Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Failure Recovery in Tool-Using Large Language Model Agents: A Structured Review of Detection, Correction, and State Consistency

Ravi Shanker Thadishetti

thadishetti.ravishanker@gmail.com

University of the Cumberlands

Vidya Manyam

Belhaven University

Research Article

Keywords: Tool-using LLM agents, Failure recovery, Silent failures, Self-correction, State consistency

Posted Date: November 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11425438/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Abstract

Large language model agents increasingly act on the world through tools such as code interpreters, application programming interfaces, search engines, perception modules and robotic planners. Every tool call is an opportunity for failure: the model may choose the wrong tool, write malformed input, misread a returned value, trust an erroneous output, or leave external systems in a half-changed condition after a partial sequence of actions. This paper is a structured, qualitative review of how the literature detects such failures, corrects them and keeps the state of the agent and its environment consistent while doing so. It adapts the feedback-loop model and the loud/silent distinction of Sun et al. (2024) to organise a taxonomy of failures, groups detection methods into execution signals, output verification, model self-assessment, external critics and human oversight, correction methods into retry, repair, reflection, replanning, rollback and escalation, and consistency mechanisms into idempotency, compensating actions, checkpointing, action logging, state verification and memory hygiene. A study-by-study comparison of nine representative works, including 2025–2026 studies, records their methods, benchmarks, findings, limitations and treatment of state consistency. Within the reviewed corpus, explicit error messages are comparatively well exploited, silent failures appear to remain a major open problem, self-correction without external feedback is reported to be unreliable on reasoning tasks, and state consistency receives less attention than detection and correction. The paper closes with recommendations for evaluation and design and an agenda for future research. The review is not exhaustive and reports no new experiments.

1. INTRODUCTION

A language model that only produces text can be wrong, but its errors are at least contained in the text it produces. A language model that calls tools is different. When it writes a database query, sends a message, edits a file, moves a robot arm or books a ticket, its output becomes an action with consequences outside the conversation, and a mistaken action may be costly or impossible to take back. Tool use has been a main route by which models acquire capabilities that they lack on their own, including exact arithmetic, access to current information, execution of code and control of software and physical systems (Lewis et al., 2020; Schick et al., 2023; G. Wang et al., 2023; Yao et al., 2023b). As deployments move from demonstrations to routine use, the question is no longer only whether an agent can complete a task under favourable conditions, but what it does when something goes wrong.

Tool-use failures are widely reported. Tools reject malformed input, return empty results, time out, change their behaviour between versions and sometimes return answers that are syntactically perfect and substantively false. The model itself may select the wrong tool, supply the wrong arguments, misunderstand a response or lose track of what it has already done. Several benchmarks that measure agents in realistic multi-step settings report that success falls as tasks grow longer or as the required consistency between steps increases (Jimenez et al., 2024; X. Liu et al., 2024; Yao et al., 2024). An agent that is excellent at the first call but unable to notice or repair its own errors is of limited practical value, because errors can compound along a chain of dependent actions.

Failure recovery is the set of capabilities that allow an agent to cope with these situations. It has three parts that are often discussed separately but are tightly connected. *Detection* is the ability to notice that something has gone wrong, ideally before the consequences spread. *Correction* is the ability to choose and carry out a

remedy. This review distinguishes two basic remedies that are often conflated: a *retry* re-issues the same request unchanged, and is appropriate only when the fault is transient or the outcome is unknown; a *repair* changes the input, the arguments or the tool in light of the error, and is appropriate only when the request itself was at fault. Further remedies are reflection, replanning, rollback and escalation to a person. *State consistency* is the ability to ensure that, after a failure and its correction, the internal record of the agent, the content of its memory and the actual condition of the external world still agree with one another. This third part appears to be the least studied in the literature reviewed here. An agent that repairs a failed call but forgets that an earlier step has already changed a record may repeat the change, and an agent that retries a payment without knowing whether the first attempt succeeded may charge the user twice.

This paper provides a structured review of the work on these three themes. It does not claim to have invented its organising concepts; Section 3.3 states explicitly what is adapted from earlier work and what the review adds. Its contributions are as follows. First, it adapts the feedback-loop model of tool use of Sun et al. (2024), shown in Figure 1, and uses it to organise a taxonomy of failures. Second, it organises detection, correction and consistency mechanisms into families and compares their strengths and weaknesses, with an explicit distinction between retry and repair. Third, it provides a study-by-study comparison of representative works, including recent 2025–2026 studies, along methods, benchmarks, findings, limitations and state consistency (Table 1). Fourth, it examines how existing evaluations treat recovery and proposes measures that would close the gaps. Fifth, it identifies open problems, with particular attention to silent failures and to state consistency across multi-step action sequences. The remainder of the paper is organised as follows. Section 2 gives background and reviews recent studies, Section 3 positions the review against earlier surveys and states its contribution, Section 4 describes the review approach, Sections 5 to 8 present the taxonomy and the three families of mechanisms, Section 9 discusses evaluation, Section 10 discusses findings, recommendations, open challenges and limitations, and Section 11 concludes.

2. BACKGROUND AND LITERATURE

2.1 Tool Use in Language Models

Early demonstrations showed that models could be taught to call external programs. Toolformer trained a model to decide for itself when to call a calculator, a search engine or a translation system and to insert the result into its text (Schick et al., 2023). Program-aided language models offloaded exact computation to an interpreter, with the model writing code and the interpreter producing the answer (Gao et al., 2023). The reasoning-and-acting pattern interleaves written reasoning with actions, so that the model can plan, act, observe the result and revise (Yao et al., 2023b). Later work scaled the idea to large collections of real application programming interfaces, with models trained or prompted to select among thousands of tools and to compose their calls (Li et al., 2023; Patil et al., 2023; Qin et al., 2024). In robotics and embodied settings, language models were coupled to perception and skill libraries, using the feedback from the scene to adjust their plans (Ahn et al., 2022; Huang et al., 2022).

Across these systems a common structure emerges, and it is the structure shown in panel (a) of Fig. 1, which is reproduced from Sun et al. (2024). A model receives an input, together with a context that may include environment observations, tool documentation, earlier tool errors and user feedback. It decides which tool to

use and what input to give it, the tool runs, and the tool output returns to the model, which may call another tool or produce a final answer. Because the context contains the outputs of earlier tools, the loop is a feedback loop, and this is what makes recovery possible in principle: the model can see what happened and act on it. It is also what makes recovery difficult in practice, since the model must interpret that feedback correctly.

2.2 Concepts of Faults, Errors and Failures

The dependability literature offers precise vocabulary that is useful here. A fault is a defect, an error is the part of the system state that deviates because of a fault, and a failure is the event in which the delivered service departs from what is required (Avizienis et al., 2004). A fault may be dormant for a long time and become active only under particular inputs. This distinction matters for agents because the same underlying weakness, such as a model's tendency to confuse two similar objects, may stay hidden for many tasks and then produce an error in the context of one tool call, which becomes a failure only if it propagates to an action or an answer. Recovery measures can therefore act at three points: they can prevent the fault from being activated, detect the error before it becomes a failure, or contain and repair the failure after it occurs.

2.3 Reasoning and Self-Correction

A large body of work asks whether language models can correct themselves. Methods such as iterative self-refinement have the model generate an answer, criticise it and revise it, and report improvements on several tasks (Madaan et al., 2023). Reflective approaches let an agent write a verbal analysis of why an attempt failed and store it for later attempts, which improved performance on decision-making, reasoning and coding tasks when real feedback such as test results or environment signals was available (Shinn et al., 2023). Search-based methods explore several candidate lines of reasoning or action and evaluate them, allowing the agent to backtrack (Yao et al., 2023a; Zhou et al., 2024). At the same time, careful studies have reported that models often fail to improve, and sometimes degrade their answers, when asked to revise reasoning without any external information, and that apparent gains in earlier reports frequently came from the use of ground-truth signals or from weak initial prompts (Huang et al., 2024; Kamoi et al., 2024). This distinction between correction with and without external feedback runs through the whole review, and it is the reason why tool outputs, which are a form of external feedback, are so central to recovery.

2.4 Failures Reported in Agent Benchmarks

Benchmarks have exposed several types of failure that agents encounter. Evaluations in interactive operating-system, web, database and game environments report recurring problems with long-horizon planning, invalid actions, failure to follow format requirements and premature termination (X. Liu et al., 2024). Software-engineering benchmarks show that agents can produce edits that look plausible but break other behaviour, a form of silent failure that is detected by running tests (Jimenez et al., 2024). Customer-service style benchmarks that require an agent to follow policies while using tools and conversing with a simulated user report that performance drops considerably when the same task is attempted repeatedly, showing that reliability, not just peak accuracy, is an issue (Yao et al., 2024); the later τ^2-bench extends this style of evaluation to a dual-control environment (Barres et al., 2025). These findings motivate the focus of this review on detection and consistency, not only on the quality of the model's reasoning.

2.5 Recent Studies on Tool-Failure Recovery (2025–2026)

Since 2024 a growing number of studies have made failure recovery in tool-using agents their direct subject. Zhu et al. (2025) introduce a modular taxonomy of agent failures spanning memory, reflection, planning, action and system-level operations, build an annotated benchmark of failure trajectories from ALFWorld, GAIA and WebShop, and propose AgentDebug, which isolates a root-cause failure and feeds corrective feedback back to the agent. They report that early errors cascading into later failures are the key bottleneck, and that the corrective feedback yields up to 26% relative improvement in task success. Chang and Geng (2025) approach the problem from the database side: SagaLLM combines the Saga transaction pattern with persistent memory, automated compensation and independent validation agents for multi-agent planning, and, by its authors' account, relaxes strict ACID guarantees while still providing workflow-wide consistency and recovery.

Two 2026 studies measure recovery under controlled fault injection. ToolMaze (Zhu et al., 2026) crosses task complexity, expressed as directed acyclic graphs of tool calls, with four perturbation modes (explicit or implicit, transient or permanent). It reports that implicit semantic failures cut the perturbation recovery rate by about 37% relative to explicit errors, and that fault tolerance improves more slowly with model scale than basic task completion. Kraishan (2026) wraps the executable environments of an established function-calling benchmark in a fault-injection layer with four typed faults and separates noticing, replanning, recovery and repetition. In that study agents treated a failure as a problem in 91.3% of trials when the tool returned an explicit error, but in 58.8% of trials when it returned a plausible wrong value, against a 26.8% rate of reporting problems when nothing was wrong; returning to the same tool three or more times in a row occurred in up to 22.2% of trials. The author cautions that the study cannot show that detection is sufficient for recovery. A different kind of evidence comes from a longitudinal case study of a production agent runtime (Wu, 2026), which documented 22 incidents over eight weeks and found that about 70% of silent failures were caught by human observation of user-visible output rather than by tests or audits.

Taken together, these studies agree on three points: agents tend to over-trust tool outputs that fail silently (consistent with Sun et al., 2024), recovery improves when external feedback or independent verification is available, and recovery effort can itself become wasteful through repeated calls. They differ in how much attention they give to state: only Chang and Geng (2025) treat consistency of state as the central design goal. Table 1 compares these studies with earlier foundational works along the dimensions used in the rest of this review.

Table 1

Study-by-Study Comparison of Representative Works on Failure Detection, Correction and State Consistency

Study	Method	Benchmark / setting	Main reported findings	Limitations	State consistency
Sun et al. (2024)	Framework for tool errors; in-context interventions (disclaimer, confidence, checklist) to detect silent tool errors	Controlled faulty-calculator arithmetic; embodied agent (ALFRED) with object detector and action planner	Models tend to over-trust tool outputs; simple in-context interventions improve detection of incorrect outputs	Detection is the focus; recovery planning only initially explored; controlled and simulated settings	Not addressed
Gou et al. (2024)	CRITIC: model verifies and revises its output through tool-interactive critiquing (search engine, code interpreter)	Free-form question answering, mathematical program synthesis, toxicity reduction	Critiquing with external tools improves results; self-verification without external feedback is reported to be unreliable	Tools serve as verifiers; tasks with external side effects are not the focus	Not addressed
Shinn et al. (2023)	Reflexion: verbal reflections on failed attempts stored in memory for later trials	Decision-making (ALFWorld), reasoning (HotPotQA), coding (HumanEval)	Gains when feedback is informative (test results, environment signals)	Depends on feedback quality; a wrong reflection can be stored and reused; no validation of stored lessons	Memory only; stored lessons are not validated
Yao et al. (2024)	τ-bench: simulated user, domain tools and policies; repeated-trial reliability (pass^k)	Retail and airline customer-service domains	Success falls when the same task is repeated, showing a reliability gap beyond single-run accuracy	A benchmark, not a recovery method; tool faults are not injected as a controlled variable	Partial: compares final database state with the goal state
Zhu et al. (2025)	AgentDebug: failure taxonomy (memory, reflection, planning, action, system), root-cause localisation, corrective feedback	AgentErrorBench built from ALFWorld, GAIA and WebShop trajectories	Error propagation is the key bottleneck; +24% all-correct and + 17% step accuracy in localisation over the strongest	Three benchmarks; analyses trajectories after failure rather than guaranteeing world state	Partial: memory- and system-level error classes; no world-state guarantee

Note. Entries summarise each study as described in its own abstract or main text. The final column records this review's reading of what the study addresses; it is not a claim made by the original authors. 2025–2026 entries include preprints that may not have completed peer review.

Study	Method	Benchmark / setting	Main reported findings	Limitations	State consistency
			baseline; up to 26% relative gain in task success		
Chang & Geng (2025)	SagaLLM: Saga transaction pattern with persistent memory, automated compensation and independent validation agents	Multi-agent planning domains	Standalone LLMs often violate interdependent constraints or fail to recover from disruptions; SagaLLM improves consistency and validation accuracy	Relaxes strict ACID guarantees; evaluated on planning domains rather than live external services	Central: compensation, checkpointing, persistent state
Zhu et al. (2026)	ToolMaze: DAG-based tasks crossed with explicit/implicit and transient/permanent perturbations; recovery rate and recovery cost metrics	270-tool simulated corpus; several recent models	Implicit failures cut recovery rate by about 37%; fault tolerance improves more slowly with scale than task completion; agents fall into futile trial-and-error	Controlled topologies and simulated tools; taxonomy of perturbations may not cover all faults	Not a reported metric; action tools with side effects are modelled as terminal nodes
Kraishan (2026)	Fault-injection layer over an executable function-calling benchmark; four typed faults; separate measures of noticing, replanning, recovery, repetition	Multi-turn tool-using agents, including reasoning models	Problems flagged in 91.3% of trials with explicit errors vs 58.8% with plausible wrong values (26.8% false-alarm baseline); repeated returns to the same tool in up to 22.2% of trials	One injected fault per trajectory; author notes detection is not shown to be sufficient for recovery	Environment consistent by design; recovery scored against the fault-free end state
Wu (2026)	Longitudinal postmortem study of 22 incidents and a taxonomy of silent failures	One production personal-assistant runtime (about 40 scheduled jobs, 8 LLM	About 70% of silent failures caught by human observation of user-visible	Single system, case-study design; generalisation unknown	Not the focus; documents failures at component seams

Note. Entries summarise each study as described in its own abstract or main text. The final column records this review's reading of what the study addresses; it is not a claim made by the original authors. 2025–2026 entries include preprints that may not have completed peer review.

Study	Method	Benchmark / setting	Main reported findings	Limitations	State consistency
		providers), eight weeks	output; longest-lived failures sat at seams between components		

Note. Entries summarise each study as described in its own abstract or main text. The final column records this review's reading of what the study addresses; it is not a claim made by the original authors. 2025–2026 entries include preprints that may not have completed peer review.

3. RELATED WORK AND CONTRIBUTION OF THIS REVIEW

3.1 Existing Surveys

Several surveys cover neighbouring topics. Surveys of language-model agents describe architectures built around planning, memory and tool use and catalogue applications (L. Wang et al., 2024). Surveys of tool learning organise work by the stages of tool selection, calling and result integration and discuss data and training methods (Qin et al., 2023). Surveys of hallucination examine the causes and mitigation of unfaithful or fabricated output and are relevant to silent failure, although they concentrate on text and not on actions (Ji et al., 2023). Surveys of self-correction classify methods according to when and by whom corrections are made and stress the importance of reliable feedback sources (Kamoi et al., 2024). To the authors' knowledge, none of these surveys takes failure recovery in tool-using agents as its organising theme or treats the consistency of state across failures as a topic in its own right. Recent empirical studies (Section 2.5) address parts of the problem but are individual studies, not syntheses.

3.2 Classical Foundations

The review draws on a second body of work outside the language-model literature: distributed systems, databases and software reliability engineering. Long-running transactions in databases led to compensating actions, in which each step in a sequence has a defined undo operation that can be applied if a later step fails (Garcia-Molina & Salem, 1987). Practitioners of resilient service design developed patterns such as timeouts, bounded retries, circuit breakers and idempotent operations to stop faults from spreading (Nygard, 2007). These ideas are directly relevant to agents, which are in effect clients that call unreliable services and perform long sequences of actions. Where the agent literature has independently rediscovered a classical technique, the review notes the correspondence, because it allows decades of engineering experience to be applied rather than reinvented.

3.3 Contribution Beyond Existing Surveys and Borrowed Frameworks

Several of the review's building blocks are borrowed, and they are credited here. The feedback-loop model of tool use (Fig. 1) and the distinction between loud failures with explicit signals and silent failures without them come from Sun et al. (2024). The vocabulary of faults, errors and failures comes from Avizienis et al. (2004). Compensating actions come from Garcia-Molina and Salem (1987), and the patterns of bounded retry, circuit

breaker and idempotent operation from Nygard (2007). The grouping of self-correction by source of feedback follows Kamoi et al. (2024).

The review adds the following. (i) It places detection, correction and state consistency in a single comparative structure in which the choice of remedy is tied to the location of the failure in the loop and to its effect on external state. Earlier work treats these parts separately or concentrates on detection. (ii) It separates retry, which re-issues an unchanged request, from repair, which changes the request, and states the conditions under which each is appropriate and safe (Section 7). (iii) It distinguishes three states that can diverge after a failure (world, agent and memory) and maps classical consistency mechanisms onto them (Section 8). (iv) It compares nine representative studies side by side, including a column on state consistency (Table 1). (v) It proposes recovery-specific evaluation measures (Table 6). The contribution is organisational and analytical. The review reports no new experiments, and the proposed measures have not been validated empirically.

4. REVIEW APPROACH

4.1 Scope and Questions

The review addresses four questions. What kinds of failure arise when language-model agents use tools? How are these failures detected? How are they corrected? How is consistency between the agent's state and the state of the world preserved when failures occur? The scope covers agents that call external tools in order to act or to obtain information, including code execution, application programming interfaces, retrieval, perception modules, planners and physical or simulated embodiments. Work on pure text generation without tools is considered only when its methods are directly transferable, as in the case of self-correction.

4.2 Source Selection and Synthesis

This is a structured, qualitative review and not a systematic review in the sense of a registered protocol with exhaustive database searches and reported screening counts. It does not follow PRISMA, no formal counts of records screened are reported, and the corpus is the set of works cited here. Readers who need exhaustive coverage should treat the paper as a structured map and extend it with a registered search.

The core corpus was assembled from the main venues for machine learning, natural language processing, software engineering and robotics, and from the dependability and distributed-systems literature for foundational concepts. To bring coverage up to date, targeted searches of arXiv, the ACL Anthology, the proceedings of VLDB and publisher pages were run in October 2026 with combinations of the terms failure recovery, tool-using LLM agents, silent tool errors, fault injection, error recovery benchmark and transactional guarantees for agents. Works were included if they proposed or evaluated a method of detecting, correcting or containing failures in tool-using agents, reported a benchmark or analysis of agent failures, or supplied a classical technique that the agent literature has adopted. Each included study in Table 1 was read for the same five attributes: method, benchmark or setting, main findings, limitations and treatment of state consistency. No formal risk-of-bias appraisal was performed. Preprints are marked as such in the reference list, and their findings should be read as provisional.

5. A TAXONOMY OF TOOL-USE FAILURES

Using the feedback-loop model of Sun et al. (2024) in Fig. 1, failures can be located by the point in the loop at which the fault arises. Four locations are distinguished, and Table 2 summarises them with examples. The first is selection: the model chooses a tool that cannot perform the required operation, or fails to use a tool when it should. The second is input: the model supplies arguments that are malformed, out of range, ambiguous or inconsistent with the documentation. The third is execution: the tool itself fails because of a bug, a timeout, a permission problem, a rate limit or a change in an external service. The fourth is interpretation: the tool returns a correct result that the model misreads, or an incorrect result that the model accepts without question. A fifth category, state failures, cuts across the other four and covers situations in which the agent loses track of what has been done or in which a partial sequence leaves the world in an unintended condition. The state category and the mapping to correction and consistency mechanisms are additions of this review; the locations themselves are an adaptation of the loop in Fig. 1, and other taxonomies organise agent failures differently, for example by agent component (Zhu et al., 2025).

Table 2
Taxonomy of Tool-Use Failures by Location in the Feedback Loop

Location	Description	Example	Typical visibility
Selection	Wrong tool chosen or required tool not used	Calling a search tool to do arithmetic	Often silent
Input	Malformed or unsuitable arguments	Invalid syntax in a call, as in Fig. 1(b)	Usually loud
Execution	Tool fails or misbehaves internally	Timeout, permission error, changed interface	Usually loud, sometimes silent
Interpretation	Result misread or wrong result trusted	Accepting a wrong product, as in Fig. 1(c)	Silent
State	Agent record and world disagree	Repeating an action that already took effect	Silent until later

5.1 Loud and Silent Failures

The most useful single distinction for recovery is between loud and silent failures (Sun et al., 2024). A loud failure produces an explicit signal, such as an exception, an error code or a rejected request. Panel (b) of Fig. 1 shows an example in which the model writes a multiplication using the wrong symbol, the interpreter reports invalid syntax, and the model, seeing the message, repairs its input. Loud failures are comparatively tractable, because the signal can trigger a recovery routine and the error text often tells the model what to change. A silent failure produces no signal at all. Panel (c) shows a call that is well formed, runs without complaint and returns a value that is wrong; the only evidence of a problem is that the value does not agree with what a careful reader would expect, in the figure the implausible final digit. Panel (d) shows a silent failure in a pipeline: an object detector labels a tomato as an apple, so the planner is given a faithful plan for a world that does not exist, and the resulting action will fail. In this case the model has some internal evidence of the problem, since it notices that the object looks more like a tomato, but the pipeline provides no route by which that doubt can be turned into a correction before the action is attempted.

The reviewed evidence is consistent with the view that silent failures are harder for agents. In the fault-injection study of Kraishan (2026), agents flagged a problem in 91.3% of trials with an explicit error but 58.8% with a plausible wrong value, and ToolMaze reports the sharpest recovery drops under implicit semantic failures (Zhu et al., 2026). Three further arguments suggest why silent failures deserve priority, although they are arguments and not established findings of the reviewed corpus. They may become relatively more common as developers remove the causes of loud failures first. They can be more damaging, because a wrong result that looks right is used by every later step; the production case study of Wu (2026) found that most silent failures were caught by human observation. And they are harder to study, because there is no natural label: determining that a result is wrong requires an independent source of truth. Much of the literature on detection can be read as a search for substitutes for such a source.

6. DETECTING FAILURES

Detection mechanisms can be grouped by the source of the signal they use. Table 3 compares five families, and the following subsections discuss each. The evidence reviewed suggests that no family is sufficient alone, and robust systems are likely to combine several, using cheap signals first and costly ones only where the risk justifies them.

Table 3
Families of Failure Detection in Tool-Using Agents

Family	Signal used	Strengths	Weaknesses
Execution signals	Exceptions, error codes, status, timeouts	Cheap and reliable for loud failures	Blind to silent failures
Output verification	Schemas, type and range checks, tests, cross-checks	Can catch many silent failures at low cost	Needs checks written for each tool
Model self-assessment	Model judges its own output or confidence	General and requires no extra tools	Unreliable without external evidence
External critics	Separate model or tool judges the step	Independent view, can use stronger evidence	Added cost, critic can also be wrong
Human oversight	Review, approval or feedback by a person	Highest reliability for high-stakes actions	Slow, costly, does not scale

6.1 Execution Signals

The simplest detector is the signal that the tool itself provides. Interpreters report syntax and runtime errors, services return status codes, and tool wrappers can report timeouts. Agents that place such messages in their context can often repair their input in one or two further attempts, as in panel (b) of Fig. 1, and code-generation agents that execute their programs and read the error output are a commonly reported success for this family (Shinn et al., 2023). The quality of the message appears to be decisive. A bare failure code gives the model little to act on, while a descriptive message that names the offending argument and the expected form is likely to lead to better repairs. Tool designers therefore have an important role: writing error messages with a model as the reader is a form of interface design that deserves the same attention as documentation. The limitation of this family is that it sees only what the tool is willing to report.

6.2 Output Verification

Output verification checks the returned result against independent expectations. Schema validation can reject results of the wrong form, range and sanity checks can reject impossible values, and plausibility rules can flag results that conflict with known constraints. For code, running a test suite is a powerful verifier, and benchmarks built on test execution have shown how often plausible code fails tests (Jimenez et al., 2024). For numerical results, a second computation by a different route, or a simple approximate estimate, can reveal a result that is out of range, which is the check implied by the question in panel (c) of Fig. 1. For perception pipelines, cross-checking an object label against a second detector or against the properties of the object, such as colour and shape, is an analogous check. For agents that change external state, verification includes querying the system after an uncertain call, for example looking up a payment by its request identifier before deciding whether it happened, and checking that the final state matches the intended state. Verification is a promising route against silent failure, but it requires someone to specify the checks, and the checks need to be tool-specific.

6.3 Model Self-Assessment

Self-assessment asks the model to judge whether its own step went well, either by explicit critique or by expressing confidence. It is attractive because it needs nothing beyond the model, and in cases where the model can spot an inconsistency, as in the comment that an object looks more like a tomato in panel (d), it can work well. The broader evidence is mixed. Studies that remove external feedback report that models often cannot reliably identify their own errors in reasoning, and that critique without grounding may lead them to change correct answers into incorrect ones (Huang et al., 2024; Kamoi et al., 2024). In the setting of tool errors, simple in-context prompts that encourage the model to doubt the tool have been reported to improve detection (Sun et al., 2024). The practical conclusion is that self-assessment is best used as a trigger for further checking, not as a final verdict: a signal of doubt should prompt the agent to consult a tool, not to guess again.

6.4 External Critics

External critics are separate components that evaluate a step. The critic may be another model, possibly with a different prompt or training, a rule-based checker, or a tool that can gather evidence, such as a search engine used to verify a factual claim or an interpreter used to test a piece of code (Gou et al., 2024). Interactive critiquing with tools has been reported to be more reliable than critiquing from memory because the critic has access to evidence that the original model lacked. Multiple critics, or the comparison of several sampled answers, can reveal disagreement that signals a likely error (X. Wang et al., 2023). The disadvantages are cost and the fact that a critic can share the blind spots of the model it checks, especially when it is the same model. Independence, whether of model, prompt or evidence, is the property that makes a critic worth having.

6.5 Human Oversight

For actions with serious or irreversible consequences, human review remains the most dependable detector. Agents can be designed to request approval before taking defined classes of action, to show the user a summary of what they intend to do, or to ask a clarifying question when the task is ambiguous. Ruan et al. (2024) treat under-specified instructions as one of the risk factors for harmful agent actions in their emulated-sandbox evaluation, which supports asking when instructions are ambiguous. The challenge is to ask neither too often, which makes the agent useless, nor too rarely, which defeats the purpose.

7. CORRECTING FAILURES

Once a failure has been detected, the agent must decide what to do. The literature describes six main responses, which are arranged in Table 4 roughly from the cheapest and most local to the most drastic. A well-designed agent chooses among them according to the kind of failure, its cost and the information available, and escalates only when simpler responses do not work. The first two responses are easily confused and are therefore defined strictly: a retry sends the same request again, while a repair sends a different one.

Table 4
Families of Failure Correction and Their Typical Use

Response	What the agent does	Appropriate when	Main risk
Retry	Re-issues the identical request (same tool, same arguments), usually after a delay	Transient execution faults, and unknown outcomes once state is verified or the call is idempotent	Repeating a non-idempotent action; futile after deterministic errors
Repair	Changes the arguments or the call in light of the error message, then sends the changed request	Input failures with an informative error	Overfitting to the message; loops of near-identical edits
Reflection	Writes an analysis of the failure for later attempts	Failures with informative feedback	Wrong diagnosis reinforced in memory
Replanning	Changes the plan or the tool	Selection and interpretation failures	Discarding useful progress
Rollback	Undoes earlier actions or restores a checkpoint	Partial sequences that left bad state	Undo may be impossible or incomplete
Escalation	Asks a person or a stronger system for help	High risk or repeated failure	Cost and delay

7.1 Retry

A retry re-issues the same request unchanged. It is appropriate when the fault lies in the environment and not in the request, as with a timeout, a rate limit or a temporary overload, and the perturbation taxonomy of ToolMaze likewise pairs transient faults with retrying (Zhu et al., 2026). Good practice, adopted from resilient service design, uses a limited number of attempts with increasing delays and a circuit-breaker rule that stops calling a service which keeps failing (Nygard, 2007). Because the request is unchanged, a retry is only safe when the action is idempotent or when the agent has first verified that the earlier attempt did not take effect (Section 8.1). A retry is futile when the original request was deterministically wrong: re-sending a malformed query cannot succeed. Observed agent behaviour shows how this goes wrong in practice, since agents have been reported to return to the same tool three or more times in a row after a fault in up to 22.2% of trials (Kraishan, 2026).

7.2 Repair

A repair changes the request. It applies where the input was at fault: the agent reads the error message, changes the arguments or the call and sends the modified request, as in the correction of the syntax in panel

(b) of Fig. 1. Repair is not a form of retry, and the two should not be mixed. Repairing in response to a timeout is usually wrong, because the input was not at fault and the changed request may create a second, different side effect. Conversely, an unchanged request after an invalid-input error repeats the error. Reports on code-generation agents suggest that a few repair rounds can resolve many simple failures (Shinn et al., 2023), but returns diminish when rounds bring no new information, and repeated rounds can turn into loops. Each repair attempt should therefore differ from earlier ones and should be grounded in new evidence such as the error text, documentation or a verification query. A repaired request that changes an external effect should also carry a new idempotency key, whereas a retry keeps the original key.

7.3 Reflection and Memory of Failures

Reflective agents convert a failed attempt into a written lesson and carry it into the next attempt, so that learning takes place without changing the model's parameters (Shinn et al., 2023). This has produced gains when the feedback was informative, as with test outcomes or environment signals, because the lesson is then grounded in evidence. The method has a particular weakness. A wrong diagnosis, once written to memory, can be retrieved and trusted later, so that a mistaken lesson spreads across tasks. This is a consistency problem in the agent's own memory and is discussed further in Section 8. Mitigations include storing the evidence with the lesson, marking lessons with a confidence level and expiry, and testing lessons before relying on them. Related work on debugging localises the root-cause step in a failed trajectory and returns targeted feedback, rather than a free-form lesson (Zhu et al., 2025).

7.4 Replanning and Search

When the fault lies in the choice of tool or in the interpretation of a result, repairing the input is not enough, and the agent must change its approach. Planning methods that maintain several alternatives and evaluate them allow the agent to abandon a failing branch and return to an earlier decision point (Yao et al., 2023a; Zhou et al., 2024). In embodied settings, the agent can use feedback from the environment, such as the report that an object could not be found, to revise its plan (Huang et al., 2022). In the example of panel (d), a replan would begin by questioning the detector output and re-observing the scene before attempting any pickup. ToolMaze evaluates this ability directly by providing alternative tool paths and measuring both whether and how efficiently agents switch to them (Zhu et al., 2026). The principal costs of replanning are that it may throw away work that was correct and that the agent may keep returning to a plan that it should have abandoned.

7.5 Rollback and Escalation

Rollback restores an earlier condition, either by undoing actions one by one or by returning to a saved checkpoint. In software settings, rollback is natural for actions on files and for work done in sandboxes or on branches, where changes can be discarded. It is much harder for actions with external effects such as sent messages or financial transactions, which may require compensating steps instead of true reversal. Escalation passes the problem to a person or to a more capable component. It is the appropriate response when the risk is high, when repeated attempts have failed or when the agent cannot determine the state of the world.

8. STATE CONSISTENCY ACROSS FAILURES

Detection and correction attract most of the attention in the literature reviewed here. State consistency, which is the requirement that the agent's beliefs, its memory and the real condition of the world stay in agreement

through failures and recoveries, appears to be less studied, yet it is where some of the most damaging failures can arise. Three different states are involved: the world state, which includes external systems and any data that tools modify; the agent state, which is what the agent believes it has done, including its plan and working notes; and the memory state, which is what persists between sessions. Failures create divergence between these. A call that times out leaves the world in an unknown state, since the action may or may not have happened. A rollback that succeeds in the world but is not recorded leaves the agent believing that the change is still in place. A reflection stored in memory records a belief that later turns out to be false. Figure 2 illustrates the first case with an original example of a booking agent, contrasting a blind retry with a recovery-aware sequence. Table 5 lists the main mechanisms that the literature and classical practice offer.

Table 5
Mechanisms for Maintaining State Consistency

Mechanism	Principle	Application to tool-using agents
Idempotent operations	Repeating a call has the same effect as making it once	Use request identifiers so that a retried action is not performed twice
Compensating actions	Each step has a defined undo step (Garcia-Molina & Salem, 1987)	Register an undo for every action that changes external state
Checkpointing	Save a known good state to return to	Snapshot files, data and agent plan before risky sequences
State verification	Check the world instead of assuming	Query the system to confirm what happened after an uncertain call
Action logging	Keep a durable record of intended and completed actions	Write intent before acting and outcome after, to support recovery
Memory hygiene	Control what is written, trusted and expired	Store evidence and confidence with lessons, and validate before reuse

8.1 Idempotency and Uncertain Outcomes

The hardest case is the uncertain outcome: the agent sent a request and received no response, so it does not know whether the request took effect. Retrying blindly risks doing the action twice, while not retrying risks leaving it undone. The standard remedy from distributed systems is to make operations idempotent, usually by attaching an identifier so that the receiving service can recognise and ignore a repeat (Nygard, 2007). The identifier makes a retry safe precisely because the retried request is identical to the first. Where tools were not designed that way, the agent should first verify the state, for example by querying whether the record exists, before deciding to retry, as in Fig. 2(b). Recent work on agent runtimes that adds transactional structure shows the same logic at the workflow level (Chang & Geng, 2025).

8.2 Compensation and Checkpoints

For sequences of actions that must succeed or fail as a whole, the database literature on long-running transactions introduced the idea of compensation, in which each step is paired with an action that semantically undoes it, and in case of failure the compensations of the completed steps are executed in reverse order (Garcia-Molina & Salem, 1987). Applied to agents, this implies that tools should be registered together with their undo operations, and that the agent should keep a log of completed steps. SagaLLM is an example that

applies this pattern to LLM planning, with persistent memory, automated compensation and independent validation (Chang & Geng, 2025). Checkpointing is a complementary approach that is practical when the environment can be snapshotted, as in code sandboxes and simulations. In the booking sequence of Fig. 2, a failure of the final confirmation step after the card has been charged calls for compensation in reverse order, refunding the charge and releasing the seat, whereas in a sandboxed coding task saving the files and the plan before each stage allows the agent to return to the last stage known to be sound.

8.3 Agent State and Memory Hygiene

The agent also needs a faithful record of what it has done. In long tasks the context window may be truncated or summarised, and details of earlier failures and recoveries may be lost, so that the agent repeats mistakes or acts on obsolete assumptions. Studies of how models use long contexts report that information in the middle of a long input is used less reliably than information at its beginning or end (N. F. Liu et al., 2024), which makes it risky to rely on the context as the only record of state. Systems that manage memory explicitly, writing important facts to structured stores and retrieving them when needed, can help (Packer et al., 2023), but they introduce the problem of stale or incorrect entries. Good memory hygiene involves recording the source and time of each entry, distinguishing observations from inferences, and revalidating before relying on old information that matters.

9. EVALUATION OF RECOVERY

How well do current evaluations measure recovery? Many benchmarks measure final task success and treat recovery only implicitly, although the studies of Section 2.5 show that this is changing. This is a limitation, because two agents with the same success rate may differ greatly in how they behave when something goes wrong, and because an agent that succeeds only when no faults occur is of limited use in real deployments. Several properties of good recovery evaluation can be identified.

First, faults should be injected deliberately and controllably. Evaluations in which tools are made to fail in specified ways, for example by raising errors, returning empty results, returning wrong but plausible values or timing out, allow recovery to be measured directly and make comparisons fair; ToolMaze and the study of Kraishan (2026) are recent examples. Tool emulation, in which a model plays the role of the tool, makes it possible to generate such conditions at scale and to explore risky situations without real consequences (Ruan et al., 2024). Second, evaluations should distinguish loud from silent faults, since methods that cope with the former may fail on the latter. Third, they should measure consistency of outcome and not just success. Repeating each task several times and requiring success every time reveals reliability problems that a single attempt conceals (Yao et al., 2024). Fourth, they should check the final state of the environment, not only the answer given, to detect duplicated or half-completed actions; τ-bench already does this by comparing the final database state with the goal state (Yao et al., 2024), but this practice is not universal. Fifth, they should record the cost of recovery, including the extra calls, delay and human interventions used, because a method that recovers by exhaustive retrying may be worse in practice than one that fails gracefully and asks for help; ToolMaze's recovery cost metric is one example (Zhu et al., 2026).

Table 6
Recommended Measures for Evaluating Failure Recovery

Measure	What it captures	Gap in many existing evaluations
Detection rate by failure type	Share of injected loud and silent faults noticed	Faults are often not injected or labelled
Time to detection	Steps between fault and its discovery	Often only end results are scored
Recovery success	Share of detected faults repaired correctly	Recovery is mixed into task success
Repeat-run consistency	Success in all of several repeated trials	Single-run scoring is common
Final state integrity	No duplicate, missing or contradictory changes	Environment state is inspected in only some benchmarks
Cost of recovery	Extra calls, delay and escalations	Cost is reported separately, if at all

10. DISCUSSION AND OPEN CHALLENGES

10.1 Principal Findings

Four findings emerge from the reviewed corpus, and each should be read with the limits stated in Section 10.4. The first is that explicit error messages are comparatively well exploited: for loud failures, simple repair loops are effective in many reported settings, provided that the error messages are informative and the number of attempts is bounded. The second is that silent failures appear to remain a major open problem. Methods that depend on the model noticing its own mistakes are unreliable without external evidence, and the recent injection studies report markedly lower detection and recovery for plausible wrong values than for explicit errors (Kraishan, 2026; Zhu et al., 2026), while methods based on independent verification work but require tool-specific effort. The third is that self-correction depends on external feedback. Where tool outputs, test results or environment signals supply information, agents improve; where they do not, reported gains are fragile (Huang et al., 2024; Kamoi et al., 2024). The fourth is that state consistency receives less attention than detection and correction. Among the nine studies compared in Table 1, one makes state consistency its central concern (Chang & Geng, 2025) and one compares final state with a goal state as part of its evaluation (Yao et al., 2024). Whether duplicated actions, incomplete sequences and corrupted memory account for a large share of the costliest failures in practice is plausible but has not, to the authors' knowledge, been measured systematically.

10.2 Design Recommendations

The review suggests a number of practical recommendations for builders of tool-using agents. They are drawn from the reviewed studies and from established dependability practice, and have not been validated as a package on agents. Tools should return structured, descriptive errors written for a model reader, and should state clearly in their documentation what they do when called twice. Every tool that changes external state should be paired with a verification query and, where possible, an undo operation. Agents should log intent before acting and outcome afterwards, so that an interrupted sequence can be examined and resumed.

Verification should be placed as close as possible to the step it checks, since early detection avoids the cost of building on a wrong result, as in the booking sequence of Fig. 2. Retries should be bounded, use increasing delays and re-send an unchanged request with the original idempotency key, and only when the fault is transient or the outcome has been verified. Repairs should be bounded too, and each repair should differ from the previous attempts and rest on new evidence. Actions should be classified by reversibility, with approval required for those that are irreversible. Lessons stored in memory should carry their evidence and an expiry. Finally, agents should be designed to escalate clearly when they cannot determine the state of the world, and to explain what they know and what they do not.

10.3 Open Research Problems

Several problems deserve research attention. The first is the detection of silent failures without a ground truth, perhaps through ensembles of independent checks, learned plausibility models or systematic use of redundancy. The second is the formal treatment of consistency for agents: specifying what guarantees an agent can offer about the state of the world after a partial failure, and building frameworks that enforce them; SagaLLM is a step in this direction but explicitly relaxes strict transactional guarantees (Chang & Geng, 2025). The third is the learning of recovery policies, in which agents are trained on data containing faults and the correct responses to them, rather than only on successful trajectories; if training data consist mostly of successes, models may seldom see what recovery looks like. The fourth is the calibration of escalation, so that agents ask for help when it is needed and not otherwise. The fifth is security, since failure-recovery channels can be exploited: an adversary who can influence tool outputs or error messages may steer the agent into harmful repairs, and a recovery routine that trusts error text without scrutiny becomes an attack surface.

10.4 Limitations of This Review

This review has limits that readers should keep in mind. It is a structured synthesis and not an exhaustive registered search, and relevant work may have been missed. The field moves quickly, and results reported for particular models may not carry over to newer ones. Several of the 2025–2026 sources are preprints that may not have completed peer review, and the comparison in Table 1 relies on each study’s own reporting. Many published results come from simulated or sandboxed environments, and their transfer to production systems with real users, real costs and adversaries is not guaranteed. The taxonomy is a conceptual tool and the boundaries between categories are not always sharp, since a failure may be assigned differently depending on whether the focus is on the model or the tool. The recommended measures are proposals and have not been validated empirically.

11. CONCLUSION

This paper reviewed failure recovery in tool-using language-model agents through the three connected themes of detection, correction and state consistency. Starting from the feedback-loop model of Sun et al. (2024), it organised failures by their location at selection, input, execution, interpretation and state, and it stressed the distinction between loud failures, which announce themselves, and silent failures, which do not. It organised detection into execution signals, output verification, self-assessment, external critics and human oversight, and correction into retry, repair, reflection, replanning, rollback and escalation, separating the unchanged retry from the changed repair. It showed how classical techniques of idempotency, compensation, checkpointing and

verified state tracking apply to agents, and it compared nine representative studies, including recent 2025–2026 work, side by side.

The review suggests that reliable agents are built from the combination of a capable model and a well-engineered surrounding system. Models can repair what they can see to be broken, but the reviewed evidence indicates that they cannot reliably see what is hidden, and they cannot guarantee the consistency of a world they only partly observe. Progress will therefore depend on tools that report faithfully, on verification placed where it matters, on explicit management of state, and on evaluations that reward graceful recovery as much as flawless first attempts. Future work should concentrate on silent-failure detection, formal consistency guarantees, recovery-aware training and shared fault-injection benchmarks, so that agents are judged not only by what they achieve when everything works, but by what they do when it does not.

Declarations

Funding

The authors received no funding for this work.

Author Contribution

R.S.T. led the literature review and drafted the manuscript. V.M. contributed to the analysis of the reviewed studies and critically revised the manuscript. Both authors reviewed and approved the final manuscript.

References

1. Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., et al. (2022). Do as I can, not as I say: Grounding language in robotic affordances. *Conference on Robot Learning*.
2. Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing, 1*(1), 11–33.
3. Barres, V., Dong, H., Ray, S., Si, X., & Narasimhan, K. (2025). τ²-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint* arXiv:2506.07982.
4. Chang, E. Y., & Geng, L. (2025). SagaLLM: Context management, validation, and transaction guarantees for multi-agent LLM planning. *Proceedings of the VLDB Endowment, 18*(12).
5. Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., & Neubig, G. (2023). PAL: Program-aided language models. *Proceedings of the International Conference on Machine Learning*, 10764–10799.
6. Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 249–259.
7. Gou, Z., Shao, Z., Gong, Y., Shen, Y., Yang, Y., Duan, N., & Chen, W. (2024). CRITIC: Large language models can self-correct with tool-interactive critiquing. *International Conference on Learning Representations*.
8. Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2024). Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations*.

9. Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., et al. (2022). Inner monologue: Embodied reasoning through planning with language models. *Conference on Robot Learning*.
10. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., et al. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys, 55*(12), 1–38.
11. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? *International Conference on Learning Representations*.
12. Kamoi, R., Zhang, Y., Zhang, N., Han, J., & Zhang, R. (2024). When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics, 12*, 1417–1440.
13. Kraishan, (2026). Loud failures, quiet failures: Fault detection and recovery in tool-using language model agents. *arXiv preprint* arXiv:2610.10062.
14. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems, 33*, 9459–9474.
15. Li, M., Zhao, Y., Yu, B., Song, F., Li, H., Yu, H., Li, Z., Huang, F., & Li, Y. (2023). API-Bank: A comprehensive benchmark for tool-augmented LLMs. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.
16. Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2024). Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics, 12*, 157–173.
17. Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., et al. (2024). AgentBench: Evaluating LLMs as agents. *International Conference on Learning Representations*.
18. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., et al. (2023). Self-Refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems, 36*.
19. Nygard, M. T. (2007). *Release It! Design and deploy production-ready software*. Pragmatic Bookshelf.
20. Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2023). MemGPT: Towards LLMs as operating systems. *arXiv preprint* arXiv:2310.08560.
21. Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive APIs. *arXiv preprint* arXiv:2305.15334.
22. Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., et al. (2023). Tool learning with foundation models. *arXiv preprint* arXiv:2304.08354.
23. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., et al. (2024). ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *International Conference on Learning Representations*.
24. Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C. J., & Hashimoto, T. (2024). Identifying the risks of LM agents with an LM-emulated sandbox. *International Conference on Learning Representations*.
25. Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems, 36*.

26. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems, 36*.
27. Sun, J., Min, S. Y., Chang, Y., & Bisk, Y. (2024). Tools fail: Detecting silent errors in faulty tools. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing* (pp. 14272–14289). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.emnlp-main.790
28. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. *arXiv preprint* arXiv:2305.16291.
29. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., et al. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science, 18*, 186345.
30. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. *International Conference on Learning Representations*.
31. Wu, W. (2026). When errors become narratives: A longitudinal taxonomy of silent failures in a production LLM agent runtime. *arXiv preprint* arXiv:2606.14589.
32. Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2024). τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint* arXiv:2406.12045.
33. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023a). Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems, 36*.
34. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023b). ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations*.
35. Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., & Wang, Y.-X. (2024). Language agent tree search unifies reasoning, acting, and planning in language models. *Proceedings of the International Conference on Machine Learning*.
36. Zhu, D., Ma, X., Shen, Y., Li, X., Zhao, Y., Wang, S., Yan, L., & Yin, D. (2026). When tools fail: Benchmarking dynamic replanning and anomaly recovery in LLM agents. *arXiv preprint* arXiv:2606.05806.
37. Zhu, K., Liu, Z., Li, B., Tian, M., Yang, Y., Zhang, J., et al. (2025). Where LLM agents fail and how they can learn from failures. *arXiv preprint* arXiv:2509.25370.

Figures

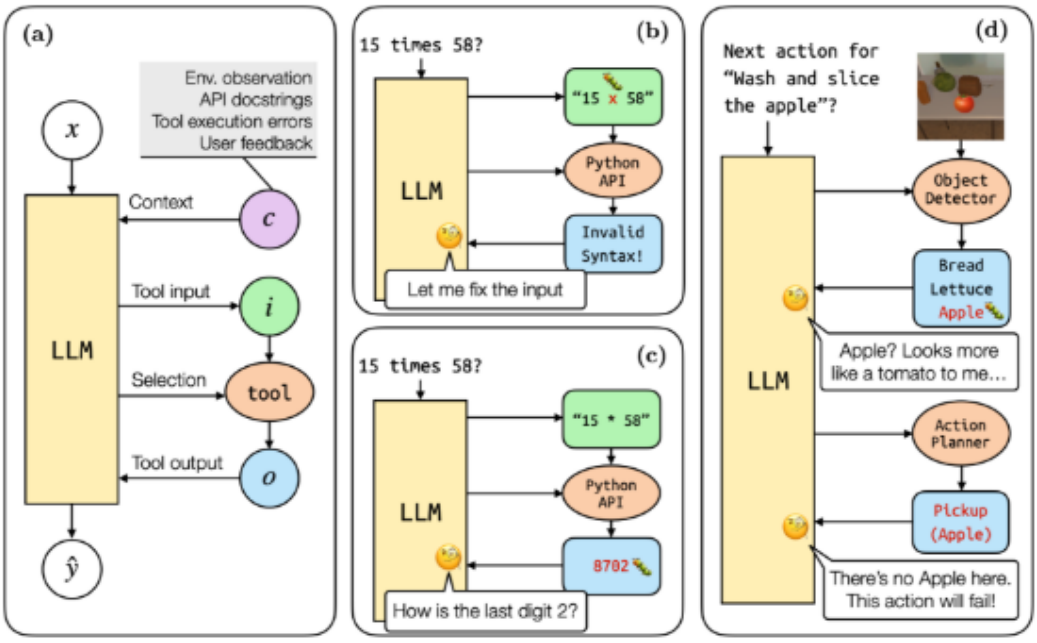


Figure 1

Feedback-loop model of tool use by a language model. (a) The general loop with context, tool input, tool selection and tool output; (b) a loud failure, in which a tool returns an explicit error and the model repairs its input; (c) a silent failure, in which a tool returns a wrong value that the model must question; (d) a failure in a multi-tool embodied pipeline, in which a perception error leads to an action that will fail. Source: reproduced from Sun et al. (2024, Figure 1), published under a CC BY 4.0 licence (ACL Anthology). The figure and the framework it depicts are the work of Sun et al.

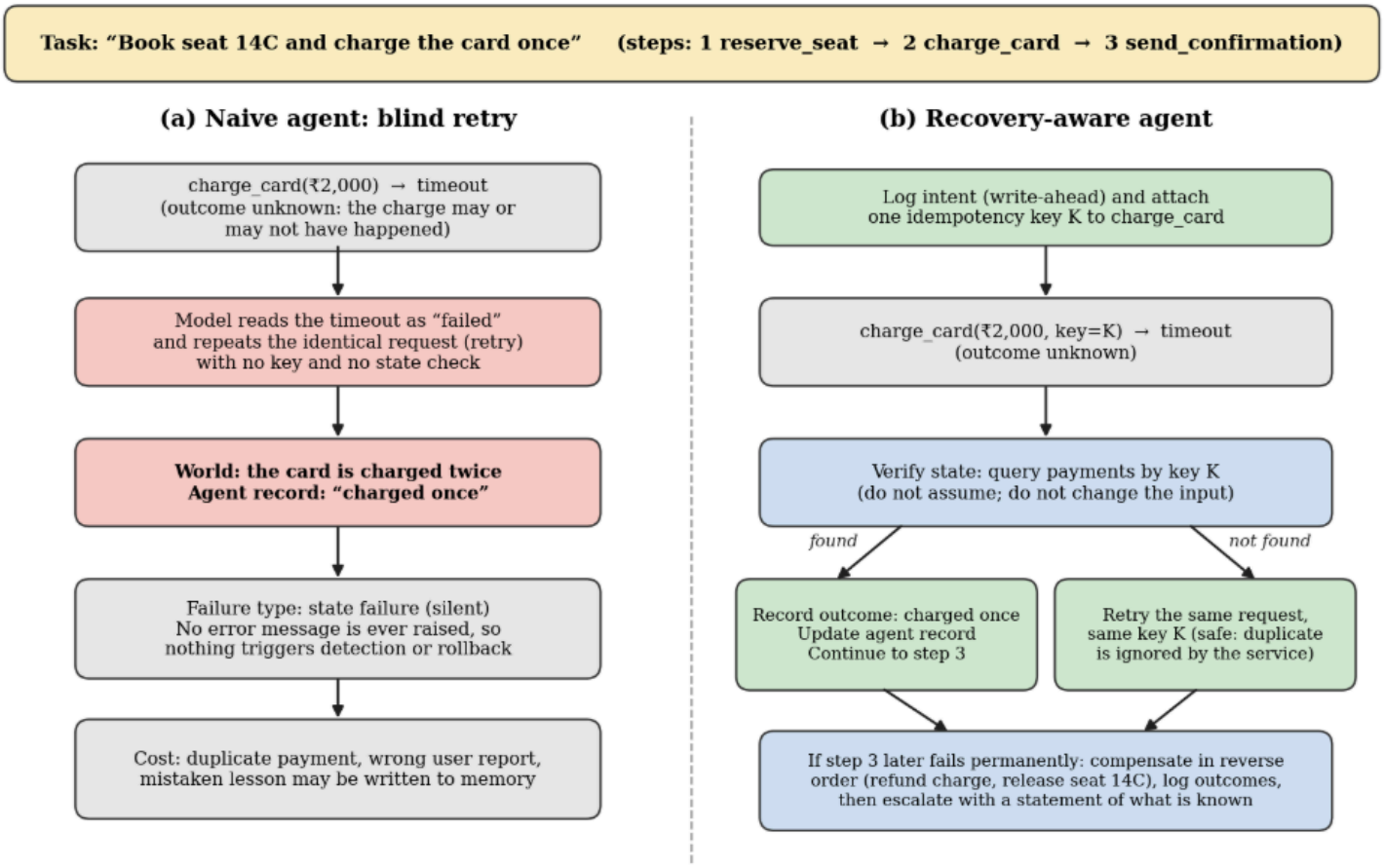


Figure 2

Original illustration of recovery after an uncertain outcome. (a) A naive agent treats a timeout as a failure and blindly repeats a non-idempotent call, so the world and the agent’s record diverge without any error message. (b) A recovery-aware agent logs its intent, attaches an idempotency key, verifies state after the timeout, retries the unchanged request only when needed, and holds a compensation plan for later failures. Prepared by the authors for this review.