CATO: CONTRASTIVE ATTRIBUTION FOR TARGETED OPTIMIZATION OF MULTI-COMPONENT LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-component LLM agents expose several editable components, such as system prompts, tool descriptions, and experience banks. Yet when an agent fails, its binary task outcome does not indicate which component to edit. Single-component optimizers cannot reach defects elsewhere, and unguided joint search can spend its budget on unrelated components. We formulate this component-selection problem as structural attribution, which assigns a failure to the agent's editable components rather than to trajectory steps. A component is an update target of a failure when a repair of the failure needs to edit it. We propose CATO, which contrasts failures with successful runs of the same tasks under a moderately different configuration, scores how strongly this evidence implicates each component, and writes a textual gradient describing the fix. CATO then proposes more single-component edits for higher-scoring components and keeps the best configurations on a held-out validation set, so multi-component repairs build up one edit at a time. In controlled fault injection, CATO identifies the faulty component in 95.7% of cases, 42.9 points more often than trajectory-only diagnosis, and its edits repair 89.8% of these failures while preserving 85.3% of solved tasks. Across AppWorld, WebArena, and BBH with three backbones, CATO outperforms all six baselines in every setting, leads the strongest by up to 5.96 points, and uses 49% fewer training tokens than editing every component on every failure.

1 INTRODUCTION

Autonomous LLM agents increasingly accomplish complex goals by planning, invoking tools, and reasoning across multi-step environments (Yao et al., 2022; Schick et al., 2023). Research on such agents has shifted from designing a single prompt to assembling systems from multiple editable components, typically including prompts (Khattab et al., 2024; Yao et al., 2022; Pryzant et al., 2023), tool descriptions (Patil et al., 2024; Qu et al., 2025; Wu et al., 2024), and experience banks (Shinn et al., 2023; Wang et al., 2023; Zhao et al., 2024). These components are text, so an optimizer can improve the agent by editing them while the model weights stay fixed. More editable components give more room to improve, but only if the optimizer knows which component to edit and how. Yet each task only succeeds or fails, and this binary outcome reveals neither which components are worth updating nor which edits to try. This raises a central question: *how should an optimizer decide which components to update from binary outcomes alone?*

Existing optimizers follow two approaches, but neither decides which component a failure calls for (Figure 1). The first approach optimizes a single fixed component, such as prompts (Yang et al., 2024; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2025), tool descriptions (Qu et al., 2025), or memory and experience banks (Cai et al., 2025; Zhang et al., 2026c; Ni et al., 2026). This helps only when the failure stems from that component. Otherwise, even a strong optimizer cannot reach the real defect (Figure 1a). The second approach optimizes several components of a compound AI system (Lee et al., 2025; Zhuge et al., 2024; Wang et al., 2025). For instance, GEPA selects candidates along a Pareto frontier and updates modules in round-robin order (Agrawal et al., 2026), while OPTIMAS chooses components based on expected reward gaps (Wu et al., 2026). These methods search a larger space, but they choose the component to update without examining why a given task failed. They can therefore spend candidate edits on components unrelated to the failure (Figure 1b).

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.

The missing piece is evidence about which component to edit. A single failure trace shows where execution broke down, but often not which component caused it. A successful run of the same task under another configuration supplies this evidence (Miller, 2019). The success shows where the failed execution diverged, and this divergence can be read against the known differences between the two configurations. We formulate the choice of which component to edit as *structural attribution*, which assigns a failure to the components that a repair would need to change. Unlike execution tracing or multi-agent debugging (Chen et al., 2024a; Cemri et al., 2025; Zhang et al., 2026a; 2025b), which explain what went wrong in a trajectory, structural attribution selects the component to edit next. Structural attribution is multi-label, because a failure may be repairable through more than one component.

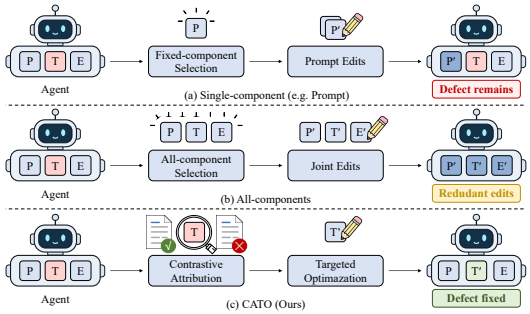


Figure 1: Component selection for a tool-description defect: (a) prompt-only, (b) all-component, and (c) attribution-guided optimization. P, T, and E denote the system prompt, tool descriptions, and experience bank. Only (c) targets the defective component exclusively.

Building on this formulation, we propose **CATO** (Contrastive Attribution for Targeted Optimization of Multi-Component LLM Agents). CATO keeps a small population of agent configurations, each produced by earlier edits, and alternates two stages over several rounds. In *contrastive attribution*, CATO selects a center configuration by validation accuracy and pairs it with a neighbor, a population member that differs from the center by a moderate amount. A nearly identical neighbor tends to shift blame onto its few differences, even when the fault lies in a shared component. A very different one hides the relevant change among unrelated ones. Both configurations run the same training tasks, and each task that one solves and the other fails forms a contrast pair. For each pair, an LLM compares the two trajectories and the configuration differences, scores the evidence for editing each component, and writes a textual gradient (Yuksekgonul et al., 2025) that describes the fix. In *targeted optimization*, CATO consolidates the textual gradients for each component and proposes more candidate edits for components with higher total scores. Each candidate changes only one component. The most accurate configurations on a held-out validation set form the next population, so multi-component repairs accumulate over rounds without a search over joint edits.

In experiments that inject a fault into one component, CATO identifies that component in 95.7% of cases, 42.9 percentage points more often than trajectory-only diagnosis. Its edits repair 89.8% of these failures and keep 85.3% of solved tasks, close to the 95.6% and 93.8% of always editing the faulty component. Across three backbones, CATO outperforms all six baselines on AppWorld (Trivedi et al., 2024), WebArena (Zhou et al., 2024; El hattami et al., 2025), and BBH (Suzgun et al., 2023), and leads the strongest competing optimizer by up to 5.96 points. On the AppWorld Test-Challenge split, whose applications are unseen during optimization, it leads by 2.15 to 5.52 points in scenario goal completion. It also uses 49% fewer training tokens than editing every component on every failure.

We make three contributions. (1) We formulate the choice of which component to update in multi-component LLM agents as multi-label *structural attribution*, which serves the next edit rather than explaining the failure after the fact. (2) We introduce CATO, a closed-loop optimizer that scores components by contrasting same-task executions of neighboring configurations and uses these scores to allocate single-component edits. (3) On controlled fault injection and three agent benchmarks, CATO identifies faulty components accurately and outperforms all six baselines on every benchmark and backbone. Ablations show that restricting each edit to one component matters most on applications unseen during optimization.

2 Related Work

Single-Component and Textual Gradient Optimization. Most agent optimizers edit one fixed component. Prompt optimizers refine instructions with textual gradients (Pryzant et al., 2023; Yuk-

sekgonul et al., 2025; Chen et al., 2026), evolutionary search (Guo et al., 2024), LLM-based meta-optimizers (Zhou et al., 2022; Yang et al., 2024; Chen et al., 2024b), or structured program search (Khattab et al., 2024; Opsahl-Ong et al., 2024). Tool-use work trains models to call APIs (Schick et al., 2023; Patil et al., 2024; Qin et al., 2024) or refines tool descriptions and tool-use prompts from feedback (Qu et al., 2025; Wu et al., 2024). Memory-based agents accumulate trajectories into experience or skill libraries (Wang et al., 2023; Zhao et al., 2024; Cai et al., 2025; Zhang et al., 2026c; Ni et al., 2026). These methods fix the edited component in advance and therefore cannot repair a defect elsewhere, whereas CATO chooses the component for each failure.

Multi-Component and Compound Agent Search. Compound-system optimizers extend search to multiple editable components (Lee et al., 2025; Wu et al., 2023). Closest to our setting, GEPA (Agrawal et al., 2026) reflects on execution traces to mutate module prompts and keeps a Pareto frontier of candidates, similar to CATO's validated population. OPTIMAS (Wu et al., 2026) updates heterogeneous components with local rewards aligned to the global objective. To choose which component to edit, GEPA cycles through modules in round-robin order and OPTIMAS follows expected reward gaps. Neither uses the evidence in a specific failure, which CATO obtains from same-task contrasts and also uses to divide the candidate budget. Another line searches agent communication graphs, multi-agent topologies, or automated code workflows (Hu et al., 2025; Zhang et al., 2025a; Zhuge et al., 2024; Zhang et al., 2024; Yuan et al., 2025; Robeyns et al., 2025; Wang et al., 2025; Zhang et al., 2026b). These methods explore unconstrained structural topologies or program spaces, whereas CATO focuses on failure-level structural attribution across a fixed modular surface, making the two directions complementary.

Credit Assignment, Fault Localization, and Causal Diagnosis. The credit assignment problem dates back to foundational reinforcement learning (Sutton & Barto, 2018). Self-reflection and self-debugging methods use execution feedback to revise the current attempt (Shinn et al., 2023; Madaan et al., 2023; Chen et al., 2024a), whereas CATO edits a persistent configuration offline and selects edits on a validation set. Multi-agent diagnostics build failure taxonomies or locate the faulty step and agent (Cemri et al., 2025; Zhang et al., 2026a; 2025b), and reinforcement-learning credit assignment decomposes trajectory rewards across time steps, sub-agents, or modular planners (Li et al., 2026b; Peng et al., 2026; Fang et al., 2026; Li et al., 2026a). These methods optimize internal policy weights or credit temporal trajectory steps, whereas CATO assigns credit to black-box editable configuration components and uses it to route the next textual edit. Delta debugging (Zeller & Hildebrandt, 2002) isolates failure causes by narrowing the difference between a passing and a failing run. CATO applies this idea to agent configurations but contrasts a neighbor at the median distance instead of searching for a minimal difference.

3 METHODOLOGY

We present **CATO** (Contrastive Attribution for Targeted Optimization), a closed-loop optimizer for multi-component LLM agents (Figure 2). Given a seed configuration, training tasks with binary outcomes, and a held-out validation set, CATO keeps a population $\mathcal{P}_i$ of configurations, alternates two stages, and returns the most accurate configuration on the validation set. **Stage 1 (Contrastive Attribution)** runs the most accurate configuration and a moderately different one on the same training tasks. For tasks where only one succeeds, an LLM scores each component and writes a textual gradient describing the fix. **Stage 2 (Targeted Optimization)** consolidates these gradients per component and generates more single-component candidates for higher-scoring components. The most accurate candidates and current members on the validation set form the next population.

3.1 PROBLEM FORMULATION

We represent an agent as an editable configuration $\theta = (\theta_d)_{d\in\mathcal{K}} \in \Theta$ over components $\mathcal{K} = \{p, t, e\}$ in a compound system (Lee et al., 2025). The system prompt θ_p governs high-level strategy, the tool description set θ_t specifies API contracts, and the experience bank θ_e provides few-shot execution memory. This formulation extends directly to additional modular components, such as procedural skill libraries (θ_s, Appendix D.5).

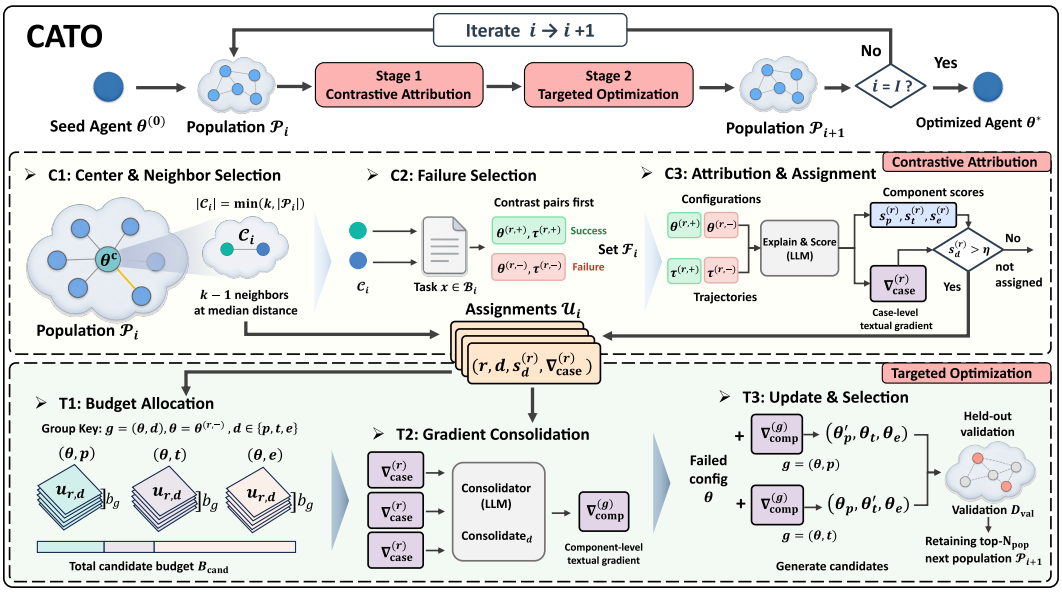


Figure 2: Overview of CATO. Contrastive attribution (C1–C3) selects a center and a neighbor, keeps failures with a same-task success, and scores each component. Targeted optimization (T1–T3) allocates candidates, consolidates textual gradients, and validates single-component edits.

Executing configuration θ on task instance x yields an execution record $r_{x,\theta} = (x, \theta, y_{x,\theta}, \tau_{x,\theta})$. Here, $y_{x,\theta} \in \{0, 1\}$ denotes binary task success. The trajectory $\tau_{x,\theta}$ records intermediate reasoning steps, tool calls, environment observations, and termination states. A failure ($y_{x,\theta} = 0$) shows that the task broke down but not which component is at fault. We therefore formulate the choice of which component to update as multi-label *structural attribution*. Analogous to structural credit assignment (Sutton & Barto, 2018), component $d \in \mathcal{K}$ is an update target of a failed record $r_{x,\theta}$ if some repair that makes x succeed needs an edit to θ_d. Because several components may qualify, the labels mark feasible points of intervention and need not identify a unique cause.

Let $\mathrm{Acc}(\theta; \mathcal{D})$ be the mean task success rate of configuration θ on dataset $\mathcal{D}$. The optimization objective is to find a configuration that maximizes accuracy on the held-out validation set:

$$\theta^\star \in \arg\max_{\theta \in \Theta} \mathrm{Acc}\big(\theta; \mathcal{D}_{\text{val}}\big). \tag{1}$$

Because Acc is non-differentiable and Θ is combinatorial, CATO searches over I rounds instead of optimizing directly. The population holds at most N_{pop} configurations ($|\mathcal{P}_i| \le N_{\text{pop}}$), and each round evaluates up to B_{cand} candidates. The training set $\mathcal{D}_{\text{train}}$ supplies the executions to contrast, and the validation set $\mathcal{D}_{\text{val}}$ selects candidates (Algorithm 1, Appendix A.1).

3.2 Contrastive Attribution

Following contrastive explanation (Miller, 2019), when two related configurations run the same task and only one succeeds, their behavior differs at some point, and CATO interprets that point in light of the known differences between the two configurations.

Center and Neighbor Selection (C1). In the first round, the population holds only the seed configuration $\theta^{(0)}$, so CATO diagnoses single failure trajectories (Appendix A.2). In later rounds, the center θ^{c} is the population member with the highest validation accuracy. If that member was the center in the previous round, the next-ranked member takes its place, since reusing a center would re-diagnose largely the same failures. The difference between two configurations is measured by the normalized structural distance

$$\delta(\theta, \theta') = \frac{1}{|\mathcal{K}|} \sum_{d \in \mathcal{K}} \delta_d(\theta_d, \theta'_d) \in [0, 1]. \tag{2}$$

Appendix A.1 defines the component distances δ_d. Nearest neighbors expose only a few differences, which can bias attribution toward changed components even when an unchanged component causes the failure. Farthest neighbors add many unrelated differences (Appendix D.2). CATO therefore selects as neighbors the population members whose distance to θ^c is closest to the median. The center and up to $k-1$ such neighbors form the contrast set $\mathcal{C}_i$, so $|\mathcal{C}_i| = \min(k, |\mathcal{P}_i|)$, where $k = 2$ by default (one neighbor). All members of $\mathcal{C}_i$ run on the same training batch $\mathcal{B}_i \subset \mathcal{D}_{\text{train}}$.

Failure Selection (C2). Tasks in $\mathcal{B}_i$ on which all members of $\mathcal{C}_i$ succeed are skipped. Tasks on which they disagree form contrast pairs, and the failure set $\mathcal{F}_i$ keeps the failed record of every contrast pair. For a failed record r, we write $\theta^{(r,-)}$ and $\tau^{(r,-)}$ for its configuration and trajectory, and $\theta^{(r,+)}$ and $\tau^{(r,+)}$ for the successful execution of the same task, which serves as a reference. CATO prefers contrast pairs because only they provide such a reference. If fewer than $N_{\min}$ tasks form contrast pairs, it also adds the center's failures on all-failure tasks, which have no successful reference (Appendix A.2).

Attribution and Assignment (C3). For each failed record $r \in \mathcal{F}_i$, attribution takes three steps. First, when a reference exists, a rule-based diff without model calls lists the configuration differences $\Delta\theta^{(r)}$ between $\theta^{(r,-)}$ and $\theta^{(r,+)}$. Second, an LLM explains where the failed execution went wrong from the trajectories alone. It sees neither the configurations nor any component labels, so its explanation does not presuppose which component to blame. Third, a second LLM call compares this explanation with the failed configuration $\theta^{(r,-)}$ and, if a reference exists, with $\Delta\theta^{(r)}$. It outputs a component score $s_d^{(r)} \in [0,1]$ for each $d \in \mathcal{K}$ and a case-level textual gradient $\nabla_{\text{case}}^{(r)}$, a short instruction that describes how to fix this failure (Appendices A.3 and F). Each score estimates whether d is an update target of r (Section 3.1). A component that did not change can still receive a high score, since behavioral evidence can implicate a component that both sides share. Scores are independent and need not sum to one. CATO assigns failure r to every component whose score exceeds the threshold η and collects these assignments in $\mathcal{U}_i$:

$$\mathcal{U}_i = \left\{ u_{r,d} = \left(r, d, s_d^{(r)}, \nabla_{\text{case}}^{(r)}\right) \;\middle|\; r \in \mathcal{F}_i,\ d \in \mathcal{K},\ s_d^{(r)} > \eta \right\}. \tag{3}$$

A failure with no score above η is not assigned, and if $\mathcal{U}_i = \emptyset$, the round generates no candidates.

3.3 Targeted Optimization

Editing a configuration from a single failure risks overfitting to that failure. Targeted optimization therefore consolidates evidence across related failures before editing.

Budget Allocation (T1). Each candidate edits one component of one failed configuration. CATO therefore groups the assignments by failed configuration and target component, $g = (\theta, d)$, and sums their scores into a total score a_g:

$$\mathcal{U}_g = \{u_{r,d} \in \mathcal{U}_i : \theta^{(r,-)} = \theta\}, \qquad a_g = \textstyle\sum_{u_{r,d} \in \mathcal{U}_g} s_d^{(r)}. \tag{4}$$

Each group receives $b_g = \lceil a_g \rceil$ candidates when the total demand fits within the round budget B_{cand}. Otherwise, candidates are allocated in proportion to a_g (Appendix A.4). The budget thus follows the amount and strength of failure-specific evidence, whereas GEPA follows a fixed module schedule and OPTIMAS a reward-gap estimate (Agrawal et al., 2026; Wu et al., 2026).

Gradient Consolidation (T2). For each group $g = (\theta, d)$ with $b_g > 0$, an LLM call Consolidate_d turns the case-level textual gradients in $\mathcal{U}_g$ into a component-level textual gradient $\nabla_{\text{comp}}^{(g)}$. This is one instruction for editing component d that applies across the grouped failures:

$$\nabla_{\text{comp}}^{(g)} = \text{Consolidate}_d\left(\theta_d,\ \{\nabla_{\text{case}}^{(r)} :\ u_{r,d} \in \mathcal{U}_g\}\right). \tag{5}$$

Consolidation keeps feedback that concerns component d, removes duplicate diagnoses, resolves conflicting ones, and drops details tied to individual tasks (Appendix F.3).

Update and Selection (T3). For each group $g = (\theta, d)$, an LLM editor generates b_g candidates θ' by rewriting θ_d according to $\nabla^{(g)}_{\text{comp}}$, so each candidate differs from θ only in component d. CATO evaluates the pool $\mathcal{Q}_i$ of all candidates from this round on the held-out validation set $\mathcal{D}_{\text{val}}$. After duplicates are removed, the N_{pop} most accurate configurations among the current population and the new candidates form the next population:

$$\mathcal{P}_{i+1} = \text{Top-}N_{\text{pop}}(\mathcal{P}_i \cup \mathcal{Q}_i)\,. \tag{6}$$

Because each candidate changes one component, its validation change can be credited to that component. Repairs that span several components thus accumulate across rounds without a search over joint edits. After I rounds, CATO returns the population member with the highest validation accuracy as its estimate of $\theta^\star$, which is then deployed without further optimization. Appendices B.1, A.5, and F give hyperparameters, edit operators, and prompts.

4 Experiments

4.1 Experimental Setup

We evaluate CATO on **AppWorld** (Trivedi et al., 2024) (API coordination), **WebArena** (Zhou et al., 2024; El hattami et al., 2025) (web navigation), and **BBH** (Suzgun et al., 2023) (symbolic reasoning) with three backbones: GLM-5.2, Qwen3.7-Plus, and DeepSeek-V4-Flash. Within each backbone setting, the same model runs the agent and all optimizer calls, and all methods use the same splits. Baselines follow their official implementations, with the maximum candidate budget aligned for methods that expose one. They comprise (1) zero-shot ReAct (Yao et al., 2022); (2) single-component optimizers ProTeGi (Pryzant et al., 2023) and MIPROv2 (Opsahl-Ong et al., 2024) (θ_p), DRAFT (Qu et al., 2025) (θ_t), and ACE (Zhang et al., 2026c) (θ_e); and (3) multi-component optimizers GEPA (Agrawal et al., 2026) and OPTIMAS (Wu et al., 2026).

By default, $I = 6$, $N_{\text{pop}} = 5$, $k = 2$, $B_{\text{cand}} = 16$, $\eta = 0.6$, and $N_{\min} = 8$. We report Task and Scenario Goal Completion (**TGC/SGC**) on AppWorld and Success Rate (**SR**) on WebArena and BBH. All scores average three independent end-to-end runs (three evaluations for ReAct). Environment setups and all hyperparameters appear in Appendix B.

4.2 Main Results

Overall Performance. Table 1 shows that CATO attains the best score in all 18 backbone–metric combinations. Margins over the strongest competing optimizer follow where failures originate (Section 4.5). They peak on AppWorld SGC (up to $+5.96$ percentage points on GLM-5.2 Test-Normal). There the score spreads across all three components, so no single-component optimizer reaches every defect. Margins are $+1.82$ to $+3.64$ points on WebArena, where ACE already edits the dominant experience bank, and smallest on BBH ($+0.76$ to $+1.17$ points). On BBH, GEPA already optimizes the dominant prompt and exceeds 91%, leaving little headroom.

Scenario-Level Gains. SGC counts a scenario only when all its tasks succeed. In all six AppWorld comparisons, CATO's margin over the strongest baseline is larger on SGC than on TGC (e.g., $+5.96$ vs. $+2.97$ points on GLM-5.2 Test-Normal). Consolidated gradients yield component-level edits that can fix a defect shared by several tasks in a scenario.

Transfer to Unseen Applications. Test-Challenge uses applications absent from training. CATO keeps the best score on every Test-Challenge metric, leading by $+0.96$ to $+3.12$ points on TGC and $+2.15$ to $+5.52$ points on SGC. OPTIMAS beats the prompt-only ProTeGi on Test-Normal TGC for all three backbones but trails it on both Test-Challenge metrics. Gains on seen applications thus do not guarantee transfer. CATO's transfer is consistent with its single-component edits, whose removal hurts Test-Challenge most (Section 4.4).

4.3 Attribution Accuracy and Repair Utility

We test attribution on controlled single-component fault injections on AppWorld with GLM-5.2 (Appendix C.1). Attribution accuracy is strict: a case counts as correct only when CATO gives the injected component a positive score higher than both other components; ties and all-zero scores count as errors.

Table 1: Main results. Scores (%) are means over three independent runs; BBH averages ten independently optimized subtasks (Appendix D.4). Within each backbone, bold and underlined values mark the best and second-best scores per column; ↑ gives the gain over ReAct in points.

Backbone	Method	AppWorld Test-Normal		AppWorld Test-Challenge		WebArena	BBH
		TGC	SGC	TGC	SGC	SR	SR
GLM-5.2	ReAct	81.55	64.29	66.35	46.52	46.06	87.02
	ProTeGi	87.30 ↑5.75	79.17 ↑14.88	79.30 ↑12.95	65.23 ↑18.71	50.91 ↑4.85	90.75 ↑3.73
	MIPROv2	88.10 ↑6.55	78.57 ↑14.28	79.14 ↑12.79	61.87 ↑15.35	50.30 ↑4.24	91.38 ↑4.36
	DRAFT	87.50 ↑5.95	78.57 ↑14.28	78.18 ↑11.83	61.15 ↑14.63	49.09 ↑3.03	87.55 ↑0.53
	ACE	88.10 ↑6.55	76.79 ↑12.50	83.05 ↑16.70	73.38 ↑26.86	52.12 ↑6.06	89.23 ↑2.21
	GEPA	90.48 ↑8.93	83.33 ↑19.04	84.97 ↑18.62	73.86 ↑27.34	51.52 ↑5.46	92.76 ↑5.74
	OPTIMAS	89.88 ↑8.33	76.79 ↑12.50	78.98 ↑12.63	62.59 ↑16.07	49.90 ↑3.84	91.13 ↑4.11
	CATO	**93.45** ↑11.90	**89.29** ↑25.00	**86.89** ↑20.54	**77.94** ↑31.42	**55.76** ↑9.70	**93.93** ↑6.91
Qwen3.7-Plus	ReAct	83.33	73.21	74.02	55.88	42.42	83.30
	ProTeGi	86.31 ↑2.98	75.60 ↑2.39	78.34 ↑4.32	66.19 ↑10.31	50.30 ↑7.88	90.12 ↑6.82
	MIPROv2	86.90 ↑3.57	82.74 ↑9.53	78.26 ↑4.24	67.15 ↑11.27	49.70 ↑7.28	92.39 ↑9.09
	DRAFT	84.52 ↑1.19	75.00 ↑1.79	78.66 ↑4.64	64.03 ↑8.15	46.67 ↑4.25	84.71 ↑1.41
	ACE	89.29 ↑5.96	83.33 ↑10.12	79.86 ↑5.84	67.39 ↑11.51	52.73 ↑10.31	88.64 ↑5.34
	GEPA	88.10 ↑4.77	82.14 ↑8.93	78.74 ↑4.72	66.91 ↑11.03	49.29 ↑6.87	93.03 ↑9.73
	OPTIMAS	87.10 ↑3.77	80.95 ↑7.74	77.94 ↑3.92	64.75 ↑8.87	47.88 ↑5.46	92.56 ↑9.26
	CATO	**90.48** ↑7.15	**85.71** ↑12.50	**80.82** ↑6.80	**69.54** ↑13.66	**55.15** ↑12.73	**94.11** ↑10.81
DeepSeek-V4-Flash	ReAct	83.13	66.67	70.50	50.12	41.21	81.86
	ProTeGi	86.51 ↑3.38	77.38 ↑10.71	79.06 ↑8.56	64.75 ↑14.63	48.48 ↑7.27	85.29 ↑3.43
	MIPROv2	87.70 ↑4.57	77.98 ↑11.31	82.33 ↑11.83	68.35 ↑18.23	47.27 ↑6.06	87.90 ↑6.04
	DRAFT	85.12 ↑1.99	76.79 ↑10.12	76.74 ↑6.24	59.71 ↑9.59	43.64 ↑2.43	82.27 ↑0.41
	ACE	90.28 ↑7.15	80.95 ↑14.28	82.97 ↑12.47	68.59 ↑18.47	50.91 ↑9.70	86.85 ↑4.99
	GEPA	89.29 ↑6.16	80.36 ↑13.69	83.53 ↑13.03	68.82 ↑18.70	46.67 ↑5.46	91.50 ↑9.64
	OPTIMAS	88.89 ↑5.76	78.57 ↑11.90	78.90 ↑8.40	64.03 ↑13.91	45.66 ↑4.45	89.99 ↑8.13
	CATO	**93.25** ↑10.12	**86.31** ↑19.64	**86.65** ↑16.15	**74.34** ↑24.22	**52.73** ↑11.52	**92.26** ↑10.40

Controlled Fault Attribution. Across 108 single-fault cases, CATO achieves 95.7% attribution accuracy, a 42.9-point gain over trajectory-only diagnosis (Figure 3a). Every component improves: tools gain most, 75.5 points (93.1% vs. 17.6%); prompts and experiences gain 29.8 and 26.0. A faulty tool description produces wrong tool calls that resemble reasoning errors in a single trace. A same-task success shows where the calls diverge, and trajectory-only diagnosis has no such reference.

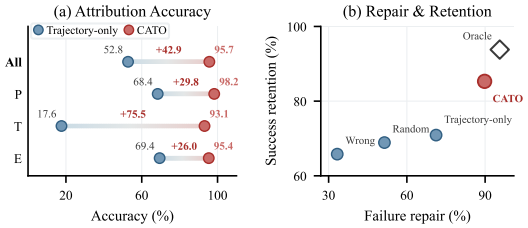


Figure 3: Attribution and repair utility on 108 controlled faults. (a) Strict attribution accuracy. (b) Failure repair versus success retention under different ways of choosing the component to edit.

Downstream Repair Utility. With candidate generation and validation fixed, CATO repairs 89.8% of injected failures and retains 85.3% of prior successes (Figure 3b). They trail an oracle that always edits the injected component by only 5.8 and 8.5 points. Trajectory-only diagnosis misses the faulty component in nearly half of the cases (Figure 3a), so its repair and retention fall to 71.2% and 70.9%. Editing a wrong component retains only 65.8%, so misdirected edits also break tasks that already worked. Appendices C.1 and G give the evaluation protocol and four end-to-end repair cases.

4.4 Ablation Studies

Necessity of Contrastive Attribution. Replacing contrastive attribution with trajectory-only diagnosis lowers Test-Normal SGC by 12.50 points, and random attribution lowers it by 17.86 points (Table 2). Both variants edit the wrong component more often, as Section 4.3 shows. Assigning every failure to all components lowers SGC by 7.74 points and nearly doubles training tokens (222.17M vs. 113.41M). Each failure then spawns candidates for components it does not implicate.

Neighbor Selection and Contrast Pairs. Using a random neighbor instead of the median-distance one lowers SGC by 4.17 points on Test-Normal and 3.84 points on Test-Challenge at similar cost

Table 2: Ablations on AppWorld with GLM-5.2 under the same per-round candidate cap. Scores are in %; Δ is the change from full CATO in points. Training cost covers the full optimization run. Variants are defined in Appendix A; further sweeps appear in Appendix D.

Variant	Test-Normal				Test-Challenge				Training Cost	
	TGC		SGC		TGC		SGC		Tokens (M)	API Calls
	Score	Δ	Score	Δ	Score	Δ	Score	Δ		
All-component assignment	88.89	**−4.56**	81.55	**−7.74**	83.53	**−3.36**	73.38	**−4.56**	222.17	28,473
Random attribution	84.92	**−8.53**	71.43	**−17.86**	81.77	**−5.12**	67.63	**−10.31**	127.36	16,723
Trajectory-only	86.31	**−7.14**	76.79	**−12.50**	82.73	**−4.16**	69.78	**−8.16**	132.94	17,175
w/o Median-distance neighbor	90.87	**−2.58**	85.12	**−4.17**	84.65	**−2.24**	74.10	**−3.84**	113.03	14,782
w/o Contrast-pair priority	88.69	**−4.76**	80.36	**−8.93**	84.01	**−2.88**	71.94	**−6.00**	120.92	15,733
w/o Score-based budget	91.67	**−1.78**	85.12	**−4.17**	85.45	**−1.44**	73.62	**−4.32**	224.33	29,608
w/o Gradient consolidation	89.88	**−3.57**	80.36	**−8.93**	84.89	**−2.00**	72.66	**−5.28**	119.18	15,824
w/o Single-component edits	90.28	**−3.17**	84.52	**−4.77**	81.53	**−5.36**	66.19	**−11.75**	98.44	12,799
Full CATO	**93.45**	—	**89.29**	—	**86.89**	—	**77.94**	—	113.41	14,851

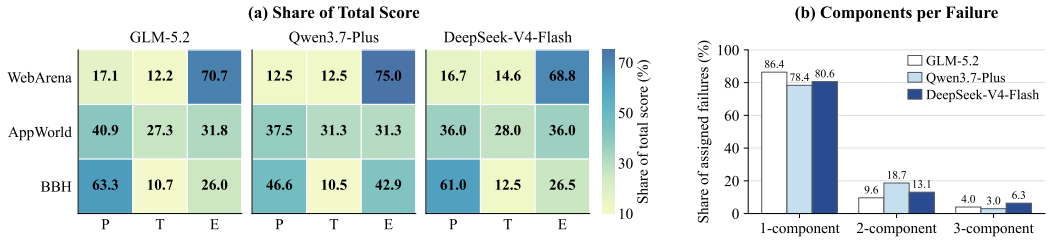


Figure 4: Share of the total component score (a) and number of components assigned per failure (b) across nine settings ($\eta = 0.6$). Panel (b) pools assigned failures across benchmarks for each backbone. P/T/E follow Figure 1.

(113.03M vs. 113.41M tokens). A random neighbor may differ too little to expose the faulty component or too much to isolate it (Appendix D.2). Removing contrast-pair priority lowers SGC by 8.93 and 6.00 points on the two splits, because fewer failures have a same-task success as reference.

Targeted Optimization and Single-Component Edits. Uniform allocation doubles training tokens (224.33M) yet lowers every score, because weakly supported groups receive as many candidates as strongly supported ones. Without gradient consolidation, candidates follow single-case advice, and Test-Normal SGC drops by 8.93 points. Jointly editing all assigned components cuts training cost to 98.44M tokens but causes the largest Test-Challenge SGC drop (11.75 vs. 4.77 points on Test-Normal). Validation covers only seen applications, so a joint edit can pass it through coupled changes that fit those applications but not unseen ones. Editing one component at a time limits such fitting, which matters most under distribution shift.

4.5 Distribution of Scores and Assignments

Benchmark-Dependent Scores. The share of the total component score differs by benchmark (Figure 4a). Experience receives 68.8–75.0% of the total score on WebArena, Prompt receives the largest share on BBH (46.6–63.3%), and AppWorld spreads the score across all three components (27.3–40.9%). Accordingly, ACE, which edits only the experience bank, is the strongest baseline on WebArena for all three backbones.

Most Failures Target One Component. Among assigned failures, 78.4–86.4% per backbone are assigned to a single component (Figure 4b), so most failures lead to one targeted edit. This supports single-component candidates, since the rest can be repaired as edits accumulate over rounds. Per-setting distributions appear in Appendix C.2.

Table 3: Resource use on AppWorld with GLM-5.2 for full training and Test-Normal inference. Tokens are in millions; ReAct has no training stage.

Method	Training			Inference			TGC (%)
	Prompt (M)	Compl. (M)	API Calls	Prompt (M)	Compl. (M)	API Calls	
ReAct	—	—	—	20.89	0.50	2,707	81.55
ProTeGi	195.56	5.64	21,440	21.76	0.63	2,584	87.30
MIPROv2	133.89	9.84	9,690	40.50	0.71	2,158	88.10
DRAFT	17.94	8.09	4,429	87.95	1.05	3,138	87.50
ACE	216.43	4.81	6,153	73.10	0.65	2,240	88.10
GEPA	137.94	3.91	14,201	22.17	0.66	2,234	90.48
OPTIMAS	242.09	8.37	26,083	26.35	0.87	2,749	89.88
CATO	108.56	4.85	14,851	21.98	0.63	2,598	93.45

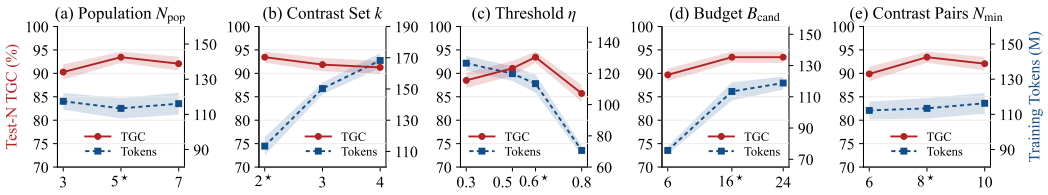


Figure 5: Hyperparameter sensitivity on AppWorld with GLM-5.2: mean Test-Normal TGC (%) over three runs and full-training tokens (M). Shading: ± 1 std; $\star$: default. Token axes differ across panels.

4.6 Resource Footprint and Deployment Cost

Training cost counts all optimizer calls plus training and validation executions under each method's original workflow, without equalization (Table 3). DRAFT trains cheaply because it rewrites tool descriptions without executing training cases, but its longer descriptions quadruple inference prompt tokens (87.95M). CATO uses 113.41M training tokens, the fewest among optimizers except DRAFT and about half of ACE (221.24M) and OPTIMAS (250.46M). Score-based allocation often leaves budget unused, so its gains do not come from more compute. Its 14,851 training API calls are mid-range, since CATO validates every candidate separately. At deployment, CATO's fixed configuration keeps inference tokens and API calls close to ReAct while raising TGC by 11.90 points. ACE, in contrast, carries its accumulated experience bank into every prompt (73.10M inference prompt tokens). Appendix D.3 reports direct optimizer costs.

4.7 Hyperparameter Sensitivity

Defaults are selected by validation accuracy on AppWorld with GLM-5.2 and reused unchanged for all other backbones and benchmarks. Increasing N_{pop} to 7 or k to 4 raises tokens from 113.41M to 116.08M or 168.27M without improving Test-Normal TGC (Figure 5a,b). $\eta = 0.6$ performs best (Figure 5c): lower thresholds admit weakly supported assignments, while $\eta = 0.8$ cuts cost to 70.64M tokens but misses valid update targets. Neither $B_{\text{cand}} > 16$ nor $N_{\text{min}} \neq 8$ improves Test-Normal TGC (Figure 5d,e); full sweeps appear in Appendix D.1.

5 Conclusion

We formulate the choice of which component to edit as structural attribution. It assigns a failure to the components a repair would need to edit, rather than to trajectory steps. CATO infers these components from same-task contrasts, allocates more edits to better-supported ones, and keeps validated single-component edits. It identifies injected faults with 95.7% accuracy and outperforms all six baselines in every setting at half the training cost of editing every component. The gains are largest when failures span components and persist on unseen applications. CATO needs a same-task success as a reference and repairs coupled defects one edit at a time.

AI USE STATEMENT

We used LLM assistants to search the literature, polish author-written text, and help implement experiment code. We did not use them to develop ideas, design the method or experiments, or interpret results. All AI-assisted code was reviewed and tested by the authors, and all references were checked against the original publications. The authors take full responsibility for the content of this paper.

ETHICS STATEMENT

All experiments use public benchmarks (AppWorld, WebArena, and BBH) in sandboxed, self-hosted environments and involve no human subjects or personal data. Because CATO accepts edits based only on task success, an edit could weaken safety-relevant instructions without lowering accuracy. Each accepted edit is an auditable single-component text diff, and we recommend reviewing optimized configurations before deployment.

REPRODUCIBILITY STATEMENT

Anonymous source code and run scripts are provided in the supplementary material. Section 3 and Appendix A specify the full algorithm; Appendix B gives hyperparameters, data splits, models, and baseline settings; Appendix C.1 describes the fault-injection protocol; and Appendix F lists all prompt templates.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. FLEX: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, 2025.

Minghui Chen, Wenlong Deng, James Y Zou, Han Yu, and Xiaoxiao Li. Textual equilibrium propagation for deep compound AI systems. In *International Conference on Learning Representations*, volume 2026, pp. 151615–151634, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024a.

Yongchao Chen, Jacob Arkin, Yilun Hao, Yang Zhang, Nicholas Roy, and Chuchu Fan. PRompt optimization in multi-step tasks (PROMST): Integrating human feedback and heuristic-based sampling. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3859–3920, 2024b.

Amine El hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. WebArena verified: Reliable evaluation for web agents. In *Workshop on Scaling Environments for Agents*, 2025.

Yangyi Fang, Jiaye Lin, Xiaoliang Fu, Cong Qin, Haolin Shi, and Chang Liu. Proximity-based multi-turn optimization: Practical credit assignment for LLM agent training. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL 2026)*, pp. 285–307, 2026.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

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*, volume 2024, pp. 54928–54958, 2024.

Yu-Ang Lee, Guan-Ting Yi, Mei-Yi Liu, Jui-Chao Lu, Guan-Bo Yang, and Yun-Nung Chen. Compound AI systems optimization: A survey of methods, challenges, and future directions. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 28760–28775, 2025.

Yanming Li, Xuelin Zhang, Wenjie Lu, Ziye Tang, Maodong Wu, Haotian Luo, Tongtong Wu, Zijie Peng, Hongze Mi, Yibo Feng, Naiqiang Tan, Chao Huang, Hong Chen, and Li Shen. Who deserves the reward? SHARP: Shapley credit-based optimization for multi-agent system. *arXiv preprint arXiv:2602.08335*, 2026a.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Y Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. In *International Conference on Learning Representations*, volume 2026, pp. 50524–50570, 2026b.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 267:1–38, 2019.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2Skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

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, 2024.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 126544–126565, 2024.

Jiangweizhi Peng, Yuanxin Liu, Ruida Zhou, Charles Fleming, Zhaoran Wang, Alfredo Garcia, and Mingyi Hong. HiPER: Hierarchical reinforcement learning with explicit credit assignment for large language model agents. *arXiv preprint arXiv:2602.16165*, 2026.

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, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Yan Lan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. From exploration to mastery: Enabling LLMs to master tools via self-driven interactions. In *International Conference on Learning Representations*, volume 2025, pp. 55675–55697, 2025.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551, 2023.

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, 2023.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, 2018.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 13003–13051, 2023.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Wenxiao Wang, Priyatham Kattakinda, and Soheil Feizi. Maestro: Joint graph & config optimization for reliable AI agents. *arXiv preprint arXiv:2509.04642*, 2025.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Shirley Wu, Shiyu Zhao, Qian Huang, Kexin Huang, Michihiro Yasunaga, Kaidi Cao, Vassilis N. Ioannidis, Karthik Subbian, Jure Leskovec, and James Zou. AvaTaR: Optimizing LLM agents for tool usage via contrastive reasoning. In *Advances in Neural Information Processing Systems*, volume 37, pp. 25981–26010, 2024.

Shirley Wu, Parth Sarthi, Shiyu Zhao, Aaron Lee, Herumb Shandilya, Adrian Mladenic Grobelnik, Nurendra Choudhary, Edward W. Huang, Karthik Subbian, Linjun Zhang, Diyi Yang, James Zou, and Jure Leskovec. Optimas: Optimizing compound AI systems with globally aligned local rewards. In *International Conference on Learning Representations*, volume 2026, pp. 149685–149706, 2026.

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*, volume 2024, pp. 12028–12068, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Dongsheng Li, and Deqing Yang. EvoAgent: Towards automatic multi-agent generation via evolutionary algorithms. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 6192–6217, 2025.

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.

Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems? In *International Conference on Learning Representations*, volume 2026, pp. 11377–11399, 2026a.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026c.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 76583–76599, 2025b.

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-Pro: Learning to evolve via policy-level reflection and optimization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5348–5375, 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 62743–62767, 2024.

TABLE OF CONTENTS

A Method Details 15

A.1 Full Algorithm . . . 15

A.2 Failure Selection for Attribution . . . 16

A.3 Three-Step Attribution and Assignment . . . 16

A.4 Candidate Budget Allocation . . . 17

A.5 Single-Component Edit Operators . . . 17

B Experimental Setup and Implementation 18

B.1 Default Hyperparameters . . . 18

B.2 Data Splits and Evaluators . . . 18

B.3 Matched Backbones and Baseline Alignment . . . 19

C Attribution Evaluation 19

C.1 Fault Injection and Repair-Utility Protocol . . . 19

C.2 Score and Assignment Distribution: Detailed Numerical Results . . . 20

C.3 Fault Injection Examples . . . 21

D Additional Experimental Results 22

D.1 Hyperparameter Sensitivity . . . 22

D.2 Neighbor Distance Strategy . . . 23

D.3 Optimizer Direct Cost Breakdown . . . 23

D.4 BBH Per-Subtask Breakdown . . . 24

D.5 Four-Component Modular Extension (Skill Component) . . . 24

D.6 Convergence Dynamics . . . 24

E Limitations and Broader Societal Impact 25

F Full Prompt Templates 25

F.1 Behavior-Level Trajectory Analysis . . . 26

F.2 Component Attribution and Case-Level Textual Gradient . . . 27

F.3 Component-Level Textual Gradient Consolidation . . . 28

F.4 Single-Component Candidate Generation . . . 28

G Case Studies: Contrastive Attribution and Validation-Task Transfer 29

G.1 Case Study 1: Experience Refinement (θ_e) . . . 29

G.2 Case Study 2: System Prompt Alignment (θ_p) . . . 32

G.3 Case Study 3: Tool Description Disambiguation (θ_t) . . . 34

G.4 Case Study 4: Skill Refinement (θ_s) . . . 36

A METHOD DETAILS

A.1 FULL ALGORITHM

Algorithm 1: CATO

Require: Seed $\theta^{(0)}$; $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}}$; hyperparameters $I, k, N_{\text{pop}}, B_{\text{cand}}, N_{\min}, \eta, N_{\text{cg}}$
Ensure: Most accurate configuration on $\mathcal{D}_{\text{val}}$

Evaluate $\theta^{(0)}$ on the held-out validation set
$\mathcal{P} \leftarrow \{\theta^{(0)}\}$
for $i = 1$ to I **do**
  if $|\mathcal{P}| = 1$ **then** ▷ failure-only attribution
    $\mathcal{C}_i \leftarrow \mathcal{P}$; skip neighbor sampling
  else
    Select a center θ^{c} ranked by validation Acc, selection count, creation iteration; if the top center was used last round, take the next-ranked center θ^{c}
    Rank neighbors by how close their distance $\delta(\cdot, \theta^{\text{c}})$ is to the median distance, breaking ties by selection count, validation Acc; form $\mathcal{C}_i$ from θ^{c} and up to $k-1$ top-ranked neighbors
  end if
  Execute $\mathcal{C}_i$ on the same training batch $\mathcal{B}_i \subset \mathcal{D}_{\text{train}}$ to obtain records $r_{x,\theta} = (x, \theta, y_{x,\theta}, \tau_{x,\theta})$
  Label each task contrast-pair / all-failure / all-success; filter failed records into $\mathcal{F}_i$
  $\mathcal{U}_i \leftarrow \emptyset$
  for each $r \in \mathcal{F}_i$ **do**
    Run three-step attribution to obtain $\{s_p^{(r)}, s_t^{(r)}, s_e^{(r)}\}$ and $\nabla_{\text{case}}^{(r)}$
    Add $u_{r,d} = (r, d, s_d^{(r)}, \nabla_{\text{case}}^{(r)})$ to $\mathcal{U}_i$ for each d with $s_d^{(r)} > \eta$
  end for
  if $\mathcal{U}_i = \emptyset$ **then**
    Keep $\mathcal{P}$; continue ▷ empty round
  end if
  Group $\mathcal{U}_i$ by $g = (\theta, d)$ with $\theta = \theta^{(r,-)}$; compute a_g; allocate candidate attempts under cap B_{cand} to obtain $\{b_g\}$
  for each group $g = (\theta, d)$ with $b_g > 0$ **do**
    Select at most N_{cg} assignments with highest scores from $\mathcal{U}_g$; consolidate them into $\nabla_{\text{comp}}^{(g)}$; generate b_g candidates θ' that edit only component d of θ
  end for
  Evaluate candidates on $\mathcal{D}_{\text{val}}$; merge with $\mathcal{P}$, deduplicate, and keep Top-N_{pop}
end for
return $\arg\max_{\theta \in \mathcal{P}} \text{Acc}(\theta; \mathcal{D}_{\text{val}})$

Algorithm 1 gives the full CATO pseudocode. Each round builds a contrastive configuration set and runs it on the same tasks, then selects failures, attributes and assigns them, allocates the candidate budget, generates single-component candidates, and accepts on validation. If every task in the batch succeeds, the round skips attribution and candidate generation.

Component distances. Each component distance lies in $[0, 1]$ before averaging, so no single component contributes more than $1/|\mathcal{K}|$ to Equation 2.

Prompt. We split each prompt into a sequence of word and punctuation tokens and compute the gestalt pattern-matching similarity $\text{sim}(a, b) \in [0, 1]$ of Ratcliff and Obershelp, as implemented in Python's `difflib` without heuristic junk filtering. Because this similarity is not symmetric, we average both directions:

$$\delta_p = 1 - \tfrac{1}{2}\big[\text{sim}(\theta_p, \theta'_p) + \text{sim}(\theta'_p, \theta_p)\big]. \tag{7}$$

Tools and experience. For $d \in \{t, e\}$, let $\mathcal{I}_d = \mathcal{I}(\theta_d) \cup \mathcal{I}(\theta'_d)$ be the union of entry IDs, and let $\theta_{d,j}$ denote entry j of component θ_d. An entry is changed if it appears on only one side or if its compared

fields differ: description and schema for tools, description and content for experience.

$$\delta_d = \frac{\left|\{j \in \mathcal{I}_d : \theta_{d,j} \neq \theta'_{d,j}\}\right|}{|\mathcal{I}_d|}, \tag{8}$$

with $\delta_d = 0$ when $\mathcal{I}_d = \emptyset$. The w/o Median-distance neighbor ablation replaces the median-distance ranking of neighbors with a uniformly random choice among the non-center population members.

A.2 Failure Selection for Attribution

Let N_{pair} count the tasks in the round that form contrast pairs, and let $N_{\min} > 0$ be the minimum contrast-pair count (default 8). CATO retains all contrast-pair tasks and supplements them with available all-failure tasks only when their count falls below this threshold. Table 4 summarizes the resulting cases. With N_{fail} available all-failure tasks, the supplement count is

$$n_{\text{fill}} = \min(\max(N_{\min} - N_{\text{pair}}, 0), N_{\text{fail}}). \tag{9}$$

Table 4: Failure selection after seed initialization, with $N_{\min} > 0$.

	Condition	Selected failures and diagnosis
A	$N_{\text{pair}} \geq N_{\min}$	All contrast-pair tasks; contrastive
B	$0 < N_{\text{pair}} < N_{\min}$	All contrast-pair tasks + n_{fill} all-failure tasks; fill-ins use failure-only diagnosis
C	$N_{\text{pair}} = 0$	$\min(N_{\min}, N_{\text{fail}})$ all-failure tasks; failure-only diagnosis
—	Batch all-success	None; skip

The seed round uses all valid failed records and does not apply this supplement rule to its batch. For a record r from a contrast pair, the successful side $\theta^{(r,+)}$ has the highest validation Acc among same-task successes, with deterministic tie-breaking. Failure-only records have no successful side: the seed round uses one failed execution, while later rounds attribute the center's failure and use the failed neighbor only to separate shared from center-specific behavior. If contrast-pair tasks already exceed $N_{\min}$, all remain selected; the threshold is not a cap on their count. Each task supplies at most one attribution sample, so the selected count cannot exceed the number of valid batch tasks. The w/o Contrast-pair priority ablation disables this priority rule ($N_{\min} = 0$), admitting both contrast pairs and unanchored all-failure records without filtering.

A.3 Three-Step Attribution and Assignment

For each failed execution record $r \in \mathcal{F}_i$, attribution centers on the failed side $(\theta^{(r,-)}, \tau^{(r,-)})$ and, when available, its same-task successful side $(\theta^{(r,+)}, \tau^{(r,+)})$. Three steps produce component scores and a case-level textual gradient $\nabla^{(r)}_{\text{case}}$.

Step 1 prepares configuration evidence without calling a language model. It always supplies the full editable content of the failed configuration $\theta^{(r,-)}$. Under contrastive diagnosis, it also builds the rule-based configuration difference $\Delta\theta^{(r)}$, a component-wise contrast of $\theta^{(r,-)}$ to $\theta^{(r,+)}$. Under failure-only diagnosis there is no successful side, so no configuration difference is built. This step supplies checkable configuration facts and does not emit component-blame labels.

Step 2 asks a language model for a behavior-level failure explanation: given visible trajectories and task outcomes, it states where the failed execution went wrong. The step forbids Prompt / Tools / Experience labels and any early choice of which component to update, so scoring in step 3 does not presuppose its own answer.

Step 3 combines the task description, the step-1 configuration evidence, the step-2 explanation, and the trajectory, and emits component scores $s^{(r)}_p, s^{(r)}_t, s^{(r)}_e \in [0, 1]$ and a case-level textual gradient $\nabla^{(r)}_{\text{case}}$ with three fields: symptom, contrast, and repair prior. Each $s^{(r)}_d$ is evidence from record r for updating component d, not a mutually exclusive cause; scores are thresholded separately and need not sum to one, so one record may support several components. For every $s^{(r)}_d > \eta$, CATO creates $u_{r,d} = (r, d, s^{(r)}_d, \nabla^{(r)}_{\text{case}})$. If $\max_d s^{(r)}_d \leq \eta$, the record is not assigned and contributes no textual gradient to refinement. Otherwise, the case-level textual gradient gives actionable advice

rather than a full component rewrite. Later, the per-component consolidation step $\mathrm{Consolidate}_d$ selects the top-scoring assignments (up to N_{cg}, breaking ties deterministically by execution ID) to synthesize reusable component-level advice, so the input set in Equation 5 contains at most N_{cg} case-level textual gradients. The w/o Gradient consolidation ablation skips this step and generates candidates from the case-level textual gradient of the highest-scoring assignment in each group.

A.4 Candidate Budget Allocation

The per-round candidate budget B_{cand} caps candidate attempts; it does not require every round to use the full budget. Assignments are grouped by failed configuration and target component, with total score $a_g = \sum_{u_{r,d} \in \mathcal{U}_g} s_d^{(r)}$. Each group requests the ceiling of its total score:

$$q_g = \lceil a_g \rceil, \qquad Q = \sum_g q_g. \tag{10}$$

If $Q \leq B_{\mathrm{cand}}$, allocate $b_g = q_g$ and leave the unused budget unspent. If $Q > B_{\mathrm{cand}}$, compute proportional quotas and initial integer allocations:

$$\tilde{b}_g = B_{\mathrm{cand}} \frac{a_g}{\sum_h a_h}, \qquad b_g = \min(q_g, \lfloor \tilde{b}_g \rfloor), \qquad R = B_{\mathrm{cand}} - \sum_g b_g. \tag{11}$$

Distribute the remaining R candidates in decreasing fractional-part order, with deterministic tie-breaking, without exceeding any group's demand q_g. Groups allocated zero candidates generate none; validation scores are not included in the total score for budget allocation. For example, total scores 0.8 and 1.2 request one and two candidates, so a budget of 16 allocates only three attempts. Invalid or duplicate candidates can consume attempts, so the number of validated candidates may be smaller than the allocated budget. The w/o Score-based budget ablation keeps the same groups but splits B_{cand} evenly across them, so every round uses the full budget regardless of the total scores. The All-component assignment ablation assigns every failure to all editable components ($s_d^{(r)} = 1.0$ for all d), allocating equal candidate quotas across all components. The Random attribution ablation randomly assigns each failure to one editable component with uniform probability while preserving the single-component candidate generation pipeline. The Trajectory-only ablation operates identically to CATO's failure-only diagnosis mode: it has full access to the failed configuration's parameters $\theta^{(r,-)}$, execution trajectory $\tau^{(r,-)}$, task query, and evaluator outcome, and follows the three-step attribution and candidate refinement pipeline; it lacks only the successful counterfactual pairing $(\theta^{(r,+)}, \tau^{(r,+)})$ and the configuration difference $\Delta\theta^{(r)}$.

A.5 Single-Component Edit Operators

Table 5 lists allowed operators. A candidate must change exactly one target component relative to its parent; otherwise it is rejected.

Table 5: Allowed edit operators.

Component	Operators	Constraints
θ_p (prompt)	Full rewrite	Required placeholders are preserved
θ_t (tool descriptions)	Description update	Tool ID and schema are fixed
θ_e (experience)	Add, update, delete	Update and delete target existing entry IDs
θ_s (skill)	Add, update, delete	Separate from experience; four-component setting only

The w/o Single-component edits ablation keeps score-based assignment and merges group allocations under the same candidate budget B_{cand}, allowing each candidate to jointly update the assigned components.

Decoding. Candidate generation uses temperature 0.7 for candidate diversity. Attribution scoring, trajectory diagnosis, and candidate validation use greedy decoding (temperature 0.0). Selection ties use deterministic rules; greedy decoding does not guarantee identical outputs across repeated model calls.

B Experimental Setup and Implementation

B.1 Default Hyperparameters

Table 6 summarizes the default hyperparameters of CATO. Candidate acceptance depends strictly on performance on the held-out validation set $\mathcal{D}_{\text{val}}$. Test evaluation is performed after optimization and is not used for candidate selection.

Table 6: Default hyperparameters of CATO.

Symbol	Default	Description
k	2	Contrast set size
N_{pop}	5	Maximum population size
I	6	Optimization rounds
B_{cand}	16	Candidate budget per round
$N_{\min}$	8	Minimum contrast-pair count (> 0)
η	0.6	Score threshold (strict)
N_{cg}	5	Maximum assignments consolidated per group (top-scoring)
$\lvert\mathcal{B}_i\rvert$	32	Training batch size per round
N_{val}	57 (AW) / 33 (WA) / 20% (BBH)	Validation evaluation size
N_{ret}	3	Experience retrieval count

B.2 Data Splits and Evaluators

B.2.1 AppWorld

We follow the official AppWorld protocol (Trivedi et al., 2024) with the standard 90/57 train/validation split. Candidate validation during offline search evaluates across the full official validation split (57 tasks). Final evaluation runs on the complete Test-Normal (168 tasks) and Test-Challenge (417 tasks) splits. The official programmatic state checker evaluates target database states and execution outputs without an LLM judge. TGC measures task success; SGC counts a scenario as successful only when all its tasks succeed.

B.2.2 WebArena

The raw WebArena dataset comprises 812 tasks across 124 (website, intent_template) templates in 5 domains. Tasks from the same template differ only in their parameters, and each optimization run executes many configurations on the training and validation tasks. We therefore sample a subset of 333 tasks by template, which reduces near-duplicate tasks and evaluation cost while keeping all five website domains represented. To prevent leakage across partitions, we split this subset at the template level, stratified by website domain with random seed 0, into 135 training tasks, 33 validation tasks, and 165 held-out test tasks. This protocol guarantees zero template and intent overlap across the train, validation, and test splits. Evaluations are conducted within the WebArena-Verified environment (Zhou et al., 2024; El hattami et al., 2025), where official programmatic evaluators score URL, DOM state, or database side effects.

B.2.3 BBH

We selected and fixed 10 diverse subtasks out of the 23 BBH tasks with random seed 265 before running the reported experiments. Each subtask has its own 40%/20%/40% train/validation/test split and is optimized independently using its training and validation data. The selected configuration for each subtask is evaluated on that subtask's test split. Scoring uses rule-based exact string and regular expression matching against reference answers. The overall BBH score is the mean of the ten subtask success rates.

BBH tools. BBH provides no native tools. To make the tool component θ_t editable on BBH, we implement a fixed set of 14 deterministic utility tools that the agent can call: counting (`count_occurrences`, `count_items`, `count_words`), comparison (`compare_numbers`, `check_equal`), list operations (`extract_items`, `sort_items`, `find_item_position`), integer arithmetic (`add_numbers`, `subtract_numbers`, `multiply_numbers`), text extraction (`extract_numbers`, `extract_words`), and bracket balance checking (`check_pairs`). The

same tool set is shared by all subtasks and methods. Optimization edits only the tool descriptions; tool implementations, identifiers, and parameter schemas remain fixed.

Feedback available to attribution. Attribution analyzes failed training executions using their visible trajectories, terminal outcomes, and the feedback supplied by the task evaluator. For BBH, the reference answer is available, so an incorrect final answer can be compared with that reference. For AppWorld and WebArena, attribution receives evaluator feedback, including available failed checks or terminal errors, rather than a reference answer.

B.3 Matched Backbones and Baseline Alignment

All primary comparisons use three matched-backbone settings: GLM-5.2, Qwen3.7-Plus, and DeepSeek-V4-Flash. Within each comparative experiment, the agent, the attribution model, and the candidate generator share the same backbone model.

Single-component baselines comprise ProTeGi (Pryzant et al., 2023) and MIPROv2 (DSPy) (Opsahl-Ong et al., 2024) for prompt optimization (θ_p), DRAFT (Qu et al., 2025) for tool description refinement (θ_t), and ACE (Zhang et al., 2026c) for experience accretion (θ_e). Multi-component baselines comprise GEPA (Agrawal et al., 2026) and OPTIMAS (Wu et al., 2026), which register identical editable component surfaces as CATO and operate under matched candidate budgets B_{cand}.

C Attribution Evaluation

C.1 Fault Injection and Repair-Utility Protocol

Controlled Fault Construction. All attribution and repair experiments in this section use GLM-5.2 as the agent, attribution, and candidate-generation backbone. Starting from successful task executions on AppWorld, we modify prompts, tool descriptions, and experiences simultaneously. Exactly one edit introduces a functional fault as the ground-truth label. The other two edits serve as neutral perturbations. We first verify that each neutral edit preserves task success in isolation. We then apply both neutral edits together and require the execution to remain successful. Finally, we add the labeled fault and retain the case only when the execution changes to failure. This joint-neutral control rules out failures caused by interactions between the two neutral edits. The resulting set contains 108 cases: 38 Prompt, 34 Tools, and 36 Experience faults. Because all three components differ in each retained case, the configuration difference $\Delta\theta$ alone does not reveal the labeled fault. Reverting the labeled fault repairs the failure, so the injected component is always an update target. Other components may also qualify, and strict accuracy tests whether CATO ranks this known target first from the execution evidence. Table 7 illustrates representative injected faults. Appendix C.3 shows concrete edits and their resulting failures.

Table 7: Examples of controlled single-component fault injection.

Injected Component	Fault Construction Example
Prompt (θ_p)	Delete a critical operational constraint or corrupt the goal specification.
Tools (θ_t)	Corrupt parameter requirements, format constraints, or capability statements in a tool description; keep the tool ID and schema unchanged.
Experience (θ_e)	Inject a misleading, incorrect, or stale corrective guideline into the retrieval bank.

Attribution Decision and Accuracy. We evaluate strict attribution accuracy. The prediction is defined only when one component has a unique positive highest score:

$$\widehat{d}_r = \begin{cases} d, & s_d^{(r)} > 0 \ \wedge \ s_d^{(r)} > s_{d'}^{(r)} \quad \forall d' \neq d, \\ \emptyset, & \text{otherwise.} \end{cases} \tag{12}$$

A case r is correct only when $\widehat{d}_r$ equals its injected ground-truth component. Tied highest scores and all-zero scores therefore count as incorrect predictions. Each repetition evaluates all 108 cases once; we perform three repetitions and report their mean accuracy. Trajectory-only diagnosis operates under the failure-only diagnosis protocol (Listing 5): it receives the failed configuration's parameter

content $\theta^{(r,-)}$, execution trajectory $\tau^{(r,-)}$, task query, and evaluator feedback, evaluating attribution without access to a successful counterpart $(\theta^{(r,+)}, \tau^{(r,+)})$ or configuration difference $\Delta\theta^{(r)}$. The $+42.9$ pp gap between CATO (95.7%) and Trajectory-only (52.8%) thus strictly isolates the contribution of same-task contrastive counterfactuals from parameter visibility. This evaluation rule differs from assignment during optimization, which selects each component whose score strictly exceeds η.

Evaluation configurations. The 108 fault-injection cases used for attribution evaluation correspond to 3 Prompt configurations, 5 Tools configurations, and 4 Experience configurations. The cases comprise 38 Prompt faults, 34 Tools faults, and 36 Experience faults. Each configuration produces both successful and failed executions on AppWorld Test-Normal.

Configuration-level repair. We compare oracle, CATO, trajectory-only, random, and wrong assignment under the same candidate-generation and validation procedures. Each assignment policy produces one shared repair for each faulty configuration. We evaluate the configuration before and after repair on the same Test-Normal tasks.

Failure repair and success retention. For each faulty configuration c, the failure repair rate is the fraction of its corresponding injected-fault cases that succeed after repair, and the success retention rate is the fraction of its initially successful Test-Normal tasks that remain successful after repair. Both metrics use the same repaired configuration. The reported scores in Table 9 are the unweighted macro-averages across all 12 faulty configurations averaged over three independent repair runs.

Attribution and repair results. Table 8 reports injected-fault identification, while Table 9 reports repair and retention under the protocol above. CATO achieves 95.7% attribution accuracy, compared with 52.8% for trajectory-only diagnosis. Tools show the largest difference: 93.1% versus 17.6%. CATO reaches 89.8% repair and 85.3% retention, compared with oracle assignment at 95.6% and 93.8%. These results leave gaps of 5.8 and 8.5 percentage points to oracle assignment, respectively. CATO's repair rate exceeds that of wrong assignment by 56.5 percentage points.

Table 8: Strict attribution accuracy under controlled single-component faults (%). Overall covers all 108 cases; each component column covers the cases with a fault in that component. Ties and all-zero scores count as incorrect.

Method	Overall	Prompt	Tools	Experience
Trajectory-only	52.8	68.4	17.6	69.4
CATO	**95.7**	**98.2**	**93.1**	**95.4**

Table 9: Failure repair and success retention under different assignment policies (%). All policies share the same candidate generation and validation; scores are unweighted macro-averages over the 12 faulty configurations and three runs.

Assignment Policy	Failure Repair Rate	Success Retention Rate
Oracle	**95.6**	**93.8**
CATO	<u>89.8</u>	<u>85.3</u>
Trajectory-only	71.2	70.9
Random	51.4	68.9
Wrong	33.3	65.8

C.2 Score and Assignment Distribution: Detailed Numerical Results

Table 10 provides complete numerical records for Figure 4. The Cases column denotes all analyzed failure events across the full optimization trajectory. The Score Share columns report the normalized distribution of cumulative component scores $\sum_r s_d^{(r)}$ across the three components (normalized to 100% per row). For failures exhibiting decisive attribution evidence ($s_d^{(r)} \approx 1$ on the target component), score mass naturally concentrates on the primary implicated component. The Assigned Components columns report the proportion of all analyzed failures (Cases) assigned to 0, 1, 2, or 3 components under threshold $\eta = 0.6$, with unassigned failures ($s_d^{(r)} \leq \eta$ for all d) categorized as None. Across benchmarks, the component score concentrates on Experience in WebArena and

on Prompt in BBH, and spreads across all three components in AppWorld. Assignment to a single component dominates every setting, accounting for 51.13%–83.67% of analyzed failures (78.4%–86.4% after pooling assigned failures across benchmarks for each backbone). Assignment to all three components is uncommon in most settings (0.00%–24.00%; above 5.5% only on AppWorld with Qwen3.7-Plus and DeepSeek-V4-Flash), confirming that CATO spends most of its candidate budget on localized single-component interventions rather than unguided multi-component rewrites.

Figure 4(b) pools assigned counts across benchmarks for each backbone and excludes unassigned failures from its denominator.

Table 10: Share of cumulative component scores and distribution of assigned components across backbones and benchmarks (%). Cases denotes the total number of analyzed failures across optimization. Prompt, Tools, and Experience columns sum to 100% per row. None, Single, Dual, and Triple report the percentage of analyzed failure cases assigned to 0, 1, 2, or 3 components under threshold $\eta = 0.6$.

Backbone	Benchmark	Cases	Score Share (%)			Assigned Components (%)			
			Prompt	Tools	Exper.	None	Single	Dual	Triple
GLM-5.2	AppWorld	22	40.91	27.27	31.82	27.27	59.09	13.64	0.00
	WebArena	43	17.07	12.20	70.73	9.30	79.07	11.63	0.00
	BBH	222	63.32	10.67	26.01	15.32	73.42	6.76	4.50
Qwen3.7-Plus	AppWorld	16	37.50	31.25	31.25	18.75	56.25	12.50	12.50
	WebArena	41	12.50	12.50	75.00	17.07	68.29	9.76	4.88
	BBH	133	46.62	10.53	42.86	34.59	51.13	14.29	0.00
DeepSeek-V4-Flash	AppWorld	25	36.00	28.00	36.00	4.00	60.00	12.00	24.00
	WebArena	49	16.67	14.58	68.75	4.08	83.67	12.24	0.00
	BBH	200	61.00	12.50	26.50	1.50	80.00	13.00	5.50

C.3 Fault Injection Examples

The following examples show the injected edits and the resulting failures for each component. Queries and experience entries are condensed from the recorded cases.

Listing 1: Prompt (θ_p) — Numeric Answer Format

Query. How many Venmo friends have I made since the start of January this year?

Success criterion. Count the matching friends and submit the numeric answer 9.

Before injection

```
Numbers must be numeric and not in words.
For example, return "10", not "ten".
```

After injection

```
Numbers must be spelled out in words, not numeric.
For example, return "ten", not "10".
```

Resulting failure. The agent correctly counted nine friends but submitted `answer="nine"` instead of `answer=9`. The answer check failed.

Listing 2: Tool Description (θ_t) — Payment Amount Units

Query. My roommates and I went for dinner yesterday. I paid for the taxi ($73), and Nancy paid for everyone's food ($122). Split both costs equally. Request the taxi shares with the note "Taxi Fare" and pay Nancy my food share with the note "Dinner Bill" on Venmo.

Success criterion. Request 18.25 from each of the three roommates and pay Nancy 30.50, using the specified notes.

Before injection

```
Tool: venmo.create_transaction
Description: Send money to a user.
```

After injection

```
Tool: venmo.create_transaction
Description: Send money to a user. The amount parameter is in
cents.
```

Resulting failure. The agent converted the food share to cents and passed `amount=3050` instead of `amount=30.50`. The API recorded a payment of 3050, causing the amount check to fail.

Listing 3: Experience (θ_e) — Request Scope and Pagination

Query. My roommate requested my share of this month's housing bill on Venmo but forgot the 15% rent increase. Reject that request and send the corrected amount with the note "Correcting for 15% rent increase."

Success criterion. Reject only the target housing request, pay 805 instead of 700 with the specified note, and leave unrelated requests unchanged.

Before injection

desc: Accepting or rejecting pending Venmo payment requests from roommates, coworkers, friends, or similar groups.

content:

Do: Resolve the named relationship groups into a target sender set. Read all pages of pending requests. Approve or reject only requests from that set, following the task instruction.

Don't: Approve or reject requests regardless of sender membership. Don't stop after the first page.

After injection

desc: Accepting or rejecting pending Venmo payment requests from roommates, coworkers, friends, or similar groups.

content:

Do: Resolve the named relationship groups into a target sender set. Read only the first page of pending requests. Approve or reject every request on that page, regardless of sender membership.

Don't: Approve or reject requests regardless of sender membership. Don't stop after the first page.

Resulting failure. The agent followed the corrupted `Do` instruction and rejected all five pending requests on the first page. Only one was the target housing request. Although the subsequent payment of 805 was correct, the four unrelated rejections caused task failure.

The injection retained the original, conflicting `Don't` constraints.

D Additional Experimental Results

D.1 Hyperparameter Sensitivity

Table 11 details the complete numerical results for single-factor sweeps on AppWorld with GLM-5.2. We vary key hyperparameters ($N_{\text{pop}}, k, \eta, B_{\text{cand}}, N_{\min}$) independently while fixing remaining parameters at their default values ($N_{\text{pop}} = 5, k = 2, \eta = 0.6, B_{\text{cand}} = 16, N_{\min} = 8$).

Observations. (1) Increasing N_{pop} from 5 to 7 raises full training tokens from 113.41M to 116.08M, while increasing k from 2 to 4 raises them to 168.27M, without improving Test-Normal TGC. (2) Both a loose threshold ($\eta = 0.3, 0.5$) and an overly strict threshold ($\eta = 0.8$) underperform the default $\eta = 0.6$, supporting an intermediate score threshold that balances noise rejection with diagnostic recall. (3) Expanding B_{cand} from 16 to 24 leaves Test-Normal TGC unchanged at 93.45% but lowers Test-Normal SGC by 1.19 percentage points, while increasing full training tokens from 113.41M to 118.81M.

Table 11: Single-factor hyperparameter sensitivity on AppWorld with GLM-5.2. Scores are in %; * marks the default configuration. Training cost covers the full optimization run; tokens are in millions.

Parameter	Value	Test-Normal		Test-Challenge		Training Cost	
		TGC	SGC	TGC	SGC	Tokens (M)	API Calls
Population N_{pop}	3	90.28	82.74	85.45	74.82	117.44	17,344
	5*	**93.45**	**89.29**	**86.89**	**77.94**	113.41	14,851
	7	92.06	87.50	85.85	75.78	116.08	16,913
Contrast Set k	2*	**93.45**	**89.29**	**86.89**	**77.94**	113.41	14,851
	3	91.87	85.71	84.65	71.94	150.25	20,762
	4	91.27	86.31	85.61	73.38	168.27	23,778
Threshold η	0.3	88.49	81.55	74.74	58.99	126.50	17,248
	0.5	91.07	86.31	84.01	73.38	119.80	15,920
	0.6*	**93.45**	**89.29**	**86.89**	**77.94**	113.41	14,851
	0.8	85.71	75.00	74.10	58.99	70.64	9,594
Budget B_{cand}	6	89.68	83.33	84.65	75.54	75.66	13,678
	16*	**93.45**	**89.29**	**86.89**	**77.94**	113.41	14,851
	24	**93.45**	88.10	86.33	76.26	118.81	16,833
Contrast Pairs $N_{\min}$	6	89.88	83.93	82.01	69.30	112.23	14,136
	8*	**93.45**	**89.29**	**86.89**	**77.94**	113.41	14,851
	10	92.06	88.69	85.85	76.74	116.31	15,707

D.2 Neighbor Distance Strategy

The distance strategy shapes the structural evidence available to the attribution model and can therefore influence its choice of update targets. Nearest selection concentrates this evidence on a narrow set of differences, potentially favoring changed components over unchanged components that also contribute to failure. Farthest selection broadens the contrast, but may introduce unrelated differences that obscure the connection between component changes and observed behavior. We therefore use middle selection by default, targeting the median distance from the current center to balance contrast coverage and interpretability.

Table 12 compares nearest, middle, and farthest selection on AppWorld with GLM-5.2. These strategies rank neighbors by increasing distance, proximity to the median distance, and decreasing distance, respectively. All other optimization settings remain fixed, including the center-selection rule and contrast-set size. We report TGC and SGC on both test splits, alongside full-training tokens and API calls.

Table 12: Neighbor distance strategies on AppWorld with GLM-5.2, with all other settings fixed. Scores are in %; * marks the default strategy. Training cost covers the full optimization run; tokens are in millions.

Strategy	Test-Normal		Test-Challenge		Training Cost	
	TGC	SGC	TGC	SGC	Tokens (M)	API Calls
Nearest	92.26	87.50	86.33	76.26	109.12	15,201
Middle*	93.45	89.29	86.89	77.94	113.41	14,851
Farthest	90.48	83.93	82.49	71.22	103.11	14,793

D.3 Optimizer Direct Cost Breakdown

Table 13 reports the direct resource footprint of the CATO optimizer, isolating trajectory diagnosis, attribution, gradient consolidation, and candidate generation from environment rollout and validation. We count unique logical requests after format deduplication and retry filtering.

On AppWorld Test-Normal, the optimizer consumes only 2.55M tokens and 158 API calls. This accounts for about 2.25% of full-training tokens (113.41M, with 14,851 training API calls). On WebArena, the optimizer uses 1.17M tokens across 176 API calls. These measurements confirm that structural attribution and candidate generation are lightweight; the primary compute footprint stems from environment execution and validation.

Table 13: Direct cost of CATO's optimizer calls with GLM-5.2, excluding environment rollouts and validation. Test scores are in %.

Benchmark	Optimizer Tokens (M)	API Calls	Test Score
AppWorld Test-Normal	2.55	158	93.45 TGC / 89.29 SGC
WebArena	1.17	176	55.76 SR
BBH (10 subsets)	9.95	993	93.93 SR (macro avg.)

D.4 BBH Per-Subtask Breakdown

Table 14 reports test success rates after independent optimization of each of the ten BBH subtasks on GLM-5.2. The macro average is the unweighted mean of these ten subtask scores. Relative to ReAct, CATO improves six subtasks, ties three, and drops by 0.90 percentage points on Causal Judgement. CATO and GEPA reach 100% on Logical Deduction (five objects).

Table 14: BBH test success rates under GLM-5.2 (%). Each subtask is optimized independently. Macro Average is the unweighted subtask mean. Bold and underlined values mark the best and second-best distinct scores in each row, including ties.

Subtask	ReAct	ProTeGi	DRAFT	MIPROv2	ACE	GEPA	OPTIMAS	**CATO**
Causal Judgement	69.82	<u>72.52</u>	69.82	**73.42**	70.27	67.57	70.27	68.92
Disambiguation QA	60.00	68.00	61.33	73.67	75.67	<u>79.00</u>	74.33	**80.67**
Dyck Languages	71.67	91.67	73.33	**96.00**	72.33	93.67	92.33	<u>95.67</u>
Ruin Names	85.67	90.00	86.67	91.67	89.33	<u>93.00</u>	84.33	**98.00**
Formal Fallacies	98.00	**99.67**	98.33	**99.67**	97.67	**99.67**	<u>99.00</u>	**99.67**
Web of Lies	**99.67**	96.33	<u>99.33</u>	97.67	99.00	99.00	**99.67**	**99.67**
Multistep Arithmetic (two)	<u>98.67</u>	98.33	<u>98.67</u>	98.00	**99.67**	<u>98.67</u>	97.67	<u>98.67</u>
Sports Understanding	87.67	<u>97.67</u>	88.67	84.33	89.00	97.33	95.67	**98.33**
Logical Deduction (five objects)	99.33	<u>99.67</u>	<u>99.67</u>	<u>99.67</u>	<u>99.67</u>	**100.00**	99.33	**100.00**
Navigate	**99.67**	93.67	**99.67**	**99.67**	**99.67**	**99.67**	<u>98.67</u>	**99.67**
Macro Average	87.02	90.75	87.55	91.38	89.23	<u>92.76</u>	91.13	**93.93**

D.5 Four-Component Modular Extension (Skill Component)

To demonstrate that CATO extends beyond three components, we introduce a four-component configuration $\theta = (\theta_p, \theta_t, \theta_e, \theta_s)$. Here, θ_s is an editable skill repository. Each skill defines a reusable task procedure with applicability conditions, dependencies between steps, and completion checks. During execution, it guides the agent through applicable tasks while preserving these dependencies and checking completion. Under the same candidate generation rules and budget B_{cand}, the attribution and budget allocation formulation in Section 3 applies to four components without architectural changes.

Table 15: Three- and four-component CATO on AppWorld. Score shares and TGC/SGC are percentages; training tokens are in millions.

	Score Share (%)				Test-Normal		Test-Challenge		Training Cost	
Method	**Prompt**	**Tools**	**Exper.**	**Skill**	**TGC**	**SGC**	**TGC**	**SGC**	**Tokens (M)**	**API Calls**
CATO (P/T/E)	40.91	27.27	31.82	—	93.45	89.29	86.89	77.94	113.41	14,851
CATO (P/T/E/S)	31.17	12.40	29.70	26.73	93.25	89.88	87.53	78.18	113.62	15,084

In Table 15, four-component CATO allocates 26.73% of the total component score to Skill, improving Test-Normal SGC from 89.29% to 89.88% and Test-Challenge SGC from 77.94% to 78.18% at comparable training cost, while Test-Normal TGC remains virtually unchanged (93.25% vs. 93.45%).

D.6 Convergence Dynamics

We track Test-Normal scores, the total component score, and population updates across ten rounds to examine performance and search activity. At the end of each round, we evaluate all candidates in

the updated population on AppWorld Test-Normal. These test evaluations describe population-level performance only; they do not affect candidate selection or population updates, which use validation scores. Curves show mean task goal completion (TGC) and scenario goal completion (SGC) across population members; shaded bands show their sample standard deviation. The total component score sums the group totals a_g defined in Section 3 and determines demand for candidate generation. We also count generated candidates and new configurations entering the population each round to track proposed and accepted updates.

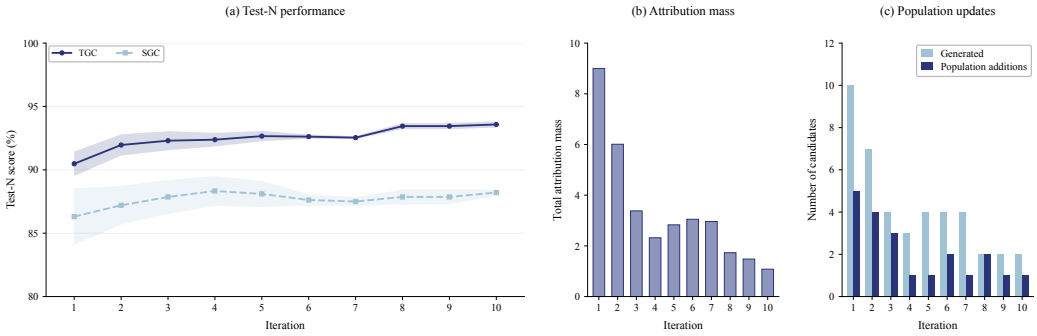


Figure 6: Convergence dynamics over an extended ten-round AppWorld run: (a) Test-Normal scores across the end-of-round population, (b) total component score, and (c) generated candidates and population additions.

Figure 6 illustrates an initial performance gain followed by stabilization, while the total component score and candidate generation generally decline. Later rounds still introduce new configurations into the population, but these updates yield limited changes in mean Test-Normal scores.

E Limitations and Broader Societal Impact

Scope and Boundary Conditions. CATO targets multi-component agents where same-task contrasts localize execution failures. When all population configurations fail on a difficult task, no same-task success is available. In such cases, CATO falls back to failure-only trajectory diagnosis. In addition, each candidate edits one component. When a repair needs changes to several components, CATO applies them one at a time across rounds, so such repairs may take several rounds. Generating coordinated edits to several components is left to future work.

Broader Societal and Environmental Impact. CATO proposes candidate edits only for components assigned by attribution rather than for every module. On AppWorld with GLM-5.2, this selective assignment reduces training token use by 49% relative to All-component assignment. Lower token use may reduce computational demand; energy consumption was not measured. All experiments ran in sandboxed benchmarks with simulated environments and mock APIs, without human subjects or private user data. Because CATO accepts edits based only on task success, an edit could weaken safety-relevant instructions without lowering accuracy. Each accepted edit is an auditable single-component text diff, and we recommend reviewing optimized configurations before deployment.

F Full Prompt Templates

This appendix presents a contract-faithful, reader-oriented rendering of four shared system templates: trajectory analysis, component attribution, component-level textual gradient consolidation, and candidate generation. Scenario and component variants are shown within their shared template rather than as separate templates. The presentation is not a character-for-character code transcription; role names are standardized to CATO. The templates keep the terms used at run time: "parameters" are the configuration components, "Parameter Changes" or "Parameter Difference" is the configuration difference $\Delta\theta^{(r)}$, "case gradient" is the case-level textual gradient $\nabla^{(r)}_{\text{case}}$, "mod-

ule gradient" is the component-level textual gradient $\nabla^{(g)}_{\text{comp}}$, "synthesize" is the consolidation step Consolidate_d, "routing" is the assignment of a failure to a component, and the "Contrastive", "Dual failure", and "Single failure" variants cover contrast pairs, all-failure tasks, and seed-round failures (Section 3).

Reading convention. Each blue listing contains shared instructions, with labeled alternatives indicating the rules or output schema selected for the current scenario or target component. Runtime evidence is supplied in a *separate user message*, summarized beneath each listing. Purple placeholders such as {{query}} denote values filled at runtime; the braces are display notation, not literal required delimiters. Angle-bracket values in output schemas describe types or required content, not input slots.

F.1 Behavior-Level Trajectory Analysis

Listing 4: Trajectory failure analysis with scenario variants.

[ROLE]
You are the CATO trajectory failure analyst. Determine the target execution's supported failure mechanism using only the visible evidence.

[TASK]
Analyze execution behavior only. Identify the earliest visible divergence that can explain the final failure, and state the supported execution-level mechanism before its terminal consequence.

[INPUTS]
Task facts, optional reference answer, terminal status, evaluator feedback, and compact trace views. Each trace preserves step order and the final effective MODEL/ACT evidence; explicit omission markers may truncate verbose content.

[CONSTRAINTS]
No Agent parameters or Parameter Difference are visible. Do not infer hidden content, assign Prompt/Tools/Experience responsibility, or propose a repair. Ground every claim in visible actions, observations, model responses, terminal facts, or evaluator feedback. If the mechanism is not supported, state the evidence boundary instead of guessing.

[SCENARIO RULES — SELECT ONE]

Contrastive. The failed execution is the target; the successful execution for the same task is only a behavioral reference. Do not treat every extra successful step as causal.

Dual failure. The center is the only target. The neighbor is failed comparison evidence, never a successful counterfactual. Mark the mechanism as center-specific, shared, or indistinguishable.

Single failure. No neighbor or successful execution exists. Compare the path only with task requirements, terminal facts, optional reference answer, and evaluator evidence.

[FIELD GUIDE]
`trajectory_contrast`: 1–3 sentences on the key factual path difference or task/evaluator relation.
`fail_cause`: 2–4 sentences giving divergence → consequence → failure.
`key_cues`: 1–5 facts with visible evidence locators.

[OUTPUT FORMAT]
Return JSON only, without Markdown, prose, or extra keys.

```
{
  "trajectory_contrast": <non-empty string>,
  "fail_cause": <non-empty string>,
  "key_cues": [<1 to 5 evidence-located strings>]
}
```

Runtime user message (separate from the system template).

Task: Query = {{query}}; Reference Answer = {{availability and value}}.

Executions 1–N: For each execution, supply {{role}}, {{answer}}, {{terminal status}}, {{evaluation feedback}}, and {{trace}}. The scenario determines whether the evidence contains a failed/successful pair, two failed executions, or one failed execution.

F.2 Component Attribution and Case-Level Textual Gradient

Listing 5: Component attribution with scenario variants.

[ROLE]
You are a parameter attribution analyst. Assess whether visible Parameter Changes explain behavior and which Prompt, Tools, or Experience components own reusable repair.

[TASK]
Identify the supported execution-level failure mechanism. Assess Parameter Changes only as explanations of behavioral difference. Assess each component's repair ownership independently, whether changed or unchanged.

[INPUTS]
Bounded failed-side parameter view; deterministic Parameter Changes for contrastive samples; independent Trajectory Analysis; bounded execution evidence. Selected/discovered/recalled states show exposure, not responsibility. Read-only Tool schema is evidence but is not editable.

[SCORING RULES]
Scores are independent responsibility strengths in $[0, 1]$ and need not sum to one. Use the following evidence levels:
0.80–1.00: direct complete chain; 0.70–0.79: clear but one indirect link; 0.61–0.69: specific uniquely scoped missing guidance; 0.40–0.60: materially ambiguous; 0.01–0.39: weak/indirect; 0.00: no relevant evidence or evidence against responsibility.
Never raise a score merely to create a route. Linked semantic changes outrank unlinked semantic or cosmetic changes. Use equal-low or all-zero scores when neither behavioral linkage nor repair ownership is supported.

[COMPONENT OWNERSHIP]
Tools own Agent-facing callable contracts expressible as description changes. **Experience** owns reusable trigger-conditioned procedures: routing, composition, ordering, interpretation, verification, and recovery. **Prompt** owns unconditional cross-task roles, strategies, protocols, output requirements, and environment-wide constraints. A correct instruction that the Agent violated is not itself proof that the component is defective.

[SCENARIO RULES — SELECT ONE]

Contrastive. The successful execution is a behavioral reference, not a controlled counterfactual. Assess changed-coordinate linkage and failed-side repair independently.

Dual failure. The neighbor only distinguishes shared from center-specific behavior. Different failed behavior is not evidence that the neighbor is correct.

Single failure. No neighbor, successful counterfactual, or Parameter Difference exists. Route only from concrete parameter content linked to visible failure.

For non-contrastive scenarios, `diff_assessment` is null and no successful parameter counterfactual may be invented.

[OUTPUT FORMAT]
Return JSON only. Complete `diff_assessment`, then `repair_ownership`, then scores. The case gradient with three fields is always diagnosed but is propagated only when a component is routed.

```
{
  "diff_assessment": {
    "prompt": {"kind": "semantic|cosmetic|unchanged", "links_to_trajectory": <bool>},
    "tools": {"kind": "semantic|cosmetic|unchanged", "links_to_trajectory": <bool>},
    "experience": {"kind": "semantic|cosmetic|unchanged", "links_to_trajectory": <bool>}
  } | null,
  "repair_ownership": {
    "prompt": <sentence or null>,
    "tools": <sentence or null>,
    "experience": <sentence or null>
  },
  "component_scores": {
    "prompt": <0..1>, "tools": <0..1>, "experience": <0..1>
  },
  "case_gradient": "symptom: ...\ncontrast: ...\nrepair_prior: ..."
}
```

For each `kind`, select one of `semantic`, `cosmetic`, or `unchanged`; vertical bars denote alternatives, not literal output.

Runtime user message (separate from the system template).

Task and diagnosis: {{query}}, {{reference answer}}, and {{trajectory analysis}} from Listing 4.

Attribution evidence: {{parameter changes}} (contrastive only), {{failed-side parameter context}}, and {{execution evidence}}.

F.3 Component-Level Textual Gradient Consolidation

Listing 6: Component-level textual gradient consolidation with component variants.

[ROLE]
You are the module-gradient planner for the target Prompt, Tool description, or Experience component.

[TASK]
Synthesize selected case gradients into one evidence-backed reusable change objective for exactly one component. Identify the unresolved component-level defect, one reusable correction outcome, and its applicability and preservation boundaries. Do not generate the final patch.

[INPUTS]
Current target-component value and the selected routed Case Gradients. The current value is both editable baseline and evidence of behavior already satisfied.

[AGGREGATION CONTRACT]
Use only evidence relevant to the target component; merge duplicate evidence. Resolve conflicts by support strength; keep different causal mechanisms separate. Drop singleton suggestions that are only case answers, entities, URLs, or unique values without a reusable mechanism. Abstract shared case-specific values into role terms while retaining the interface and data relationships that make the mechanism actionable. Address an unresolved weakness rather than repeat guidance the current value already expresses.

[COMPONENT RULES — SELECT ONE]

Prompt. Plan general strategies, constraints, decision rules, or output requirements. Do not encode one case, copy a Tool contract, or create a bounded workflow.

Tools. Plan only description-level contract clarification. Preserve IDs and read-only schema; do not invent capabilities, parameters, defaults, outputs, or side effects.

Experience. Plan exactly one reusable trigger-conditioned workflow with the narrowest domain boundary, decision order, failure boundary, and supported verification condition.

[OUTPUT FORMAT]
Return JSON only. `textual_gradient` is at most 4000 characters and contains exactly three labeled paragraphs. Operation selection is performed by the candidate stage, not by this planner.

```
{
  "textual_gradient": "Diagnosis: <unresolved weakness, consequence, evidence
 pattern>\n\nChange Objective: <one reusable target-component outcome>\n\nBoundaries:
 <applicability, preserved behavior, prohibited generalization>"
}
```

Runtime user message (separate from the system template).

Current component: Supply {{current component value}} in exactly one block, using <AGENT_PROMPT>, <AGENT_TOOLS>, or <AGENT_EXPERIENCE> and its corresponding closing tag.

Selected evidence: Follow the component block with Selected Case Gradient 1–N, each filled with {{selected case gradient}}.

F.4 Single-Component Candidate Generation

Listing 7: Candidate generator with component-specific output schemas.

[ROLE]
You are the target-component candidate generator.

[TASK]
Convert the module-level textual gradient into one complete applicable result for this component.

[INPUTS]
Current editable value, optional explicitly read-only context, required preservation constraints, and textual gradient. Original case gradients and failure trajectories are intentionally unavailable.

[CONSTRAINTS]
Change only the target component and only exposed editable values. Use the smallest change satisfying the objective. Preserve unrelated effective behavior and immutable identity. Do not add case-specific entities, unsupported facts, unrelated improvements, rationale, summary, or a compliance checklist.

[OUTPUT FORMAT — SELECT ONE COMPONENT]
Return JSON only in the schema for the target component.

Prompt candidate. Return the complete changed Prompt. Preserve unrelated text, formatting, literal escapes, and every required placeholder with its exact count.

```
{"prompt": <complete non-empty changed Prompt>}
```

Tool-description candidate. Return the smallest non-empty subset of existing Tool description updates. IDs and schemas remain immutable; each description is a complete supported selection/calling contract.

```
{
  "updates": [
    {"tool_id": <exposed Tool ID>, "description": <complete changed description>}
  ]
}
```

Experience candidate. Return a non-empty `operations` array containing one or more allowed operations, all restricted to the Experience component. Use only the operations needed to satisfy the textual gradient. Each add creates one reusable record; each update returns one complete changed record; each delete is limited to an eligible ID and requires a reason.

```
{"operations": [<operation object>, ...]}
```

Each operation object uses one of the following schemas; operation types may be repeated or combined as needed:

```
{"operation": "add", "description": <text>, "content": <Do/Don't text>}
{"operation": "update", "experience_id": <existing ID>, "description": <text>,
 "content": <Do/Don't text>}
{"operation": "delete", "experience_id": <eligible ID>, "reason": <text>}
```

Runtime user message (separate from the system template).

Editable baseline and objective: {{editable component value}} and {{textual gradient}} from Listing 6.

Conditional fields: Required Placeholders = {{required placeholders}}; Allowed Operations = {{allowed operations}}; Delete Eligible IDs = {{delete eligible IDs}}; Read-only Tool Context = {{read-only tool context}}. Supply these only when applicable to the target component and requested operations.

G Case Studies: Contrastive Attribution and Validation-Task Transfer

This section presents four cases covering the experience bank, system prompt, tool descriptions, and procedural skills. Each case follows same-task execution evidence through attribution, a single-component edit, and application to a separate validation task. All four cases contrast failed and successful executions; the fourth examines Skill refinement in the four-component setting. The transfer tasks belong to the held-out validation set used for candidate selection.

G.1 Case Study 1: Experience Refinement (θ_e)

This case shows when CATO has sufficient evidence to update Experience. The failed execution approves all 26 requests before resolving sender relationships, whereas the successful contrast first resolves `coworker` and `friend` from contact data and then approves only the 18 qualifying requests. This contrast provides evidence for updating the task-level constraint-handling strategy; the resulting Experience-only candidate subsequently transfers the same joint-filtering principle to a separate Spotify task.

Query — Source Task Specification

- **Question**: Accept all pending Venmo payment requests from my coworkers and friends.
- **Success Criterion**: Approve every and only pending payment request whose sender is verified as a coworker or friend. Approving any non-qualifying request or omitting any qualifying request constitutes failure.

Failed Execution — Observed Execution Breakdown

Action Turns 1–7 (Preparation): Authenticate with Venmo and inspect the APIs for retrieving and approving pending requests.

Action Turn 8 (Retrieve Candidate Requests):

- **Tool**: Venmo
- **Sub-goal**: Retrieve all pending incoming payment requests with pagination.
- **Call**: `apis.venmo.show_received_payment_requests(status='pending', page_index=0)`
- **Result**: Retrieved 26 pending requests.

Action Turn 9 (Direct Irreversible Action):

- **Tool**: Venmo
- **Sub-goal**: Approve every candidate request without resolving the relationship constraints.
- **Call**: `for req in pending_requests: apis.venmo.approve_payment_request(...)`
- **Result**: **All 26 requests were approved, including 8 from senders who were neither coworkers nor friends.**

Action Turn 10 (Completion): Call `complete_task()`; the approved set exceeds the task target.
Execution Verifier: **FAIL** – The failed execution approved all 26 pending requests, while the successful execution resolved `coworker` and `friend` and approved only the correct 18.

Successful Contrast — Same-Task Reference Execution

Action Turns 1–5 (Preparation): Authenticate with Venmo and Phone.

Action Turn 6 (Retrieve Requests): Retrieve the same 26 pending requests from Venmo.

Action Turns 7–11 (Contact Inspection): Inspect the Phone APIs and confirm the `coworker` and `friend` relationship categories.

Action Turns 12–13 (Resolve Relationships):

- **Tool**: Phone
- **Sub-goal**: Resolve the coworker and friend contact sets separately.
- **Call**: `search_contacts('coworker'); search_contacts('friend')`
- **Result**: Identified 9 coworkers and 8 friends.

Action Turn 14 (Construct Target Set):

- **Operation**: Jointly filter request senders against the two verified relationship sets.
- **Call**: `sender in coworker_names or sender in friend_names`
- **Result**: Isolated exactly 18 target requests from the original 26.

Action Turn 16 (Targeted Approval):

- **Tool**: Venmo
- **Sub-goal**: Approve only the filtered target set.
- **Call**: `for req in requests_to_accept: approve_payment_request(...)`
- **Result**: **Correctly approved 18 requests.**

Action Turn 17 (Completion): Verify the approved set and call `complete_task()`.
Execution Verifier: **PASS**

Contrastive Structural Attribution — Step-by-Step Walkthrough

Configuration Difference:

- **Prompt** (θ_p): The successful contrast adds semantic instructions to resolve friend or family relationships through `phone.search_contacts`.
- **Tools** (θ_t): No difference; both executions can access the same tools.
- **Experience** (θ_e): The failed configuration has an empty experience bank. The successful configuration adds a reminder to resolve relationship-qualified recipients through Phone contacts and filter app-specific items before acting.

Behavior-Level Diagnosis: The failed execution retrieves 26 pending requests and approves them without resolving sender relationships, including 8 that satisfy neither relationship constraint. The successful execution resolves both relationship sets through Phone, aligns them with Venmo senders, and approves only the correct 18.

Attribution Rationale: The failed source has an empty experience bank, and the recorded case-level textual gradient identifies a missing trigger-conditioned procedure. That procedure resolves relationship constraints through Phone before selecting Venmo requests, placing the proposed repair within Experience's reusable workflow role. The successful Prompt change supplies behavioral evidence for this procedure; it does not require copying that change into Prompt. These scores therefore select a repair target rather than establish Experience as the unique cause of the original failure.

Attribution Component Score ($\eta = 0.6$):

$$s_p = 0.25, \quad s_t = 0.00, \quad s_e = \mathbf{0.65} \quad (\text{Assigned to } \theta_e)$$

Case-Level Textual Gradient $\nabla^{(r)}_{\text{case}}$: *When a task constrains its targets by relationships or other predicates, resolve every constraint from an authoritative source, construct and verify the target set with the joint predicate, and only then perform irreversible actions.*

Targeted Single-Component Candidate — Experience Refinement

- **Parent Configuration**: The configuration that produced the failed trajectory.
- **Assigned Component**: Experience only; Prompt and Tools remain unchanged.
- **Consolidated Group**: The component-level textual gradient consolidates 3 assignments, including the case shown here.
- **Edit Operation**: Add an Experience entry; Prompt and Tools remain unchanged.

Component Diff — **Added Experience Entry** (excerpt; `[...]` marks omitted content):

```
description: Resolve every request qualifier through its authoritative source or
exhaustive search before acting on qualifying items

content: Do: When a request qualifies items by one or more predicates, resolve each
qualifier through its authoritative source or exhaustive search scope before selecting
targets. Apply all qualifiers as jointly required predicates to produce the target set,
then execute the requested action only on items in that verified set. [...]
```

The single-component candidate obtains **52/57** on the held-out validation set with no infrastructure failures and enters the next population.

Validation-Task Transfer — Spotify Task

- **Validation Query**: *Download all the songs from my Spotify playlists that I have liked.*
- **Parent Baseline**: Misreads the constraint as "liked playlists," collects and downloads 43 songs, and fails evaluation.

Action Turns 1–2 (Preparation): Authenticate with Spotify and confirm the relevant APIs.

Action Turns 3–5 (Collect Playlist Songs):

- **Tool**: Spotify
- **Sub-goal**: Traverse 6 playlists and aggregate their songs.
- **Call**: `show_playlist_library(...); show_playlist(...)`
- **Result**: Collected 42 unique playlist songs.

Action Turn 7 (Collect Liked Songs):

- **Tool**: Spotify
- **Call**: `show_liked_songs(...)`
- **Result**: Retrieved 37 liked songs.

Action Turn 8 (Apply Joint Predicate):

- **Operation**: Intersect playlist songs with liked songs.
- **Call**: `playlist_song_ids.intersection(liked_song_ids)`
- **Result**: Identified 22 songs satisfying both constraints.

Action Turns 10–11 (Execute and Recover):

- **Tool**: Spotify
- **Sub-goal**: Download each intersected song and treat `already downloaded` as already satisfied.
- **Call**: `try: download_song(...); except already_downloaded: continue`
- **Result**: All 22 target songs were handled correctly.

Transfer: The candidate no longer interprets the constraint as "liked playlists" and instead explicitly constructs the intersection of the two sets; this behavioral change turns the validation task from failure to success.

G.2 Case Study 2: System Prompt Alignment (θ_p)

This case shows when CATO should update the Prompt that governs global behavior. The failed execution emits explanatory prose and a malformed, non-executable navigation action before task completion, whereas the successful contrast continuously produces browser actions that the environment can execute and completes the task. CATO therefore generates a Prompt-only global constraint and repairs the same missing-next-action failure on a separate GitLab issue-creation task.

Query — Source Task Specification

- **Question**: Like all submissions created by `ThetaGang_wsb` in the `wallstreetbets` forum.
- **Success Criterion**: Ensure that every submission authored by `ThetaGang_wsb` in the `wallstreetbets` forum is in the liked state, while leaving non-qualifying submissions unchanged.

Failed Execution — Observed Execution Breakdown

Action Turns 1–2 (Initial Navigation): Repeatedly attempt the current-page entry point without reaching `wallstreetbets`.

Action Turn 3 (Malformed Navigation):

- **Tool**: Browser
- **Sub-goal**: Navigate directly to the target forum.
- **Call**: `<goto><url>http://.../f/wallstreetbets</url>`
- **Result**: **The output mixes explanatory prose with an unsupported action format, and the run terminates with `agent_failure`.**

Execution Verifier: **FAIL** – The failed execution emits a malformed `goto` and terminates; the successful execution continues with executable actions, reaches the target forum, and verifies both target submissions.

Successful Contrast — Same-Task Reference Execution

Action Turns 1–15 (Navigation Preparation): Continue submitting parseable `click` actions and advance from each updated page state.

Action Turn 16 (Executable Navigation):

- **Tool**: Browser
- **Sub-goal**: Enter the target forum with a supported action format.
- **Call**: `goto('http://.../f/wallstreetbets')`
- **Result**: Successfully opened `wallstreetbets`.

Action Turns 17–22 (Inspect Target Submissions): Open `ThetaGang_wsb`'s submission list and inspect the complete target set.

Action Turn 23 (Verify and Complete):

- **Tool**: Browser
- **Sub-goal**: Verify the complete target set and report completion.
- **Call**: `send_msg_to_user('{"task_type":"MUTATE","status":"SUCCESS",...}')`
- **Result**: **Confirmed that both target submissions were already liked.**

Execution Verifier: **PASS**

Contrastive Structural Attribution — Step-by-Step Walkthrough

Configuration Difference:

- **Prompt** (θ_p): The successful execution adds an unconditional pre-completion output contract: every step must be a supported browser action executable by the environment.
- **Tools** (θ_t): No difference; both executions use the same browser-action set.
- **Experience** (θ_e): No decisive difference related to action-output formatting.

Behavior-Level Diagnosis: Before task completion, the failed execution emits explanatory prose and expresses `goto` with unsupported nested tags. Action parsing fails and the run terminates. The successful execution continues to emit executable browser actions, including a valid `goto(...)`, inspects all target submissions, and reports success.
Attribution Rationale: The missing rule governs every pre-completion response, so its scope belongs to the system prompt's global output contract. The browser actions themselves are available and usable; the failure concerns how the agent formats its next action.
Attribution Component Score ($\eta = 0.6$):

$$s_p = \mathbf{0.88}, \quad s_t = 0.00, \quad s_e = 0.00 \quad \text{(Assigned to } \theta_p\text{)}$$

Case-Level Textual Gradient $\nabla^{(r)}_{\text{case}}$: *Establish a global Prompt invariant: before task completion, every step must contain only a supported-format browser action that the environment can execute; prohibit prose, explanations, and malformed action-like text, and allow result reporting only after completion.*

Targeted Single-Component Candidate — Prompt Refinement

- **Parent Configuration**: The configuration that produced the failed trajectory.
- **Assigned Component**: Prompt only; Tools and Experience remain unchanged.
- **Consolidated Group**: The component-level textual gradient consolidates 2 assignments, including the case shown here.
- **Edit Operation**: Add the following Prompt text; Tools and Experience remain unchanged.

Component Diff — Added Prompt Text:

```
* Use report_infeasible only when the task cannot be completed with the available
actions.

Before the task is complete, every step must be an executable browser action expressed
in the supported action format. Prose, explanations, and malformed or non-executable
action-like text are prohibited before completion; emit only well-formed actions that
the environment can execute so the next step can be taken. After task completion, you
may report the final result as required.
```

The single-component candidate improves held-out validation accuracy from **6/33** to **7/33** with no infrastructure failures and enters the final population.

Validation-Task Transfer — GitLab Task

- **Validation Query**: *Open a new issue to discuss the implementation of default plugins for .zsh for the current project. Use an appropriate issue title.*
- **Parent Baseline**: After reaching the New Issue form, emits a prose-only response; the environment returns `Model returned an invalid browser action`, and evaluation fails.

Action Turns 1–2 (Open New Issue Form):

- **Tool**: Browser
- **Sub-goal**: Open the current project's Issues page and then New Issue.
- **Call**: `click('328'); click('601')`
- **Result**: Opened the New Issue form.

Action Turns 3–4 (Fill Issue):
- **Tool**: Browser
- **Sub-goal**: Fill a title and discussion text about default `.zsh` plugins.
- **Call**: `fill('551', title); fill('634', description)`
- **Result**: Filled both fields successfully.

Action Turn 5 (Create Issue):
- **Tool**: Browser
- **Call**: `click('799')`
- **Result**: Submitted the form and reached the new issue's detail page.

Action Turn 6 (Verify and Complete):
- **Tool**: Browser
- **Call**: `send_msg_to_user(success_payload)`
- **Result**: Verified that the issue was created, and the task passed evaluation.

Transfer: The parent substitutes a prose-only response for the next action on the same form and fails. The Prompt-only candidate continues with executable `fill` and `click` actions and completes the validation task.

G.3 Case Study 3: Tool Description Disambiguation (θ_t)

This case shows when CATO should modify the description of a single callable in Tools. The failed and successful executions both complete the requested Spotify deletions correctly; the decisive difference occurs only at the terminal call, where the failed execution supplies a descriptive `answer` for an action-only task while the successful execution omits the argument under an explicit tool contract. CATO accordingly updates the `complete_task` description and transfers the same invocation boundary to a separate Venmo reminder task.

Query — Source Task Specification

- **Question**: Remove all songs from my Spotify song library and playlists that were released before 2021 year.
- **Success Criterion**: Remove every song released before 2021 from both the Spotify song library and all playlists, while leaving songs released in or after 2021 unchanged.

Failed Execution — Observed Execution Breakdown

Action Turns 1–7 (Preparation): Authenticate with Spotify and inspect the song-library, playlist, and removal APIs.

Action Turns 8–10 (Clean Song Library):
- **Tool**: Spotify
- **Sub-goal**: Identify and remove songs released before 2021 from the song library.
- **Call**: `show_song_library(...); show_song(...); remove_song_from_library(...)`
- **Result**: Correctly removed 5 target songs.

Action Turns 11–13 (Clean Playlists):
- **Tool**: Spotify
- **Sub-goal**: Traverse playlists and remove songs released before 2021.
- **Call**: `show_playlist_library(...); show_song(...); remove_song_from_playlist(...)`
- **Result**: Correctly removed 9 target songs.

Action Turns 14–15 (Verify State): Read back the song library and playlists and confirm that no pre-2021 songs remain.

Action Turn 16 (Contract Violation):

- **Tool**: Supervisor
- **Sub-goal**: Submit completion with a descriptive answer.
- **Call**: `complete_task(answer="Successfully removed...")`
- **Result**: **The underlying operations are correct, but the action-only evaluator rejects the extra `answer`.**

Execution Verifier: **FAIL** – Both executions correctly remove 14 pre-2021 songs; the failed execution supplies a descriptive `answer`, while the successful execution omits it under the tool contract.

Successful Contrast — Same-Task Reference Execution

Action Turns 1–9 (Preparation): Authenticate and inspect the same Spotify APIs.

Action Turns 10–12 (Clean Song Library): Identify and remove the same 5 target songs from the library.

Action Turns 13–15 (Clean Playlists): Identify and remove the same 9 target songs from the playlists.

Action Turn 16 (Contract-Aligned Completion):

- **Tool**: Supervisor
- **Sub-goal**: Submit completion under the action-only tool contract.
- **Call**: `complete_task()`
- **Result**: **The `answer` argument is omitted, and the task passes evaluation.**

Execution Verifier: **PASS**

Contrastive Structural Attribution — Step-by-Step Walkthrough

Configuration Difference:

- **Prompt** (θ_p): Other non-decisive differences are present; both executions complete the same Spotify operations.
- **Tools** (θ_t): The failed execution's `complete_task` description implies that completion always requires an `answer`; the successful execution explicitly distinguishes question-type from action-only tasks.
- **Experience** (θ_e): No difference related to the terminal submission error.

Behavior-Level Diagnosis: The agent correctly removes 5 library songs and 9 playlist songs and verifies the resulting state, but supplies a descriptive `answer` to `complete_task` in an action-only task. The successful execution performs the same operations and omits `answer`; the two executions show that the intent is sound and the missing boundary lies in how the callable description presents the argument contract.

Attribution Rationale: Both executions complete the requested operations, and their decisive difference is the argument supplied to one terminal callable. The repair therefore belongs to that tool's description, which must distinguish question-type and action-only tasks without changing its schema.

Attribution Component Score ($\eta = 0.6$):

$$s_p = 0.00, \quad s_t = \mathbf{0.80}, \quad s_e = 0.00 \quad \text{(Assigned to } \theta_t\text{)}$$

Case-Level Textual Gradient $\nabla^{(r)}_{\text{case}}$: *Modify only the `supervisor.complete_task` description: provide a concise `answer` for question-type tasks, omit `answer` for action-only tasks, and leave the schema, defaults, and all other tools unchanged.*

Targeted Single-Component Candidate — Tool Refinement

- **Parent Configuration**: The configuration that produced the failed trajectory.
- **Assigned Component**: Tools only; Prompt and Experience remain unchanged.
- **Consolidated Group**: The component-level textual gradient consolidates 2 assignments, including the case shown here.
- **Edit Operation**: Update the `complete_task` tool description only.

Component Diff — `supervisor.complete_task` description:

Before (Removed):

```
Mark the currently active task as complete with the given answer.
```

After (Added):

```
Mark the currently active task as complete. Provide the answer parameter only for
question-type tasks that request a specific value such as a count, yes/no, or entity;
for action-only tasks that require only execution of operations, omit the answer and
leave it at its default.
```

The single-component candidate obtains **50/57** on the held-out validation set with no infrastructure failures and enters the next population.

Validation-Task Transfer — Venmo Task

- **Validation Query**: *Send a reminder on Venmo for all my payment requests to my roommates which have not been approved or denied for 30 or more days.*
- **Parent Baseline**: Sends all 7 Venmo reminders and then supplies a narrative action summary as `answer`; evaluation fails.

Action Turns 1–8 (Resolve Roommates):

- **Tool**: Phone
- **Sub-goal**: Authenticate and resolve the roommate contact set.
- **Call**: `search_contacts(relationship='roommate')`
- **Result**: Retrieved the roommate set.

Action Turns 9–12 (Construct Target Set):

- **Tool**: Venmo
- **Sub-goal**: Retrieve pending requests and select those sent to roommates at least 30 days ago.
- **Call**: `show_sent_payment_requests(...); age_days >= 30 and recipient in roommates`
- **Result**: Identified 7 target requests.

Action Turn 14 (Send Reminders):

- **Tool**: Venmo
- **Call**: `for req in requests_to_remind: remind_payment_request(...)`
- **Result**: Sent all 7 reminders successfully.

Action Turn 15 (Contract-Aligned Completion):

- **Tool**: Supervisor
- **Call**: `complete_task()`
- **Result**: Omitted `answer` under the updated tool description, and the task passed evaluation.

Transfer: The parent supplies a narrative `answer` after completing the reminders. The Tools-only candidate follows the updated `complete_task` description and omits the argument, turning the same type of validation task from failure to success.

G.4 Case Study 4: Skill Refinement (θ_s)

This four-component case identifies a missing task procedure for replying to a message using information stored in notes. The failed execution sends an unfiltered movie list, whereas the successful contrast recovers the sender's request before selecting titles. The Skill-only candidate organizes request recovery, source filtering, recipient checks, and completion verification into a reusable procedure. A separate validation task tests this procedure with a different sender, director constraint, and source note.

Query — Source Task Specification

- **Question**: Leslie has asked for my movie recommendations via phone text message. Reply to them with a list of comma-separated movie titles from my Simple Note account as per their request.
- **Success Criterion**: Read Leslie's original text request, select movies from Simple Note that match the requested director, and reply to the same person with comma-separated titles. Do not send unrelated titles or modify the notes.

Failed Execution — Request Constraints Omitted

Action Turns 1–8 (Preparation): Inspect Phone and Simple Note APIs, authenticate, locate Leslie, and prepare to search notes.

Action Turns 9–11 (Read Source Without Request):

- **Tool**: Simple Note
- **Sub-goal**: Read movie recommendations without first retrieving Leslie's text message.
- **Call**: `simple_note.search_notes(query='movie', ...); simple_note.show_note(note_id=2705, ...)`
- **Result**: Retrieve recommendations spanning several directors without recovering the request's Peter Jackson constraint.

Action Turn 12 (Unfiltered Extraction):

- **Operation**: Extract and deduplicate movie titles without filtering by director.
- **Call**: Parse note content and collect all distinct titles.
- **Result**: Construct a list of 19 titles, including nonmatching movies such as No Country for Old Men and Forrest Gump.

Action Turns 14–15 (Send and Complete):

- **Tool**: Phone and Supervisor
- **Call**: `phone.send_text_message(message=movie_list, ...); supervisor.complete_task()`
- **Result**: The message is sent successfully, but its contents include movies outside the requested director's works.

Execution Verifier: **FAIL** – The reply includes nonmatching titles, such as No Country for Old Men; successful delivery does not satisfy the request.

Successful Contrast — Request-Grounded Reply

Action Turns 1–7 (Preparation): Inspect message-retrieval APIs, authenticate with Phone, and prepare to resolve Leslie's contact details.

Action Turns 8–9 (Recover the Request):

- **Tool**: Phone
- **Sub-goal**: Resolve Leslie and read the incoming request before selecting recommendations.
- **Call**: `phone.search_contacts(query='Leslie', ...); phone.search_text_messages(phone_number=recipient, ...)`
- **Result**: Recover the request: "Do you have any recommendations for a movie from Peter Jackson?"

Action Turns 13–16 (Read and Filter):

- **Tool**: Simple Note
- **Sub-goal**: Apply the recovered director constraint to the movie recommendations.
- **Call**: `simple_note.search_notes(query='movie', ...); simple_note.show_note(note_id=2705, ...)`
- **Result**: Retain Peter Jackson's three The Lord of the Rings movies and exclude works by other directors.

Action Turns 17–18 (Send the Matching Set):

- **Tool**: Phone and Supervisor
- **Call**: `phone.send_text_message(message=filtered_titles, ...); supervisor.complete_task()`

- **Result**: Send The Lord of the Rings: The Two Towers, The Lord of the Rings: The Fellowship of the Ring, and The Lord of the Rings: The Return of the King.

Execution Verifier: **PASS** – The execution provides a same-task reference for recovering the request, filtering source entries, and replying to the requester.

Contrastive Structural Attribution — Missing Task Procedure

Configuration Difference:

- **Prompt** (θ_p): No difference; both executions use the same prompt.
- **Tools** (θ_t): The executions use different descriptions of `supervisor.complete_task`, including guidance on when to omit `answer`.
- **Experience** (θ_e): No difference; both executions use the empty experience bank.
- **Skill** (θ_s): No difference; both skill repositories are empty.

Behavior-Level Contrast: Both executions find a contact, read a note, and send a message; they diverge before constructing the reply. The failed execution copies titles directly, whereas the successful execution first recovers the director constraint and then filters the same note. The final API call is not the source of the content mismatch.

Attribution Rationale: The repair target is a complete procedure with applicability conditions, step dependencies, and completion checks. Request recovery must precede selection; filtering must preserve item attributes; recipient and output checks must precede sending. Skill stores this complete procedure, while the narrower reminder to read the message first can also belong to Experience.

Attribution Component Scores ($\eta = 0.6$):

$$s_p = 0.00, \quad s_t = 0.05, \quad s_e = 0.55, \quad s_s = \mathbf{0.85} \quad (\text{Assigned to } \theta_s)$$

Case-Level Textual Gradient $\nabla^{(r)}_{\text{case}}$: *Create a reusable Skill procedure for note-based message replies: recover the sender's request before selecting items, preserve source attributes while filtering by the recovered conditions, send only matching items to the requester, and verify the recipient and output before completion.*

Targeted Single-Component Candidate — Skill Refinement

- **Parent Configuration**: The configuration that produced the failed execution, with an empty skill repository.
- **Assigned Component**: Skill only; Prompt, Tools, and Experience retain their parent values.
- **Contrast Evidence**: The failed execution and same-task successful reference motivate a dependency from request recovery to final reply construction.
- **Edit Operation**: Add one Skill, abstracting the director, contact, and movie titles into conditions recovered at runtime.

Component Diff — Added Skill Entry:

```
name: reply-to-message-request-from-notes

description: Reply to a phone text request using matching items from Simple Note.
Recover the sender's constraints, filter note entries, and send only the requested
content to the same person.

## Preconditions
- The task asks you to reply to a phone text request using information stored in Simple
Note.
- The current instruction refers to the sender's request rather than specifying every
selection condition.
- Phone message search, note search/detail, and text-message sending APIs are available.

## Procedure
1. Resolve the recipient through contacts or message metadata; retain the verified phone
number.
2. Retrieve the relevant incoming request with phone.search_text_messages. Match both
the sender and the requested topic; inspect further pages when the relevant request is
not yet found. An outgoing acknowledgment is not the request.
3. Extract the source, item type, inclusion/exclusion conditions, output fields, and
reply format before composing. If the request cannot be recovered, stop without sending.
4. Search Simple Note and read the source with simple_note.show_note. Preserve item
attributes, including each movie title's director, for filtering.
```

```
5. Apply the recovered conditions to every candidate entry. Deduplicate matching titles,
preserve their wording, and exclude items whose attributes do not establish a match. Use
the condition from the message, never a hardcoded person, director, or title list.
6. Build the reply in the requested format. Before sending, check that every output item
comes from the note, every output item satisfies the request, no matching distinct item
was omitted, and the recipient matches the request sender.
7. Send the checked message once with phone.send_text_message. Confirm the send
response, then call supervisor.complete_task() without an answer for this action task.
Do not modify the source notes or contacts.

## Completion Criteria
- The request sender and the recipient are the same person.
- The reply contains all and only the distinct matching titles from the source note, in
the requested format.
- The send API confirms success; source notes and contacts are unchanged.
- No reply is sent solely from an unfiltered note list when the referenced request has
not been read.
```

Validation Contract: With a different sender or director, recover the request before filtering and sending. Verify task success and check that no source-task titles are reused as fixed answers.

Validation-Task Transfer — New Sender and Constraint

- **Validation Query**: *Laura has asked for my movie recommendations via phone text message. Reply to them with a list of comma-separated movie titles from my Simple Note account as per their request.*
- **Skill Selection**: The task requires recovering a request from a message, filtering note entries, and returning the matching set to the requester.
- **Parent Baseline**: Reads the movie note without checking Laura's request and sends all 21 distinct titles, including the nonmatching Interstellar; evaluation fails.

Action Turns 1–6 (Preparation): Inspect Phone and message-retrieval APIs, obtain account details, and authenticate with Phone.

Action Turn 7 (Recover a New Constraint):

- **Tool**: Phone
- **Sub-goal**: Identify Laura's incoming request from the message sender and contents.
- **Call**: `phone.search_text_messages(query='movie', ...)`
- **Result**: Recover the Quentin Tarantino constraint rather than reusing Peter Jackson from the source task.

Action Turns 11–14 (Filter a Different Note):

- **Tool**: Simple Note
- **Sub-goal**: Read the validation task's movie note and apply the recovered director constraint.
- **Call**: `simple_note.search_notes(query='movie', ...); simple_note.show_note(note_id=1556, ...)`
- **Result**: Select Django Unchained and Pulp Fiction, excluding Interstellar and other nonmatching titles.

Action Turns 15–16 (Send and Complete):

- **Tool**: Phone and Supervisor
- **Sub-goal**: Reply with the matching titles in the requested format.
- **Call**: `phone.send_text_message(message='Django Unchained, Pulp Fiction', ...); supervisor.complete_task()`
- **Result**: The reply is sent successfully, and the task passes evaluation.

Execution Verifier: **PASS**
Transfer Check: The procedure succeeds with a different contact, director, and note; Skill-level retention requires further evaluation.