: which tasks got worse, under what budgets, and whether improvements persist once tools or contexts change [102, 106]. For self-training methods that report large relative improvements, the most informative additional evidence is a controlled ablation of the loop: hold tool inventory and budgets fixed, separate data-generation from evaluation, and report how performance scales with additional self-improvement compute [110, 56].

Finally, adaptation interacts with coordination and governance. An agent that changes its own policy can create inconsistency across roles in a multi-agent system or across time in a long-running deployment. Coordination-oriented protocols and rollback mechanisms can mitigate this by requiring consensus or external approval before updates, whereas unconstrained self-updates can accumulate silent failures that are difficult to detect until they surface as incidents [109, 85]. Ethical and contextual considerations also matter: what counts as "improvement" may differ across user groups and settings, so evaluation should make the target preference model explicit rather than assuming a universal objective [72, 61].

Self-improvement loops can be described as a three-stage control system: propose an update, evaluate it under a gate, and either accept, rollback, or continue searching. Agent-evolution style methods make this explicit by searching over prompts or policies, whereas autonomous self-training pipelines optimize internal models and can introduce distribution shift between the data generator and the final deployed agent [2, 39]. Because the loop itself consumes compute, the meaningful comparison is not only final score but the slope of improvement per additional budget and the stability of the result under regressions [56, 82]. Several systems therefore emphasize guarded adaptation: maintain audit logs, cap self-rewrites, and require external validation before an update is committed [85, 106]. In contrast, unguarded loops may overfit to the evaluator or exploit scoring loopholes, producing gains that disappear once tool access or benchmarks change [13, 110]. A practical takeaway is to treat adaptation as part of the deployment contract—specify the feedback channel, acceptance criteria, and rollback policy—so that learning claims remain interpretable in safety-critical settings [72, 61].

A final practical implication is that self-improvement should be evaluated like a continuous deployment process. Faultline-style analyses highlight that regressions and reward hacking are common when the feedback channel is imperfect; therefore, stable self-improvement requires acceptance tests, rollback criteria, and explicit budgets for how much the agent can change itself between evaluations [56, 85]. Treating the loop as a controlled system, rather than as a one-shot training trick, also clarifies which claims are about sample efficiency versus total compute, which is crucial for deployment decisions [106].

From a governance angle, these controls also enable accountability: if an update causes harm, an audit log plus rollback policy provides a concrete remediation path. Without such mechanisms, learning in production risks turning evaluation variance into operational incidents [85].

Additionally, guarded adaptation policies make self-improvement safer to deploy. This implies that future benchmarks should report not only gains, but also rollback and regression behavior under matched budgets.

A key limitation is that many adaptation papers do not fully specify the data-generation and evaluation loop: what trajectories are collected, how feedback is computed, and whether the final policy is evaluated on held-out tasks with matched tool access. Without such details, it remains unclear which gains reflect genuine generalization versus protocol coupling. For practitioners, a practical takeaway is to treat self-improvement as a deployment-time control problem: log the adaptation process, enforce budgets, and validate improvements under the same threat model and tool constraints expected in production [106, 91].

The remaining uncertainty is roles / communication / debate, and resolving it makes the next trade-offs easier to interpret.

### 5.2 Multi-agent coordination

Multi-agent systems replace a single policy with an ensemble of interacting agents that divide labor, debate, or verify each other. The motivating tension is that richer coordination protocols can increase robustness and task coverage, whereas they also introduce new failure modes: communication overhead, collusion, inconsistent goals, and protocol gaming. The key point is that coordination trades additional compute for verification and can create correlated failures

if diversity assumptions break. As with single-agent loops, coordination results are only meaningful when protocols specify roles, message budgets, and what counts as a verified outcome [10, 80].

A common comparison is role specialization versus symmetric swarms. Role-specialized systems assign explicit responsibilities (planner, critic, executor, verifier), which can improve debuggability and enable targeted evaluation; in contrast, symmetric swarms rely on redundancy and aggregation (voting or consensus) to reduce individual agent errors. These two designs imply different cost structures and different security surfaces, especially when agents can call tools or write persistent state [97, 81].

Debate and referee-style protocols aim to elicit better reasoning by creating structured disagreement and verification. Such methods can improve reliability in settings where a single agent is brittle, whereas they can also fail when agents share the same blind spots or when incentives encourage persuasive but incorrect arguments. In contrast, hierarchical agent designs emphasize decomposition and coordination across sub-tasks, trading off communication complexity against clearer interfaces and evaluation hooks [10, 109].

Quantitative examples highlight the scale and diversity of coordination settings. Across multiple design iterations, one system reports developing 13,000 LLM agents to simulate crowd movement and communication during large-scale gatherings under various emergency scenarios [43]. While such results demonstrate the breadth of multi-agent applications, they also underscore that evaluation protocols (simulation fidelity, safety constraints, and reproducibility) must be specified to interpret outcomes across domains.

A cross-paper synthesis is that coordination mechanisms often act as *verification layers*: they trade additional compute for reduced error rates by adding redundancy, critique, or structured aggregation. In contrast, coordination can also amplify risk when agents share tools or memory: an injected or compromised agent can steer the group, and message-level defenses become part of the threat model. Systems that explicitly measure coordination cost (messages, tool calls) alongside success make these trade-offs visible [4, 11].

Evaluation of coordination should be multi-dimensional. Success-rate improvements are meaningful only when paired with coordination costs (messages, rounds, tool calls) and with a clear protocol for aggregation and verification. Debate-style protocols can reduce errors by forcing explicit counterarguments, whereas aggregation-based protocols rely on diversity and independence assumptions that may not hold when agents share the same base model or retrieval sources [10, 80]. Federated or distributed variants introduce additional constraints (communication topology, privacy, partial observability) that change what coordination means and what failure modes dominate, so comparisons should state these protocol choices explicitly [81, 109, 4].

Coordination introduces additional failure modes beyond individual-agent errors. Group protocols can suffer from correlated mistakes (shared blind spots), persuasive-but-wrong arguments in debate, or collusion when incentives are misaligned; in contrast, redundancy can reduce variance when agent errors are closer to independent. These issues are hard to diagnose when papers report only a single aggregate score, so multi-agent evaluations should report per-role error patterns and sensitivity to message budgets and aggregation rules [11, 10, 65]. Moreover, many multi-agent systems implicitly assume a shared tool environment; when tool access differs by role, coordination strategies can fail in ways that are invisible on aggregate metrics. Explicitly logging message graphs and tool-call traces, and reporting per-role failure modes, would make multi-agent results more comparable across settings and would help diagnose correlated errors and collusion risks [81, 4]. These costs matter when coordination is itself a limited resource and must be budgeted like tool calls. When budgets are fixed, adding agents often trades raw success for higher reliability only if diversity is real and aggregation rules are well specified.

Recent surveys of multi-agent LLM systems emphasize that coordination spans several orthogonal choices: communication topology (who talks to whom), message formats (free-form

versus structured), and role assignment (fixed experts versus dynamic). These choices affect both performance and auditability: structured protocols can make verification easier, whereas free-form chats can hide disagreement and make failure diagnosis hard [60, 25]. Tooling for multi-agent pipelines also starts to standardize orchestration primitives (assignment, aggregation, escalation), which helps make coordination artifacts comparable across implementations [57, 42].

Coordination benchmarks are beginning to formalize what authentic collaboration looks like. DEBATE explicitly evaluates the authenticity of interaction in multi-agent settings rather than treating messages as uninterpreted tokens, and evaluation work on physically grounded collaboration highlights that shared environments can induce new coordination failure modes beyond text-only debate [10, 95, 11]. In contrast, purely outcome-based tasks can overstate coordination gains when success can be achieved by a single dominant agent, making it important to report per-role contributions and sensitivity to communication budgets.

Learning to coordinate also raises alignment questions: aggregation can suppress minority but correct views, and specialized roles can be exploited if one agent can manipulate shared state. Alignment-oriented studies of multi-agent behavior and evolutionary approaches therefore recommend stress-testing protocols under adversarial roles and under evolving populations, rather than assuming benevolent cooperation [49, 78, 14]. Practical reinforcement learning approaches for coordinating agents suggest that reward design and budget constraints can materially change emergent behaviors, so comparisons should state the optimization objective and compute budget as part of the protocol [4, 81].

Multi-agent coordination is sometimes motivated as a way to scale breadth—cover more subtasks in parallel—but its main benefit is often error shaping: debate and aggregation turn unstructured uncertainty into explicit disagreement that can be audited. However, this only holds when interaction protocols prevent degenerate agreement (agents echoing each other) and when the budget for communication is treated as a first-class resource, similar to tool-call budgets in single-agent systems [10, 4]. Systems that emphasize exploration in open-ended environments also blur the line between single- and multi-agent behavior: a single system may instantiate many ephemeral roles over time, so coordination overhead shows up as latency and cost even if the base model is shared [75, 57]. Empirical studies of agent populations and evolving protocols suggest that more agents is not monotonic: coordination can fail under tight budgets, and groups can become more brittle if all agents share the same retrieval source or policy initialization [78, 80]. Therefore, evaluations should report scaling with number of agents and message budget, and include stress tests where one role is adversarial or misaligned to measure protocol robustness [49, 25, 81].

For survey synthesis, the biggest obstacle is that coordination protocols are rarely standardized. A small change in message format, role prompts, or aggregation rule can change outcomes as much as a model change, yet these details are often omitted. Survey and tooling work therefore motivates treating the protocol itself as a benchmark artifact: publish role specs, message budget, aggregation rules, and logging schema so that other groups can reproduce coordination behavior under the same constraints [60, 57]. Without this, multi-agent results remain difficult to compare even when they are compelling within a single implementation [80].

Even lightweight protocol cards—roles, message budget, aggregation rule—would materially improve comparability across studies [60].

Additionally, coordination protocols should be stress-tested under constrained message budgets. This suggests that many reported multi-agent gains may be fragile unless diversity and aggregation assumptions are measured explicitly.

Therefore, many multi-agent evaluations lack standardized baselines: differences in role definitions, communication channels, and budget constraints make head-to-head comparison unclear. For practitioners, a pragmatic approach is to treat the coordination protocol as a first-class artifact: specify roles, message formats, and verification rules, then evaluate under

matched budgets and explicit threat models to avoid overstating gains that depend on hidden protocol choices [65, 75].

# 6 Evaluation & Risks

Evaluation is the bottleneck for evidence-backed progress in agent research: results are rarely comparable unless protocols align on tool access, budgets, observability/logging, and the definition of success. At the same time, deployed agents face new security and governance constraints that directly reshape both interfaces and evaluation objectives [53, 100].

We first survey benchmarks and evaluation protocols, emphasizing what they measure and what they leave ambiguous (leakage, reproducibility, cost accounting, threat models). We then synthesize safety, security, and governance work, treating threat models and defense surfaces (sandboxing, monitoring, policy enforcement) as first-class design axes rather than "post-hoc guardrails" [18, 33].

## 6.1 Benchmarks and evaluation protocols

Agent evaluation asks a deceptively simple question—did the system solve the task?—but in agentic settings the answer depends on protocol details: tool access, budgets, observability/logging, and what constitutes a valid action sequence. A key tension is realism versus comparability: more realistic environments (noisy tools, partial observability) can expose failure modes, whereas they also introduce confounds that make cross-paper synthesis difficult. This motivates evaluation taxonomies that explicitly separate objectives (capability, reliability, safety) from protocol instantiations [53, 58].

Benchmarks differ in what they operationalize. Some suites emphasize end-to-end task completion, while others stress robustness to attacks, tool misuse, or protocol deviations. As a result, "success rate" is not a universal metric: it must be interpreted alongside cost (token/query/tool usage), latency, and failure recovery behavior. Protocol-aware benchmarks and reviews therefore recommend reporting both outcome and resource usage, and making tool inventories and budgets explicit [35, 6].

Moreover, security and robustness benchmarks make the protocol contract concrete by specifying adversarial objectives and tool-level constraints. For example, RAS-Eval comprises 80 test cases and 3,802 attack tasks mapped to 11 Common Weakness Enumeration (CWE) categories, with tools implemented in JSON, LangGraph, and Model Context Protocol (MCP) formats [18]. Such designs highlight that evaluation artifacts can (and should) encode interface formats and threat models, not only task descriptions.

Reliability work further emphasizes that evaluation should measure *behavior under monitoring* and *distribution shift*. Monitor red-teaming workflows vary agent and monitor situational awareness and model adversarial strategies (e.g., prompt injection) to test whether monitoring is effective under realistic assumptions [33]. In contrast, benchmarks for context-aware proactive agents focus on whether the agent can act appropriately given evolving context and tool availability, which requires careful specification of what context is observable and when [90, 53].

From a measurement standpoint, three protocol dimensions repeatedly dominate variance: (i) tool access and reliability (idealized APIs vs noisy tools), (ii) resource budgets (tokens, queries, latency/cost), and (iii) observability/logging (whether intermediate tool I/O is retained and audited). Benchmarks that control these factors can attribute improvements to agent policies more credibly; whereas benchmarks that leave them implicit can conflate policy changes with protocol engineering [53, 35]. This also explains why leaderboards alone are insufficient for survey synthesis: without protocol metadata, the same system can occupy different design points depending on deployment assumptions.

Reproducibility and leakage are additional evaluation axes that are especially acute for web- or tool-based tasks. When agents interact with changing external resources, results can drift over time, and hidden leakage channels (cached answers, retriever contamination, tool output changes) can inflate success. Dataset- and benchmark-building work therefore increasingly treats environment snapshots, tool mocks, and evaluation harnesses as part of the artifact, aiming to make agent evaluation closer to systems benchmarking rather than one-off demonstrations [99, 18]. In contrast, broad surveys note that many published agent results remain difficult to compare precisely because these artifacts are missing or not standardized [73, 58].

A practical implication for survey writing is to treat evaluation artifacts as comparable objects. When a paper reports only a benchmark name without describing tool access or budgets, the result is often unclear: the same benchmark can yield different outcomes depending on tool catalogs, prompt policies, and cost models. Evaluation surveys therefore recommend minimum reporting standards (task suite, metric definition, budget, tool access, logging) and encourage publishing harness code so that comparisons become reproducible rather than rhetorical [53, 90, 35]. Standardizing these reporting fields would also support better meta-analysis: surveys could group results by tool realism or budget class and avoid mixing incomparable settings. In the near term, even a lightweight protocol card accompanying each benchmark would make it easier to align evaluation objectives with deployment needs [35]. This is especially important for interactive web and tool benchmarks whose environments evolve over time.

Several works argue that agent evaluation needs a shared vocabulary before it needs more leaderboards. Taxonomies of agent objectives and failure modes propose organizing benchmarks by the capability being measured (planning, tool use, robustness, safety) and by the protocol knobs (tool realism, budgets, observability), so that synthesis can compare like with like [31, 76]. This complements broader agent surveys that catalog systems but may not normalize protocols, and it motivates reporting templates that record the key protocol fields alongside results [58, 53].

Budget normalization is particularly important for agents because the dominant cost is often interaction, not inference alone. Evaluations increasingly track token/query counts, tool latency, and explicit cost models, and show that the same planner can move along a Pareto frontier depending on budget settings and tool availability [35, 48]. In contrast, benchmarks that ignore budgets can reward brittle strategies that spam tools or rely on expensive verification, which is unlikely to transfer to constrained deployments.

Benchmark construction is becoming a research contribution in its own right. BuildBench-style efforts treat benchmark design as engineering: define realistic tool stacks, specify failure models, and ship harness code so that results are reproducible over time [104, 22]. For web and data-science tasks, snapshotting environments and auditing leakage channels are necessary to prevent silent drift; suites that standardize these artifacts make progress easier to interpret across years [99, 6]. Security benchmarks can also be read as evaluation design work, because they operationalize adversarial objectives and tool constraints in a way that generic leaderboards do not [100, 105].

Even within a single benchmark name, protocol variants can dominate. An agent evaluated under a cached environment, generous budgets, and permissive tool access is solving a different problem than the same agent under strict budgets, noisy tools, and full logging requirements. Evaluation papers increasingly recommend publishing protocol descriptors (tool catalog snapshot, budget model, logging schema, failure model) alongside leaderboard scores so that surveys can normalize across these axes [35, 31]. This aligns with benchmark-as-system efforts: ship the harness, not just the dataset, so reproduction is possible across time and across tool ecosystems [104, 6]. In practice, these protocol descriptors are also what enable safety scores to be comparable: without an explicit interface and threat model, a security result cannot be interpreted across agent frameworks [100, 76].

One implication is that evaluation-first work should provide not only tasks but also protocol defaults. When a benchmark release defines a reference tool catalog snapshot, a budget model,

and a logging schema, later papers can report results without re-implementing the environment, and surveys can aggregate evidence without guessing hidden assumptions [104, 35]. As benchmarks become more systems-like, harness-level decisions (timeouts, retries, caching) become part of the scientific claim, which is why papers that treat the harness as the deliverable are increasingly valuable for reproducible agent research [6].

Two practical reporting artifacts would make benchmarks easier to compare in surveys. First, a protocol card describing tool access, budget model, logging, and environment snapshot, so that results can be grouped by deployment realism rather than by benchmark name alone [31, 35]. Second, an error taxonomy and audit log template—what failures were tool-selection errors, argument errors, retrieval errors, or monitor failures—so that improvements can be attributed to specific system components [53, 33]. Without these artifacts, benchmark-driven progress will continue to mix incomparable settings and invite over-interpretation of headline metrics [76].

In practice, even a small set of standardized protocol fields would let meta-analyses separate tool realism from planner quality and reduce over-interpretation of headline metrics [53].

Additionally, treating protocol descriptors as first-class benchmark outputs enables survey-level aggregation across benchmarks without guessing hidden assumptions.

A recurring limitation is that many published results remain hard to reproduce: papers may omit tool versions, budgets, or logging, and may not clearly distinguish between "policy improvements" and "protocol engineering." This makes it unclear which gains will transfer to new tool ecosystems or deployment settings. A practical recommendation is to treat protocol files (tool schema, permission policy, budget, logging spec) as part of the benchmark release, enabling direct comparisons and reducing accidental drift across evaluations [35, 102].

With function calling / tool schema / routing as context, threat model / prompt/tool injection / monitoring becomes the next handle for comparing approaches under shared constraints.

### 6.2 Safety, security, and governance

Agent systems introduce new security and governance problems because they can take actions with real side effects. The core tension is autonomy versus control: broader tool access and longer horizons can unlock capability, whereas they also increase the blast radius of mistakes and adversarial manipulation. Threat models therefore need to include not only prompt injection, but also tool injection, data exfiltration, permission escalation, and monitoring evasion [100, 77].

A first comparison is between policy-only defenses and system-level defenses. Policy-only defenses attempt to constrain model behavior via prompts or learned refusal behavior, whereas system defenses constrain the action space via sandboxing, typed interfaces, and permission checks. System-level controls can reduce worst-case risk, but they also shift the evaluation target: success depends on whether the agent can still accomplish tasks under constrained tool access and logging requirements [34, 45].

Empirical security evaluations demonstrate how severe protocol-level vulnerabilities can be. One security taxonomy work reports evaluating nine popular LLM agents across 10 domains and 400+ tools, producing 2,000 attack instances [100]. Such studies highlight that agent safety is not only about the base model; it is about the full stack: interfaces, tool descriptions, runtime permissions, and monitoring assumptions.

In addition, several lines of work focus on monitoring and red-teaming as operational governance mechanisms. Monitor red-teaming frameworks vary what the monitor knows and how attackers behave to test whether monitoring can detect evasive strategies; in contrast, quitting/degenerate behaviors under adversarial pressure can reveal whether agents fail safely or fail catastrophically [33, 3]. Evaluations that report both attack success and task completion under matched budgets make it easier to compare defenses without overstating robustness [18].

Enforcement mechanisms can be grouped by where control is applied: before action (permissioning and interface typing), during action (sandboxing and runtime monitors), and after

action (auditing, rollback, and incident response). Pre-action controls can prevent entire classes of tool misuse, whereas they may reduce capability if the action space becomes too constrained; runtime monitoring can preserve capability but depends on what the monitor can observe and how quickly it can intervene [34, 33]. Post-hoc auditing is essential for governance but is limited when logging is incomplete or when tool side effects are irreversible, which motivates designing interfaces with observability as a first-class goal [45].

Another recurring safety failure mode is degenerate autonomy, where agents quit early, loop on unproductive actions, or follow adversarial instructions that appear helpful. Evaluations that explicitly measure quitting behavior and monitor evasion make these modes visible and support more actionable mitigations; in contrast, success-only reporting can hide near-misses or unsafe trajectories that should matter for deployment decisions [3, 18]. Finally, governance-oriented benchmarks that stress-test impossible or policy-violating requests can help calibrate refusal and escalation policies, but the evidence remains limited until threat models and permissions are specified consistently across studies [107, 15].

Open questions remain about how to balance capability with enforceability. Hard constraints (strict permissioning) can prevent severe failures but may reduce usefulness, whereas softer constraints (monitoring and escalation) preserve capability but depend on detection quality and response latency. This suggests evaluating defenses under realistic adversaries and deployment constraints, including impossible requests and policy-violating tool calls, and reporting both task completion and safety outcomes under matched budgets [107, 34, 33]. Another open issue is responsibility assignment: when an agent violates policy via a tool call, it is unclear whether blame lies in the model, the router, or the permission policy. Auditable logs and clear escalation pathways can reduce this ambiguity, but they require standardization of what is logged and how monitors are evaluated in the first place [33, 34]. In practice, this requires joint design of policy constraints, system controls, and evaluation protocols. Until such artifacts are standardized, safety and governance claims will remain limited and hard to compare across tool ecosystems.

A practical threat model for agents should treat every interface surface as an injection channel: user prompts, retrieved content, and tool metadata. Work that bridges tool-use and security emphasizes that attacks can live in tool descriptions, schema fields, or tool outputs, and that defenses must assume the agent will faithfully execute what it believes is a valid tool call unless constrained by interface contracts and monitors [20, 100]. Input sanitization and isolation mechanisms are therefore not optional add-ons but core parts of the agent stack, especially when retrieval and tool catalogs are dynamic [1, 105].

Several benchmarks also show that governance controls need to be evaluated under realistic interaction formats. RAS-Eval implements tools in multiple formats (JSON, LangGraph, MCP), making it possible to test whether a defense generalizes across interface standards rather than overfitting to a single tool wrapper [18, 22]. Similarly, enhancements to monitoring and tool-use safety often hinge on what signals the monitor can see (full tool I/O versus summaries), which is why monitor design is inseparable from logging policy and interface structure [33, 23].

From a governance perspective, the relevant question is often not whether the model is safe, but what the escalation path is when the system is uncertain or under attack. Studies of refusal and policy compliance suggest that agents should be evaluated on impossible or policy-violating requests with explicit escalation policies, and that these evaluations should report both safety outcomes and the cost of intervention (delays, false positives) [107, 15]. Systems that integrate runtime controls, audit logs, and rollback procedures can reduce operational risk, but they also create new points of failure if the controller is mis-specified or too expensive to run at scale [52, 76, 108].

Security also interacts with reasoning and planning: a system that is robust to prompt injection in isolation may still be vulnerable once the planner can be induced to call a dangerous tool, or once an attacker can manipulate intermediate state that the planner treats as trusted. Reasoning failures under adversarial constraints—overconfident tool calls, refusal bypass via de-

composition, or exploitation of ambiguous policies—suggest that governance must be evaluated end-to-end, including how the planner selects actions under uncertainty and how monitors interpret partial evidence [108, 77]. Enforcement mechanisms that operate at the interface (typed schemas, permission checks) can reduce some classes of attacks, whereas they can also create incentives for agents to find loopholes in the allowed action space unless monitors and audits are aligned with the same policy [34, 45]. Practical tool-use safety work argues for layered defenses: isolate untrusted inputs, constrain tool capabilities, and log enough context for incident response, because no single mitigation is sufficient across threat models [63, 52]. This reinforces a broader governance principle: what matters is not only preventing bad actions, but making uncertainty visible and recoverable through escalation paths, safe failure modes, and post-hoc accountability artifacts [15, 33].

Finally, safety evaluation benefits from the same protocol discipline as capability evaluation: specify tools, budgets, and observability. Security benchmarks emphasize that many attacks succeed because the agent is allowed to call unsafe tools or because logs are insufficient for monitors to detect manipulation; monitoring and enforcement work likewise shows that defense effectiveness depends on what signals are visible and how quickly interventions can occur [100, 33]. As a result, comparisons between defenses are only meaningful when evaluated under matched tool access and threat models, as exemplified by end-to-end suites that include both task completion and attack outcomes [18].

Taken together, this suggests that governance should be evaluated as an operational pipeline: define the policy, enforce it at the interface, monitor at runtime, and audit afterwards. A defense that looks strong under one tool ecosystem may fail under another unless these layers are explicitly specified and tested [100].

A key limitation is that governance claims are often underspecified: papers may not state what permissions the agent had, what logs were retained, or what constitutes a policy violation. Without these details, it remains unclear whether a defense would work in a different tool ecosystem. For practitioners, the pragmatic approach is to treat governance as a contract: define the threat model, specify the defense surface (sandbox, monitoring, policy enforcement), and evaluate under realistic tool access and adversarial conditions before deployment [107, 15].

# 7 Discussion

A consistent lesson across the design space is that "agent quality" is inseparable from interfaces and protocols. Loop semantics, tool schemas, routing policies, and logging assumptions jointly determine what failures are observable and what comparisons are legitimate. This suggests that future agent papers should treat protocol artifacts (tool inventory, budget model, permission policy, logging spec) as first-class contributions, not as appendix details [53, 12].

A second lesson is that cost and reliability are coupled. Many improvements that look like "better reasoning" are actually better budget allocation: fewer redundant tool calls, better routing, or more selective planning/search. Conversely, stronger planning can still be brittle and expensive under realistic tool-use tasks, so evaluation should routinely report both outcome and resource usage rather than single success numbers [46, 55].

Security and governance are not optional add-ons for deployed agents. Tool access creates a broad attack surface, and defenses often require system-level controls (sandboxing, monitoring) that change the feasible action space and thus the evaluation objective. The field would benefit from shared threat-model templates and benchmark releases that encode realistic tool behaviors and adversarial strategies [100, 33, 18].

Finally, the current evidence base remains heterogeneous. Even within the same benchmark family, protocol details (tool versions, budgets, logging) vary enough to make synthesis fragile. The most actionable research direction is therefore methodological: build benchmarks and reporting standards that make protocols reproducible and enable ablations that separate policy

changes from protocol engineering.

## 8 Conclusion

LLM agents should be understood as closed-loop systems whose behavior is shaped by action spaces, tool interfaces, planning and memory mechanisms, adaptation strategies, and evaluation protocols. Across the literature, interface contracts and protocol choices repeatedly determine whether empirical claims are comparable and whether improvements transfer beyond a single benchmark setup.

For practitioners, the key takeaway is to treat interfaces, budgets, permissions, and logging as part of the agent design, not as implementation plumbing. For researchers, the key open problems lie in evaluation: protocol standardization, cost-aware and safety-aware metrics, and benchmark designs that make threat models and tool behaviors explicit.

## References


[1] Hengyu An, Jinghuai Zhang, Tianyu Du, Chunyi Zhou, Qingming Li, Tao Lin, and Shouling Ji. Ipiguard: A novel tool dependency graph-based defense against indirect prompt injection in llm agents. *arXiv preprint arXiv:2508.15310v1*, 2025. URL http://arxiv.org/abs/2508.15310v1.

[2] Nikolas Belle, Dakota Barnes, Alfonso Amayuelas, Ivan Bercovich, Xin Eric Wang, and William Wang. Agents of change: Self-evolving llm agents for strategic planning. *arXiv preprint arXiv:2506.04651v2*, 2025. URL http://arxiv.org/abs/2506.04651v2.

[3] Vamshi Krishna Bonagiri, Ponnurangam Kumaragurum, Khanh Nguyen, and Benjamin Plaut. Check yourself before you wreck yourself: Selectively quitting improves llm agent safety. *arXiv preprint arXiv:2510.16492v2*, 2025. URL http://arxiv.org/abs/2510.16492v2.

[4] Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R. Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-agent: Efficient rl training for multi-turn llm agent. *arXiv preprint arXiv:2511.16108v1*, 2025. URL http://arxiv.org/abs/2511.16108v1.

[5] Arthur Chen, Zuxin Liu, Jianguo Zhang, Akshara Prabhakar, Zhiwei Liu, Shelby Heinecke, Silvio Savarese, Victor Zhong, and Caiming Xiong. Grounded test-time adaptation for llm agents. *arXiv preprint arXiv:2511.04847v3*, 2025. URL http://arxiv.org/abs/2511.04847v3.

[6] Chaoran Chen, Bingsheng Yao, Ruishi Zou, Wenyue Hua, Weimin Lyu, Yanfang Ye, Toby Jia-Jun Li, and Dakuo Wang. Towards a design guideline for rpa evaluation: A survey of large language model-based role-playing agents. *arXiv preprint arXiv:2502.13012v3*, 2025. URL http://arxiv.org/abs/2502.13012v3.

[7] Jizhou Chen and Samuel Lee Cong. Agentguard: Repurposing agentic orchestrator for safety evaluation of tool orchestration. *arXiv preprint arXiv:2502.09809v1*, 2025. URL http://arxiv.org/abs/2502.09809v1.

[8] Yize Cheng, Arshia Soltani Moakhar, Chenrui Fan, Parsa Hosseini, Kazem Faghih, Zahra Sodagar, Wenxiao Wang, and Soheil Feizi. Your llm agents are temporally blind: The misalignment between tool use decisions and human time perception. *arXiv preprint arXiv:2510.23853v2*, 2025. URL http://arxiv.org/abs/2510.23853v2.

[9] Jae-Woo Choi, Hyungmin Kim, Hyobin Ong, Minsu Jang, Dohyung Kim, Jaehong Kim, and Youngwoo Yoon. Reactree: Hierarchical llm agent trees with control flow for long-horizon task planning. *arXiv preprint arXiv:2511.02424v1*, 2025. URL http://arxiv.org/abs/2511.02424v1.

[10] Yun-Shiuan Chuang, Ruixuan Tu, Chengtao Dai, Smit Vasani, Binwei Yao, Michael Henry Tessler, Sijia Yang, Dhavan Shah, Robert Hawkins, Junjie Hu, and Timothy T. Rogers. Debate: A large-scale benchmark for role-playing llm agents in multi-agent, long-form debates. *arXiv preprint arXiv:2510.25110v1*, 2025. URL http://arxiv.org/abs/2510.25110v1.

[11] João Vitor de Carvalho Silva and Douglas G. Macharet. Can llm agents solve collaborative tasks? a study on urgency-aware planning and coordination. *arXiv preprint arXiv:2508.14635v1*, 2025. URL http://arxiv.org/abs/2508.14635v1.

[12] Jia-Kai Dong, I-Wei Huang, Chun-Tin Wu, and Yi-Tien Tsai. Msc-bench: A rigorous benchmark for multi-server tool orchestration. *arXiv preprint arXiv:2510.19423v1*, 2025. URL http://arxiv.org/abs/2510.19423v1.

[13] Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253v1*, 2024. URL http://arxiv.org/abs/2402.04253v1.

[14] Andreas Einwiller, Kanishka Ghosh Dastidar, Artur Romazanov, Annette Hautli-Janisz, Michael Granitzer, and Florian Lemmerich. Benevolent dictators? on llm agent behavior in dictator games. *arXiv preprint arXiv:2511.08721v1*, 2025. URL http://arxiv.org/abs/2511.08721v1.

[15] Junfeng Fang, Zijun Yao, Ruipeng Wang, Haokai Ma, Xiang Wang, and Tat-Seng Chua. We should identify and mitigate third-party safety risks in mcp-powered agent systems. *arXiv preprint arXiv:2506.13666v1*, 2025. URL http://arxiv.org/abs/2506.13666v1.

[16] Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. *arXiv preprint arXiv:2505.10978v3*, 2025. URL http://arxiv.org/abs/2505.10978v3.

[17] Xiaohan Fu, Shuheng Li, Zihan Wang, Yihao Liu, Rajesh K. Gupta, Taylor Berg-Kirkpatrick, and Earlence Fernandes. Imprompter: Tricking llm agents into improper tool use. *arXiv preprint arXiv:2410.14923v2*, 2024. URL http://arxiv.org/abs/2410.14923v2.

[18] Yuchuan Fu, Xiaohan Yuan, and Dongxia Wang. Ras-eval: A comprehensive benchmark for security evaluation of llm agents in real-world environments. *arXiv preprint arXiv:2506.15253v1*, 2025. URL http://arxiv.org/abs/2506.15253v1.

[19] Stefano Fumero, Kai Huang, Matteo Boffa, Danilo Giordano, Marco Mellia, Zied Ben Houidi, and Dario Rossi. Cybersleuth: Autonomous blue-team llm agent for web attack forensics. *arXiv preprint arXiv:2508.20643v1*, 2025. URL http://arxiv.org/abs/2508.20643v1.

[20] Tarek Gasmi, Ramzi Guesmi, Ines Belhadj, and Jihene Bennaceur. Bridging ai and software security: A comparative vulnerability assessment of llm agent deployment paradigms. *arXiv preprint arXiv:2507.06323v1*, 2025. URL http://arxiv.org/abs/2507.06323v1.

[21] Amur Ghose, Andrew B. Kahng, Sayak Kundu, and Zhiang Wang. Orfs-agent: Tool-using agents for chip design optimization. *arXiv preprint arXiv:2506.08332v2*, 2025. URL http://arxiv.org/abs/2506.08332v2.

[22] Tsimur Hadeliya, Mohammad Ali Jauhar, Nidhi Sakpal, and Diogo Cruz. When refusals fail: Unstable safety mechanisms in long-context llm agents. *arXiv preprint arXiv:2512.02445v1*, 2025. URL http://arxiv.org/abs/2512.02445v1.

[23] Dongyoon Hahm, Woogyeol Jin, June Suk Choi, Sungsoo Ahn, and Kimin Lee. Enhancing llm agent safety via causal influence prompting. *arXiv preprint arXiv:2507.00979v1*, 2025. URL http://arxiv.org/abs/2507.00979v1.

[24] Bingguang Hao, Zengzhuang Xu, Yuntao Wen, Xinyi Xu, Yang Liu, Tong Zhao, Maolin Wang, Long Chen, Dong Wang, Yicheng Chen, Cunyin Peng, Xiangyu Zhao, Chenyi Zhuang, and Ji Zhang. From failure to mastery: Generating hard samples for tool-use agents. *arXiv preprint arXiv:2601.01498v1*, 2026. URL http://arxiv.org/abs/2601.01498v1.

[25] Yuzhi Hao and Danyang Xie. A multi-llm-agent-based framework for economic and public policy analysis. *arXiv preprint arXiv:2502.16879v1*, 2025. URL http://arxiv.org/abs/2502.16879v1.

[26] Kostas Hatalis, Despina Christou, and Vyshnavi Kondapalli. Review of case-based reasoning for llm agents: Theoretical foundations, architectural components, and cognitive integration. *arXiv preprint arXiv:2504.06943v2*, 2025. URL http://arxiv.org/abs/2504.06943v2.

[27] Yufei He, Ruoyu Li, Alex Chen, Yue Liu, Yulin Chen, Yuan Sui, Cheng Chen, Yi Zhu, Luca Luo, Frank Yang, and Bryan Hooi. Enabling self-improving agents to learn at test time with human-in-the-loop guidance. *arXiv preprint arXiv:2507.17131v2*, 2025. URL http://arxiv.org/abs/2507.17131v2.

[28] Joey Hong, Anca Dragan, and Sergey Levine. Planning without search: Refining frontier llms with offline goal-conditioned rl. *arXiv preprint arXiv:2505.18098v2*, 2025. URL http://arxiv.org/abs/2505.18098v2.

[29] Hanjiang Hu, Changliu Liu, Na Li, and Yebin Wang. Training task reasoning llm agents for multi-turn task planning via single-turn reinforcement learning. *arXiv preprint arXiv:2509.20616v2*, 2025. URL http://arxiv.org/abs/2509.20616v2.

[30] Jingyi Huang, Yuyi Yang, Mengmeng Ji, Charles Alba, Sheng Zhang, and Ruopeng An. Use of retrieval-augmented large language model agent for long-form covid-19 fact-checking. *arXiv preprint arXiv:2512.00007v1*, 2025. URL http://arxiv.org/abs/2512.00007v1.

[31] Zimo Ji, Xunguang Wang, Zongjie Li, Pingchuan Ma, Yudong Gao, Daoyuan Wu, Xincheng Yan, Tian Tian, and Shuai Wang. Taxonomy, evaluation and exploitation of ipi-centric llm agent defense frameworks. *arXiv preprint arXiv:2511.15203v1*, 2025. URL http://arxiv.org/abs/2511.15203v1.

[32] Jingyi Jia and Qinbin Li. Autotool: Efficient tool selection for large language model agents. *arXiv preprint arXiv:2511.14650v1*, 2025. URL http://arxiv.org/abs/2511.14650v1.

[33] Neil Kale, Chen Bo Calvin Zhang, Kevin Zhu, Ankit Aich, Paula Rodriguez, Scale Red Team, Christina Q. Knight, and Zifan Wang. Reliable weak-to-strong monitoring of llm agents. *arXiv preprint arXiv:2508.19461v1*, 2025. URL http://arxiv.org/abs/2508.19461v1.

[34] Adharsh Kamath, Sishen Zhang, Calvin Xu, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Enforcing temporal constraints for llm agents. *arXiv preprint arXiv:2512.23738v1*, 2025. URL http://arxiv.org/abs/2512.23738v1.

[35] Doyoung Kim, Zhiwei Ren, Jie Hao, Zhongkai Sun, Lichao Wang, Xiyao Ma, Zack Ye, Xu Han, Jun Yin, Heng Ji, Wei Shen, Xing Fan, Benjamin Yao, and Chenlei Guo. Beyond perfect apis: A comprehensive evaluation of llm agents under real-world api complexity. *arXiv preprint arXiv:2601.00268v1*, 2026. URL http://arxiv.org/abs/2601.00268v1.

[36] Myung Ho Kim. Bridging symbolic control and neural reasoning in llm agents: The structured cognitive loop. *arXiv preprint arXiv:2511.17673v3*, 2025. URL http://arxiv.org/abs/2511.17673v3.

[37] Andrew Kiruluta. A novel architecture for symbolic reasoning with decision trees and llm agents. *arXiv preprint arXiv:2508.05311v1*, 2025. URL http://arxiv.org/abs/2508.05311v1.

[38] Jia Li, Xianjie Shi, Kechi Zhang, Ge Li, Zhi Jin, Lei Li, Huangzhao Zhang, Jia Li, Fang Liu, Yuwei Zhang, Zhengwei Tao, Yihong Dong, Yuqi Zhu, and Chongyang Tao. Graphcodeagent: Dual graph-guided llm agent for retrieval-augmented repo-level code generation. *arXiv preprint arXiv:2504.10046v2*, 2025. URL http://arxiv.org/abs/2504.10046v2.

[39] Weitang Li, Jiajun Ren, Lixue Cheng, and Cunxi Gong. Autonomous quantum simulation through large language model agents. *arXiv preprint arXiv:2601.10194v1*, 2026. URL http://arxiv.org/abs/2601.10194v1.

[40] Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, Fengli Xu, and Yong Li. Agentswift: Efficient llm agent design via value-guided hierarchical search. *arXiv preprint arXiv:2506.06017v2*, 2025. URL http://arxiv.org/abs/2506.06017v2.

[41] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, Rui Kong, Yile Wang, Hanfei Geng, Jian Luan, Xuefeng Jin, Zilong Ye, Guanjing Xiong, Fan Zhang, Xiang Li, Mengwei Xu, Zhijun Li, Peng Li, Yang Liu, Ya-Qin Zhang, and Yunxin Liu. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459v2*, 2024. URL http://arxiv.org/abs/2401.05459v2.

[42] Yuanhao Li, Mingshan Liu, Hongbo Wang, Yiding Zhang, Yifei Ma, and Wei Tan. Draft-rl: Multi-agent chain-of-draft reasoning for reinforcement learning-enhanced llms. *arXiv preprint arXiv:2511.20468v1*, 2025. URL http://arxiv.org/abs/2511.20468v1.

[43] Yuxuan Li, Sauvik Das, and Hirokazu Shirado. What makes llm agent simulations useful for policy? insights from an iterative design engagement in emergency preparedness. *arXiv preprint arXiv:2509.21868v1*, 2025. URL http://arxiv.org/abs/2509.21868v1.

[44] Zichuan Li, Jian Cui, Xiaojing Liao, and Luyi Xing. Les dissonances: Cross-tool harvesting and polluting in pool-of-tools empowered llm agents. *arXiv preprint arXiv:2504.03111v3*, 2025. URL http://arxiv.org/abs/2504.03111v3.

[45] Ilija Lichkovski, Alexander Müller, Mariam Ibrahim, and Tiwai Mhundwa. Eu-agent-bench: Measuring illegal behavior of llm agents under eu law. *arXiv preprint arXiv:2510.21524v1*, 2025. URL http://arxiv.org/abs/2510.21524v1.

[46] Jiayu Liu, Cheng Qian, Zhaochen Su, Qing Zong, Shijue Huang, Bingxiang He, and Yi R. Fung. Costbench: Evaluating multi-turn cost-optimal planning and adaptation in dynamic environments for llm tool-use agents. *arXiv preprint arXiv:2511.02734v1*, 2025. URL http://arxiv.org/abs/2511.02734v1.

[47] Marianne Menglin Liu, Daniel Garcia, Fjona Parllaku, Vikas Upadhyay, Syed Fahad Allam Shah, and Dan Roth. Toolscope: Enhancing llm agent tool use through tool merging and context-aware filtering. *arXiv preprint arXiv:2510.20036v1*, 2025. URL http://arxiv.org/abs/2510.20036v1.

[48] Shuang Liu, Ruijia Zhang, Ruoyun Ma, Yujia Deng, Lanyi Zhu, Jiayu Li, Zelong Li, Zhibin Shen, and Mengnan Du. Llm agents in law: Taxonomy, applications, and challenges. *arXiv preprint arXiv:2601.06216v1*, 2026. URL http://arxiv.org/abs/2601.06216v1.

[49] Tianming Liu, Jirong Yang, Yafeng Yin, Manzi Li, Linghao Wang, and Zheng Zhu. Aligning llm agents with human learning and adjustment behavior: a dual agent approach. *arXiv preprint arXiv:2511.00993v1*, 2025. URL http://arxiv.org/abs/2511.00993v1.

[50] Wenrui Liu, Zixiang Liu, Elsie Dai, Wenhan Yu, Lei Yu, and Tong Yang. Mcpagentbench: A real-world task benchmark for evaluating llm agent mcp tool use. *arXiv preprint arXiv:2512.24565v2*, 2025. URL http://arxiv.org/abs/2512.24565v2.

[51] Elias Lumer, Anmol Gulati, Vamse Kumar Subbiah, Pradeep Honaganahalli Basavaraju, and James A. Burke. Memtool: Optimizing short-term memory management for dynamic tool calling in llm agent multi-turn conversations. *arXiv preprint arXiv:2507.21428v1*, 2025. URL http://arxiv.org/abs/2507.21428v1.

[52] Weidi Luo, Shenghong Dai, Xiaogeng Liu, Suman Banerjee, Huan Sun, Muhao Chen, and Chaowei Xiao. Agrail: A lifelong agent guardrail with effective and adaptive safety detection. *arXiv preprint arXiv:2502.11448v2*, 2025. URL http://arxiv.org/abs/2502.11448v2.

[53] Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. Evaluation and benchmarking of llm agents: A survey. *arXiv preprint arXiv:2507.21504v1*, 2025. URL http://arxiv.org/abs/2507.21504v1.

[54] Nayantara Mudur, Hao Cui, Subhashini Venugopalan, Paul Raccuglia, Michael P. Brenner, and Peter Norgaard. Feabench: Evaluating language models on multiphysics reasoning ability. *arXiv preprint arXiv:2504.06260v1*, 2025. URL http://arxiv.org/abs/2504.06260v1.

[55] Katsuaki Nakano, Reza Fayyazi, Shanchieh Jay Yang, and Michael Zuzak. Guided reasoning in llm-driven penetration testing using structured attack trees. *arXiv preprint arXiv:2509.07939v2*, 2025. URL http://arxiv.org/abs/2509.07939v2.

[56] Vikram Nitin, Baishakhi Ray, and Roshanak Zilouchian Moghaddam. Faultline: Automated proof-of-vulnerability generation using llm agents. *arXiv preprint arXiv:2507.15241v1*, 2025. URL http://arxiv.org/abs/2507.15241v1.

[57] Charidimos Papadakis, Angeliki Dimitriou, Giorgos Filandrianos, Maria Lymperaiou, Konstantinos Thomas, and Giorgos Stamou. Atlas: Adaptive trading with llm agents through dynamic prompt optimization and multi-agent coordination. *arXiv preprint arXiv:2510.15949v2*, 2025. URL http://arxiv.org/abs/2510.15949v2.

[58] Aske Plaat, Max van Duijn, Niki van Stein, Mike Preuss, Peter van der Putten, and Kees Joost Batenburg. Agentic large language models, a survey. *arXiv preprint arXiv:2503.23037v3*, 2025. URL http://arxiv.org/abs/2503.23037v3.

[59] Ron F. Del Rosario, Klaudia Krawiecka, and Christian Schroeder de Witt. Architecting resilient llm agents: A guide to secure plan-then-execute implementations. *arXiv preprint arXiv:2509.08646v1*, 2025. URL http://arxiv.org/abs/2509.08646v1.

[60] Anjana Sarkar and Soumyendu Sarkar. Survey of llm agent communication with mcp: A software design pattern centric review. *arXiv preprint arXiv:2506.05364v1*, 2025. URL http://arxiv.org/abs/2506.05364v1.

[61] Vishnu Sarukkai, Asanshay Gupta, James Hong, Michaël Gharbi, and Kayvon Fatahalian. In-context distillation with self-consistency cascades: A simple, training-free way to reduce llm agent costs. *arXiv preprint arXiv:2512.02543v1*, 2025. URL http://arxiv.org/abs/2512.02543v1.

[62] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761v1*, 2023. URL http://arxiv.org/abs/2302.04761v1.

[63] Zeyang Sha, Hanling Tian, Zhuoer Xu, Shiwen Cui, Changhua Meng, and Weiqiang Wang. Agent safety alignment via reinforcement learning. *arXiv preprint arXiv:2507.08270v1*, 2025. URL http://arxiv.org/abs/2507.08270v1.

[64] Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. *arXiv preprint arXiv:2410.06153v3*, 2024. URL http://arxiv.org/abs/2410.06153v3.

[65] Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324v3*, 2024. URL http://arxiv.org/abs/2401.07324v3.

[66] Tianneng Shi, Jingxuan He, Zhun Wang, Hongwei Li, Linyu Wu, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for llm agents. *arXiv preprint arXiv:2504.11703v2*, 2025. URL http://arxiv.org/abs/2504.11703v2.

[67] Wenqi Shi, Ran Xu, Yuchen Zhuang, Yue Yu, Jieyu Zhang, Hang Wu, Yuanda Zhu, Joyce Ho, Carl Yang, and May D. Wang. Ehragent: Code empowers large language models for few-shot complex tabular reasoning on electronic health records. *arXiv preprint arXiv:2401.07128v3*, 2024. URL http://arxiv.org/abs/2401.07128v3.

[68] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366v4*, 2023. URL http://arxiv.org/abs/2303.11366v4.

[69] Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Zhicheng Dou, and Ji-Rong Wen. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808v1*, 2026. URL http://arxiv.org/abs/2601.05808v1.

[70] Dehao Tao, Guoliang Ma, Yongfeng Huang, and Minghu Jiang. Membox: Weaving topic continuity into long-range memory for llm agents. *arXiv preprint arXiv:2601.03785v1*, 2026. URL http://arxiv.org/abs/2601.03785v1.

[71] Vali Tawosi, Salwa Alamir, Xiaomo Liu, and Manuela Veloso. Meta-rag on large codebases using code summarization. *arXiv preprint arXiv:2508.02611v1*, 2025. URL http://arxiv.org/abs/2508.02611v1.

[72] Elizaveta Tennant, Stephen Hailes, and Mirco Musolesi. Moral alignment for llm agents. *arXiv preprint arXiv:2410.01639v4*, 2024. URL http://arxiv.org/abs/2410.01639v4.

[73] Minh-Hao Van, Prateek Verma, Chen Zhao, and Xintao Wu. A survey of ai for materials science: Foundation models, llm agents, datasets, and tools. *arXiv preprint arXiv:2506.20743v1*, 2025. URL http://arxiv.org/abs/2506.20743v1.

[74] Nikhil Verma. Active context compression: Autonomous memory management in llm agents. *arXiv preprint arXiv:2601.07190v1*, 2026. URL http://arxiv.org/abs/2601.07190v1.

[75] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291v2*, 2023. URL http://arxiv.org/abs/2305.16291v2.

[76] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432v7*, 2023. URL http://arxiv.org/abs/2308.11432v7.

[77] Zizhao Wang, Dingcheng Li, Vaishakh Keshava, Phillip Wallis, Ananth Balashankar, Peter Stone, and Lukas Rutishauser. Adversarial reinforcement learning for large language model agent safety. *arXiv preprint arXiv:2510.05442v1*, 2025. URL http://arxiv.org/abs/2510.05442v1.

[78] Kavindu Warnakulasuriya, Prabhash Dissanayake, Navindu De Silva, Stephen Cranefield, Bastin Tony Roy Savarimuthu, Surangika Ranathunga, and Nisansa de Silva. Evolution of cooperation in llm-agent societies: A preliminary study using different punishment strategies. *arXiv preprint arXiv:2504.19487v3*, 2025. URL http://arxiv.org/abs/2504.19487v3.

[79] Chunlong Wu, Ye Luo, Zhibo Qu, and Min Wang. Meta-policy reflexion: Reusable reflective memory and rule admissibility for resource-efficient llm agent. *arXiv preprint arXiv:2509.03990v2*, 2025. URL http://arxiv.org/abs/2509.03990v2.

[80] Haolun Wu, Zhenkun Li, and Lingyao Li. Can llm agents really debate? a controlled study of multi-agent debate in logical reasoning. *arXiv preprint arXiv:2511.07784v1*, 2025. URL http://arxiv.org/abs/2511.07784v1.

[81] Panlong Wu, Kangshuo Li, Junbao Nan, and Fangxin Wang. Federated in-context llm agent learning. *arXiv preprint arXiv:2412.08054v1*, 2024. URL http://arxiv.org/abs/2412.08054v1.

[82] Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079v1*, 2025. URL http://arxiv.org/abs/2510.16079v1.

[83] Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis N. Ioannidis, Karthik Subbian, Jure Leskovec, and James Zou. Avatar: Optimizing llm agents for tool usage via contrastive reasoning. *arXiv preprint arXiv:2406.11200v3*, 2024. URL http://arxiv.org/abs/2406.11200v3.

[84] Ziqiao Xi, Shuang Liang, Qi Liu, Jiaqing Zhang, Letian Peng, Fang Nan, Meshal Nayim, Tianhui Zhang, Rishika Mundada, Lianhui Qin, Biwei Huang, and Kun Zhou. Toolgym: an open-world tool-using environment for scalable agent testing and data curation. *arXiv preprint arXiv:2601.06328v1*, 2026. URL http://arxiv.org/abs/2601.06328v1.

[85] Yu Xia, Yiran Shen, Junda Wu, Tong Yu, Sungchul Kim, Ryan A. Rossi, Lina Yao, and Julian McAuley. Sand: Boosting llm agents with self-taught action deliberation. *arXiv preprint arXiv:2507.07441v2*, 2025. URL http://arxiv.org/abs/2507.07441v2.

[86] Jingao Xu, Shuoyoucheng Ma, Xin Song, Rong Jiang, Hongkui Tu, and Bin Zhou. Exemplar-guided planing: Enhanced llm agent for kgqa. *arXiv preprint arXiv:2510.15283v1*, 2025. URL http://arxiv.org/abs/2510.15283v1.

[87] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110v11*, 2025. URL http://arxiv.org/abs/2502.12110v11.

[88] Weihao Xuan, Qingcheng Zeng, Heli Qi, Yunze Xiao, Junjue Wang, and Naoto Yokoya. The confidence dichotomy: Analyzing and mitigating miscalibration in tool-use agents. *arXiv preprint arXiv:2601.07264v1*, 2026. URL http://arxiv.org/abs/2601.07264v1.

[89] Bufang Yang, Lilin Xu, Liekang Zeng, Yunqi Guo, Siyang Jiang, Wenrui Lu, Kaiwei Liu, Hancheng Xiang, Xiaofan Jiang, Guoliang Xing, and Zhenyu Yan. Proagent: Harnessing on-demand sensory contexts for proactive llm agent systems. *arXiv preprint arXiv:2512.06721v1*, 2025. URL http://arxiv.org/abs/2512.06721v1.

[90] Bufang Yang, Lilin Xu, Liekang Zeng, Kaiwei Liu, Siyang Jiang, Wenrui Lu, Hongkai Chen, Xiaofan Jiang, Guoliang Xing, and Zhenyu Yan. Contextagent: Context-aware proactive llm agents with open-world sensory perceptions. *arXiv preprint arXiv:2505.14668v2*, 2025. URL http://arxiv.org/abs/2505.14668v2.

[91] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629v3*, 2022. URL http://arxiv.org/abs/2210.03629v3.

[92] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601v2*, 2023. URL http://arxiv.org/abs/2305.10601v2.

[93] Ye Ye. Task memory engine (tme): Enhancing state awareness for multi-step llm agent tasks. *arXiv preprint arXiv:2504.08525v4*, 2025. URL http://arxiv.org/abs/2504.08525v4.

[94] Ye Ye. Task memory engine: Spatial memory for robust multi-step llm agents. *arXiv preprint arXiv:2505.19436v1*, 2025. URL http://arxiv.org/abs/2505.19436v1.

[95] Yauwai Yim, Chunkit Chan, Tianyu Shi, Zheye Deng, Wei Fan, Tianshi Zheng, and Yangqiu Song. Evaluating and enhancing llms agent based on theory of mind in guandan: A multi-player cooperative game under imperfect information. *arXiv preprint arXiv:2408.02559v1*, 2024. URL http://arxiv.org/abs/2408.02559v1.

[96] Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885v1*, 2026. URL http://arxiv.org/abs/2601.01885v1.

[97] Rasoul Zahedifar, Sayyed Ali Mirghasemi, Mahdieh Soleymani Baghshah, and Alireza Taheri. Llm-agent-controller: A universal multi-agent large language model system as a control engineer. *arXiv preprint arXiv:2505.19567v1*, 2025. URL http://arxiv.org/abs/2505.19567v1.

[98] Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279v12*, 2024. URL http://arxiv.org/abs/2411.18279v12.

[99] Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. Datascibench: An llm agent benchmark for data science. *arXiv preprint arXiv:2502.13897v1*, 2025. URL http://arxiv.org/abs/2502.13897v1.

[100] Dongsen Zhang, Zekun Li, Xu Luo, Xuannan Liu, Peipei Li, and Wenjun Xu. Mcp security bench (msb): Benchmarking attacks against model context protocol in llm agents. *arXiv preprint arXiv:2510.15994v1*, 2025. URL http://arxiv.org/abs/2510.15994v1.

[101] Guibin Zhang, Haiyang Yu, Kaiming Yang, Bingli Wu, Fei Huang, Yongbin Li, and Shuicheng Yan. Evoroute: Experience-driven self-routing llm agent systems. *arXiv preprint arXiv:2601.02695v1*, 2026. URL http://arxiv.org/abs/2601.02695v1.

[102] Kaiyuan Zhang, Zian Su, Pin-Yu Chen, Elisa Bertino, Xiangyu Zhang, and Ninghui Li. Llm agents should employ security principles. *arXiv preprint arXiv:2505.24019v1*, 2025. URL http://arxiv.org/abs/2505.24019v1.

[103] Xin Zhang, Lissette Iturburu, Juan Nicolas Villamizar, Xiaoyu Liu, Manuel Salmeron, Shirley J. Dyke, and Julio Ramirez. Large language model agent for structural drawing generation using react prompt engineering and retrieval augmented generation. *arXiv preprint arXiv:2507.19771v1*, 2025. URL http://arxiv.org/abs/2507.19771v1.

[104] Zehua Zhang, Ati Priya Bajaj, Divij Handa, Siyu Liu, Arvind S Raj, Hongkai Chen, Hulin Wang, Yibo Liu, Zion Leonahenahe Basque, Souradip Nath, Vishal Juneja, Nikhil Chapre, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. Buildbench: Benchmarking llm agents on compiling real-world open-source software. *arXiv preprint arXiv:2509.25248v1*, 2025. URL http://arxiv.org/abs/2509.25248v1.

[105] Zhexin Zhang, Shiyao Cui, Yida Lu, Jingzhuo Zhou, Junxiao Yang, Hongning Wang, and Minlie Huang. Agent-safetybench: Evaluating the safety of llm agents. *arXiv preprint arXiv:2412.14470v2*, 2024. URL http://arxiv.org/abs/2412.14470v2.

[106] Haiteng Zhao, Junhao Shen, Yiming Zhang, Songyang Gao, Kuikun Liu, Tianyou Ma, Fan Zheng, Dahua Lin, Wenwei Zhang, and Kai Chen. Achieving olympia-level geometry large language model agent via complexity boosting reinforcement learning. *arXiv preprint arXiv:2512.10534v2*, 2025. URL http://arxiv.org/abs/2512.10534v2.

[107] Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. Impossiblebench: Measuring llms' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270v1*, 2025. URL http://arxiv.org/abs/2510.20270v1.

[108] Xingfu Zhou and Pengfei Wang. Reasoning-style poisoning of llm agents via stealthy style transfer: Process-level attacks and runtime monitoring in rsv space. *arXiv preprint arXiv:2512.14448v1*, 2025. URL http://arxiv.org/abs/2512.14448v1.

[109] Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. Archer: Training language model agents via hierarchical multi-turn rl. *arXiv preprint arXiv:2402.19446v1*, 2024. URL http://arxiv.org/abs/2402.19446v1.

[110] Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. *arXiv preprint arXiv:2506.01716v1*, 2025. URL http://arxiv.org/abs/2506.01716v1.

[111] Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370v1*, 2025. URL http://arxiv.org/abs/2509.25370v1.