# EXTERNALIZING AGENT REASONING INTO SYNTHETIC ENVIRONMENTS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large language model agents increasingly use tools to perform tasks in software environments. Success requires applying rules to the current state and anticipating how an action affects later steps. We introduce EARIS, a framework that externalizes this reasoning into executable synthetic environments. We first describe environments through a common formulation of interface, internal states, and transition function. Using this formulation, we combine available records, implementations, and rules into an executable environment, then test and refine it. This environment can retain partial state without replicating the entire system. The agent then tries proposed actions on disposable copies and uses feedback about results, state changes, and tested follow-up actions to revise its decisions before real execution. Real interaction updates the stored synthetic state without training the agent. Across six benchmarks, GPT-5.4 task success rises from 22% to 49% on airline, 5% to 56% on TravelPlanner, and 22% to 55% on SOPBench when rules are hidden from the agent but available to the environment. Controls separate the benefits of rule access, checks, and feedback about later actions, while identifying settings where simple checks or explicit reminders suffice.

## 1 INTRODUCTION

Large language model (LLM) agents have been moving beyond conversation to complete tasks in increasingly complex external environments. They use tools to manage customer requests, operate applications, and plan travel (Yao et al., 2025; Trivedi et al., 2024; Xie et al., 2024). Such tasks require agents to understand the current state, apply operational rules, and anticipate how each action affects later decisions. For example, a customer may request both a different shirt size and a new shipping address. Changing the item first can prevent the address edit, whereas changing the address first permits both changes. The agent needs to reason about the order of actions even when each is initially valid.

As constraints accumulate, the model needs to track more conditions and dependencies to predict action outcomes. One approach is to train agents for a system (Zeng et al., 2024; Chen et al., 2023), while instructions and memory provide knowledge without parameter updates (Yao et al., 2023; Shinn et al., 2023; Zhao et al., 2024; Zhang et al., 2026a; Suzgun et al., 2026). Even with this knowledge, the agent needs to apply it to the current situation, and self-verification can repeat an earlier error (Huang et al., 2024). Real execution provides direct feedback, but an unwanted change may already have occurred before the agent recognizes the mistake.

Our key insight is that an executable environment can compute action consequences that the agent would otherwise have to predict in language. We therefore *externalize environment-dependent reasoning into a synthetic environment* where proposals can be tested without changing real state. To test these consequences, we need the relevant operations and records, rather than a complete system replica. We can run available code on separate records or use a copy of an existing application.

We introduce EARIS (Externalizing Agent Reasoning Into Synthetic Environments), which connects environment synthesis with trial execution (Figure 1). Our implementations span databases, service tools, and applications, combining sampled or copied state with accessible code and documented rules. The agent tries proposals on disposable copies and observes state changes and whether selected follow-up calls remain possible. In the retail example, this feedback reveals why the address



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

needs to change before the item. Real observations and successful writes then update the stored state for later previews.

Across six benchmarks, we evaluate synthesis accuracy, task completion, and compatibility with memory, with controls for rule access and explicit advice. We make two contributions. First, we provide a unified environment formulation and synthesis process that combines available state, operations, and rules into executable synthetic environments (§2). Second, we introduce an inference-time reasoning method that uses these environments to inspect action consequences before real execution, evaluated through task-level comparisons and component analysis (Sections 3 to 5).

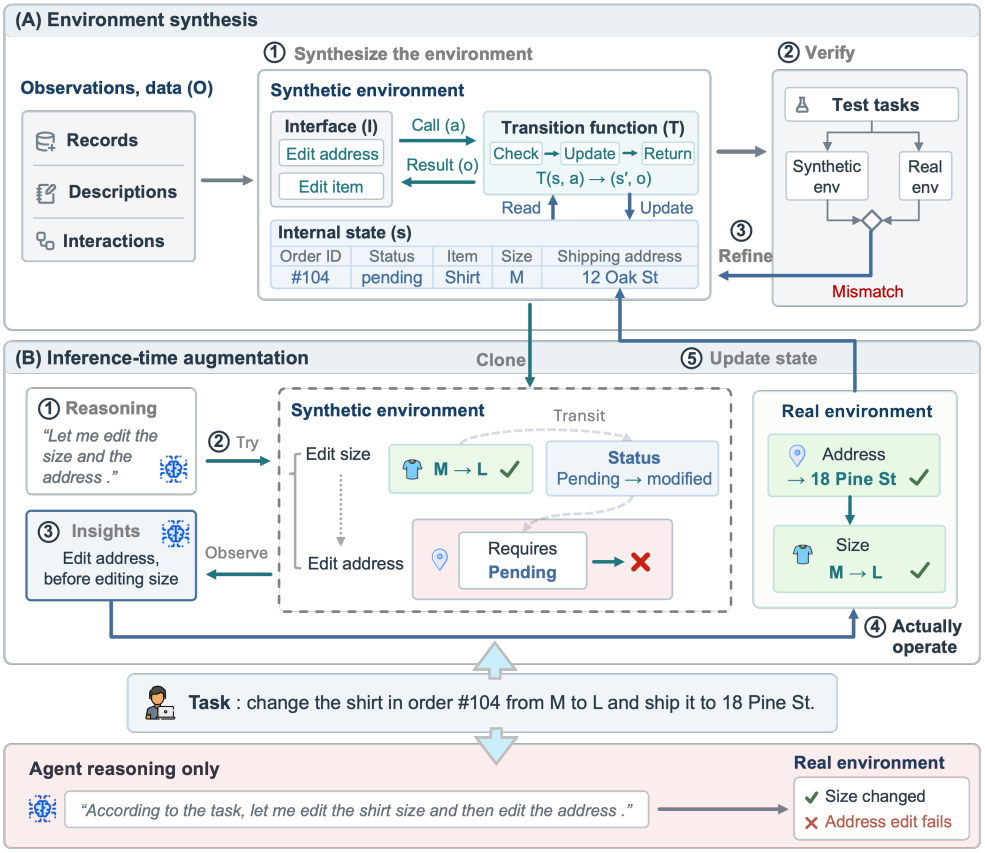


Figure 1: Environment synthesis and reasoning with EARIS. (A) Synthesize (1), verify (2), and refine (3). (B) Propose an action (1), try it on a copy (2), use the feedback (3), execute in the real environment (4), and update synthetic state (5).

## 2 ENVIRONMENT SYNTHESIS

Starting from the observations and data in Figure 1A, we define the environment's components, synthesize them, and test and refine their behavior. Section 3 follows their use in panel B.

### 2.1 ENVIRONMENT FORMULATION

We describe an environment as $E = (I, S, T)$, comprising an *interface* $I$, a set of *internal states* $S$, and a *transition function* $T$. The interface defines the calls an agent can make, including their arguments and returned results. In Figure 1A, these include editing an order's address and items. The internal state stores the data these calls read and modify, such as the order's status, size, and shipping address. The transition function implements the *check, update, return* sequence in the figure. Given state $s$ and call $a$, it checks whether the call is allowed, applies any permitted changes,

and returns the resulting state and observation,

$$T(s, a) = (s', o). \tag{1}$$

We include constraints as checks within $T$. If a call violates a constraint, the transition returns a reason and leaves the state unchanged. We use deterministic notation, with stochastic transitions represented by distributions over $(s', o)$.

For trial execution, we create a synthetic environment $\hat{E} = (\hat{I}, \hat{S}, \hat{T})$ that implements supported calls on its own state $\hat{s}$. It can retain only part of the real state, provided that the records needed by a tested call are available. The agent can inspect results and state changes on a copy without modifying the real environment. Missing records or inaccurate transitions can nevertheless produce incorrect feedback, motivating the verification and refinement steps below.

### 2.2 Synthesis procedure

The inputs on the left of Figure 1A are records, descriptions, and observed interactions. We denote the available information by $\mathcal{O}$, which also includes tool implementations when accessible. Environment synthesis produces an executable environment and initial state,

$$\mathrm{Synth}(\mathcal{O}) = (\hat{E}, \hat{s}_0). \tag{2}$$

We reuse available tool code or runnable applications to implement transitions. Our experiments rely on this access and do not establish recovery of unknown transitions from observations alone.

**Synthesize the environment.** In Figure 1A, step 1, we use the available inputs to implement the three components. For the retail example, we implement address and item edits over records containing order `#104`, size `M`, status `pending`, and address `12 Oak St`. The item-edit transition checks the status, changes the size, and sets the status to `modified`. The address edit requires `pending`, so it fails after the item edit.

For database and service environments, we initialize state from $K\%$ of the records and preserve references between related records, such as an order and its customer. Some applications instead copy their task state. We connect available tool implementations to this separate state and add checks for documented rules that the tools do not enforce. Appendix C details sampling and rule checks for each benchmark.

**Verify the environment.** To check whether this environment behaves as expected, step 2 runs the same test calls in the synthetic environment and a separate copy of the reference environment. We compare their returned results, state changes, and decisions to allow or refuse a call. Recorded interactions can also provide expected results when reference execution is unavailable. Without either source, tests can check internal consistency but cannot establish agreement with the real environment.

**Refine the environment.** When these results disagree, step 3 uses the mismatch to identify what needs correction. If a record is missing, we add it or repair its references. If a call produces the wrong result, we correct its transition checks or updates and rerun the tests. For example, leaving the order status unchanged after an item edit would incorrectly allow a later address edit. Corrections can require application-specific inspection (Appendix B). We accept environments that meet the specified test thresholds, except in diagnostic experiments that deliberately retain failures. Passing these tests does not guarantee correct behavior on untested calls.

## 3 Inference-time agent reasoning

Figure 1B follows a request to change the shirt in order `#104` from `M` to `L` and update its shipping address from `12 Oak St` to `18 Pine St`. The agent initially plans to change the size first. We follow how trial execution reveals the problem with this order and helps the agent revise it before changing the real order.

### 3.1 TRIAL EXECUTION AND FEEDBACK

**Propose an action.** In step 1, the agent uses the task and interaction history to propose a call $a$. Here, it proposes changing the shirt size first and passes this call to the synthetic environment for a preview.

**Try the action.** To try the proposal without committing a change, step 2 uses the environment synthesized in panel A, following the clone arrow. We copy the current synthetic state $\hat{s}$ and execute the proposed call through its interface,

$$(\tilde{s}', \tilde{o}) = \hat{T}(\text{copy}(\hat{s}), a). \tag{3}$$

The preview reports the result $\tilde{o}$ and the changes from $\hat{s}$ to $\tilde{s}'$. If the call succeeds, selected follow-up calls run on separate copies of $\tilde{s}'$ to test what remains possible. A refused call returns a reason without changing the copy. All trial states are discarded after feedback is produced, leaving both real state and stored synthetic state unchanged.

In the illustrated trial, the item edit succeeds. It changes the size from `M` to `L` and the status from `pending` to `modified`. The subsequent address edit is refused because it requires `pending`. Thus, the first action succeeds but prevents completion of the second request.

**Use the feedback.** In step 3, the agent uses these results to revise the order to address first, then size. The address edit leaves the status `pending`, so the item edit can still succeed afterward. A success message for the size change alone would not reveal this dependency.

The feedback is limited to the follow-up calls and arguments we test. Candidates are specified per application (Appendix C.2). TravelPlanner additionally searches its planning data for feasible alternatives, while AppWorld reports changed application records. Values absent from sampled records produce warnings rather than prohibitions.

### 3.2 REAL EXECUTION AND STATE UPDATES

**Execute the revised action.** In step 4, the agent first previews the address edit. Repeating the same operation and arguments then executes it in the real environment,

$$(s', o) = T(s, a). \tag{4}$$

The agent then previews and executes the size change, completing both requests. A revised call receives a new preview. Reads and messages run immediately, while each new write first produces a preview. A preview refusal is advisory and can be overridden by repeating the call. Real state may change between preview and execution, so the actual result $o$ determines whether the action succeeded.

**Update the synthetic state.** After real execution, we update the stored synthetic state so that later trials use the latest available records. This is the return arrow in step 5 from panel B to panel A. Real reads add missing records and overwrite older versions with the same identifiers. Successful real writes are replayed on $\hat{s}$. Thus, after the real address edit succeeds, subsequent previews start with `18 Pine St` as the shipping address. The hypothetical size change from the earlier trial is not retained. If records needed for a preview are still missing, the agent is asked to read them before trying again. These updates change the stored state without automatically relearning the transition function. External changes and nondeterministic behavior can still cause disagreement.

We adapt this service-tool procedure to other interfaces, previewing code blocks in AppWorld, plans in TravelPlanner, and queries in Spider. Application copies require access to initial task state beyond the agent's observations. Appendix C details these adaptations and the interaction algorithm.

## 4 EXPERIMENTAL SETUP

We evaluate both parts of the method in Figure 1. First, we test whether the synthesized environments reproduce the behavior needed for trial execution. We then examine whether agents use this feedback to complete tasks more successfully.

**Benchmarks.** We choose six benchmarks to cover service interactions, travel planning, application use, and database queries. These are $\tau$-bench airline (50 tasks) and retail (115) (Yao et al., 2025), TravelPlanner (180 validation queries with tool use) (Xie et al., 2024), AppWorld (168 normal tasks) (Trivedi et al., 2024), Spider 2.0-lite (135 tasks with local databases) (Lei et al., 2025), and SOPBench (830 tasks in seven domains) (Li et al., 2025). We report AppWorld's 417 challenge tasks separately because they informed protocol development. To examine additional settings, we use BFCL multi-turn (Patil et al., 2025), Gaia2 (Froger et al., 2025), and $\tau^2$-bench telecom (Barres et al., 2025). Controlled retail tasks let us vary rule complexity, while 60 designed tasks isolate dependencies between actions.

**LLM settings.** We use GPT-5.4 and GPT-5 as the models driving the agents' decisions and tool calls, testing whether the gains hold across both. For GPT-5.4, we disable reasoning on airline, retail, AppWorld, and Spider and use minimal reasoning on TravelPlanner and SOPBench. For GPT-5, we use minimal reasoning throughout. We also test medium reasoning to examine whether additional model reasoning changes the value of previews (Appendix D.2). Where a benchmark requires a simulated user, we use GPT-5.4 for that role. To extend the comparison to an open-weight model, we evaluate Qwen3-14B (Yang et al., 2025) with thinking disabled on airline, retail, and SOPBench. EARIS requires no parameter training.

**Baselines.** We start with *ReAct* (Yao et al., 2023), which reasons and calls tools without synthetic execution. To test whether additional review alone explains the gains, *self-verify* uses the same preview-and-repeat protocol but asks the model to predict outcomes. Its model-call counts are similar on airline, retail, and SOPBench (Appendix C). To isolate rule enforcement, our *rule-check baseline* blocks forbidden actions and explains why, without reporting state changes or testing follow-up calls (Zwerdling et al., 2025; Reddy et al., 2026). We also evaluate ACE (Zhang et al., 2026a) and Dynamic Cheatsheet (DC) (Suzgun et al., 2026) alone and with EARIS to test whether previews complement memory. Using their published code, we let both update a context of at most 6,000 characters from each task's transcript and official outcome. ReAct and EARIS alone receive no outcome signal (Appendix C).

**Policy settings.** In the main airline, retail, and SOPBench comparisons, we withhold written policy from the agent's prompt but retain it in EARIS's checks. These comparisons therefore include an information advantage. To distinguish that advantage from the value of execution feedback, we also give every agent the rules and compare against the baseline with the same checks (Sections 5.3 and 5.4).

**Evaluation metrics.** We measure task success using each benchmark's official criterion. We also report the *error-free task rate* to show whether agents avoid rejected or rule-breaking changes, constraint violations on TravelPlanner, or failed queries on Spider. This measure complements task success, since avoiding action can avoid errors without completing the task. Although producing feedback requires no model call on the six main benchmarks, interpreting it can add turns, so we report model calls per task. Previews do not consume the real-action budget. We report paired wins and losses as $+w/-l$ and assess uncertainty through paired tests and bootstrap intervals, correcting the twelve ReAct comparisons for multiple testing. Because ACE and DC share evolving memory across tasks, we treat their tests as descriptive (Appendix F).

# 5 EVALUATION

We first assess the synthesized environments and task completion, then use controlled comparisons to explain when execution feedback helps.

## 5.1 ENVIRONMENT SYNTHESIS ACCURACY

**Database consistency.** We first test whether sampled databases retain the structure needed for execution and joins (Appendix B). All 30 local Spider databases pass at sample fractions from 1% to 50%, and the tests detect all 120 deliberately damaged variants.

**Environment synthesis at scale.** To examine synthesis on a larger database, we use an IMDb snapshot with 144 million rows. It occupies 12.79 GB, while the sampled database occupies 14.8 MB. Environment synthesis takes about 10 seconds and testing 190 seconds. It passes execution and result-structure checks on 12 analyst queries used to repair joins. This demonstrates compact executable state, not held-out answer accuracy (Appendix B.1).

**State coverage and behavior.** We next vary how much state is initially available. In controlled retail, success stays near 0.73 as the initial sample varies from 1% to 50%, because real reads supply missing records. Inverting the refund-unit rule instead lowers success to 0.33 (Appendix E). At the 1% sample, a missing `pending` order causes an initial validation test to fail, yet real reads restore task state during interaction. These updates recover missing records but do not correct an inverted rule, distinguishing state coverage from transition accuracy.

On SOPBench, omitting reference-based correction of generated checks retains similar success, although their generation instructions were developed on the benchmark.

**Errors during use.** Validation can miss errors during use. In our evaluation, an environment generated with Agent World Model (AWM) (Wang et al., 2026) passes the retail tests after repair yet allows 49 calls that fail in real execution. The inputs differ from EARIS, so this is not a controlled ranking, but it motivates measuring errors beyond validation (Appendix B.3).

### 5.2 END-TO-END AGENT PERFORMANCE

**Task completion.** Turning to task completion, Table 1 shows the largest GPT-5.4 gains on airline (0.22 to 0.49), TravelPlanner (0.05 to 0.56), and SOPBench (0.22 to 0.55). The airline and SOPBench comparisons include access to policy withheld from the base agent. After Holm correction over twelve comparisons, gains remain significant on TravelPlanner and SOPBench with both models and airline with GPT-5.4 (adjusted $p \leq 0.02$). Retail and AppWorld normal gains are not significant. Spider gains do not survive this correction. AppWorld uses only the normal split, which did not inform protocol development.

Table 1: Task success with and without EARIS. Shading and arrows indicate gains or losses. Bold marks the row maximum. DC denotes Dynamic Cheatsheet, and † marks learning from task outcomes.

| | ReAct | | ACE† | | DC† | |
|---|---|---|---|---|---|---|
| Benchmark | alone | +EARIS | alone | +EARIS | alone | +EARIS |
| GPT-5.4 | | | | | | |
| Airline | 0.22 | **0.49** (↑0.27) | 0.27 | 0.45 (↑0.19) | 0.18 | 0.45 (↑0.26) |
| Retail | 0.53 | 0.55 (↑0.03) | 0.53 | 0.59 (↑0.05) | 0.53 | **0.60** (↑0.07) |
| TravelPlanner | 0.05 | 0.56 (↑0.51) | 0.03 | **0.57** (↑0.54) | 0.07 | 0.53 (↑0.46) |
| AppWorld | 0.82 | **0.84** (↑0.02) | 0.79 | 0.83 (↑0.05) | 0.82 | **0.84** (↑0.02) |
| Spider 2.0 | 0.45 | **0.55** (↑0.10) | 0.46 | 0.48 (↑0.02) | 0.47 | 0.50 (↑0.03) |
| SOPBench | 0.22 | 0.55 (↑0.33) | 0.18 | 0.52 (↑0.34) | 0.22 | **0.56** (↑0.34) |
| GPT-5 | | | | | | |
| Airline | 0.29 | 0.44 (↑0.15) | 0.36 | **0.46** (↑0.10) | 0.18 | 0.38 (↑0.20) |
| Retail | 0.43 | **0.54** (↑0.10) | 0.50 | 0.41 (↓0.09) | 0.49 | 0.49 (±0.00) |
| TravelPlanner | 0.01 | 0.36 (↑0.36) | 0.00 | **0.41** (↑0.41) | 0.00 | 0.24 (↑0.24) |
| AppWorld | 0.43 | 0.42 (↓0.02) | **0.46** | 0.41 (↓0.05) | 0.36 | 0.32 (↓0.05) |
| Spider 2.0 | 0.33 | **0.41** (↑0.08) | 0.33 | 0.37 (↑0.04) | 0.36 | 0.35 (↓0.01) |
| SOPBench | 0.13 | 0.43 (↑0.30) | 0.23 | 0.54 (↑0.31) | 0.35 | **0.56** (↑0.20) |

**Compatibility with memory.** We add EARIS to ACE and DC to test whether previews complement memory. Both improve on airline, TravelPlanner, and SOPBench with both models. For example, GPT-5.4 TravelPlanner success rises from 0.03 to 0.57 with ACE and 0.07 to 0.53 with DC. Because both methods learn from official outcomes, adding previews also changes the experience stored in memory. On AppWorld normal with GPT-5, both combinations reduce success.

**Repeatability and cost.** Three airline runs yield 0.49 to 0.52 for EARIS versus 0.22 to 0.25 for ReAct. A second SOPBench run also retains the gain (Appendix E). Most other settings have one run. Previews increase GPT-5.4 model calls from 17.6 to 21.3 on airline and 6.6 to 9.4 on SOPBench (Table 5).

### 5.3 RULE-ACCESS CONTROLS

**Policy access.** Because the main comparisons give EARIS access to rules withheld from the agent, we next provide the same policy to all agents. Giving the policy to ReAct raises success from 0.22 to 0.53 on both airline and SOPBench (Table 2). On SOPBench, checking raises it further to 0.67 with EARIS ($+154/-36$), or 0.68 with the simpler rule check. On airline, however, previews lower success from 0.53 to 0.40 ($+1/-7$, $p = 0.07$). None of the lost tasks involves a refusal. Additional interaction can hurt even when checks do not reject the proposal.

Table 2: Rule access and action checks (GPT-5.4). SV denotes self-verify, and Check denotes the rule check. *Retail has no added policy checks.

| | Policy hidden | | | Policy given | | | |
|---|---|---|---|---|---|---|---|
| Benchmark | ReAct | Check | EARIS | ReAct | SV | Check | EARIS |
| Airline | 0.22 | 0.48 | 0.49 | **0.53** | 0.50 | 0.50 | 0.40 |
| Retail | 0.53 | 0.60* | 0.55 | 0.59 | 0.56 | **0.66*** | 0.61 |
| SOPBench | 0.22 | 0.56 | 0.55 | 0.53 | 0.58 | **0.68** | 0.67 |
| TravelPlanner | 0.05 | 0.15 | **0.56** | 0.23 | 0.17 | 0.34 | 0.42 |

**Planning constraints.** For TravelPlanner, we make the same comparison by supplying the constraints used by the checker. This raises ReAct from 0.05 to 0.23. With the same list, the rule check reaches 0.34 and EARIS 0.42 ($+30/-16$, $p = 0.054$). This difference is uncertain in one run. With the list hidden from the agent but retained by the checker, EARIS reaches 0.56. The long list also increases step-limit failures. Raising ReAct's limit from 30 to 45 actions improves success to 0.317 (Appendix C). These comparisons do not isolate preview execution from its additional search for alternatives.

**Self-verification.** We next compare execution feedback with the model's own review. With the rules given, self-verify reaches 0.58 on SOPBench and 0.17 on TravelPlanner, versus 0.67 and 0.42 for EARIS. On airline it reaches 0.50, above EARIS's 0.40. Table 4 gives the other comparisons. Similar model-call counts in several settings help control for extra review, but do not equate token cost or the information in the feedback.

### 5.4 COMPONENT ANALYSIS

**Rule checks.** To separate rule enforcement from richer feedback, we compare EARIS with the rule-check baseline. With the policy hidden, the latter nearly matches EARIS, scoring 0.48 versus 0.49 on airline and 0.56 versus 0.55 on SOPBench. Removing rule checks but keeping copied state lowers airline success to 0.29 (Figure 2a). Much of the gain in these settings therefore comes from checking rules, not showing later consequences. The preview uses sampled state augmented by real reads. The rule check reads real state directly. Their error-free task rates on airline are 0.98 and 0.86.

**Feasible alternatives.** For planning, suggesting revisions proves useful. On two-stage TravelPlanner, reporting violated constraints yields 0.15 success, while adding feasible alternatives raises it to 0.56. When planning information is supplied in advance, these values are 0.52 and 0.83 (Figure 2b).

**Action dependencies.** We construct 60 retail tasks to test dependencies between changes (Appendix D). In 30 address tasks, the user requests an item change followed by an address change, but completing both changes requires editing the address first. With the policy hidden, EARIS solves 26/30, versus 0/30 for ReAct and 0/30 when only the follow-up feedback is removed. With the policy given, these counts are 21, 2, and 1. Across all 60 tasks, full feedback outperforms the version without follow-up feedback by $+30/-3$ and $+24/-6$, respectively. The other 30 tasks reveal

a second item request later. Previews do not help (10 versus 11 successes with the policy hidden). Feedback can reveal a consequence of the current proposal, but does not elicit an unstated request.

**Explicit reminders.** To test whether an explanation can supply the same dependency, we repeat all 30 address tasks with the full policy and a fixed reminder to ask about address changes and perform them before item edits (Appendix D.2). With reasoning off, full previews solve 20/30 versus 2/30 without follow-up feedback ($+18/-0$). Adding the reminder to the latter raises success to 25/30. Direct execution with the reminder reaches 28/30 (Figure 2c). At medium reasoning, both EARIS and reminded direct execution solve 30/30, with 11.7 versus 9.8 agent calls. The reminder explicitly supplies the ordering dependency, showing that this information can also be communicated without execution.

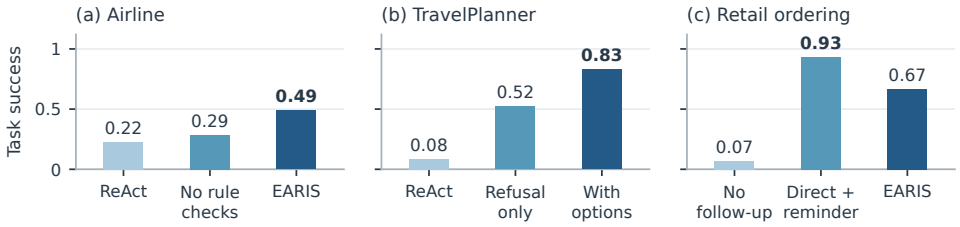


Figure 2: Contributions of execution feedback (GPT-5.4). (a) Rule checks. (b) Feasible alternatives. (c) Follow-up feedback and explicit reminders.

**What execution contributes.** The retail controls distinguish action validity from consequences for later decisions. A rule check accepts the initial item edit, while follow-up execution reveals the lost address edit. An explicit reminder can supply the same dependency. The benefit therefore depends on providing information relevant to the decision, whether through execution or an adequate explanation.

### 5.5 CONSTRAINT AND BUDGET ANALYSIS

**Real-action efficiency.** We vary rule complexity and the real-action budget on 60 controlled retail tasks per condition. This tests whether identifying invalid calls in previews leaves more real actions available for completing the task. At 20 real actions, increasing hidden rules from zero to eight lowers ReAct success from 0.78 to 0.33, while EARIS stays between 0.72 and 0.78 (Figure 3a). At eight rules, increasing the budget from 8 to 30 raises EARIS from 0.27 to 0.82 and self-verify from 0.18 to 0.60. ReAct remains at or below 0.33. At 20 actions, EARIS and self-verify use 17.4 and 16.9 model calls. A second seed retains gains at two, four, six, and eight rules (Appendix E). The benefit is fewer failed real actions, not less total computation.

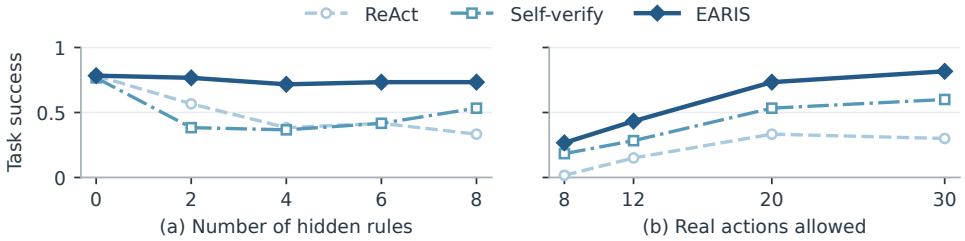


Figure 3: Success under varying constraints and budgets (GPT-5.4). (a) Hidden rules with 20 real actions. (b) Real-action budget with eight hidden rules.

### 5.6 FAILURE ANALYSIS

**Execution ambiguity.** AppWorld agents sometimes mistook previews for completed execution. Changes to execution instructions, reported records, and secret-dependent operations raised challenge success from 0.667 to 0.779. These changes used the same challenge tasks. Table 7 reports

them separately. On normal tasks, unused in this development, GPT-5.4 changes from 0.82 to 0.84 ($+8/-5$), and GPT-5 from 0.43 to 0.42.

**Unresolved decisions.** Accurate feedback is useful only when the agent acts on it. Qwen3-14B improves on SOPBench (0.04 to 0.32) but overrides refusals on airline and retail in 55% and 64% of cases. Retail and BFCL tools already explain many errors, limiting the additional value of previews. Many Gaia2 failures involve actions never proposed. The second-item tasks require information never requested. On telecom, reconstructing state from customer statements adds model calls without beating self-verify (Appendix D.4).

## 6 Related work

**Agent adaptation.** Training, reflection, and memory help agents acquire environment knowledge (Zeng et al., 2024; Chen et al., 2023; Shinn et al., 2023; Zhao et al., 2024). For example, ACE and Dynamic Cheatsheet update context from task experience (Zhang et al., 2026a; Suzgun et al., 2026). EARIS complements these methods by letting the agent execute a proposal and inspect its consequences in the current state. This use of execution feedback also connects to execution-guided SQL and self-debugging (Wang et al., 2018; Chen et al., 2024). Self-Refine instead uses model-generated feedback (Madaan et al., 2023). Studies of intrinsic self-correction show that such feedback can be unreliable on reasoning tasks (Huang et al., 2024).

**Environment synthesis.** Executable environments serve several purposes in prior work. WorldCoder and related methods build models for planning (Tang et al., 2024; Dainese et al., 2024), while GenSim, RoboGen, EnvGen, and Agent World Model create environments for training (Wang et al., 2024a;b; Zala et al., 2024; Wang et al., 2026). We organize synthesis around a common interface, state, and transition formulation for use during inference. Our implementations combine sampled or copied state with accessible code and documented rules, rather than all generating new transition code.

**Execution-based reasoning.** Gecko combines validation, simulated responses, estimated state, and task feedback. GATS uses this environment to refine tool-call plans before real execution (Zhang et al., 2026b). EARIS instead executes available code on reconstructed state and interleaves individual previews with real interaction. RAP, WebDreamer, and WMA also simulate outcomes for planning (Hao et al., 2023; Gu et al., 2024; Chae et al., 2025), whereas ToolEmu emulates tool execution to evaluate risks (Ruan et al., 2024). Our self-verify control examines local outcome prediction and does not reproduce GATS.

**Action validation.** Policy guards, generated checks, solvers, and graph constraints determine whether a call may run (Zwerdling et al., 2025; Xiang et al., 2025; Winston et al., 2026; Wu & Gong, 2026). Such checks can improve success when tools silently permit violations (Reddy et al., 2026). We examine when these checks suffice and when executing follow-up calls supplies useful information about later decisions. GoEX addresses the related problem of undoing actions and limiting damage after execution (Patil et al., 2024).

## 7 Conclusion

We introduced EARIS to externalize environment-dependent reasoning into executable synthetic environments. The method begins by organizing available records, implementations, and rules into an interface, state, and transition function, then testing and refining the resulting environment. Agents can then try actions on disposable copies and use the consequences to revise their decisions before real execution. As interaction proceeds, real observations and successful writes update the state used by subsequent trials. Across six benchmarks, the results show gains in task completion and compatibility with memory. The controlled comparisons show why these gains vary. Rule access explains much of the improvement in some settings, while other tasks benefit from learning how an action affects a later step. Simple checks or explicit advice can suffice when they supply the relevant information. Synthetic environments provide a way to obtain that information through trial execution, before the agent commits changes to the real system.

### AI USE STATEMENT

We used generative AI tools to draft parts of the manuscript, polish the writing, and refine figures. AI tools also assisted with experimental planning and implementation, preparation of additional evaluation cases, and result analysis. The research ideas originated with the authors. The authors take full responsibility for the accuracy and integrity of the paper, including all AI-assisted material.

### REPRODUCIBILITY STATEMENT

Appendices B to F specify synthesis, execution, comparisons, and evaluation corrections. The supplementary archive contains frozen outcomes, numerical and figure reproduction scripts, and the additional retail controls. It reproduces the main comparisons without model calls. Repeating agent runs requires API access and benchmark dependencies.

## REFERENCES


Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. $\tau^2$-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. URL https://arxiv.org/abs/2506.07982.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. Web agents with world models: Learning and leveraging environment dynamics in web navigation. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2410.13232.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. FireAct: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023. URL https://arxiv.org/abs/2310.05915.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2304.05128. arXiv:2304.05128.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.15383. arXiv:2405.15383.

Romain Froger, Pierre Andrews, Matteo Bettini, Amar Budhiraja, Ricardo Silveira Cabral, et al. ARE: Scaling up agent environments and evaluations. *arXiv preprint arXiv:2509.17158*, 2025. URL https://arxiv.org/abs/2509.17158.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, Huan Sun, and Yu Su. Is your LLM secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559*, 2024. URL https://arxiv.org/abs/2411.06559.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. URL https://aclanthology.org/2023.emnlp-main.507/. arXiv:2305.14992.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2310.01798. arXiv:2310.01798.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2411.07763.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, and Xifeng Yan. SOPBench: Evaluating language agents at following standard operating procedures and constraints. *arXiv preprint arXiv:2503.08669*, 2025. URL https://arxiv.org/abs/2503.08669.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.17651. arXiv:2303.17651.

Shishir G. Patil, Tianjun Zhang, Vivian Fang, Noppapon C., Roy Huang, Aaron Hao, Martin Casado, Joseph E. Gonzalez, Raluca Ada Popa, and Ion Stoica. GoEX: Perspectives and designs towards a runtime for autonomous LLM applications. *arXiv preprint arXiv:2404.06921*, 2024. URL https://arxiv.org/abs/2404.06921.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning*, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Vikas Reddy, Sumanth Reddy Challaram, and Abhishek Basu. Reason less, verify more: Deterministic gates recover a silent policy-violation failure mode in tool-using LLM agents. *arXiv preprint arXiv:2607.07405*, 2026. URL https://arxiv.org/abs/2607.07405.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris J. Maddison, and Tatsunori Hashimoto. Identifying the risks of lm agents with an lm-emulated sandbox. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2309.15817. arXiv:2309.15817.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. In *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2026. URL https://aclanthology.org/2026.eacl-long.333/.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2402.12275.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. URL https://aclanthology.org/2024.acl-long.850/.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding. *arXiv preprint arXiv:1807.03100*, 2018. URL https://arxiv.org/abs/1807.03100.

Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang, Huazhe Xu, and Xiaolong Wang. GenSim: Generating robotic simulation tasks via large language models. In *International Conference on Learning Representations*, 2024a. URL https://arxiv.org/abs/2310.01361.

Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards unleashing infinite data for automated robot learning via generative simulation. In *International Conference on Machine Learning (ICML)*, 2024b. URL https://arxiv.org/abs/2311.01455.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2602.10090.

Cailin Winston, Claris Winston, and René Just. Solver-aided verification of policy compliance in tool-augmented LLM agents. *arXiv preprint arXiv:2603.20449*, 2026. URL https://arxiv.org/abs/2603.20449.

Jie Wu and Ming Gong. Policy-invisible violations in LLM-based agents. *arXiv preprint arXiv:2604.12177*, 2026. URL https://arxiv.org/abs/2604.12177.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. GuardAgent: Safeguard LLM agents via knowledge-enabled reasoning. In *International Conference on Machine Learning (ICML)*, 2025. URL https://proceedings.mlr.press/v267/xiang25a.html.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. TravelPlanner: A benchmark for real-world planning with language agents. In *International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2402.01622.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. $\tau$-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2406.12045.

Abhay Zala, Jaemin Cho, Han Lin, Jaehong Yoon, and Mohit Bansal. EnvGen: Generating and adapting environments via LLMs for training embodied agents. In *Conference on Language Modeling (COLM)*, 2024. URL https://arxiv.org/abs/2403.12014.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In *Findings of the Association for Computational Linguistics (ACL)*, 2024. URL https://arxiv.org/abs/2310.12823.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations (ICLR)*, 2026a. URL https://arxiv.org/abs/2510.04618.

Zeyu Zhang, Guohao Li, Zhenchang Xing, Alexandros Apostolopoulos, Yu Lin Lee, and Liang Zheng. Gecko: A simulation environment with stateful feedback for refining agent tool calls. *arXiv preprint arXiv:2602.19218*, 2026b. URL https://arxiv.org/abs/2602.19218.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: LLM agents are experiential learners. In *AAAI*, 2024. URL https://arxiv.org/abs/2308.10144.

Naama Zwerdling, David Boaz, Ella Rabinovich, Guy Uziel, David Amid, and Ateret Anaby Tavor. Towards enforcing company policy adherence in agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2025. URL https://aclanthology.org/2025.emnlp-industry.41/.

# A CASE STUDY

We follow the retail scenario in Figure 1 from the customer's request to its completion. The example shows how we synthesize the environment and how the agent uses it to discover the order in which to act. We use an illustrative customer, dialogue, and abbreviated readouts, with transitions and previews checked against the implementation. The direct-execution path shows how the ordering error can occur rather than reproducing a recorded baseline trajectory. Section 5.4 evaluates the dependency across tasks.

## A.1 TASK AND ACTION DEPENDENCY

Maya notices two mistakes in an order she has just placed. Her shirt is size `M` rather than `L`, and the order still lists her old address. She asks the store's assistant to correct both mistakes.

> "Please change the shirt in order `#104` from size `M` to `L` and send it to `18 Pine St, Boston, MA 02111`, instead of `12 Oak St`."

The assistant looks up Maya's customer record (`U7`), order `#104`, and the shirt's product record. The order is `pending`, its shipping address is `12 Oak St, Boston, MA 02110`, and its single item is variant `shirt-m`. The catalogue lists both `shirt-m` and `shirt-l` as available at `$50`, and Maya has a valid card on file. The assistant explains the two changes and confirms that there is no price difference. Maya replies, "Yes, please make both changes." She wants a size-`L` shirt delivered to the new address.

**Direct execution.** An agent may follow the order of Maya's request and respond, "I'll change the size first, then update the address." It calls the item-edit tool on the real order. The call succeeds, replacing `shirt-m` with `shirt-l`, but also changes the order status to `modified`. The agent then submits the address change, which fails because the order is no longer `pending`. Both tools require `pending`, so the first edit removes a condition needed for the second. The agent can only report, "The shirt is now size `L`, but I could not update the shipping address." It has satisfied half the request and lost the ability to finish through these tools. Checking only the first call would not reveal the ordering problem because that call is valid.

## A.2 ENVIRONMENT SYNTHESIS

With EARIS, the agent can discover this dependency before modifying Maya's real order. The retail implementation first constructs an environment for trial execution, then incorporates the records retrieved during the conversation. Its interface exposes order-edit calls, its internal state stores linked customer, order, and product records, and its transition function implements edits and their restrictions. This instance reuses tool code and the full catalogue. It constructs separate state rather than generating behavior from scratch. The same components organize other implementations, whose input requirements differ.

**1. Construct state.** Environment synthesis samples customers together with their orders, rather than selecting unrelated rows independently. It retains the full product catalogue beyond this sample budget, so an order's item identifier can still be resolved to a product and its available variants. It repairs references by checking that retained orders point to retained users, item variants exist in the catalogue, and users' order lists contain only retained orders. It can also restore status classes missing from the sample to exercise different tool paths. These links allow the tools to find the customer, order, and product records needed to execute a call.

**2. Attach operations.** We next connect the retail tools to these records so that calls can read and modify the separate state. For `#104`, the relevant tools are `modify_pending_order_items` and `modify_pending_order_address`. The former checks the order, replacement variant, and payment method, then changes the item and status. The latter checks the order status and changes its address without closing it to further edits. These rules come from the tool code, not from guessing patterns in the records. No additional rule check is needed for this dependency. Implementations with unenforced written rules add checks separately (§2.2). Connecting these operations to separate records makes their effects inspectable without changing the real order.

**3. Test and refine.** The benchmark provides reference implementations that can run on separate copies, so validation tests compare results, refusals, and state changes without modifying live orders (Appendix B). If the results disagree, we correct the records, links, or checks responsible and rerun the tests. The illustrated example produces no such mismatch. Backend tests confirm that editing the item before the address fails on the second call, whereas reversing the order permits both edits. This validates the illustrated dependency, not every possible request.

**4. Update task state.** The initial sample need not contain Maya or her order. Her successful customer, order, and product lookups are merged into the synthetic environment, overwriting sampled versions with the returned records. Thus the task-specific state now links `U7` to order `#104`, the size-`M` variant, the old address, and the available size-`L` replacement. If a required record is still missing, EARIS asks for a read rather than inventing it. This separates reusable environment synthesis from adding the current records for a particular request.

### A.3 REASONING AND EXECUTION

**Trial execution.** Suppose the agent initially proposes changing the item first, as in the direct-execution example. EARIS executes it on a disposable copy of the updated synthetic state. On that copy, the size becomes `L` and the status becomes `modified`. The implementation then tries candidate follow-up calls on copies of this resulting state. The address-edit tool rejects the `modified` status. The agent receives an abbreviated readout of the result.

> **Not executed.** The item edit would succeed.
> **Effect.** Size `M` → `L`. Status `pending` → `modified`.
> **Remaining actions.** Editing the address on this order would be refused.

The warning arrives while the real order is still size `M`, `pending`, and addressed to `12 Oak St`. The disposable copy is discarded. Neither the real order nor the persistent synthetic state has been changed by this proposal.

**Revised execution.** The readout connects the proposed action to the part of Maya's request still outstanding. The agent can then explain its revised plan. "I'll update your address before changing the size, because changing the item first would prevent the address update." It now proposes the address edit on a fresh copy. This preview shows `18 Pine St` as the shipping address, leaves the status `pending`, and indicates that the item edit remains possible. The agent reissues the identical address call to execute it in the real system, and EARIS mirrors the successful change into its persistent state.

The agent next proposes the size change. Its fresh preview starts from the *updated* address, so the result is a size-`L` shirt at `18 Pine St`, with status `modified`. Further edits are now unavailable, but both requested changes are accounted for. After reissuing this call and receiving the actual result, the agent confirms completion.

> "I've changed the shirt in order `#104` to size `L` and updated the shipping address to `18 Pine St, Boston, MA 02111`. There is no additional charge."

**Decision evidence.** The decisive feedback is not merely that the item edit would succeed. It reveals that success would remove an operation needed for the remaining goal. The agent uses this evidence to reorder the actions, preview each revised call, and execute each separately. By testing these consequences before real execution, the environment gives the agent evidence it can connect to Maya's remaining request while both changes are still possible.

# B ENVIRONMENT SYNTHESIS

We implement the common formulation using sampled databases, service tools connected to separate records, and copied applications. For each implementation, we retain the state and operations needed for trial execution and test whether they behave as expected. The following sections explain these choices, with execution details in Appendix C.

## B.1 STATE AND BEHAVIOR

**SQL databases.** To let agents test SQL on a smaller database, we retain the schema and sample rows while preserving real combinations of values. We use a common set of key values for related tables so that queries can still join their records. Across the 30 local Spider databases, synthesis takes about half a second per database. We accept the sampled database when it has an identical schema and achieves at least 90% query execution, 70% nonempty results when the reference returns rows, 95% result-shape agreement, 90% error-type agreement, and 70% retention of join results. All 30 databases pass at $K \in \{1, 5, 10, 25, 50\}\%$. Across four deliberate damage types per database, the tests accept all 30 undamaged databases and reject all 120 damaged ones. These tests assess whether the reduced database supports the selected queries and detects specified faults. They do not estimate naturally occurring synthesis failures.

**Retail records and tools.** For retail, we sample customers together with their orders and retain the full product catalogue so that tools can resolve item references. We then repair references and restore status classes missing from the sample to exercise the relevant tool paths. Here, $K$ limits customer and order records, while the catalogue and tool code remain available. Connecting that code to the sampled records provides a separate state on which to execute calls. Acceptance requires 95% read agreement, 90% agreement on state changes, complete agreement on rule-check decisions and record formats, and 80% tool coverage. Successful real reads subsequently supply task-specific records, as the case study illustrates.

**Airline policy checks.** For airline, we retain sampled customers and reservations together with the full flight catalogue and tool implementations. Because the tools do not enforce all written policies, we add checks covering cancellation eligibility, flight-change restrictions, baggage allowances, passenger counts, and payment combinations. They omit compensation rules that depend on the conversation. The validation test contains 70 calls. This access to policy is separated from the agent's access in the hidden and given settings.

**Database scale.** We use an IMDb snapshot to examine whether synthesis remains practical for a larger database. From a 12.79 GB SQLite file with 144,188,197 rows, we produce a 14.8 MB sample in about 10 seconds and test it in 190 seconds. At $K = 1\%$ and $10\%$, it passes the revised validation test and matches execution, nonempty-result, and result-shape checks on 12 additional analyst queries. These queries informed revisions to multi-table join handling, so this is a regression check, not a held-out estimate or a test of exact answers.

## B.2 VALIDATION SCOPE

We interpret validation as evidence about the tested calls, since two implementations can pass the same tests yet disagree on an untested input. Moreover, we use these tests to guide repairs, so the resulting agreement is not a held-out estimate of correctness. Recorded outcomes and executable references allow direct comparisons where available. Tests without such evidence can establish consistency, not agreement with an unknown implementation. We report agreement on the specified tests and the faults they detect.

## B.3 ERRORS BEYOND VALIDATION TESTS

To examine what validation can miss, we also measure errors during agent use of an environment generated with Agent World Model (AWM). After repair, the retail environment passes every validation test, yet 49 of 186 calls it predicts will succeed fail during agent use. All five executed calls it predicts will fail instead succeed. The generated airline environment agrees with only 10% of

rule-check tests. These cases show that passing a finite test set does not establish accurate behavior on later calls. AWM receives descriptions and sampled records, whereas EARIS reuses tool code, so these results do not isolate generation quality under equal inputs. The measurements cover only calls later executed and exclude discarded proposals. Complete synthesis and execution records remain in the supplementary material.

# C EXPERIMENTAL DETAILS

The comparisons in Section 5 depend on what information each method receives and how it uses that information. We detail the controlled tasks and follow-up calls first, then explain how we adapt the method and baselines to each benchmark.

## C.1 CONTROLLED TASKS

To vary rules while keeping task generation reproducible, we use a seeded retail engine with typed state, actions, events, and constraints. Each action specifies parameters, a precondition, an effect, and rejection behavior. Events execute in fixed priority order. Task templates generate an initial state, goal, hard constraints, and stopping condition from a seed and index. Success requires reaching the goal before stopping without violating a hard constraint. For external benchmarks, we retain their own environment code.

We increase complexity by successively adding eight rules covering identifier formats, login, paging, reading before writing, amounts in cents, status checks, escalation, and passcodes. Each tested rule count has 60 tasks. Real rejections reveal the violated rule but consume the action budget. Previews do not. Thus rule complexity and budget jointly determine the value of discovering errors before execution.

## C.2 FOLLOW-UP CALLS

After a proposed change succeeds, we test selected follow-up calls to show the agent which operations remain possible. We choose these calls for each application, so the feedback covers a finite set of candidates. Retail tests cancellation, address and payment edits, item edits, returns, and exchanges on the affected order. Arguments use a known address, a recorded payment method, and, where needed, the first recorded item and an available alternative variant. Airline tests supported changes to the affected reservation using available passenger, flight, and payment records. Each candidate runs on a separate copy, so one test does not affect another. Reported availability applies only to the tested calls. A failed item-change candidate does not imply that every item change would fail. Neither this list nor the preview searches complete multi-step plans. The retail case's address restriction follows from the order status, so it applies independently of the replacement address.

## C.3 SERVICE-TOOL INTERACTION

Algorithm 1 details the retail and airline interaction protocol. The history $h$ starts with the task and available instructions. The stored state $\hat{s}$ comes from environment synthesis. To distinguish a new proposal from a request to execute a previewed action, we store its operation and arguments as the pending call $p$. We ignore argument-key order when comparing calls. Reads and messages leave $p$ unchanged. A different write replaces it with a new preview, and any real write clears it. Real reads, messages, and attempted writes count against $B$, including rejected real calls. Previews consume model calls but no real-action budget. The runner also stops on conversation termination, an unrecoverable API or parsing error, or any separately configured model-call limit.

Algorithm 2 expands the preview operation. The transition includes documented rule checks in addition to the available tool behavior. The follow-up candidates use the application-specific operations and arguments defined above. Their results describe those tested calls, rather than all possible future actions.

**Algorithm 1** Agent interaction with service tools

**Input** agent $M$, real environment $E$, synthetic environment $\hat{E}$, state $\hat{s}$
**Input** initial history $h$, real-action budget $B$
Set pending call $p \leftarrow \emptyset$ and real-action count $b \leftarrow 0$
**while** the conversation continues, no stopping condition holds, and $b < B$ **do**
    Obtain and parse the next call $a \leftarrow M(h)$
    **if** $a$ is a read or a message **then**
        Execute $a$ in the real system or send the message to the customer
        Append $a$ and the actual response to $h$, and increment $b$
        Merge records from a successful read into $\hat{s}$, replacing older versions
    **else if** $a$ has the same operation and arguments as $p$ **then**
        Execute $a$ in $E$ and append $a$ and its actual result to $h$
        Increment $b$, including when real execution rejects the call
        **if** the real write succeeds **then**
            Replay the write on stored state $\hat{s}$
        **end if**
        Clear $p$
    **else**
        $(f, k) \leftarrow \textsc{Preview}(\hat{E}, \hat{s}, a)$ ▷ Algorithm 2
        Append $a$ and feedback $f$, marked NOT EXECUTED, to $h$
        **if** $k$ indicates missing state **then**
            Clear $p$ ▷ Repeating the call cannot bypass a missing-state preview
        **else**
            Set $p \leftarrow a$ ▷ Both successful and refused previews may be repeated
        **end if**
    **end if**
**end while**

**Algorithm 2** Preview a proposed write without changing stored state

**Input** synthetic environment $\hat{E}$, stored state $\hat{s}$, proposed write $a$
**if** a record required by $a$ is absent from $\hat{s}$ **then**
    **return** a request to read the missing record, with status MISSING STATE
**end if**
Create disposable state $\tilde{s} \leftarrow \mathrm{copy}(\hat{s})$
Run $(\tilde{s}', \tilde{o}) \leftarrow \hat{T}(\tilde{s}, a)$
**if** the call is refused or fails **then**
    Discard the disposable state
    **return** the refusal or error reason, with status OUTCOME AVAILABLE
**end if**
Initialize feedback $f$ with the result and changes from $\hat{s}$ to $\tilde{s}'$
Add warnings for argument values absent from sampled records, when applicable
**for** each selected follow-up call $c$ on the affected record **do**
    Run $c$ on a fresh copy of $\tilde{s}'$
    Add whether this call succeeds or is refused to $f$
    Discard the follow-up copy
**end for**
Discard $\tilde{s}'$ ▷ Do not merge hypothetical changes into $\hat{s}$
**return** $f$, with status OUTCOME AVAILABLE

### C.4 BENCHMARK IMPLEMENTATIONS

**$\tau$-bench.** Our implementation uses the benchmark's tools and scoring and simulates the customer with its published prompt. The agent emits one JSON action per step, using the model settings in Section 4. We keep length limits on tool descriptions and policy text fixed across comparisons. We allow 30 real actions per task. The customer can end the conversation earlier, and malformed outputs or API failures also stop a run. Previews do not count against this budget. The original service experiments impose no separate limit on preview turns or total model calls. The new controls in Appendix D.2 impose a 60-agent-call limit and report whether it is reached. Tool descriptions are limited to 110 characters each, policy text to 7,000 characters, and tool results to 4,000 characters in this implementation. These fixed limits apply across its comparisons. The hidden setting removes the policy from the agent's prompt but retains it in the checks. The given setting supplies it to both.

**TravelPlanner.** We evaluate two planning settings with the official evaluator. In two-stage planning, the agent gathers flights, hotels, and restaurants before proposing a plan for each of 180 validation queries. In sole planning, we supply this information in advance to separate plan revision from information gathering. Our checker implements the documented budget, accommodation, cuisine, transportation, city-order, restaurant-reuse, and minimum-stay constraints using the evaluator's planning database. It agrees with the evaluator on all 180 plans in the main run. Thus this benchmark tests the use of an accurate checker. Previews provide itemized costs and feasible alternatives for up to five revisions.

To give the agent access to the same constraints as the checker, we add the checked constraints and query parameters (about 6,000 characters) to every prompt. ReAct then reaches the 30-step limit on 47% of tasks, versus 6% without the list. Raising the limit to 45 steps increases its success to 0.317, still below EARIS ($+57/-14$). On 3-day trips, which never exhaust the limit, EARIS solves 46 of 60 versus 25 for ReAct with the list. The additional rule-check run with the list solves 61/180 queries (0.339), versus 75/180 for EARIS. The paired difference is $+30/-16$ ($p = 0.054$, unadjusted). All 180 queries are retained, with no API errors. 85 reach the 30-action limit before producing a plan. This run uses the same two-stage procedure and five-revision limit, omitting suggested alternatives. In sole planning, self-verify with the list reaches 0.417. The checker accepts four plans in the EARIS constraint-list run that the evaluator rejects. The discrepancy arises because the official cuisine check skips the rest of a day when breakfast is in the origin city.

**AppWorld.** To preview application code, we start a separate process from the same task state and replay executed code blocks to synchronize its state with real interaction. Discarding a preview rebuilds the copy from this history. This requires task-state access beyond reconstruction from reads. Feedback includes the code's output, API status, and changed records. Main results use the 168 normal tasks under the final preview format, scored by the benchmark's tests. The 417 challenge tasks used during protocol development are reported separately in Table 7. Appendix D.3 discloses how that format was developed.

**SOPBench.** The 830 tasks cover bank, DMV, healthcare, library, hotel, online market, and university services.

GPT-5.4 simulates the user, and the benchmark's rule-based evaluator scores completion. We use a language model to turn the procedures into executable checks, then correct these checks against reference decisions on recorded actions from published trajectories. Four domains needed no correction. Final checks agree on 99.3% of 3,500 recorded changes. To assess the effect of these corrections, we generate fresh checks without that step. They reach 98.4% agreement and success of 0.559/0.659 with procedures hidden/given, versus 0.554/0.670 with corrected checks ($+45/-41$ and $+61/-70$). The success rates are similar, although the generation instructions were themselves developed on SOPBench. Even with procedures in the prompt, ReAct makes a forbidden change in 20% of tasks, versus 1% for EARIS and the rule check.

**Spider 2.0-lite.** We use the 135 tasks with local databases and official expected results. EARIS executes SQL on a sampled database and shows the result for up to three review rounds. Self-verify reviews without execution. The final scorer uses official expected results, since some released gold queries disagree with them (e.g., `local017`). Because a query can execute successfully yet answer

the wrong question, Table 3 separates execution validity from review of returned results. Within run A, sampled versus real execution gives $+7/-8$. Within run B, EARIS versus model review gives $+11/-7$.

Table 3: SQL execution validity and result review. Task success on 135 Spider queries. Comparisons are paired within each run.

| Run | Feedback used for revision | Success |
|---|---|---|
| A | None (zero-shot) | 0.474 |
| | Model review, no execution | 0.526 |
| | Repair until SQL runs, sampled database | 0.481 |
| | Repair until SQL runs, real database | 0.489 |
| B | None (zero-shot) | 0.452 |
| | Model review, no execution | 0.519 |
| | Repair until SQL runs, sampled database | 0.511 |
| | Review returned results (EARIS) | 0.548 |

**Additional benchmarks.** For BFCL, we copy the benchmark's Python classes and detect state-changing calls through static analysis. We evaluate four categories of 200 entries with its official multi-turn checker. For Gaia2, we copy application state and evaluate 160 scenarios with its judge. For telecom, we use 40 tasks and reconstruct a phone simulator from customer statements because the agent cannot read phone state. This reconstruction adds one model call per customer message, unlike the other implementations.

### C.5 BASELINES

**Rule check.** We use this baseline to test whether enforcing rules is sufficient without previewing state changes. It applies the same checks as EARIS to a state-changing call, using real state. It blocks calls that fail a check and returns the same reason text, or executes immediately after a pass. Blocked calls do not consume the real-action budget. Retail has no additional written-policy checks, so this baseline never refuses. It reaches 0.60 with the policy hidden and 76/115 (0.66) with it given. Seven API refusals in the new given-policy run are counted as failures. These values reflect run variation and the added instructions describing a rule check, not a benefit from rejecting actions. On airline and SOPBench, neither method has a statistically significant success advantage. With GPT-5 on airline, they score 0.47 and 0.44. Despite similar success, the two methods differ in execution. On hidden-policy airline with GPT-5.4, their error-free task rates are 0.86 and 0.98 at 21.8 and 21.3 model calls per task.

**Self-verification.** To distinguish executable feedback from additional model review, self-verify asks the model to predict and review a call without running it in a synthetic environment. Table 4 reports this control for both models. It outperforms EARIS on GPT-5.4 retail, while executable feedback has much larger advantages on TravelPlanner and SOPBench.

Table 4: Task success with execution feedback or model prediction. SV denotes self-verify.

| | GPT-5.4 | | | GPT-5 | | |
|---|---|---|---|---|---|---|
| Benchmark | ReAct | SV | EARIS | ReAct | SV | EARIS |
| Airline | 0.22 | 0.29 | **0.49** | 0.29 | 0.30 | **0.44** |
| Retail | 0.53 | **0.60** | 0.55 | 0.43 | 0.33 | **0.54** |
| TravelPlanner | 0.05 | 0.05 | **0.56** | 0.01 | 0.01 | **0.36** |
| AppWorld (normal) | 0.82 | 0.72 | **0.84** | **0.43** | 0.39 | 0.42 |
| Spider 2.0 | 0.45 | 0.52 | **0.55** | 0.33 | 0.32 | **0.41** |
| SOPBench | 0.22 | 0.23 | **0.55** | 0.13 | 0.21 | **0.43** |

**Task memory.** To evaluate whether previews complement accumulated experience, we run ACE and DC using their published code through our gateway, adapted to each benchmark. After each task, they receive the transcript and official outcome and update a context capped at 6,000 characters. We supply official outcomes in this evaluation, although DC does not require them. Tasks run

in benchmark order. With EARIS, the transcript includes previews, so the learned memory also changes. Table 1 evaluates the complete combination. The GPT-5 AppWorld memory runs maintain one continuous memory per split. Previews reduce success with GPT-5 ACE on AppWorld and GPT-5.4 DC on Gaia2 (Appendix D.4).

### C.6 INFERENCE COST

Previews can improve decisions while adding interaction, so Table 5 reports model calls per task alongside success. These counts exclude synthesis and memory updates. We omit cost where the counter is incomplete for AppWorld self-verify or unavailable for Spider. For airline and retail, EARIS uses fewer calls than self-verify. On SOPBench their costs are similar. AppWorld costs in Table 5 use the normal split, consistently with the main comparison. With GPT-5.4, EARIS raises the error-free task rate from 0.84 to 0.98 on airline and from 0.58 to 0.99 on SOPBench. These improvements describe execution, not task completion or a safety guarantee.

Table 5: Inference cost (GPT-5.4). Mean model calls per scored task. SV denotes self-verify. A dash indicates unavailable data.

| Benchmark | ReAct | SV | EARIS |
|---|---|---|---|
| Airline | 17.6 | 23.4 | 21.3 |
| Retail | 20.9 | 24.4 | 22.4 |
| TravelPlanner | 43.3 | 47.6 | 46.1 |
| AppWorld | 14.2 | – | 19.5 |
| SOPBench | 6.6 | 9.0 | 9.4 |

## D FEEDBACK ANALYSIS

Execution feedback can help only if it exposes information relevant to a decision and the agent uses it correctly. We examine action dependencies, direct reminders, execution instructions, and failures that remain despite accurate previews.

### D.1 ACTION DEPENDENCIES

We design the retail study to distinguish an action dependency the agent can discover by execution from a customer preference it needs to ask about. We constructed 60 tasks, each with a different customer, a `pending` order, and a valid PayPal or card payment. In 30 address tasks, the customer requests an item change and then a new shipping address. Success requires updating the address first. In 30 second-item tasks, the customer reveals another requested item change only when asked or after the first edit. Success requires changing both items in one call. We verified that the reference solutions execute and that the customer's requested ordering fails. Scoring, user simulation, budgets, and preview format remain unchanged. To isolate the value of follow-up feedback, we remove only that feedback and retain the initial call's result and effects.

With policy hidden, ReAct, self-verify, EARIS, and EARIS without that feedback solve 11, 9, 36, and 9 tasks, respectively (60 scored except 59 without that feedback). Their model-call counts are 20.4, 26.5, 21.4, and 22.3 per task. With policy given, they solve 27, 27, 44, and 26 tasks (59 scored for self-verify and EARIS, 60 otherwise). The benefit is concentrated in the hidden-policy address tasks. EARIS solves 26 of 30, versus zero for ReAct and the version without follow-up feedback. For second-item tasks, it solves 10 versus ReAct's 11. A preview can expose a dependency in a proposed action. It cannot reveal a customer preference the agent has not elicited.

### D.2 EXPLICIT REMINDERS

The address dependency can also be supplied directly in language. We test this alternative on the same 30 tasks in seven fixed conditions (210 episodes), without changing task scripts or scoring. Every agent receives the full policy. The reminder states that item edits change the status to `modified`, preventing address edits, and asks the agent to check for address changes and perform them first. It supplies the tested ordering dependency directly, is identical across tasks, and uses no task-specific reference solution.

We compare full previews with previews that omit follow-up feedback, then add the reminder to either that reduced feedback or direct execution. These comparisons test whether an explicit explanation supplies the dependency otherwise revealed by trial execution. To examine state coverage separately, another condition gives the preview all initial records instead of a 10% sample, keeping tools, policy, and feedback fixed. Finally, we repeat full previews and reminded direct execution at medium reasoning to examine whether additional model reasoning changes the comparison. The customer remains GPT-5.4 with reasoning off throughout. All conditions allow 30 real actions and at most 60 agent calls. Output limits are 3,000 tokens with reasoning off and 6,000 at medium reasoning. No run hits the call limit or has an API error. All 210 episodes enter the results. The constructed environment passes its recorded checks before these runs. The comparisons use existing designed tasks and do not test transfer to unseen dependencies.

Table 6: Explicit reminders on retail address tasks. Agent calls and tokens are means per task. Time is median seconds per task.

| Reasoning | Method | Success | Calls | Tokens (k) | Time (s) |
|---|---|---|---|---|---|
| Off | EARIS | 20/30 | 13.7 | 48.6 | 34.4 |
| Off | Without follow-up feedback | 2/30 | 12.7 | 43.3 | 33.3 |
| Off | Without follow-up feedback + reminder | 25/30 | 12.7 | 42.4 | 30.5 |
| Off | Direct execution + reminder | 28/30 | 10.5 | 32.4 | 25.9 |
| Off | EARIS, all initial records | 15/30 | 13.1 | 45.2 | 34.7 |
| Medium | EARIS | 30/30 | 11.7 | 40.4 | 41.4 |
| Medium | Direct execution + reminder | 30/30 | 9.8 | 31.6 | 40.5 |

Full feedback outperforms the version without follow-up feedback by $+18/-0$. Adding the reminder to reduced feedback gives $+23/-0$. Both differences survive Holm correction ($p < 0.001$) over six comparisons. These compare full previews against each alternative at the same reasoning setting and test adding the reminder to reduced feedback. Full previews versus reminded direct execution give $+1/-9$ with reasoning off (uncorrected $p = 0.021$, adjusted $p = 0.086$). Both solve all 30 tasks at medium reasoning. Thus execution and explicit advice can supply the same useful ordering information. Because both direct-execution comparisons include it, the medium-reasoning result does not isolate reasoning strength alone. Using all initial records gives 15/30 versus 20/30 for sampled state ($+10/-5$ in favor of sampling, $p = 0.30$). This small comparison does not establish a benefit from either state size.

We measure calls, tokens, and elapsed time in Table 6 because similar call counts can conceal differences in computation. Agent tokens include input and output, excluding the fixed customer simulator. Elapsed time measures concurrently executed episodes and excludes synthesis. Reminded direct execution uses fewer agent calls and tokens than full previews in both reasoning settings. The reminder and reasoning setting change the intervention, so this is a comparison of performance and cost, not a strict equal-compute test. Elapsed times depend on the shared API and concurrent execution. They are not deployment latency estimates.

### D.3 EXECUTION PROTOCOL

When inspecting GPT-5.4 failures on AppWorld's challenge split, we found that agents sometimes treated preview outputs as evidence of completed execution. We therefore revised the interaction format to make this distinction explicit. The final format holds a previewed block until the agent executes or discards it, labels feedback as not executed, reports changed records, and bypasses known operations requiring secrets. It also requests a read when state is unknown and discards pending blocks after repeated unrelated replies.

Success rose from 0.667 with the initial format to 0.779 with the final format, versus ReAct's 0.727 ($+42/-20$, $p = 0.007$). The intermediate formats were developed and evaluated sequentially on these same 417 tasks. This improvement is not an independent test or a controlled change to one component. The normal split and GPT-5 runs were unused for this protocol development and show small or absent gains. The agent can override a refusal by repeating the call. Known secret-dependent operations can execute without a preview.

Table 7: AppWorld task success by split. Challenge tasks informed protocol development.

| | | ReAct | | ACE | | DC | |
|---|---|---|---|---|---|---|---|
| Model | Split | alone | +EARIS | alone | +EARIS | alone | +EARIS |
| GPT-5.4 | Normal | 0.821 | 0.839 | 0.786 | 0.833 | 0.821 | 0.839 |
| GPT-5.4 | Challenge (development) | 0.727 | 0.779 | 0.767 | 0.794 | 0.763 | 0.763 |
| GPT-5 | Normal | 0.435 | 0.417 | 0.464 | 0.411 | 0.363 | 0.315 |
| GPT-5 | Challenge (development) | 0.336 | 0.324 | 0.391 | 0.333 | 0.173 | 0.161 |

### D.4 FAILURE MODES

Table 8 reports settings in which accurate execution feedback is insufficient, using the final method throughout. BFCL's informative tool errors already supply much of the useful feedback, and previews do not improve success in any category. On Gaia2, the final copy produces only one false accept among 696 executed previews, but task success remains close to ReAct. Many failures involve required actions that the agent never proposes. DC alone reaches 0.388, versus 0.275 with EARIS ($+5/-23$). The combination reaches the step limit in 34 scenarios versus 21, making added turns one source of loss.

Telecom illustrates the cost of incomplete state access. With the manual hidden, EARIS versus ReAct yields $+8/-2$ ($p = 0.109$) on 40 tasks, but does not outperform self-verify. Reconstructing phone state increases calls from 10.5 to about 34 per task. With the manual given, both preview methods fall below ReAct.

Table 8: Task success on additional benchmarks (GPT-5.4). SV denotes self-verify.

| Benchmark / setting | ReAct | SV | EARIS |
|---|---|---|---|
| BFCL, base | 0.600 | 0.275 | 0.575 |
| BFCL, missing parameters | 0.435 | 0.575 | 0.425 |
| BFCL, missing functions | 0.505 | 0.470 | 0.465 |
| BFCL, long context | 0.475 | 0.490 | 0.475 |
| Gaia2 | 0.287 | 0.319 | 0.306 |
| Telecom, manual given | 0.600 | 0.525 | 0.525 |
| Telecom, manual hidden | 0.100 | 0.300 | 0.250 |

## E ROBUSTNESS ANALYSIS

We vary state coverage, transition accuracy, task seeds, and agent models to distinguish reliable feedback from reliable use of that feedback.

### E.1 STATE COVERAGE AND TRANSITION ERRORS

We vary state coverage and transition behavior separately in Figure 4 to understand which errors can be resolved during interaction. With tool code fixed and real reads supplying missing records, success remains near 0.73 as the initial sample varies from 1% to 50%. At 1%, a missing `pending` order leaves a validation probe without a target, lowering agreement to 0.89 and failing the checks. The diagnostic agent run nevertheless succeeds after reads restore state. We use 60 tasks with eight hidden rules and a separate ReAct run for each condition. We deliberately retain environments that fail validation so that we can measure how these errors affect agent use.

Omitting the newest rule gives success of 0.77 because the real environment's error reveals it. Inverting the cents rule instead lowers success to 0.33 and probe agreement to 0.80, also failing the checks. These diagnostic runs show the consequences of using environments that fail validation checks. Real reads can fill missing records, but state updates do not repair an incorrect transition function.

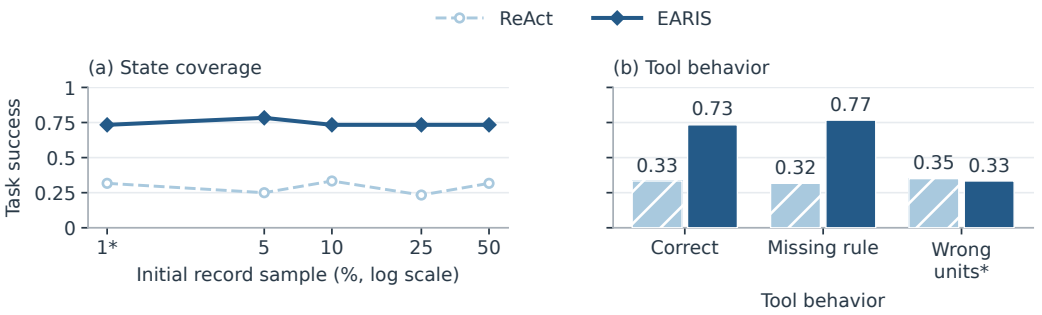


Figure 4: State coverage and transition accuracy (GPT-5.4). (a) Initial sample size on a log scale. (b) Changes to tool behavior at a 10% sample. Asterisks mark environments that fail validation tests.

### E.2 Runs and models

**Repeated runs.** We repeat the controlled retail study with a second task seed to check whether its gains depend on the sampled tasks. The benefit under additional rules recurs. At eight rules, ReAct and EARIS score 0.38 and 0.77, respectively, compared with 0.87 and 0.88 without hidden rules. A second GPT-5.4 SOPBench run also retains the gain, with 0.224 versus 0.554 success and $+294/-20$ paired wins and losses. ReAct and EARIS agree with their first-run task outcomes in 93% and 90% of cases, respectively. The supplementary material records the remaining repeated runs, including the incomplete retail run with 104 of 115 tasks scored.

**Model dependence.** We also evaluate GPT-5.4-mini to examine how a different model uses the same feedback. It improves from 0.033 to 0.506 on TravelPlanner with planning information supplied, but declines from 0.530 to 0.494 on AppWorld normal. In controlled retail with eight rules, it changes from 0.33 to 0.38 at 12 actions and from 0.45 to 0.38 at 20 actions. Increasing its budget therefore does not reproduce the GPT-5.4 pattern in Figure 3b.

**Response to refusals.** For the open-weight comparison, we run Qwen3-14B in bf16 with vLLM, greedy decoding, and thinking disabled. We keep GPT-5.4 as the simulated user so that only the agent model changes. On SOPBench, EARIS raises success from 0.035 to 0.317 ($+245/-11$) and error-free tasks from 0.39 to 0.97. Airline and retail improve only from 0.20 to 0.28 ($+8/-4$, $p = 0.39$) and 0.22 to 0.26 ($+7/-3$, $p = 0.34$). Qwen executes refused calls in 53 of 97 airline and 173 of 270 retail refusals. These repeats account for all failed real changes under EARIS. GPT-5.4 does so in 1 of 46 and zero of 101 refusals. The $\tau$-bench preview invites the agent to repeat or change the call, whereas the SOPBench refusal does not. This wording was not adapted to Qwen. Fewer than 0.4% of Qwen calls are unparseable. Five retail tasks remain filtered by the model gateway after up to three retries and are excluded (110 of 115 paired tasks).

## F Statistical analysis

### F.1 Run selection

We select runs of the final method for the main tables and use complete memory streams so that each memory-based agent's experience follows a single task sequence. For AppWorld, we use the normal split because challenge tasks informed development. We recompute statistics from saved outcomes and check model identity, task identifiers, completeness, and errors against checkpoints. Source hashes record which files were used. Across methods and models, scored counts range from 44 to 50 on airline and 109 to 115 on retail. All other main benchmarks are complete. Per-method counts are retained with the supplementary results.

**Unscored tasks.** The $\tau$-bench loaders exclude recorded errors and API refusals. Counting unscored tasks as failures changes GPT-5.4 airline ReAct/EARIS from 0.224/0.490 to 0.220/0.480, and retail from 0.526/0.553 to 0.522/0.548. ACE and DC with EARIS on airline become 0.400 and

0.420, versus 0.260 and 0.180 alone. The gains retain their direction under this treatment, although the excluded tasks may still differ systematically from those scored.

### F.2 Paired comparisons

We report paired wins and losses as $+w/-l$ and compute table differences before rounding. We use exact two-sided sign tests and apply Holm correction to account for the twelve comparisons between EARIS and ReAct in Table 1. Unadjusted sign-test $p$-values are below $10^{-10}$ for TravelPlanner and SOPBench with both models. For GPT-5.4, they are 0.002 (airline), 0.70 (retail), 0.58 (AppWorld normal), and 0.011 (Spider). For GPT-5, they are 0.039, 0.108, 0.74, and 0.035, respectively. Gains survive Holm correction on TravelPlanner and SOPBench with both models and on airline with GPT-5.4, for five comparisons in total. Figure 5 shows paired effect sizes with task-bootstrap intervals (10,000 resamples, seed 20260920). Paired comparisons use only tasks scored by both methods, so their counts can differ from individual main-table entries. Paired counts for GPT-5.4/GPT-5 are 49/49 on airline, 113/114 on retail, 180/180 on TravelPlanner, 168/168 on AppWorld normal, 135/135 on Spider, and 830/830 on SOPBench. The intervals are conditional on these saved runs.

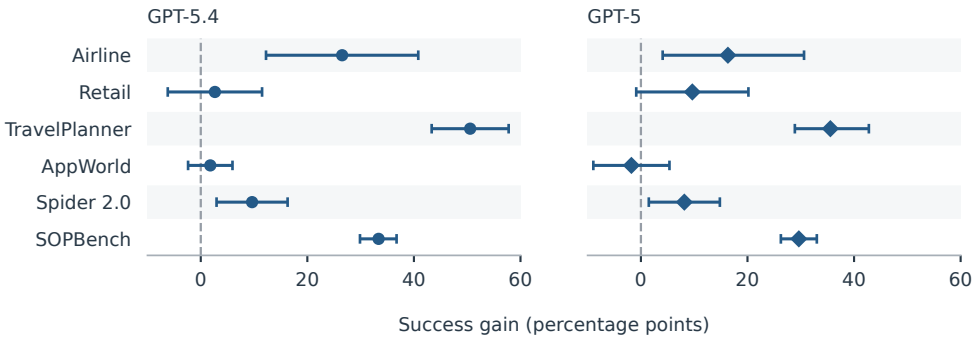


Figure 5: Success gains over ReAct on shared tasks. Error bars show 95% paired task-bootstrap intervals. Zero denotes no gain.

**Shared templates and memory.** Table 9 reports paired task outcomes on AppWorld normal. To account for its three tasks per template, we also resample the 56 templates. The ReAct intervals include zero for both models. Neither bootstrap captures dependence from evolving ACE or DC memory, or variation across new agent runs.

Table 9: Paired outcomes on AppWorld normal. Wins/losses count tasks helped or hurt by adding EARIS.

| Model | Base method | Wins/losses |
|---|---|---|
| GPT-5.4 | ReAct | 8/5 |
| GPT-5.4 | ACE | 13/5 |
| GPT-5.4 | DC | 10/7 |
| GPT-5 | ReAct | 17/20 |
| GPT-5 | ACE | 13/22 |
| GPT-5 | DC | 13/21 |

### F.3 Evaluation checks

Before reporting results, we checked the evaluation interfaces as well as the saved outcomes. We corrected customer roles in $\tau$-bench, used application tests in AppWorld, and scored Spider against official expected outputs. Airline episodes affected by malformed-payment handling were rerun. Interrupted GPT-5 memory streams were replaced by complete runs. Validation tests cover missing columns, invalid formats, and join failures. Ineffective damage cases were excluded. Reference-based SQL and SOPBench development, and challenge-based AppWorld development, limit claims about generalization as disclosed above.