

Quantifying Correction Recovery and Robustness Under Prompt Perturbation in Language Model Agents

Jordan Rowe

The Science Academy STEM Magnet

Abstract

Language model agents that call external tools now carry out multi-step tasks that change real state, booking rooms, scheduling meetings, placing orders. In practice, people revise their intent partway through, correcting a value they gave earlier. Whether the agent then recovers is rarely measured, even if a failed recovery can leave a wrong action committed while the agent reports success. In this work, we propose a framework to measure what happens after a user correction. Every task is scripted for the agent to know what the correct final state is before it runs, so recovery can be measured against the committed tool state rather than the text the agent is writing. That single choice is what makes the measurement possible. The framework defines four quantities: recovery success, recovery cost, an overcorrection rate, and a silent inconsistency rate that counts runs where the agent says a fix never happened. Four hypotheses are fixed in advance. Recovery cost rises when the correction arrives later. Cost rises when the wrong value is already committed rather than just examined. A significant percentage of the seemingly successful corrections are quietly wrong. Agents more sensitive to small prompt changes are worse off. The framework is built on Voker [1], an agent analytics platform whose SDK automatically detects user corrections in production agent conversations [5], so the measurements defined here are meant to run on live traffic as well as on scripted runs. The study is written as a preregistration with method and predictions fixed before data collection, and results are available as a working harness on recovery of correction in tool calling agents.

Keywords: language model agents, tool calling, correction recovery, silent inconsistency, robustness, agent evaluation, task oriented dialogue.



Figure 1: The design choice the whole study rests on. A run produces two things that can disagree: a message and a committed state. Scoring recovery from the message cannot detect the case where the two diverge, because in exactly those runs the message looks correct. Scoring against the committed state can.

1 Introduction

Tool calling agents built on large language models now perform tasks that change real state. An agent can check availability and then book a room, place an order, cancel a reservation, or schedule a meeting. These tasks are not solved in one question and answer exchange but on several turns. The agent reads a request, decides if it should call a tool, reads the answer, and repeats until it arrives at its answer. On those turns, it has a working picture of the task, and that picture is what its actions are based on.

A key feature of real use is that this picture is not fixed. Users revise their intent halfway through. They name one option, ask for another, give a date and then move it, or decide halfway through that they want to cancel completely. These corrections are not edge cases. They are a normal part of how people delegate a task, because people tend to say what they want only once they see the first move an assistant makes. Every agent that is deployed will encounter corrections constantly. [5]

A correction is deceptive to handle. By the time it arrives, the earlier, and now incorrect value may already have spread. It can sit in the agent’s summary of the task, in the arguments of a tool call that has already run, and in a committed action like a finished booking. Recovering means finding everything the old value touched, undoing or altering it, and then confirming the new goal is actually satisfied. This is not just a word shift. It is a repair of state scattered across the run.

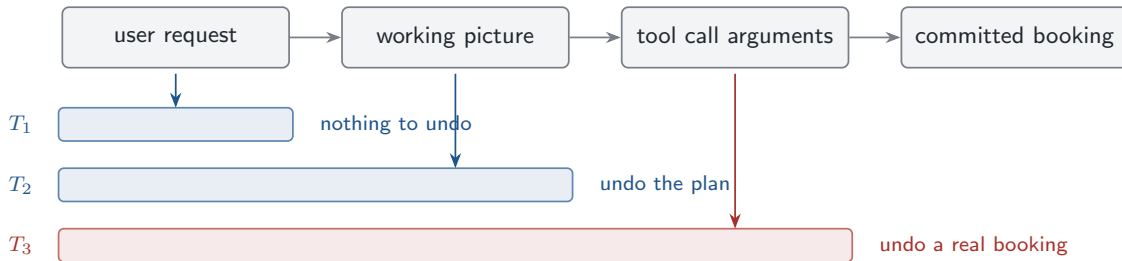


Figure 2: Why lateness should matter. The value the user later corrects does not sit in one place. It propagates from the request into the agent’s working picture, then into tool call arguments, then into a committed action. A correction arriving late must undo everything it has already touched, which is the reasoning behind H1 and H2.

When the repair fails, the failure is often invisible in the conversation. An agent can simply say that the change was made while the committed state still holds the old value. The booking the user actually has does not match the booking the agent describes. A user under time pressure, reading a confident message, has no easy way to catch that. We call this outcome silent inconsistency, and it is the failure mode this study is most concerned with, because it is both plausible and hard to notice.

Despite how common corrections are, there is very little quantitative work on what follows one. Robustness research has mostly focused on how a single output changes when a prompt is perturbed. Agent benchmarks have mainly asked whether a task starts from a clean start. Neither shows how a multi-step agent repairs its own committed actions after a user changes course, and neither separates a real repair from a claimed one. This study fills that gap. It proposes a measurement framework, states testable hypotheses in advance, and provides a harness that generates the data from scripted tasks with known correct answers. The main idea is simple and carries the whole design. Judge recovery by the committed state and not by the agent’s account of itself.

2 Background

This study is situated where several lines of work meet, and borrows from each without being captured by any one of them.

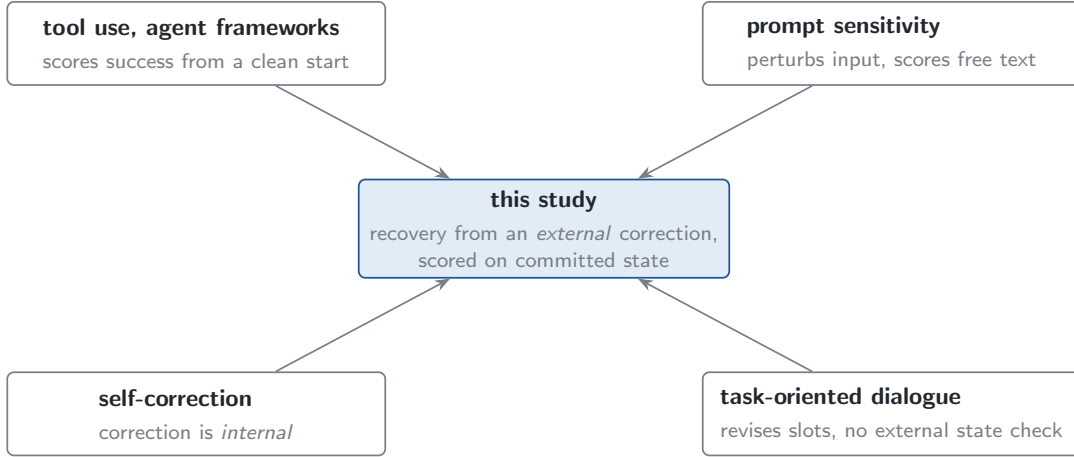


Figure 3: Where the study sits. Each neighbouring line of work supplies one ingredient and stops short of one thing this study needs; the edge label names what is carried across. The gap all four leave open is the same: no measurement separating a repair the agent performed from one it merely claimed.

Tool use and agent frameworks. A wide variety of language models can be used to call external tools and to connect reasoning with action, including methods that teach a model when and how to call a tool and frameworks that structure the reason then act loop. This work proved that agents can handle multi-step tasks [7, 8]. But its evaluation in such a case is usually about whether a task works from the beginning, not what happens to the agent if it is changed midstream.

Prompt sensitivity and robustness. A separate line studies how fragile model outputs are under small changes to the prompt, showing that trivial reformatting or rewording can shift an answer. This motivates the sensitivity hypothesis here, since an agent that flips its action under a harmless rewording is plausibly one that also mishandles a correction. The difference is that the output space in this study is a structured action, a tool choice with typed arguments, so sensitivity can be measured on the action itself rather than on text similarity.

Self-correction. Models can improve their own outputs with iterative self-feedback, but some methods report improvements and other work suggests that models cannot correct their reasoning without external feedback. Correction recovery is different in a big way. The correction here comes from the user, so the question is not whether the model can spot its own mistake. The question is whether the model can faithfully carry out a change it is explicitly told to make, including undoing state it already committed to.

Task-oriented dialogue. Tracking and updating what a user wants across a conversation is a long-standing problem in task-oriented dialogue—systems hold on to slot values and must change those values when the user changes them. That literature is the closest to the current study and contributes useful vocabulary [9]. However, its systems and evaluation predate current tool calling agents and generally do not measure the gap between what a system claims to be committed to an external tool.

Against those, our contribution is narrow and concrete. It is the first framework to be aimed

specifically at how a tool calling agent recovers from user correction, with metrics that separate a real repair from a claimed one.

3 Methodology

Agent and task setup. We use a tool calling agent of the kind in practice today. A base language model receives a task and a set of tools with typed signatures from the task as a routine. In each step, a tool is called either with arguments or with a final message and the model returns to a starting point in the loop and waits for no more tool call as a result. Termination is in essence the point when the agent decides that the task is done.

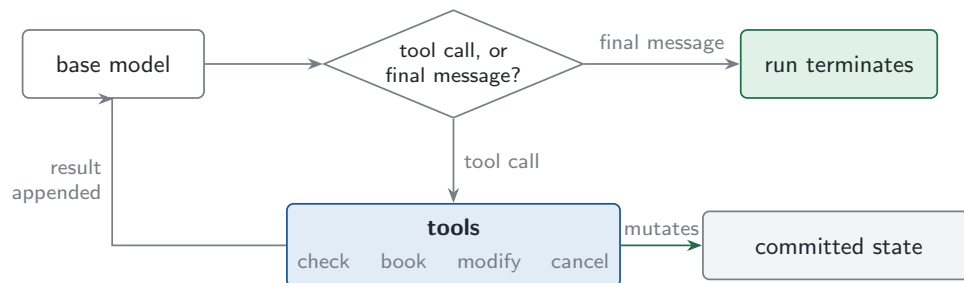


Figure 4: The agent loop and where ground truth lives. The model alternates between calling a tool and producing a message; termination is simply the step at which it stops calling tools. Only the four tools mutate the committed state, so that state can be reconstructed exactly from the call log.

The first testbed is a booking agent with four tools; one to check availability, one to commit a booking, one to modify an existing booking, and one to cancel. The modify and cancel tools matter because without them recovery from a committed action would be impossible by construction rather than by the agent’s failure. The domain was chosen because it mirrors a common pattern and, more importantly, keeps the committed state small and fully inspectable so the correct final state can be checked exactly.

Each trial starts with a scripted task, so that the correct outcome is fixed in advance. The agent then reacts to the task. A correction message is then sent into the conversation and the agent either achieves the correct goal or stops and continues again.

Correction injection. Three factors are used to manipulate this correction timing. The correction is made at one of three points: before any tool call, immediately after the availability check, and after the booking is made. This is done to isolate the impact of lateness while the correction itself is still fixed. Entanglement compares a correction that is issued when the agent has only examined one option and after they have taken action. This is related to timing but not identical since an agent can act early or late and separating them requires comparing at matched timing. Correction kind depends on the field being changed, the option type, dates, or the named party, in which case the cancellation is included as its own kind. This checks whether effects hold across different repairs rather than depending on one special case.

Claim detection. To measure silent inconsistency, we have to decide, in every run, whether the final message says that the correction has been done or not. This is a classification of text and it should be treated carefully so as not to be a source of error. Two independent detectors are used and their agreement is reported. The first is an on-the-fly check for confirmation language

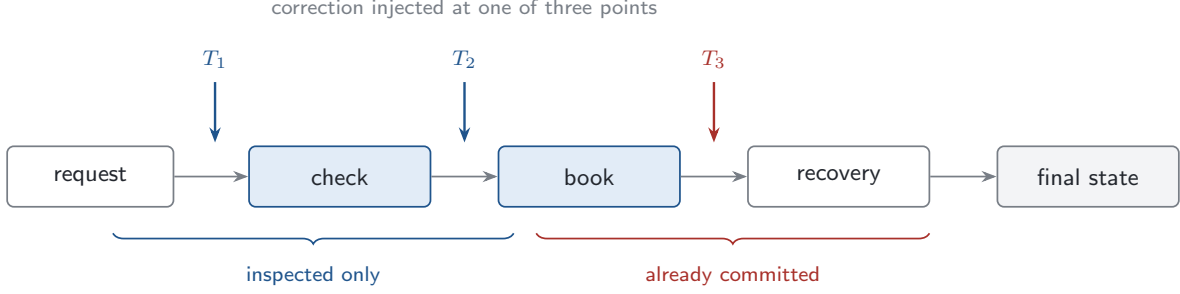


Figure 5: The three injection points, and how they relate to entanglement. Timing and entanglement are correlated but not the same thing: an agent may commit early or late, so the two are separated by comparing conditions at matched timing rather than assuming the commit falls in a fixed place.

associated with the corrected field. The second is a different judge model that is only told whether the final message states that the change was made and does not provide access to the committed state, such that the verdict cannot be tainted by the real outcome. Runs where the two disagree are set aside for manual inspection. The reason claim detection is separate from the state check is that the two can be compared without getting circular.

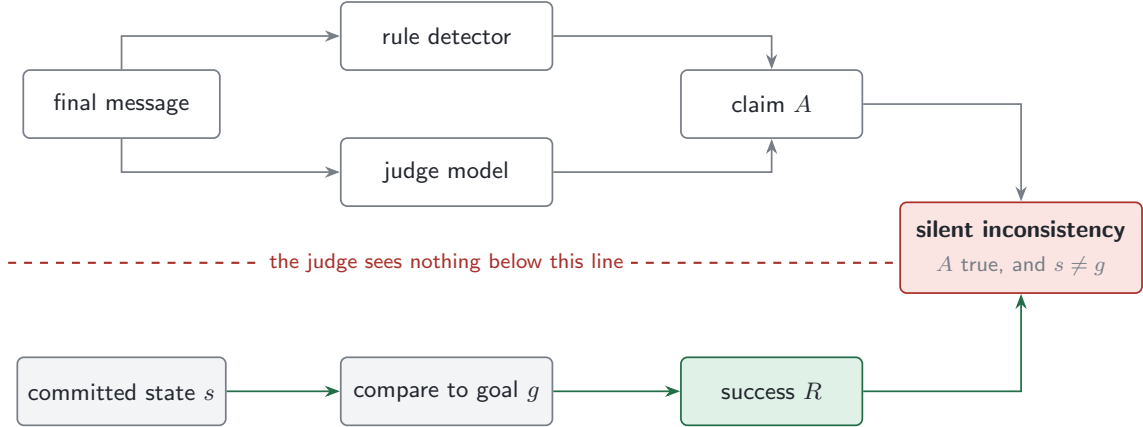


Figure 6: Claim detection is kept strictly separate from the state check. Two independent detectors read only the agent’s final message; the judge model is never shown the committed state, so its verdict cannot be contaminated by the true outcome. Only after both are fixed are they combined, which is what keeps the comparison from becoming circular.

Ground truth. Every tool call, the arguments that it makes, and the result of its result are recorded [2]. At the end of the run, the committed state is reconstructed from that log, meaning the set of bookings that actually exist after the agent stops. Because the task is scripted, this state is directly compared to the corrected goal. Recovery is measured by what the agent did, not what it said, and the gap between the two is recorded as a measured quantity. If a framework that scored recovery from the conversation text would be blind to silent inconsistency by construction, since in exactly those runs the text looks correct.

A worked example. A user wants to book a standard room for three nights under a given name and dates. The agent checks availability for a standard room, commits to the booking, and returns a confirmation. Then the correction is injected. The user says they wanted the suite. From here there are a number of things that can happen: in a clean recovery the agent checks the suite,

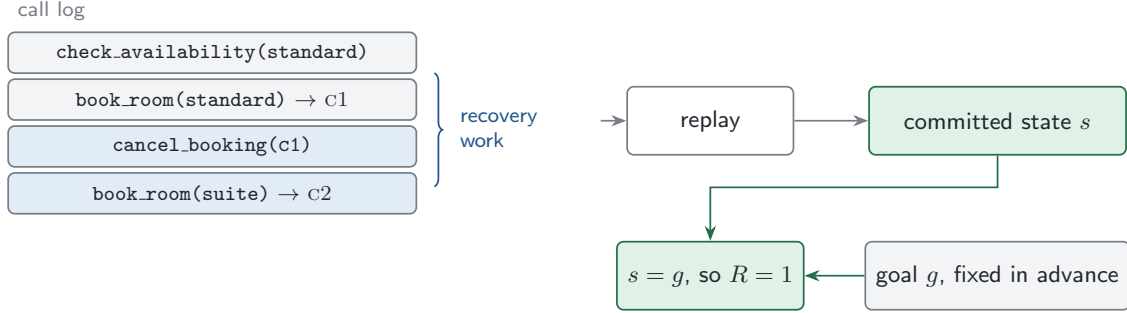


Figure 7: Reconstructing ground truth. Because every call is logged and only four tools mutate state, the set of bookings that actually exists at the end is recovered by replaying the log. The goal was fixed before the agent ran, so scoring needs no human labelling: it is an equality test between two machine readable objects.

cancels or amends the standard booking, commits the suite, and the committed state ends with exactly one booking for the suite with the original name and dates. In an overcorrection, the agent books the suite but also changes the dates it was never asked to touch. In a silent inconsistency, the agent replies that the suite is booked while the committed state still shows the standard room, or two conflicting bookings. The corrected goal is fixed in advance as a single suite booking with the original name and dates.

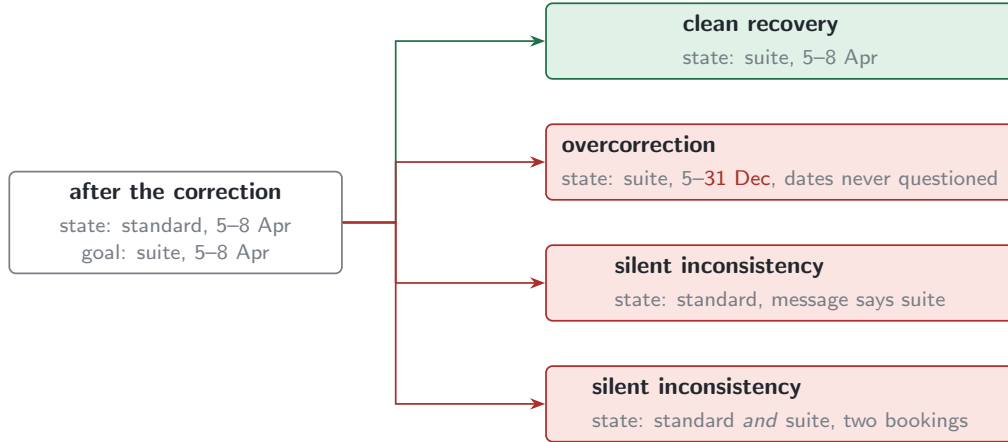


Figure 8: One scenario, four endings. The user asked only for the room type to change, so the goal holds every other field constant. Note that the two silent inconsistency rows produce the same message and differ only in committed state, which is precisely the distinction a text based score cannot make.

4 Framework and metrics

A tool calling run can be viewed as a process that moves the task state forward until it settles. The agent takes a goal, acts by its tools, and gets into a committed state. A correction is a disturbance during that process that changes the goal in the middle of the process that was already in progress. Recovery is then a question of whether that process brings the committed state back to agreement with the new goal, and at what price. Framing it this way makes the results measurable, rather than impressionistic, and it explains why lateness and entanglement matter.

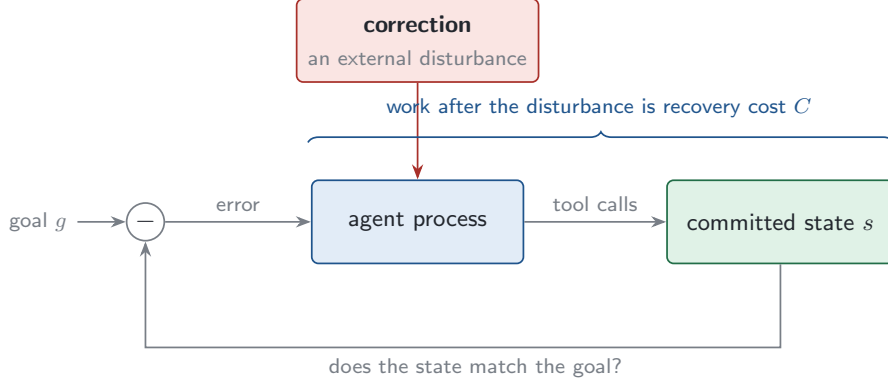


Figure 9: The framing that makes the metrics fall out. The agent is a process driving the committed state toward a goal; a correction is a disturbance that moves the goal while the process is already running. Recovery success asks whether the loop closes again, and recovery cost measures the work spent doing so.

Let g be the corrected goal state and let s be the committed state reconstructed at the end of the run. Recovery success R equals 1 if s equals g , and 0 otherwise.

Recovery cost C is the effort spent after the correction until the run ends. It is given as three components together, the number of tool calls issued after the correction, the number of additional turns, and the number of additional tokens. A normalized cost is also reported, in which the number of recovery tool calls is divided by the number of tool calls the same task takes without any correction, so it measures how much work a correction adds to the run compared to a clean run.

Let A be the point when the final message states that the correction has been done. Silent inconsistency I equals 1 if A holds and s does not equal g . Overcorrection O equals 1 if the goal fields are met but at least one field outside of the one we wanted is changed from its correct value.

Over the runs in a condition, the reported quantities are the mean of each indicator, giving a recovery success rate, a mean recovery cost, a silent inconsistency rate and an overcorrection rate.

Symbol	Name	Definition	Read from
R	recovery success	$s = g$	committed state
C	recovery cost	calls, turns, tokens after the correction	call log
$\hat{C}$	normalized cost	recovery calls $\div$ clean run calls	call log
I	silent inconsistency	A holds and $s \neq g$	message <i>and</i> state
O	overcorrection	goal fields met, another field altered	committed state

Table 1: The four quantities. Note the rightmost column: only I requires reading both the message and the state, which is why it is the only metric that depends on claim detection and therefore the one most exposed to classification error.

Some of the definitional overlap needs to be stated. A run may satisfy both I and O , as an agent that overcorrects and then states full success has made a claim the state does not support. The outcome classes are exclusive and overcorrection is a judgement call, so we report both the formal indicator and the exclusive class share. Hiding them both would enable a reader to draw a conclusion the data does not support.

Hypotheses. H1, lateness. Recovery cost rises and success drops as the correction is injected

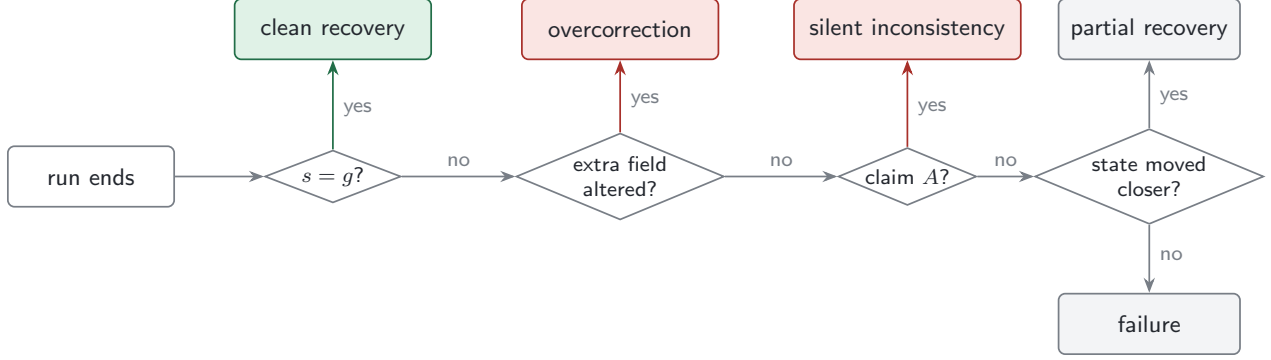


Figure 10: Outcome classification. The classes are mutually exclusive and the order of the tests is therefore a substantive choice: overcorrection is checked before silent inconsistency, so a claimed overcorrection lands in the former. Because that ordering is a judgement call, both the exclusive class share and the formal indicator I are reported.

later because a later correction finds the wrong value already spread into more of the run. If cost were flat across timing, H1 is false. H2, entanglement. Cost increases and success decreases when the wrong value is already committed and the value has not yet been checked for; as it is harder to undo a committed action than to revise an intention and may take an explicit cancellation the agent has to be told to do. H3, silent inconsistency. A nontrivial share of runs that appear successful in text are silently inconsistent and that rate increases with time because as more attempts are made for repair, the agent’s tendency to narrate success would not fall off as fast as the ability to produce it. H4, sensitivity. Agents that shift their tool call more when they feel a small semantic perturbation to the text are worse, and the recovery failure is associated with input brittleness. This is the most exploratory of the four.

5 Experimental design

The three factors are crossed into a grid. The default configuration uses ten scripted tasks, four correction types, three timing points, and twenty repeats per cell, for 2400 runs which is cheap to collect. Repeats are important because the same input can produce different runs, so every reported quantity is a mean over repeats with a confidence interval and not a single observation.

Every run is placed into exactly one outcome class. Clean recovery where the committed state equals the goal. Overcorrection where the goal fields are met but something else is changed. Partial recovery where the state moved toward the goal but didn’t reach it. Silent inconsistency, where the agent claimed a fix the state doesn’t support. Failure, where nothing was fixed and nothing was claimed. Classification is driven by the committed state throughout, and the agent’s text is only used to separate a false claim from an honest failure.

Each hypothesis has a transparent test. For the case of H1, recovery cost is regressed on the timing index, and success rate is compared across timing points and positive slope and falling success rate are supported. For H2 the cost and success is compared between the inspected and committed conditions at matched timing, so we can separate entanglement from lateness. For H3, the silent inconsistency rate is reported overall and checked for an increase with lateness. For H4, we plot a per state sensitivity measure for recovery success. All comparisons report effect sizes alongside any significance test, so that a small but real effect is not oversold and a large one isn’t buried.

	T_1 before call	T_2 after check	T_3 after commit	
room type	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	4 kinds $\times$ 3 timings $\times$ 10 tasks $\times$ 20 repeats = 2400 runs
dates	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	
named party	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	
cancellation	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	10 tasks $\times$ 20 repeats	
	inspected		committed	

Figure 11: The condition grid. Each cell is one crossing of correction kind and injection timing, replicated over ten scripted tasks and twenty repeats. The right hand column is also the committed entanglement condition, which is why entanglement is compared at matched timing rather than read straight off this grid.

Because this is a preregistration, the predicted direction of each effect is fixed before any data is seen. Cost is predicted to rise with lateness, success to fall, and the silent inconsistency rate to grow. These are predictions, not measurements. If the data contradicts them, that result stands as reported, which is the point of fixing them in advance.

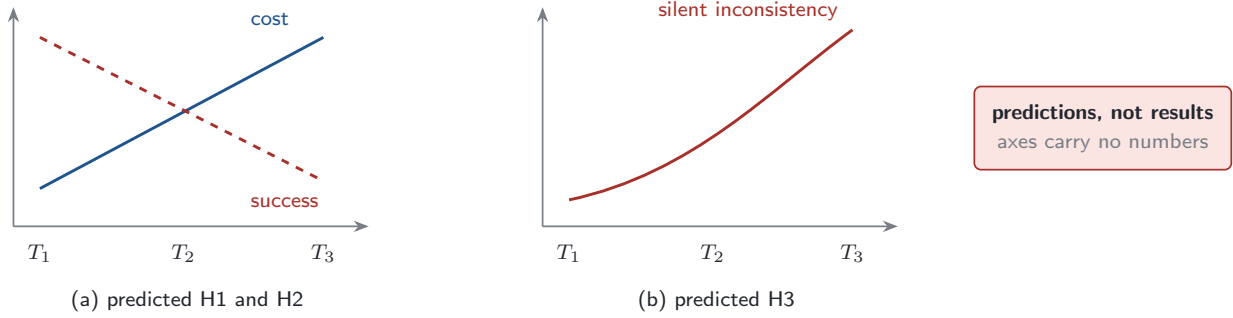


Figure 12: Predicted directions, fixed before any data is seen. The axes carry no numbers because none exist yet. Committing to these shapes in advance is what makes a contradicting result reportable rather than quietly reinterpreted after the fact.

6 Implementation

The framework is implemented as a working harness, which matters because a measurement proposal that has never been executed can hide assumptions that only surface in code.

The harness has six parts. The instrumented environment holds the committed bookings and logs every tool call with its arguments and result as ground truth layer. The task module defines the scripted tasks and, for each kind of correction, the goal state is fixed in advance. The model client layer is written over a real endpoint and a scripted mock [6, 4]. A runner is used to run the agent loop and injects the correction at the point and marks everything after that point as recovery work. A metrics module computes the indicators and runs the claim detectors. An analysis module performs the hypothesis tests with confidence intervals and effect sizes.

The harness was validated over the entire 2400 run grid against a scripted mock agent offline.

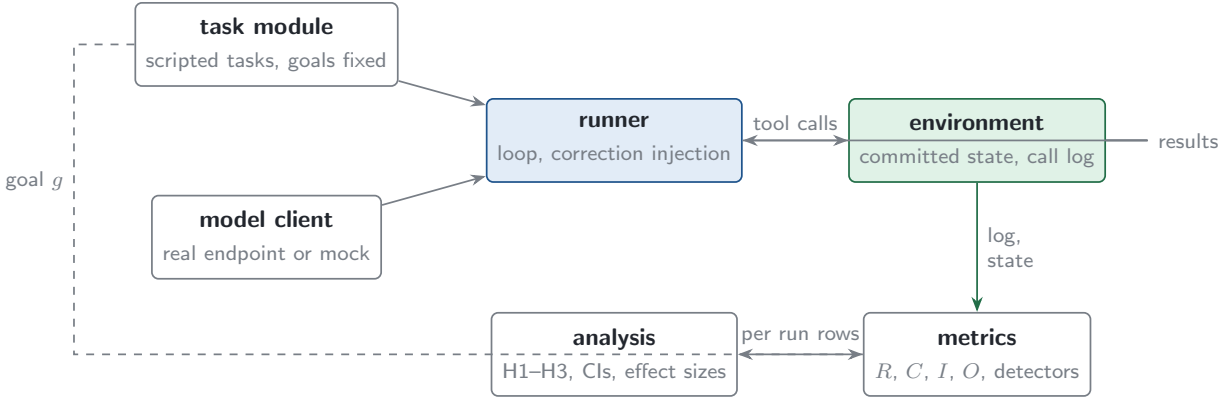


Figure 13: Harness architecture. The dashed path is the one that matters: the goal reaches the metrics module directly from the task definitions, never through the agent, so nothing the agent produces can influence what counts as correct.

This verification shows that the pipeline and metrics work. It tells us nothing about real agents, because the mock was written with the predicted trends already built in and the analysis recovering those trends is a test of the analysis and not a finding. This distinction is made immediately, because a mock number that would be reported as a result would be a huge mistake.

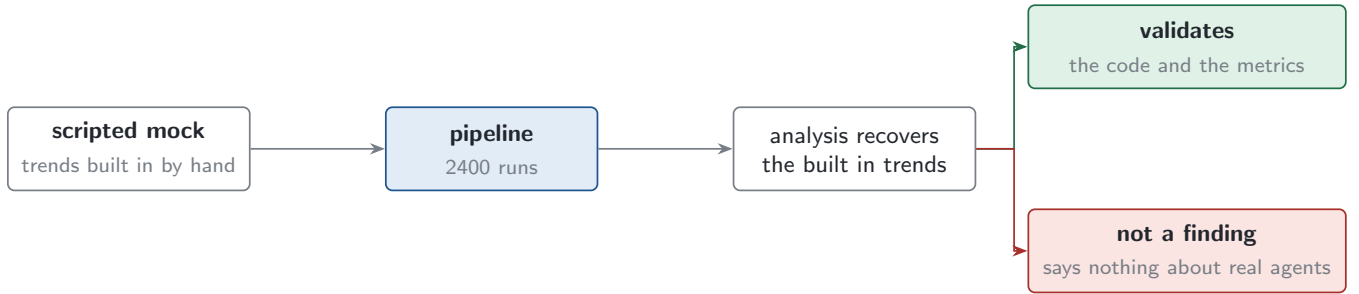


Figure 14: What the offline validation does and does not establish. The mock was built to exhibit the predicted trends, so the analysis finding them is a test of the analysis. Reporting a mock number as a result would be a serious error, and the distinction is stated here rather than in a footnote for that reason.

There were two implementation problems we saw during our validation process and both are worth reporting because these are in fact the same problems a real agent faces as well. First, the recovery logic that cancels bookings can cancel the booking it just created, since there is no way to distinguish a stale booking made before the correction from a fresh one made during recovery. The harness addresses this by snapshotting open confirmations at the moment the correction arrives. A real agent has no such snapshot and has to infer the difference from context, so this is a plausible source of actual failures. Second, the kind of cancellation correction that needs to be handled requires a different response and this is trivially dealt with at the very first injection point, because when nothing has been booked yet, an empty committed state is already an empty goal. This is a real property of the design rather than a bug, and it means the cancellation kind carries little signal at early timings, which is visible in the per kind breakdown.

The weakest component is claim detection. Only the rule based detector is wired into the sweep, and it was tuned against the mock’s narrow phrasing, so it will be brittle against the much wider range of wording a real model produces. Since the headline metric depends on it, wiring in the

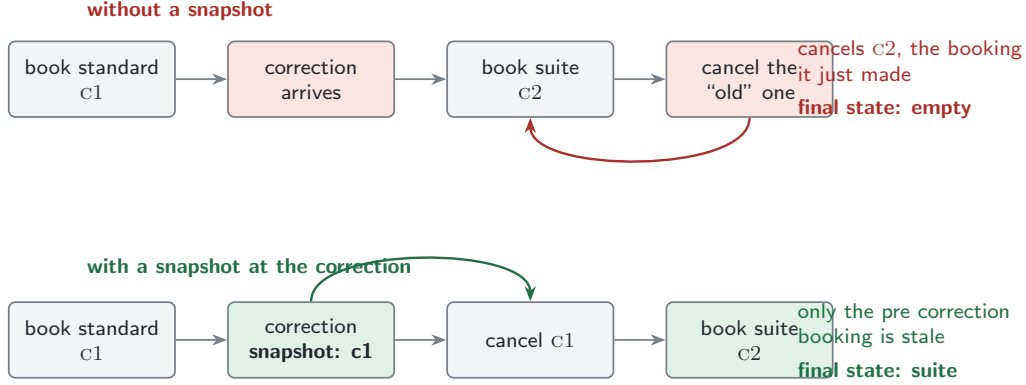


Figure 15: The stale booking failure, and the fix. Without a record of what existed when the correction landed, “cancel the old booking” can resolve to the booking just created during recovery. A real agent has no snapshot and must infer the distinction from context, which is why this is reported as a plausible source of genuine failures rather than merely a harness bug.

judge model detector and measuring agreement between the two is the first task before collecting real data.

7 Discussion

The framework will provide the first reasonably clear numbers for how tool calling agents deal with corrections, and a reusable harness others can point to their own agents. The most meaningful expected outcome will be a measure of silent inconsistency rate. A correction accepted in conversation and not reflected in the committed state is a failure the user cannot see and measuring how often it happens and when turns a vague worry into a monitored quantity. If H1 and H2 hold, the results will also yield a practical rule, that corrections are cheapest to honor before an action is committed, and that one should be able to see when an agent ought to wait to confirm rather than take an action.

The application is immediate for anyone who has agents in production. A detector for silent inconsistency, based on the same committed state check used here, can flag runs where the agent’s account and its actions disagree, without human labeling, because the check compares two machine readable objects. The same harness can also be used to test regression, as a change of prompt or model update can be run on the grid to see that the correction handling has not fallen away [5, 3]. This sort of drift is a common thing and hard to see when the underlying model changes under a fixed prompt. But over time the grid becomes a set of values to measure the progress of subsequent agent versions on a behavior that many task success metrics ignore.

There are a number of challenges to validity in advance. The testbed is a single booking domain and it cannot be generalized, so it was planned to add scheduling and purchasing domains as well. Scripted corrections may not be consistent with how actual users phrase and time their corrections; so testing against a sample of real transcripts would help to corroborate the claims. Claim detection is a classification and can result in error, so two independent detectors are used and disagreements are investigated. The results depend upon the model tested and more than one model is needed before any general statement is reached. H4 depends on the sensitivity measure chosen so its result should be interpreted as evidence of a link rather than a settled mechanism.

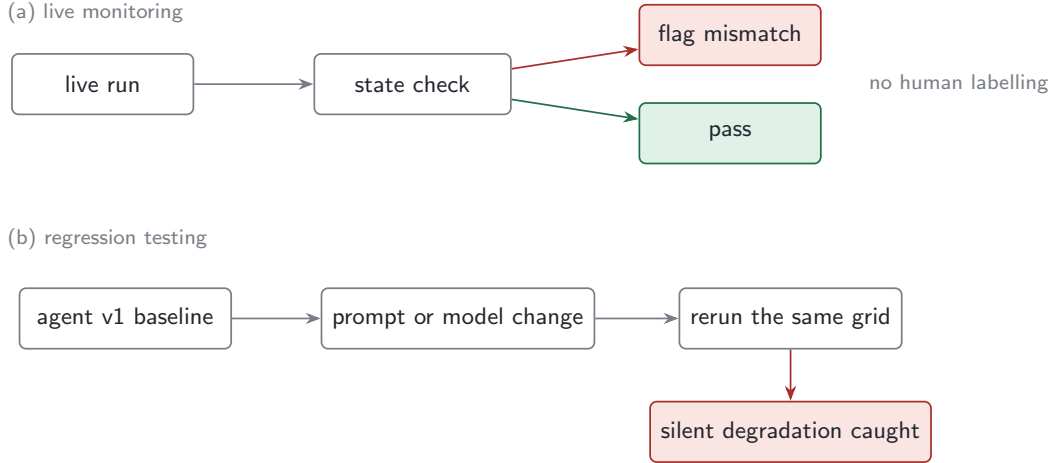


Figure 16: Two deployment uses of the same check. Because scoring compares two machine readable objects, it needs no human labelling and can run live (a). Holding the grid fixed across agent versions turns it into a regression test for a behaviour that ordinary task success metrics do not observe (b).

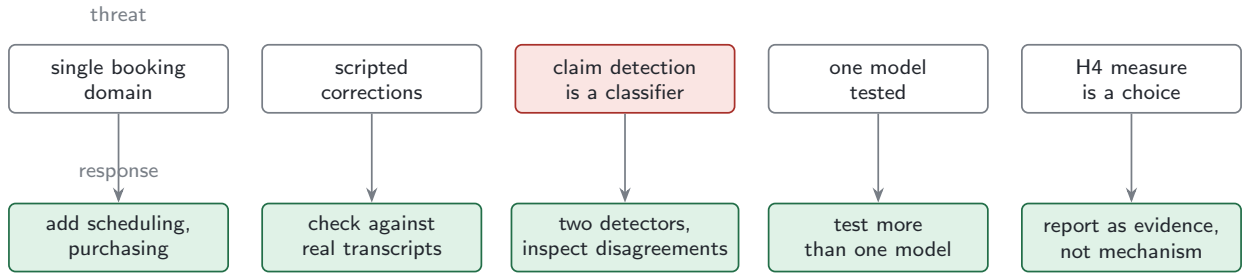


Figure 17: Threats to validity, each paired with the response already built into the design. The highlighted threat is the one with no adequate response yet: claim detection is the single component the headline metric depends on, and only the weaker of its two detectors is currently in use.

Three extensions follow naturally. The expansion of the domain sets shifts the results from one task family to one about tool calling agents. The silent inconsistency detector can be developed on its own and tested to see how well it can detect false claims from live traffic. The sensitivity link in H4 can be extended into a more general study examining the local sensitivity of an agent as a prediction of multiple failure types, linking this work to the effort to assess agent robustness more generally.

References

- [1] Voker. *Voker: AI Agent Analytics Platform*. Voker Documentation. <https://docs.voker.ai/overview>
- [2] Voker. *Voker Events, Sessions, Messages, and Fingerprints*. Voker Documentation, Concepts. <https://docs.voker.ai/concepts/events>
- [3] Voker. *Agents and Agent Versions: Versioned AI Assistants*. Voker Documentation, Concepts. <https://docs.voker.ai/concepts/agents>

- [4] Voker. *Production Monitoring for Customer Support Chat Agents*. Voker Cookbooks. <https://docs.voker.ai/cookbooks/customer-support-agent>
- [5] Voker. *Optimize Agents with Version Tracking*. Voker Cookbooks. <https://docs.voker.ai/cookbooks/agent-version-tracking>
- [6] Voker. *Voker Python SDK*. Voker Documentation, SDK Reference. <https://docs.voker.ai/sdks/python>
- [7] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations (ICLR)*. arXiv:2210.03629.
- [8] Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. (2024). τ -bench: A benchmark for tool-agent-user interaction in real-world domains. arXiv:2406.12045.
- [9] Rana, M., Man, C., Msiwa, A. E., Paine, J., Zhu, K., Dev, S., Sharma, V., & M R, A. (2025). AgentChangeBench: A multi-dimensional evaluation framework for goal-shift robustness in conversational AI. *NeurIPS 2025 Workshop on Multi-Turn Interactions in Large Language Models*. arXiv:2510.18170.