ACTION VERIFICATION THROUGH CONJECTURE AND REFUTATION FOR LANGUAGE MODEL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language model agents increasingly act on users' behalf by manipulating external environments through tools, often from high-level instructions that leave critical task- and environment-specific context unstated. To prevent resulting misaligned actions, recent work verifies proposed actions before execution, largely by establishing whether the conditions supporting them hold. We argue that this confirmation-oriented view leaves a fundamental blind spot: a wrong action often passes every check that asks whether it is right. This motivates a different question for verification: rather than asking what evidence supports an action, can the agent find evidence that refutes it? We pursue this direction with BLACKSWAN, a refutation-oriented, expect-suspect-test procedure that predicts the intended effect of each state-mutating action, formulates specific and empirically testable hypotheses for how it could be wrong, and tests them using existing observations or additional non-mutating actions, executing the action only if none survives testing. On challenging AppWorld tasks, BLACKSWAN improves success rate over the strongest baseline by 14.9 and 25.2%p with and without the backbone model's thinking mode, respectively. Even without thinking or in-context demonstrations, it outperforms stronger baselines equipped with them. The gains also transfer beyond AppWorld, with BLACKSWAN improving WebShop performance by 12.7%p. More broadly, our results suggest that language model agents have an underused capacity to recognize and self-correct misaligned actions, which can be systematically elicited through active conjecture and refutation before execution.

1 INTRODUCTION

A central promise of language model agents is that they can complete delegated tasks autonomously: on behalf of users, they perceive and manipulate user environments through tools (Yao et al., 2023; Wang et al., 2024b; 2025b). Yet, in practice, users typically specify high-level goals, leaving critical information implicit in their environments (Shah et al., 2019; Trivedi et al., 2024). Worse, the user-specific environment states and dynamics lie beyond an agent's parametric knowledge (Chae et al., 2025). Faced with this information gap, agents may take actions that deviate from the user's actual intent (Vijayvargiya et al., 2026b; Shayegani et al., 2026). Unlike a flawed chat response, which can simply be ignored, a flawed action changes the environment itself, turning an incorrect inference about unstated context into a potentially hard-to-reverse consequence.

One natural response to this information gap is to equip agents with richer knowledge of their environments (Fu et al., 2024; Chen et al., 2024; Wang et al., 2025c; Zhang et al., 2026; Ouyang et al., 2026; Chen et al., 2026b). Yet user environments are open-ended: it is difficult to know in advance which facts will matter to actions that arise later in a task. Such up-front knowledge acquisition therefore faces a basic question of sufficiency—when has the agent learned enough?

More recent work shifts the focus from understanding the environment in general to verifying individual proposed actions on the fly (Fang et al., 2025; Liu et al., 2026; Ning et al., 2026; Chen et al., 2023; Jiang et al., 2026). These methods intervene before execution and establish whether task- and environment-specific conditions for the action hold. This reflects a *confirmation-oriented* view of action readiness: ask what must be established for the proposed action to proceed, then seek evidence for those conditions. But a wrong action can pass every check that asks whether it is right.

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.

For example, asked to "split last night's dinner bill among the attendees", an agent may confirm the bill amount and the guest list on the event invitation, yet miss that one invitee canceled by message.

In this work, we argue for a *refutation-oriented* view (Popper & Hudson, 1963) of preemptive action verification. The central question is not merely how much evidence verification should gather, but in which direction it should look: should it seek evidence for an action, or evidence *against* it? The confirmation-oriented view takes the former, admitting an action once the conditions required to support it are established. In contrast, the refutation-oriented view takes the latter: it asks how the action *could be wrong*, constructs potential falsifiers, and admits the action only if none survives empirical scrutiny. The action is thus corroborated by withstanding attempted refutation, rather than confirmed by accumulated support.

To operationalize this view, we propose BLACKSWAN[1], which subjects each state-mutating action to an expect-suspect-test procedure: the action executes only if it withstands the attempt to refute it; otherwise, it is withheld, and the agent drafts a replacement in light of the refuting evidence. Specifically, **expect** predicts what the proposed action itself would accomplish toward the user's task, yielding an action-local claim that can be challenged without conflating failures of the current write with unfinished parts of the task. Given this claim, **suspect** raises specific, empirically testable hypotheses about how it could be wrong. We keep this stage separate from testing, following the principle that hypotheses must not depend on the data used to test them (Huang et al., 2025): if hypotheses were formulated while evidence is gathered, the evidence could shape the very hypotheses it is then used to adjudicate. Finally, **test** tries to refute the action by seeking the observations these fixed hypotheses predict, drawing on those already in the trajectory or acquiring new ones through non-mutating actions as needed.

Our experiments with BLACKSWAN support the promise of refutation-oriented verification. On the most challenging tasks of AppWorld (Trivedi et al., 2024), BLACKSWAN outperforms the strongest baseline by 14.9%p and 25.2%p in success rate with and without the backbone model's thinking mode, respectively. Even without thinking, BLACKSWAN outperforms every baseline with thinking by 11.7%p on these tasks, suggesting that the direction of verification matters more than the amount of reasoning. Moreover, even without any in-context demonstration, BLACKSWAN still outperforms every baseline given three by 9.3%p. Beyond AppWorld, with only its prompts made domain-neutral, BLACKSWAN improves over the same agent without verification on WebShop (Yao et al., 2022) by 12.7%p. More broadly, our results suggest that language model agents have an underused capacity to recognize and self-correct misaligned actions. Our approach elicits this capacity through a process of falsification: active conjecture and refutation before execution. Our contributions are:

- We recast preemptive action verification as a question of direction: whether an agent should seek evidence *for* a proposed action or actively seek evidence *against* it.
- We introduce BLACKSWAN, an expect-suspect-test procedure that frames each proposed write as a conjecture and executes it only after attempted refutation.
- We suggest that the corrective capacity of agents can be systematically elicited at test time, across models and environments, even under unfamiliar and task-specific conditions.

2 RELATED WORK

Task-level information acquisition. One line of work equips agents with missing information before acting. A line of approaches acquire information from the environment: they rely on a dedicated exploration phase (Chen et al., 2026b), leverage world knowledge models (Qiao et al., 2024), learn from synthesized interactions (SU et al., 2025), or accumulate reusable artifacts such as manuals (Chen et al., 2024), guidelines (Fu et al., 2024), workflows (Wang et al., 2025c), and playbooks (Zhang et al., 2026). Another line of approaches resolve missing information through clarification queries (Qian et al., 2024; Zhang et al., 2024; 2025; Kobalczyk et al., 2025; Vijayvargiya et al., 2026a). Yet both approaches acquire information before a concrete action exists and therefore cannot target the evidence needed to test a particular write.

[1]The name alludes to Popper's classic black-swan example: observing more white swans cannot verify that all swans are white, whereas a single black swan can refute the claim.

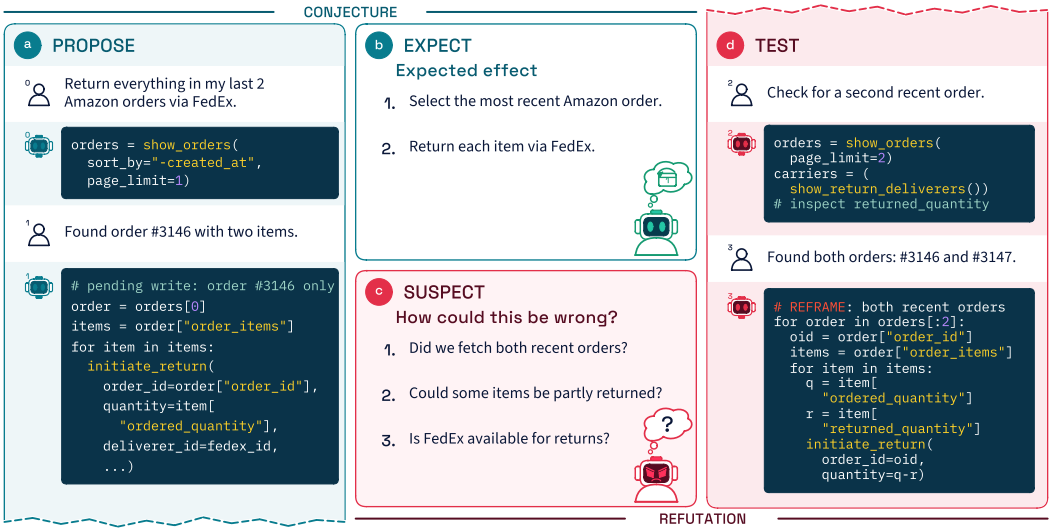


Figure 1: The verification process of BLACKSWAN. (a) **propose** drafts a pending action. (b) **expect** states its intended outcome. (c) **suspect** lists specific ways the action could be wrong. (d) **test** checks these suspicions with read-only probes. If any is confirmed, the action is withheld and reframed using the refuting evidence. The test stage continues directly from the proposal-stage context window.

Trajectory-based verification and correction. A second line of work intervenes during execution, examining actions against the trajectory. InferAct (Fang et al., 2025) infers the task the trajectory has pursued and approves actions consistent with the user's instruction; our **expect** instead states what the pending write should do, as a claim to be falsified rather than a basis for approval. DeAction (Ning et al., 2026) combines alignment checks with outcome analysis to detect misaligned actions before execution. Other methods revise beliefs during execution, reflecting on the current state relative to the goal (Kim et al., 2025) or comparing expected outcomes with observations once an action has taken effect (Wang et al., 2026). Devil's Advocate (Wang et al., 2024a) anticipates failures before execution but checks them only afterward and recovers by backtracking, presuming reversibility. These methods reason over trajectories produced under the very interpretation they aim to correct, and so rarely find evidence against it before execution.

Action-level evidence acquisition. A third line of work obtains evidence beyond the trajectory before an action executes. Agents may ask the user mid-task (Chen et al., 2023; Wang et al., 2025a; Suri et al., 2026), which inherits the limits of asking noted above, or simulate candidate actions with learned world models (Chae et al., 2025), where evidence remains predicted rather than observed. Other methods probe the environment directly. EnvProbe (Song & Cai, 2026) allocates a probing budget over uncertain or action-critical beliefs, while AGrail (Luo et al., 2025) performs action-specific safety checks based on predefined criteria. ACCORD (Jiang et al., 2026) intercepts state-changing actions and uses read-only probes to verify their assumptions about constraints, entities, and environment states. BLACKSWAN shares this access to evidence but differs in how the target of verification is formed: it first specifies what the action should achieve and what would falsify that claim, and only then consults the environment. Prior methods gather evidence to support execution or check predefined conditions; BLACKSWAN acquires evidence to challenge the agent's own interpretation before a write occurs.

3 CONJECTURE AND REFUTATION FOR PREEMPTIVE VERIFICATION

Setup. Given a natural-language task instruction u, an agent interacts with an environment through a set of available APIs. At each step $t \geq 1$, the agent generates an action $a_t \sim \pi(\cdot \mid c_{t-1})$ based on its previous context $c_{t-1} = (u, a_1, o_1, \ldots, a_{t-1}, o_{t-1})$, with $c_0 = (u)$, and receives an observation o_t, so that $c_t = c_{t-1} \parallel (a_t, o_t)$. We focus on agents that act through executable code (Wang et al., 2024b): each action a_t is a piece of code executed in a persistent interpreter, and o_t is the output

of its execution (e.g., printed results or errors). A single action may invoke multiple APIs and bind their results to variables, which can be reused both later in the same code and in subsequent actions.

Overview. We propose BLACKSWAN, a refutation-first approach for preemptive action verification. If an action a_t invokes at least one state-mutating API (hereafter, a write), its execution is withheld and the action is subjected to our conjecture-and-refutation process; otherwise, it is executed as usual. BLACKSWAN is inspired by the Popperian view of conjecture and refutation. Rather than asking if a_t appears correct, we ask what observable facts would show a_t to be wrong, and test those possibilities before execution. Figure 1 illustrates our pipeline: **expect** makes explicit what a_t claims to accomplish (Section 3.1); **suspect** identifies concrete ways in which that claim could be wrong (Section 3.2); and **test** determines whether those suspicions are supported by observations of the environment (Section 3.3).

3.1 EXPECT: MAKING THE ACTION'S INTENT EXPLICIT

Given the previous context c_{t-1} and a proposed write a_t, the language model predicts what executing a_t would accomplish toward fulfilling the user's task u:

$$\kappa_t \sim \pi(\cdot \mid c_{t-1}, a_t) \,. \tag{1}$$

This prediction κ_t makes explicit the conjecture behind the proposed action: "executing a_t should produce κ_t in a way that advances the user's task." It then serves as the claim that the subsequent **suspect** stage to challenge.

To make refutation meaningful, we place two constraints on the prediction. First, κ_t is generated before any new observation is gathered, so that it cannot be shaped to fit the very evidence used to test it. Second, κ_t is kept action-local: it is intended to capture effects attributable to the proposed action itself, rather than broader outcomes of the overall task execution.

3.2 SUSPECT: CONSTRUCTING POTENTIAL FALSIFIERS

Given the previous context c_{t-1}, the proposed write a_t, and its prediction κ_t, the language model identifies concrete ways in which the conjecture could fail, producing a set of suspicions

$$S_t = \{s_1, \ldots, s_k\} \sim \pi(\cdot \mid c_{t-1}, a_t, \kappa_t) \,, \tag{2}$$

where each s_i describes a condition under which executing a_t could lead to an effect inconsistent with κ_t or with the user's task u. These suspicions serve as potential falsifiers, rather than conclusions that the action is wrong. The subsequent **test** stage determines from observations whether any of them actually holds.

Consequential and testable suspicions. Our goal is not to surface every possible error in the prior context c_{t-1} or the proposed write a_t, but only those that **(1)** could affect the pending write and **(2)** can be empirically tested against observations. A suspicion may concern any part of c_{t-1} or a_t, but its effect must *reach* the write action *through some concrete code element* z_i of a_t, such as local variables and API parameters. To operationalize these requirements, we require the model to emit each suspicion s_i together with the corresponding element as a pair (s_i, z_i), encouraging it to reason about the provenance of z_i and the ambient state surrounding the associated API call (Figure 2). As a result, the suspicions focus on consequential failures rather than on the potentially unbounded set of concerns that never affect the proposed action, while remaining amenable to empirical testing.

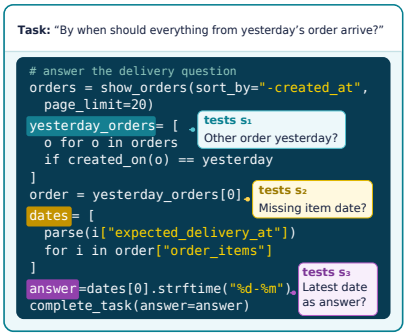


Figure 2: Code-grounded suspicions identify testable assumptions that may change the final answer.

Preregistered suspicions. We deliberately separate **suspect** from **test**, rather than letting the model raise and check concerns in a single interleaved loop. When the two are combined, the model tends

to raise only concerns it expects to resolve easily (Huang et al., 2025), so the hypotheses end up shaped by the very observations meant to test them. Instead, S_t is generated before any probing, from c_{t-1}, a_t, and κ_t alone as in (2): at this stage the model reasons only over the proposed code and what has already been observed, without issuing any new action. Much like preregistration in empirical science, where hypotheses are committed before data collection, this fixes what would count as a refutation before any evidence is gathered. It also holds **test** accountable: since **test** must reach a verdict on every suspicion in S_t, each registered suspicion is explicitly confirmed, dismissed, or left unresolved, and none can be silently dropped or reframed once probing begins.

3.3 TEST: TRYING TO REFUTE THE CONJECTURE

Given the previous context c_{t-1}, the proposed write a_t, and the potential falsifiers S_t, the language model attempts to resolve each suspicion $s_i \in S_t$ using observations from the environment. Rather than judging a suspicion from the proposed code and reasoning alone, the model must resolve each s_i from observable evidence: it can revisit observations already available in c_{t-1} or acquire new ones as needed by issuing non-mutating actions.

Probing and verdict. The pending write a_t is appended to c_{t-1} as an unexecuted action, after which the agent receives an observation o_t that presents S_t as suspicions to resolve, yielding $c_t = c_{t-1} \parallel (a_t, o_t)$. Concretely, o_t is a user-role message of the form "I'm suspicious of executing this action for the following reasons: S_t. Please verify whether each of them is the case." The agent then continues from c_t. At each probing step $k \geq 1$, it either issues a non-mutating action $a_{t+k} \sim \pi(\cdot \mid c_{t+k-1})$, whose observation o_{t+k} extends the context as $c_{t+k} = c_{t+k-1} \parallel (a_{t+k}, o_{t+k})$, or emits a verdict $v_t \colon S_t \to \{\text{true}, \text{false}, \text{deferred}\}$ that resolves each suspicion as follows:

$$v_t(s_i) = \begin{cases} \text{true} & \text{if observations confirm } s_i \text{ ,} \\ \text{false} & \text{if observations dismiss } s_i \text{ ,} \\ \text{deferred} & \text{otherwise ,} \end{cases} \tag{3}$$

where deferred covers suspicions that probing can neither confirm nor dismiss, typically because they depend on the outcome of executing a_t itself (e.g., a transient network failure during an API call). Testing ends when the agent emits v_t at some step K. Unlike the probing actions, v_t is not appended, leaving the context at c_{t+K-1}.

Acting on the verdict. The verdict determines whether the pending write is released as follows:

$$\forall s_i \in S_t \colon\ v_t(s_i) = \text{false} \vee v_t(s_i) = \text{deferred} \quad \Rightarrow \quad \text{execute } a_t, \tag{4a}$$

$$\exists s_i \in S_t \colon\ v_t(s_i) = \text{true} \quad \Rightarrow \quad \text{withhold } a_t \text{ and draft } a'_t. \tag{4b}$$

In (4a), the proposed action a_t has survived every refutation attempt; suspicions labeled deferred do not block execution because they cannot be resolved without observing the effect of a_t. Thus, a_t is reissued as the next action, $a_{t+K} = a_t$, this time executed. In (4b), a single confirmed suspicion suffices to withhold a_t, as the action is no longer consistent with the conjecture it was intended to fulfill. The agent drafts a replacement a'_t from c_{t+K-1} with v_t provided as additional context (e.g., as an assistant prefill). If a'_t is another write action, it undergoes the same expect-suspect-test procedure. In either case, the context then grows as before, $c_{t+K} = c_{t+K-1} \parallel (a_{t+K}, o_{t+K})$.

4 EXPERIMENTS

Unless otherwise stated, all methods use the official FP8 checkpoint of Qwen3.5-27B (Qwen Team, 2026) as the underlying model, following ACCORD (Jiang et al., 2026).

Benchmarks. Our primary benchmark is AppWorld (Trivedi et al., 2024), where an agent with access to 457 APIs across 9 simulated applications completes user requests by discovering and invoking the ones it needs. AppWorld captures key challenges of assistant deployment: agents must discover persistent user data, avoid unintended state changes, and account for irreversible writes. Tasks are grouped into scenarios of three similar variants that differ in user and environment states. On the normal (168 tasks) and challenge (417 tasks) test splits, we report the two official metrics:

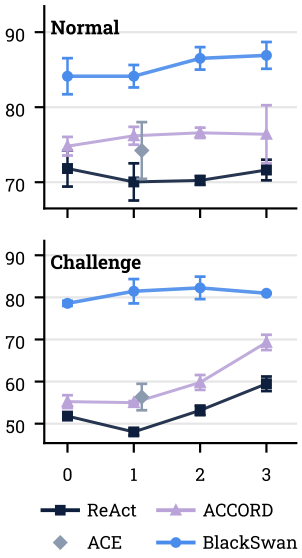


Table 1: Results on the normal and challenge splits of AppWorld, with all methods using a single in-context task examples. Mean ± standard deviation over three runs. We report results with the backbone model's thinking on (✓) and off (✗). †Reported in the original paper; its number of in-context demonstrations is unspecified.

	Method	Test (normal) TGC	Test (normal) SGC	Test (challenge) TGC	Test (challenge) SGC
✓	REACT	72.4 ±1.9	56.5 ±3.7	58.4 ±0.8	43.2 ±2.5
	ACE	80.2 ±3.8	68.2 ±4.8	65.5 ±4.2	49.6 ±4.3
	ACCORD†	81.5	67.9	69.8	53.2
	ACCORD	79.4 ±1.8	65.5 ±5.7	63.5 ±1.7	48.4 ±1.5
	BLACKSWAN	**88.3** ±3.0	**76.8** ±5.4	**84.7** ±1.3	**72.4** ±2.2
✗	REACT	70.0 ±2.5	48.2 ±5.4	48.0 ±0.8	33.6 ±1.1
	ACE	74.2 ±3.8	57.7 ±5.7	56.3 ±3.1	41.8 ±3.4
	ACCORD	76.2 ±1.2	59.5 ±2.7	55.0 ±0.4	44.8 ±2.3
	BLACKSWAN	**84.1** ±1.5	**69.6** ±1.8	**81.5** ±2.9	**65.0** ±4.1

Figure 3: TGC (vertical, %) versus the number of in-context demonstrations (horizontal, 0 to 3), with the backbone model's thinking mode off, on the normal (top) and challenge (bottom) splits. Error bars show the standard deviation over three runs.

task goal completion (TGC), the fraction of tasks that pass all of their unit tests (8 on average, up to 22), and *scenario goal completion* (SGC), the fraction of scenarios in which all three tasks pass. We additionally evaluate on WebShop (Yao et al., 2022) and ALFWorld (Shridhar et al., 2021) to show that BLACKSWAN generalizes to environments with different action spaces.

Baselines. REACT (Yao et al., 2023) interleaves reasoning with code actions (Wang et al., 2024b) and executes every action immediately, serving as the unguarded reference on which BLACKSWAN and the other baselines are built. ACE (Zhang et al., 2026) enriches the agent's knowledge of its environment with a playbook of strategies and pitfalls distilled from training tasks.[2] Comparing BLACKSWAN against it tests whether knowledge supplied in advance suffices, or whether erroneous mutations also stem from task-specific facts that it cannot anticipate. ACCORD (Jiang et al., 2026) admits each write once revisited or newly probed observations support its required conditions.[3] As an instance of the confirmation-oriented view, it isolates the direction in which verification looks: ACCORD seeks evidence for the write, whereas BLACKSWAN seeks evidence against it.

4.1 MAIN RESULTS

Overall performance. Table 1 shows that BLACKSWAN achieves the highest performance on AppWorld, regardless of whether the underlying model's thinking mode is on or off. With thinking on, it improves over the strongest baseline in each column (underlined) by 6.8 TGC and 8.6 SGC points on the normal split, and by 14.9 and 19.2 points on the challenge split. Without thinking, the corresponding margins are 7.9 and 10.1 points on the normal split, and 25.2 and 20.2 points on the challenge split. As ACE and ACCORD represent the two prevailing responses to the information gap, namely knowledge acquired in advance and confirmation-oriented verification, these margins sug-

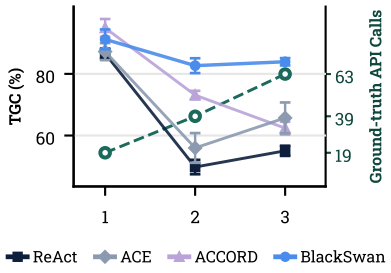


Figure 4: AppWorld results by difficulty, from 1 (easiest) to 3 (hardest).

[2]We run its released implementation (https://github.com/ace-agent/ace).

[3]As no implementation of ACCORD was publicly available as of September 25, 2026, we reimplemented it from the paper, resolving unspecified design choices as consistently with its description as possible (Section C).

Table 2: **(a)** More results on AppWorld with Qwen3-4B-Thinking-2507, MiniMax M2.7, and GLM-5.3-Flash as backbones. **(b)** Mean ± standard deviation of performance on WebShop andALFWorld. The dash (–) indicates a metric not reported in the paper. N/A indicates a metric not defined by the benchmark. We note that, unlike ours, ACCORD on ALFWorld uses a dedicated procedure.

(a) AppWorld results with three more backbone models.

	Qwen3-4B		MiniMax		GLM	
Method	TGC	SGC	TGC	SGC	TGC	SGC
Test (normal)						
REACT	23.2	14.3	61.3	41.1	83.9	67.9
ACCORD	14.9	3.6	71.4	50.0	82.1	66.1
BLACKSWAN	**41.7**	**30.4**	**77.4**	**64.3**	**91.1**	**82.1**
Test (challenge)						
REACT	12.2	4.3	53.0	35.3	77.5	55.4
ACCORD	8.6	2.2	63.1	46.0	76.7	56.8
BLACKSWAN	**27.6**	**13.7**	**73.9**	**53.2**	**83.5**	**66.9**

(b) Two further environments.

	Performance	
Method	Score	SR
WebShop (test 500)		
REACT	55.4 ±1.0	38.1 ±0.8
Harness-R1	–	42.0
BLACKSWAN	**72.8** ±0.5	**50.8** ±1.2
ALFWorld (unseen)		
REACT	N/A	98.8 ±0.9
ACCORD	N/A	99.3
BLACKSWAN	N/A	**99.5** ±0.4

gest that subjecting each proposed action to conjecture and refutation resolves errors that both often leave unaddressed.

Robustness under distribution shift. Moving from the normal to the challenge split in Table 1, the TGC of BLACKSWAN drops by only 3.6 points with thinking on and 2.6 without, whereas even the most robust baseline, ACCORD† with thinking on, drops by 11.7 points. ACE and ACCORD improve over REACT on both splits but drop nearly as much as REACT. That is, knowledge acquired in advance and confirmation-oriented verification raise performance, but do not reduce the drop on the challenge split. For ACE, this is because its playbook is distilled from training tasks, none of which involve the applications required by the challenge split. The gap is also evident in SGC: on the challenge split, BLACKSWAN completes entire scenarios more often than any baseline completes individual tasks, both with thinking on (72.4 SGC versus at most 69.8 TGC) and off (65.0 versus 56.3). The gap also widens *within* the challenge split. Across AppWorld's difficulty levels (Figure 4), our margin over ACCORD grows to +21.5 points at the hardest, which requires orchestrating the largest number of API calls.

Gains beyond reasoning and demonstrations. BLACKSWAN outperforms the baselines even when they are given more reasoning or more in-context supervision. In Table 1, BLACKSWAN without thinking outperforms every baseline with thinking, including ACCORD†, on all four metrics. Figure 3 varies the number of in-context demonstrations from zero to three, with thinking off. Additional demonstrations benefit the baselines, most notably ACCORD on the challenge split, whose TGC rises from about 55 to 69 points. BLACKSWAN, in contrast, stays within about 3 points across demonstration counts and, even without any demonstration, outperforms every baseline given three. These results suggest that the verification procedure of BLACKSWAN offers benefits that additional reasoning or demonstrations do not readily provide.

4.2 TRANSFER TO OTHER MODELS AND ENVIRONMENTS

Other backbone models. Table 2a repeats the AppWorld comparison with three further models: the dense Qwen3-4B-Thinking-2507 (Team, 2025) and two large mixture-of-experts models, MiniMax M2.7 (Chen et al., 2026a) and GLM-5.3-Flash (GLM-5-Team et al., 2026). With every model, BLACKSWAN outperforms both baselines on all metrics and splits. In contrast, ACCORD improves over REACT in TGC only with MiniMax M2.7. With GLM-5.3-Flash and Qwen3-4B, it falls below REACT on both splits, by up to 8.3 points with Qwen3-4B (14.9 versus 23.2 on the normal split). These results suggest that the benefit of confirmation-oriented verification depends on the model, whereas our conjecture and refutation improves performance with every model we test.

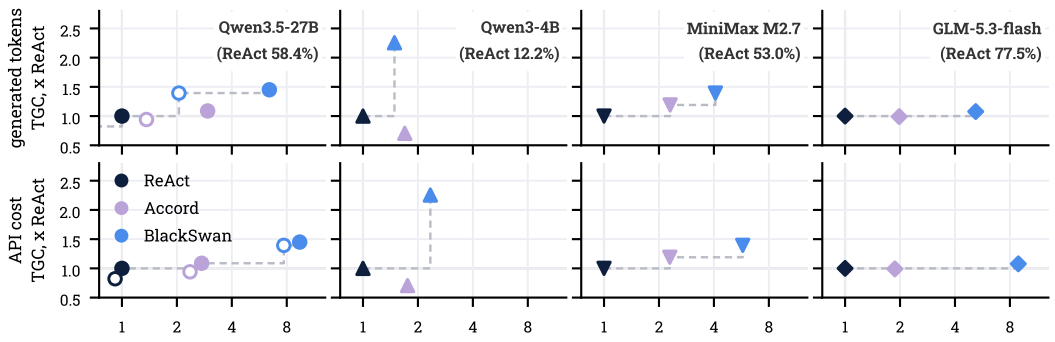


Figure 5: Cost (horizontal) and completion rate (vertical) on test challenge split of AppWorld, relative to REACT with thinking on the same backbone, whose completion rate is given in each panel. The top row counts generated tokens only; the bottom row adds input tokens at a common price. Filled markers denote thinking on, and hollow markers thinking off.

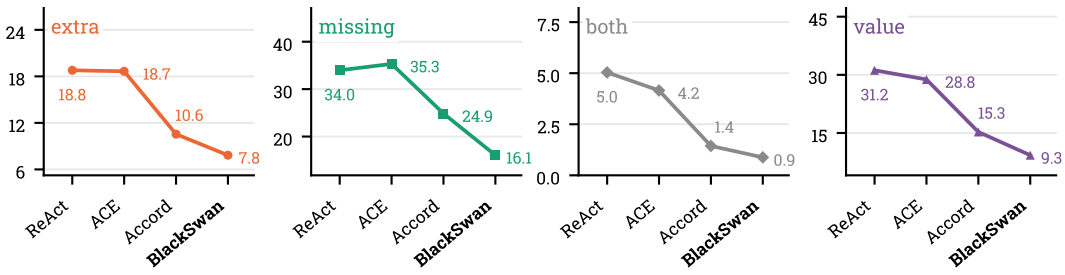


Figure 6: Failure types on the challenge split with thinking: collateral changes (*extra*), required changes left undone (*missing*), both together (*both*), and changes to the right items with wrong values (*value*). All counts are errors per 100 tasks, and an agent may make several errors on a task.

Other environments. Table 2b evaluates BLACKSWAN on two benchmarks with different action spaces: WebShop, where the agent navigates an online store through search and click actions (e.g., `click[Buy Now]`), and ALFWorld, where it acts in a household through text commands (e.g., `take apple 1 from countertop 2`). BLACKSWAN runs the same, proposed procedure as on AppWorld, with only its prompts made domain-neutral. On WebShop, it improves the success rate over REACT by 12.7 points and over Harness-R1 (Shao et al., 2026) by 8.8. On ALFWorld, it improves over REACT by 0.7 points and over ACCORD by 0.2. However, ACCORD here uses a procedure designed specifically for ALFWorld, unlike ours. Note that the applicable baselines differ across benchmarks; for example, ACCORD is defined over write APIs, which WebShop does not expose in the same form. The results in Table 2b therefore test whether BLACKSWAN transfers to other environments, rather than replicating the comparison in Table 1.

5 ANALYSIS

Ablation. Table 3 varies two design choices of BLACKSWAN separately on test-challenge with thinking off: whether the verifier asks how the proposed action could be wrong (refutation) or what it requires to hold (confirmation), and whether the potential falsifiers are written out and fixed before testing (explicit) or formed by the agent internally as it tests (implicit). For reference, ACCORD, which conceptually corresponds to confirmation with implicit suspicions, reaches $69.3 \pm 1.8\%$ TGC. We find that the two choices are effective only in combination. Refutation improves TGC by 5.6 points when suspicions are explicit but by 0.1 when they are implicit, and fixing suspicions in advance improves TGC by 8.9 points under refutation but by

Table 3: Verification orientation and externalization, varied separately.

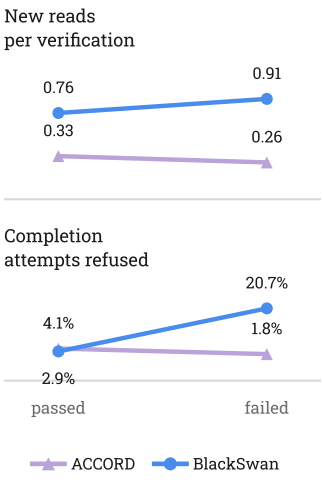
	Suspicions		
Orientation	Explicit	Implicit	Δ
Refutation	**81.6**	72.7	8.9
Confirmation	76.0	72.6	3.4
Δ	5.6	0.1	

3.4 under confirmation. This supports the preregistration of suspicions in Section 3.2: asking how an action could be wrong improves verification when the resulting suspicions are fixed before testing.

Computational cost. Figure 5 compares each agent with REACT on the same backbone. ACCORD costs 1.7–3.0× as much as REACT without consistently improving completion, and falls below it on Qwen3-4B (0.71×). BLACKSWAN lies on the frontier on every backbone: on Qwen3.5-27B, Qwen3-4B, and MiniMax M2.7, it completes 1.39–2.25× as many tasks at 1.5–9.4× the cost, gaining most on the weakest, Qwen3-4B. On GLM-5.3-flash, where REACT already completes 77.5% of tasks, BLACKSWAN gains less but is the only agent to improve. Both cost measures rank the agents identically, so the comparison is insensitive to cache reuse.

Failures reduced by verification. Omission, a required change that the agent never makes, is the most frequent error for every agent, and it is the error that verifying proposed writes is least able to see, since no write for it is ever proposed. Both ACCORD and BLACKSWAN reduce every kind of error relative to REACT, and BLACKSWAN reduces each more than ACCORD. Per 100 tasks, BLACKSWAN leaves 16.1 required changes undone, against 24.9 for ACCORD, a reduction of 35%; it also makes 39% fewer wrong values and 26% fewer collateral changes. In absolute terms, the largest reduction is in omission, which calls for an explanation.

Burden of proof. The two verifiers place the burden of proof differently. ACCORD rejects an action only when an observation confirms a violation and otherwise is told to "default to APPROVE" (Jiang et al., 2026), placing the burden on stopping the action; asking what would show the action wrong places it on executing it. Two consequences follow (right). First, BLACKSWAN gathers more new evidence: per verification, it makes 0.76 reads the agent had not already made on completed tasks and 0.91 on failed ones, against 0.33 and 0.26 for ACCORD. As a result, no verification read reached the faulted application in 53.1% of ACCORD's failures, against 11.3% for BLACKSWAN. Second, the verdict of BLACKSWAN tracks the outcome. On the action that ends the task, the last chance to catch an omission, BLACKSWAN rejects it in 20.7% of failed tasks and 2.9% of successful ones, whereas ACCORD rejects it in 1.8% and 4.1%. This reversal is specific to that gate: over all verifications, BLACKSWAN rejects 54.4% of actions against 8.4% for ACCORD, and both reject more often on failed trials.



6 CONCLUSION

We introduced a refutation-oriented approach to preemptive action verification, where agents actively seek evidence against proposed state-mutating actions before execution. BLACKSWAN operationalizes this idea with an expect-suspect-test procedure that fixes testable suspicions before gathering evidence. On AppWorld, BLACKSWAN achieves 84.7 TGC and 72.4 SGC on the challenge split, improving over the strongest baseline by 14.9 and 19.2 points, with even larger TGC gains when thinking is disabled; it also remains stronger without demonstrations than baselines given three. The gains are consistent across four backbones and transfer to WebShop and ALFWorld, while ablations show that generating suspicions helps substantially only when they are fixed before testing. Overall, the results suggest that agents have an underused capacity to recognize and correct misaligned actions at test time through active conjecture and refutation before execution.

Limitations and future work. Refutation cannot resolve genuine ambiguity: when user instructions admit multiple plausible interpretations, it may repeatedly surface the same unresolved issue across successive proposals, increasing verification cost. A natural extension is to treat such recurring suspicions as a signal of ambiguity and escalate them to the user for clarification.

AI USE STATEMENT

In this work, we used generative AI tools (Claude Code) to implement methods, to design and give feedback on experiments, to propose and refine hypotheses, to support qualitative analysis of agent trajectories, and to interpret results. We have not used generative AI tools to generate synthetic data sets, to develop the conceptual framework, to assist with translation, or to clean or reformat data sets, and formulating mathematical claims or proofs is not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

All experiments run in the simulated environments of AppWorld (Trivedi et al., 2024), WebShop (Yao et al., 2022), and ALFWorld (Shridhar et al., 2021). They involve no human subjects and no real personal data, as the users in AppWorld are fictitious. Pre-execution verification reduces but does not eliminate erroneous agent actions, so agents acting on real accounts still require further safeguards. Models accessed through a commercial API received only benchmark content.

REPRODUCIBILITY STATEMENT

Implementation details are given in the appendix: our agent (Section A) and its prompts (Section B), the reproduction of ACCORD and its prompts (Sections C and D), and cost measurement (Section E).

REFERENCES

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 *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=moWiYJuSGF.

Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, Changhao Zhang, Changqing Yu, Chao Wang, Cheng Ma, et al. The minimax-m2 series: Mini activations unleashing max real-world intelligence. *arXiv preprint arXiv:2605.26494*, 2026a.

Arthur Chen, Zuxin Liu, Jianguo Zhang, Akshara Prabhakar, Zhiwei Liu, Shelby Heinecke, Silvio Savarese, Victor Zhong, and Caiming Xiong. Test-time adaptation for LLM agents via environment interaction. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=OH4PE0TDo0.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Generating instruction manuals by LLM agents via interactive environmental learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Pwl9n4zlf5.

Xiaoyu Chen, Shenao Zhang, Pushi Zhang, Li Zhao, and Jianyu Chen. Asking before acting: Gather information in embodied decision making with language models. *arXiv preprint arXiv:2305.15695*, 2023.

Haishuo Fang, Xiaodan Zhu, and Iryna Gurevych. Preemptive detection and correction of misaligned actions in LLM agents. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 222–244, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.12. URL https://aclanthology.org/2025.emnlp-main.12/.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mRIQz8Zd60.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Kexin Huang, Ying Jin, Ryan Li, Michael Y. Li, Emmanuel Candes, and Jure Leskovec. Automated hypothesis validation with agentic sequential falsifications. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 25372–25437. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/huang25n.html.

Lai Jiang, Cheng Qian, Zhenhailong Wang, Pan Lu, Heng Ji, and Hao Peng. Accord: Action-conditioned contextual grounding for language agents. *arXiv preprint arXiv:2606.16432*, 2026.

Jeonghye Kim, Sojeong Rhee, Minbeom Kim, Dohyung Kim, Sangmook Lee, Youngchul Sung, and Kyomin Jung. ReflAct: World-grounded decision making in LLM agents via goal-state reflection. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33433–33465, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1697. URL https://aclanthology.org/2025.emnlp-main.1697/.

Kasia Kobalczyk, Nicolás Astorga, Tennison Liu, and Mihaela van der Schaar. Active task disambiguation with LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JAMxRSXLFz.

Yanming Liu, Xinyue Peng, Jiannan Cao, Xinyi Wang, Songhang Deng, Jintao Chen, Jianwei Yin, and Xuhong Zhang. ToolGate: Contract-grounded and verified tool execution for LLMs. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 9653–9684, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.470. URL https://aclanthology.org/2026.findings-acl.470/.

Weidi Luo, Shenghong Dai, Xiaogeng Liu, Suman Banerjee, Huan Sun, Muhao Chen, and Chaowei Xiao. AGrail: A lifelong agent guardrail with effective and adaptive safety detection. In

Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8104–8139, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.399. URL https://aclanthology.org/2025.acl-long.399/.

Yuting Ning, Jaylen Jones, Zhehao Zhang, Chentao Ye, Weitong Ruan, Junyi Li, Rahul Gupta, and Huan Sun. When actions go off-task: Detecting and correcting misaligned actions in computer-use agents. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=Q3UQSz4Giw.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=jL7fwchScm.

Karl R Popper and George E Hudson. Conjectures and refutations, 1963.

Cheng Qian, Bingxiang He, Zhong Zhuang, Jia Deng, Yujia Qin, Xin Cong, Zhong Zhang, Jie Zhou, Yankai Lin, Zhiyuan Liu, et al. Tell me more! towards implicit user intention understanding of language model driven agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1088–1113, 2024.

Shuofei Qiao, Runnan Fang, Ningyu Zhang, Yuqi Zhu, Xiang Chen, Shumin Deng, Yong Jiang, Pengjun Xie, Fei Huang, and Huajun Chen. Agent planning with world knowledge model. *Advances in Neural Information Processing Systems*, 37:114843–114871, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Rohin Shah, Dmitrii Krasheninnikov, Jordan Alexander, Pieter Abbeel, and Anca Dragan. Preferences implicit in the state of the world. *arXiv preprint arXiv:1902.04198*, 2019.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Erfan Shayegani, Keegan Hines, Yue Dong, Nael Abu-Ghazaleh, Roman Lutz, Spencer Whitehead, Vidhisha Balachandran, Besmira Nushi, and Vibhav Vineet. Just do it!? computer-use agents exhibit blind goal-directedness. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=9W4bPRsEIT.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. {ALFW}orld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Xinyuan Song and Zekun Cai. Ask the world before acting: Budgeted environment probing for world-model calibration. *arXiv preprint arXiv:2606.31422*, 2026.

Hongjin SU, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan O Arik. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=3UKOzGWCVY.

Manan Suri, Puneet Mathur, Nedim Lipka, Franck Dernoncourt, Ryan A. Rossi, and Dinesh Manocha. Structured uncertainty guided clarification for LLM agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 40811–40838, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.2028. URL https://aclanthology.org/2026.findings-acl.2028/.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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 Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Sanidhya Vijayvargiya, Vijay Viswanathan, and Graham Neubig. Asking what matters: Reward-driven clarification for software engineering tasks. In *Forty-third International Conference on Machine Learning*, 2026a. URL https://openreview.net/forum?id=YxsIwwCfjf.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. AmbigSWE: Interactive agents to overcome underspecificity in software engineering. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=X2yzXtH4wp.

Hanlin Wang, Chak Tou Leong, Jian Wang, and Wenjie Li. Seeing isn't believing: Mitigating belief inertia via active intervention in embodied agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 37796–37815, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1884. URL https://aclanthology.org/2026.findings-acl.1884/.

Haoyu Wang, Tao Li, Zhiwei Deng, Dan Roth, and Yang Li. Devil's advocate: Anticipatory reflection for llm agents. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 966–978, 2024a.

Wenxuan Wang, Shi Juluan, Zixuan Ling, Yuk-Kit Chan, Chaozheng Wang, Cheryl Lee, Youliang Yuan, Jen-tse Huang, Wenxiang Jiao, and Michael R. Lyu. Learning to ask: When LLM agents meet unclear instruction. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 21773–21784, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1104. URL https://aclanthology.org/2025.emnlp-main.1104/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025c. URL https://openreview.net/forum?id=NTAhi2JEEE.

Shunyu Yao, Howard Chen, John Yang, and Karthik R Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=R9KnuFlvnU.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Michael Zhang, W Bradley Knox, and Eunsol Choi. Modeling future conversation turns to teach llms to ask clarifying questions. In *International Conference on Learning Representations*, volume 2025, pp. 60722–60742, 2025.

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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=eC4ygDs02R.

Xuan Zhang, Yang Deng, Zifeng Ren, See-Kiong Ng, and Tat-Seng Chua. Ask-before-plan: Proactive language agents for real-world planning. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10836–10863, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.636. URL https://aclanthology.org/2024.findings-emnlp.636/.

A IMPLEMENTATION DETAILS OF BLACKSWAN

Algorithm 1 gives the loop of BLACKSWAN on one task, and Table 4 its limits. BLACKSWAN is a code agent in the style of REACT: each step drafts one Python cell c and runs it. h is the history the model sees, and $h \oplus (\cdot)$ appends turns to it. Each call marked with a prompt reads h but adds nothing to it; only the appends in Algorithm 1 do.

Algorithm 1 BLACKSWAN on one task.

Require: instruction x, model M

$h \leftarrow$ the REACT prompt with x; $T \leftarrow \emptyset$ ▷ word–endpoint tables
repeat
  for $a = 0, \ldots, 3$ **do** ▷ at most 3 redrafts
    $c \leftarrow \textsc{Draft}(M, h)$ ▷ Prompt 1; form rules
    if c only reads **then break**
    $e \leftarrow \textsc{Expect}(M, h, c)$ ▷ Prompt 3
    $S \leftarrow \textsc{Suspect}(M, h, c, e, \textsc{Anatomy}(c), T)$ ▷ Prompts 4, 5
    if $S = \emptyset$ **then break**
    $(P, V) \leftarrow \textsc{Test}(M, h, c, S)$ ▷ Prompt 6
    $h \leftarrow h \oplus (c, S, P, V)$ ▷ probes P, verdicts V
    if no verdict in V is *confirmed* **then break**
  end for
  $o \leftarrow$ run c; $h \leftarrow h \oplus (c, o)$ ▷ the last draft runs even if blocked
  if c lists a new app u's endpoints **then** $T \leftarrow T \cup \textsc{Ground}(M, h, u, x)$ ▷ Prompt 2
until c calls `complete_task`

Draft. Three rules check the form of a drafted cell. A cell that calls an endpoint whose documentation the agent has not read is replaced by that lookup; a cell that reads an endpoint's documentation before the app's endpoint list is replaced by the list; and a cell that calls `complete_task` together with other code is drafted again. A cell *only reads* when every API call in it is a GET request, a documentation lookup, or `login`; a cell with no API call is verified.

Verification. ANATOMY is a static analysis of the cell: its variable bindings and API calls in evaluation order, each call with every parameter, including those left at their documented defaults. On a cell that calls `complete_task`, SUSPECT asks instead what part of the task is not done yet (Prompt 5), at most twice per task. TEST judges all suspicions together. At each turn the model either writes a probe, which the harness runs and answers with its output, or ends with a verdict for every suspicion: *confirmed*, *dismissed*, or *deferred*. A probe may only read. A probe that makes the cell's own write defers every verdict, since only running the cell can settle it; a probe that makes any other write is refused, and the model is asked again. Only a confirmed suspicion blocks the cell. The verdicts then end h, so the next draft continues from them.

Context. Before each model call, while h exceeds the context budget, the harness shortens its cell outputs one step at a time, oldest first; every value it clips stays retrievable from the environment.

Table 4: Limits of BLACKSWAN in the runs of Table 1.

Item	Limit
Word–endpoint table	once per app, at most 8 entries
EXPECT	one expectation, at most 500 characters
SUSPECT	at most 5 suspicions, each at most 1,000 characters
TEST	at most 50 read-only probes per verification; grounds at most 400 characters
Redrafts of a blocked cell	at most 3 per step
Decoding	temperature 0, at most 8,192 generated tokens per call

B BLACKSWAN PROMPTS

BLACKSWAN runs the same stages on all three benchmarks. The prompt the agent starts from and the API it reads are the benchmark's own. The verification prompts are AppWorld's on ALFWorld as well; on WebShop, SUSPECT and TEST also state how the shop works, SUSPECT words two of its five kinds for a shop, and TEST names the shop's read endpoints and defers two kinds of suspicion (Prompts 11 and 12). Values filled in at run time are shown in angle brackets; limits are those of the reported runs.

B.1 APPWORLD

The agent drafts each action on AppWorld's official ReAct prompt with its one demonstration, to which it adds the turns of Prompt 1. When it lists an app's endpoints, it answers Prompt 2 once for that app. Each drafted action that writes to the environment then goes through EXPECT (Prompt 3), SUSPECT (Prompt 4, or Prompt 5 when the action ends the task), and TEST (Prompts 6 and 7), and the agent's next turn begins with the verdicts (Prompt 8).

Table 5: The five kinds of falsifier the SUSPECT prompt names

Kind	The falsifier asserts that	Derived from
Wrong object	the records acted on are not the ones the task names	the action
Incomplete set	the task names more records than the action covers	the action
Wrong value	an argument or endpoint is not what the task means	the action
Hidden state	existing state changes what the call does	the action
The task's words	the part of the task this action carries out asks for something the action does not do	the task and the word–endpoint table

````
[[[ user ]]]
My name is: <first_name> <last_name>. My personal email is <email> and phone number is <phone_number>.
Task: How many playlists do I have in Spotify?

[[[ assistant ]]]
Let me first store your details and today's date in variables for later use.
Code:
```python
<code that stores these details and today's date in variables>
```

[[[ user ]]]
Output:
```
User: <first_name> <last_name> (<email>, <phone_number>); today: 2023-05-18 12:00:00
```
````

Prompt 1: Draft: the turns our agent adds to AppWorld's official ReAct prompt, after its first user turn. The rest of the prompt, including its one demonstration, is the official one; the demonstration's API documentation is rendered from the live environment.

````
The list above is every endpoint of `<app>`. Before going on, match the task to it.

The task, verbatim:

> <task instruction>

For each thing the task asks this app to do, write one entry:

- `quote`: the task's words, copied word for word.
- `endpoint`: the endpoint whose description says it does that.
- `evidence`: the words of that description, copied exactly.
- `general`: if a second endpoint could also do it but its description is about a broader act or object,
    name it here; otherwise leave it empty. `endpoint` is then the one whose description names the same
    object or act as the quoted words.

At most 8 entries. Answer in exactly one fenced ```json``` block:

```json
````
````

```
[{"quote": "<words copied from the task>", "endpoint": "<endpoint name>", "evidence": "<words copied from
     that endpoint's description>", "general": "<the broader endpoint, or empty>"}, ...]
```

Prompt 2: Word–endpoint table, asked once per app after the agent lists that app's endpoints.

```
You proposed the code above. Before it executes, state clearly what you expect this code to accomplish
     for the user's task.

- Describe the resulting task-relevant state that should hold if this code succeeds. Do not describe your
     future plans.
- Be precise about which objects, values, or parts of the user's request this code is intended to affect.
     Do not claim that a broader requirement such as "all," "each," "only," or "exactly" will be
     satisfied unless this single execution actually establishes it.

Answer in exactly one fenced ```json``` block, and keep it under 500 characters:

```json
{"expectation": "..."}
```
```

Prompt 3: EXPECT.

```
Before that cell runs, argue against yourself. What could make what you just said false when it actually
     runs? List the specific ways it could turn out wrong -- not generic caution, and not doubts the
     environment rules out.

In the order it would run, that cell binds these variables and makes these API calls -- every parameter
     of every call, including the optional ones it does not pass (`"passed": false`), which take the
     `default` shown:

```json
<the cell's variables and API calls, every parameter included>
```

Look for failures of these kinds:

1. **The wrong thing.** The objects this acts on may not be the ones the task asked for -- a name that
     merely looks right, an id read off the wrong record.
2. **Not all of them.** The task may name a set or an amount this cell does not cover. If it acts on some
     members, say what would show there are others it missed.
3. **The wrong value or operation.** The arguments, quantities, or the endpoint itself may not be what
     the request means, even when the target is right.
4. **Hidden state.** Something already in the environment may change the outcome -- what is in the cart,
     what the account already holds, what an earlier cell left behind.
5. **The task's words.** The part of the task this code carries out may ask for something the code does
     not do: an endpoint whose description matches those words more closely than the one it calls, or a
     condition, an amount or a second action in those words that the code leaves out.

When you read the apps' endpoints, you matched the task's words to these endpoints:

<the word--endpoint table>

Each entry is one way it could fail, in terms that a read of the environment could settle, and under 1000
     characters. Say what it is about as its `ref`: one of the `ref` values listed above, exactly as
     written there. Give EVERY doubt you actually have, up to 5 -- the number is not fixed, and the shape
     below is not a hint at how many. If you genuinely see none, answer with an empty list.

Answer in exactly one fenced ```json``` block:

```json
[{"doubt": "<one way it could turn out wrong>", "ref": "<one of the ref values listed above>"}, ...]
```
```

Prompt 4: SUSPECT. The five kinds are those of Table 5; the table is the agent's answer to Prompt 2.

```
That cell ends the task. Before it runs, argue against yourself: what part of the task is not done yet?

This is the task:

<task instruction>

When you read the apps' endpoints, you matched these words of the task to endpoints that change state,
     and no cell above has called any of them:

<the task's words matched to state-changing endpoints that no cell has called>
```

```
Go through the task's words one by one -- every action it asks for, every object it names, every
      condition or amount attached to them -- and check each against the cells above and their Outputs.
      Look for:

1. **Not done.** An action the task asks for that no cell above carried out.
2. **Done in part.** An action carried out for only some of the objects the task names, or without a
      condition, amount or detail the task attaches to it.
3. **Done to the wrong thing.** An action carried out on objects other than the ones the task names.

Each entry is one part of the task that may not be done, in terms that a read of the environment could
      settle -- what a read would show if it were not done -- and under 1000 characters. Set its `ref` to
      `"task"`. Give EVERY such part you actually suspect, up to 5 -- the number is not fixed. If every
      part of the task is done, answer with an empty list.

Answer in exactly one fenced ```json``` block:

```json
[{"doubt": "<one part of the task that may not be done>", "ref": "task"}, ...]
```
```

Prompt 5: SUSPECT on a cell that ends the task.

```
Wait. I have not run that code, because I'm suspicious about:

<the suspicions, numbered>

None of it has run, so none of the API calls in it have had any effect -- what you read now is the world
      as it stands before that code.
---

Do your best to gather thorough and sufficient evidence to determine whether each suspicion should be
      confirmed or dismissed. Investigate them seriously in either direction: do not try to defend the
      proposed cell, but do not assume a suspicion is true either.

**Do not treat plausibility, possibility, or expectation as evidence.** When the current evidence does
      not decisively settle a suspicion, actively use evidence-gathering probes to obtain observations
      that can. Choose probes for their ability to distinguish the possibilities, and keep probing until
      you have concrete evidence or no useful probe remains.

Answer with exactly either a ```python``` or ```json``` block.

If more evidence is needed, return one ```python``` block containing an evidence-gathering probe:

```python
# Write a code to probe
```

To gather evidence, you may use any GET endpoints (including API documentation). If you want to access
      apps that are not currently logged in, you may perform the necessary `apis.<app>.login` action
      first. Reuse any tokens or variables already available from previous cells.

You can attempt <n> probes. Use them efficiently: prefer probes that are most likely to settle a
      suspicion and avoid redundant observations.

When you can judge, return one ```json``` block covering **<the suspicions' numbers>** -- those are the
      ones an observation can settle, and the rest are not yours to decide here:

```json
{"<number>": {"verdict": "suspicion_is_confirmed" | "suspicion_is_dismissed", "grounds": "<concrete
      evidence, 400 characters at most>"}}
```

Use `"judgement_is_deferred"` for a suspicion no available evidence or probe can settle. If a useful
      probe could still distinguish the possibilities, probe instead of deferring.

A suspicion about the task's words -- another endpoint, or a part of the task not done -- is confirmed
      only by a line you quote as the grounds: an endpoint's description from the API docs, or an Output
      line showing the part is not done. A suspicion that only reads the task's words another way is
      `judgement_is_deferred`.
```

Prompt 6: TEST: the opening turn, stated once for all suspicions.

```


--- after a probe ---
```

```
Here is the output of your probe.

Use this new evidence to update your judgement.

Return exactly one fenced block:
- JSON if you can judge the suspicion.
- Python if another useful probe is needed.

--- suspicions left without a verdict ---

You did not give a verdict for these:

<the suspicions without a verdict>

Answer for them now, in one ```json``` block in the same shape -- or probe again first if that is what
     they need. You can attempt <n> probes.

--- a probe that would write ---

I did not run that. It calls <the write endpoints it calls>, which the cell I am asking about does not
     call.

A probe only READS. The cell under suspicion has not run, so anything a probe changes changes the world
     for a cell that may still be rewritten -- and a call that cell never makes is not evidence about
     that cell either.

Answer again: a ```json``` block if you can decide now, or a
```python``` block that only reads. <n> reads left.
```

Prompt 7: TEST: follow-up turns, after a probe's output, for suspicions left without a verdict, and for a probe that would write.

```
<one verdict per suspicion, with its grounds>

So, nothing here stops it -- running the cell as written:

--- otherwise ---

<one verdict per suspicion, with its grounds>

So,
```

Prompt 8: The agent's next turn begins with the verdicts; the sentence ends one way when no suspicion is confirmed (the cell runs as written) and the other way otherwise (the agent writes a revised cell).

B.2 WEBSHOP

Thinking is off, and a task allows at most 50 steps. The agent starts from Prompt 9: its system turn is the one-sentence task definition of the default WebShop prompt of Harness-R1 (Shao et al., 2026), and its user turns are a benchmark-neutral seed with no persona and no demonstration. It reads the shop's API (Prompt 10) through `api_docs`. A task ends only with a purchase: before buy, a call to `complete_task` is refused with "This shopping session ends only with a purchase: call apis.webshop.buy." SUSPECT and TEST state how the shop works (Prompts 11 and 12). The word–endpoint table, EXPECT, SUSPECT on a cell that ends the task, and the later turns of TEST use the AppWorld prompts (Prompts 2, 3, 5, 7, and 8).

```
[[[ system ]]]
You are a web shopping agent. Follow the task instruction to find and buy the correct product.

[[[ user ]]]
You are a super intelligent AI Assistant whose job is to achieve the given task completely autonomously.
     The shopping instructions above govern everything you do; where they and this message disagree, they
     win.

To do this, you will interact with a set of apps through their associated APIs. For this you will
     undertake a *multi-step conversation* using a python REPL environment. That is, you will write
     python code and the environment will execute it and show you the result, based on which you will
     write python code for the next step, and so on, until you've achieved the goal.

Here are three key APIs that you need to know to get more information

# To get the list of apps that are available to you.
```
```

````
```python
print(apis.api_docs.show_app_descriptions())
```

# To get the list of apis under any app listed above, e.g. an app named `<app>`

```python
print(apis.api_docs.show_api_descriptions(app_name='<app>'))
```

# To get the specification of a particular api, e.g. app `<app>`'s api `<api>`

```python
print(apis.api_docs.show_api_doc(app_name='<app>', api_name='<api>'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you
      can now generate code, that I will execute, to solve the task.

The apps currently available to you are:
[
 {
  "description": "An online shop. Search its catalogue, open a product and read its sections, choose the
      product\u0027s options, and buy it. One purchase per session; buying ends the session.",
  "name": "webshop"
 },
 {
  "description": "Task-lifecycle APIs (mark the task complete).",
  "name": "supervisor"
 }
]

**Key instructions**:
(1) Make sure to end code blocks with ``` followed by a newline(\n).

(2) Remember you can use the variables in your code in subsequent code blocks.

(3) Always look at API specifications (using apis.api_docs.show_api_doc) before calling an API.

(4) Write small chunks of code and only one chunk of code in every step. Make sure everything is working
      correctly before making any irreversible change.

(5) Once you have completed the task, make sure to call apis.supervisor.complete_task(). If the task
      asked for some information, return it as the answer argument, i.e. call
      apis.supervisor.complete_task(answer=<answer>). Many tasks do not require an answer, so in those
      cases, just call apis.supervisor.complete_task() i.e. do not pass any argument.


[[[ assistant ]]]
Understood. I'll consult the app and API docs as needed, work in small verified steps, and call
      `apis.supervisor.complete_task()` when done. What is the task?

[[[ user ]]]
Using these APIs, generate code to solve the actual task:

Task: <task instruction>
````

Prompt 9: WebShop: the first turns. The system turn is the task definition of Harness-R1's default WebShop prompt; the user turns are our benchmark-neutral seed. In the first user turn, `<app>`, `<api>`, and `<answer>` are part of the prompt itself.

````
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_descriptions(app_name='webshop'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_descriptions {"app_name": "webshop"} -> [{"name": "search", "description":
      "Search the catalogue. Returns one page of ten results (asin, title, price) and the total number of
      results."}, {"name": "view_product", "description": "Open a product from your search results: its
      title, price, rating, the options it can be bought with, and the options selected so far."},
      {"name": "product_section", "description": "Read one section of a product from your search results:
      `Description`, `Features` or `Reviews`."}, {"name": "select_option", "description": "Choose one
      value of one of a product's options, e.g. its size or color. Choosing again replaces the earlier
      value."}, {"name": "buy", "description": "Buy a product with the options selected for it. Ends the
      session; it cannot be undone."}]
````
````

````
[
 {
  "name": "search",
  "description": "Search the catalogue. Returns one page of ten results (asin, title, price) and the
      total number of results."
 },
 {
  "name": "view_product",
  "description": "Open a product from your search results: its title, price, rating, the options it can
      be bought with, and the options selected so far."
 },
 {
  "name": "product_section",
  "description": "Read one section of a product from your search results: `Description`, `Features` or
      `Reviews`."
 },
 {
  "name": "select_option",
  "description": "Choose one value of one of a product's options, e.g. its size or color. Choosing again
      replaces the earlier value."
 },
 {
  "name": "buy",
  "description": "Buy a product with the options selected for it. Ends the session; it cannot be undone."
 }
]
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='webshop', api_name='search'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "webshop", "api_name": "search"} -> {"app_name": "webshop",
      "api_name": "search", "path": "/webshop/search", "method": "GET", "description": "Search the
      catalogue. Returns one page of ten results (asin, title, price) and the total number of results.",
      "parameters": [{"name": "query", "required": true, "type": "string"}, {"name": "page", "required":
      false, "type": "integer"}], "response_schemas": {"success": {"page": "any", "total": "any",
      "results": "any"}, "failure": "an exception is raised"}}
{
 "app_name": "webshop",
 "api_name": "search",
 "path": "/webshop/search",
 "method": "GET",
 "description": "Search the catalogue. Returns one page of ten results (asin, title, price) and the total
      number of results.",
 "parameters": [
  {
   "name": "query",
   "required": true,
   "type": "string"
  },
  {
   "name": "page",
   "required": false,
   "type": "integer"
  }
 ],
 "response_schemas": {
  "success": {
   "page": "any",
   "total": "any",
   "results": "any"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='webshop', api_name='view_product'))
```

[[[ user ]]]
Output:
```
````

````
INFO apis.api_docs.show_api_doc {"app_name": "webshop", "api_name": "view_product"} -> {"app_name":
     "webshop", "api_name": "view_product", "path": "/webshop/view_product", "method": "GET",
     "description": "Open a product from your search results: its title, price, rating, the options it
     can be bought with, and the options selected so far.", "parameters": [{"name": "asin", "required":
     true, "type": "string"}], "response_schemas": {"success": {"asin": "any", "title": "any", "price":
     "any", "rating": "any", "options": "any", "selected_options": "any"}, "failure": "an exception is
     raised"}}
{
 "app_name": "webshop",
 "api_name": "view_product",
 "path": "/webshop/view_product",
 "method": "GET",
 "description": "Open a product from your search results: its title, price, rating, the options it can be
     bought with, and the options selected so far.",
 "parameters": [
  {
   "name": "asin",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "asin": "any",
   "title": "any",
   "price": "any",
   "rating": "any",
   "options": "any",
   "selected_options": "any"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='webshop', api_name='product_section'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "webshop", "api_name": "product_section"} -> {"app_name":
     "webshop", "api_name": "product_section", "path": "/webshop/product_section", "method": "GET",
     "description": "Read one section of a product from your search results: `Description`, `Features` or
     `Reviews`.", "parameters": [{"name": "asin", "required": true, "type": "string"}, {"name":
     "section", "required": true, "type": "string"}], "response_schemas": {"success": {"asin": "any",
     "section": "any", "content": "any"}, "failure": "an exception is raised"}}
{
 "app_name": "webshop",
 "api_name": "product_section",
 "path": "/webshop/product_section",
 "method": "GET",
 "description": "Read one section of a product from your search results: `Description`, `Features` or
     `Reviews`.",
 "parameters": [
  {
   "name": "asin",
   "required": true,
   "type": "string"
  },
  {
   "name": "section",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "asin": "any",
   "section": "any",
   "content": "any"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
````

````
print(apis.api_docs.show_api_doc(app_name='webshop', api_name='select_option'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "webshop", "api_name": "select_option"} -> {"app_name":
     "webshop", "api_name": "select_option", "path": "/webshop/select_option", "method": "POST",
     "description": "Choose one value of one of a product's options, e.g. its size or color. Choosing
     again replaces the earlier value.", "parameters": [{"name": "asin", "required": true, "type":
     "string"}, {"name": "option_type", "required": true, "type": "string"}, {"name": "value",
     "required": true, "type": "string"}], "response_schemas": {"success": {"asin": "any",
     "selected_options": "any"}, "failure": "an exception is raised"}}
{
 "app_name": "webshop",
 "api_name": "select_option",
 "path": "/webshop/select_option",
 "method": "POST",
 "description": "Choose one value of one of a product's options, e.g. its size or color. Choosing again
     replaces the earlier value.",
 "parameters": [
  {
   "name": "asin",
   "required": true,
   "type": "string"
  },
  {
   "name": "option_type",
   "required": true,
   "type": "string"
  },
  {
   "name": "value",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "asin": "any",
   "selected_options": "any"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='webshop', api_name='buy'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "webshop", "api_name": "buy"} -> {"app_name": "webshop",
     "api_name": "buy", "path": "/webshop/buy", "method": "POST", "description": "Buy a product with the
     options selected for it. Ends the session; it cannot be undone.", "parameters": [{"name": "asin",
     "required": true, "type": "string"}], "response_schemas": {"success": {"message": "any", "asin":
     "any", "selected_options": "any"}, "failure": "an exception is raised"}}
{
 "app_name": "webshop",
 "api_name": "buy",
 "path": "/webshop/buy",
 "method": "POST",
 "description": "Buy a product with the options selected for it. Ends the session; it cannot be undone.",
 "parameters": [
  {
   "name": "asin",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "message": "any",
   "asin": "any",
   "selected_options": "any"
  },
  "failure": "an exception is raised"
 }
````

````
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='supervisor', api_name='complete_task'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "supervisor", "api_name": "complete_task"} -> {"app_name":
      "supervisor", "api_name": "complete_task", "path": "/supervisor/complete_task", "method": "POST",
      "description": "Signal that the task is finished. Call once, last, with the answer if the task asks
      a question.", "parameters": [{"name": "answer", "required": false, "type": "string"}],
      "response_schemas": {"success": {"message": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "supervisor",
 "api_name": "complete_task",
 "path": "/supervisor/complete_task",
 "method": "POST",
 "description": "Signal that the task is finished. Call once, last, with the answer if the task asks a
      question.",
 "parameters": [
  {
   "name": "answer",
   "required": false,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "message": "string"
  },
  "failure": "an exception is raised"
 }
}
```
````

Prompt 10: WebShop: the shop's API documentation as it enters the agent's context. Each `api_docs` cell is followed by its Output, verbatim from the runs: the list of the shop's endpoints, then the specification of each.

````
Before that cell runs, argue against yourself. What could make what you just said false when it actually
      runs? List the specific ways it could turn out wrong -- not generic caution, and not doubts the
      environment rules out.

In the order it would run, that cell binds these variables and makes these API calls -- every parameter
      of every call, including the optional ones it does not pass (`"passed": false`), which take the
      `default` shown:

```json
<the cell's variables and API calls, every parameter included>
```

About this shop -- how it works, not a hint at any answer:
- A task here is completed by buying exactly one product with `apis.webshop.buy`. Words such as "find",
      "looking for", "need" or "search" in the task say which product to buy; they do not ask for a report
      instead of a purchase.
- `apis.webshop.select_option` sets one option of one product per call. A cell that selects some of a
      product's options is one step: whether every option the task names is selected, and whether the
      product is bought, is settled at the cell that calls `apis.webshop.buy`, not at this one.

Look for failures of these kinds:

1. **The wrong thing.** The objects this acts on may not be the ones the task asked for -- a name that
      merely looks right, an id read off the wrong record.
2. **Not all of them.** At the cell that buys: the task may name options or an amount the product as
      bought does not have -- say which, and what a read would show.
3. **The wrong value or operation.** The arguments, quantities, or the endpoint itself may not be what
      the request means, even when the target is right.
4. **Hidden state.** Something already in the shop may change the outcome -- the options already selected
      for this product, what an earlier cell left behind.
5. **The task's words.** The part of the task this code carries out may ask for something the code does
      not do: an endpoint whose description matches those words more closely than the one it calls, or a
      condition, an amount or a second action in those words that the code leaves out.

When you read the apps' endpoints, you matched the task's words to these endpoints:
````
````

```
<the word--endpoint table>

Each entry is one way it could fail, in terms that a read of the environment could settle, and under 1000
      characters. Say what it is about as its `ref`: one of the `ref` values listed above, exactly as
      written there. Give EVERY doubt you actually have, up to 5 -- the number is not fixed, and the shape
      below is not a hint at how many. If you genuinely see none, answer with an empty list.

Answer in exactly one fenced ```json``` block:

```json
[{"doubt": "<one way it could turn out wrong>", "ref": "<one of the ref values listed above>"}, ...]
```
```

Prompt 11: WebShop: SUSPECT. It is Prompt 4 with the block *About this shop* added and kinds 2 and 4 worded for a shop.

```
Wait. I have not run that code, because I'm suspicious about:

<the suspicions, numbered>

None of it has run, so none of the API calls in it have had any effect -- what you read now is the world
      as it stands before that code.
---

Do your best to gather thorough and sufficient evidence to determine whether each suspicion should be
      confirmed or dismissed. Investigate them seriously in either direction: do not try to defend the
      proposed cell, but do not assume a suspicion is true either.

**Do not treat plausibility, possibility, or expectation as evidence.** When the current evidence does
      not decisively settle a suspicion, actively use evidence-gathering probes to obtain observations
      that can. Choose probes for their ability to distinguish the possibilities, and keep probing until
      you have concrete evidence or no useful probe remains.

Answer with exactly either a ```python``` or ```json``` block.

If more evidence is needed, return one ```python``` block containing an evidence-gathering probe:

```python
# Write a code to probe
```

To gather evidence, you may use the shop's read endpoints -- `apis.webshop.search`,
      `apis.webshop.view_product`, `apis.webshop.product_section` -- and the API documentation. Reuse any
      variables already available from previous cells.

About this shop -- how it works, not a hint at any answer:
- A task here is completed by buying exactly one product with `apis.webshop.buy`. Words such as "find",
      "looking for", "need" or "search" in the task say which product to buy; they do not ask for a report
      instead of a purchase.
- `apis.webshop.select_option` sets one option of one product per call. A cell that selects some of a
      product's options is one step: whether every option the task names is selected, and whether the
      product is bought, is settled at the cell that calls `apis.webshop.buy`, not at this one.
- Every task in this shop was written for a product in its catalogue.

You can attempt <n> probes. Use them efficiently: prefer probes that are most likely to settle a
      suspicion and avoid redundant observations.

When you can judge, return one ```json``` block covering **<the suspicions' numbers>** -- those are the
      ones an observation can settle, and the rest are not yours to decide here:

```json
{"<number>": {"verdict": "suspicion_is_confirmed" | "suspicion_is_dismissed", "grounds": "<concrete
      evidence, 400 characters at most>"}}
```

Use `"judgement_is_deferred"` for a suspicion no available evidence or probe can settle. If a useful
      probe could still distinguish the possibilities, probe instead of deferring.

A suspicion about the task's words -- another endpoint, or a part of the task not done -- is confirmed
      only by a line you quote as the grounds: an endpoint's description from the API docs, or an Output
      line showing the part is not done. A suspicion that only reads the task's words another way is
      `judgement_is_deferred`.

A suspicion that a `select_option` cell leaves other options unselected, or that it does not buy, is not
      about that cell: judge it `judgement_is_deferred`. A suspicion that no product in the shop fits the
      task is not confirmed by searches that did not find one: judge it `judgement_is_deferred`.
```

Prompt 12: WebShop: TEST, opening turn. It is Prompt 6 with the shop's read endpoints named in place of AppWorld's login step, the block *About this shop*, and the last paragraph added.

B.3 ALFWORLD

Thinking is on. The agent starts from Prompt 13: a benchmark-neutral seed with no persona, then the two demonstrations that ReAct (Yao et al., 2023) shows for the task's type, verbatim, and the game's opening text. It acts through one app whose endpoints are the game's commands (Prompt 14). `look`, `inventory`, and `examine` are reads: a cell that calls only these runs without verification, and they are the commands a probe may send. An episode allows 50 game commands, ALFWorld's standard limit, and every command counts, probes included. All verification prompts are the AppWorld ones (Prompts 2 to 8).

```
[[[ user ]]]
You are a super intelligent AI Assistant whose job is to achieve the given task completely autonomously.

To do this, you will interact with a set of apps through their associated APIs. For this you will
     undertake a *multi-step conversation* using a python REPL environment. That is, you will write
     python code and the environment will execute it and show you the result, based on which you will
     write python code for the next step, and so on, until you've achieved the goal.

Here are three key APIs that you need to know to get more information

# To get the list of apps that are available to you.

```python
print(apis.api_docs.show_app_descriptions())
```

To get the list of apis under any app listed above, e.g. an app named `<app>`

```python
print(apis.api_docs.show_api_descriptions(app_name='<app>'))
```

To get the specification of a particular api, e.g. app `<app>`'s api `<api>`

```python
print(apis.api_docs.show_api_doc(app_name='<app>', api_name='<api>'))
```

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you
     can now generate code, that I will execute, to solve the task.

The apps currently available to you are:
[
 {
  "description": "The household you are in. Look around, check what you carry, examine things, move
     between receptacles, open and close them, take and put objects, and heat, cool, clean or switch on
     things. Objects and receptacles are named by type and index (`mug 1`, `cabinet 3`); a type name is
     exact \u2014 a similarly named type is a different type. The task is judged on the household\u0027s
     state when you finish: where every object is, in your hands included, and what has been done to it
     (heated, cooled, cleaned, switched on).",
  "name": "alfworld"
 },
 {
  "description": "Task-lifecycle APIs (mark the task complete).",
  "name": "supervisor"
 }
]

Key instructions:
(1) Make sure to end code blocks with ``` followed by a newline(\n).

(2) Remember you can use the variables in your code in subsequent code blocks.

(3) Always look at API specifications (using apis.api_docs.show_api_doc) before calling an API.

(4) Write small chunks of code and only one chunk of code in every step. Make sure everything is working
     correctly before making any irreversible change.

(5) Once you have completed the task, make sure to call apis.supervisor.complete_task(). If the task
     asked for some information, return it as the answer argument, i.e. call
```

````
      apis.supervisor.complete_task(answer=<answer>). Many tasks do not require an answer, so in those
      cases, just call apis.supervisor.complete_task() i.e. do not pass any argument.


Here are 2 example episodes of this household game as its players see it -- `>` lines are the player's
      commands, the lines after each are the game's reply, and `think:` lines are the player's own notes;
      each command corresponds to one API of the app above.

```
<ReAct's first demonstration for the task's type>
```
```
<ReAct's second demonstration for the task's type>
```


[[[ assistant ]]]
Understood. I'll consult the app and API docs as needed, work in small verified steps, and call
      `apis.supervisor.complete_task()` when done. What is the task?

[[[ user ]]]
Using these APIs, generate code to solve the actual task.

<the game's opening text: the room and the task>

Task: <task instruction>
````

Prompt 13: ALFWorld: the first turns, our benchmark-neutral seed with the game's two ReAct demonstrations for the task's type appended to the first turn, verbatim. In the first user turn, <app>, <api>, and <answer> are part of the prompt itself.

````
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_descriptions(app_name='alfworld'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_descriptions {"app_name": "alfworld"} -> [{"name": "look", "description":
      "Describe your current location and what is visible there."}, {"name": "inventory", "description":
      "List what you are carrying."}, {"name": "examine", "description": "Look closely at an object or a
      receptacle you are at \u2014 a description, nothing more; for a receptacle it repeats what arriving
      there showed."}, {"name": "go_to", "description": "Move to a receptacle, e.g. `countertop 1`; on
      arrival you are told what is on or in it (a closed one must be opened first)."}, {"name": "open",
      "description": "Open a closed receptacle you are at."}, {"name": "close", "description": "Close an
      open receptacle you are at."}, {"name": "take", "description": "Pick up an object from the
      receptacle you are at. You can carry one object at a time: put it down before taking another."},
      {"name": "put", "description": "Put an object you carry in or on the receptacle you are at."},
      {"name": "heat", "description": "Heat an object you carry with an appliance you are at."}, {"name":
      "cool", "description": "Cool an object you carry with an appliance you are at."}, {"name": "clean",
      "description": "Clean an object you carry with a basin you are at."}, {"name": "use", "description":
      "Switch on an object that can be turned on (a lamp, for instance), at the receptacle you are at."}]
[
 {
  "name": "look",
  "description": "Describe your current location and what is visible there."
 },
 {
  "name": "inventory",
  "description": "List what you are carrying."
 },
 {
  "name": "examine",
  "description": "Look closely at an object or a receptacle you are at \u2014 a description, nothing
      more; for a receptacle it repeats what arriving there showed."
 },
 {
  "name": "go_to",
  "description": "Move to a receptacle, e.g. `countertop 1`; on arrival you are told what is on or in it
      (a closed one must be opened first)."
 },
 {
  "name": "open",
  "description": "Open a closed receptacle you are at."
 },
 {
  "name": "close",
  "description": "Close an open receptacle you are at."
````
````

```
 },
 {
  "name": "take",
  "description": "Pick up an object from the receptacle you are at. You can carry one object at a time:
      put it down before taking another."
 },
 {
  "name": "put",
  "description": "Put an object you carry in or on the receptacle you are at."
 },
 {
  "name": "heat",
  "description": "Heat an object you carry with an appliance you are at."
 },
 {
  "name": "cool",
  "description": "Cool an object you carry with an appliance you are at."
 },
 {
  "name": "clean",
  "description": "Clean an object you carry with a basin you are at."
 },
 {
  "name": "use",
  "description": "Switch on an object that can be turned on (a lamp, for instance), at the receptacle you
      are at."
 }
]
```

[[[assistant]]]

```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='look'))
```

[[[user]]]

Output:

```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "look"} -> {"app_name": "alfworld",
      "api_name": "look", "path": "/alfworld/look", "method": "GET", "description": "Describe your current
      location and what is visible there.", "parameters": [], "response_schemas": {"success":
      {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "look",
 "path": "/alfworld/look",
 "method": "GET",
 "description": "Describe your current location and what is visible there.",
 "parameters": [],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[assistant]]]

```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='inventory'))
```

[[[user]]]

Output:

```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "inventory"} -> {"app_name":
      "alfworld", "api_name": "inventory", "path": "/alfworld/inventory", "method": "GET", "description":
      "List what you are carrying.", "parameters": [], "response_schemas": {"success": {"observation":
      "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "inventory",
 "path": "/alfworld/inventory",
 "method": "GET",
 "description": "List what you are carrying.",
 "parameters": [],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
```

````
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='examine'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "examine"} -> {"app_name":
      "alfworld", "api_name": "examine", "path": "/alfworld/examine", "method": "GET", "description":
      "Look closely at an object or a receptacle you are at \u2014 a description, nothing more; for a
      receptacle it repeats what arriving there showed.", "parameters": [{"name": "object", "required":
      true, "type": "string"}], "response_schemas": {"success": {"observation": "string"}, "failure": "an
      exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "examine",
 "path": "/alfworld/examine",
 "method": "GET",
 "description": "Look closely at an object or a receptacle you are at \u2014 a description, nothing more;
      for a receptacle it repeats what arriving there showed.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='go_to'))
```

[[[ user ]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "go_to"} -> {"app_name": "alfworld",
      "api_name": "go_to", "path": "/alfworld/go_to", "method": "POST", "description": "Move to a
      receptacle, e.g. `countertop 1`; on arrival you are told what is on or in it (a closed one must be
      opened first).", "parameters": [{"name": "receptacle", "required": true, "type": "string"}],
      "response_schemas": {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "go_to",
 "path": "/alfworld/go_to",
 "method": "POST",
 "description": "Move to a receptacle, e.g. `countertop 1`; on arrival you are told what is on or in it
      (a closed one must be opened first).",
 "parameters": [
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='open'))
````

```
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "open"} -> {"app_name": "alfworld",
     "api_name": "open", "path": "/alfworld/open", "method": "POST", "description": "Open a closed
     receptacle you are at.", "parameters": [{"name": "receptacle", "required": true, "type": "string"}],
     "response_schemas": {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "open",
 "path": "/alfworld/open",
 "method": "POST",
 "description": "Open a closed receptacle you are at.",
 "parameters": [
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[assistant]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='close'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "close"} -> {"app_name": "alfworld",
     "api_name": "close", "path": "/alfworld/close", "method": "POST", "description": "Close an open
     receptacle you are at.", "parameters": [{"name": "receptacle", "required": true, "type": "string"}],
     "response_schemas": {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "close",
 "path": "/alfworld/close",
 "method": "POST",
 "description": "Close an open receptacle you are at.",
 "parameters": [
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[assistant]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='take'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "take"} -> {"app_name": "alfworld",
     "api_name": "take", "path": "/alfworld/take", "method": "POST", "description": "Pick up an object
     from the receptacle you are at. You can carry one object at a time: put it down before taking
     another.", "parameters": [{"name": "object", "required": true, "type": "string"}, {"name":
     "receptacle", "required": true, "type": "string"}], "response_schemas": {"success": {"observation":
     "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
```

```
 "api_name": "take",
 "path": "/alfworld/take",
 "method": "POST",
 "description": "Pick up an object from the receptacle you are at. You can carry one object at a time:
     put it down before taking another.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  },
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

```
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='put'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "put"} -> {"app_name": "alfworld",
      "api_name": "put", "path": "/alfworld/put", "method": "POST", "description": "Put an object you
      carry in or on the receptacle you are at.", "parameters": [{"name": "object", "required": true,
      "type": "string"}, {"name": "receptacle", "required": true, "type": "string"}], "response_schemas":
      {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "put",
 "path": "/alfworld/put",
 "method": "POST",
 "description": "Put an object you carry in or on the receptacle you are at.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  },
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

```
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='heat'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "heat"} -> {"app_name": "alfworld",
      "api_name": "heat", "path": "/alfworld/heat", "method": "POST", "description": "Heat an object you
      carry with an appliance you are at.", "parameters": [{"name": "object", "required": true, "type":
      "string"}, {"name": "receptacle", "required": true, "type": "string"}], "response_schemas":
      {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
```

```
 "api_name": "heat",
 "path": "/alfworld/heat",
 "method": "POST",
 "description": "Heat an object you carry with an appliance you are at.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  },
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

```
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='cool'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "cool"} -> {"app_name": "alfworld",
     "api_name": "cool", "path": "/alfworld/cool", "method": "POST", "description": "Cool an object you
     carry with an appliance you are at.", "parameters": [{"name": "object", "required": true, "type":
     "string"}, {"name": "receptacle", "required": true, "type": "string"}], "response_schemas":
     {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "cool",
 "path": "/alfworld/cool",
 "method": "POST",
 "description": "Cool an object you carry with an appliance you are at.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  },
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

```
[[[ assistant ]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='clean'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "clean"} -> {"app_name": "alfworld",
     "api_name": "clean", "path": "/alfworld/clean", "method": "POST", "description": "Clean an object
     you carry with a basin you are at.", "parameters": [{"name": "object", "required": true, "type":
     "string"}, {"name": "receptacle", "required": true, "type": "string"}], "response_schemas":
     {"success": {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "clean",
```

```
 "path": "/alfworld/clean",
 "method": "POST",
 "description": "Clean an object you carry with a basin you are at.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  },
  {
   "name": "receptacle",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[assistant]]]
```python
print(apis.api_docs.show_api_doc(app_name='alfworld', api_name='use'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "alfworld", "api_name": "use"} -> {"app_name": "alfworld",
      "api_name": "use", "path": "/alfworld/use", "method": "POST", "description": "Switch on an object
      that can be turned on (a lamp, for instance), at the receptacle you are at.", "parameters":
      [{"name": "object", "required": true, "type": "string"}], "response_schemas": {"success":
      {"observation": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "alfworld",
 "api_name": "use",
 "path": "/alfworld/use",
 "method": "POST",
 "description": "Switch on an object that can be turned on (a lamp, for instance), at the receptacle you
      are at.",
 "parameters": [
  {
   "name": "object",
   "required": true,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "observation": "string"
  },
  "failure": "an exception is raised"
 }
}
```

[[[assistant]]]
```python
print(apis.api_docs.show_api_doc(app_name='supervisor', api_name='complete_task'))
```

[[[user]]]
Output:
```
INFO apis.api_docs.show_api_doc {"app_name": "supervisor", "api_name": "complete_task"} -> {"app_name":
      "supervisor", "api_name": "complete_task", "path": "/supervisor/complete_task", "method": "POST",
      "description": "Signal that the task is finished. Call once, last, with the answer if the task asks
      a question.", "parameters": [{"name": "answer", "required": false, "type": "string"}],
      "response_schemas": {"success": {"message": "string"}, "failure": "an exception is raised"}}
{
 "app_name": "supervisor",
 "api_name": "complete_task",
 "path": "/supervisor/complete_task",
 "method": "POST",
 "description": "Signal that the task is finished. Call once, last, with the answer if the task asks a
      question.",
 "parameters": [
```

```
  {
   "name": "answer",
   "required": false,
   "type": "string"
  }
 ],
 "response_schemas": {
  "success": {
   "message": "string"
  },
  "failure": "an exception is raised"
 }
}
```

Prompt 14: ALFWorld: the game's API documentation as it enters the agent's context. Each `api_docs` cell is followed by its Output, verbatim from the runs: the list of the game's endpoints, one per command, then the specification of each. The agent never read the specification of `close` in these runs; it is shown as the environment returns it.

C Reproducing Accord

Accord (Jiang et al., 2026) verifies an agent's action before it runs. A second model, the grounding agent, receives the task, the agent's trajectory and the code the agent is about to execute; it may issue read-only calls to observe the environment, and it returns `APPROVE` or `REJECT` together with a note of what it observed. On `REJECT` the code is discarded, the note is appended to the agent's history, and the agent drafts again. The paper releases no code, so we re-implemented the method from its description, its Algorithm 1 and its prompt appendix. This section records what the paper fixes, what it leaves open and the choice we made in each case, the prompt and model configuration, and how our numbers compare with the reported ones.

What the paper specifies. The backbone and its decoding (Qwen3.5-27B in FP8, temperature 0); the grounding-agent prompt (its Appendix A.5, reproduced verbatim in §D); the inputs to that prompt, namely the pending code, the context and the task; three policy items added to the agent's instructions, numbered (10) to (12) in the paper: inspect raw content before writing parsing logic, re-examine the API specification when a response is not as expected, and list the APIs of every app before starting the task; and the control flow of Algorithm 1: an action that calls a sensitive API is withheld, the grounding agent observes and decides, and on `REJECT` the attempt is removed from the agent's history and the grounding agent's evidence is appended in its place.

What it leaves open, and what we chose. Table 6 lists the remaining decisions and our choice for each.

Table 6: Decisions the paper leaves open and the choice taken in our implementation of ACCORD.

Left open by the paper	Our choice
Which calls are sensitive	Every endpoint whose HTTP method is not `GET` (`POST`, `PATCH`, `DELETE`), plus `supervisor.complete_task`. The set is computed from the code before execution, by a syntax-tree walk over `apis.<app>.<endpoint>(` calls.
Grounding-agent model	The same model and decoding as the agent.
Grounding-agent memory	One chat per task, kept across verifications. Each verification after the first shows only the steps taken since the previous one.
Where observations run	In the agent's own namespace, so the grounding agent can read the agent's variables. Write endpoints raise instead of executing, and the observation code runs inside a function so that it cannot rebind the agent's variables.
Observation budget	Five observation rounds per verification. A reply with neither code nor a decision is answered with a one-line reminder of the output format and costs a round. When the budget is exhausted the decision is forced to `APPROVE`.
Rejections per task	Three. After the third, a `REJECT` is ignored and the code executes.
Message on `REJECT`	The `OBSERVE` note, appended to the last user turn as "Your previous code was NOT executed. Before executing sensitive API calls, the environment was observed and the following was reported: … Revise your code accordingly." (§D). The discarded attempt is removed, as Algorithm 1 specifies; the wording around the note is ours.
Layout of the grounding input	Four labeled blocks: the task instruction, the agent trajectory, the code about to execute, and the triggered sensitive APIs with their HTTP methods (§D).
Policy item (12)	The item tells the agent to begin by calling `show_api_descriptions(app_name="")`, which lists the APIs of every app. Left to the model, it repeats this call until the context is full, so we issue the call for it as the first step of every task and let the environment answer the empty name with the list of all apps.
Step limit	None. An episode ends when the agent calls `complete_task` or when the context limit is reached.

Base prompt and demonstrations. The paper names no base prompt for the main agent, and its policy items are numbered from (10). We use the ACE generator prompt (Zhang et al., 2026) with its seed playbook, whose key instructions end at (8), and insert the three policy items after the last key instruction. AppWorld's official prompt, the other public candidate, ends its demonstration by passing an answer to `complete_task`, and with it the agent passes answers on tasks that expect none. The ACE prompt carries three demonstrations. For Table 1, where every method receives one demonstration, we keep only the first: the other two demonstration blocks are removed, and the playbook, the key instructions, and the task turn are unchanged. Table 7 reports both versions, and Figure 3 varies the number of demonstrations for every method.

Backbone and decoding. Qwen3.5-27B in FP8 at temperature 0, for both the agent and the grounding agent. We report ACCORD with thinking on and with thinking off, since the paper marks only its 9B model as non-thinking. No cap is placed on generated tokens other than the 262k-token context. As for every other agent in this paper, observations are truncated to 20k characters and the history to 400k characters.

Evaluation. AppWorld test-normal (168 tasks) and test-challenge (417 tasks), scored by the official evaluator: TGC is the fraction of tasks whose checks all pass, and SGC the fraction of scenarios whose tasks all pass. Each row is the mean and sample standard deviation over three runs.

Result. Table 7 sets our runs against the paper's Table 1. With the three-demonstration prompt, our reproduction meets the reported test-challenge numbers with thinking on (71.9 against 69.8 TGC; 59.0 against 53.2 SGC) and falls short on test-normal by 4.3 TGC points (77.2 against 81.5), more than the run-to-run spread. The budgets, the memory of the grounding agent, and the base prompt are all unstated in the paper, so we cannot attribute the gap to any one choice in Table 6. With one demonstration, ACCORD completes more test-normal tasks (79.4 TGC) and fewer test-

challenge tasks (63.5 TGC) than with three. The budgets rarely bind: in the first one-demonstration test-challenge run with thinking, the grounding agent was invoked 2,916 times over the 417 tasks and rejected 171 actions; the observation budget ran out 9 times, and the rejection cap overrode 3 rejections.

Table 7: ACCORD as reported and as reproduced here (Qwen3.5-27B, FP8, temperature 0), with the ACE prompt's three demonstrations and with its first demonstration only. The one-demonstration rows are those of Table 1. The paper does not state how many demonstrations its prompt carried.

			test-normal		test-challenge	
Thinking	Source	Demos	TGC	SGC	TGC	SGC
on	reported (Jiang et al., 2026)	–	81.5	67.9	69.8	53.2
on	reproduced (3 runs)	3	77.2 ±1.4	62.5 ±4.7	71.9 ±1.0	59.0 ±1.9
on	reproduced (3 runs)	1	79.4 ±1.8	65.5 ±5.7	63.5 ±1.7	48.4 ±1.5
off	reproduced (3 runs)	3	76.4 ±3.9	62.5 ±3.1	69.3 ±1.8	52.3 ±1.5
off	reproduced (3 runs)	1	76.2 ±1.2	59.5 ±2.7	55.0 ±0.4	44.8 ±2.3

D ACCORD PROMPTS

The ACCORD agent is the ACE generator prompt (Zhang et al., 2026) (with its three demonstrations, or its first one only in Table 1; Section C), with the paper's policy items (10) to (12) inserted after its eighth key instruction (Prompt 18) and the seed playbook of Prompt 19 in its playbook slot. The grounding agent runs on Prompt 15, the paper's Appendix A.5 copied verbatim with its three worked examples; its final turn is filled with the reflection context of Prompt 16, whose layout is ours. When the decision is `REJECT`, the agent receives the `OBSERVE` note wrapped as in Prompt 17.

```
USER:
You are a verification agent. Before an AI assistant executes code with sensitive API calls, you must
      verify that the code is correct and safe to execute.

## Core principle: Observe, then verify, then decide

Your primary tool is **observation** -- executing read-only Python code to inspect the actual environment
      state. Do NOT reason about correctness from the code alone. Instead, query the environment to gather
      evidence, then use that evidence to verify or refute the agent's approach. Only after you have
      gathered sufficient evidence should you make a decision.

## How to observe

Write read-only Python code in ```python ...``` blocks. You have access to the same `apis` object and all
      variables from the main agent's execution environment (e.g., access_token, results from prior steps).

**What you can call in observations:**
- Any read-only API (write operations will be blocked)
- API documentation: `apis.api_docs.show_api_descriptions(app_name=...)` and
      `apis.api_docs.show_api_doc(app_name=..., api_name=...)` to check API specifications

**Leverage the trajectory before observing:**
You have access to the agent's full execution trajectory, including all intermediate print outputs and
      API responses. Before writing any observation code, review the trajectory carefully to identify what
      has already been explicitly checked or printed. If the trajectory already contains the evidence you
      need (e.g., the agent printed a query result, iterated through pages, or displayed entity details),
      you can trust that printed output as evidence -- there is no need to re-query it. Only observe what
      is **missing or ambiguous** in the trajectory.

**Observation strategies (apply only to gaps not already covered by the trajectory):**
- **Re-query the data source**: If the agent operated on data it never actually inspected in the
      trajectory, re-fetch the raw data yourself to verify that the agent's logic matches the real data
      structure and content.
- **Check API specifications**: If the agent did not consult the API docs before calling an API, use
      `apis.api_docs.show_api_doc()` to verify it used correct parameter names, types, and values.
- **Verify entity identity**: If the trajectory does NOT show the agent confirming it resolved an
      ambiguous entity (e.g., which "John" or which "note"), observe the environment to verify.
- **Test boundary conditions**: If the trajectory does NOT show the agent handling pagination
      exhaustively or checking edge cases, re-query to verify completeness.

## Verification checklist

For each verification item below, use observation to gather evidence before making a judgment.
```

```
### 1. Constraint satisfaction
Re-read the task instruction word by word. Extract and QUOTE every explicit constraint verbatim (e.g.,
      recipients, amounts, dates, conditions, filters, set operations like "X that are in Y"). Do NOT
      paraphrase -- copy the exact wording. Then for each quoted constraint, **observe** the environment
      to verify the code satisfies it. Only REJECT if your observation confirms a constraint is violated
      or ignored.

### 2. Ambiguity resolution
Identify any ambiguity in the agent's actions -- such as unverified entity references (e.g., which
      "John"? which "meeting"?), assumed IDs, or implicit choices made without evidence. Also check
      whether the action assumes a particular environment state (e.g., an empty cart before checkout, no
      existing draft before creating one) that has not been verified. For each ambiguity, **observe the
      environment yourself first** to check whether the agent's assumption is actually correct. Only
      REJECT if your observation reveals a conflict with the agent's action -- in that case, specify the
      exact API calls the agent should make to resolve it.

## Workflow

1. Read the agent's trajectory since the last reflection (including all intermediate steps, API
      responses, and the code it is about to execute) and the task instruction.
2. Identify what needs to be verified -- constraint satisfaction and ambiguity resolution (including
      unverified environment state).
3. **Review the trajectory** -- check which verification items already have explicit evidence in the
      agent's printed outputs. Note these as "covered by trajectory".
4. **Observe** -- for items NOT covered by the trajectory, write one code block to query the environment.
      Wait for the result.
5. Repeat step 4 as needed until you have enough evidence for all items.
6. **Decide** -- if all checks pass, APPROVE with an OBSERVE summary (what you observed and what it
      verified). If any check fails, REJECT with an OBSERVE directive specifying the exact API calls the
      main agent should make before proceeding.

## Output format

When observing (one code block per message):
```python
# Describe what you are checking
print(...)
```

When deciding (only after sufficient observation):
DECISION: APPROVE
OBSERVE: <what you observed (which APIs you called, what data you checked) and what it verified (which
      constraints and assumptions were confirmed)>

or

DECISION: REJECT
OBSERVE: <specify the API calls the main agent should make and what information to pay attention to -- do
      NOT include your own judgment or conclusion>

Key rules

- NEVER output observation code and a DECISION in the same message. Observe first, wait for results, then
      decide in a separate message.
- Write one code block per observation step. Do not combine multiple unrelated checks.
- You may take multiple observation steps before deciding.
- If you cannot determine correctness after observation, default to APPROVE.
- Do NOT attempt to fix or rewrite the code. Only APPROVE or REJECT.

Examples

USER:
[Reflection context omitted: contains task instruction, full agent trajectory, code about to execute, and
      triggered sensitive APIs]

ASSISTANT:
The agent is about to execute:
```python
message = "Here are my David Fincher movie recommendations: Fight Club, The Social Network"
apis.phone.send_text_message(access_token=phone_access_token, phone_number=christopher_phone,
      message=message)
```

Task: "Christopher has asked for my movie recommendations via phone text message. Reply to them with a
      list of comma-separated movie titles from my Simple Note account as per their request."

Constraints to verify:
```

```
- "Christopher has asked ... via phone text message" -- need to observe the original message to confirm
      what Christopher asked for
- "movie titles from my Simple Note account" -- need to observe the note content to confirm the movies
      match
- "as per their request" -- need to confirm the movies satisfy Christopher's specific request

Let me observe Christopher's original message first.
```python
messages = apis.phone.search_text_messages(access_token=phone_access_token,
      phone_number=christopher_phone, query="movie")
print(messages)
```

USER:
Output:
```
[
  {
    "text_message_id": 16794,
    "sender": {
      "name": "Christopher Burch",
      "phone_number": "4275730168"
    },
    "message": "Hey, I'm planning to watch a movie. Do you have any recommendations for a movie from
      David Fincher?",
    "sent_at": "2023-05-18T10:17:40"
  }
]
```

ASSISTANT:
Christopher specifically asked for "recommendations for a movie from David Fincher". The agent's message
      includes "Fight Club" and "The Social Network". Now I need to observe the Simple Note content to
      verify these are actually David Fincher movies from the note.
```python
movie_note = apis.simple_note.show_note(access_token=simple_note_access_token, note_id=1608)
print(movie_note['content'])
```

USER:
Output:
```
# Movie Recommendations

Interstellar
- director: Christopher Nolan
- genre: Sci-Fi, Drama

Fight Club
- director: David Fincher
- genre: Drama

The Social Network
- director: David Fincher
- genre: Biography, Drama

The Godfather
- director: Francis Ford Coppola
- genre: Crime, Drama
```

ASSISTANT:
DECISION: APPROVE
OBSERVE: Called `phone.search_text_messages` to read Christopher's original request, and
      `simple_note.show_note` to read the note content. Verified that (1) Christopher asked for
      "recommendations for a movie from David Fincher", (2) Fight Club and The Social Network are both
      listed as David Fincher movies in the note, (3) the reply format uses comma-separated titles as
      required.

USER:
[Reflection context omitted: contains task instruction, full agent trajectory, code about to execute, and
      triggered sensitive APIs]

ASSISTANT:
The agent is about to execute:
```python
apis.supervisor.complete_task(answer=23)
```
```

````
Task: "How many playlists do I have in Spotify?"

I need to independently verify the count of 23. Let me observe by re-querying the playlist library with
     full pagination.
```python
playlists = []
page_index = 0
while True:
    page = apis.spotify.show_playlist_library(access_token=spotify_access_token, page_index=page_index)
    if not page:
        break
    playlists.extend(page)
    page_index += 1
print(f"Total playlists: {len(playlists)}, pages iterated: {page_index}")
```

USER:
Output:
```
Total playlists: 23, pages iterated: 3
```

ASSISTANT:
DECISION: APPROVE
OBSERVE: Called `spotify.show_playlist_library` with pagination (page_index 0-2), collected all 23
     playlists across 3 pages. Verified that the total count matches the agent's answer of 23.

------------------------------------------------

USER:
[Reflection context omitted: contains task instruction, full agent trajectory, code about to execute, and
     triggered sensitive APIs]

ASSISTANT:
The agent is about to execute:
```python
apis.venmo.send_money(access_token=venmo_access_token, recipient_id=recipient['user_id'], amount=50.00,
     note="Dinner split")
```

Task: "Split the $120 dinner bill equally among me and my 3 friends on Venmo."

Constraints to verify:
- "Split the $120 dinner bill equally" -- need to verify the amount is correct
- "among me and my 3 friends" -- 4 people total, each share = $30, but agent sends $50

The amount looks wrong. Let me observe to confirm.
```python
total = 120
people = 4  # "me and my 3 friends" = 4 people
share = total / people
print(f"Correct share per person: ${share}")
print(f"Agent is sending: $50.00")
```

USER:
Output:
```
Correct share per person: $30.0
Agent is sending: $50.00
```

ASSISTANT:
DECISION: REJECT
OBSERVE: Before sending money, call `apis.venmo.show_profile(access_token=venmo_access_token)` to check
     whether the current user should be included in the split. Pay attention to the total number of
     people involved and recalculate the per-person share accordingly.

------------------------------------------------

USER:
Now verify the following code that the agent is about to execute.

{{ reflect_context }}
````

Prompt 15: The grounding-agent prompt, Appendix A.5 of the ACCORD paper, verbatim. The placeholder on the last line is replaced by the reflection context of Prompt 16.

```
Task instruction: {task}
```

```
Agent trajectory (new steps since the last reflection):
{steps}
Code about to execute:
```python
{code}
```

Triggered sensitive APIs: apis.{app}.{endpoint} ({METHOD}), ...
```

Prompt 16: The reflection context rendered into the grounding agent's final turn. On the first verification of a task the heading reads "Agent trajectory:" and the whole trajectory follows; later verifications show only the steps since the previous one.

```
Your previous code was NOT executed. Before executing sensitive API calls, the environment was observed
     and the following was reported:
{observe}

Revise your code accordingly.
```

Prompt 17: The message appended to the agent's last user turn when the grounding agent rejects; the discarded attempt itself is removed from the history.

```
(10) If the content or format of returned information is uncertain, inspect the actual raw content first
     (e.g., via read/show APIs) and do not guess structure or write parsing logic based on assumptions.

(11) If the content of returned information is not as expected, re-examine the API specification, your
     request parameters, and any assumptions you made about the response structure. Adjust your approach
     accordingly and verify the result is correct before proceeding to the next step.

(12) Before executing the task, you MUST call show_api_descriptions(app_name="") to get the complete API
     list for ALL apps at once. Do NOT selectively check only apps you think are relevant--you must
     examine every app's APIs exhaustively.
```

Prompt 18: Policy items (10) to (12) of the ACCORD paper, inserted after key instruction (8) of the agent prompt.

```
## STRATEGIES AND HARD RULES
[shr-00001] Make sure to end code blocks with ``` followed by a newline(\n).
[shr-00005] Always look at API specifications (using apis.api_docs.show_api_doc) before calling an API.
[shr-00006] Write small chunks of code and only one chunk of code in every step. Make sure everything is
     working correctly before making any irreversible change.

## APIs TO USE FOR SPECIFIC INFORMATION
[api-00004] You can use the "supervisor" app to get information about my accounts and use the "phone" app
     to get information about friends and family.

## USEFUL CODE SNIPPETS AND TEMPLATES

## COMMON MISTAKES AND CORRECT STRATEGIES

## PROBLEM-SOLVING HEURISTICS AND WORKFLOWS
[psw-00002] Remember you can use the variables in your code in subsequent code blocks.
[psw-00007] Many APIs return items in "pages". Make sure to run through all the pages by looping over
     `page_index`.

## VERIFICATION CHECKLIST

## TROUBLESHOOTING AND PITFALLS:

## OTHERS
[misc-00003] Remember that the email addresses, access tokens and variables (e.g. spotify_password) in
     the example above are not valid anymore.
[misc-00008] Once you have completed the task, make sure to call apis.supervisor.complete_task(). If the
     task asked for some information, return it as the answer argument, i.e. call
     apis.supervisor.complete_task(answer=<answer>). Many tasks do not require an answer, so in those
     cases, just call apis.supervisor.complete_task() i.e. do not pass any argument.
```

Prompt 19: The seed playbook placed in the playbook slot of the ACE generator prompt, used by ACCORD and by REACT; our agent uses no playbook.

E COST MEASUREMENT

Token counts. For every model call, the serving engine reports the number of input tokens it received and the number of tokens it generated: SGLang for the Qwen backbones, which we serve ourselves, and the API provider (OpenRouter) for GLM-5.3-flash and MiniMax M2.7. We sum both counts over all model calls of an episode, including the verification calls of our agent and the grounding calls of ACCORD. Input tokens count every token sent on every call, including a prefix that a serving cache could reuse; generated tokens include thinking tokens. For each task we average over the runs of a configuration, and then average over the tasks of the split. When a task has more than one trial in a run (a re-run after a lost worker), we use the latest.

Cost measures. The first measure is the number of generated tokens per task. The second prices input and generated tokens per task at one common price, the OpenRouter list price of Qwen3.5-27B ($0.195 per million input tokens and $1.56 per million generated tokens, September 2026), so that the measure reflects how many tokens a harness uses rather than which model is cheap. Figure 5 divides both measures, and the task goal completion, by those of REACT with thinking on the same backbone. An agent lies on the frontier of its backbone if no other agent on the same backbone costs no more and completes more tasks.

All measured points. Table 8 lists every point on both test splits, including Qwen3-30B-A3B, which Figure 5 omits. On Qwen3.5-27B, the runs are those of Table 1 (one demonstration, three runs). The other backbones have one run each, and on the two API backbones the baselines use the released three-demonstration prompt, so comparisons across backbones mix configurations; comparisons within a backbone do not.

Table 8: Task goal completion and cost of every backbone and harness on AppWorld, with the backbone model's thinking on (✓) and off (✗). Tokens: generated tokens per task, in thousands. Cost: input and generated tokens per task priced at the common price, in dollars. In parentheses, the ratio to REACT with thinking on the same backbone. ↑: higher is better; ↓: lower is better. Mean ± standard deviation of TGC over three runs on Qwen3.5-27B; one run on the other backbones. Bold is the best TGC within its thinking block, underline the runner-up.

		Test (normal)			Test (challenge)		
	Method	TGC↑	Tokens↓	Cost↓	TGC↑	Tokens↓	Cost↓
Qwen3.5-27B							
	REACT	72.4 ±1.9	2.7 (1.0×)	0.032 (1.0×)	58.4 ±0.8	3.6 (1.0×)	0.047 (1.0×)
✓	ACCORD	<u>79.4</u> ±1.8	12.9 (4.7×)	0.092 (2.9×)	<u>63.5</u> ±1.7	10.5 (2.9×)	0.130 (2.7×)
	BLACKSWAN	**88.3** ±3.0	17.7 (6.5×)	0.235 (7.4×)	**84.7** ±1.3	22.9 (6.4×)	0.447 (9.4×)
	REACT	70.0 ±2.5	2.4 (0.9×)	0.028 (0.9×)	48.0 ±0.8	2.2 (0.6×)	0.044 (0.9×)
✗	ACCORD	<u>76.2</u> ±1.2	3.9 (1.4×)	0.079 (2.5×)	<u>55.0</u> ±0.4	4.9 (1.4×)	0.112 (2.4×)
	BLACKSWAN	**84.1** ±1.5	5.3 (2.0×)	0.195 (6.1×)	**81.5** ±2.9	7.3 (2.1×)	0.366 (7.7×)
Qwen3-30B-A3B							
	REACT	<u>32.1</u>	22.4 (1.0×)	0.055 (1.0×)	<u>15.8</u>	29.9 (1.0×)	0.092 (1.0×)
✓	ACCORD	29.8	30.0 (1.3×)	0.101 (1.8×)	13.7	47.9 (1.6×)	0.175 (1.9×)
	BLACKSWAN	**52.4**	43.8 (2.0×)	0.193 (3.5×)	**34.8**	54.0 (1.8×)	0.292 (3.2×)
Qwen3-4B							
	REACT	<u>23.2</u>	60.1 (1.0×)	0.134 (1.0×)	<u>12.2</u>	64.6 (1.0×)	0.154 (1.0×)
✓	ACCORD	14.9	108.6 (1.8×)	0.287 (2.1×)	8.6	109.4 (1.7×)	0.269 (1.8×)
	BLACKSWAN	**41.7**	77.7 (1.3×)	0.293 (2.2×)	**27.6**	96.0 (1.5×)	0.359 (2.3×)
GLM-5.3-Flash							
	REACT	<u>83.9</u>	1.3 (1.0×)	0.027 (1.0×)	<u>77.5</u>	1.6 (1.0×)	0.041 (1.0×)
✓	ACCORD	82.1	2.6 (2.0×)	0.058 (2.1×)	76.7	3.3 (2.0×)	0.076 (1.9×)
	BLACKSWAN	**91.1**	6.9 (5.3×)	0.226 (8.3×)	**83.5**	8.6 (5.2×)	0.363 (8.9×)
MiniMax M2.7							
	REACT	61.3	3.1 (1.0×)	0.038 (1.0×)	53.0	3.5 (1.0×)	0.053 (1.0×)
✓	ACCORD	<u>71.4</u>	6.9 (2.2×)	0.090 (2.4×)	<u>63.1</u>	8.1 (2.3×)	0.121 (2.3×)
	BLACKSWAN	**77.4**	12.8 (4.1×)	0.199 (5.2×)	**73.9**	14.4 (4.1×)	0.302 (5.8×)

F QUALITATIVE EXAMPLES

Tables 9 and 10 show two tasks from the AppWorld test-challenge split, run with Qwen3.5-27B and thinking off: ACCORD with the ACE prompt's three demonstrations on the left and BLACKSWAN on the right, one run each. On both tasks the two agents take the same first steps and propose the same action. ACCORD's grounding agent reads the environment to confirm the action and approves it once what it reads agrees with the action. BLACKSWAN instead asks what would make the action wrong and reads for that. In the first task it finds a third cancellation, worded without the search terms; in the second it finds that a returned wrench has the product type "crescent wrench" although its name does not say so. BLACKSWAN then revises the action and completes both tasks, while ACCORD fails both. Model outputs are verbatim, shortened with [...]; green marks a right verification, red a wrong one, and blue the ground truth.

Table 9: Task c1091c7_1: disable the alarms of meetings that coworkers canceled by email.

ACCORD **(approves with one alarm still on)**	BLACKSWAN **(refutes and revises)**
Task: I have alarms for all my work meetings. Some of my coworkers have canceled their meeting with me via email. Please disable the corresponding alarms from my phone. (GT: set `enabled=False` on alarms 744, 745, and 743. Three coworkers canceled; Stephen's email is titled "Skip this time?" and reads "Cannot make it.") **Shared prefix:** both agents search Gmail for `"cancel meeting"`, find the two "Cancel Meeting?" threads (Stacy → alarm 744, Kristin → alarm 745), and propose `phone.delete_alarm(alarm_id=744)`.	
Round 1 **Grounding agent:** reads the `update_alarm` docs, then: **DECISION: REJECT.** OBSERVE: The task instruction says to "disable" the alarms, not delete them. The `update_alarm` API has an `enabled` parameter that can be set to `false` to disable an alarm without deleting it. [...] ✓	**Round 1** **Suspect:** (1) The task asks to 'disable' the alarms, but the code calls `delete_alarm` which permanently removes them. (2) The code only deletes the alarm for Stacy (ID 744). [...] alarm labeled '1-1 with Kristin' (ID 745) also needs to be disabled, but this code cell does not address it. **Test:** reads the `update_alarm` docs → **both confirmed**. ✓
Both revise to `update_alarm(alarm_id=744, enabled=False)` and `update_alarm(alarm_id=745, enabled=False)`.	
Round 2 — the two updates run; the agent proposes `supervisor.complete_task()`. **Grounding agent:** "Let me verify that no other cancellation emails were missed" → probes `show_inbox_threads(query="cancel")` → Thread 47823: "Cancel Meeting?" from Stacy Maldonado — handled. Thread 47825: "Cancel Meeting?" from Kristin White — handled. Thread 47831: "Cancel Reunion" from Leslie Ball — this is about a reunion, not a work meeting. Other threads are Amazon notifications, dinner party RSVPs, [...] none are meeting cancellations. **DECISION: APPROVE. OBSERVE: [...] Confirmed that only two work meeting cancellations were found** (Stacy Maldonado and Kristin White), both of which have been handled [...] The task is complete. × (alarm 743 "1-1 with Stephen" stays enabled)	**Round 2** — the agent proposes the two `update_alarm` calls. **Expect:** Disables alarms 744 ('1-1 with Stacy') and 745 ('1-1 with Kristin') by setting `enabled=False`, matching the two meeting cancellations found in emails [...] **Suspect:** [...] **The real risk is if there are other cancellation emails not captured by the query 'cancel meeting' (e.g., 'reschedule', 'postpone', 'cannot make it') that correspond to alarms.** If other coworkers canceled via different phrasing, their alarms would remain enabled [...] `ref:` `phone.update_alarm.alarm_id`. *[Two further suspicions, on the label–sender mapping and on the meaning of 'disable', are dismissed in the test and omitted here.]* **Test:** probes `show_inbox_threads(query="reschedule OR postpone OR cannot make OR conflict")` → thread 47821 "Skip this time?"; then `show_thread(47821)` → sender Stephen Mccoy, "Cannot make it.", and alarm 743 "1-1 with Stephen" is enabled. **Verdict:** **confirmed** — Email thread 47821 from Stephen Mccoy says 'Cannot make it' (cancellation), and alarm 743 '1-1 with Stephen' exists but was not included in the proposed code. **Reframe:** disables 744, 745, and 743. ✓ (all three alarms disabled)

Table 10: Task 3650990_3: has the user returned a crescent wrench this or last month?

ACCORD **(sees the wrench, approves "no")**	BLACKSWAN **(asks what would make "no" wrong)**
Task: Have I returned any crescent wrench on amazon in this or last month? Say yes or no. (GT: yes — "Craftsman 8-Inch Adjustable Wrench", `product_type` "crescent wrench", returned on 2023-04-28.) **Shared prefix:** both agents fetch all 33 returns, keep those from April/May 2023 whose `product_name` contains "crescent wrench" (none), and propose `supervisor.complete_task(answer="no")`.	
Grounding agent: "Let me observe to verify the agent's logic by checking if there are any returns with 'crescent' or 'wrench' in the product name that might have been missed." → probes `all_returns` → Output: Total wrench returns in April/May 2023: 1 — Craftsman 8-Inch Adjustable Wrench (initiated: 2023-04-28). Total crescent returns in April/May 2023: 0. **DECISION: APPROVE. OBSERVE:** Verified by re-analyzing all 33 returns that (1) the date range April/May 2023 is correct [...], (2) **there is 1 wrench return ("Craftsman 8-Inch Adjustable Wrench") but 0 crescent wrench returns** in the target months, (3) the agent's case-insensitive search for "crescent wrench" correctly found no matches. The answer "no" is accurate. × (answer: no; GT: yes)	**Expect:** The code will mark the current task as complete with the answer 'no', indicating that the user has not returned any crescent wrench on Amazon in the current or previous month, based on the analysis of all returned items. **Suspect:** The code searched for the exact phrase 'crescent wrench' in the product name. **However, the returned item 'Craftsman 8-Inch Adjustable Wrench' is a type of crescent wrench.** [...] The code did not account for synonyms or broader category matching. `ref: task`. **Test:** probes `amazon.search_products(query="crescent wrench")` → Output: `product_id` 2139, `product_type` "crescent wrench", name "Craftsman 8-Inch Adjustable Wrench" [...] **Verdict:** **confirmed** — Search results show `product_id` 2139 has `product_type` 'crescent wrench' but name 'Craftsman 8-Inch Adjustable Wrench'. The return list contains this `product_id`. **Reframe:** re-checks the returns by `product_id` → found, 2023-04-28 → `complete_task(answer="yes")`. √ (answer: yes)