METAACT: SELF-EVOLVING EXECUTABLE ACTION CONSTRAINTS FOR EMBODIED LANGUAGE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving language agents have emerged as a promising paradigm for continuously improving agent behavior through experience. However, most existing approaches evolve textual artifacts, which can influence action preferences but cannot reliably enforce action admissibility in embodied environments with discrete and often irreversible actions. We propose MetaAct, which shifts self-evolution from textual guidance to executable action constraints by synthesizing GEAR (Grounded Executable Action Rules) from execution feedback. Each rule is represented as a typed specification, compiled into a deterministic guard, and validated offline through trajectory replay without additional model calls or environment interactions. Across four benchmarks, MetaAct improves task performance; on ScienceWorld and WebShop, violations drop from 70% to 0% and 40% to 1%, with score gains of +10.1 and +8.0. Our code is available at `https://anonymous.4open.science/r/MetaAct-D567`.

1 INTRODUCTION

Language agents can improve by modifying textual artifacts, including specialist roles and communication structures (Hong et al., 2024; Wu et al., 2023), natural-language reflections and distilled lessons from previous failures (Shinn et al., 2023; Zhao et al., 2024), and evolving task contexts (Zhang et al., 2026). These methods alter the context from which the model generates its next output and have shown value in tasks where behavior is mediated primarily through text. Embodied interaction differs in a consequential way. At each step, an agent selects a discrete action from an environment-defined set; once executed, the action changes the environment and may be difficult or impossible to reverse.

Figure 1 illustrates this distinction in ScienceWorld, where focusing on the object under test before the thermometer can terminate the episode immediately. Distilling this failure into a natural-language lesson gives the planner the correct ordering rule, but leaves the violating action available. Compiling the same rule into an executable constraint instead removes that action from the admissible set until its prerequisite is satisfied. Prompt adaptation therefore reshapes which action the model prefers, whereas executable constraints directly restrict which actions can be executed, in the spirit of action masking and shielding in reinforcement learning (Huang & Ontañón, 2022; Alshiekh et al., 2017).

To address this gap, we introduce MetaAct, a framework that synthesizes executable action constraints from execution feedback. Rather than letting the model write executable code directly (Wang et al., 2023a), each constraint is represented as a typed GEAR (Grounded Executable Action Rules) specification and compiled into a deterministic action guard. Candidate guards are validated offline using previously logged trajectories and, once accepted, can be reused across episodes without additional environment interactions or LLM token cost at inference.

We isolate the effect of representation through a controlled comparison in which the failure evidence, evaluation instances, language model, action budget, and online lesson-evolution mechanism are held fixed; the only difference is whether the additional evidence is retained as natural-language guidance (Shinn et al., 2023; Chen et al., 2024; Sarukkai et al., 2025) or compiled into an executable constraint. Although the lessons state the correct rules, the episode-level violation rate is unchanged at 70% on ScienceWorld and 39% versus 40% on WebShop, whereas the compiled constraints reduce it to 0% and 1% and improve task scores by $+10.1$ and $+8.0$. Our contributions are:

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.

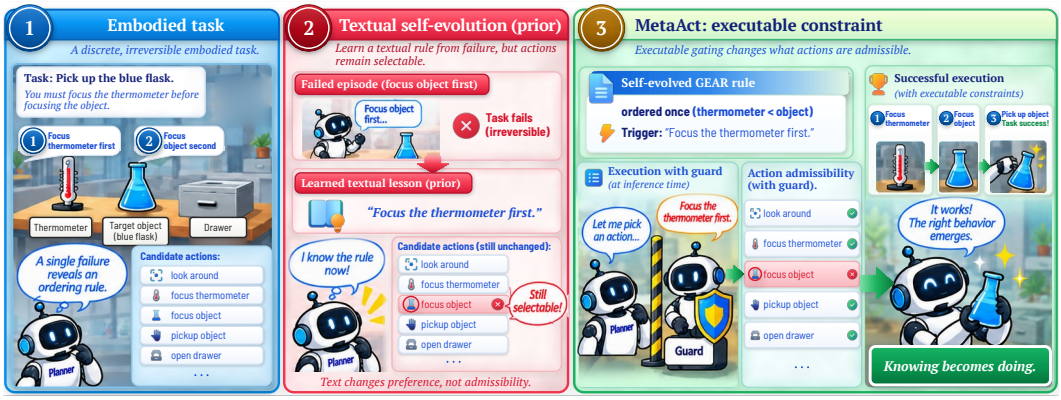


Figure 1: **From textual lessons to executable constraints.** (1) In ScienceWorld, a single irreversible failure reveals an ordering rule: the thermometer must be focused before the target object. (2) Textual self-evolution distills the failure into a correct lesson, but the violating action remains in the candidate set and is still selectable. (3) MetaAct compiles the same rule into a GEAR guard that removes the violating action from the admissible set at execution time: text changes preference, whereas the constraint changes admissibility.

1. We introduce MetaAct, a framework that synthesizes typed, executable action constraints from execution feedback and compiles them into deterministic guards.
2. We develop an offline replay procedure that validates and refines candidate constraints from logged trajectories without additional model calls or environment interactions.
3. We evaluate MetaAct on four embodied-language benchmarks, showing consistent task-level improvements. On ScienceWorld and WebShop, executable constraints reduce violation rates from 70% to 0% and from 40% to 1%, while improving task scores by $+10.1$ and $+8.0$, respectively.

2 Related Work

2.1 Textual and Parameter-Space Adaptation

Most language-agent adaptation changes model context or parameters. Reflections, distilled lessons, and test-time memories reuse prior experience (Shinn et al., 2023; Zhao et al., 2024; Zhang et al., 2026); role, topology, and prompt optimization restructure how the model is queried (Hong et al., 2024; Wu et al., 2023; Wang et al., 2026b; Khattab et al., 2024); replanning intervenes during execution (Sun et al., 2023; Kim et al., 2025); and fine-tuning updates weights (Zeng et al., 2024). However, all of these act only through the model's subsequent interpretation of the artifact; MetaAct's advantage is that feedback is compiled into a deterministic action gate whose effect does not depend on the model's compliance.

2.2 Executable Artifacts

A complementary line generates artifacts that act directly: verified skill libraries (Wang et al., 2023a), policy code and situated programs for embodied planners (Liang et al., 2023; Singh et al., 2022; Liu et al., 2023), affordance-grounded closed-loop feedback (Huang et al., 2023), and program libraries validated by execution (Ellis et al., 2021). However, these methods add executable behavior that must be run to be assessed; MetaAct's advantage is a subtractive guard whose effect on logged trajectories can be checked by replay before deployment.

2.3 Gating and Runtime Enforcement

Action-space control spans symbolic preconditions (Fikes & Nilsson, 1971), shielding and action masking in reinforcement learning (Alshiekh et al., 2017; Huang & Ontañón, 2022), and constraints

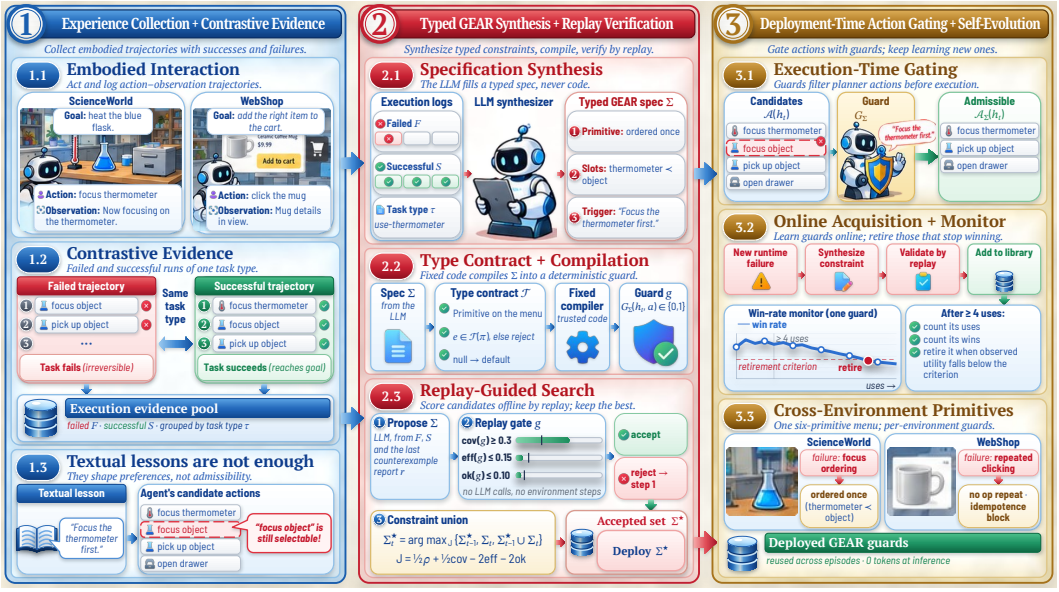


Figure 2: **Overview of MetaAct.** (1) Failed and successful trajectories provide contrastive execution evidence. (2) The LLM synthesizes typed GEAR specifications, which are compiled into deterministic guards and verified offline by trajectory replay. (3) Accepted guards constrain action execution at deployment time and can be continually expanded from new interaction feedback.

induced from traces (Yang et al., 2007; Guan et al., 2023; Mordoch et al., 2024). Closest to ours are runtime guards for LLM agents: AgentSpec enforces LLM-written rules (Wang et al., 2026a) and OntoGuard checks actions against ontologies mined from demonstrations (Liu et al., 2026). However, both fix their rules before deployment without behavioral verification; MetaAct's advantage is a gate synthesized from the agent's own execution feedback and validated by replay.

3 Method: Self-Evolving GEAR

3.1 Typed GEAR Specifications

MetaAct places the LLM at the specification level rather than letting it write executable code: the model fills a typed specification and a fixed compiler produces the executable artifact. This trades some expressiveness for deterministic compilation and offline validation (Section 2).

As illustrated in Figure 2, a GEAR specification contains three components: a rule primitive, typed parameters instantiated through extractor slots, and a trigger message shown to the planner when the rule fires. The primitive is selected from a fixed menu of six operators covering the failure patterns observed during development: `ordered_once` enforces a one-time precedence between two action classes (the thermometer ordering in ScienceWorld); `no_op_repeat` blocks an immediate repetition when the observation is unchanged; `idempotence_block` rejects an option whose effect has already occurred, including alternating A–B–A–B patterns; `revisit_block` forbids returning to a location visited since an anchoring action; `generic_deref` resolves a generic reference such as "the substance" to the entity implied by the goal; and `relevance_filter` restricts an action class to a candidate set extracted from the goal text.

Extractor slots are likewise drawn from a fixed menu of predefined readers over the goal string, so model errors are limited to invalid primitive–extractor combinations and incorrect parameter values, both detectable during compilation and validation. The same restriction bounds coverage: failure patterns outside the primitive menu cannot be constrained (Appendix C.4).

3.2 EXECUTABLE ACTION GATING

A compiled GEAR specification can be viewed as an execution-time action gate. Let h_t denote the interaction history, $\mathcal{A}(h_t)$ the actions exposed by the environment, and $G_\Sigma(h_t, a) \in \{0, 1\}$ the deterministic decision produced by the compiled specification. The admissible action set is

$$\mathcal{A}_\Sigma(h_t) = \{a \in \mathcal{A}(h_t) : G_\Sigma(h_t, a) = 1\}. \tag{1}$$

Over a finite candidate set this is equivalent to masking actions with $G_\Sigma = 0$ before selection; in free-form action generation, a violating proposal is rejected at the execution interface and the trigger message is returned for replanning.

Unlike gating in neural systems, the gate here is neither latent nor learned: its implementation and blocking semantics are fixed, and what evolves is the explicit specification Σ. MetaAct thus learns *what to gate, not how to gate*. The guarantee is local to the represented constraint: actions rejected by the current guard cannot pass through it, but failures outside the expressivity or evidence of Σ remain possible.

3.3 REPLAY-BASED CONSTRAINT VERIFICATION

A candidate specification passes through two pre-deployment checks and one post-deployment monitor. The first check is a **type contract** applied during compilation: a contract $\mathcal{T}$ maps each primitive π to the extractor families it admits, COMPILE rejects any specification with $e \notin \mathcal{T}(\pi)$, and `null` parameters are replaced by declared defaults. This prevents the two implementation failures observed during development, primitive–extractor incompatibility and parameter-type mismatch.

The second check is the **replay gate**, an empirical verifier in a counterexample-guided inductive synthesis loop (Solar-Lezama et al., 2006): each candidate is compiled and replayed, and a rejected candidate returns concrete counterexamples to the next synthesis round. Unlike classical CEGIS, no formal specification is assumed; previously logged trajectories serve as the evidence.

Let F and S denote the failed and successful trajectories, respectively, recorded for task type τ on the training slice. Replaying a compiled guard g over a trajectory produces a blocking decision for each action. We define failure coverage and the effective-action block rate as

$$\mathrm{cov}(g) = \frac{|\{\, f \in F : g \text{ blocks at least one action of } f \,\}|}{|F|}, \quad \mathrm{eff}(g) = \frac{|\{\, a \in A^+ : g \text{ blocks } a \,\}|}{|A^+|}, \tag{2}$$

where A^+ is the set of *effective* actions—those that change the observation—in the safety reference

$$S \cup \{\, f \in F : \mathrm{score}(f) \geq \sigma_{\min} \,\}.$$

Restricting the denominator of $\mathrm{eff}(g)$ to effective actions prevents no-op actions from being counted as false blocks. Such actions are precisely the behavior that repetition constraints are intended to suppress.

The replay gate accepts a candidate if and only if

$$\underbrace{\mathrm{cov}(g) \geq \tau_{\mathrm{cov}}}_{\text{efficacy}} \;\wedge\; \underbrace{\mathrm{eff}(g) \leq \tau_{\mathrm{eff}} \;\wedge\; \mathrm{ok}(g) \leq \tau_{\mathrm{safe}}}_{\text{safety}}, \tag{3}$$

where $\mathrm{ok}(g)$ denotes the block rate over successful trajectories alone. We set $\tau_{\mathrm{cov}} = 0.3$, $\tau_{\mathrm{eff}} = 0.15$, and $\tau_{\mathrm{safe}} = 0.10$; Appendix A.3 analyzes the contribution of the individual acceptance criteria.

All quantities in Eqs. 2 and 3 are computed by replaying guards over existing logs, so validation needs no additional LLM calls or environment interactions; this is possible because whether a deterministic guard blocks a logged action can be observed directly, whereas the effect of natural-language guidance can only be measured by rerunning the model. A deployment-time **win-rate monitor** additionally retires an accepted constraint whose observed utility falls below a threshold after four uses; only 2 of the 23 ScienceWorld constraints reach four uses, so it is inactive in all reported results.

3.4 REPLAY-GUIDED SEARCH IN SPECIFICATION SPACE

Constraint synthesis is formulated as an iterative search over the specification space, with candidate quality evaluated by the replay gate. At round t, the model proposes

$$\Sigma_t \sim \mathrm{LLM}(\tau, F, S, \mathcal{M}, r_{t-1}), \tag{4}$$

where $\mathcal{M}$ denotes the available primitive and extractor menu, and r_{t-1} is the counterexample report returned by the previous round. The search uses three complementary mechanisms: contrastive evidence, counterexample-driven repair, and constraint union.

Contrastive evidence. The model receives both failed and successful trajectories, allowing it to identify rules that separate ineffective behavior from effective behavior rather than merely describing patterns observed in failures. In an initial WebShop experiment that supplied only failed trajectories, the synthesized candidate blocked both unproductive and successful behavior, mis-blocking 61.4% of successful trajectories. Including successful trajectories provides the evidence needed to distinguish wasteful actions from productive ones.

Counterexample-driven repair. When the replay gate rejects a candidate, it produces a report r_t containing concrete examples of incorrectly blocked effective actions and missed wasted steps. These examples are provided directly to the model in the next synthesis round rather than being reduced to a scalar score; unlike text-based optimizers (Yang et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2026), the signal is generated from the behavior of the compiled artifact on logged trajectories.

Constraint union. A single candidate may capture only part of the observed failure pattern. At each round, we therefore evaluate the incumbent specification, the newly proposed specification, and their union:

$$\Sigma_t^\star = \arg \max_{\Sigma \in \{\Sigma_{t-1}^\star,\, \Sigma_t,\, \Sigma_{t-1}^\star \cup \Sigma_t\}} J(\Sigma), \qquad J = \tfrac{1}{2}\rho + \tfrac{1}{2}\mathrm{cov} - 2\,\mathrm{eff} - 2\,\mathrm{ok}, \tag{5}$$

where ρ is the fraction of wasted steps detected by the compiled guard. Because the union of two specifications is compiled into the conjunction of their guards, its coverage and false-block rates remain directly measurable through offline replay.

Union lets complementary candidates accumulate across rounds: on WebShop it expands a minimal gate-passing candidate into the final constraint set, and repeated accumulation can grow a specification to 17 constraints for `use-thermometer` (Figure 9).

Search terminates when the replay gate accepts the current specification and $\rho \geq 0.8$, or after a maximum of $T = 3$ rounds. Appendix C.4 shows that the third round never produces the first accepted candidate, indicating that successful synthesis typically occurs within the first two rounds.

4 EXPERIMENTS

Our experiments address five questions in turn: (1) Does MetaAct improve performance across embodied-language benchmarks (Section 4.2)? (2) Why do executable constraints outperform textual guidance (Section 4.3)? (3) Can synthesized constraints be reliably verified before deployment (Section 4.4)? (4) Does the benefit generalize across language-model backbones (Section 4.5)? (5) Can executable constraints be acquired and deployed online (Section 4.6)?

4.1 SETUP

Benchmarks. We evaluate four text-based embodied environments: ALFWorld (134 `valid_unseen` episodes of ALFRED household tasks in a TextWorld-style simulator (Shridhar et al., 2021; 2020; Côté et al., 2019)), SciWorld (30 task types with two held-out variants each, 60 episodes (Wang et al., 2022)), TextCraft (60 crafting episodes with success rate and partial-credit score), and WebShop-100k (100 test instances under a 15-step cap with partial-credit score (Yao et al., 2023a)).

Table 1: **Main benchmark comparison.** Performance of published baselines and MetaAct on four embodied-language benchmarks using the DeepSeek-V4-flash backbone. MetaAct-synth denotes automatically synthesized executable constraints and MetaAct-oracle the hand-written reference. Best values are bold; the second-best distinct value in each metric column is underlined.

	ALFWorld (134)	SciWorld (60)		TextCraft (60)		WebShop (100)	
	SR$^{\uparrow}$	SR$^{\uparrow}$	score$^{\uparrow}$	SR$^{\uparrow}$	score$^{\uparrow}$	SR$^{\uparrow}$	score$^{\uparrow}$
Single-agent methods							
Act-only	0.799	0.167	44.3	0.617	80.6	0.180	57.6
ReAct	0.851	0.050	27.5	0.667	72.3	0.120	21.7
SC-ReAct (k=3)	0.851	0.150	45.7	0.850	91.2	0.090	18.0
Reflexion (≤ 3)	**0.955**	0.217	53.5	0.750	82.7	0.210	36.2
AdaPlanner	0.254	0.033	28.6	0.050	16.1	0.090	16.3
ReflAct	0.716	0.133	44.1	0.783	83.9	0.060	9.9
Agent orchestration frameworks							
AutoGen	0.888	0.267	54.6	**0.983**	**98.3**	0.100	17.4
MetaGPT-SOP	0.388	0.067	38.5	0.317	38.4	0.160	41.2
Experience-driven adaptation							
ACE (ICLR'26)	0.925	0.117	34.2	0.667	82.4	0.170	35.3
DC (EACL'26)	<u>0.948</u>	0.183	39.1	0.617	74.5	0.180	43.6
RPMS (2026)	0.940	0.167	46.9	0.750	84.6	**0.290**	61.7
SkillRL-SB[§]	0.888	0.233	52.2	0.833	89.6	**0.290**	59.0
Runtime action guards							
AgentSpec (ICSE'26)	0.336	0.083	37.4	0.233	29.9	0.000	0.0
OntoGuard (ACL'26)	0.299	0.067	34.1	0.783	88.7	0.040	9.3
MetaAct-synth	**0.955**	**0.367**	**62.1**	<u>0.950</u>	<u>97.5</u>	<u>0.270</u>	<u>62.1</u>
MetaAct-oracle	—	<u>0.300</u>	<u>55.3</u>	0.617	76.0	0.260	**63.6**

Baselines. Baselines in Table 1 span four families. Single-agent methods: Act-only, ReAct (Yao et al., 2023c), SC-ReAct with $k = 3$ rollouts (Wei et al., 2023; Wang et al., 2023b), Reflexion (Shinn et al., 2023), AdaPlanner (Sun et al., 2023), and ReflAct (Kim et al., 2025). Orchestration frameworks: AutoGen and MetaGPT-SOP (Wu et al., 2023; Hong et al., 2024). Experience-driven adaptation methods retain, update, or retrieve reusable task experience. Runtime action guards intercept each proposed action: AgentSpec enforces LLM-generated trigger–check–enforce rules (Wang et al., 2026a), and OntoGuard checks actions against ontologies mined from demonstrations (Liu et al., 2026). All methods share the planner prompt, step cap, backend, and temperature; deliberate search over reasoning states (Yao et al., 2023b) is omitted because it multiplies environment interactions per decision under irreversible actions.

Details. The main benchmark comparison uses DeepSeek-V4-flash; cross-backbone experiments additionally evaluate GPT-4.1-mini and Qwen3-8B under the same SciWorld protocol. The replay-gate thresholds ($\tau_{\text{cov}} = 0.3$, $\tau_{\text{eff}} = 0.15$, $\tau_{\text{safe}} = 0.10$) were selected once without tuning on the test sets; Appendix A.3 analyzes the gate criteria and Appendix C.4 the factors that limit synthesis coverage. Our primary statistic is the paired difference from the hand-written reference with uncertainty over seeds; we also report the recovery rate $R = (\text{synth} - \text{w/o})/(\text{oracle} - \text{w/o})$ with its bootstrap interval, and fidelity, the per-action agreement between synthesized and oracle guards.

4.2 Main Benchmark Results

Table 1 compares MetaAct with representative single-agent methods, agent orchestration frameworks, experience-driven adaptation methods, and existing runtime action guards across four embodied-language benchmarks. MetaAct-synth achieves the best SciWorld results (0.367 success rate and 62.1 score), matches the best success rate on ALFWorld (0.955), and remains competitive on TextCraft (0.950 success rate and 97.5 score). On WebShop, it achieves a score of 62.1, outperforming all published baselines in task score while remaining close to the best success rate. Notably, MetaAct also substantially outperforms existing runtime guards such as AgentSpec and OntoGuard,

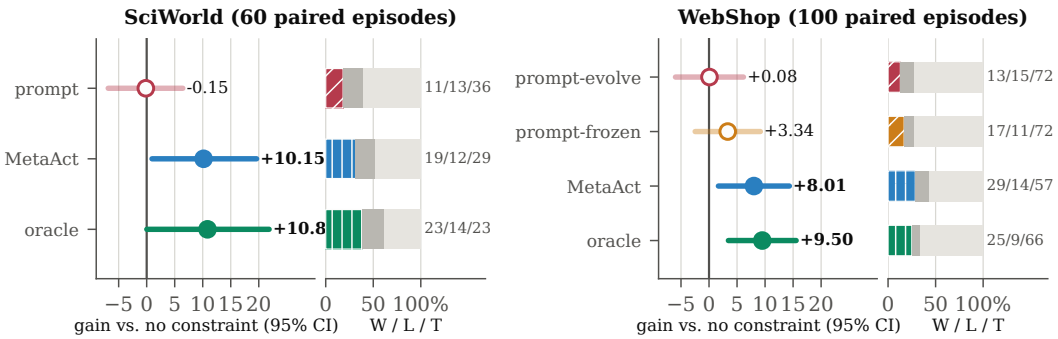


Figure 3: **Matched-evidence comparison.** Each row shows one variant. Left: task-score change relative to the unconstrained baseline, with 95% confidence intervals. Right: episode-level win/loss/tie counts against the same baseline. Filled markers indicate confidence intervals that do not include zero.

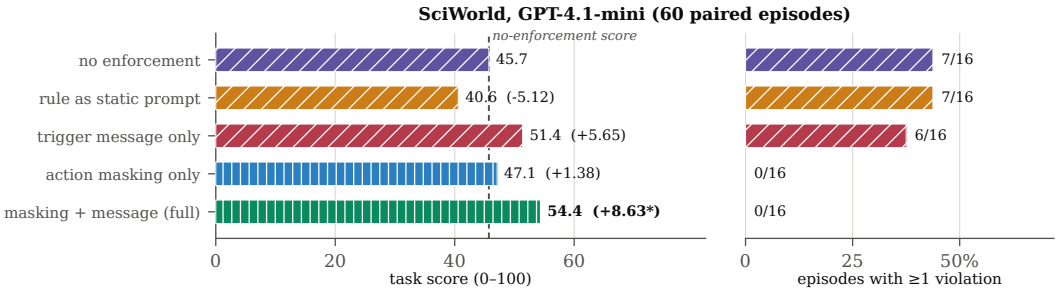


Figure 4: **Effect of action masking and rule messages at execution time.** Results on SciWorld with GPT-4.1-mini over 60 episodes per setting. Left: task score under different enforcement strategies (*: the 95% confidence interval for the score difference is entirely above zero). Right: fraction of episodes with at least one rule violation. Action masking removes violations, while combining masking with rule messages gives the largest score improvement.

indicating that synthesizing and verifying constraints from execution feedback is more effective than applying fixed pre-deployment rules.

4.3 Why Do Executable Constraints Outperform Textual Guidance?

Matched-evidence comparison. Figure 3 compares textual guidance with executable constraints under matched evidence. On SciWorld, textual guidance changes the score by only -0.15, while MetaAct improves it by $+10.15$. On WebShop, MetaAct achieves a $+8.01$ gain, whereas neither prompt-based variant has a confidence interval excluding zero. Across both environments, executable constraints produce substantially larger gains than representing the same rules as text.

The knowing–doing gap. We further audit the completed trajectories using the same synthesized guard to measure whether the target rule is actually followed. Despite having the relevant rule explicitly stated in the prompt, the violation rate remains nearly unchanged: 0.390 versus 0.400 on WebShop, and 0.700 for both the prompt and unconstrained variants on SciWorld. Compiling the same rule into an executable constraint reduces the violation rate to 0.010 on WebShop and to 0 on SciWorld. This exposes the knowing–doing gap: stating a rule in context does not ensure that the agent follows it during action selection.

From rule knowledge to rule compliance. Figure 4 examines how the same rule affects behavior under different forms of enforcement. Providing the rule only as a static prompt leaves violations largely unchanged, while returning a trigger message improves task performance by $+5.65$ but does not fully prevent violations. Action masking, in contrast, reduces violations to zero, and combining

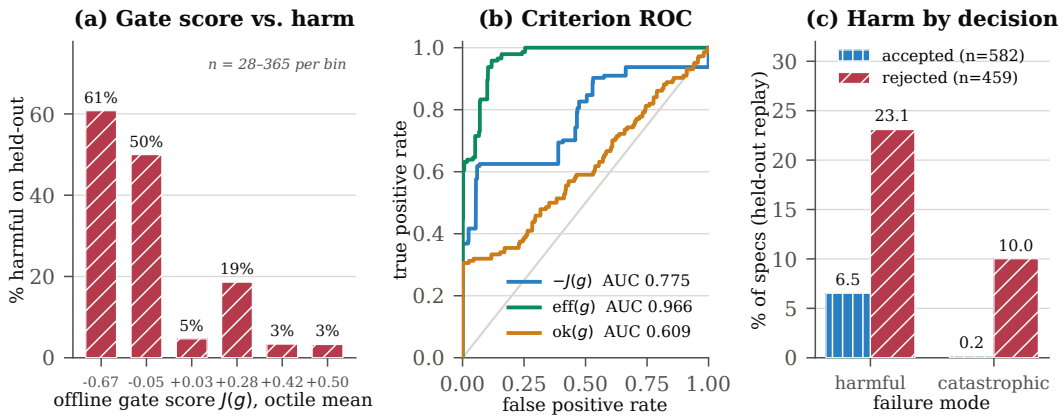


Figure 5: **Offline replay predicts harmful constraint behavior before deployment.** SciWorld candidate specifications. (a) Held-out harmful blocking as a function of the offline gate score. (b) ROC curves for the gate criteria. (c) Harmful and catastrophic blocking rates for accepted and rejected candidates on held-out replay.

Table 2: **Ablation study on SciWorld.** Offline metrics are computed by replay; online metrics are averaged over three evaluation seeds with 60 episodes per seed.

		Offline			Online	
	Variant	Coverage$^{\uparrow}$	Fidelity $F_1{}^{\uparrow}$	Harmful$^{\downarrow}$	Violation$^{\downarrow}$	Score$^{\uparrow}$
A0	**Full MetaAct**	0.211	0.135	0.000	**0.167**	**50.77**
A1	− Contrastive Evidence	0.189	0.411	0.000	0.229	47.62
A2	− Counterexample Repair	0.211	0.164	0.000	0.229	50.66
A3	− Constraint Union	0.189	0.164	0.000	0.292	47.37
A4	− Replay Verification	**0.433**	0.135	0.069	0.229	48.36

masking with the trigger message achieves the largest score gain of $+8.63$. These results highlight the distinction between knowing a rule and enforcing it: textual feedback can guide the planner, whereas executable constraints prevent violating actions from being carried out. Taken together, Figure 3 establishes the effect, the compliance audit locates its cause, and Figure 4 identifies the mechanism: masking guarantees compliance, and the trigger message helps the planner recover a useful action.

4.4 Reliability of Constraint Synthesis and Verification

A synthesized constraint is useful only if a wrong one can be caught before it blocks correct actions. We therefore ask whether replay verification identifies harmful constraints, and whether each component of the synthesis pipeline contributes to the deployed result.

Replay verification. Figure 5 evaluates whether offline replay can identify harmful constraints before deployment. The effective-action block rate $\mathrm{eff}(g)$ is the strongest predictor of held-out harm, with an AUROC of 0.966. Accepted constraints show substantially lower harmful blocking than rejected ones (6.5% vs. 23.1%), while catastrophic blocking drops from 10.0% to 0.2%. These results show that replay verification provides a useful pre-deployment filter for unsafe constraints.

Ablation analysis. Table 2 evaluates the four components used to build and verify executable constraints. The full method achieves the lowest violation rate (0.167). Removing contrastive evidence or constraint union increases violations and lowers task performance, showing that both help produce more effective constraints. Replay verification plays a different role: without it, coverage increases from 0.211 to 0.433, but harmful blocking also appears (0.069), indicating that the additional coverage comes partly from blocking correct actions. Counterexample repair has a smaller effect, but removing it still increases the violation rate from 0.167 to 0.229.

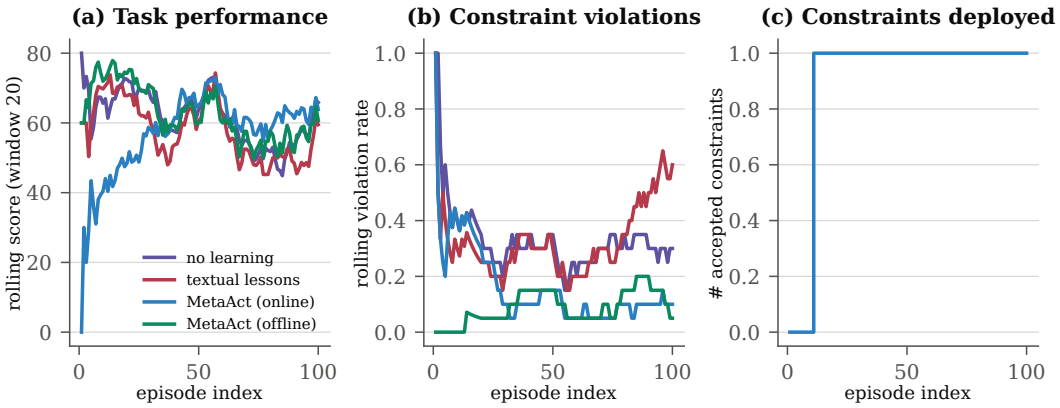


Figure 6: **Executable constraints can be acquired and deployed during an interaction stream.** WebShop with GPT-4.1-mini over 100 consecutive episodes. The panels show 20-episode rolling task score, rolling violation rate, and the cumulative number of accepted constraints.

Table 3: **Executable-constraint gains persist across language-model backbones.** SciWorld task score (60 paired episodes per backbone) with no added constraint, the textual prompt rule, MetaAct, and the hand-written reference.

Backbone	No constr.	Prompt rule	MetaAct	Oracle	Δ vs. prompt [95% CI]
DeepSeek-V4-flash	44.45	44.30	**54.60**	55.30	$+10.30\ [+1.15, +19.83]^*$
GPT-4.1-mini	45.72	40.60	**54.35**	50.83	$+13.75\ [+6.43, +21.92]^*$
Qwen3-8B	34.77	30.90	**36.73**	32.97	$+5.83\ [-0.77, +12.93]$

4.5 Cross-Backbone Generalization

Table 3 evaluates whether the benefit of executable constraints generalizes across language-model backbones. Under the same SciWorld protocol, we compare DeepSeek-V4-flash, GPT-4.1-mini, and Qwen3-8B with no added rule, a textual prompt rule, MetaAct, and a hand-written reference. MetaAct outperforms the prompt rule on all three backbones, with confidence intervals excluding zero for DeepSeek-V4-flash and GPT-4.1-mini. The gain varies across models, but the advantage of executable enforcement is not specific to the DeepSeek backbone.

4.6 Online Self-Evolution of Executable Constraints

Figure 6 evaluates whether MetaAct can acquire executable constraints during deployment. Starting with no constraint, the agent accumulates failure evidence, synthesizes and verifies rules, and deploys accepted constraints as the interaction stream proceeds. Online MetaAct reduces the episode violation rate to 0.130, compared with 0.310 for no learning and 0.350 for textual lesson evolution. Its rolling score also rises from 48.40 in the first 20 episodes to 63.83 in the last 20, showing that executable constraints can be learned and put into use during ongoing interaction. MetaAct therefore not only uses executable constraints; it acquires and deploys them during ongoing interaction.

5 Conclusion

We presented MetaAct, a framework that extends language-agent self-improvement from textual guidance to executable action constraints. MetaAct synthesizes typed GEAR rules from execution feedback, compiles them into deterministic guards, and validates them through offline trajectory replay before deployment. Across four embodied-language benchmarks, MetaAct consistently improves task performance; on SciWorld and WebShop, executable constraints reduce target-rule violations from 70% to 0% and from 40% to 1%, while improving task scores by $+10.1$ and $+8.0$, respectively. These results show that self-evolution can extend beyond textual adaptation: execution feedback can be converted into reusable constraints that directly shape future action selection.

AI USE STATEMENT

In this work, we did not use generative AI tools for any of the tasks that require disclosure: generating synthetic datasets, developing theoretical models or conceptual frameworks, formulating mathematical claims or proofs, proposing or refining hypotheses, designing or evaluating the experimental methodology, implementing methods, translation, cleaning or reformatting data, qualitative or thematic analysis, or interpreting results. Generative AI tools were used for two purposes only. First, to aid and polish the writing: the text was written by the authors, and AI tools were used to improve its readability, grammar and phrasing; all technical content, claims and numbers were written and verified by the authors. Second, to produce the decorative artwork (characters and scene elements) in the illustrations of Figures 1 and 2; the layout, the text and the results these illustrations depict were designed and checked by the authors, and no data figure or table in the paper was generated by AI. Neither use affects the methods, experiments or conclusions of the paper. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Mohammed Alshiekh, Roderick Bloem, Ruediger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding, 2017. URL https://arxiv.org/abs/1708.08611.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning, 2024. URL https://arxiv.org/abs/2405.16247.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games, 2019. URL https://arxiv.org/abs/1806.11532.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 835–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(71)90010-5. URL https://www.sciencedirect.com/science/article/pii/0004370271900105.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=zDbsSscmuj.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022. ISSN 2334-0762. doi: 10.32473/flairs.v35i.130584. URL http://dx.doi.org/10.32473/flairs.v35i.130584.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Tomas Jackson, Noah Brown, Linda Luu, Sergey Levine, Karol Hausman, and brian ichter. Inner monologue: Embodied reasoning through planning with language models. In Karen Liu, Dana Kulic, and Jeff Ichnowski (eds.), *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 1769–1782. PMLR, 14–18 Dec 2023. URL https://proceedings.mlr.press/v205/huang23c.html.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

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/.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023. URL https://arxiv.org/abs/2209.07753.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+p: Empowering large language models with optimal planning proficiency, 2023. URL https://arxiv.org/abs/2304.11477.

Pengxiang Liu, Tao Ren, Wei Xiong, Tingrui Yang, Junjie Wang, and Jun HU. OntoGuard: Enforcing action admissibility for LLM agents in complex interactive environments. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20937–20952, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1051. URL https://aclanthology.org/2026.findings-acl.1051/.

Argaman Mordoch, Enrico Scala, Roni Stern, and Brendan Juba. Safe learning of pddl domains with conditional effects – extended version, 2024. URL https://arxiv.org/abs/2403.15251.

Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-generated in-context examples improve LLM agents for sequential decision-making tasks. In *Second Workshop on Test-Time Adaptation: Putting Updates to the Test! at ICML 2025*, 2025. URL https://openreview.net/forum?id=YurjMGGTTj.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks, 2020. URL https://arxiv.org/abs/1912.01734.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning, 2021. URL https://arxiv.org/abs/2010.03768.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models, 2022. URL https://arxiv.org/abs/2209.11302.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News*, 34(5):404–415, October 2006. ISSN 0163-5964. doi: 10.1145/1168919.1168907. URL https://doi.org/10.1145/1168919.1168907.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models, 2023. URL https://arxiv.org/abs/2305.16653.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023a. URL https://arxiv.org/abs/2305.16291.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 2938–2950. ACM, April 2026a. doi: 10.1145/3744916.3764546. URL http://dx.doi.org/10.1145/3744916.3764546.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023b. URL https://arxiv.org/abs/2203.11171.

Yimeng Wang, Jiaxing Zhao, Hongbin Xie, Hexing Ma, Yuzhen Lei, Shuangxue Liu, Xuan Song, Zichen Zhang, and Haoran Zhang. Metagen: Self-evolving roles and topologies for multi-agent llm reasoning, 2026b. URL https://arxiv.org/abs/2601.19290.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2):107–143, 2007. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2006.11.005. URL https://www.sciencedirect.com/science/article/pii/S0004370206001408.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023a. URL https://arxiv.org/abs/2207.01206.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023b. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023c. URL https://arxiv.org/abs/2210.03629.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.181. URL https://aclanthology.org/2024.findings-acl.181/.

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.

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

A Additional Experiments

A.1 State-Grounded Reasoning Baseline

ReflAct (Kim et al., 2025) tests whether more strongly grounded textual reasoning can address the same failure mode. Its backbone replaces free-form planning with continuous reflection on the agent's state relative to the goal. We adopt the backbone and hold the instruction, few-shot examples, environment interface, step cap, backend, and temperature fixed relative to our ReAct baseline.

Table 4: ReflAct compared with ReAct under matched settings.

SciWorld test60	SR	score	steps	tokens/ep
ReAct	0.050	27.5	24.5	43.9k
ReflAct	0.050	**31.9**	**17.9**	48.7k
AutoGen (strongest baseline)	0.267	54.6	15.0	27.7k
constraint variant (ours)	**0.300**	**55.3**	33.2	139.9k

Relative to ReAct, ReflAct improves partial-credit score by 4.4 points and reaches that score with 27% fewer environment steps. The success rate remains $3/60$ in both conditions, and the ReflAct score remains approximately 23 points below the constrained variant. These results suggest that better state-grounded reasoning can improve progress without preventing the irreversible focus error targeted by the action constraint.

ReflAct's published gains use ALFWorld and a different backend, so this comparison should be interpreted only within the present SciWorld configuration.

A.2 Robustness to Goal Rewriting

We replay the same compiled guards while rewriting only the natural-language goal. Four transformations preserve the task semantics; a fifth deliberately changes the meaning.

The four meaning-preserving rewrites leave every blocking decision unchanged, whereas changing the goal semantics produces 22 additional blocks. The nontrivial response comes from goal-dependent primitives such as `relevance_filter`; repetition guards that do not read the goal are invariant by construction. The result therefore shows stability to surface form while preserving sensitivity to the task semantics used by the guard.

Table 5: **Guard stability under goal rewriting.** SciWorld: 39 episodes, 715 actions, and 63 guard firings under the original goals. The same compiled guards are replayed after each rewrite.

Goal perturbation	Consistency$^{\uparrow}$	Jaccard$^{\uparrow}$	Δ blocks
paraphrase	1.0000	1.000	+0
clause reordering	1.0000	1.000	+0
synonym substitution	1.0000	1.000	+0
distractor clause	1.0000	1.000	+0
meaning-changing control	0.9692	0.774	+22

Table 6: **Contribution of the replay-gate criteria on SciWorld.** The candidate pool is fixed; only the acceptance rule changes.

Gate variant	Pass rate	Coverage	False block$^{\downarrow}$	Harmful$^{\downarrow}$	Catastrophic$^{\downarrow}$
Full gate	0.643	16/23	0.0248	0.188	0.000
− `cov`	0.829	20/23	0.0198	0.150	0.000
− `eff`	0.700	19/23	0.0208	0.316	0.000
− `ok`	0.643	16/23	0.0248	0.188	0.000
− `eff` & `ok`	0.814	20/23	0.0231	0.300	0.000
no gate	1.000	23/23	0.0843	0.261	0.087

A.3 Ablation of Replay-Gate Criteria

To identify which replay statistics carry the selection signal, we re-evaluate the same candidate pool while changing only the acceptance criterion.

The effective-action criterion `eff` carries the strongest safety signal: removing it raises the harmful rate from 0.188 to 0.316. Removing `ok` leaves the result unchanged in this SciWorld pool because successful reference trajectories are sparse. The coverage threshold is conservative in this setting: removing it admits four additional task types without increasing harm. The no-gate condition is the only setting that admits catastrophic candidates, with rate 0.087.

A.4 Replay-Gate Robustness to Corrupted Evidence

We corrupt the validation evidence while keeping the candidate constraints fixed, then evaluate the accepted set on clean held-out trajectories.

With 30% label noise, acceptance falls from 0.643 to 0.411, while the harmful rate among accepted candidates changes from 0.089 to 0.116. At 30% trajectory truncation, acceptance is 0.543 and the harmful rate is 0.078. The dominant response to corrupted validation data is thus increased selectivity rather than a large increase in held-out harmful blocking.

A.5 Scaling Behavior under Constraint Accumulation

We stress-test conjunctions of increasingly many verified constraints to separate compiler-level composability from behavioral over-restriction.

Compilation succeeds in every tested setting, including unions of roughly 33 rules. The action block rate rises from 0.3465 for the self-only specification to 0.4461 for large compatible unions and 0.5120 when scopes overlap. The corresponding fraction of heavily blocked episodes rises to 0.367 in the overlapping condition. The scaling limit observed here is therefore behavioral scope overlap rather than syntactic or type-level incompatibility.

A.6 Compliance under Localized Rule Feedback

When a guard fires, its message is written to `last_error` and inserted verbatim into the actor and planner prompts on the next turn. The agent then receives two retries. This condition minimizes con-

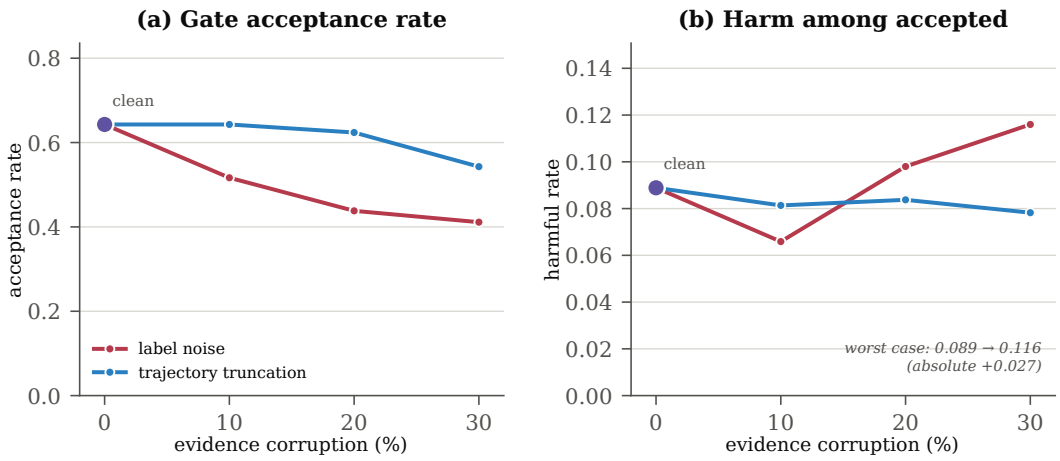


Figure 7: **Replay validation remains selective under corrupted evidence.** SciWorld candidate constraints under label noise and trajectory truncation. (a) Gate acceptance rate. (b) Harmful rate among the accepted constraints, evaluated on clean held-out trajectories.

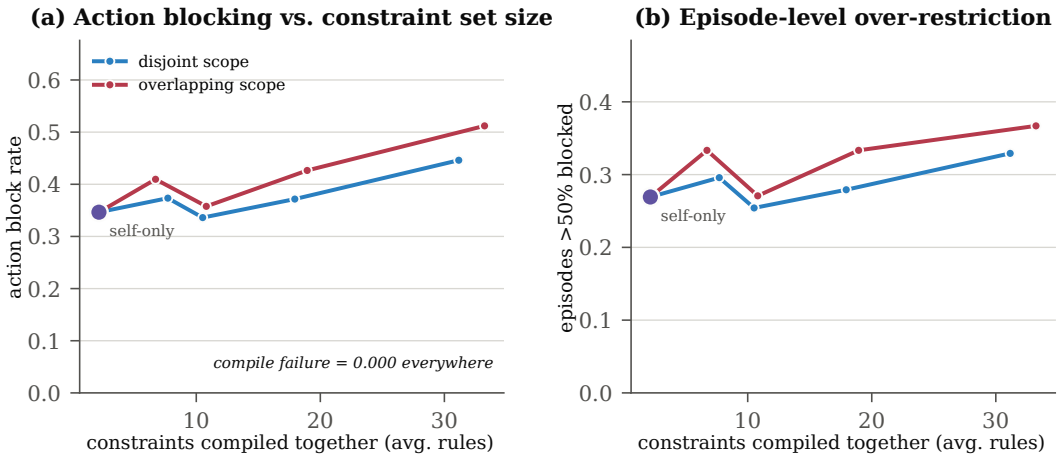


Figure 8: **Constraint accumulation preserves compiler composability while overlapping scopes increase behavioral restriction.** SciWorld. (a) Action block rate as the number of compiled rules grows. (b) Fraction of episodes in which more than half of the actions are blocked.

text distance and provides the exact rule at the time of violation, so the fraction of guard activations resolved within the retry budget provides a direct measure of compliance.

Compliance is 35.1% for synthesized SciWorld guards and 33.1% for hand-written SciWorld guards, compared with 88.1% and 85.3% on WebShop. The SciWorld result directly exposes the knowing–doing gap under a maximally local intervention: even when the exact rule is supplied at the moment of violation, most attempts are not corrected within the retry budget. This motivates enforcement at the action interface rather than relying on textual compliance alone.

B Main-Result and Efficiency Details

B.1 Three-Seed Comparison with the Hand-Written Reference

The same evaluation instances are repeated across seeds, so the seed is the unit of replication. The per-seed differences (synth − oracle) are $+6.8/-1.4/-3.4$ on SciWorld and $-1.5/-1.0/-0.4$ on WebShop, yielding mean differences of $+0.67$ and -0.97, respectively. On WebShop the lower

Table 7: Compliance when the exact rule is supplied at the time of violation with two retries.

Variant	complied within budget	budget exhausted	compliance
SciWorld, synthesized	60	111	**35.1%**
SciWorld, hand-written	116	234	**33.1%**
WebShop, synthesized	52	7	88.1%
WebShop, hand-written	64	11	85.3%

Table 8: **Three-seed comparison with the hand-written reference.** Results use seeds 42/43/44 and are paired by episode ID. $R = (\text{synth} - \text{w/o})/(\text{oracle} - \text{w/o})$ measures recovery of the hand-written headroom.

	w/o	synth$^{\uparrow}$	oracle	$R^{\uparrow}$(95% CI)	Tokens/ep$^{\downarrow}$
SciWorld test60	41.8 ± 3.5	55.3 ± 5.9	54.6 ± 0.9	0.95 $[0.26, 3.78]$	101.3k / 137.6k (-26%)
WebShop test100	55.8 ± 1.5	60.4 ± 1.5	61.4 ± 2.0	0.78 $[-2.83, 4.75]$	19.4k / 21.0k (-8%)

confidence bound remains within the predefined three-point non-inferiority margin. SciWorld has a wider seed-level interval, so its aggregate similarity to the hand-written reference is less precisely estimated.

B.2 Performance Stratified by Synthesis Decision

Table 9: **SciWorld performance stratified by whether synthesis emits a verified constraint.** Strata are determined from training evidence; intervals are computed over the three evaluation seeds.

Stratum	types	per seed	synth $-$ oracle$^{\uparrow}$	95% CI (seeds)
Constraint emitted	10	$-0.8/-5.8/-9.2$	-5.27	$[-15.8, +5.2]$
Abstained	20	$+10.6/+0.8/-0.4$	$+3.67$	$[-11.3, +18.7]$
Pooled	30	$+6.8/-1.4/-3.4$	$+0.67$	$[-12.8, +14.1]$

The synthesizer emits constraints for 10 of 30 task types and abstains on the remaining 20. On the emitted types, the hand-written reference has the higher point estimate; on the abstained types, the unconstrained synthesized pipeline has the higher point estimate. The pooled result therefore combines partial constraint quality with selective abstention rather than assuming that a rule must be emitted for every task type.

B.3 Per-Episode Token Use in the Main Benchmark

Token use reflects the full interaction loop rather than the cost of evaluating a guard. Methods with few model calls can be inexpensive but weak, while constraint-triggered replanning can increase token use even when the compiled guard itself is deterministic and token-free.

B.4 Computational Cost

Constraint synthesis is a one-time cost per task type, and evaluating a compiled guard itself requires no LLM call. Total episode-level token use can nevertheless change because blocking an action may alter the trajectory and trigger additional planning or retries. On SciWorld, synthesis uses 487k tokens across 23 task types (21k per type); on WebShop it uses 33k tokens for the single task type. The per-episode token totals for the main baseline comparison are reported in Appendix B.3. Because blocking can trigger corrective replanning or retries, fewer environment actions do not necessarily imply fewer model tokens; we therefore report token use separately from task-level performance.

Table 10: **Per-episode model-token use for the methods in Table 1.**

Method	ALFWorld	SciWorld	TextCraft	WebShop
Act-only	32.9k	22.2k	31.9k	11.2k
ReAct	40.2k	43.9k	49.8k	32.6k
SC-ReAct (k=3)	116.0k	113.7k	106.9k	89.6k
Reflexion (≤ 3)	58.3k	138.4k	91.5k	97.3k
AdaPlanner	4.8k	4.1k	4.1k	39.8k
AutoGen	34.2k	27.7k	26.7k	41.4k
MetaGPT-SOP	19.3k	15.2k	21.1k	39.0k
ReflAct	79.0k	44.2k	49.1k	66.2k
ACE (ICLR'26)	44.6k	34.9k	39.4k	35.5k
DC (EACL'26)	39.2k	26.0k	47.8k	34.1k
RPMS (2026)	45.4k	62.7k	48.3k	16.7k
SkillRL-SB[§]	44.6k	42.2k	47.4k	21.4k
AgentSpec (ICSE'26)	34.0k	63.1k	5.4k	28.2k
OntoGuard (ACL'26)	66.2k	81.8k	50.6k	42.4k
MetaAct-synth	21.3k	84.7k	113.6k	18.6k
MetaAct-oracle	—	139.9k	80.8k	22.8k

C Constraint Synthesis and Guard Analysis

C.1 Behavioral Stability Across Synthesis Runs

We distinguish textual reproducibility of the emitted specification from behavioral reproducibility after compilation. Across three independent synthesis runs with identical evidence and settings, only 2 of 20 shared task types receive the same primitive set in all three runs, and 12 of 20 receive the same replay-gate verdict.

Despite this textual variation, the compiled specifications agree on 86.1% of per-action decisions ($949/1102$) when replayed over the same 60 held-out trajectories. The replay gate evaluates candidate behavior rather than exact textual form, so different specifications may be accepted when they implement similar blocking decisions.

Most residual disagreement is concentrated in `lifespan-longest-lived` (49.2% agreement), `inclined-plane-determine-angle` (59.0%), and `mendelian-genetics-unknown-plant` (62.5%). These are also task types for which candidates frequently block no recorded failure action and are rejected by the gate (Appendix C.4). Because synthesis is performed once per task type, a deployed system can cache the accepted specification; run-to-run variation affects reproduction of the synthesis procedure rather than execution of an already verified artifact.

C.2 Agreement with Hand-Written Reference Guards

On WebShop, the synthesized and hand-written guards agree on all 1307 held-out actions, yielding $P = R = F_1 = 1.000$ with no false positives or false negatives. On SciWorld, under the focus-isolated evaluation, precision is 1.000 and recall is 0.812. Across three independent synthesis runs, precision remains $1.000/1.000/1.000$, with constraints covering an average of 12.3 task types (range 11–13). The synthesized SciWorld guards are therefore conservative in this evaluation: observed disagreement arises from missed blocks rather than false blocks.

C.3 Cross-Environment Primitive Utilization

The same six-primitive menu is available in every environment, while the selected specifications are determined separately for each task type.

The repetition-related primitives `no_op_repeat` and `idempotence_block` are selected in both SciWorld and WebShop. The remaining primitives are selected only in environments containing the corresponding ordering, reference-resolution, or navigation failures. No specification is emitted on ALFWorld, consistent with its lack of failure evidence under the saturated backbone.

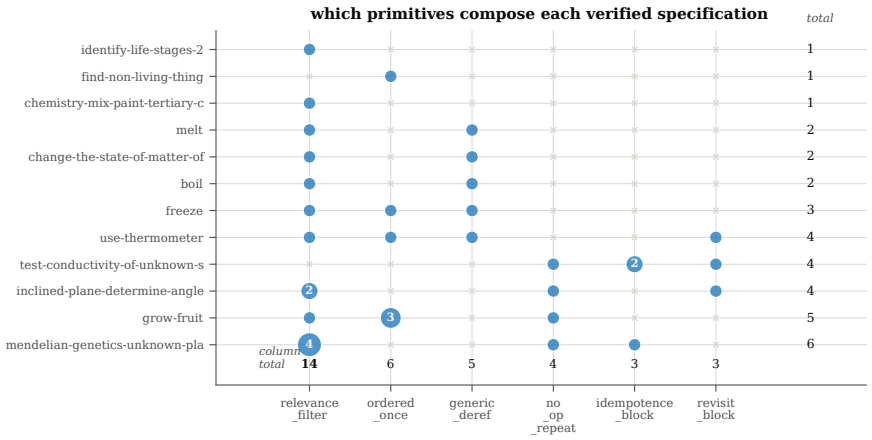


Figure 9: Composition of verified specifications. Marker area indicates the number of constraints using each primitive; $\times$ indicates that a primitive is absent. Column totals report aggregate use.

The replay gate also removes cross-environment misapplications. Under the strict gate, SciWorld selections of `idempotence_block` and `revisit_block` each decrease from 3 to 0, while ordering and reference-resolution primitives are retained or become more frequent. Thus, primitive selection is driven by both the task-level proposal and empirical validation on the environment's trajectories.

This comparison includes two environments that emit specifications, and WebShop contributes only one task type; the usage pattern should therefore be interpreted as descriptive rather than as a broad cross-environment estimate.

C.4 Analysis of Synthesis Coverage

We evaluate three potential explanations for partial coverage: primitive expressiveness, replay-gate strictness, and the amount of failure evidence.

First, adding a primitive targeted at a common rejected diagnosis does not increase aggregate coverage. Seven of ten gate-rejected candidates describe the agent leaving the task-relevant location, a pattern not expressible by the original five-primitive menu. We therefore add `revisit_block`, which keeps the agent at the location associated with the anchoring action, and rerun synthesis with the primitive menu as the only change. The new primitive is selected for 2–3 task types per run and corrects specific failures, but the six-primitive menu verifies $8/9/9$ of the 16 shared types, compared with $8/9/10$ for the five-primitive menu. The mean difference is -0.33 types relative to a within-condition standard deviation of 0.82.

Second, relaxing the replay gate provides limited additional coverage. Re-evaluating all 112 specifications generated across five synthesis runs shows that 71% of rejections occur because the candidate blocks no action in the failure trajectories. Reducing τ_{cov} from 0.3 to 0 therefore admits no additional task type. Removing both safety criteria increases the specification-level pass rate from 50.9% to 65.2%, but does not resolve the dominant zero-block failure.

Third, the amount of recorded evidence does not separate firing and zero-block candidates. Both groups contain a median of two failure episodes, with means of 1.82 and 1.85, respectively. The zero-block group also contains slightly more blockable wasted steps (2.7 versus 2.2).

Together, these analyses do not identify primitive vocabulary, gate thresholds, or evidence count as the primary determinant of coverage in the tested runs. The remaining difficulty is the mapping from a natural-language diagnosis to a specification that fires on the recorded failure. For example, `chemistry-mix` and `measure-melting-point-unknown-substance` produce zero-block candidates in all five runs. Improving diagnosis-to-specification grounding is therefore a direct target for future work.

D Evaluation Protocol and Metric Diagnostics

D.1 Dynamics of Online Lesson Memory

Table 11 summarizes how the lesson pool changes during evaluation.

Table 11: Online lesson evolution under the evaluation protocols.

Benchmark	Variant	Episodes	Online additions/ep	Median `uses`
SciWorld	w/o	60	0.78	1
	prompt (seeded)	60	0.18	0
WebShop	w/o	100	0.73	4
	prompt-evolve	100	0.76	6
	prompt-frozen	100	0.00	100

On WebShop, the retirement rule retains a lesson only when its win rate is at least 0.5 after three uses, while the base success rate is 0.22. The lesson pool consequently turns over at 0.76 additions per episode, producing a score change of $+0.08$ with 6% more tokens. Freezing the same initial lessons yields $+3.3$, indicating that the retirement rule removes potentially useful lessons together with unhelpful ones.

On SciWorld, each task type has only two test episodes, fewer than the three uses required for retirement. The seeded pool is already at capacity, so online additions are also limited (0.18 per episode, with median `uses` equal to 0). In this setting, the maintenance mechanism is largely inactive. Accepted executable constraints do not require the same online retirement loop because they are validated before deployment and then held fixed.

D.2 Protocol Controls for Retries and Lesson Retirement

The unguarded variants also record retries, but those retries result from parsing and grounding failures rather than action-constraint violations. Their rates (88.6% for SciWorld w/o and 79.5% for SciWorld prompt) therefore measure a different event and are not directly comparable with Table 7.

The lesson-retirement rule keeps a lesson while `uses` $<$ `3` or when `win-rate` ≥ 0.5. On SciWorld, each task type has only two test episodes, so retirement never occurs and seeded lessons persist. On WebShop, there is one task type and the success rate is 0.22, causing each lesson to be removed after reaching three uses unless it happens to exceed the 0.5 threshold. We observe 76 redistillations across 100 episodes. The `--freeze_lessons` condition aligns WebShop with SciWorld's effective protocol; all three initial lessons remain available for all 100 episodes (`uses`=100, with no online additions).

D.3 Violation Timing and Metric Selection

Primary metric. Per-action violation rates are affected by survivorship in SciWorld. The unconstrained variant often terminates within one or two steps after a focus error, producing 220 actions across 20 episodes, whereas constrained variants continue for 625 actions and therefore have more opportunities to act. We use the episode-level violation rate as the primary SciWorld metric. WebShop has a fixed 15-step cap and comparable action counts across variants (579–744), so episode- and action-level metrics are both appropriate and lead to the same conclusion.

Evaluator scope. Applying the unrestricted SciWorld specification reports 78 violations in the constrained variant, but these activations arise from `no_op_repeat` disagreements on repeated `look around` actions. That rule family remains unchanged across the conditions and is not the component under evaluation. Restricting the evaluator to the focus-family primitives gives the relevant count of zero. This example motivates evaluating a modified sub-module only over the rule family it replaces.

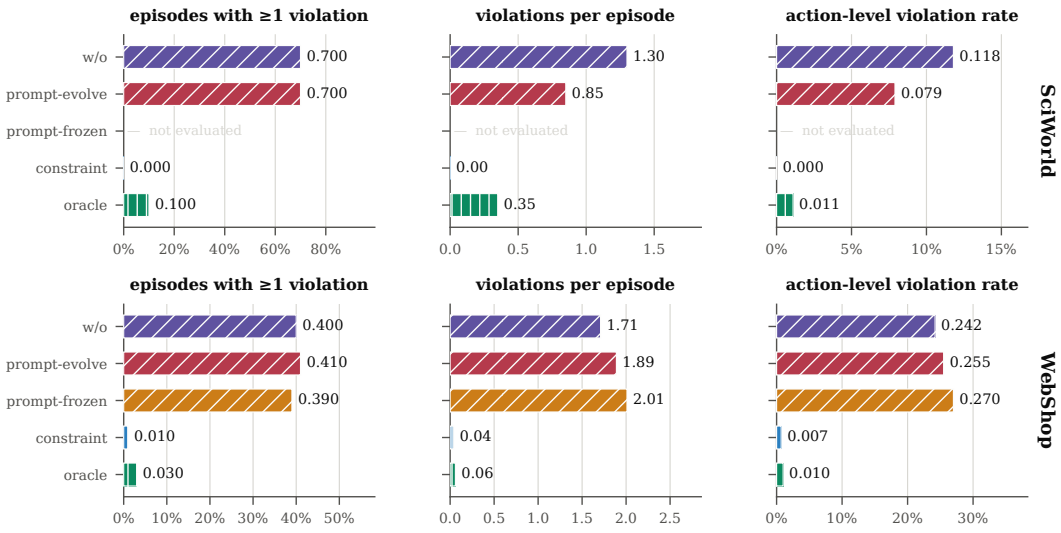


Figure 10: Episode-level rule-violation rates. Diagonal hatching denotes rules provided in the prompt, and grid hatching denotes compiled constraints.

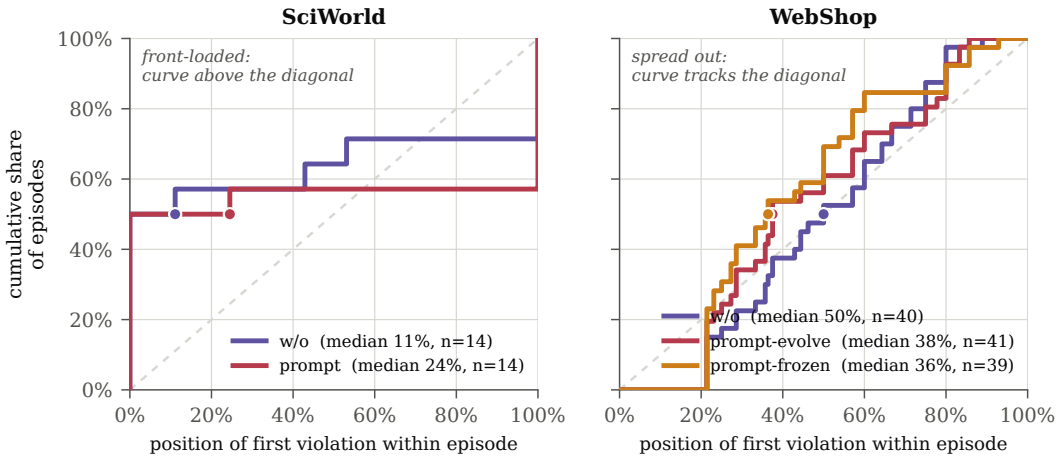


Figure 11: Cumulative distribution of the first violation's relative position within an episode. Curves above the diagonal indicate violations concentrated earlier in the episode.

D.4 Scope-Aware Evaluation of Composite Agents

Metrics aggregated over all actions can include behavior controlled by agent components that are unchanged across the compared conditions. We observe this issue in three forms.

Dominance by an unchanged primitive. In the SciWorld fidelity calculation, 85 of 98 true positives come from `no_op_repeat`, which is active in both variants. An aggregate F_1 score therefore mostly reflects agreement on a rule family that is not under evaluation.

Scope spillover. False positives can occur on non-focus actions outside the scope of the focus guard. Under the aggregate metric, increasing synthesis evidence appears to reduce fidelity ($F_1 = 0.222$); under focus-isolated precision, the ordering reverses (0.778 versus 0.625).

Violation-rate contamination. Using the unrestricted specification to evaluate Section 4.3 produces 78 apparent violations in the constrained SciWorld variant. All are `no_op_repeat` disagreements on repeated `look around` actions, while the focus-family count is zero.

These cases show that action-level evaluation of a composite agent should be restricted to the scope of the sub-module being changed. Error types should also be interpreted according to their costs, because a false block and a missed block can have substantially different consequences in an irreversible action space.

D.5 Selection Effects in Win-Rate-Based Validation

The surviving lessons in the prompt-evolve variant can have high observed win rates (e.g., `uses`=6 and `wins`=4), but the retention rule itself keeps only lessons whose win rate exceeds 50%. High win rates among retained lessons are therefore partly a consequence of conditioning on survival. With a base success rate of 0.22, some retained lessons may reflect favorable short sequences rather than reliable improvements. Evaluations of win-rate-gated memories should therefore account for this selection effect.