CANTANTE: Optimizing Agentic Systems via Contrastive Credit Attribution

Tom Zehle
University of Freiburg
ELLIS Institute Tübingen
tom.zehle@tue.ellis.eu

Abstract

LLM-based multi-agent systems have demonstrated strong performance across complex real-world tasks, such as software engineering, predictive modeling, and retrieval-augmented generation. Yet automating their configuration remains a structural challenge, as scores are available only at the system level, whereas the parameters governing agent behavior are local. We argue that optimizing these systems is fundamentally a credit-assignment problem. We therefore introduce CANTANTE, a framework that decomposes system-level rewards into per-agent update signals by contrasting rollouts of multiple joint configurations on the same query. We instantiate it for prompt optimization, treating agent prompts as learnable system parameters. We evaluate CANTANTE against GEPA and MIPROv2 on programming (MBPP), mathematical reasoning (GSM8K), and multi-hop question answering (HotpotQA). Across these benchmarks, CANTANTE achieves the best average rank among all evaluated optimizers and consistently outperforms unoptimized prompts. It improves over the strongest baseline by $+18.9$ percentage points on MBPP and $+12.5$ percentage points on GSM8K, while incurring a lower inference cost. It remains within one standard deviation of the strongest baseline on HotpotQA. Crucially, our credit correlation analysis confirms that the attributer produces meaningful per-agent signals rather than echoing the global system score.

1 Introduction

Agentic systems built on large language models have demonstrated strong empirical performance across a range of complex, real-world tasks, from autonomous software engineering [1] to end-to-end machine learning pipelines [2] and multi-hop retrieval [3]. More broadly, multi-agent systems (MAS) turn LLMs from passive conversational models into autonomous problem-solving systems. Yet realizing this potential in practice remains labor-intensive: each deployment requires manual selection of agents, hand-authored prompts, workflow design, and tool setup.

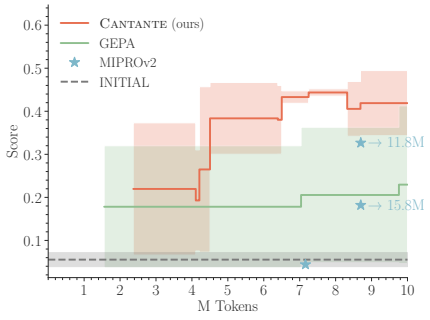


Figure 1: Trajectories on MBPP. CANTANTE improves steadily, reaching the highest final accuracy.

Manually tuning MAS configurations, however, contradicts the main paradigm of machine learning: Just as no practitioner hand-tunes neural network weights, manual iteration over agent prompts and roles is a problem that should be solved via optimization. The manual labor required grows combinatorially with each agent requiring its own configuration and each prompt requiring a co-adaption to the behavior of the others, making trial-and-error tuning both tedious and ineffective. Automating the configuration

of MAS would reduce this burden and enable principled adaptation to new tasks and models, but doing so is non-trivial. Not only is the search space high-dimensional, but more fundamentally, the optimization problem has a structural mismatch: agent-level parameters are *local*, while evaluation feedback is *global*, available only as a scalar signal over the full workflow output.

We argue that *multi-agent system optimization is fundamentally a credit-assignment problem*: global feedback must be converted into per-agent update signals. In a planner–coder–evaluator workflow, for example, faulty code introduced by the coder should not trigger equally negative updates in the planner and evaluator. This problem is well-established in multi-agent reinforcement learning, where cooperative agents must each receive a learning signal derived from a shared global reward [4, 5]. Existing approaches to LLM-based MAS optimization largely sidestep this issue by propagating global scores as update signals to individual agents [6, 7].

We therefore introduce CANTANTE, a novel framework for node-level optimization in MAS based on system-level rewards and reasoning trajectories. The central mechanism is *contrastive in-group attribution*: by comparing rollouts of multiple joint configurations for the same query, the attribution model isolates each agent's parameterization's contribution to the observed differences in outcomes. This method is independent of the choice of local optimizer. We instantiate this framework for prompt optimization, treating agent prompts as learnable system parameters rather than fixed implementation details.

We evaluate CANTANTE on workflow graphs with non-trivial structure, including conditional edges, ensemble nodes, and tool-using agents, across programming (MBPP), mathematical reasoning (GSM8K), and multi-hop question answering (HotpotQA). CANTANTE obtains the best average rank among all evaluated optimizers, with the largest gains occurring on MBPP and GSM8K, where CANTANTE improves over the strongest baseline by 18.9 percentage points (pp) and 12.5 pp, respectively. The MBPP trajectory plot in Figure 1 further shows that CANTANTE reaches these gains steadily over optimization, rather than through a single late-stage jump. We show that these gains are achieved without substantially increasing inference-time token cost relative to the unoptimized prompts. On HotpotQA, CANTANTE remains within a standard deviation of the best baseline and improves over the unoptimized initial configuration. Finally, our credit correlation analysis suggests that the attributer produces meaningful per-agent signals rather than merely echoing the global system score and that these signals can serve as a diagnostic tool for MAS topology design.

Contributions. We make the following contributions: I) We formalize the optimization of LLM-based multi-agent systems as a credit-assignment problem and introduce CANTANTE. II) We instantiate CANTANTE for prompt optimization, treating agent prompts as learnable system parameters, and evaluate it against MAS-capable prompt optimizers across multiple benchmarks. III) We isolate contrastive attribution itself — rather than the underlying local optimizer — as the principal driver of CANTANTE's gains via an identity-attribution ablation, and further demonstrate robustness to attributer prompt, attributer model, local optimizer choice, and dataset size. IV) We release the full implementation of CANTANTE to support adoption by practitioners and to enable further research on attribution-driven MAS optimization, available at `https://github.com/finitearth/cantante`.

2 Related Work

LLM-Based Multi-Agent Systems. Multi-agent systems consist of multiple autonomous agents that coordinate their actions to solve a shared task [8]. LLM-based variants extend this paradigm through a language interface, composing agents via role specialization, communication, planning, memory, and tool use [9]. Representative applications include autonomous software engineering with SWEAgent [1], automated predictive modeling with MLZero [2], and agent-based retrieval with MAIN-RAG [3].

Credit Assignment in Multi-Agent Reinforcement Learning. In multi-agent reinforcement learning (MARL), credit assignment under shared rewards is a well-studied problem. Methods such as QMIX [10] decompose the global value function into per-agent contributions, while counterfactual approaches like COMA [5] estimate individual agent impact by comparing observed outcomes against baselines that marginalize out each agent's action. More recently, LLM-MCA [11] utilizes LLM critics to decompose global rewards into individual feedback. However, it remains tied to updating the neural network weights of traditional reinforcement learning agents, and it relies on a task-specific

attribution prompt. It adapts the comparative intuition behind counterfactual credit assignment to LLM-based workflows by comparing concrete alternative trajectories for the same query and using a prompted LLM to attribute observed differences in outcomes to individual agents. Unlike value-based decomposition methods such as QMIX, CANTANTE does not impose a constraint requiring credits to reconstruct the global score.

Prompt Optimization. Prompt design affects model behavior, with surface-level formatting choices alone producing substantial performance variation [12]. EvoPrompt [13] performs evolutionary search over prompt candidates using LLMs in its mutation and selection operators. CAPO [14] builds on evolutionary search by incorporating AutoML techniques such as racing to reduce optimization cost, and introduces prompt length penalties. TextGrad [15] simulates gradient descent via LLM feedback, iteratively identifying and correcting failure cases in generated outputs. MIPROv2 [16] samples prompt candidates and selects among them via Bayesian optimization, treating prompt search as a structured hyperparameter problem. GEPA [6] frames prompt optimization as a multi-objective problem over a Pareto front, where each dataset sample constitutes its own objective, and uses LLM-generated critiques of execution traces to guide prompt updates. These methods operate at the level of individual agents and can be applied to the CANTANTE framework by utilizing them as local optimizers.

Optimization of Multi-Agent Systems. MIPROv2 in its multi-agent setting treats the joint prompt configuration as a single monolithic parameter, without decomposing the reward signal. GEPA approaches multi-agent configurations by updating agents sequentially via round-robin, optimizing one agent per step while holding all others fixed. Unlike these approaches, CANTANTE decouples attribution from the update mechanism, enabling the use of arbitrary, sophisticated local optimizers. ADAS [17] uses an optimizer LLM to iteratively propose and evaluate modifications to the MAS topology, while AFLOW [18] formulates workflow optimization as Monte Carlo Tree Search over system configurations. MASS [7] interleaves prompt and topology optimization but propagates the global reward without agent-level attribution. More recent closed-source attribution-based works like HiveMind [19] and MAPRO [20] do isolate contributions, but couple their credit assignment to built-in LLM-oracle mutations. HiveMind relies on Shapley-based attribution, which requires per-agent credits to reconstruct the global score; CANTANTE does not impose this decomposition. Furthermore, CANTANTE drives node-level optimization through contrastive attribution grounded in reasoning traces across multiple joint rollouts, updates all agents simultaneously at each step, and leaves workflow topology fixed.

3 Contrastive Attribution via In-Group Comparison

We formalize the optimization of a multi-agent system as a parameter optimization problem over joint agent configurations. We first present the general problem setting, then introduce CANTANTE and its contrastive attribution mechanism, and finally describe its instantiation for prompt optimization.

3.1 Problem Formulation

Agents. A multi-agent system (MAS) consists of N agents $\{A_a\}_{a=1}^{N}$. Each agent A_a is a parameterized function mapping an input string to an observable output string,

$$y_a = A_a(x_a;\, \theta_a), \tag{1}$$

where $\theta_a \in \Theta_a$ denotes the local parameter set searched by a per-agent local optimizer O_a. The input x_a may be a task query or the output of an upstream agent, which is injected into the agent's prompt. Agent behavior is primarily governed by a prompt p_a specifying role and instructions; additional local parameters could include, but are not limited to, model weights, model choice, or decoding settings. Agents may interact with external tools, incorporating tool outputs into their computation, and may produce internal reasoning that is not part of the extracted output y_a.

Multi-Agent System. The MAS is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = N$ agents as vertices. Each edge $(u \to v) \in \mathcal{E}$ carries the variables passed from agent A_u to agent A_v, and the graph supports structured workflow designs including conditional routing, parallel execution, and ensembling. A *rollout* is a single execution of $\mathcal{G}$ on a query q; the resulting *trajectory* ξ is the ordered sequence

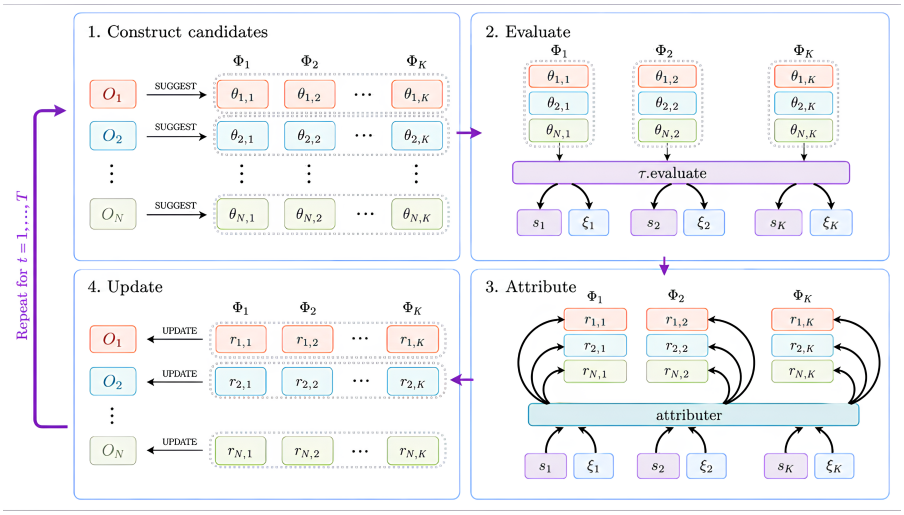


Figure 2: **Overview of CANTANTE. (1)** At each iteration, every local optimizer O_a proposes K candidate parameterizations $\theta_{a,i}$, yielding K joint configurations. **(2)** Each joint configuration is evaluated on task τ, producing system-level scores s_i and an execution trace ξ_i per configuration. **(3)** The attributer receives the full set of scores and traces and performs contrastive attribution within random comparison groups, estimating aggregated credit $r_{a,i}$. **(4)** Each parameterization's credit is forwarded to its local optimizer O_a to perform local update steps.

of all agent outputs y_a produced during that execution. Given a query q and a joint configuration $\Phi = (\theta_1, \ldots, \theta_N)$, the graph produces a prediction $Y = \mathcal{G}(q;\, \Phi)$.

Optimization Objective. Each task τ is associated with a scoring function $S_\tau(Y)$ that maps a prediction to a global system score, e.g., a reward signal, human ranking, or accuracy in classification settings. In this work, we focus on optimizing the local parameters Φ, treating the workflow graph $(\mathcal{V}, \mathcal{E})$ as a fixed upstream design choice. The objective is to find the joint configuration maximizing expected score over the data distribution $\mathcal{D}$:

$$\arg\max_{\Phi} \; \mathbb{E}_{q\sim\mathcal{D}}[S_\tau(\mathcal{G}(q;\, \Phi))]\,. \tag{2}$$

In practice, optimization proceeds over a fixed query set Q:

$$\arg\max_{\Phi} \; \frac{1}{|Q|} \sum_{q\in Q} S_\tau(\mathcal{G}(q;\, \Phi))\,. \tag{3}$$

This optimization is subject to a finite budget, which can, for example, be defined in terms of tokens consumed by the downstream LLMs.

The Credit Assignment Problem. The reward signal S_τ is defined at the system level, yet the parameters θ_a that determine agent behavior are local. Directly optimizing Equation 3 over the joint space Φ is intractable for large N, as it requires exhaustive evaluation of numerous configurations.

3.2 CANTANTE

CANTANTE (**C**ontrastive **A**ttributio**n** for **T**uning of Multi-A**g**e**n**t Syst**e**ms) makes multi-agent systems learnable by decomposing the joint optimization over Φ into N parallel local optimization problems, using contrastive attribution. We offer a visual overview in Figure 2, and detail its pseudo-algorithm in Appendix C.

Assigning credit in absolute terms is difficult: judging whether an agent performed well requires knowledge of what constitutes good behavior for every task, which is not generally available. Relative attribution within a comparison group is more tractable: We utilize an attribution model that, for each

agent, independently assesses how well its output contributed to the observed system outcome by comparing it with a group of contrastive rollouts.

This contrastive framing offers an additional advantage in degenerate cases where all configurations in a group achieve identical system scores. In these cases, a purely score-based optimizer would stall because the absence of score differences eliminates the signal to guide optimization. The attribution LLM, instead, can fall back on assessing intermediate agent outputs y_a directly. As a result, CANTANTE can still distinguish between configurations with identical system-level scores.

The attributer is a prompted LLM that receives a group of trajectories and their system scores and outputs a scalar credit estimate $c_a \in [-1, +1]$ per agent per rollout, where positive credit indicates that the agent's behavior was relatively supportive of high system performance and negative credit indicates the converse. Concretely, the attributer receives the query q, the per-agent outputs $\{y_a\}_{a=1}^{N}$ extracted from each trajectory $\xi_{q,i}$, and the system scores $S_\tau(Y)$. When an agent is invoked multiple times within a rollout, we use its final produced output as y_a. We assume the attributer to be instruction-following and non-adversarial, consistent with its prompt design, which includes a few-shot example grounding its behavior; the full prompt is provided in Appendix D.2.

At each iteration, the local optimizers propose K candidate parameterizations per agent, which are assembled into K joint configurations $\{\Phi_i\}_{i=1}^{K}$ (Figure 2, Panel 1). Exhaustively evaluating all K^N cross-agent combinations would be prohibitively expensive. Instead, each joint configuration pairs one candidate from each agent, requiring only K rollouts per query. The contrastive attributer then estimates contributions per agent parameterization despite the jointly varying co-parameterizations. The resulting joint configurations are evaluated on task τ, producing system scores s_i, and execution traces ξ_i (Figure 2, Panel 2).

In the attribution stage (Figure 2, Panel 3), the attributer assesses each agent's contribution relative to the other configurations in its comparison group, yielding per-query credits $c_{q,a,i} \in [-1, +1]$. We decide to assign scalar credits rather than ranks, enabling more nuanced attribution. The signed range allows the attributer to express both positive and negative contributions while remaining within an interpretable, normalized range. We aggregate these credits into a per-agent-and-configuration score by averaging uniformly over the query set, mirroring the per-query weighting of the empirical objective in Equation 3: $r_{a,i} = \frac{1}{|Q|} \sum_{q \in Q} c_{q,a,i}$.

Finally, each local optimizer O_a consumes the resulting parameter-credit pairs $\{(\theta_{a,i}, r_{a,i})\}_{i=1}^{K}$ (Figure 2, Panel 4) and proposes updated parameterizations for the next iteration.

4 Experiments

We instantiate CANTANTE for prompt optimization, treating agent prompts p_a as learnable parameters and using CAPO as the default local optimizer O_a. We evaluate CANTANTE against GEPA and MIPROv2 on three benchmarks, spanning code generation (MBPP), mathematical reasoning (GSM8K), and multi-hop question answering (HotpotQA), each paired with a distinct workflow graph. We additionally analyze credit attribution behavior and conduct experiments to broaden understanding of the method.

4.1 Setup

Benchmarks and Graphs. We score MBPP [21] as the proportion of programming tasks for which the system produces code that passes all test cases, using a conditional workflow with a planner, executor, and validator agent, with tool access for code execution. GSM8K [22] is evaluated by exact match and uses an ensembling graph in which three parallel executor agents produce predictions that a consensus agent aggregates into a final answer. HotpotQA [23] is scored by exact match and uses an agentic retrieval-augmented generation graph with a retriever, reader, synthesizer, and hallucination checker. These benchmarks were chosen to cover three qualitatively different MAS structures: conditional execution, parallel ensembling, and retrieval-augmented multi-hop reasoning. Full experimental details, including implementation details, parameterizations, benchmark descriptions, as well as respective topologies, hardware, and model details, are provided in Appendix B.

Models and Optimizers. All downstream task agents use Qwen3 with 30 billion total parameters, 3 billion of which are active. GPT-OSS-120B, as a model from a distinct family, serves as both an

Table 1: Test accuracy (%) averaged over three seeds (±std). **Bold: best**, underlined: second-best per benchmark. Average rank computed across benchmarks and seeds. The bottom rows show CANTANTE's absolute gain over the initial prompt and the best baseline.

Optimizer	MBPP	GSM8K	HotpotQA	Avg Rank
Initial	5.54 ± 1.62	59.20 ±10.73	9.67 ± 3.00	3.44
GEPA	22.96 ±18.30	61.27 ± 2.66	10.93 ± 3.91	2.67
MIPROv2	18.42 ±14.13	69.80 ± 7.10	**14.20** ± 5.72	2.33
CANTANTE (ours)	**41.89** ± 7.56	**82.33** ± 4.35	11.93 ± 5.06	**1.44**
Δ vs. Initial	36.35	23.13	2.27	-2.00
Δ vs. Best	18.93	12.53	-2.27	-0.89

optimizer and an attribution model. We evaluate CANTANTE against GEPA [6] and MIPROv2 [16], representing two dominant paradigms of MAS-capable prompt optimization.

Protocol. To ensure a fair starting point, all initial prompts are generated automatically by prompting GPT-OSS-120B, as detailed in Appendix D.1. We report the initial prompts in Appendix D.3. All optimizers operate under a shared optimization budget of 10 million tokens, which includes input and output tokens from downstream task-agent calls, as well as optimizer and attribution calls. The reported score is the development-set-selected configuration at the last optimization step completed within budget. On MBPP seeds 42 and 47, MIPROv2 did not return a configuration within the shared 10M-token budget. We extended its budget on these seeds to 15.8M and 11.8M tokens, respectively, favoring MIPROv2 in the comparison. All results are reported as mean and standard deviation across three seeds using *Bessel's correction*.

4.2 Main Results

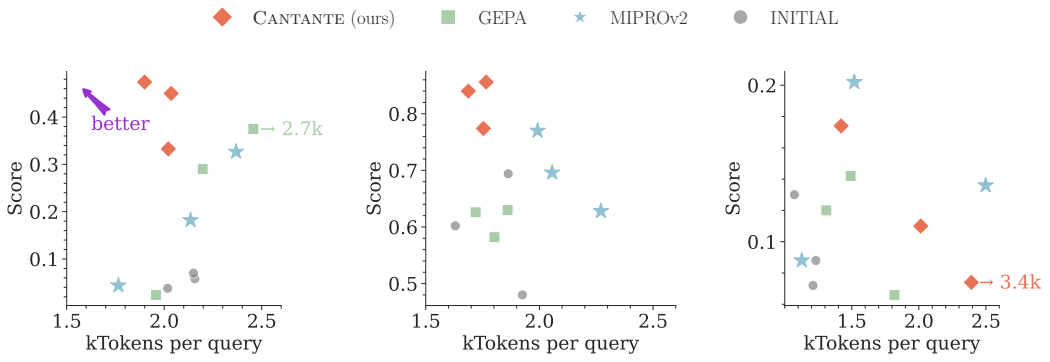
Table 1 reports the main results. Across the nine benchmark-seed combinations, CANTANTE obtains the highest test score in 6 cases and achieves the best average rank of 1.44, compared to 2.33 for the strongest baseline MIPROv2. CANTANTE outperforms the unoptimized initial prompts on 8 of 9 seeds, with gains on MBPP and GSM8K that substantially exceed the cross-seed standard deviations. The highest-scoring prompt sets obtained by CANTANTE are provided in Appendix D.4.

On MBPP, CANTANTE achieves a mean accuracy of 41.89 %, improving over the strongest baseline by 18.93 percentage points (pp) and over the unoptimized initial prompts by 36.35 pp. Beyond mean performance, CANTANTE exhibits substantially lower standard deviation than the baselines (± 7.56 vs. ± 14–18 pp). The relatively low variance across seeds indicates more stable optimization behavior, suggesting that per-agent attribution provides a more consistent update signal than system-level feedback alone.

On GSM8K, CANTANTE achieves 82.33 %, improving over MIPROv2 by 12.53 pp and over GEPA by 21.06 pp. Its variance is lower than MIPROv2's but higher than GEPA's (± 4.35 vs. ± 7.10 and ± 2.66 pp, respectively). The high variance of the initial prompts (± 10.73 pp) indicates that GSM8K performance is particularly sensitive to prompt quality. Notably, GEPA fails to meaningfully improve over the initial prompts, plateauing near the baseline performance.

On HotpotQA, performance differences across all optimizers and for unoptimized prompts are small (ranging from 9.67 % to 14.20 %) relative to cross-seed standard deviations (ranging from 3.00 % to 5.72 %), suggesting that prompt optimization has limited leverage on this benchmark regardless of the method. Within this constrained regime, CANTANTE remains within one standard deviation of the strongest baseline and improves over the unoptimized prompts.

Figure 3 compares evaluated optimizers in terms of inference cost and achieved score. We distinguish the optimization budget from the evaluation-time inference cost. The former counts all input and output tokens consumed during optimization, including those consumed by downstream, optimizer, and attribution calls. The latter measures the tokens consumed (both input and output) by the final selected MAS when evaluated on the held-out test set, averaged over queries. On MBPP and GSM8K, CANTANTE achieves the strongest accuracy while simultaneously requiring the fewest mean evaluation-time tokens per inference (1.99 and 1.74 thousand tokens per invocation), beating



(a) MBPP (b) GSM8K (c) HotpotQA

Figure 3: Accuracy vs. evaluation-time inference cost across benchmarks, where inference cost denotes mean token usage per query for the final selected prompt sets.

the token efficiency of the unoptimized initial prompts (2.11 and 1.81), as well as MIPROv2 (2.09 and 2.11) and GEPA (2.28 and 1.79), demonstrating that the accuracy gains on these datasets are not an artifact of expensive reasoning traces, but rather a direct result of role-specific parameter updates. In contrast, on the retrieval-augmented HotpotQA benchmark, CANTANTE favors a more expansive configuration, using substantially more tokens per inference (2.28 vs. 1.17 for the initial prompts, 1.71 for MIPROv2, and 1.54 for GEPA). This increased cost reflects a regime where the optimizer attempts to resolve complex multi-hop reasoning through more detailed prompting, yet remains constrained by underlying topological bottlenecks that limit further performance or efficiency gains. A detailed breakdown of token usage across optimizers and benchmarks is provided in Appendix A.1.

4.3 Analysis of Attributed Credits

To further analyze our method's behavior, we examine the credits assigned in the main experiments by computing the Spearman correlation ρ between the system score for a joint configuration and the attributions to the respective agent prompts received during optimization. Figure 4 reports this analysis for all three benchmarks, and Appendix A.2 provides the per-seed values.

On MBPP, the validator receives the highest and most stable attribution across seeds, suggesting that it is a particularly relevant target for optimization. The planner and coder show substantially weaker correlations. This suggests that, within this topology, many prompt variants are sufficient for these roles, or the validator's final judgment constrains their outputs.

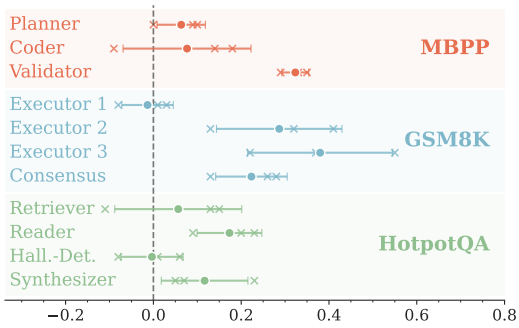


Figure 4: Spearman correlation between attribution credits and system scores, per agent and benchmark. Dots denote mean; crosses per-seed values; whiskers standard deviation.

On GSM8K, an ordering emerges across the three executor agents despite their identical tasks: Executor 1 receives near-zero attribution, while Executors 2 and 3 carry progressively stronger signals. This pattern is consistent with recency bias documented by [24] and [25]: because executor outputs appear in a fixed order in the consensus prompt, later outputs, especially that of Executor 3, are more likely to be attended to and therefore receive stronger attribution. The consensus agent receives moderate attribution, as expected from its role in aggregating the executor outputs.

On HotpotQA, the hallucination detector receives near-zero attribution across seeds, and the retriever straddles zero, while the reader and synthesizer show moderate positive signal. Although these magnitudes are not far below some agents in MBPP and GSM8K, the pattern is less sharply differentiated: no single agent dominates the correlation analysis. This may indicate that structural bottlenecks limit the extent to which improvements in any individual prompt affect the final system score. Alternatively, it may reflect a limitation of the attributer, which may have struggled to identify meaningful per-agent contributions in this setting.

4.4 Additional Experiments

We analyze the sensitivity of our method to key design choices and hyperparameters by conducting additional experiments on GSM8K, ablating the attributer components, alternating the local optimizer, and testing sensitivity to the group-size parameter and to varying dataset sizes. Table 2 reports the performance differences against the default setting from the main experiments.

To isolate the impact of the attributer, we replaced it with an "identity attributer" that propagates the global system score directly to the local optimizer. Under equal optimization steps, this removal causes a severe 13.4 pp drop in accuracy, highlighting contrastive attribution as the primary driver of optimization efficiency. Because the identity attributer bypasses LLM attribution costs, we also evaluated an "equal budget" setting that allows the local optimizer to execute substantially more optimization steps. Given this extended budget, the identity attributer alone achieves 79.6 %, which is roughly 9.8 pp higher than the strongest baseline (MIPROv2). Yet, default CANTANTE still outperforms this budget-advantaged run by 4.4 pp. This breakdown clarifies that while a strong local optimizer sets a high performance floor, per-agent attribution yields a superior learning signal than global rewards.

Table 2: Ablation results on GSM8K (Seed 42). Test accuracy relative to the default setting, in percentage points.

Configuration	**Δ Score**
Default setting	84.00
Identity Attribution	
Equal Steps	− 13.40
Equal Budget	− 4.40
Attribution component	
alt. Prompt	+ 0.60
alt. Model	+ 0.20
Prompt optimizer	
w/ Few-Shots	− 9.20
alt. Optimizer	+ 3.20
Group size	
g = 2	− 54.40
g = 5	+ 0.80
Dataset size	
|DS| = 75	− 2.80
|DS| = 150	− 3.40

Exchanging the attributer prompt or the LLM underlying the optimization and attribution model (Qwen3-Next-80B instead of GPT-OSS-120B) yields scores of 84.6 % and 84.2 % against the default of 84.0 %, which is within the cross-seed standard deviation of the original experiment. This suggests that the method is robust against concrete choices of prompt and LLMs.

Prior work suggests performance gains from introducing few-shot examples into prompt optimization [14, 26]. However, bootstrapping these from initial rollouts caused a 9.2 pp drop in test accuracy. This performance decay suggests that, when the initial agent configuration is weak, its rollouts provide poor demonstrations of the intended agent behavior—reusing these traces as exemplars biases later prompt candidates toward the same flawed reasoning patterns. The credit correlation analysis in Appendix A.2 further shows that introducing few-shot examples causes the attributer to assign elevated and less differentiated credit-score correlations across agents, suggesting that the resulting agent behaviors become less agent-specific.

To test the generalizability of CANTANTE, we initialized it with EvoPrompt as a local optimizer and observed a 3.2 pp increase in performance. This suggests that our method successfully extends to other local optimizers and that performance gains cannot be attributed solely to CAPO.

To understand the impact of the group-size hyperparameter g, which controls the number of configurations within a contrastive comparison group, we vary it around the default $g = 3$.[1] At $g = 2$, each group contains only a single pair of configurations. This binary contrast yields a severe performance drop of 54.4 pp. The credit correlation analysis in Appendix A.2 shows that the aggregate attribution direction is largely preserved at $g = 2$, indicating the collapse is driven by elevated per-query credit variance rather than systematic misdirection — a single pair of rollouts might lack the overlapping variation needed to disentangle individual agent contributions reliably. The 0.8 pp gain from $g = 5$ over the default indicates diminishing returns for larger group sizes, supporting the premise that $g = 3$ provides an effective trade-off between attribution accuracy and sample efficiency.

Finally, even when reducing the development set by half or three-quarters, CANTANTE retains most of its performance. With 150 and 75 samples, it achieves 80.6 % and 81.2 %, corresponding to modest drops of only 3.4 pp and 2.8 pp relative to the default 84.0 % setting. This suggests that CANTANTE extracts useful credit signals from a few examples and remains effective in low-data regimes.

[1] We additionally increase the number of parameterizations per agent from 9 to 10 to ensure divisibility between K and g.

4.5 Discussion

CANTANTE achieves the best average rank across benchmarks, outperforming the strongest baseline by up to 18.9 pp; on MBPP and GSM8K, these gains are obtained while improving over the token efficiency of the initial prompts, showing that improvements are not driven by increased test-time compute.

The credit correlation analysis shows that the attributer does not merely echo system scores. These patterns can help diagnose MAS topology by indicating which agents are associated with performance gains. On HotpotQA, weaker correlations suggest that prompt optimization has limited leverage, due to topological weaknesses of the respective MAS.

The identity attributer ablation further shows that global-score propagation remains at a comparable level to MIPROv2 under an equal-steps setting, but remains 13.4 pp below CANTANTE. Together with the robustness to the attributer prompt and model choice, this suggests that the central mechanism driving performance is the in-group comparison structure. We also find that CANTANTE generalizes to other local optimizers and remains effective under substantially reduced development sets.

5 Limitations and Future Work

Limitations. CANTANTE holds the workflow graph topology fixed throughout optimization, treating the number of agents, edge structure, and tool assignments as upstream design choices. Performance, therefore, depends on the quality of this initial graph design. The attribution model is a prompted LLM; consequently, attribution quality is bounded by the capabilities of the underlying model. In settings where the attributer is weaker than the downstream agents, credit estimates may become unreliable. The use of an LLM as an attributer model incurs additional optimization budget, which might be allocated to additional optimization steps. Finally, as the number of agents grows, the attributer receives increasingly long trajectories, which may exceed context limits or degrade attribution quality due to information overload. Our experiments cover topologies of up to four agents; scalability to substantially larger systems remains empirically untested. Many practical agentic workflows, however, use a moderate number of agents, and CANTANTE remains directly applicable in this regime.

Future Work. The most natural extension is to lift the fixed-graph assumption, using the credit correlations analyzed in Section 4.3 as a signal for topology optimization: agents whose credits show consistently low correlation with system performance may indicate structural bottlenecks warranting redesign. Extending the local optimizer interface to model weights or decoding configurations would allow CANTANTE to operate as a general MAS training framework beyond prompt optimization. A further direction is to revisit few-shot prompt optimization under warm-started settings, where demonstrations are bootstrapped only after the system has reached a sufficiently reliable initial configuration. Finally, incorporating richer supervision sources, such as execution traces or human preference signals, can meaningfully guide attribution quality.

6 Conclusion

We argued that optimizing LLM-based multi-agent systems is fundamentally a credit-assignment problem, a challenge well recognized in MARL but underexplored in MAS. We introduced CANTANTE, a contrastive attribution framework that converts system-level reward into per-agent update signals. CANTANTE achieves the best average rank and, on two of three benchmarks, while matching or improving the token efficiency of the unoptimized prompts. The credit correlation analysis suggests that the attributer produces meaningful per-agent signals rather than echoing global scores. We see this work as evidence that contrastive attribution is a more effective optimization primitive for LLM-based MAS than global score propagation, and a step toward treating multi-agent system parameters as objects of machine learning rather than manual configuration.

Broader Impact. CANTANTE reduces the manual effort required to configure MAS, improving efficiency and lowering deployment barriers. As a method for improving MAS, it inherits its dual-use risks, including misuse for disinformation or harmful automation, but does not differentially enable such applications beyond the underlying models.

Acknowledgments

Tom Zehle received funding by the European Union. This work was supported by the European Union's Horizon Europe research and innovation program under grant agreement No. 101214398 (ELLIOT).

References

[1] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[2] Haoyang Fang, Boran Han, Nick Erickson, Xiyuan Zhang, Su Zhou, Anirudh Dagar, Jiani Zhang, Ali Caner Turkmen, Cuixiong Hu, Huzefa Rangwala, Ying Nian Wu, Bernie Wang, and George Karypis. MLZero: A multi-agent system for end-to-end machine learning automation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[3] Chia-Yuan Chang, Zhimeng Jiang, Vineeth Rakesh, Menghai Pan, Chin-Chia Michael Yeh, Guanchu Wang, Mingzhi Hu, Zhichao Xu, Yan Zheng, Mahashweta Das, and Na Zou. MAIN-RAG: Multi-agent filtering retrieval-augmented generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2607–2622, Vienna, Austria, July 2025. Association for Computational Linguistics.

[4] David H. Wolpert and Kagan Tumer. Optimal payoff functions for members of collectives. *Advances in Complex Systems*, 4(2–3):265–279, 2001.

[5] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

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

[7] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö. Arık. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv:2502.02533 [cs.LG]*, 2025.

[8] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[9] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv:2402.01680 [cs.AI]*, 2024.

[10] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *Journal of Machine Learning Research*, 21(178):1–51, 2020.

[11] Kartik Nagpal, Dayi Dong, and Negar Mehr. Leveraging large language models for effective and explainable multi-agent credit assignment. In *Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '25, pages 1501–1510, Richland, SC, 2025. International Foundation for Autonomous Agents and Multiagent Systems.

[12] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How I learned to start worrying about prompt formatting. In *The Twelfth International Conference on Learning Representations*, 2024.

[13] 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 *The Twelfth International Conference on Learning Representations*, 2024.

[14] Tom Zehle, Moritz Schlager, Timo Heiß, and Matthias Feurer. CAPO: Cost-aware prompt optimization. In *4th International Conference on Automated Machine Learning*, 2025.

[15] 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.

[16] 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*, pages 9340–9366, 2024.

[17] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025.

[18] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

[19] Yihan Xia, Taotao Wang, Shengli Zhang, Zhangyuhua Weng, Bin Cao, and Soung Chang Liew. HiveMind: Contribution-guided online prompt optimization of LLM multi-agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 29767–29774, 2026.

[20] Zheyuan Zhang, Lin Ge, Hongjiang Li, Weicheng Zhu, Chuxu Zhang, and Yanfang Ye. MAPRO: Recasting multi-agent prompt optimization as maximum a posteriori inference. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 4458–4480, 2026.

[21] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv:2108.07732 [cs.PL]*, 2021.

[22] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv:2110.14168 [cs.LG]*, 2021.

[23] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, 2018.

[24] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[25] Tom Zehle and Matthias Aßenmacher. Can calibration of positional encodings enhance long context utilization? In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 2268–2280, 2026.

[26] Xingchen Wan, Ruoxi Sun, Hootan Nakhost, and Sercan Ö. Arık. Teach better or show smarter? On instructions and exemplars in automatic prompt optimization. *Advances in Neural Information Processing Systems*, 37:58174–58244, 2024.

[27] Tom Zehle, Timo Heiß, Moritz Schlager, Matthias Aßenmacher, and Matthias Feurer. promptolution: A unified, modular framework for prompt optimization. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 282–296, 2026.

[28] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, 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 *The Twelfth International Conference on Learning Representations*, 2024.

[29] LangChain AI. LangGraph, 2024. URL `https://github.com/langchain-ai/langgraph`.

[30] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv:2505.09388 [cs.CL]*, 2025.

[31] OpenAI. gpt-oss-120b & gpt-oss-20b model card. *arXiv:2508.10925 [cs.CL]*, 2025.

[32] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

A Extended Results

A.1 Token Breakdown

We provide a detailed breakdown of the inference cost for the optimized prompts, in thousands of tokens per invocation. Table 3 reports the token usage of the optimizers on the main experiments in Section 4.1, aggregated over seeds with mean and standard deviation, using Bessel's correction.

Table 4 reports the fraction of token budget utilized by optimization and attribution LLMs per optimization method, for the main experiments in Section 4.1.

Table 5 reports the token usage for the further experiments in Section 4.4.

Table 3: Evaluation-time token usage on the held-out test sets. We report the mean number of input and output tokens per inference, in thousands, with standard deviation across three seeds using Bessel's correction. **Bold** and underlined values indicate the **lowest** and second-lowest mean token usage for each benchmark, respectively.

Optimizer	MBPP	GSM8K	HotpotQA	Avg Rank
Initial	2.11 ± 0.08	1.81 ± 0.16	**1.17** ± 0.09	**2.11**
GEPA	2.28 ± 0.36	1.79 ± 0.07	1.54 ± 0.26	2.78
MIPROv2	2.09 ± 0.30	2.11 ± 0.15	1.71 ± 0.71	3.00
CANTANTE (ours)	**1.99** ± 0.07	**1.74** ± 0.04	2.28 ± 1.02	2.11

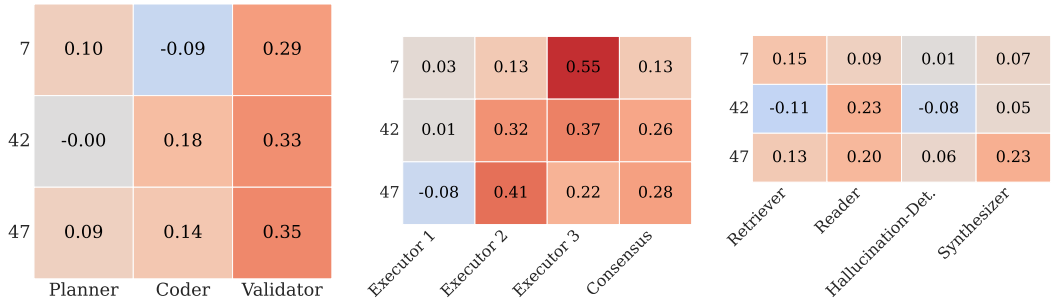
Table 4: Fraction of optimization tokens spent on meta (attribution and optimization) calls, as a percentage of total tokens used up to the token budget. Values are averaged across seeds.

Optimizer	MBPP	GSM8K	HotpotQA
GEPA	24.74 ± 1.56	24.19 ± 5.17	23.41 ± 2.15
MIPROv2	0.45 ± 0.18	0.82 ± 0.06	0.66 ± 0.09
CANTANTE (ours)	25.91 ± 1.49	25.55 ± 1.48	20.11 ± 2.25

Table 5: Evaluation token usage per MAS invocation for ablation configurations on GSM8K, seed 42. Values are average evaluation tokens divided per query, in thousands.

Configuration	Tokens/inv (k)
Default setting	1.69
Attribution Prompt	1.63
Attribution Model	1.97
w/ Few-Shot Examples	2.44
|DS| = 75	1.72
|DS| = 150	1.66
|G| = 2	1.96
|G| = 5	1.65
Identity Attribution (Equal Budget)	1.66
Identity Attribution (Equal Steps)	1.66
PO = EvoPrompt	1.83

A.2 Further Analysis of Credit Attribution



(a) MBPP (b) GSM8K (c) HotpotQA

Figure 5: Spearman correlation (ρ) between attribution credits and system scores, per agent and seed.

Figure 5 provides the per-seed Spearman correlations underlying the aggregated analysis in Section 4.3.

Figure 6 extends the credit correlation analysis of Section 4.4 to the ablation configurations. Across most settings – including an alternative attribution prompt, alternative attribution model, and varying dataset sizes – the correlation structure remains consistent with the default: Executor 1 receives near-zero credit correlation while Executors 2 and 3 carry the dominant signal, reflecting the positional bias discussed in Section 4.3. This stability supports the robustness of the attribution mechanism to concrete design choices. The most notable deviation occurs in the w/ Few-Shots configuration, where all four agents show elevated and roughly uniform correlations (0.32-0.65). This flattening of the per-agent signal indicates that the attributer reverts to echoing the global system score rather than isolating individual contributions — providing a potential explanation for the 9.20 pp accuracy drop reported in Table 2. For $g = 2$, the aggregate correlation structure is largely preserved relative to the default, with Executor 1 remaining near zero. This suggests the severe performance degradation at $g = 2$ is not caused by systematic misdirection of credits, but is instead consistent with elevated per-query credit variance. With only two rollouts available for comparison, individual credit estimates become unreliable even when their average direction is maintained.

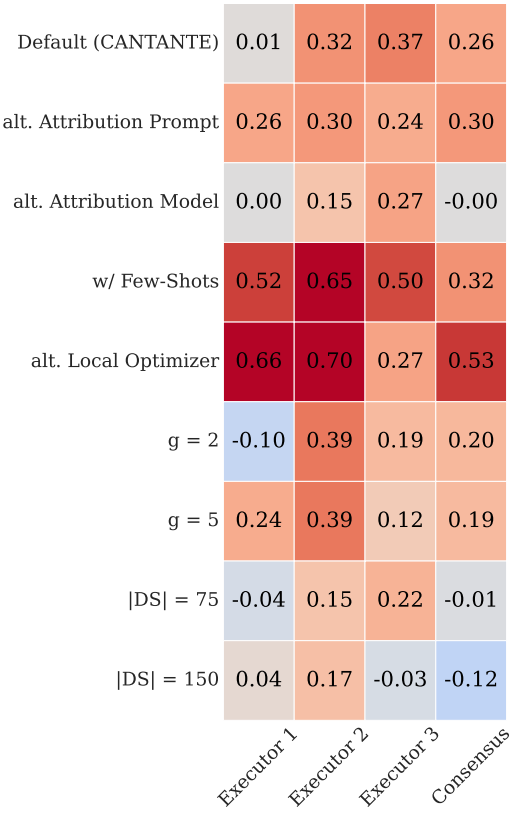


Figure 6: Spearman correlation between attribution credits and system scores for the additional experiments, on GSM8K, seed 42.

B Experimental Details

B.1 Implementation Details

We implement CAPO and EvoPrompt using promptolution [27] and MIPROv2 and GEPA using DSPy [28]. Agent architectures are constructed with LangGraph [29]. Experiments are conducted on NVIDIA L40 GPUs. The main experiments took roughly 100 GPU-hours to finish, and the ablation studies 30, totaling 130 GPU-hours.

B.2 Parameterizations

We use the default parameters of the respective implementations where possible. To control for computational budget, we apply the following modifications. For MIPROv2, we reduce the number of trials to 10 and set the number of candidates to 4. For GEPA, we set the maximum number of metric calls to 150 as suggested in the documentation, with optimization allowed to continue in increments of 150 per step. For CAPO and EvoPrompt, we reduce the population size to 6 and the number of offspring to 3; additionally, for CAPO, we reduce the block size to 10 and the maximum number of block evaluations to 5. In the group-size ablation, the number of offspring is increased to 4 to ensure the divisibility of the group and configuration sizes. For LLM inference, we use a temperature of 0.7 for the downstream model and 0.1 for the optimizer LLM. Maximum token lengths for the downstream model are set to 512 for GSM8K and HotpotQA, and 1024 for MBPP. The optimizer model operates with a context window of 4096 tokens, extended to 8192 in the group size ablation to accommodate five rollouts within the attributer's context.

B.3 Benchmark Details

B.3.1 Seeding

For each benchmark, we sample a development set of 300 instances and an evaluation set of 500 instances from the respective official splits. The random seed controls LLM generation, the train-test split sampling, the sampling of initial prompts for the optimizer, and the random partitioning of agents into groups within CANTANTE. We report results across three seeds (7, 42, 47). Note that LLM inference is not fully replicable due to hardware-level non-determinism.

B.3.2 Scoring and Topologies

MBPP [21] is a code generation benchmark consisting of crowdsourced Python programming tasks, each accompanied by a natural-language description and a set of unit tests. Following the standard evaluation protocol, a task is scored as 1 if the system produces code that passes all hidden test cases upon termination of the multi-agent loop, and 0 otherwise; the reported metric is the fraction of tasks solved on the held-out test split. During execution, the validator agent has access to one of the three test cases per query; the remaining two are held out for scoring and are not observed by any agent at any point during the rollout.

The workflow graph consists of three agents arranged in a conditional loop. A *planner* agent receives the natural language task description and produces a step-by-step implementation plan. The *executor* agent has access to a code execution tool that enables it to write, run, and modify Python code within the rollout. The *validator* agent evaluates the generated code against the visible test case and, when needed, returns targeted feedback to the executor for repair. If no bug is detected or the maximum number of repair iterations is reached, the loop terminates, and the final code is submitted for scoring; we set this maximum to 3.

GSM8K [22] is a benchmark of grade-school mathematical reasoning problems requiring multi-step arithmetic and algebraic reasoning, scored by exact match against the ground-truth numerical answer.

The workflow graph implements a parallel ensembling strategy. Three *executor* agents independently receive the query, and each produces a numerical prediction together with a step-by-step justification. A *consensus* agent receives all three predictions and justifications and produces the final answer by aggregating the evidence. No agent in this graph has access to external tools; all computation is performed via language model inference.

HotpotQA [23] is a multi-hop question-answering benchmark that requires synthesizing information from multiple supporting documents and is scored by exact match against the ground-truth answer string. The benchmark is designed to probe robustness to distractor passages and to test multi-step reasoning over retrieved evidence.

The workflow graph implements an agentic retrieval-augmented generation pipeline. A *retriever* agent receives the query and issues targeted queries to a document store via a retrieval tool, returning a set of relevant passages. A *reader* agent receives the retrieved passages and extracts the facts most

relevant to the query. A *synthesizer* agent receives the extracted facts and produces a candidate answer. A *hallucination checker* agent receives the candidate answer together with the supporting passages and verifies factual consistency; if a hallucination is detected, it generates corrective feedback and conditionally routes execution back to the reader agent, forming a verification loop. If no hallucination is detected or if the maximum number of verification iterations is reached, the candidate answer is submitted for scoring; we set this maximum to 3. The retriever agent has access to a document retrieval tool; all other agents operate via language model inference only.

B.4 Model Details

We evaluate all optimizers using three language models: Qwen3-30B-A3B, Qwen3-Next-80B-A3B [30], and GPT-OSS-120B [31]. All three are mixture-of-experts models and are used as released, without additional quantization. Models can reason within the agentic loop, as the workflow requires them to produce structured outputs with explicit output tags, yielding their respective intermediate outputs y_a. All models are served via an API of a compute cluster using vLLM version 0.17 [32].

C Pseudo Algorithm of CANTANTE

Algorithm 1 provides the pseudo code of CANTANTE, summarizing the method described in Section 3.

Algorithm 1 CANTANTE (Contrastive Attribution for Tuning of Multi-Agent Systems)

Require: Number of agents $\boldsymbol{N}$; local optimizers $\{\boldsymbol{O_a}\}_{a=1}^{N}$; scorable task $\boldsymbol{\tau}$, with queries $\boldsymbol{Q} = \{\boldsymbol{q}_i\}_{i=1}^{n}$; number of joint configurations $\boldsymbol{K}$; group size $\boldsymbol{g}$; attributer ATTRIBUTER; number of iterations $\boldsymbol{T}$;
Ensure: Updated optimizers $\{\boldsymbol{O_a}\}_{a=1}^{N}$

$t \leftarrow 0$
while $t < T$ **do**
  for $(a, i) \in \{1, \dots, N\} \times \{1, \dots, K\}$ **do**
    $\theta_{a,i} \leftarrow O_a$.SUGGEST()
  for $i = 1$ **to** K **do**
    $\Phi_i \leftarrow (\theta_{1,i}, \dots, \theta_{N,i})$ ▷ *Construct candidate joint configurations*
  for $(i, q) \in \{1, \dots, K\} \times Q$ **do**
    $(\xi_{q,i},\ s_{q,i}) \leftarrow \tau$.EVALUATE$(q, \Phi_i)$ ▷ *Evaluate joint configurations*
  for each $q \in Q$ **do**
    Randomly partition $\{1, \dots, K\}$ into groups of size g
    for each group G in the partition **do** ▷ *Perform contrastive attribution*
      $\{c_{q,a,i}\}_{a,\,i} \leftarrow$ ATTRIBUTER.ATTRIBUTE$(q,\ \{(\xi_{q,i}, s_{q,i})\}_{i \in G})$
  for $(a, i) \in \{1, \dots, N\} \times \{1, \dots, K\}$ **do**
    $r_{a,i} \leftarrow \dfrac{1}{|Q|} \displaystyle\sum_{q \in Q} c_{q,a,i}$ ▷ *Aggregate credits per agent across queries*
  for $a = 1$ **to** N **do**
    O_a.UPDATE$\big(\{(\theta_{a,i},\ r_{a,i})\}_{i=1}^{K}\big)$
  $t \leftarrow t + 1$
return $\{O_a$.SUGGEST$()\}_{a=1}^{N}$

D Prompts

All prompts utilized in this work, including the full prompt sets for every optimizer, seed, and ablation study, as well as the alternative attribution prompt utilized in the additional experiments, are available in the supplementary material of this paper at `https://github.com/finitearth/cantante`.

D.1 Initial Prompt Creation

We use the following Prompt 1 to generate initial prompts from the agent's task descriptions. The placeholders for task description, input, and output variables are replaced with the respective agent-specific information before inference. We regenerate only outputs that violate the required interface, e.g., missing input variables, missing output tags, or malformed structured-output instructions.

Prompt 1: Prompt Generation Prompt

```
You are a prompt engineer designing prompts for nodes in a **multi-agent
↪  pipeline**. Each prompt will be executed by an LLM agent that receives
↪  structured inputs and must return structured outputs.

Create **6 diverse prompts** for the following task. They should be:
- **Linguistically diverse** (always in English), with **varying lengths and
↪  complexities** - some are short and high-level, others more detailed and
↪  elaborate
- **Formatting varies** across prompts (e.g. plain prose, markdown headers,
↪  bullet lists, tables, numbered steps - mix these up)
- **Variable order varies** across prompts - don't always introduce inputs and
↪  outputs in the same sequence
- **Output format is always XML tags** - every output variable must be returned
↪  inside a named XML tag (e.g. `<summary>...</summary>`, `<score>...</score>`)

**Task:** <task_description>

**Inputs:** <Inputs>

**Outputs:** <Outputs>

Each prompt must:

1. **Preserve all input placeholders exactly** as provided (e.g.
↪  `{variable_name}`). These are injected at runtime by the pipeline - do not
↪  paraphrase, inline, or remove them. Separate each input clearly so there is
↪  no ambiguity about where one ends and the next begins (e.g. use labels, XML
↪  tags, or clear delimiters). Feel free to change the order though.

2. **Include a few-shot section** using the <few_shots_section> and
↪  </few_shots_section> tags.
However make sure, that any mention of the few shot examples are inside of the
↪  tags, as potentially there might be NO few-shot example, in which case the
↪  content of the tag will be ignored. Ensure, that you make it explicit to the
↪  model, that these are few shot examples.

3. **Explicitly state the expected output format** as part of each prompt,
↪  specifying that each output variable must be wrapped in its corresponding
↪  XML tag.

Return the 6 prompts as a JSON array of strings:

```json
["prompt1", "prompt2", "prompt3", "prompt4", "prompt5", "prompt6"]
```

A few things to keep in mind across the 6 prompts:
- Vary which section comes first: sometimes present the inputs before the
↪ instructions, sometimes after
- Vary the tone: some prompts can be terse and directive, others explanatory
```

```
- Vary how the output format is described: sometimes a brief note, sometimes a
↪  detailed schema
- The few-shot block can appear before or after the main instructions, but
↪  always uses the `<few_shots_section>` wrapper
```

D.2 Prompt of the Attribution Model

Prompt 2 shows the full prompt used for the attribution model. The prompt instructs the attributer to compare multiple executions of the same query, each produced under a different agent parameterization, and to assign a credit score in $[-1, +1]$ to each agent per execution based on behavioral differences across executions, their effect on downstream reasoning, and the resulting system scores. To guide structured and consistent output, the prompt includes explicit formatting rules and a few-shot example covering three archetypal attribution scenarios: a fully correct execution, a failure originating in an early agent, and a localized formatting error. The placeholders for the number of parameterizations, the agent's name, and the rollouts are replaced with the respective runtime values before inference.

Prompt 2: Prompt of the Attribution Model

```
# Task
You are an attribution agent for a multi-agent system.

You will receive multiple executions of the SAME query.
Each execution uses different agent parametrizations but follows the same
↪  workflow.

Your goal is to estimate how each agent's behavior contributed to the outcome of
↪  its execution by comparing differences across executions.

Base your attribution on:
- the system score differences across executions,
- the system prediction
- how agent outputs differ across executions,
- whether these differences improve or degrade downstream reasoning,
- whether an agent introduces useful reasoning or merely follows,
- whether an agent propagates, corrects, or ignores errors.

## Attribution Scale
Assign a float score in [-1.0, 1.0] to each agent for each execution:
- positive = helpful contribution
- negative = harmful contribution
- 0 = neutral or unclear impact

## Critical Reasoning Guidelines
- Identical predictions or system scores do not imply identical contributions;
↪  base attribution on differences in reasoning and role adherence.

## Input
You will receive JSON with:
- "query": same for all executions
- "executions": list of execution results, each containing:
  - "system_score": this is the reward signal for attribution, computed by the
  ↪  evaluator based on the execution's prediction. The goal is to maximize
  ↪  this quantity.
  - "agent_outputs": mapping from agent name to that agent's output. The output
  ↪  with the key "prediction" is the final prediction, on which the system is
  ↪  scored on.
```

```
## Output
First, briefly compare the executions and identify the key behavioral
↪  differences that lead to differences in the system score.

Then return JSON inside <attribution>...</attribution>.

## Critical Formatting Rules
- The content inside <attribution>...</attribution> must be valid JSON.
- Do not use markdown code fences inside the <attribution> block.
- The top-level JSON object inside <attribution> must have exactly one key:
↪  "executions".
- "executions" must be a list with exactly {n_parametrizations} items.
- Preserve the same execution order as given in the input.
- Each item in "executions" must be an object with exactly one key:
↪  "agent_credits".
- "agent_credits" must be an object containing exactly these agent names and no
↪  others:
  {agent_names}
- Every credit must be a float between -1.0 and 1.0.
- Do not omit any execution.
- Do not put explanations, summaries, or any other text inside the JSON.
- Any reasoning or explanation must be placed outside the <attribution> block.

# Few-Shot Example

## Input
{
  "query": "A number doubled and then increased by 3 equals 11. What is the
  ↪  number?",
  "executions": [
    {
      "system_score": 1.0,
      "agent_outputs": {
        "planner": {
            "plan": "Let x be the number. Then 2x + 3 = 11. Subtract 3 to get 2x
            ↪  = 8. Divide by 2 to get x = 4."
        },
        "formatter": {
            "formatted_answer": "4"
        },
        "critic": {
            "correct: "Yes",
            "prediction": "4"
        }
      }
    },
    {
      "system_score": 0.0,
      "agent_outputs": {
        "planner": {
            "plan": "A possible number is 5 because doubling gives 10 and that
            ↪  is close to 11."
        },
        "formatter": {
            "formatted_answer": "5"
        },
        "critic": {
            "correct: "True",
```

```
            "prediction": "5"
        }
      }
    },
    {
      "system_score": 0.0,
      "agent_outputs": {
        "planner": {
            "plan": "Let x be the number. Then 2x + 3 = 11. Subtract 3 to get 2x
            ↪  = 8. Divide by 2 to get x = 4."
        },
        "formatter": {
            "formatted_answer": "x = 4"
        },
        "critic": {
            "correct: "True",
            "prediction": "x = 4"
        }
      }
    }
  ]
}

## Output

The first execution is fully correct. The planner sets up and solves the
↪  equation properly, the formatter states the correct answer, and the critic
↪  confirms consistency between reasoning and output. An attribution of 1.0 for
↪  all agents is appropriate as all agents performed their roles without any
↪  error.

The second execution fails primarily due to the planner. The planner does not
↪  formulate the equation and instead relies on an unsupported guess, which
↪  leads to an incorrect solution. The formatter correctly follows its role by
↪  presenting the answer clearly and in the required format, but it propagates
↪  the incorrect result. The critic fails to identify that the reasoning is
↪  invalid and does not challenge the incorrect answer. Therefore, the planner
↪  deserves an attribution of -1.0, as it is the source of the error, while the
↪  critic shares some blame for not catching the error, giving an attribution
↪  of -0.8. The formatter, while it does propagate the error, is not the cause
↪  and still fulfills its role in terms of format and clarity, so an
↪  attribution of 1.0 is reasonable.

The third execution shows a localized failure due to formatting. The planner
↪  correctly solves the equation, and the critic confirms that the reasoning is
↪  valid. However, the formatter outputs the answer in an incorrect format,
↪  violating the output requirements. A system score of 0.4 is reasonable here:
↪  the underlying reasoning and value are correct, but the final output is not
↪  usable due to formatting errors.

<attribution>
{
  "executions": [
    {
      "agent_credits": {
        "planner": 1.0,
        "formatter": 1.0,
        "critic": 1.0
```

```
      }
    },
    {
      "agent_credits": {
        "planner": -1.0,
        "formatter": 1.0,
        "critic": -0.8
      }
    },
    {
      "agent_credits": {
        "planner": 1.0,
        "formatter": -1.0,
        "critic": 0.9
      }
    }
  ]
}
</attribution>

# Attribution

Now perform attribution for the following context.

## Query and Executions
{rollouts}

Compare all {n_parametrizations} executions and assign scores for agents in
↪ {agent_names}.
```

D.3 Initial Prompts

We report the initial prompt sets from seed 42 for each benchmark. All initial prompts are available in the paper's supplementary material at `https://github.com/finitearth/cantante`.

Prompts 3–5 show the initial prompt set for MBPP corresponding to seed 42, which achieved 7.00 % on the respective evaluation set.

Prompt 3: Initial Prompts on MBPP: Planner Prompt

```
You will receive three pieces of information:

- **Query**: {query}
- **Test_cases**: {test_cases}


Your job:
- Analyze the query and test cases.
- Draft a step-by-step implementation plan (no code).
- Identify the required function signature, core algorithm, edge cases, and any
↪  helpers.

**Return format**: The entire plan must be wrapped in a single `<plan>` XML
↪  element, e.g.
<plan> ... </plan>
No additional tags or prose outside this element.
```

Prompt 4: Initial Prompts on MBPP: Executor Prompt

```
You will receive the following items:

- `<query>`: {query}
- `<plan>`: {plan}
- `<fix_feedback>`: {fix_feedback} (may be empty)



Your responsibility is to translate the **plan** into a correct Python function.
↪  Employ the tools `write_code`, and `run_code` as many times as needed until
↪  the solution passes every supplied test case. When solving the task:
1. Decide if a tool is needed.
2. If yes, call it using:
   <tool_call>{{"name": "<tool_name>", "arguments": {{...}}}}</tool_call>
3. Use the returned result to continue reasoning.
4. Repeat until the task is complete. No extra functionality should be
↪  introduced unless the `{fix_feedback}` explicitly directs a change.

**Result format**
The final answer must be a single XML element:
<code>
# complete python code here
</code>
No other text should appear.

Available tools:
- run_code: Execute the currently stored Python code and return its output.

        The stored code is run in a restricted subprocess with time and memory
        limits. Both standard output and standard error are captured. If the
        program exits with a non-zero status, the exit code is included in the
        returned message.

        Returns:
            A string containing stdout, optional stderr, and optional exit code.
            Returns an error-style message if no code is stored, execution times
            out, or another execution error occurs. Arguments: ``.
- write_code: Write or replace the current Python code in the coding
↪  environment.

        This tool stores the provided code so it can later be inspected,
        executed, or tested with other tools. Code is sanitized before being
        saved: Markdown code fences are stripped, and unsafe imports or calls
        are rejected.

        Args:
            code: Python source code to store in the environment.

        Returns:
            A confirmation string if the code was stored successfully,
            or a rejection message if the code violates safety checks.
            ↪  Arguments: `code`.
```

Prompt 5: Initial Prompts on MBPP: Validator Prompt

```
1. Verify the implementation generated by the coder agent.
2. Execute every test case found in {test_cases} against the supplied {code}.
```

````
3. Report the outcome using XML tags.

Inputs (do not modify or rename):
{query}
{code}
{test_cases}


Output schema:
- <fix_required>: true if a test fails or runtime error, else false.
- <fix_feedback>: description of the problem (required only when fix_required is
↪  true).
- <prediction>: the verified code (required only when fix_required is false).

All three tags must be present in the response, wrapped exactly as shown.
↪  Continue using tools until you are confident the task is complete.
The implementation can be inspected using `read_code`, and its correctness
↪  verified using `check_tests`.
Use the following format to call them:

```
<tool_call>{{"name": "read_code", "arguments": {{...}}}}</tool_call>
<tool_call>{{"name": "check_tests", "arguments": {{...}}}}</tool_call>
```
Only produce the final output when no further tool calls are necessary.
The final output must strictly follow the required XML schema.

Available tools:
- check_tests: Run all stored test assertions against the current code.

        Each test is executed together with the stored code in an isolated,
        restricted subprocess. The result reports whether each test passed,
        failed, timed out, or raised an execution error.

        Returns:
            A summary string containing the number of passed tests and
            detailed per-test results. Returns an error-style message if
            no code is stored or no tests are available. Arguments: ``.
- read_code: Read the current Python code stored in the coding environment.

        Returns:
            The full stored code as a string, or "(empty)" if no code
            has been written yet. Arguments: ``.
````

Prompts 6–9 show the initial prompt set for GSM8K corresponding to seed 42, which achieved 69.40 % on the respective evaluation set.

Prompt 6: Initial Prompts on GSM8K: Executor 1 Prompt

```
**Query to solve:** `{query}`

You are an expert mathematician. Using the examples above (if any) as guidance,
↪  work through the problem step-by-step. After completing your calculations,
↪  respond with the following XML structure:

<proposal_1>numeric answer</proposal_1>
<justification_1>detailed reasoning</justification_1>
```

```
Important: The `<proposal_1>` element must contain only the number, and no
↪  external tools are permitted.
```

Prompt 7: Initial Prompts on GSM8K: Executor 2 Prompt

```
### Task
Given the mathematical expression below, compute its value.

1. Read the query: `{query}`.
2. Perform a clear, logical derivation.
3. Produce **exactly** two XML elements:
   - `<proposal_2>` - the numeric answer, plain integer/float, no extra text.
   - `<justification_2>` - the reasoning that led to the answer.

Do not include any other text.
```

Prompt 8: Initial Prompts on GSM8K: Executor 3 Prompt

```
You're tasked with solving a math problem and articulating your thought process.

> **Problem:** {query}



**Please**:
- Walk through the solution logically.
- End with the answer wrapped in `<proposal_3>` (just the number) and a short
↪  justification in `<justification_3>`.

**Return format** (exactly):
```xml
<proposal_3>NUMBER</proposal_3>
<justification_3>TEXT</justification_3>
```
```

Prompt 9: Initial Prompts on GSM8K: Consensus Agent Prompt

```
Task Overview
------------
You must review the following data, assess the validity of each justification,
↪  and emit a single numeric prediction.

Data Table:
| Item | Content |
|------|---------|
| Query | {query} |
| Proposal 1 | {proposal_1} |
| Justification 1 | {justification_1} |
| Proposal 2 | {proposal_2} |
| Justification 2 | {justification_2} |
| Proposal 3 | {proposal_3} |
| Justification 3 | {justification_3} |



Steps:
1. Read the query.
2. For each proposal, evaluate the justification.
```

```
3. Select the most reliable numeric value.

Output must be exactly: `<prediction>NUMBER</prediction>`.
```

Prompts 10–13 show the initial prompt set for HotpotQA corresponding to seed 42, which achieved 7.20 % on the respective evaluation set.

Prompt 10: Initial Prompts on HotpotQA: Retriever Prompt

```
Imagine you are a research assistant whose sole responsibility right now is to
↪  gather evidence for a question.

Your inputs are:
{query}



Using the corpus tools (`list_passages`, `retrieve_passage_by_id`), locate the
↪  passages that most directly address the query. Do not attempt to answer the
↪  query-just collect the texts.

When you are finished, produce ONE XML block named `<retrieved_passages>`
↪  containing each passage verbatim, together with its title.

Exact required output:
<retrieved_passages>
  ...
</retrieved_passages>

Available tools:
- list_passages: List all available passages with their chunk IDs and titles.

        Args:
            query: The question to answer (passed by the agent).

        Returns:
            JSON string containing a list of objects with `chunk_id` and
            ↪  `title`. Arguments: `query`.
- retrieve_passage_by_id: Retrieve the full content of a passage by its chunk
↪  ID.

        Args:
            chunk_id: The zero-based ID of the passage to retrieve.

        Returns:
            JSON string containing `chunk_id`, `title`, and `content`,
            or an error object if the chunk_id is invalid. Arguments:
            ↪  `chunk_id`.
```

Prompt 11: Initial Prompts on HotpotQA: Reader Prompt

```
### Facts Extraction Prompt
**Purpose**: Isolate verifiable statements from each retrieved document that
↪  address the user's {query}. These statements will later feed the
↪  answer-generation agent.

| Section | Content |
|---------|---------|
```

```
| Query | {query} |
| Passages | {retrieved_passages} |
| Hallucination Feedback (optional) | {hallucination_feedback} |
 |

**Instructions**
- Scan each passage independently.
- Pull out any sentence or clause that is a *fact* answering the query.
- Quote the text exactly and prepend the passage number in brackets, e.g., `[3]
↪  "..."`.
- Discard any claim listed in {hallucination_feedback} as hallucinated.

**Output**
Provide a single XML element named `<passage_analysis>` containing one fact per
↪  line, formatted as shown above. No additional prose is allowed.

```xml
<passage_analysis>
[1] "..."
[2] "..."
</passage_analysis>
```
```

Prompt 12: Initial Prompts on HotpotQA: Hallucination Detector Prompt

```
**Goal**: Confirm that every statement in the synthesizer's `answer` is backed
↪  by the `retrieved_passages` using the provided `supporting_facts`.

**Procedure**
- Scan the `supporting_facts` list.
- For each fact, search the `retrieved_passages` for a matching passage
↪  (verbatim or semantically equivalent).
- Flag any fact that cannot be found.
- If any flag appears, output hallucination details; otherwise output the
↪  verified answer.

**Inputs**
{query}
---
Passages:
{retrieved_passages}
---
Answer:
{answer}
---
Facts:
{supporting_facts}
---



**Expected XML output**
- `<hallucination_detected>` (`true`/`false`)
- If true → `<hallucination_feedback>` with claim & passage reference.
- If false → `<prediction>` containing the original answer.
```

Prompt 13: Initial Prompts on HotpotQA: Synthesizer Prompt

```
Produce a final answer with supporting citations.



**Steps**
1. Examine the `{query}` and the `{passage_analysis}`.
2. Pull out every sentence that directly answers or contributes to the answer.
3. For each pulled sentence, note its passage title and sentence number.
4. Write a concise `<answer>` and list the collected citations inside
↪  `<supporting_facts>`.

**Data**
- Query: {query}
- Analysis: {passage_analysis}

**Output**
Return exactly:
```xml
<answer>...</answer>
<supporting_facts>...</supporting_facts>
```
```

D.4 Best Prompts per Task

We report the highest-scoring prompt sets identified by CANTANTE for each benchmark.

D.4.1 MBPP

Prompts 14–16 show the prompt set for MBPP corresponding to seed 47, which achieved 47.40 % on the respective evaluation set.

Prompt 14: CANTANTE Prompts on MBPP: Planner Prompt

```
# Overview
At runtime you will be given three inputs:

`{query}` → {query}
`{test_cases}` → {test_cases}


The `<few_shots_section>` may be empty.

## Task
Compose a concise, human-readable, step-by-step implementation plan for the
↪  Python programming request described by the inputs. **Do not write any
↪  code**-just outline:

- The exact function signature (name, parameters, and return type if relevant).
- The main algorithmic strategy to solve the problem.
- All edge cases and special conditions that the provided test cases reveal.
- Any auxiliary/helper functions that would clarify the solution.

The plan should be written in plain language so a coder agent can follow it
↪  directly.

**Output format**
Return a single XML element that encloses the plan:
```

```xml
<plan>
...your step-by-step plan...
</plan>
```

Do not include any other tags, explanations, or markup outside the `<plan>`
↪ element.

Prompt 15: CANTANTE Prompts on MBPP: Executor Prompt

```
### Output Specification
Return **exactly** one XML element named `<code>` that wraps the full Python
↪ implementation.

### Workflow
- Use the `write_code`, and `run_code`, tools to iteratively build and verify
↪ the solution.
- Adhere strictly to the supplied `{plan}`; only incorporate modifications if
↪ they are mandated by. Tool calls must be emitted exactly in the following
↪ format:
<tool_call>{{"name": "<tool_name>", "arguments": {{...}}}}</tool_call>

Do not include any additional text in the same message as a tool call.
Your final answer must not contain any tool_call tags.
- Ensure the code passes **all** test cases before emitting it.

---
**Data you will receive**
- Query: `{query}`
- Plan: `{plan}`
- Fix feedback (optional): `{fix_feedback}`


---
**Final answer example**
```xml
<code>
import sys

def solution(...):
    # implementation
    pass
</code>
```

Available tools:
- run_code: Execute the currently stored Python code and return its output.

        The stored code is run in a restricted subprocess with time and memory
        limits. Both standard output and standard error are captured. If the
        program exits with a non-zero status, the exit code is included in the
        returned message.

        Returns:
            A string containing stdout, optional stderr, and optional exit code.
            Returns an error-style message if no code is stored, execution times
```

```
            out, or another execution error occurs. Arguments: ``.
- write_code: Write or replace the current Python code in the coding
↪  environment.

        This tool stores the provided code so it can later be inspected,
        executed, or tested with other tools. Code is sanitized before being
        saved: Markdown code fences are stripped, and unsafe imports or calls
        are rejected.

        Args:
            code: Python source code to store in the environment.

        Returns:
            A confirmation string if the code was stored successfully,
            or a rejection message if the code violates safety checks.
            ↪  Arguments: `code`.
```

Prompt 16: CANTANTE Prompts on MBPP: Validator Prompt

```
# Validation Prompt

## Overview
You are the final validator in the code-generation pipeline. Your job is to
↪  confirm that the coder agent's implementation is correct and that it passes
↪  every supplied test case. **Do not modify the code**-only assess it.

**Provided inputs**
- Query: {query}
- Code: {code}
- Test Cases: {test_cases}


## Procedure
1. (Optional) Use the `read_code` tool to inspect the stored code.
2. Run the test suite with the `check_tests` tool on the given {test_cases}.
3. Continue invoking tools until you are confident the verification is complete.
4. If any test fails, times out, or the code raises an exception, output:
   ```xml
   <fix_required>true</fix_required>
   <fix_feedback>a concise description of the first failing test or error and
   ↪  the required change</fix_feedback>
   ```
5. If all tests succeed, output:
   ```xml
   <fix_required>false</fix_required>
   <prediction>{code}</prediction>
   ```

Output format
Exactly the three XML tags above must appear, in this order, with no extra text
↪ or explanations outside these tags.

Tool invocation
```
<tool_call>{{"name": "read_code", "arguments": {{}}}}</tool_call>
<tool_call>{{"name": "check_tests", "arguments": {{}}}}</tool_call>
```
```

```
## Available tools
- **read_code**: Returns the full stored code as a string (or ‘‘(empty)’’ if
↪  none).
- **check_tests**: Executes all test assertions against the current code and
↪  returns a summary indicating passed, failed, timed-out, or error results.
```

D.4.2 GSM8K

Prompts 17–20 show the prompt set for GSM8K corresponding to seed 7, which achieved 85.60 % on the respective evaluation set.

Prompt 17: CANTANTE Prompts on GSM8K: Executor 1 Prompt

```
--- INPUT SECTION ---
Query: `{query}`

--- END INPUT ---

--- INSTRUCTION ---
1. Read the `{query}` carefully.
2. If a `` block is supplied, examine its contents inside the
↪  `<few_shots_section>` block.
3. Perform a complete, manual, step-by-step calculation, making every piece of
↪  reasoning explicit.
4. Do **not** employ any external tools.
5. Return the numeric result together with a concise justification.

--- OUTPUT SPECIFICATION ---
After solving, output **exactly** the following XML tags and nothing else:

<proposal_1>NUMBER</proposal_1>
<justification_1>EXPLANATION</justification_1>

The `<proposal_1>` element must contain **only** the number-no units, symbols,
↪  or additional text. The `<justification_1>` element should contain the full
↪  step-by-step reasoning that leads to that number.
```

Prompt 18: CANTANTE Prompts on GSM8K: Executor 2 Prompt

```
Hey LLM!
Here's what you've got:

**Query:** {query}

Your job: crunch the numbers, show your work, then spit out:

```xml
<proposal_2>NUMBER</proposal_2>
<justification_2>YOUR REASONING</justification_2>
```

Make sure the proposal is just digits (or decimal) - no symbols, no units.
```

Prompt 19: CANTANTE Prompts on GSM8K: Executor 3 Prompt

```
### Agent 3 - Mathematical Query Solver

**Input Query**
```

```
{query}


**Task**
1. Read the mathematical query attentively.
2. Execute a transparent, step-by-step calculation or logical deduction, writing
↪  each intermediate result on its own line.
3. Finish your response **solely** with the two XML elements shown below-no
↪  extra characters, units, or commentary:

<proposal_3>numeric answer only</proposal_3>
<justification_3>concise logical justification</justification_3>
```

Prompt 20: CANTANTE Prompts on GSM8K: Consensus Prompt

```
### Task
You are given a mathematical query and three candidate answers with their
↪  justifications. Compare the proposals, verify the reasoning, and output the
↪  single best numerical answer.

#### Inputs
- Query: `{query}`
- Proposal 1: `{proposal_1}`
- Justification 1: `{justification_1}`
- Proposal 2: `{proposal_2}`
- Justification 2: `{justification_2}`
- Proposal 3: `{proposal_3}`
- Justification 3: `{justification_3}`



**Output**
Return only the final answer wrapped in XML: `<prediction>...</prediction>`. No
↪  extra text.
```

D.4.3 HotpotQA

Prompts 21–24 show the prompt set for HotpotQA corresponding to seed 7, which achieved 17.40 % on the respective evaluation set.

Prompt 21: CANTANTE Prompts on HotpotQA: Retriever Prompt

```
Imagine you are a research assistant whose sole responsibility right now is to
↪  gather evidence for a question.

Your inputs are:
{query}



Using the corpus tools (`list_passages`, `retrieve_passage_by_id`), locate the
↪  passages that most directly address the query. Do not attempt to answer the
↪  query-just collect the texts.

When you are finished, produce ONE XML block named `<retrieved_passages>`
↪  containing each passage verbatim, together with its title.

Exact required output:
```

```
<retrieved_passages>
  ...
</retrieved_passages>

Available tools:
- list_passages: List all available passages with their chunk IDs and titles.

        Args:
            query: The question to answer (passed by the agent).

        Returns:
            JSON string containing a list of objects with `chunk_id` and
            ↪  `title`. Arguments: `query`.
- retrieve_passage_by_id: Retrieve the full content of a passage by its chunk
↪  ID.

        Args:
            chunk_id: The zero-based ID of the passage to retrieve.

        Returns:
            JSON string containing `chunk_id`, `title`, and `content`,
            or an error object if the chunk_id is invalid. Arguments:
            ↪  `chunk_id`.
```

Prompt 22: CANTANTE Prompts on HotpotQA: Reader Prompt

```
<!-- Brief Directive -->
Extract grounded facts from each item in {retrieved_passages} that answer
↪  {query}. Omit anything mentioned in {hallucination_feedback}.



**Data Provided**
{query}
{retrieved_passages}
{hallucination_feedback}

**Return**
Exactly one XML tag:
<passage_analysis>
[PassageNumber] "Exact fact"
[PassageNumber] "Exact fact"
</passage_analysis>
```

Prompt 23: CANTANTE Prompts on HotpotQA: Hallucination Detector Prompt

```
**Instructions for the verification stage**

Read the sections below; the placeholders will be substituted when the prompt
↪  runs.

<query>{query}</query>
<retrieved_passages>{retrieved_passages}</retrieved_passages>
<answer>{answer}</answer>
<supporting_facts>{supporting_facts}</supporting_facts>
```

```
**Objective**
Ensure that every claim made in the synthesizer's answer is completely supported
↪  by the retrieved passages, using the list of supporting facts supplied.

**Steps**
1. Examine each entry in the <supporting_facts> list.
2. For each fact, locate a passage in <retrieved_passages> that contains the
↪  same information (either word-for-word or in equivalent meaning).
3. If a fact cannot be matched, treat the corresponding claim as ungrounded.

**Required output**
- When at least one claim is ungrounded, output exactly:
  <hallucination_detected>true</hallucination_detected>
  <hallucination_feedback>Identify the ungrounded claim and indicate which
  ↪  passage should have been consulted.</hallucination_feedback>
- When all claims are properly grounded, output exactly:
  <hallucination_detected>false</hallucination_detected>
  <prediction>{answer}</prediction>

Provide only the XML elements specified above; do not add any extra text or
↪  commentary.
```

Prompt 24: CANTANTE Prompts on HotpotQA: Synthesizer Prompt

```
### Inputs
- Query: {query}
- Passage Analysis: {passage_analysis}



**Objective:** Combine the facts extracted in the passage analysis to answer the
↪  query. Do **not** introduce any information that isn't in the analysis. Each
↪  supporting fact must indicate the exact passage title and the sentence (or
↪  index) it originates from.

**Output Requirements:**
- `<answer>` element with the final response to the query.
- `<supporting_facts>` element listing each fact, each prefixed by its source
↪  title and sentence index.

Provide **only** these two XML tags, nothing else.
```