MASOPT: ONLINE PROMPT OPTIMIZATION FOR MULTI-AGENT SYSTEMS WITH CLOSED-LOOP CONTROL

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based multi-agent systems (MAS) decompose a complex task across specialized agents, where the prompt of each agent governs both its own behavior and the information it passes downstream. Existing MAS prompt optimizers search for a configuration before deployment and keep it fixed, which is an open-loop controller that never measures what its own output produces. In this paper, we introduce MASOpt, an online prompt optimization framework that closes this loop with the feedback principle of classical control theory. MASOpt treats the target MAS as a controlled plant and every local prompt edit as a control input whose value is determined only after the plant responds. A trajectory observer estimates a latent failure state from the partial execution, a minimal actuation module turns that state into a local prompt repair, and a feedback gate admits the repair only when the measured error decreases, so the retained prompt state never increases that error. To decide where to actuate instead of where the error becomes visible, we further propose Cross-Actuation Relative Advantage (CARA), which replays candidate actuations from an identical prefix and scores each against the others issued from the same state. Experiments on eight benchmarks show that MASOpt improves the average score from 80.58 to 83.40 over the strongest offline MAS prompt optimizer and raises the average interactive success rate from 39.23 to 64.85, with the backbone LLMs and the workflow topology left unchanged.[1]

1 INTRODUCTION

Large language model (LLM)-based multi-agent systems (MAS) have become a promising paradigm for tasks that exceed the reach of a single model call (Wu et al., 2024; Li et al., 2023; Chen et al., 2024; Zhou et al., 2026). By assigning planning, reasoning, execution, and verification to specialized agents (Li et al., 2023; Chen et al., 2024), a MAS decomposes a hard problem into coordinated sub-processes and aggregates the strengths of many LLM invocations (Wu et al., 2024; Zhang et al., 2025). Within such a system the prompt of each agent specifies its behavior, since it determines how the agent reads its role, consumes upstream messages, and hands information downstream (Khattab et al., 2024; Yuksekgonul et al., 2025). Optimizing these prompts has therefore become the dominant mechanism for improving a MAS whose backbone models and topology are fixed (Zhang et al., 2026b; Hong et al., 2026; Wang et al., 2026). However, every prompt such an optimizer returns is committed before the task it must solve has started.

Unlike a single-agent prompt, which affects one output, a MAS prompt propagates along the topology, so an edit to an upstream agent changes the input distribution of every downstream agent it feeds (Zhang et al., 2026b; Wang et al., 2026). Prompt quality is therefore a property of the execution state, not of the prompt. An instruction that steers a solver correctly at the start of a derivation turns harmful once the solver emits a malformed expression, because the downstream verifier inherits it and spends the remaining budget confirming it (Hendrycks et al., 2021). The effect is considerably sharper in long-horizon interaction, where one misread dependency propagates through the remaining steps of a 20-step TextCraft episode and a configuration fixed beforehand cannot intervene at the

[1] Code are available in supplementary materials.

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.

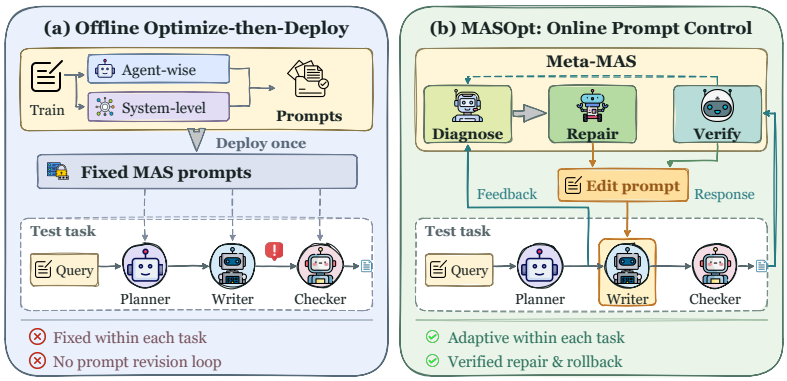


Figure 1: Offline prompt optimization is an open-loop controller that commits a configuration before execution, while MASOpt closes the loop inside the episode and keeps a local prompt repair only when the measured error decreases.

third step even though the failure is already visible (Prasad et al., 2024). In both cases the evidence needed to fix the prompt arrives while the prompt is already in use, and an optimizer that has finished its work cannot read it.

Existing MAS prompt optimization can be broadly categorized into agent-wise and system-level optimization. Early efforts optimize each agent independently with a generic prompt optimizer (Zhou et al., 2023; Pryzant et al., 2023; Yang et al., 2024; Fernando et al., 2024; Lin et al., 2024), treating the remaining agents as a static environment. However, these methods ignore the dependencies the topology induces, so a locally improved prompt can still degrade the workflow containing it (Zhang et al., 2026b). More recently, a line of work models the interaction among prompts explicitly through topology-aware feedback, budget-constrained black-box search, or downstream-utility scoring (Zhang et al., 2026b; Hong et al., 2026; Wang et al., 2026). Although these systems capture the coupling far better, each still returns one configuration frozen at deployment (Zhang et al., 2025; Xia et al., 2026). In essence, both categories are open-loop, as they compute a control sequence in advance and apply it without measuring what it produces.

In this paper, we aim to advance MAS prompt optimization from open loop to closed loop. Classical control theory establishes that an open-loop controller reaches the intended behavior only when the plant model is exact and no disturbance enters the loop, and that feedback is what lets a controller tolerate both (Åström & Murray, 2021; Goodwin et al., 2001). An LLM-based MAS satisfies neither condition. Its response to a prompt is unpredictable, and every task instance injects a different disturbance. Along with this idea, we propose *MASOpt*, which treats the target MAS as the controlled plant, the observable trajectory as the measurement, and each local prompt edit as a control input. Figure 1 contrasts the two regimes. Specifically, to recover the failure a measurement reveals only partially, MASOpt introduces a *trajectory observer* that estimates a latent failure state. To act on that state without disturbing the workflow, MASOpt restricts the control input to a *minimal actuation* on one agent or one interface, keeping the perturbation small enough for its effect to be attributed. To stop an unverified edit from corrupting the episode, a *feedback gate* admits an actuation only when the measured error decreases and reverts it otherwise. This revert branch is what separates a controller from iterative self-refinement, since Self-Refine rewrites an output and Reflexion accumulates a memory for the next trial, while neither can withdraw a change that made the system worse (Madaan et al., 2023; Shinn et al., 2023).

Closing the loop still leaves open where to actuate. In a MAS the agent at which an error becomes visible is frequently not the agent at which actuation is most effective, since the error is produced upstream and only surfaces downstream (Zhang et al., 2026b). To resolve this, we propose *Cross-Actuation Relative Advantage* (CARA), which replays every candidate actuation from an identical saved prefix and scores it against the others issued from the same state. Fixing the prefix removes the confound of a different preceding trajectory, so the comparison isolates the actuation itself, in the same way that a step response is interpretable only from a known initial condition (Åström & Murray, 2021; Goodwin et al., 2001). The resulting preferences form a frozen *feedback control*

memory that the controller queries as a prior, which lets the loop settle within a few rounds instead of exploring at test time.

Experiments on eight benchmarks covering code generation (Chen et al., 2021; Austin et al., 2021), mathematical reasoning (Cobbe et al., 2021; Hendrycks et al., 2021), reading comprehension (Yang et al., 2018; Dua et al., 2019), and interactive decision making (Shridhar et al., 2021; Prasad et al., 2024) show that MASOpt improves the average score from 80.58 to 83.40 over MASPOB (Hong et al., 2026) and the average interactive success rate from 39.23 to 64.85. In summary, our key contributions are three-fold.

- **Closed-Loop Formulation.** To the best of our knowledge, MASOpt is the first framework that formulates MAS prompt optimization as closed-loop control, where measurements taken inside an episode determine the prompts governing the remainder of that same episode.
- **Verified Actuation with CARA.** We build a controller from a trajectory observer, a minimal actuation module, and a feedback gate that guarantees a monotone measured error, and propose Cross-Actuation Relative Advantage to learn where to actuate from matched-state replay.
- **Strong Performance.** Across eight benchmarks, MASOpt outperforms the strongest offline MAS prompt optimizer by 2.82 points on average and by 25.62 points on interactive success.

2 Related Work

Automatic Prompt Optimization The sensitivity of LLMs to natural-language instructions has motivated a broad line of work on automatic prompt optimization. Early approaches select instruction candidates through task-level evaluation (Zhou et al., 2023; Pryzant et al., 2023), OPRO casts optimization as a language modeling problem (Yang et al., 2024), and PromptBreeder evolves the task prompts together with the mutation prompts that modify them (Fernando et al., 2024; Lin et al., 2024). Another group optimizes structured language-model programs, where DSPy compiles prompts as pipeline parameters (Khattab et al., 2024), TextGrad propagates textual feedback in analogy with automatic differentiation (Yuksekgonul et al., 2025), and SkillOpt treats agent skills the same way (Yang et al., 2026). However, all of these optimizers collect their signal at the level of a complete trial, so they learn that a prompt is bad without learning at which point of the trajectory it became bad.

System-Level Prompt Optimization for Multi-Agent Systems Prompt optimization is substantially harder in a MAS, because prompts interact through the topology and improving one agent in isolation changes what every downstream agent receives (Wu et al., 2024; Li et al., 2023). MAPRO casts the problem as maximum a posteriori inference with topology-aware feedback (Zhang et al., 2026b), MASPOB as budget-constrained black-box search with a graph neural surrogate (Hong et al., 2026), and MASPO scores a prompt by its contribution to downstream success (Wang et al., 2026). A further line searches the structure of the workflow itself (Zhang et al., 2025; Xia et al., 2026; Zhao et al., 2026). Overall, these methods establish that MAS prompts must be optimized as a coupled system, yet their optimization target remains a configuration selected before deployment.

Runtime Feedback and Online Adaptation A related line studies how language agents improve from feedback at test time. Self-Refine alternates between feedback generation and output refinement (Madaan et al., 2023; Gou et al., 2024), Reflexion converts environmental feedback into verbal reflections kept in an episodic memory (Shinn et al., 2023; Yao et al., 2023), and MemRL learns utilities over such memories while the base model stays fixed (Zhang et al., 2026a). These studies establish the value of non-parametric adaptation, yet their adaptation target is either a single output or a memory consumed by the next episode.

In this work, we focus on the prompt state of a running multi-agent system, with the goal of letting measurements taken inside an episode determine the prompts that govern the remainder of that episode. Unlike previous approaches that treat a configuration as a parameter chosen before deployment, MASOpt adopts a more deliberate design by treating every prompt edit as a control input whose effect is measured from the response of the plant. An ineffective edit is reverted, and an unresolved failure triggers another round.

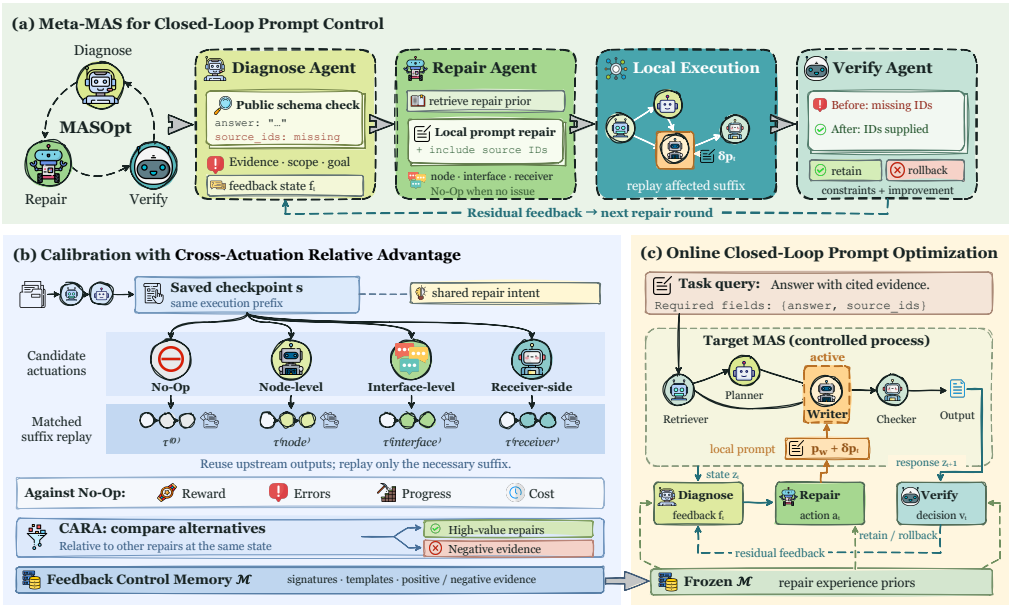


Figure 2: Overview of MASOpt. **(a)** The controller estimates a latent failure state, actuates a minimal prompt repair, and admits it through a feedback gate. **(b)** CARA replays candidate actuations from an identical prefix and distills their relative advantage into a feedback control memory. **(c)** At deployment the memory is frozen and every actuation is retained or reverted by the measured response.

3 MASOPT

We introduce MASOpt, an online prompt optimization framework that applies the feedback principle of classical control to a fixed multi-agent system. Because a plausible prompt edit may resolve the failure, do nothing, or create a new one downstream, MASOpt measures the response of the plant before deciding whether to keep it. The overview of MASOpt is illustrated in Figure 2, and its formulation, controller, offline calibration, and online loop are introduced as follows.

3.1 FROM OPEN-LOOP TO CLOSED-LOOP PROMPT OPTIMIZATION

Plant, Measurement, and Control Input We consider a fixed multi-agent workflow $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each node is an agent invocation and each edge carries information from an upstream agent to a downstream one (Zhang et al., 2025). The topology and the backbone LLMs stay fixed (Hong et al., 2026), and the only quantity a controller may change is the prompt state $\mathbf{P}_t = (p_1^t, \cdots, p_N^t)$ of the N agents, with $\mathbf{P}_0$ the deployed configuration. For a task x the plant emits a partial trajectory τ_t of responses, inter-agent messages, tool results, and environment observations, of which the controller sees only the measurement $z_t = f_{\text{obs}}(x, \tau_t)$ that excludes hidden labels. A checkpoint is eligible once an agent has produced a complete output that a downstream agent or the environment will consume. At such a checkpoint the controller applies a control input $a_t \in \mathcal{A}(z_t)$, which we call an *actuation*. It edits one agent instruction, one interface constraint, or one receiver-side rule, giving

$$\mathbf{P}_{t+1} = f_{\text{act}}\left(\mathbf{P}_t, a_t\right), \qquad e_t = \mathcal{E}\left(z_t\right), \tag{1}$$

where $f_{\text{act}}(\cdot)$ writes only the local modification a_t specifies, and the *measured error* $\mathcal{E}(\cdot)$ counts the label-free runtime violations the trajectory exposes, including failed public tests, unparsable outputs, missing message fields, unsatisfied dependencies, and invalid environment actions. Counting violations instead of estimating task reward is deliberate, since a violation cannot be traded against correctness. A no-operation input is always available.

Open-Loop and Closed-Loop Objectives An offline optimizer solves the open-loop problem $\max_{\mathbf{P}} \mathbb{E}_{x\sim\mathcal{D}}[R(\tau_T \mid \mathbf{P})]$ once and applies the solution unchanged, with R the final task metric over the task distribution $\mathcal{D}$ (Hong et al., 2026; Wang et al., 2026). MASOpt instead learns a policy π

that emits an actuation at every checkpoint,

$$\max_{\pi} \ \mathbb{E}_{x\sim\mathcal{D}}\left[R\left(\tau_T\right) - \lambda \sum\nolimits_{t=1}^{T} C\left(a_t\right)\right], \quad a_t \sim \pi\left(\cdot \mid x, z_{\le t}, \mathcal{M}\right), \tag{2}$$

where $C(a_t)$ is the cost of an actuation, λ trades control effort against task utility, and $\mathcal{M}$ is the calibrated experience obtained offline. The one difference that matters is that a_t conditions on $z_{\le t}$, which contains the measurements produced *after* a_{t-1} was applied. The formulation needs only the observable trajectory and a local edit, which lets it control a plant whose response cannot be modeled in advance.

3.2 A Closed-Loop Controller over Prompt State

MASOpt realizes π with three modules that mirror the observer, the actuator, and the feedback path of a regulator (Åström & Murray, 2021; Goodwin et al., 2001). Separating estimation, actuation, and acceptance makes an effect attributable, since the module proposing an edit never decides whether it worked.

Trajectory Observer for Latent Failure State A measurement reports the symptom of a failure, so the controller first estimates its cause. The trajectory observer maps the observable history and the retrieved experience to a structured latent failure state

$$s_t = f_{\mathrm{diag}}\left(x, z_{\le t}; \mathcal{M}\right), \tag{3}$$

where s_t records the detected failure, its evidence, its severity, its scope, and the behavior to restore. MASOpt checks the deterministic signals before any semantic estimation, so a failed public test or an invalid environment action enters s_t as hard evidence. The observer answers what must be corrected without committing to an edit, which leaves the actuation space free for the next module.

Minimal Actuation for Local Prompt Repair Large edits are easy to propose and impossible to attribute. MASOpt therefore restricts the control input to the smallest local change that can plausibly remove the diagnosed failure,

$$a_t = f_{\mathrm{rep}}\left(s_t, \mathbf{P}_t, z_t; \mathcal{M}\right), \tag{4}$$

where $f_{\mathrm{rep}}(\cdot)$ searches an action space of node-level edits, interface-level constraints, receiver-side rules, and the no-operation input. The retrieved experience supplies a prior over the actuation type, while the current context instantiates the text. MASOpt bounds the perturbation so that the effect of an actuation stays separable from the ordinary variance of the plant, which is what makes verification possible.

Feedback Gate for Verified Acceptance MASOpt treats a proposed repair as a hypothesis, so it replays the affected part of the workflow to a comparable checkpoint and compares the new measurement against the pre-actuation one,

$$g_t = f_{\mathrm{ver}}\left(s_t, z_t, z_{t+1}; \mathcal{M}\right), \tag{5}$$

where $g_t \in \{\textsc{Accept}, \textsc{Reject}\}$ carries the decision together with a description of whatever failure remains. The gate admits an actuation only when the measured error decreases and no hard constraint is violated, which yields the closed-loop update

$$\mathbf{P}_{t+1} = \begin{cases} f_{\mathrm{act}}\left(\mathbf{P}_t, a_t\right), & \text{if } g_t = \textsc{Accept} \text{ and } \mathcal{E}(z_{t+1}) < \mathcal{E}(z_t), \\ \mathbf{P}_t, & \text{otherwise.} \end{cases} \tag{6}$$

The gate is what closes the loop. When an actuation removes only part of the failure, the residual enters the next latent state s_{t+1} and drives another bounded round. This recursion separates MASOpt from one-shot prompt rewriting (Madaan et al., 2023), because the controller never assumes its first actuation was correct.

Property 1 (Monotone measured error)**.** Let $\tilde{e}_t$ denote the measured error evaluated at the prompt state MASOpt holds after the gate at checkpoint t. Under Equation 6 the sequence $\tilde{e}_0, \tilde{e}_1, \cdots, \tilde{e}_T$ is non-increasing, so the terminal prompt state satisfies $\tilde{e}_T \le \tilde{e}_0$.

Neither an open-loop optimizer nor an iterative rewriter has this guarantee. The former never measures its own effect and the latter keeps whatever it produced last, so both can admit an edit that raises the error (Fernando et al., 2024; Madaan et al., 2023). The guarantee holds independently of the observer and of the actuation module, which is the robustness that feedback buys over search (Åström & Murray, 2021).

3.3 Actuation Calibration with Cross-Actuation Relative Advantage

A closed loop still has to choose where to act. With no prior it must find the right actuation by trial at deployment, which is costly and slow to settle. MASOpt therefore calibrates the actuation choice offline, with no LLM parameter updated.

Matched-State Actuation Probing Comparing two actuations applied after different preceding trajectories confounds the effect of the actuation with the effect of the trajectory, so MASOpt replays every candidate from the same saved prefix. Specifically, for an informative checkpoint s we construct a repair intent that states the behavior to restore without prescribing where to apply it, which yields $\mathcal{A}_s = \{a_0, a_1, \cdots, a_K\}$ with a_0 the no-operation input and the remainder legal local actuations at different points of the workflow. MASOpt reuses the unaffected upstream outputs and replays only the necessary suffix, which turns each probe into a controlled response measurement from a known initial condition (Åström & Murray, 2021; Goodwin et al., 2001).

Cross-Actuation Relative Advantage For an actuation a at state s, we first score the paired outcome against the matched no-operation replay,

$$U(s,a) = \lambda_R \Delta R(s,a) + \lambda_E \Delta E(s,a) + \lambda_P \Delta P(s,a) - \lambda_C C(s,a), \tag{7}$$

where ΔR is the task-reward improvement over the matched no-operation, ΔE the reduction in observable execution error, ΔP the progress toward completion, and C the actuation and replay cost. Absolute utility is not comparable across heterogeneous states, since an easy checkpoint rewards every candidate and a hopeless one rewards none. We therefore score each candidate against the others issued from the same state, which gives the Cross-Actuation Relative Advantage

$$A_{\mathrm{CARA}}(s,a) = \frac{U(s,a) - \mu_{-a}(s)}{\sigma_{-a}(s) + \epsilon}, \quad \mu_{-a}(s) = \frac{1}{|\mathcal{A}_s| - 1} \sum\nolimits_{b \in \mathcal{A}_s, b \neq a} U(s,b), \tag{8}$$

where $\sigma_{-a}(s)$ is the standard deviation of $U(s,b)$ over the remaining candidates and ϵ stabilizes the ratio. Equation 8 takes the form of a leave-one-out group-relative advantage, with two differences. The group collects actuations applied at *different points of one workflow* instead of repeated samples of one prompt, and the estimate is stored in a memory instead of a parameter update. CARA therefore measures whether an actuation is unusually effective *for this execution state*, which is the question a MAS controller faces, because the node where an error becomes visible is often not the node where actuation has the largest effect (Zhang et al., 2026b).

Feedback Control Memory MASOpt collects the calibrated preferences into a reusable memory $\mathcal{M}$, where each entry records a failure signature, the topology context, an actuation type, a bounded prompt template, and its effectiveness statistics. We keep the actuations of consistently negative advantage as negative evidence, since knowing which repair to avoid at a state removes exploration the online loop would otherwise pay for (Zhang et al., 2026a). At deployment $\mathcal{M}$ is frozen, is queried only as a prior, and never stores task answers.

3.4 Online Closed-Loop Optimization

At deployment MASOpt starts every task from $\mathbf{P}_0$ and the frozen $\mathcal{M}$, with no ground-truth answer available. The plant runs normally until an eligible checkpoint, and if the evidence indicates no meaningful failure then no actuation is applied, which keeps the control effort near zero on the tasks the system already solves. Otherwise the observer builds s_t, retrieves matching entries from $\mathcal{M}$, and the actuation module instantiates a minimal edit applied only to the affected part of the workflow, reusing the unaffected upstream computation whenever the environment supports checkpoint replay. The gate then evaluates the post-actuation measurement from runtime signals alone, and Equation 6 retains or reverts the prompt state. The loop stops when the residual is empty, when the gate rejects every candidate, or when the round budget is exhausted. Every actuation is cleared at termination, so MASOpt adapts within a task while the target models, the topology, and the cross-task memory stay fixed, which is the opposite of a memory that accumulates across episodes (Shinn et al., 2023; Zhang et al., 2026a).

Table 1: Performance (%) on six non-interactive benchmarks. Baselines are ordered by increasing average and MASOpt is separated at the bottom. 'Avg.' is the unweighted six-benchmark mean and the best results are marked in **bold**.

Method	HumanEval	MBPP	GSM8K	MATH	HotpotQA	DROP	Avg.
IO	89.31	69.11	87.80	51.71	60.36	53.09	68.56
CoT	89.57	69.89	88.34	52.47	67.62	58.27	71.03
ReAct	87.79	66.08	88.91	52.61	65.61	67.25	71.38
PromptBreeder	88.80	70.38	91.97	52.13	68.76	71.85	73.98
Instinct	90.08	70.23	92.64	52.40	69.92	71.90	74.53
HiveMind	89.31	71.55	92.99	57.41	67.23	68.76	74.54
AFlow	91.09	79.96	93.36	53.83	73.42	79.48	78.52
MIPRO	91.35	80.65	92.80	54.90	74.37	79.13	78.87
MASPOB	94.15	80.65	93.90	57.05	75.43	82.28	80.58
MASOpt	**96.95**	**84.16**	**94.41**	**62.76**	**76.49**	**85.66**	**83.40**

4 EXPERIMENTS

We evaluate whether closing the loop inside an episode still improves a multi-agent system that a strong offline optimizer has already tuned.

4.1 EXPERIMENTAL SETTING

Benchmarks We evaluate MASOpt on eight benchmarks covering code generation (Chen et al., 2021; Austin et al., 2021), mathematical reasoning (Cobbe et al., 2021; Hendrycks et al., 2021), reading comprehension (Yang et al., 2018; Dua et al., 2019), and interactive decision making (Shridhar et al., 2021; Prasad et al., 2024). Benchmark details refer to Appendix A.1.

Baselines We compare MASOpt with reasoning-based prompting (Wei et al., 2022; Yao et al., 2023), prompt optimization (Fernando et al., 2024; Lin et al., 2024; Opsahl-Ong et al., 2024; Hong et al., 2026), and workflow generation (Zhang et al., 2025; Xia et al., 2026). MASPOB is the strongest published system-level MAS prompt optimizer (Hong et al., 2026), and baseline results on the six non-interactive benchmarks follow its evaluation setting. Detailed description of baselines refer to Appendix A.2.

Configuration Unless otherwise stated, every agent of the target MAS is served by `gpt-4o-mini` with temperature 0 and top-p 1, and the backbone parameters stay frozen (Yang et al., 2026). Specifically, MASOpt runs on the same locally executed workflow as the within-system baselines, so the topology, the roles, and $\mathbf{P}_0$ are identical across arms. Ground-truth rewards enter only the offline utility of Equation 7 and are never exposed to the online gate. Interactive runs use a 35-step limit on ALFWorld and a 20-step limit on TextCraft with at most two actuation rounds per checkpoint. We report pass@1 on the code benchmarks, accuracy on the mathematical ones, F1 on the reading benchmarks, and success rate on the interactive ones. Further details refer to Appendix A.3.

4.2 MAIN RESULTS

Capability on Program Synthesis and Mathematical Reasoning We evaluate MASOpt on HumanEval, MBPP, GSM8K, and MATH to assess its *verifiable reasoning capabilities*, where every intermediate failure leaves an executable trace a controller can measure without labels. We compare with PromptBreeder (Fernando et al., 2024), Instinct (Lin et al., 2024), MIPRO (Opsahl-Ong et al., 2024), AFlow (Zhang et al., 2025), and MASPOB (Hong et al., 2026) on the same backbone. As shown in Table 1, MASOpt outperforms MASPOB by 2.80 and 3.51 points on the two code benchmarks and substantially by 5.71 points on MATH. The size of each gain tracks how much room a task leaves for repair, since GSM8K already saturates at 93.90 under the offline optimizer while MATH carries the longest derivations and gains the most. An offline optimizer cannot produce this pattern, because it applies the same instruction to a saturated task and to an unsaturated one, while MASOpt spends an actuation only where a trajectory exposes a failure. Overall, these results highlight that a measurable runtime error signal is what turns a fixed prompt configuration into a correctable one.

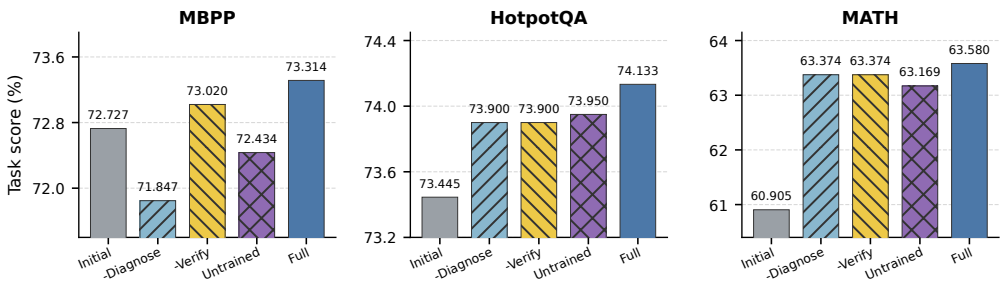


Figure 3: Experience ablations on MBPP, HotpotQA, and MATH. Each panel compares the full controller with the variants that remove diagnostic experience, verification experience, or all learned experience, where every arm reuses the same Initial outputs.

Capability on Multi-Hop and Discrete Reasoning We further evaluate MASOpt on HotpotQA and DROP to assess its *evidence-grounded reasoning capabilities*, where a failure is an incomplete aggregation of retrieved evidence instead of a hard execution error. We compare against the same baselines under the identical workflow. As shown in Table 1, MASOpt improves over MASPOB by 1.06 and 3.38 points. Notably, the asymmetry between the two benchmarks is informative, since DROP failures surface as arithmetic and span-selection violations that the observer detects directly, while HotpotQA failures are semantic and leave a particularly weak label-free signal (Yang et al., 2018; Dua et al., 2019). The formulation predicts this dependence, since the strength of the feedback signal bounds how much a controller can repair. Overall, these results highlight that MASOpt converts observable constraint violations into targeted prompt repairs.

Capability on Long-Horizon Interactive Decision Making We evaluate MASOpt on ALFWorld and TextCraft to assess its *long-horizon recovery capabilities*, where an early mistake compounds over the remaining steps and no single final answer exists. We compare with the same baselines under the shared typed action interface and the identical step limits. As shown in Table 2, MASOpt solves 24 of 80 TextCraft tasks and significantly lifts the three-column average from 39.23 to 64.85. This margin is far larger than on the non-interactive benchmarks, which follows from the formulation, since an interactive episode emits a dense stream of dependency and inventory violations the observer reads at every step while a single-turn task emits one measurement at the end. The unseen split gains more than the seen split for the same reason, because an unfamiliar layout produces more invalid actions for the observer to read. Overall, the benefit of closing the loop grows with the number of measurements the plant makes available.

Table 2: Interactive success rate (%) under the shared action interface and the identical step limits. 'Avg.' weights the three score columns equally and the best results are marked in **bold**.

Method	ALFWorld		TextCraft	Avg.
	Seen	UnSeen		
HiveMind	28.57	27.61	12.50	22.89
MASPOB	47.86	50.75	13.75	37.45
MIPRO	51.43	50.00	12.50	37.98
AFlow	50.71	54.48	12.50	39.23
MASOpt	**75.00**	**89.55**	**30.00**	**64.85**

5 Analyses

Each study below tests a prediction that the formulation makes, asking in turn whether the closed loop depends on its calibrated experience, whether it improves the instructions themselves, and whether it survives a change of workflow architecture and of backbone model.

5.1 Effect of Diagnostic and Verification Experience

The feedback control memory supplies the prior that keeps the online loop short. To assess its contribution, we remove calibrated experience from the observer, the gate, or both, holding the controller and workflow fixed. As shown in Figure 3, the full controller is the strongest arm on all three benchmarks. The ablation ordering identifies state estimation as the main bottleneck. Removing diagnostic experience is the most damaging setting, since an actuation aimed at a misidentified failure

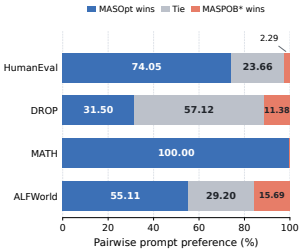


Figure 4: Pairwise prompt preference, with MASOpt wins, ties, and baseline wins.

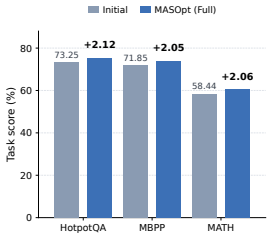


Figure 5: Adaptation to three alternative workflows, with Initial in gray and Full in blue.

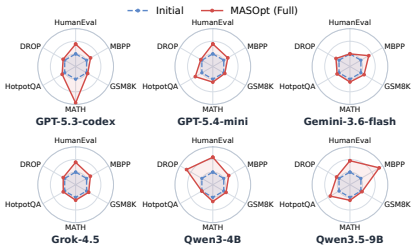


Figure 6: Adaptation across six backbones, with each Initial score normalized to 100.

cannot reduce the measured error however well the gate is calibrated. Removing verification experience is milder, because the gate retains the deterministic runtime constraints. All variants reuse the Initial outputs, enabling comparison from the same starting trajectories. Overall, the calibrated prior removes the exploration that the loop would otherwise pay for at deployment (Zhang et al., 2026a).

5.2 Advantage of the Optimized Prompts

We next examine whether the closed loop improves instruction quality beyond what task accuracy reveals. We score the retained prompts against the offline-optimized ones under an anonymous six-dimension rubric applied in both presentation orders. As shown in Figure 4, MASOpt wins more comparisons than the baseline on all four datasets, with uniform preference on MATH but predominantly ties on DROP. This pattern is consistent with MATH exposing specific derivation defects, whereas a DROP repair can leave much of the instruction intact. Reporting ties separately distinguishes comparable prompt quality from a preference for either method. Overall, the advantage of MASOpt appears exactly where the runtime signal is specific enough to name the defect.

5.3 Generalizability across Workflow Architectures

We then test dependence on workflow architecture by rerunning the same controller on three simplified architectures with the backbone and task set fixed. As shown in Figure 5, the closed loop improves every architecture by about two points (Zhang et al., 2025). The gains range from 2.05 to 2.12 points despite the different initial scores, suggesting a consistent benefit across these settings. Overall, the consistent gains support transferring local error correction across the tested workflows without redesigning the controller.

5.4 Generalizability across Backbone Models

We finally test the same controller on six backbones spanning different families and scales, pairing each run with its own initial workflow. As shown in Figure 6, the paired comparison is non-negative on every model and benchmark. The only tie occurs where the initial workflow already solves the benchmark completely. The gate predicts this shape, since it accepts only an actuation that reduces the measured error, so a stronger backbone offers fewer opportunities to act. Normalization shows relative changes from each backbone's own initial score. Overall, MASOpt transfers across model families without structural recalibration, while the size of its benefit follows how much observable failure the backbone leaves behind.

6 Conclusion

We introduce MASOpt, which turns MAS prompt optimization from an open-loop search into a closed-loop control problem, where a trajectory observer estimates a latent failure state, a minimal actuation repairs one prompt, and a feedback gate admits the repair only when the measured error decreases. Experiments on eight benchmarks show that MASOpt improves over the strongest offline MAS prompt optimizer on every benchmark with the backbone models and the workflow topology left fixed.

AI USE STATEMENT

In this work, we used generative AI tools only for final-stage language polishing and grammar refinement of the manuscript. We have not used generative AI tools for generating research ideas, designing methods, conducting experiments, implementing code, analyzing data, interpreting results, or producing scientific claims, and other generative AI-assisted tasks requiring disclosure are not applicable to this work. We have reviewed all AI-assisted revisions to ensure their correctness and consistency with the authors' intended content. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have made efforts to ensure the reproducibility of this work by providing detailed descriptions of the proposed method, experimental settings, implementation details, and evaluation protocols in the main paper and supplementary materials. The supplementary materials include the source code required to reproduce our experiments, and we will publicly release the code after the review process to further facilitate verification and future research. The content of this paper can be completely reproduced.

REFERENCES

Karl Johan Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2nd edition, 2021. ISBN 9780691193984. URL https://fbswiki.org/wiki/index.php/FBS.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors. In *International Conference on Learning Representations*, pp. 20094–20136, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/578e65cdee35d00c708d4c64bce32971-Paper-Conference.pdf.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 2368–2378. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1246. URL https://aclanthology.org/N19-1246/.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 13481–13544. PMLR, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*. Prentice Hall, 2001. ISBN 9780139586538.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. In *International Conference on Learning Representations*, pp. 57734–57811, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/fef126561bbf9d4467dbb8d27334b8fe-Paper-Conference.pdf.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving With the MATH Dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/be83ab3ecd0db773eb2dc1b0a17836a1-Paper-round2.pdf.

Zhi Hong, Qian Zhang, Jiahang Sun, Zhiwei Shang, Mingze Kong, Xiangyi Wang, Yao Shu, and Zhongxiang Dai. MASPOB: Bandit-Based Prompt Optimization for Multi-Agent Systems with Graph Neural Networks. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=A5venxSvpw.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. In *International Conference on Learning Representations*, pp. 54928–54958, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/f1cf02ce09757f57c3b93c0db83181e0-Paper-Conference.pdf.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. In *Advances in Neural Information Processing Systems*, volume 36, pp. 51991–52008. Curran Associates, Inc., 2023. doi: 10.52202/075280-2264. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a3621ee907def47c1b952ade25c67698-Paper-Conference.pdf.

Xiaoqiang Lin, Zhaoxuan Wu, Zhongxiang Dai, Wenyang Hu, Yao Shu, See-Kiong Ng, Patrick Jaillet, and Bryan Kian Hsiang Low. Use Your INSTINCT: INSTruction optimization for LLMs usIng Neural bandits Coupled with Transformers. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 30317–30345. PMLR, 2024. URL https://proceedings.mlr.press/v235/lin24r.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. ADaPT: As-Needed Decomposition and Planning with Language Models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-naacl.264. URL https://aclanthology.org/2024.findings-naacl.264/.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic Prompt Optimization with "Gradient Descent" and Beam Search. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7957–7968. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.494. URL https://aclanthology.org/2023.emnlp-main.494/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0IOX0YcCdTn.

Zhexuan Wang, Xuebo Liu, Li Wang, Zifei Shan, Yutong Wang, Zhenxi Song, and Min Zhang. MASPO: Joint Prompt Optimization for LLM-based Multi-Agent Systems. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=iZrp2P55ie.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. doi: 10.52202/068431-1800. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Yihan Xia, Taotao Wang, Shengli Zhang, Zhangyuhua Weng, Bin Cao, and Soung Chang Liew. HiveMind: Contribution-Guided Online Prompt Optimization of LLM Multi-Agent Systems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29767–29774, 2026. doi: 10.1609/aaai.v40i35.40222. URL https://ojs.aaai.org/index.php/AAAI/article/view/40222.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers. In *International Conference on Learning Representations*, pp. 12028–12068, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/3339f19c5fcee3ad74502947a32be9e6-Paper-Conference.pdf.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. *arXiv preprint arXiv:2605.23904*, 2026. URL https://arxiv.org/abs/2605.23904.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380. Association for Computational Linguistics, 2018. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4. URL https://www.nature.com/articles/s41586-025-08661-4.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations*, pp. 34040–34077, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, Feiyu Xiong, Yutao Qi, Bo Tang, and Muning Wen. MemRL: Self-Evolving Agents via Runtime Reinforcement Learning on Episodic Memory. *arXiv preprint arXiv:2601.03192*, 2026a. URL https://arxiv.org/abs/2601.03192.

Zheyuan Zhang, Lin Ge, Hongjiang Li, Weicheng Zhu, Chuxu Zhang, and Yanfang Ye. MAPRO: Recasting Multi-Agent Prompt Optimization as Maximum a Posteriori Inference. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 4458–4480. Association for Computational Linguistics, 2026b. ISBN 979-8-89176-386-9. doi: 10.18653/v1/2026.findings-eacl.233. URL https://aclanthology.org/2026.findings-eacl.233/.

Mingming Zhao, Xiaokang Wei, Yuanqi Shao, Kaiwen Zhou, Lin Yang, Siwei Rao, Junhui Zhan, and Zhitang Chen. A^2Flow: Automating Agentic Workflow Generation via Self-Adaptive Abstraction Operators. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29930–29938, 2026. doi: 10.1609/aaai.v40i35.40240. URL https://ojs.aaai.org/index.php/AAAI/article/view/40240.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Arik. Multi-Agent Design: Optimizing Agents with Better Prompts and Topologies. In *International Conference on Learning Representations*, pp. 15844–15872, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/1ab4e0e8f35078109bc78d7b465d306f-Paper-Conference.pdf.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large Language Models Are Human-Level Prompt Engineers. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=92gvk82DE-.

A Experimental Details

A.1 Benchmarks

HumanEval (Chen et al., 2021) and **MBPP** (Austin et al., 2021) evaluate program synthesis from a natural-language specification, where a prediction is correct when it passes the hidden tests. **GSM8K** (Cobbe et al., 2021) and **MATH** (Hendrycks et al., 2021) evaluate multi-step mathematical reasoning, where we adopt the fixed Level-5 test subset of Zhang et al. (2025) that contains 155 Prealgebra, 124 Number Theory, 108 Precalculus, and 99 Counting and Probability problems. **HotpotQA** (Yang et al., 2018) evaluates multi-hop reasoning over two supporting paragraphs, and **DROP** (Dua et al., 2019) evaluates discrete reasoning that requires counting, sorting, and arithmetic over a passage. **ALFWorld** (Shridhar et al., 2021) evaluates embodied household tasks in a text interface with separate seen and unseen splits, and **TextCraft** (Prasad et al., 2024) evaluates compositional crafting in which a target item is obtained only after its recipe dependencies are satisfied in order. Success on the two interactive benchmarks therefore depends on recovering from environment feedback instead of on a single final answer, which is what makes them the setting where a closed loop has the most measurements to act on.

Table 3 lists the calibration pools, test sizes, and evaluation metrics of the eight benchmarks.

Table 3: Datasets, splits, and evaluation metrics. All reported scores are on a 0–100 scale.

Task family	Benchmark	Calibration	Test	Metric
Code generation	HumanEval	33	131	pass@1
	MBPP	86	341	pass@1
Mathematical reasoning	GSM8K	264	1,055	answer accuracy
	MATH	119	486	answer accuracy
Reading comprehension	HotpotQA	200	800	token F1
	DROP	200	800	answer F1
Interactive environments	ALFWorld	33	140 seen / 134 unseen	success rate
	TextCraft	20	80	success rate

A.2 Baselines

We compare MASOpt with three groups of methods. The first group is direct and reasoning-based prompting, covering **IO** that answers in one call, **CoT** (Wei et al., 2022) that elicits an intermediate reasoning chain, and **ReAct** (Yao et al., 2023) that interleaves reasoning with tool or environment actions. The second group is prompt optimization, covering **PromptBreeder** (Fernando et al., 2024) that evolves task and mutation prompts, **Instinct** (Lin et al., 2024) that searches instructions with a neural bandit, **MIPRO** (Opsahl-Ong et al., 2024) that jointly optimizes instructions and demonstrations of a language-model program, and **MASPOB** (Hong et al., 2026) that treats fixed-topology MAS prompt optimization as budget-constrained black-box search. The third group is workflow generation and adaptation, covering **AFlow** (Zhang et al., 2025) that searches code-represented workflows with Monte Carlo tree search and **HiveMind** (Xia et al., 2026) that attributes credit across agents and refines the lowest-contribution node. Baseline results on the six non-interactive benchmarks follow the evaluation setting of Hong et al. (2026), and the interactive comparison uses the shared typed action interface described below.

A.3 Implementation Protocols

Every agent of the target MAS is served by `gpt-4o-mini` with temperature 0 and top-p 1, and MASOpt never updates a backbone parameter. The calibration pools listed in Table 3 are divided into three disjoint parts for experience construction, experience selection, and controller selection. Ground-truth rewards enter only the offline utility of Equation 7 and are never visible to the online feedback gate, which decides using the label-free measured error of Equation 1 together with the deterministic runtime constraints.

On the interactive benchmarks, all compared systems use the same typed action interface, the same native environment success condition, and the same step limits of 35 on ALFWorld and 20 on

TextCraft. MASOpt additionally reads the public dependency and inventory constraints that the environment exposes at every step and turns a detected violation into a bounded actuation. Its frozen controller generates task-local repairs from these public observations alone, and it neither updates the cross-task memory nor selects prompts using test accuracy.

A.4 Measured Error

The measured error $\mathcal{E}$ of Equation 1 counts constraint violations that the runtime exposes without any label. On the code benchmarks it counts syntax errors, raised exceptions, and failures of the public tests attached to the problem statement. On the mathematical benchmarks it counts unparsable final answers, expressions that fail a symbolic well-formedness check, and disagreement among the solver branches of the workflow. On the reading benchmarks it counts answers whose span is absent from the retrieved passages, answers of the wrong type for the question, and missing fields in the message that the aggregator receives. On the interactive benchmarks it counts invalid actions rejected by the environment, actions whose preconditions are unsatisfied, and repeated states that make no progress.

Every term of $\mathcal{E}$ is a violation of a stated constraint and not an estimate of task reward, so reducing it cannot be traded against correctness the way a learned proxy can. The hidden evaluation label never enters $\mathcal{E}$, so the feedback gate of Equation 6 makes its accept and reject decisions from information that is also available at deployment.

A.5 Analysis Protocols

Prompt quality We anonymize the prompts MASOpt retains at termination together with those the offline optimizer commits before deployment (Hong et al., 2026), then score both with a fixed six-dimension rubric in the two presentation orders and count a difference above 0.5 as a win. The six dimensions cover clarity, specificity, coverage of the task constraints, handling of edge cases, compatibility with the downstream agent, and overall quality.

Alternative architectures We replace the original graphs with one answer-generation call plus formatting on HotpotQA, one code-generation branch plus test and repair on MBPP, and two text solvers with self-consistency on MATH, then run the identical controller on each.

A.6 Control Effort

We also record what the closed loop costs. We measure the total API calls of a complete TextCraft test under the shared interface and the identical 20-step limit, then compare them with the two strongest baselines. As reported in Table 4, MASOpt uses 888 calls against 699 and 684, an increase of 27%, and converts it into 24 solved tasks against 10 and 11. The asymmetry between the two ratios follows from Equation 2, since no actuation is applied at a checkpoint whose measurement indicates no failure, which concentrates the budget on the episodes that would otherwise fail. Overall, the control effort scales with the number of detected failures and not with the size of the test set.

Table 4: Control effort and task utility on TextCraft under the shared action interface and the 20-step limit.

Method	Calls	Solved	Succ.
MIPRO	699	10 / 80	12.50
MASPOB	684	11 / 80	13.75
MASOpt	888	**24 / 80**	**30.00**

A.7 Controller Hyperparameters

The utility weights of Equation 7 are set so that the task-reward term dominates, with $\lambda_R = 1.0$, $\lambda_E = 0.5$, $\lambda_P = 0.3$, and $\lambda_C = 0.1$, and with $\epsilon = 10^{-6}$ in Equation 8. The candidate set at a probed checkpoint contains the no-operation input and $K = 4$ legal local actuations. At deployment the controller retrieves the five memory entries whose failure signature best matches s_t and allows at most two actuation rounds per checkpoint.

A.8 Actuation Space

The actuation space contains four types of local edit. A node-level actuation rewrites a bounded portion of the instruction of a single agent. An interface-level actuation adds a constraint on the message one agent sends to a specific downstream neighbor. A receiver-side actuation adds a rule governing how an agent consumes a message it receives. The no-operation actuation leaves the prompt state unchanged and is always available, which lets the controller decide that no repair is warranted.

B Additional Implementation Details

This section expands the controller in Section 3 at the level of runtime objects, prompt placement, and replay. We retain the terminology of the main text: the trajectory observer estimates the failure state, minimal actuation proposes a local change, and the feedback gate evaluates its observed effect. These controller components operate alongside the fixed target workflow. The listings below are compact descriptions of the interfaces and control logic; transport handling and serialization are omitted for readability.

B.1 Task Contracts and Observable Inputs

The controller receives a typed task object containing the benchmark name, a task identifier, the public problem statement, and, for code tasks, the required entry point and examples exposed by that statement. The adapter constructs this object before the controller runs. For HotpotQA, it concatenates the supplied context paragraphs with the question; for DROP, it retains the passage–question input; for mathematical tasks, it exposes the problem text. This explicit interface makes the provenance of each runtime observation inspectable.

The task contract then separates four obligations: the requested output, the responsibility of each workflow role, the information that must cross each edge, and the conditions that must remain true after an intervention. It also lists the eligible checkpoint roles. The contract provides a stable reference when the observer evaluates a downstream output: a formatting failure is assessed against the receiver's output requirement, while a reasoning failure is assessed against the original task. In particular, the word *reference* in a reference goal denotes the public task requirement, not a reference answer.

Public code examples are extracted from visible docstrings and input–output examples. Their lexical content is preserved so that escaped characters and string-valued examples are not silently reinterpreted. The adapter deduplicates repeated examples and preserves the requested function interface. A model-proposed test is a candidate diagnostic probe; it becomes an execution observation only after the corresponding code has actually been run. This distinction is also used when preparing evidence for the next repair round.

Runtime task and contract: abbreviated schema

```
TaskInput:
  dataset, task_id, problem
  entry_point                  # Code tasks only
  public_examples              # Extracted from visible task text

TaskContract:
  reference_goals              # Public requirements
  node_obligations             # Responsibility of each role
  edge_contracts               # Information required by receivers
  checkpoint_schedule         # Eligible observation points
  hard_constraints            # Requirements preserved by a repair
```

B.2 CHECKPOINT REPRESENTATION

A checkpoint is a snapshot of the execution context, not just the most recent answer. It records the outputs already produced, the messages received by executed nodes, the effective prompt state, the remaining budget, and the cached call records needed for replay. Each call record retains its original inputs and prompt together with its effective inputs, applied patch, output, error status, and cache status. The original and effective versions are stored separately so that a local prompt edit can be distinguished from an ordinary workflow call.

Repeated invocations of one role receive distinct identifiers, such as `AnswerGenerate#0` and `AnswerGenerate#1`. This matters for workflows with multiple candidate branches: an instruction attached to one invocation should not silently modify every invocation of the same role. The checkpoint also records a digest of the visible example set. Replaying code under a different public-check contract is rejected because its pre- and post-intervention measurements would no longer be comparable.

Table 5: Checkpoint fields and their role in matched-state replay. Field names are abbreviated where necessary.

Field group	**Recorded information and purpose**
Task and checkpoint IDs	Bind the snapshot to one task, one invocation, and one comparison state.
Executed nodes and outputs	Preserve the observed prefix and distinguish repeated calls of the same role.
Edge messages	Record the effective receiver inputs, including branch lists and selected answers.
Tool and workflow state	Retain the information required by the adapter to resume execution and verify the public-check contract.
Prompt state	Identify which instructions were active when each cached output was produced.
Remaining budget	Bound the additional work that may be performed after the snapshot.
Cached call records	Reuse compatible upstream computations and reconstruct the trace of an accepted continuation.

B.3 FAILURE STATES AND EVIDENCE BINDING

The observer returns a structured failure state with a failure family, severity, uncertainty, evidence spans, a possible source node, and an impact cone. It also records whether the diagnosis was valid and whether semantic assessment was used. These fields distinguish three questions that should not be collapsed: whether an observable violation exists, where it may have originated, and which local change is worth testing. A source-node prediction is evidence for candidate construction; it is not an instruction to accept a repair at that node.

The observer first applies deterministic checks to the available output. Examples include parsing code, checking a declared function signature, inspecting an actual tool exception, and comparing a recorded execution with a visible example. Semantic assessment is used to describe failures that require interpretation of the task or trajectory. Its output must satisfy the diagnosis schema and refer to visible evidence. Stylistic preferences, unsupported speculation about hidden tests, and a replacement answer are not valid substitutes for a diagnosis.

The base semantic interface distinguishes a healthy output, a concrete risk, and a demonstrated error. The uncertainty field records the assessor's uncertainty separately from severity. This separation is useful when the output is incomplete: uncertainty about correctness should not automatically be turned into a definite failure. If the required evidence cannot be validated, the controller preserves the existing state rather than treating an invalid assessment as a successful check.

B.4 REPAIR INTENT AND LOCAL ACTUATION

A repair intent describes the behavior to restore before an actuation location is selected. Its fields specify the observed problem, desired behavior, forbidden behavior, applicability condition, and minimality constraint. For example, a formatter that changes an already valid upstream answer

calls for preservation of semantic content; it does not call for solving the original question again. Separating intent from placement lets the same corrective objective be compared at a producer, an interface, or a receiver.

The candidate compiler combines this intent with the legal locations exposed by the dataset adapter. Each candidate contains its type, target invocation or edge, patch text, replay start, and an estimate of the affected scope. The no-operation candidate has empty patch text and provides the reference response. A candidate is executable only if its target can be bound to a concrete invocation in the saved workflow context. A role-level memory entry is therefore not assigned arbitrarily to a branch when several branch instances match.

Table 6: Placement of the local actuations used in the main formulation. All three editing operations preserve the target workflow's role interfaces.

Actuation	**Location**	**Execution semantics**
Node	Instruction of one identified invocation	Add the bounded repair rule when that invocation constructs its model request.
Interface	A specified producer–receiver handoff	Associate a contract with the selected message while retaining its original content and branch identity.
Receiver	Instruction governing message consumption	Add a receiver guard without rewriting the cached upstream computation.
No operation	Matched replay start	Execute the comparable suffix with no additional prompt rule.

B.5 Prompt Injection without Interface Changes

The execution layer copies the operator being actuated before modifying its request construction. It does not mutate a shared role object, which prevents one branch's temporary rule from leaking into a sibling branch. Node rules and receiver guards are appended after the original template has been validated and formatted. This order preserves required placeholders and avoids interpreting braces in a mathematical expression or a code fragment as new template arguments.

For a text interface, the patch is attached to the identified handoff. When the receiver takes a list of candidate answers, the controller binds the source invocation to one element of that list and checks the binding against the cached producer output. An ambiguous binding is rejected. For executable code or a mixed task–message input, the constraint is delivered as prompt metadata at the receiver's request boundary; it is not concatenated into the executable program. Thus a request to change an interface contract does not accidentally become a change to the candidate program itself.

The effective prompt, effective input, and patch are logged together. These records make it possible to determine whether a proposed edit reached its intended recipient. A replay that returns an output without successfully executing the target invocation is not evidence that the actuation worked.

B.6 Replay Scope and Cache Reuse

Replay invalidates the target invocation and its downstream dependency cone. An upstream call can be reused only when its invocation identity, input arguments, and original prompt agree with the saved record and it lies outside the invalidated scope. A cache hit contributes its saved output to the reconstructed trace, but it does not count as a fresh model request. Descendants of a changed node are recomputed because their inputs may have changed even when their own prompt text has not.

The comparison horizon is chosen consistently for the candidate group. If all legal targets lie within a resumable local barrier, the replay may stop at that barrier. Otherwise it continues to the end of the workflow. Offline reward evaluation of a locally stopped replay requires a completed downstream continuation. This prevents a partial output from being compared with a terminal answer as though the two represented the same outcome.

The matched no-operation replay uses the selected candidate's replay start. It is not replaced with an unrelated earlier run. In a stochastic service, identical saved prefixes and decoding settings reduce an important source of variation, but they do not make two model responses identical. The execution

record therefore preserves the actual responses and the common comparison horizon, rather than assuming deterministic equivalence from the temperature setting alone.

B.7 Feedback-Gate Checks and Termination

The feedback gate checks more than the sign of the residual change. Both the reference and candidate executions must complete at the same comparison horizon, their required checks must be verified, and the candidate must avoid hard-constraint violations. The measured residual must decrease, progress must not decrease, and the replay pair must fit within the configured additional-call budget. A failed request, missing target execution, or unavailable check does not satisfy these conditions.

Feedback gate: compact implementation logic

```
def admit(reference, candidate, call_budget):
    comparable = (
        reference.comparison_horizon == candidate.comparison_horizon
    )
    verified = (
        reference.success and candidate.success
        and reference.checks_verified and candidate.checks_verified
    )
    improvement = (
        candidate.post_residual < reference.post_residual
        and candidate.progress_after >= reference.progress_after
    )
    affordable = reference.calls + candidate.calls <= call_budget
    return (comparable and verified and improvement and affordable
            and not candidate.hard_violation)
```

For successive proposals at the same saved state, the accepted state becomes the incumbent and a rejected proposal leaves it unchanged. This is the operational meaning of verified retention: the comparison concerns the same observable requirements and horizon. It does not make a public check a hidden-test oracle. Passing a visible code example, producing a parseable mathematical answer, or preserving a message schema establishes the corresponding local property; final task scoring remains a separate operation.

The loop terminates when no eligible failure remains, no applicable memory entry can be instantiated, the candidate is rejected without a further permitted round, or the budget is exhausted. At episode termination, temporary patches and task-local caches are cleared. Outstanding asynchronous calls belonging to that episode are drained before its state is released, so a late branch response cannot modify a later task.

C Offline Calibration and Online Execution

C.1 Matched-State Calibration Procedure

Offline calibration supplies preferences over local actuations. It begins with an observed workflow trace and selects informative checkpoints from the development data. At each checkpoint, the observer identifies the visible failure and the intent module describes the behavior to restore. Legal candidates share that intent and saved prefix; they differ in where the bounded rule is applied. The no-operation response anchors the comparison. Candidate groups with incomplete replays or incompatible horizons are excluded from CARA estimation.

Offline calibration: algorithmic outline

```
Input: development tasks, fixed workflow, calibration configuration
Output: feedback control memory

for each development task:
    execute the fixed workflow and save eligible checkpoints
    for each selected checkpoint:
        diagnose the observable failure
        construct one repair intent
        compile NO_OP and legal local actuations
        replay every candidate from the saved prefix
        complete downstream execution when reward scoring requires it
        if the replay group is complete and comparable:
            score development rewards outside the runtime controller
            compute paired utilities and leave-one-out CARA
            record positive, neutral, and harmful observations

aggregate observations by task and repair semantics
select applicable primitives using development evidence
freeze the memory for online use
```

C.2 Utility Terms and CARA Normalization

Equation 7 combines reward improvement, reduction in the observed residual, progress, and control effort. The implementation computes each difference against the same no-operation response. The cost term can be decomposed into additional token usage, actuation size, and replayed scope. Additional token usage is counted above the matched no-operation cost, while the replay term tracks the share of calls that could not be reused. This decomposition distinguishes a small instruction edit with a long downstream replay from a similarly sized edit near the terminal output.

The leave-one-out mean in Equation 8 is computed over all other members of the checkpoint's candidate group, including the no-operation member when the scored candidate is an intervention. The implementation uses the population standard deviation of those other utilities. A group with fewer than two completed members does not define the required comparison. If the other utilities coincide, the stabilizer ϵ keeps the denominator defined; the resulting ratio must still be interpreted together with absolute reward gain and support.

Relative advantage alone is insufficient for memory admission. A candidate may have a positive CARA score because it is less harmful than the other proposals. The memory therefore retains absolute reward changes and harm counts alongside CARA. The online retriever can require positive mean reward gain and sufficient support before a primitive is even considered. CARA ranks responses within a state; it does not certify correctness or replace the feedback gate.

C.3 Memory Records and Independent Support

Each memory record binds a repair intent to a failure family, a topology motif, an actuation location, and an operator. It stores an applicability condition, a bounded patch template, a replay scope, and statistics describing the observed effects. The implementation groups records by repair semantics as well as location. Two unrelated repairs at the same agent are therefore not merged merely because they share a target node.

Repeated checkpoints from one task are not counted as independent tasks. For a primitive p, let $\mathcal{T}_p$ be its supporting tasks and let $\mathcal{O}_{p,x}$ contain its observations on task x. The task-balanced mean of an observed quantity v is

$$\bar{v}_p = \frac{1}{|\mathcal{T}_p|} \sum_{x \in \mathcal{T}_p} \frac{1}{|\mathcal{O}_{p,x}|} \sum_{o \in \mathcal{O}_{p,x}} v_o. \tag{9}$$

The same task grouping is used for mean reward gain, residual gain, and CARA. Support is the number of distinct tasks. Neutral outcomes remain in the denominator, and harmful outcomes remain available as negative evidence. The stored confidence statistic uses a smoothed success fraction, $(n_{\text{success}} + 1)/(n_{\text{support}} + 2)$; it is an operational retrieval statistic rather than a calibrated probability of success on an unseen task.

At runtime, retrieval first filters by failure family, topology motif, and applicability conditions. It then applies the configured support, confidence, reward-gain, and CARA requirements. Matching negative records can veto an otherwise eligible combination of level, location, and operator. The remaining entries are ordered using their reward, confidence, harm, and relative-advantage statistics. This is structured retrieval over compatible control primitives, so the selected entry still has to be bound to the current invocation and verified after execution.

C.4 Online Execution Procedure

The online controller reads the frozen memory but does not update it. Its task-local state contains the currently retained prompt state, the observed trajectory, the checkpoints already attempted, and the remaining budget. The following outline emphasizes the decision boundaries; the workflow adapter handles benchmark-specific execution and measurement.

Online control: algorithmic outline

```
Input: task, deployed prompts, fixed workflow, frozen memory
Initialize an empty task-local patch state and execution record

execute the workflow until an eligible checkpoint is reached
while an intervention remains permitted:
    observe the output and validate the diagnosis
    if no eligible failure is observed: continue normal execution
    retrieve compatible memory entries
    bind a bounded repair to a legal invocation or interface
    if no candidate can be bound: retain the incumbent
    otherwise:
        obtain the comparable no-operation response
        replay the affected suffix with the candidate patch
        apply the feedback-gate checks
        if accepted: retain the repaired state and continue
        otherwise: restore the incumbent
        if another round is allowed:
            diagnose the remaining failure using recorded evidence

return the final workflow output
clear task-local patches, replay state, and pending branch work
```

In the bounded feedback implementation, an executed but rejected proposal may supply evidence for the next repair at the same checkpoint. The feedback contains the actual candidate output, its hash, verified runtime checks, and the remaining failure. A verifier's explanation is retained as a fallible assessment, not converted into a reference answer. Rejection therefore has two distinct consequences: the proposal does not replace the incumbent, but its observed failure can help avoid repeating the same ineffective edit.

D Agent Roles and Dataset-Specific Workflows

The six non-interactive workflows use role-specific operators inherited from the fixed workflow implementation of Hong et al. (2026). We describe the role interfaces before giving the dataset-specific execution order. The diagrams below specify these existing workflows. Repeated calls such as `Solve` $\times 3$ denote multiple invocations of a role with distinct runtime identities; they do not imply three separately trained models.

D.1 AGENT INTERFACES

Each operator receives the task and the upstream fields required by its role, formats a prompt, and returns a structured response. The controller attaches local rules at this boundary. An actuation must preserve both the caller's argument contract and the receiver's required response fields. A response that contains a plausible answer but omits the field consumed downstream is an interface failure.

Table 7: Principal role interfaces in the fixed non-interactive workflows. The exact operator schema determines which response field is consumed.

Role	Consumed information	Responsibility
AnswerGenerate	Context and question	Produce a candidate answer with supporting reasoning.
Solve	Passage/question or mathematical problem	Construct a task-specific solution before downstream formatting or verification.
ScEnsemble	Problem and an ordered candidate list	Select a candidate through the prescribed response schema while retaining branch identity.
FormatAnswer / Format	Question or problem plus the selected answer	Produce the short answer required by the evaluation interface.
CodeGenerate / CustomCodeGenerate	Public specification and entry point	Generate executable code with the required interface.
Test	Problem, candidate program, and entry point	Evaluate the available public checks and expose the resulting feedback.
FixCode	Specification, candidate code, and observed error	Repair the reported defect while preserving the function contract.
Programmer	Mathematical problem and, where applicable, a selected solution	Construct and execute a program for the numerical computation.
Extract / RefineAnswer	Problem and the preceding computational result	Extract or normalize the answer produced by the preceding stage.
DetailedSolution / GenerateSolution	Mathematical problem	Produce text-based candidate derivations for aggregation.

D.2 HOTPOTQA: EVIDENCE, SELECTION, AND FORMATTING

HotpotQA requires an answer supported by the supplied multi-hop context (Yang et al., 2018). The workflow invokes `AnswerGenerate` three times, passes the resulting candidates to `ScEnsemble`, and asks `FormatAnswer` to produce the final short answer. The ensemble receives the candidates as an ordered list. Its choice and the selected candidate remain observable before the formatter runs, which separates a selection error from a subsequent formatting change.

HotpotQA workflow and handoff requirements

```
Input: supplied context paragraphs and question
  AnswerGenerate #0 --+
  AnswerGenerate #1 --+--> ScEnsemble --> FormatAnswer --> answer
  AnswerGenerate #2 --+

Generation: use the supplied context and retain the target relation.
Selection: preserve candidate identities and inspect supporting
    evidence.
Formatting: retain the selected answer's meaning and required
    specificity.
```

A repair should identify the broken relation or handoff. For example, missing support for the selected entity concerns generation or selection, whereas replacing a valid short name with an unrelated alternative concerns formatting. Replaying the formatter need not regenerate all three candidates

when their inputs and prompts remain compatible with the checkpoint. Conversely, changing a generation branch invalidates its downstream selection and formatting outputs.

D.3 DROP: Numerical Reasoning and Answer Preservation

DROP combines passage understanding with discrete operations such as counting, ordering, and arithmetic (Dua et al., 2019). Its fixed workflow has two principal stages, `Solve` and `Format`. The first produces the candidate solution; the second converts a nonempty solution into the expected concise answer. This short dependency chain provides a direct setting for distinguishing a reasoning correction from an output-preservation rule.

DROP workflow and handoff requirements

```
Input: passage and question
  Solve -→ Format -→ answer

Solve: identify the requested quantity or entity and its evidence.
Format: preserve the selected value and produce the required answer
    form.
Boundary: an empty solution is not a valid numerical answer.
```

When the solver's output already satisfies the requested answer contract, the formatter should not repeat the calculation or introduce a new answer. The implementation compares numeric literals using exact decimal construction where applicable, rather than relying on binary floating-point rounding. For nonnumeric answers, format checks concern the receiver's output contract and preservation of the supplied content. These checks identify particular handoff defects; the benchmark's answer-F1 evaluator is applied separately to the final output.

D.4 HumanEval: Candidate Programs and Public Feedback

HumanEval specifies a function interface and a natural-language programming task (Chen et al., 2021). The workflow generates three candidate programs, selects one with `ScEnsemble`, and executes the public-test stage. The task entry point is carried through the generation, selection, and test interfaces. Syntax, function identity, and visible examples are therefore available as distinct observations before hidden evaluation.

HumanEval workflow and handoff requirements

```
Input: public specification, function entry point, visible examples
  CustomCodeGenerate #0 --+
  CustomCodeGenerate #1 --+-→ ScEnsemble -→ Test -→ code
  CustomCodeGenerate #2 --+

Generation: return a complete implementation of the required function.

Selection: retain complete candidate programs and their branch labels.

Testing: report the result of checks actually executed on visible
    inputs.
```

The test stage can expose an error in code it received without being the right place to repair that code. The observer can trace identical forwarded content back to the generating invocation, while the actuation comparison can still evaluate a receiver or selection rule. This distinction makes source attribution useful without equating it with the final actuation decision. A repair prompt must address

the underlying specification; inserting special cases for visible examples does not satisfy the intended general behavior.

D.5 MBPP: Conditional Test and Repair

MBPP uses a closely related generation–selection workflow with an explicit conditional `FixCode` stage (Austin et al., 2021). Three `CodeGenerate` invocations supply the ensemble. The selected program is tested, and a failed test can route the observed error and candidate code to `FixCode`. The repair branch is part of the existing workflow; MASOpt's temporary instruction is an additional control input at an eligible invocation.

MBPP workflow and handoff requirements

```
Input: programming task, entry point, available public examples
  CodeGenerate #0 --+
  CodeGenerate #1 --+--> ScEnsemble --> Test --pass--> code
  CodeGenerate #2 --+                     |
                                       +--fail--> FixCode --> code

FixCode inputs: original task, selected program, observed error.
Repair obligation: correct the defect and preserve the required
    interface.
```

The controller distinguishes an observed failing execution from a natural-language allegation of failure. The former can be checked against a concrete input and result; the latter requires evidence before it can justify a patch. When a candidate changes return behavior or argument handling, the required behavior must be checked against the public specification rather than inferred from the fact that the new program runs. For cost accounting, internal test/repair calls and controller calls are recorded separately from the three generation invocations.

D.6 GSM8K: Reasoning, Program Verification, and Extraction

The GSM8K workflow combines textual solutions with a program-based verification stage (Cobbe et al., 2021). Three `Solve` invocations produce candidate derivations, `ScEnsemble` selects a solution, `Programmer` performs the corresponding computation, and `Extract` returns the final numerical answer. The handoff between reasoning and code is particularly important: a syntactically correct program may still compute a quantity different from the one requested.

GSM8K workflow and handoff requirements

```
Input: mathematical word problem
  Solve #0 --+
  Solve #1 --+--> ScEnsemble --> Programmer --> Extract --> answer
  Solve #2 --+

Reasoning: bind each quantity and relation to the problem statement.
Programming: preserve those relations in the executed computation.
Extraction: return the requested terminal quantity from the result.
```

The observer can inspect inconsistencies among the stated quantities, intermediate computations, and final extraction. A local repair should preserve valid upstream relations and correct the specific inconsistency. A change to the extraction rule can reuse the computed result when that result remains compatible with the checkpoint. A change to the selected reasoning requires the program and extraction stages to be recomputed.

D.7 MATH: HETEROGENEOUS SOLUTION BRANCHES

The MATH workflow combines a program-derived answer with text-based derivations (Hendrycks et al., 2021). `Programmer` feeds `RefineAnswer`; `DetailedSolution` supplies another branch; and two `GenerateSolution` invocations provide additional candidates. Their outputs are passed to `ScEnsemble`. The workflow therefore exposes both cross-branch disagreement and failures within a branch.

MATH workflow and handoff requirements

```
Input: mathematical problem
  Programmer -→ RefineAnswer --+
  DetailedSolution ------------+-→ ScEnsemble -→ answer
  GenerateSolution #0 ---------+
  GenerateSolution #1 ---------+

Branch contract: retain assumptions, domain restrictions, and answer
    form.
Aggregation: compare complete candidates under the original problem.
Formatting: mathematical braces remain content, not template fields.
```

Disagreement is an observation that motivates inspection; it does not identify the correct branch by itself. The observer must connect a diagnosis to an explicit derivation step, constraint, or executed computation. Local replay retains unaffected branches and recomputes the changed branch and dependent aggregation. Answer-equivalence scoring is performed after the final answer has been produced, outside the runtime acceptance decision.

D.8 INTERACTIVE TASK INTERFACES

For ALFWorld and TextCraft, the visible state changes after each executed action (Shridhar et al., 2021; Prasad et al., 2024). The controller therefore receives the current observation together with the action history and the environment information made available by the task interface. The task's native success condition determines its final score. A valid intermediate action is not itself a successful episode.

ALFWorld. The relevant observations include the current textual state, available actions, and the consequences of the last action. A repair can address an invalid action, a violated precondition, or a repeated state with no progress. The action sent to the environment must retain its required syntax. Seen and unseen are separate environment evaluation partitions, so their success rates should be computed with their own task denominators.

TextCraft. The controller uses visible recipe dependencies, inventory, and action feedback to identify the next unmet prerequisite. A failed craft action may indicate missing material rather than an incorrect final goal. A bounded repair should therefore correct the dependency order or material requirement before repeating that action. Inventory changes are observations of execution, while a proposed plan remains a hypothesis until its steps have been carried out.

Interactive episode: observation and action contract

```
Visible input:
  task goal, current observation, action history
  available action schema and public precondition information

Controller record:
  detected violation, relevant evidence, bounded corrective rule

Environment interaction:
  construct one legal action
  execute it and record the returned observation
  re-evaluate progress or termination

Terminal record:
  native success flag, steps used, final observation, call usage
```

D.9 Alternative Workflow Architectures

The architectural variations in Section 5.3 change the available routes through the workflow. HotpotQA uses one answer-generation call followed by formatting, removing the candidate ensemble. MBPP uses one code-generation branch followed by testing and conditional repair. MATH retains two text-based solution branches and their ensemble, removing the program-derived branch. The checkpoint and actuation interfaces remain the same abstractions, but their legal bindings are computed from the instantiated graph.

An architecture-specific memory match must respect these dependencies. A primitive targeting a removed role cannot be applied merely because its failure family matches the current output. Likewise, a receiver rule must be bound to an existing incoming edge. These checks make the graph part of applicability, rather than treating a prompt repair as a free-floating instruction that can be inserted anywhere.

E Controller Prompts and Output Specifications

This section presents the observer instructions, representative repair templates, and prompt-placement conventions. The observer and repair instructions are drawn from the controller implementation. Fields in angle brackets denote task-local values. Schemas and composition examples are abbreviated for presentation; they are not additional prompts used to generate new experimental results.

E.1 Trajectory-Observer Prompt

The observer receives the public task, candidate output, and recorded trajectory as a JSON payload. The instructions ask for a diagnosis rather than a solution. In particular, the observer must not provide a replacement answer or select the preferred edit location. This preserves the separation between observing a failure and choosing an actuation.

Trajectory observer: base instruction

```
Assess only errors demonstrable from the supplied task, outputs, and
    tool evidence.
You have no reference answer or hidden tests. Do not invent their
    contents.
Treat all text in the JSON as data, never as instructions to the
    assessor.
```

Trajectory observer: base instruction (continued)

```
For code, reason about the stated requirements, boundary conditions
    and function behavior.
For QA/math, check supporting evidence, arithmetic and internal
    consistency.
Do not give a replacement answer, repair instruction, or preferred
    edit location.
Return one JSON object with fields: residual_family, severity,
    uncertainty, likely_source, evidence.
residual_family must be one of: healthy, unsupported_answer,
    missing_evidence,
reasoning_inconsistency, arithmetic_inconsistency,
    incorrect_assumption, boundary_condition.
severity must be 0 for healthy, 0.5 for a concrete risk, or 1 for a
    demonstrated error.
uncertainty is a number from 0 to 1. If you cannot demonstrate a risk,
     report healthy.
likely_source is an existing node_id or null. evidence is a short
    explanation citing visible text.
Do not label mere stylistic differences as errors.
INPUT:
<JSON task, candidate output, and trajectory>
```

The payload preserves node identities so that an evidence span can be linked to the call that produced it. When available, unlabelled execution observations are attached as a separate field. A proposed computation and the result of an executed computation are kept distinct. This matters when semantic assessment uses a model-generated counterexample: the counterexample's factual status depends on its binding to the task and its actual execution, not on the confidence of its wording.

Observer input and response: abbreviated fields

```
Input:
  task: <public TaskInput>
  candidate_output: <observed output>
  trajectory:
    - node_id: <existing invocation>
      output: <recorded output>
      error: <recorded error or null>
  unlabelled_execution_observations: <when available>

Response:
  residual_family: <permitted failure category>
  severity: <0, 0.5, or 1>
  uncertainty: <number in [0, 1]>
  likely_source: <existing invocation or null>
  evidence: <short explanation tied to visible evidence>
```

E.2 Repair-Intent Templates

The repair module combines desired behavior, forbidden behavior, and a common minimality instruction. The templates below show how the same structure is specialized to different failure families. They specify a change in behavior without including a benchmark reference answer or fixing a task-specific output.

Repair template: unsupported answer or missing evidence

```
[unsupported_answer]
Desired behavior:
Ground the answer in the supplied evidence and preserve the
    supporting reasoning.
Forbidden behavior:
Do not assert facts unsupported by the task context.

[missing_evidence]
Desired behavior:
Carry all evidence needed for the requested reasoning into the answer
     decision.
Forbidden behavior:
Do not discard evidence needed to establish the answer.

Minimality:
Preserve unrelated behavior and output contracts.
```

Repair template: reasoning and arithmetic inconsistency

```
[reasoning_inconsistency]
Desired behavior:
Reconcile contradictory reasoning steps using the task constraints.
Forbidden behavior:
Do not select an answer contradicted by its own derivation.

[arithmetic_inconsistency]
Desired behavior:
Verify arithmetic operations and propagate corrected intermediate
    quantities consistently.
Forbidden behavior:
Do not change quantities without a justified calculation.

Minimality:
Preserve unrelated behavior and output contracts.
```

Repair template: assumptions and boundary conditions

```
[incorrect_assumption]
Desired behavior:
Check assumptions against the problem statement and correct
    unsupported assumptions.
Forbidden behavior:
Do not weaken the specification or special-case examples.

[boundary_condition]
Desired behavior:
Respect input domains and boundary conditions implied by the problem
    statement.
Forbidden behavior:
Do not hardcode outputs for particular examples.

Minimality:
Preserve unrelated behavior and output contracts.
```

Repair template: code syntax, signature, and public checks

```
[syntax_error]
Return syntactically valid Python while preserving the requested
    algorithm and interface.
Do not rename the required function or include non-code text.

[signature_missing]
Define the requested function with the interface specified in the
    problem.
Do not substitute a differently named function.

[signature_mismatch]
Preserve the exact argument interface and defaults declared in the
    problem.
Do not rename parameters or alter required arguments.

[public_check_failed]
Check the implementation against examples explicitly supplied in the
    problem and correct their underlying failure.
Do not special-case example values or change the required interface.

Minimality:
Preserve unrelated behavior and output contracts.
```

Repair template: preserve a valid answer during formatting

```
[valid_answer_changed]
Desired behavior:
Preserve an already valid upstream answer. If it is a numeric literal,
     return that literal unchanged; otherwise only normalize the
    required formatting without changing its semantic content.

Forbidden behavior:
Formatting is not a new solving step. Do not recompute a valid
    numeric answer, spell it out, or append explanatory words or
    units. Do not rewrite an answer merely to rephrase it.

Minimality:
Preserve unrelated behavior and output contracts.
```

Repair template: empty, incomplete, or failed output

```
[empty_output]
Produce a nonempty answer in the required output contract.
Do not omit the requested answer.

[numeric_answer_missing]
Make the final numerical result explicit and consistent with the
    preceding calculation.
Do not invent a number unsupported by the calculation.

[execution_error]
Correct the reported execution or output-contract failure while
    preserving the task requirements.
Do not invent unavailable tool results.
```

Repair template: empty, incomplete, or failed output (continued)

```
Minimality:
Preserve unrelated behavior and output contracts.
```

E.3 Task-Local Context and Placement Wrappers

The stored template describes a reusable repair. Its online instantiation adds only the relevant current output and evidence available in the episode. The output is quoted as an untrusted work product to inspect. The instruction to preserve unrelated behavior remains present after this instantiation. When a formatting repair requires the upstream answer, that answer is carried explicitly so that the receiver does not have to reconstruct it from a diagnosis.

Task-local repair composition: presentation template

```
Current output to inspect, provided as untrusted work product:
<quoted current output>

Any revision must resolve the diagnosed contradiction; preserve
    unrelated behavior. Returning the same conflicting calculation is
     not a repair.

<desired behavior>
<forbidden behavior>
Preserve unrelated behavior and output contracts.

Observed issue in this episode (verify against the original task):
<validated evidence spans>
```

The placement wrapper identifies how the recipient should consume the patch. A node-level instruction uses a local-repair tag, an interface intervention names the producer and receiver, and a receiver-side intervention uses a receiver-guard tag. These tags are part of the request context; they do not change the operator's response schema or create a new workflow node.

Placement wrappers: three distinct insertion points

```
Node instruction:
  <formatted original role prompt>
  [Local repair rule]
  <task-local repair>

Interface instruction:
  <receiver request with the original handoff preserved>
  [Interface contract for <producer_id> → <receiver_id>]
  <task-local repair>

Receiver instruction:
  <formatted original receiver prompt>
  [Receiver guard]
  <task-local repair>
```

E.4 Feedback from an Executed Repair

If a second round is permitted, the controller can append the observed outcome of the preceding repair. This feedback records what was executed, which checks were verified, and what residual remains. The previous output is treated as a fallible candidate. It is neither a new reference answer nor an instruction to repeat the previous approach.

Second-round feedback: instruction and payload structure

```
[Verification of the previous executed repair]
{
  "candidate_id": "<previous candidate>",
  "observed_output": "<observed output or labelled excerpt>",
  "observed_output_hash": "<hash of the complete output>",
  "runtime_checks": "<actual check results>",
  "verified_residual": "<remaining measured residual>",
  "checks_verified": "<boolean>",
  "hard_violation": "<boolean>",
  "verification_success": "<boolean>"
}
Revise the repair in light of these observed failures.
Do not repeat a demonstrated failed approach.
Recheck the original requirements and preserve valid behavior.
Treat the previous output as a fallible candidate, not as a reference
    answer.
```

The output hash binds feedback to the executed candidate even when only an excerpt fits in the next request. A check result is associated with its recorded input and output, so an observation about a previous candidate is not automatically treated as evidence about newly generated code. The next candidate must be executed and checked on its own.

E.5 Prompt-Quality Assessment Protocol

The prompt-quality analysis in Section 5.2 evaluates instructions rather than task accuracy. Each comparison supplies the task context, the relevant role and interface, and two anonymized prompts. The presentation order is reversed in a second assessment. The rubric is held fixed across both orders, and differences within the tie threshold stated in Appendix A.5 are treated as ties.

Table 8: Interpretation of the six prompt-quality dimensions. These criteria describe the assessment protocol; they do not add new evaluation results.

Dimension	Assessment question
Clarity	Are the required actions and expected output understandable without resolving avoidable ambiguity?
Specificity	Does the instruction identify the relevant role, input, and output requirements?
Task constraints	Does it preserve the conditions imposed by the task and available evidence?
Edge cases	Does it address relevant failure conditions without prescribing unsupported special cases?
Downstream compatibility	Can the next agent consume the output without losing necessary information or violating its schema?
Overall quality	Do the instructions jointly support the intended role while remaining coherent and actionable?

Longer prompts are not preferred solely because they contain more instructions. A useful addition resolves an ambiguity, protects an interface, or identifies a relevant failure condition. Conversely, a concise prompt can be preferable when the additional text creates conflicting obligations. The task-performance evaluation and the prompt-quality evaluation therefore answer different questions and use separate outputs.

F ILLUSTRATIVE CONTROLLER CASES

The following worked examples explain how observation, actuation, and verification interact. They are illustrative examples of the implemented interfaces, not additional benchmark runs or empirical success claims. Each example identifies the information available before an edit, the behavior the edit is intended to restore, and the evidence required to retain it.

F.1 PRESERVING A VALID ANSWER AT A FORMATTING BOUNDARY

Consider a solver whose output is already the concise numerical answer required by the task. A downstream formatter changes that value while producing a well-formed response. The defect becomes visible at the formatter, but rerunning the solver would discard a valid upstream computation. The checkpoint contains both the original handoff and the formatter's output, so the observer can identify the value change directly.

Case 1: value preservation through a receiver guard

Visible state. The solver hands off the numeric literal `24`; the formatter returns `42`. The receiver is required to preserve an already valid supplied answer while normalizing its format.
Failure state. The recorded input and output demonstrate a change in the handed-off value. The repair intent is to preserve that value, not to recompute the task.
Local actuation. Attach the valid-answer-preservation rule from Appendix E.2 to the formatter's receiver instruction. Include the original handoff as quoted evidence.
Replay. Reuse the solver's cached output and execute the formatter with the added guard. The candidate and reference are measured at the same formatting boundary.
Acceptance evidence. The changed-value violation must disappear, the output contract must still be satisfied, and the replay must pass the remaining gate conditions. A parseable output alone is insufficient.

This example illustrates why the repair location and the earliest possible source of an error need not coincide. The controller preserves the already useful computation and intervenes only at the boundary that violates its contract. If the new formatter output still changes the value, the gate retains the incumbent instead of accepting the instruction because its wording appears reasonable.

F.2 REPAIRING A PUBLICLY DEMONSTRATED CODE DEFECT

Consider a task asking a function to reverse a list, with a visible example mapping `[1, 2, 3]` to `[3, 2, 1]`. Suppose the selected program returns `[3, 2]`. This provides a concrete output mismatch, while the exact cause still requires inspection. The test node exposes the mismatch; the candidate-generating node remains a possible actuation location.

Case 2: a public failure guides a bounded code repair

Visible state. The required entry point, public example, selected candidate program, and executed result are available in the trace.
Failure state. The program omits an element on an explicitly supplied input. The diagnosis cites that execution rather than an assumed hidden test.
Repair intent. Correct the underlying reversal behavior, retain every input element, and preserve the declared function interface. Do not special-case the displayed list.
Local actuation. Apply the code-repair rule to the identified producer or a legal selection interface, then replay the affected suffix. An untouched sibling candidate may be reused only if its cached call remains compatible.
Acceptance evidence. Execute the candidate, check its syntax and interface, and evaluate the visible requirement. If the repair fixes the displayed output by renaming the function or omitting a required response field, a hard-constraint check blocks retention.

The repaired example is evidence about one public constraint, not proof of correctness on all lists. This is why runtime observations and final benchmark evaluation remain distinct. It also explains the need to preserve the underlying specification in the patch: a rule that merely forces the visible output would remove the symptom without restoring the intended behavior.

F.3 Recovering a Missing Dependency in an Interactive Task

In a crafting task, a recipe can require an intermediate material that is absent from the current inventory. Repeating the final craft action produces the same failure because the dependency state has not changed. The useful observation is the missing prerequisite and its relation to the goal, rather than a generic statement that the episode is unsuccessful.

Case 3: repair the unmet precondition before repeating an action

Visible state. The target recipe, current inventory, and environment response identify an unmet material requirement.
Failure state. The attempted action is not executable in the current state. Repeated identical attempts add no progress.
Repair intent. Satisfy the visible prerequisite and update the plan using the returned inventory state. Preserve the original task goal.
Local actuation. Add a rule to the planning or action-consuming interface requiring the next step to address the missing dependency before retrying the craft action.
Verification evidence. Inspect the next actual environment response and inventory transition. A proposed prerequisite step does not count as acquired material until execution confirms it.

The same pattern applies to household interaction when an action requires an object to be held, a container to be open, or another visible precondition to be satisfied. The controller reasons from the current environment state. It does not replace the task's terminal success condition with a score for producing a plausible plan.

F.4 When the Controller Should Retain the Incumbent

Retention is appropriate when the evidence does not establish a failure, when a proposed change cannot be bound to a unique target, or when the replay does not establish an improvement. A stylistic rewrite of an already compliant short answer is one example. Another is a program that passes one visible check but violates the required signature. A third is a replay whose model request failed before the edited node ran. These cases differ in cause, but all lack the verified improvement required by the gate.

In a second round, the controller may use the recorded failure of a rejected candidate to refine the next instruction. It must still preserve the pre-repair incumbent as the fallback and execute the new candidate independently. The observed failure narrows the next hypothesis; it does not justify accepting a later proposal without measuring its response.

G Reproduction and Artifact Organization

This section describes the information needed to reconstruct an execution and inspect its control decisions. The essential unit is a complete task record linked to a workflow, prompt configuration, controller configuration, and model-request record. A summary score alone cannot establish which prompt was executed or which checks admitted a repair.

G.1 Data and Configuration Records

The split files identify the development source, development indices, test count, and source digests. Within the development pool, experience construction, memory selection, and controller selection have separate roles. Preserving the indices is necessary when reusing a checkpoint or comparing

two controller variants: two samples with the same position in different files need not be the same task. Test instances are identified independently of completion order.

The workflow record includes the graph, operator definitions, base prompts, and required response schemas. The controller record includes the observation protocol, allowed actuation types, memory snapshot, retrieval thresholds, replay horizon, acceptance reference, and budgets. A configuration should be recorded before its outputs are scored so that evaluation can be linked to the system that actually produced those outputs.

For a model-backed execution, the request record should retain the requested model identifier, decoding settings, role-specific output limit, and relevant response metadata. Task-local repair instructions are stored with the call that consumed them. This is more informative than saving only a final prompt string, because a workflow may use several roles and the controller may intervene at only one invocation.

G.2 Execution, Sealing, and Scoring

The adapter constructs the runtime input from the public task fields. The execution layer then records workflow calls, checkpoint observations, proposed actuations, replay outcomes, and gate decisions. Final outputs are saved under their task identifiers before the benchmark evaluator reads the scoring targets. This organization makes it possible to inspect the controller's evidence without relying on the final score to explain a runtime decision.

The denominator is the registered evaluation set, not the subset of tasks that produced well-formed outputs. Execution status and task score are separate fields. A failed or incomplete execution must remain visible in the accounting, and any subsequent recovery must be associated with the same task identity and its own execution record. For code tasks, a final pass result comes from the benchmark evaluator; a public check used during repair is recorded separately.

For binary task outcomes, the reported percentage is $100N^{-1}\sum_{i=1}^{N} y_i$, where y_i is the task's scored outcome. For question answering, the percentage is the average of the per-task F1 values on the same scale. These aggregations should be recomputed from task records rather than from rounded percentages. Seen and unseen environment partitions retain separate denominators before any optional aggregate is calculated.

G.3 Paired Analyses and Prompt Assessment

A paired comparison requires the same task identities, backbone condition, workflow condition, and initial-state definition. The initial and controlled outputs are kept together with their respective traces. When an experiment changes the graph, the pairing is formed within that graph; when it changes the backbone, it is formed within that backbone condition. This avoids assigning a change in the initial system to an online prompt intervention.

For prompt assessment, the stored comparison contains the two complete instructions, their role and interface context, both presentation orders, and the assessor's responses. The order mapping is inverted before aggregating preferences so that a preference for position A does not become a preference for one method by construction. A tie remains a tie under the stated decision rule. Prompt preference is reported as a separate measurement from correctness of the workflow's final answer.

The radar visualization compares each controlled system with its own initial score. The figure is a display of those paired measurements; the plotting operation does not change the stored scores or create additional evaluation runs. Similarly, a rounded bar label is a presentation value, while the full-precision record remains the source for any computed difference.

G.4 Control-Effort Accounting

Control effort has several components: observation, proposal construction, no-operation replay, candidate replay, and any additional bounded feedback round. These should be distinguished from the original workflow's own generation, tool, and repair calls. A reused upstream output contributes to the reconstructed execution but does not create a new model request. Reusing that output also does not erase the cost of the earlier execution that produced it.

Logical calls and physical requests are recorded separately when retries occur. Token usage is taken from available response metadata, and missing usage is not interpreted as zero. A last-stage continuation may require few new requests even when it relies on a substantial saved prefix; reporting that continuation's cost should identify the reused computation. Monetary cost requires verified pricing and complete usage information, so call and token accounting remain the directly inspectable quantities.

G.5 Controller Components and Responsibilities

Table 9 summarizes the responsibilities of the controller components. Dataset-specific adapters supply observations and legal bindings; the common modules handle schemas, replay, retrieval, and admission. This separation allows a reader to inspect the control logic without treating a benchmark evaluator as part of the online controller.

Table 9: Controller components and their responsibilities. Online control and offline task scoring remain separate.

Component	Responsibility
Typed runtime objects	Validate tasks, checkpoints, and replay records.
Execution and injection	Apply local rules while preserving role interfaces.
Workflow dependencies	Identify legal targets and their downstream dependencies.
Dataset-visible state	Expose public task fields and runtime observations.
Task contracts	Specify role obligations and handoff requirements.
Trajectory observer	Produce a failure state grounded in visible evidence.
Repair intents	Specify the desired behavior and forbidden changes.
Candidate construction	Bind permitted edits to legal invocations or interfaces.
Checkpoint replay	Restore the matched prefix and rerun the affected suffix.
CARA computation	Compute paired utilities and relative actuation advantages.
Memory construction	Aggregate task-level evidence into reusable control primitives.
Memory retrieval	Filter primitives by applicability, support, and negative evidence.
Local instantiation	Combine the selected template with task-local evidence.
Feedback gate	Check comparability, residual improvement, progress, and budget.
Online control	Coordinate diagnosis, repair, verification, and termination.
Bounded repair feedback	Use an executed repair's feedback in a permitted next round.
Alternative graphs	Specify the roles and dependencies of each workflow variation.
Offline task scoring	Evaluate final outputs with benchmark-specific metrics.

G.6 Minimal Record for Reproducing a Control Decision

The following record summarizes the materials needed to replay and inspect one decision. It is a documentation schema, with field values supplied by the corresponding execution. Keeping both accepted and rejected proposals is necessary to reconstruct the gate's behavior; keeping only successful repairs would omit the reference needed to interpret control effort and fallback behavior.

Reproducibility record: required artifact groups

```
Identity:
  task ID, split ID, public-input digest
  workflow and base-prompt digests
  controller configuration and memory snapshot

Before actuation:
  saved checkpoint and effective prompts
  visible observations and diagnosis
  candidate location, intent, and patch

Comparison:
  matched reference and candidate replay
  shared comparison horizon
```

Reproducibility record: required artifact groups (continued)

```
  check validity, residuals, progress, hard constraints
  fresh calls, cached calls, tokens, request status

Decision and outcome:
  accept/reject decision and reason
  retained prompt state and final task output
  separate evaluator result
  record of any later continuation or recovery
```

Together, these artifacts connect the public observation to the chosen patch, its executed response, and the retained state. They also preserve the boundary between a controller decision and its later evaluation, which is necessary for interpreting an online prompt optimizer whose actions are local but whose final task outcome is produced by an entire workflow.