MAPPA: SCALING MULTIAGENT SYSTEMS WITH PROCESS REWARDS

Ed Li*
Yale University

Junyu Ren*
University of Chicago

Cat Yan
University of Oxford

ABSTRACT

While multiagent systems have shown promise for tackling complex tasks via specialization, finetuning multiple agents simultaneously faces two key challenges: (1) credit assignment across agents, and (2) sample efficiency of expensive multiagent rollouts. In this work, we propose finetuning multiagent systems with per-action process rewards from AI feedback (MAPPA) to address both. Through assigning credit to individual agent actions rather than only at task completion, MAPPA enables fine-grained supervision without ground truth labels while extracting maximal training signal from each rollout. We demonstrate our approach on competition math problems and tool-augmented data analysis tasks. On unseen math problems, MAPPA achieves +5.0–17.5pp on AIME and +7.8–17.2pp on AMC. For data analysis tasks, our method improves success rate by +16.7pp while quality metrics improve by up to 47%, validating that per-action supervision can lead to improvements across different multiagent systems on various domains. By addressing these challenges, our work takes a first step toward scaling multiagent systems for complex, long-horizon tasks with minimal human supervision.

Code Blog

1 INTRODUCTION

Large reasoning models trained with reinforcement learning have emerged as the dominant paradigm for state-of-the-art AI systems (OpenAI, 2024; DeepSeek-AI, 2025; Snell et al., 2024). Augmented with external tools such as code interpreters and search engines (Schick et al., 2023), these models achieve remarkable performance on mathematical reasoning (Shao et al., 2024), coding (Jimenez et al., 2024), and scientific problems (Novikov et al., 2025). Building on these capabilities, multiagent systems—where multiple language model-based agents interact to solve complex tasks—offer an intuitive path forward as specializations and the diversity of perspectives can tackle problems better than any single agent. (Wu et al., 2023; Hong et al., 2023).

The bulk of existing multiagent work implements specialization through prompt engineering: assigning different system prompts, personas, or tool access to agents within the system. While effective for orchestrating pre-trained models, the true strength of multiagent systems lies in the ability to *modify the weights* of individual agents, enabling specialization without interference. Recent evidence suggests such specialization naturally emerges through reinforcement learning: reasoning models trained solely for accuracy spontaneously develop diverse internal personas and expertise (Kim et al., 2026). By separating weights across agents, improving one agent's capabilities cannot degrade another's performance, allowing scaling without catastrophic forgetting (Figure 1).

Recent work has begun exploring this direction of fine-tuning agents in multiagent systems simultaneously (Subramaniam et al., 2025; Liao et al., 2025; Park et al., 2025; Liu et al., 2025). However, two significant challenges remain:

1. **Credit assignment**: How do we attribute the performance of the overall system to actions taken by individual agents?

*Equal contribution.

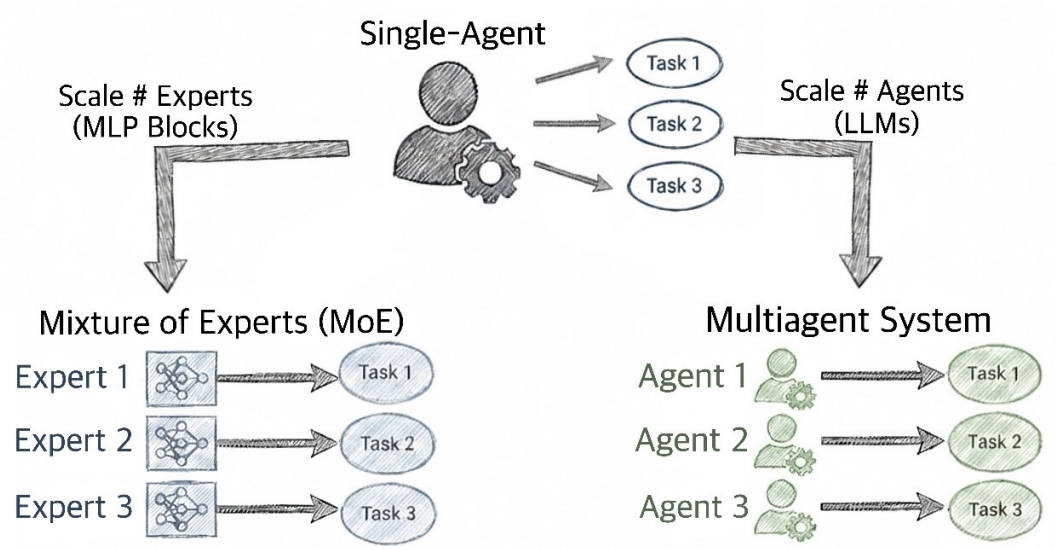

Figure 1: Multiagent architectures with separate weights enable specialization through end-to-end training, sidestepping catastrophic forgetting that limits single-model scaling just like mixture-of-experts (MoE) architecture.

2. **Computational cost**: Rollouts of the entire multiagent system can take minutes or even hours, yet yield only a single outcome reward signal.

We address these challenges through finetuning **M**ulti**A**gent system with **P**er-action **P**rocess rewards from **A**I feedback (**MAPPA**). Rather than assigning a single sparse reward at trajectory's end according to fixed instructions in LLM-as-a-Judge, we leverage language models as *coaches* to assess the quality of each agent's action given its role, inputs, and environment feedback such as tool execution results. This yields dense learning signal throughout the trajectory, enabling effective training even when tasks fail entirely. Crucially, the coach performs implicit credit assignment: when a downstream agent encounters a file-not-found error for an artifact that an upstream agent should have produced, the coach assigns low scores to the upstream agent's faulty actions rather than penalizing the downstream agent. Furthermore, the number of training signals now scales with the number of actions taken, dramatically improving sample efficiency over outcome-based methods.

We validate this approach on two separate domains: mathematical reasoning with code execution (MathChat) and end-to-end data science pipelines (DSBench (Jing et al., 2025)). On MathChat, our three-agent system achieves improvements of **+5.0–17.5pp on AIME** and **+7.8–17.2pp on AMC** across two model configurations. On DSBench, where multiagent pipelines must engineer features, train models, and generate predictions, training improves success rate by **+16.7pp** while quality metrics improve by up to **47%**.

We present a general framework for training multiagent systems on complex, tool-augmented tasks via coach-guided reinforcement learning. The key components—agent topology, reward structure, and training pipeline—are domain-agnostic and can be configured to support diverse applications. Our source code is available at `https://github.com/ltjed/multiagent-coaching`. Overall, our results suggest that **scaling the number of specialized agents represents a promising new dimension for improving performance on complex, long-horizon tasks**.

2 Methodology

2.1 Problem Setting

MAPPA finetunes tool-augmented multiagent systems where LLM agents collaborate to solve tasks. Each agent is initialized from a pretrained LLM with independent policy parameters trained separately, enabling specialization. All agents live in a terminal environment and execute in a pre-

defined topology (e.g., sequential pipelines, debate, mixture-of-agents). Within each agent's turn, it generates one or more actions that may include tool calls. When tool calls are generated as part of an agent's action, they are processed and executed in a sandboxed environment, the agent will then receive environment feedback (e.g., stdout, stderr, error messages). Depending on the task, the agent may also need to call tools to read, write files or other artifacts in the terminal environment.

2.2 Process Rewards from AI Feedback

Multi-turn tool use with inter-agent dependencies creates challenging credit assignment—which agent's actions led to success or failure? Unlike existing RL approaches that rely on sparse outcome rewards (success/failure for entire trajectories), we leverage a *coach* LLM to perform per-action credit assignment, providing dense process-quality rewards on a 0–10 scale.[1] We use the term "coach" rather than "judge" to emphasize that while LLM-as-a-judge evaluates final outputs against fixed rubrics, a coach provides context-aware feedback on each intermediate action to guide improvement throughout the trajectory. The coach evaluates each action holistically based on: (1) the agent's role and responsibilities, (2) the input context the agent observed, (3) the agent's action, and (4) tool execution results (stdout, stderr, error messages) when the output contains tool calls. This enables the coach to identify the responsible agent when failures occur: if a downstream agent encounters a missing file error, the coach assigns low scores to the upstream agent that should have produced it, not the downstream agent that correctly reported the issue. See Appendix D for example coach evaluations demonstrating this context-aware feedback.

Crucially, MAPPA can work *with or without ground truth/verifier* that most other alternatives RL approaches such as RLVR (Lightman et al., 2023; DeepSeek-AI, 2025) require. When ground truth is available (e.g., ROC-AUC, F1, RMSE for DSBench), the coach incorporates it to inform process scores via intelligently synthesizing multi-dimensional metrics into a single reward; When ground truth is unavailable, the coach simply evaluates how sensible it is for the agent to take that action given its context to the best of the coach's ability. In this work, ground truth is only provided for the last action of the last agent in the multiagent system.

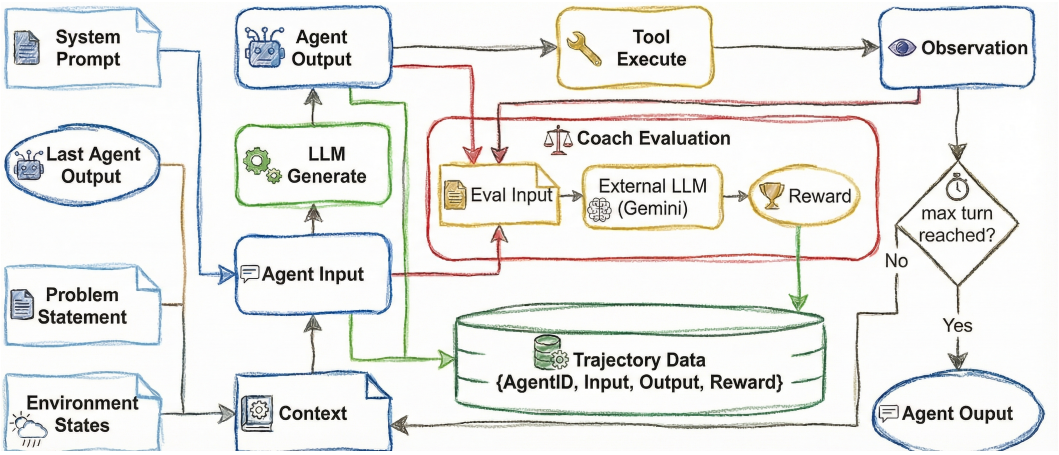

Figure 2: Agent execution loop with per-action coach evaluation.

2.3 Training Algorithm

As shown in Figure 2, each agent action generates a trajectory tuple $(\text{agent_id}, \text{input}, \text{action}, \text{reward})$ where the reward is provided by the coach evaluation. We then finetune each agent's underlying model with its actions' corresponding tuples via REINFORCE++ (Hu et al., 2025). We choose REINFORCE++ rather than Group-Relative Policy Optimization (GRPO) (DeepSeek-AI, 2025) for a fundamental reason: GRPO normalizes advantages within groups of samples sharing the same

[1]We found the 0–10 scale empirically outperforms continuous 0–1 scores, likely due to the prevalence of 0–10 rating samples in the pre-training corpus, i.e., a "7" or a "4," carries more semantic meaning than decimal values like 0.305.

prompt, assuming identical input states. This assumption breaks in end-to-end multiagent training—each agent's input depends on upstream agents' stochastic outputs, so even rollouts from the same initial prompt produce different intermediate states. REINFORCE++ instead applies global batch normalization across all experiences, naturally handling state diversity. Global normalization also reduces variance from noisy coach rewards compared to deterministic ground truth metrics. Each action receives an independent coach reward $r_t^{\text{coach}} \in [0, 1]$, with KL penalties computed per-token. The KL-penalized reward and advantage are:

$$r_t = r_t^{\text{coach}} - \beta \cdot D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}), \quad A_t = \sum_{\tau \geq t} r_\tau \tag{1}$$

where $\beta = 0.01$ and A_t is the undiscounted return-to-go, propagating downstream rewards to earlier actions. Advantages are then globally normalized across all agents and experiences in the batch: $\hat{A}_t = (A_t - \mu)/\sigma$. The policy gradient uses the standard PPO clipped objective with $\epsilon = 0.2$. See Appendix E for full derivations. Our distributed training infrastructure, extended from MARTI (Zhang et al., 2025), uses Ray for coordination, vLLM for inference, and DeepSpeed ZeRO-3 for memory-efficient updates. Each agent maintains independent policy parameters with parallel trajectory collection and synchronous gradient updates. See Appendix G for architecture details.

3 RESULTS

We demonstrate our approach on two separate multiagent systems across two domains: competition math with code execution (MathChat) and end-to-end data science pipelines (DSBench). While competition math can be solved by single agents (OpenAI, 2024; DeepSeek-AI, 2025), applying MAPPA to multiagent systems with different underlying models and constraints tests the generality and robustness. DSBench, in contrast, more closely resembles real-world complex tasks that multiagent systems are built for.

3.1 MATHCHAT: COMPETITION MATH WITH CODE EXECUTION

Task and Dataset. MathChat involves solving AIME (American Invitational Mathematics Examination) competition math problems with code execution. We randomly sample 512 problems from 1983–2024 for training and use disjoint held-out sets for evaluation: 30 AIME problems from 2025 and 32 AMC (American Mathematics Competition, similar to but slightly simpler than AIME) problems.

Multiagent Configuration. We implement a three-agent sequential pipeline designed to separate reasoning, computation, and verification:

- *Problem Solver*: Drafts step-by-step reasoning for downstream agents. The prompt instructs: "Your job is to draft a solution to the problem..."
- *Code Executor*: Has the option to write and execute Python code to verify and compute solutions. The prompt instructs: "You can execute Python code. Write code in ```python``` blocks and it will be automatically executed."
- *Verifier*: Synthesizes information from upstream agents and outputs the final answer. The prompt instructs: "You are the last agent. The system succeeds only if YOU output the correct answer." The final answer must be formatted as \boxed{answer} to be counted as correct.

Each agent is initialized from the same pretrained checkpoint but finetuned separately, enabling specialization. We experiment with two reasoning models: DeepSeek-R1-Distill-Qwen-1.5B (DeepSeek-AI, 2025) and Qwen3-4B (Team, 2025). All agents use reasoning model inference with <think> tags, allowing internal deliberation before generating outputs. Each agent has a 4K token output limit after which the response is cut off, content within <think> tags included. See Appendix B for complete agent prompts.

We finetune using REINFORCE++ with Gemini 2.5 Flash as the coach on 8 NVIDIA H100 GPUs (training details in Appendix A.1).

Table 1: MathChat performance on held-out competition math problems.

	R1-Distill-Qwen-1.5B	Qwen3-4B
AMC		
Baseline	60.9%	78.1%
Best	78.1%	85.9%
Δ	+17.2pp	+7.8pp
AIME		
Baseline	24.2%	49.2%
Best	29.2%	66.7%
Δ	+5.0pp	+17.5pp

Results. As shown in Table 1, MAPPA improves performance across both model configurations, with gains ranging from +5.0pp to +17.5pp depending on model capacity and task difficulty. The larger Qwen3-4B achieves the biggest improvement (+17.5pp) on AIME, while the smaller DeepSeek-R1-Distill-Qwen-1.5B shows asymmetric gains: +17.2pp on the more accessible AMC but only +5.0pp on AIME 2025, suggesting capacity limitations on harder reasoning tasks.

Interestingly, behavioral metrics reveal divergent learning patterns between the two models (Figure 3). The larger Qwen3-4B shows dramatic behavioral adaptation: successful tool call rate increases substantially while response lengths decrease across all three agents. In contrast, the smaller 1.5B model maintains relatively stable behavioral metrics throughout training. This suggests that with greater capacity, Qwen3-4B learns to leverage tool calls more effectively through MAPPA, while the 1.5B model improves accuracy without developing such qualitative behavioral changes—demonstrating that process rewards can drive improvement even when model capacity limits behavioral adaptation. We also evaluate under partial information constraints, where each agent observes only the immediately preceding agent's output with no access to earlier context; results in Appendix A.2 show MAPPA still achieves consistent improvements (+3.9–5.8pp), further demonstrating robustness.

3.2 DSBENCH: DATA SCIENCE PIPELINES

Task and Dataset. DSBench presents 72 Kaggle-style machine learning tasks requiring complete end-to-end pipeline execution. Each task provides training/test CSV files, task type (classification/regression), and ground truth labels for evaluation. We use 64 tasks for training and 6 for held-out evaluation (4 classification, 2 regression)—an 8.6% held-out ratio that is reasonable given the limited size of existing data science benchmarks.

Multiagent Configuration. We implement a three-agent sequential pipeline where each agent specializes in a distinct phase: the *Data Engineer* performs exploratory data analysis, preprocessing, and feature engineering (maximum 3 turns); the *Modeler* handles algorithm selection, model training, and hyperparameter tuning (maximum 5 turns); and the *Analyst* generates final predictions and formats submissions (maximum 2 turns). Each agent is initialized from the same pretrained checkpoint (Qwen3-4B) with independent policy parameters. Agents communicate via base64-encoded file passing: the Data Engineer produces `X_train.pkl`, `y_train.pkl`, `X_test.pkl`; the Modeler consumes these and produces `model.pkl`; the Analyst loads both to generate `submission.csv`. This explicit file dependency enables the coach to assign blame when pipeline failures occur. Figure 4 illustrates this three-agent pipeline; see Appendix C for complete agent prompts.

We train using REINFORCE++ with Gemini 2.5 Pro as the coach; all agents have access to SandboxFusion for code execution (training details in Appendix A.1). The pipeline structure creates natural dependencies: the Modeler cannot train without properly formatted data from the Data Engineer, and the Analyst cannot predict without the trained model from the Modeler. This makes DSBench an ideal testbed for evaluating multiagent credit assignment and process-focused training. We train for 21 epochs (84 steps) and evaluate checkpoints every 2 steps. Table 2 presents metrics at three checkpoints: baseline (epoch 0), peak success rate (epoch 11), and late training (epoch 21). Figure 6 (Appendix A.3) shows the full training dynamics across all 84 steps.

Training initially improves both success rate (+16.7pp) and quality across task types (Accuracy +28.8%, RMSE −41.4% fair), validating that coach rewards translate to downstream metric im-

Figure 3: Behavioral metrics during MathChat training. Code Executor Tool Call % (top-left) shows the percentage of Code Executor actions that contain a tool call; the other three panels show average response length (in tokens) for each agent at each training step. Faint lines show raw data; solid lines show exponential moving average ($\alpha = 1/3$). Qwen3-4B shows dramatically increased tool usage and more concise responses, while R1-Distill-Qwen-1.5B maintains stable behavior throughout training.

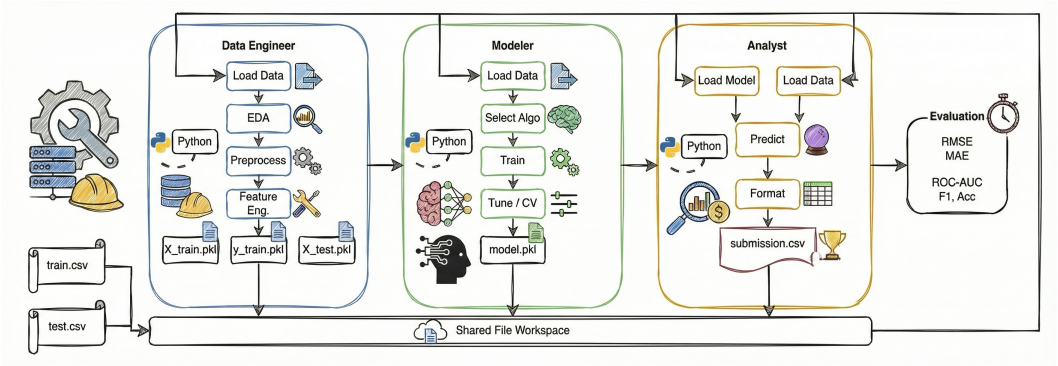

Figure 4: DSBench three-agent pipeline. Each agent executes Python code via a shared sandbox, reading inputs from and writing outputs to a shared file workspace. The Data Engineer preprocesses raw CSV files into pickle artifacts; the Modeler trains and saves a model; the Analyst generates the final submission for evaluation.

provements. Extended training (epoch 11→21) shows a pattern that might appear as overfitting: classification success drops from 56.2% back to baseline 43.8% (−12.4pp), and classification quality metrics decline (Accuracy-Fair −6.8%, F1-Fair −33.9%). However, regression tells a different story—success maintains its peak (87.5%) while quality continues improving: RMSE drops from 9.5% to 8.0% (−15.8%). This asymmetry suggests specialization to regression rather than simple

Table 2: DSBench evaluation comparing baseline (epoch 0), peak success rate (epoch 11), and late training (epoch 21). Epoch 11 maximizes success rate and fair metrics across both task types. By epoch 21, the model specializes in regression: RMSE continues improving while classification metrics regress to baseline. Fair metrics penalize failures (0.5 for Accuracy, 0 for F1, 50% for MAE/RMSE). Each epoch consists of 4 training steps.

Metric	Epoch 0	Epoch 11	$\Delta_{0\to11}$	Epoch 21	$\Delta_{11\to21}$
Success Rate ↑ (24 eval samples: 16 classification + 8 regression)					
Classification	43.8%	**56.2%**	+12.4pp	43.8%	−12.4pp
Regression	62.5%	**87.5%**	+25.0pp	**87.5%**	0pp
Total	50.0%	**66.7%**	+16.7pp	58.3%	−8.4pp
Classification Quality					
Accuracy ↑ (Raw)	0.690	**0.889**	+28.8%	**0.889**	0%
Accuracy ↑ (Fair)	0.583	**0.719**	+23.3%	0.670	−6.8%
F1 ↑ (Raw)	0.288	**0.309**	+7.3%	0.263	−14.9%
F1 ↑ (Fair)	0.126	**0.174**	+38.1%	0.115	−33.9%
Regression Quality					
MAE ↓ (Raw)	7.1%	6.9%	−2.8%	**5.7%**	−17.4%
MAE ↓ (Fair)	23.2%	12.3%	−47.0%	**11.2%**	−8.9%
RMSE ↓ (Raw)	9.8%	9.5%	−3.1%	**8.0%**	−15.8%
RMSE ↓ (Fair)	24.9%	14.6%	−41.4%	**13.2%**	−9.6%
Coach Scores by Agent ↑ (0–10 scale)					
Data Engineer	5.43	5.09	−6.3%	**5.60**	+10.0%
Modeler	4.55	**5.28**	+16.0%	5.10	−3.4%
Analyst	5.56	**7.24**	+30.2%	7.17	−1.0%

overfitting; we investigate this phenomenon in Section 4. Fair metrics enable reliable model selection by identifying epoch 11 as optimal for balanced performance across both task types. The per-agent coach scores in Table 2 demonstrate that credit assignment enables targeted learning, with each agent receiving feedback specific to its role rather than shared outcome-based rewards.

4 Discussion

Despite the potential of multiagent systems, recent work studying inference-time coordination finds that naively scaling agent count *without changing weights* actually *degrades* performance on sequential reasoning tasks by 39–70% (Kim et al., 2025). Our results show that finetuning multiagent systems with process rewards achieves substantial improvements on both MathChat and DSBench.

The DSBench results reveal an unexpected pattern: while regression performance continues improving, classification metrics regress to baseline. To understand this specialization, we stratify coach scores by task type (Table 3). The analysis reveals a systematic pattern: the coach assigns higher scores to regression tasks across all agents, with deltas ranging from +0.51 to +1.80. Notably, the Data Engineer's delta widens from +1.15 to +1.67 over training, while the Analyst's narrows from +1.80 to +0.77. We hypothesize that agents learn to exploit the coach's preference for regression tasks, leading to the observed specialization. This pattern illustrates a fundamental limitation of stateless evaluation: our coach judges each action in isolation without awareness that its scores drive gradient updates, and thus cannot detect or correct emergent imbalances in its own scoring behavior. We discuss how future coaches might overcome this limitation in Appendix E.4.

Comparison with Prior Work. Most existing multiagent frameworks like AutoGen (Wu et al., 2023), MetaGPT (Hong et al., 2023), and CAMEL (Li et al., 2023) focus on collaboration through prompt engineering but do not update agent weights. Recently, more work has begun experimenting with finetuning specific multiagent system setups: Subramaniam et al. (2025) propose *Multiagent Finetuning*, establishing that training multiple agents outperforms training a single agent by preserving diverse reasoning chains; however, their approach uses supervised fine-tuning rather than RL, and focuses specifically on debate where homogeneous agents generate competing solutions to the same problem—a narrower setting than our sequential pipelines with heterogeneous specialized

Table 3: Coach score delta (Regression − Classification) by agent across training epochs. Positive values indicate higher scores for regression tasks. The Data Engineer's widening delta correlates with the observed specialization.

Agent	Epoch 0	Epoch 11	Epoch 21
Data Engineer	+1.15	+1.40	+1.67
Modeler	+0.51	+1.06	+1.02
Analyst	+1.80	+1.17	+0.77

agents. Liao et al. (2025) introduce *MARFT*, providing theoretical foundations for adapting MARL to LLM-based multiagent systems; our work is complementary, focusing on the practical reward design challenge rather than algorithmic formalism. Park et al. (2025) present *MAPoRL*, which co-trains agents through multi-turn debate where a verifier scores both answer correctness and discussion quality (rewarding corrective and persuasive exchanges). While MAPoRL demonstrates that co-training is essential—training individual LLMs alone fails to induce collaboration—their approach requires ground truth labels to train the verifier, uses sparse outcome-based rewards augmented with discussion incentives rather than dense per-action process feedback, and focuses on homogeneous debate agents rather than heterogeneous specialized pipelines. Our coach provides dense supervision at every action without requiring ground truth, enabling learning even when tasks fail entirely. Liu et al. (2025) formalize LLM collaboration as a Dec-POMDP and propose *MAGRPO*, extending GRPO to multiagent multi-turn settings; however, GRPO's same-state assumption breaks in heterogeneous pipelines where upstream stochasticity causes state divergence (see Section 2.3). We use REINFORCE++ with global batch normalization, and our coach's context-aware evaluation enables stable gradient estimation even when actions arise from divergent trajectories.

Mitigating Coach Biases. Using LLMs to assess LLM outputs has well-documented biases that affect evaluation reliability (Ye et al., 2024; Chen et al., 2024). *Verbosity bias* leads to preference for longer responses regardless of quality. *Self-enhancement bias* emerges when LLMs favor outputs similar to their own generations (Wataoka et al., 2024). Our results demonstrate the MAPPA is effective but such biases may still influence scores. In the future, we believe more investigation is warranted, with potential mitigation strategies including ensembling multiple coach models (e.g., combining Claude, Gemini, and GPT-4).

Beyond Scalar Rewards. Currently, our approach only uses scalar rewards from AI feedback, but the coaches are capable of much richer feedback. For instance, when an agent's action is suboptimal, the coach could generate a corrected action that the agents *should have* taken, enabling supervised finetuning (SFT) on improved demonstrations or preference learning via DPO (Rafailov et al., 2023), that combine these with RL. Extracting richer signal from each coach interaction could further improve sample efficiency, especially when agents perform a task to low success rate—even though this improvement may come at the expense of reduced exploration compared to pure RL as in our current work.

Scaling to Large-Scale Multiagent Systems. MAPPA simplifies end-to-end training of multiagent systems to crafting an effective LLM coach and providing it with relevant information to make accurate assessments. The true potential of this approach lies in scaling to large multiagent systems with dozens or more agents for complex, long-horizon tasks such as scientific research, where multiagent systems start to show promise (Li et al., 2025). A key challenge, however, is reward hacking: coach-assigned scores may improve while overall system success stagnates or declines. We believe collecting behavioral metrics throughout training, such as response length and tool call rate shown in Results, is key. These metrics serve as sanity checks and provide interpretable signals for diagnosing what is happening inside a multiagent system as it evolves.

5 Conclusion

MAPPA demonstrates that multiagent systems can be effectively trained end-to-end using process rewards from AI feedback. Dense per-action supervision solves credit assignment, improves sample

efficiency, and generalizes across domains—suggesting that scaling specialized agents through finetuning, rather than prompting alone, represents a promising frontier for complex, long-horizon tasks.

REFERENCES

Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. Humans or LLMs as the judge? a study on judgement bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. REINFORCE++: Stabilizing critic-free policy optimization with global advantage normalization. *arXiv preprint arXiv:2501.03262*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. DSBench: How far are data science agents to becoming data science experts? In *International Conference on Learning Representations*, 2025.

Junsol Kim, Shiyang Lai, Nino Scherrer, Blaise Agüera y Arcas, and James Evans. Reasoning models generate societies of thought. *arXiv preprint arXiv:2601.10825*, 2026.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025.

Ed Li, Junyu Ren, Xintian Pan, Cat Yan, Chuanhao Li, Dirk Bergemann, and Zhuoran Yang. Build your personalized research group: A multiagent framework for continual and interactive science automation. *arXiv preprint arXiv:2510.15624*, 2025.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, 2023.

Junwei Liao, Muning Wen, Jun Wang, and Weinan Zhang. MARFT: Multi-agent reinforcement fine-tuning. *arXiv preprint arXiv:2504.16129*, 2025.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

Shuo Liu, Tianle Chen, Zeyu Liang, Xueguang Lyu, and Christopher Amato. LLM collaboration with multi-agent reinforcement learning. *arXiv preprint arXiv:2508.04652*, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Chanwoo Park, Seungju Han, Xingzhi Guo, Asuman E. Ozdaglar, Kaiqing Zhang, and Joo-Kyung Kim. MAPoRL: Multi-agent post-co-training for collaborative large language models with reinforcement learning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, 2023.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Zhang, Yi Zhang, Xin Bi, Daya Guo, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. *arXiv preprint arXiv:2501.05707*, 2025.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Koki Wataoka, Tsubasa Takahashi, and Ryokan Ri. Self-preference bias in LLM-as-a-judge. *arXiv preprint arXiv:2410.21819*, 2024.

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.

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, et al. Justice or prejudice? quantifying biases in LLM-as-a-judge. *arXiv preprint arXiv:2410.02736*, 2024.

Kaiyan Zhang, Runze Liu, Xuekai Zhu, Kai Tian, Sihang Zeng, Guoli Jia, Yuchen Fan, Xingtai Lv, Yuxin Zuo, Che Jiang, Ziyang Liu, Jianyu Wang, Yuru Wang, Ruotong Zhao, Ermo Hua, Yibo Wang, Shijie Wang, Junqi Gao, Xinwei Long, Youbang Sun, Zhiyuan Ma, Ganqu Cui, Lei Bai, Ning Ding, Biqing Qi, and Bowen Zhou. MARTI: A framework for multi-agent LLM systems reinforced training and inference. `https://github.com/TsinghuaC3I/MARTI`, 2025.

IMPACT STATEMENT

This work advances methods for training multiagent AI systems with minimal human supervision, contributing to the broader goal of building self-improving AI systems. While this could accelerate progress on complex tasks that benefit society, training AI with AI supervisors raises value alignment concerns, as trained agents may inherit biases from coach models. We recommend auditing coach models for biases and monitoring behavioral metrics to detect reward hacking.

A EXTENDED EXPERIMENTAL RESULTS

A.1 TRAINING CONFIGURATION

MathChat. We finetune the multiagent systems using REINFORCE++ with Gemini 2.5 Flash as the coach on 8 NVIDIA H100 GPUs. Key hyperparameters: actor learning rate 10^{-6}, rollout batch size 32, training temperature 1.0, evaluation temperature 0.6, KL penalty coefficient $\beta = 0.01$. With 512 training problems processed in rollout batches of 32, one epoch equals 16 training steps (one complete pass through all problems). We evaluate every 4 steps (every 0.25 epochs) on held-out problems. Each evaluation samples every problem 4 times and reports mean accuracy to reduce variance from stochastic generation. See Appendix G.9 for computational cost details.

DSBench. We train using REINFORCE++ with Gemini 2.5 Pro as the coach. All agents have access to SandboxFusion for Python code execution. When ground truth labels are available, we compute task-specific metrics (Accuracy, F1 for classification; RMSE, MAE for regression) and provide them to the coach. Rather than naively averaging these metrics or picking a single one, the coach intelligently weighs them in context—penalizing reward hacking (e.g., optimizing one metric at the expense of others), attributing poor metrics to the responsible agent (Data Engineer's preprocessing vs. Analyst's predictions), and producing a fair process score that reflects overall execution quality. Training uses Ray, vLLM, and DeepSpeed ZeRO-3 on 8 NVIDIA H100 GPUs. Key hyperparameters: actor learning rate 10^{-6}, rollout batch size 16, training temperature 1.0, evaluation temperature 0.6. Checkpoints are evaluated every 2 steps on 6 held-out tasks.

A.2 MATHCHAT: PARTIAL INFORMATION RESULTS

We also evaluate MAPPA under partial information constraints, where each agent observes only the immediately preceding agent's output (except the Problem Solver, which sees the problem statement), with no access to earlier context or the original question. This tests whether MAPPA can train agents that must operate with limited context.

Table 4: MathChat performance under partial information constraints (Qwen3-4B). Despite degraded baseline performance compared to full context (−5–12pp), MAPPA achieves consistent improvements.

Benchmark	Baseline	Best	Δ
AMC (32)	72.7%	76.6%	+3.9pp
AIME 2025 (30)	37.5%	43.3%	+5.8pp

As expected, partial information constraints degrade baseline performance: AMC drops from 78.1% to 72.7% (−5.4pp) and AIME from 49.2% to 37.5% (−11.7pp) compared to full context. However, MAPPA still achieves consistent improvements of +3.9pp on AMC and +5.8pp on AIME, demonstrating that per-action process rewards provide effective supervision even when agents have limited observability.

Figure 5 shows behavioral metrics for this run. Under partial information constraints, agents start with higher tool usage and shorter responses compared to full-context runs, then further optimize during training: tool calls increase while response lengths continue to decrease.

A.3 DSBENCH: EXTENDED QUALITY METRICS

Figure 7 presents the complete set of quality metrics tracked during DSBench training, showing both raw metrics (computed only over successful samples) and fair metrics (which penalize failures: 0.5 for Accuracy, 0 for F1, 50% for MAE/RMSE). The vertical green line marks step 44 (epoch 11), where overall success rate peaks.

Key observations: (1) Classification metrics (Accuracy, F1, ROC-AUC) peak around step 44 then decline, while regression metrics (MAE, RMSE) continue improving through step 84. (2) Fair metrics show larger improvements than raw metrics because they capture both quality improvement and increased success rate. (3) The divergence between raw and fair metrics after step 44 reflects the

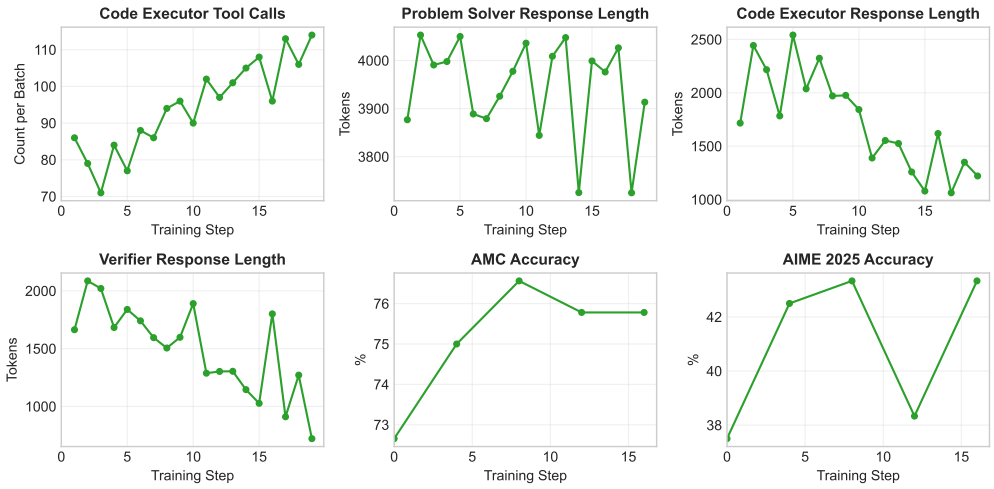

Figure 5: Behavioral metrics for the partial information run (Qwen3-4B). From left to right, top to bottom: Code Executor tool calls, Problem Solver response length, Code Executor response length, Verifier response length, AMC accuracy, and AIME 2025 accuracy. Despite limited context, agents show similar behavioral adaptations as full-context runs.

model's specialization toward regression tasks—raw classification metrics remain stable (successful runs maintain quality) while fair metrics decline (fewer successful runs).

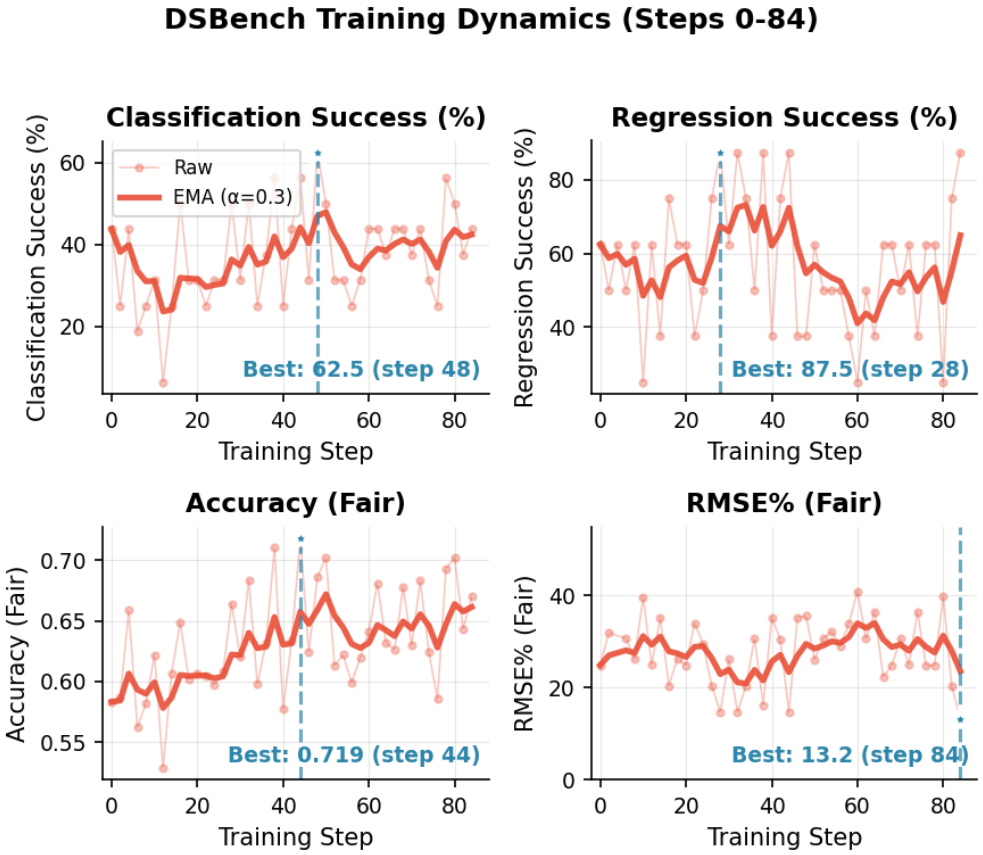

Figure 6: DSBench training dynamics over 84 steps (21 epochs). Light points show raw metrics; solid line shows EMA (α=0.3). Dashed lines mark peak raw values. Classification metrics peak early (steps 44–48) then decline, while regression RMSE continues improving through step 84, illustrating specialization to regression tasks.

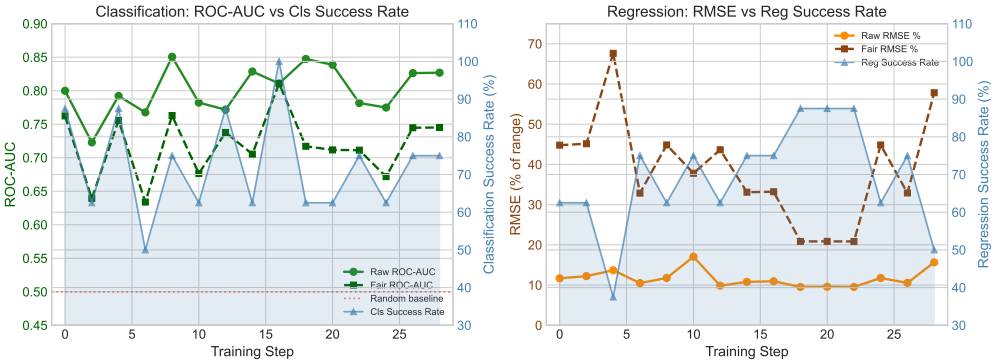

Figure 7: DSBench quality metrics over 84 training steps. Top row: success rates by task type. Middle rows: classification metrics (Accuracy, F1, ROC-AUC) in raw and fair variants. Bottom row: regression metrics (MAE, RMSE) in raw and fair variants. Green vertical line and markers indicate step 44 (peak success rate). Fair metrics penalize failures (Accuracy: 0.5, F1: 0, MAE/RMSE: 50%), capturing both quality and reliability.

B MATHCHAT AGENT PROMPTS

This section provides the complete prompts used by each agent in the MathChat pipeline. All agents receive the original math problem and role-specific instructions.

B.1 PROBLEM SOLVER PROMPT

```
You are Problem Solver in a 3-agent system:
Problem Solver (you) -> Code Executor -> Verifier.

The system succeeds only if the Verifier (final agent)
outputs the correct answer. Your job is to draft a
solution to the problem.

You have a strict 4k token limit (your thinking inside
<think> and </think> tags also counts). Anything beyond
that will be truncated.

## Problem
{problem}
```

B.2 CODE EXECUTOR PROMPT

```
You are Code Executor in a 3-agent system:
Problem Solver -> Code Executor (you) -> Verifier.

The system succeeds only if the Verifier (final agent)
outputs the correct answer. Your job is to compute/verify
the solution using Python code.

You can execute Python code. Write code in ```python```
blocks and it will be automatically executed by the user
on your behalf, based on which you can iterate further
or output final answers.

You have a strict 4k token limit (your thinking inside
<think> and </think> tags also counts). Anything beyond
that will be truncated.

## Problem
{problem}

## Input from Problem Solver
{solution}
```

B.3 VERIFIER PROMPT

```
You are Verifier, the final agent in a 3-agent system:
Problem Solver -> Code Executor -> Verifier (you).

You are the last agent. The system succeeds only if YOU
output the correct answer. Evaluate the information below
and provide the final answer.

You have a strict 4k token limit (your thinking inside
<think> and </think> tags also counts). Anything beyond
that will be truncated. Output your final answer as:
**\boxed{answer}**

## Problem
{problem}

## Input from Code Executor
```

```
{execution}
```

B.4 COACH EVALUATION PROMPT

The coach evaluates each agent action using the following prompt template:

```
You are evaluating a specific agent in a multiagent
system that collaboratively solves a math competition
problem. The system has 3 specialized agents working
sequentially:

1. **Problem Solver**: Reasons through the problem
   step-by-step
2. **Code Executor**: Writes and executes Python code
   to verify/compute solutions
3. **Verifier**: Synthesizes the code executor's output
   and provides the final answer.

The success of the overall system is determined by the
Verifier's final answer.

**Workflow Constraints:**
- Each agent has a 4K token output limit. Outputs beyond
  this are truncated.
- All agents can see the original problem statement.

When evaluating, consider whether the agent:
- (IMPORTANT) Performed well given their role and the
  information provided
- Stayed within output length limit

**Problem that the overall system is solving:**
{problem}

**Current agent being evaluated:**
{agent_role}

**What the agent received as input:**
{agent_input}

**What the agent outputted:**
{agent_output}

**Environment feedback for the agent's tool call
(if applicable):**
{tool_observation}

**Ground truth answer (if applicable):**
{ground_truth_answer}

Evaluate the agent's decision quality based on:
- Given the agent's role and the information it had as
  input, how helpful is its output towards the overall
  system's success?
- For Code Executor: Did it output code in the right
  format? Is the code syntactically correct and does it
  solve the problem?
- For Verifier: Did it correctly synthesize the
  information and output the final answer?

Strictly assigns a score from 0 (terrible) to 10 (perfect).

Additionally, if ground truth is provided (not "N/A") and
the agent is "Verifier", you MUST also verify answer
```

```
correctness:

Provide binary correctness as:
- ANSWER_CORRECT: 1 (if the Verifier's final answer
  matches the ground truth)
- ANSWER_CORRECT: 0 (if the Verifier's final answer
  is incorrect)

Structure your output in the following format exactly:

PROCESS_SCORE: [0 to 10]
ANSWER_CORRECT: [0 or 1, but only if ground truth
provided and agent is Verifier]
```

C DSBENCH AGENT PROMPTS

This section provides the complete prompts used by each agent in the DSBench pipeline. Prompts include placeholders for task-specific information and file system state.

C.1 DATA ENGINEER PROMPT

```
You are a Data Engineer in a data science pipeline.

**CRITICAL: Keep your <think> reasoning BRIEF (under
1000 words). Then IMMEDIATELY write executable code in
a ```python block. Do NOT describe code in thinking -
just write it!**

**Task Description:**
{task_description}

## CRITICAL FILE SYSTEM OBSERVATION
**Files currently available in your workspace:**
{available_files_list}

NOTE: The file names shown above are the ACTUAL files
you can access. Always check this list!

**Your Responsibilities:**
1. **Load Data**: Read training data from 'train.csv'
   and test data from 'test.csv'
2. **Exploratory Data Analysis (EDA)**:
   - Inspect schema (dtypes, shape, missing values)
   - Analyze distributions, correlations
   - Identify data quality issues
3. **Data Preprocessing**:
   - Handle missing values (imputation/removal)
   - Encode categorical variables
   - Scale numerical features
   - Engineer new features based on domain insights
4. **CRITICAL: Save ALL Required Artifacts**:
   - `X_train.pkl` - preprocessed training features
   - `y_train.pkl` - training labels
   - `X_test.pkl` - preprocessed TEST features (CRITICAL!)
   - `scaler.pkl` or similar - fitted transformers
5. **Output Summary**: Print what files you saved

## CRITICAL: You MUST save X_test.pkl!
The Analyst agent downstream will load X_test.pkl to
generate predictions. If you don't save X_test.pkl,
the entire pipeline will FAIL.

**Constraints:**
- Apply SAME transformations to both train and test data
- Do NOT use test data for fitting (no data leakage!)
- Make preprocessing deterministic (set random_state)
- ALWAYS save X_test.pkl - downstream Analyst needs it!
```

C.2 MODELER PROMPT

```
You are a Modeler in a data science pipeline.

**CRITICAL: Keep your <think> reasoning BRIEF (under
1000 words). Then IMMEDIATELY write executable code in
a ```python block.**

**Task Description:**
```

```
{task_description}

## CRITICAL FILE SYSTEM OBSERVATION
**Files currently available in your workspace (saved by
previous agents):**
{available_files_list}

NOTE: The file names shown above are the ACTUAL files
you can load. You MUST use these exact file names!

**Preprocessing Summary from Data Engineer:**
{preprocessing_summary}

**Your Responsibilities:**
1. **Check Available Files FIRST**: Verify which files
   exist before loading
2. **Load Data**: Load the preprocessed data saved by
   Data Engineer
3. **Algorithm Selection**:
   - Analyze the problem type (classification/regression)
   - Consider dataset characteristics
   - Select appropriate algorithms (RandomForest,
     XGBoost, LightGBM)
4. **Model Training**:
   - Train selected models with cross-validation
   - Tune hyperparameters
   - Evaluate performance
5. **Model Selection**: Choose best model based on
   validation metrics
6. **CRITICAL: Save Model**: Save the trained model as
   `model.pkl` - the Analyst needs this!
7. **Output**: Print files saved and performance metrics

**Constraints:**
- Check available files BEFORE trying to load them
- Set random_state for reproducibility
- Use cross-validation for model selection
- Report validation metrics honestly
- ALWAYS save your trained model to model.pkl
```

C.3 ANALYST PROMPT

```
You are an Analyst in a data science pipeline.

**CRITICAL: Keep your <think> reasoning BRIEF (under
1000 words). Then IMMEDIATELY write executable code.**

**Task Description:**
{task_description}

## CRITICAL FILE SYSTEM OBSERVATION
**Files currently available in your workspace:**
{available_files_list}

NOTE: The file names shown above are the ACTUAL files
you can load. You MUST use these exact file names!

**Preprocessing Summary from Data Engineer:**
{preprocessing_summary}

**Model Summary from Modeler:**
{model_summary}

**Your Responsibilities:**
```

```
1. **CHECK AVAILABLE FILES FIRST**: Look at the file
   list above CAREFULLY!
   - You NEED `model.pkl` to make predictions
   - You NEED `X_test.pkl` with preprocessed test features
   - If these files are MISSING, report the issue!
2. **Load Saved Artifacts**: Load the model and
   preprocessed test data
3. **Generate Predictions**: Make predictions on test set
4. **Get IDs from test.csv**: Load original test.csv
   to get the ID column
5. **Create Submission**:
   - Save predictions to 'submission.csv'
   - MUST include correct ID column and prediction column
   - Follow competition format
6. **Quality Checks**:
   - Validate no missing predictions
   - Check value ranges
   - Verify column names and order

## CRITICAL: Check file availability before loading!
If `X_test.pkl` or `model.pkl` is NOT in the available
files list, you CANNOT create submission.csv.

**Constraints:**
- CHECK THE FILE LIST ABOVE before trying to load
- If required files are missing, report which ones
- Load the model from model.pkl
- Load preprocessed test data from X_test.pkl
- Get IDs from original test.csv
- Ensure predictions match test set size
- Save predictions to submission.csv
```

C.4 Coach Evaluation Prompt

The DSBench coach uses a detailed prompt that includes root-cause analysis for pipeline failures. Key excerpts:

```
You are evaluating a specific agent in a multiagent
data science system. The system has 3 specialized agents:

1. **Data Engineer**: Performs EDA, data cleaning,
   feature engineering
   - MUST save: X_train.pkl, y_train.pkl, X_test.pkl
2. **Modeler**: Selects algorithms, trains models
   - MUST save: model.pkl
3. **Analyst**: Generates predictions, creates submission
   - MUST save: submission.csv (FINAL DELIVERABLE)

## CRITICAL: OUTCOME-BASED ROOT CAUSE ANALYSIS

If ground_truth_answer shows "ERROR: No predictions
found" or similar failure, this means submission.csv
was NOT created. This is a PIPELINE FAILURE.

You MUST perform root cause analysis:

1. **Check "FILES SAVED BY PREVIOUS AGENTS"**:
   - Did Data Engineer save X_test.pkl?
   - Did Modeler save model.pkl?
   - If these files are missing, the UPSTREAM AGENT
     is at fault!

2. **Check "tool_observation" for errors**:
   - FileNotFoundError for X_test.pkl -> Data Engineer's
```

```
     fault
   - FileNotFoundError for model.pkl -> Modeler's fault
   - KeyError, TypeError -> Likely Analyst's own bug

3. **Assign blame to the responsible agent**:
   - Data Engineer's fault if: X_test.pkl is NOT in
     "FILES SAVED BY PREVIOUS AGENTS"
   - Modeler's fault if: model.pkl is NOT in
     "FILES SAVED BY PREVIOUS AGENTS"
   - Analyst's fault if: Both files were available but
     Analyst's code is buggy

## Scoring Guidelines

**Score 10**: EXCEPTIONAL - Agent exceeded expectations
**Score 8-9**: Code executed successfully, all required
  files saved, methodology is sound
**Score 6-7**: Code executed, most files saved, minor
  issues
**Score 4-5**: Agent tried but code had errors
**Score 2-3**: Code crashed due to upstream failures
**Score 0-1**: Agent produced no useful output

## Using Ground Truth Metrics (Analyst Only)

If ground truth metrics are provided, apply HARD CAPS:

**Classification Tasks:**
| ROC-AUC >= 0.85 | Max 9 |
| ROC-AUC 0.75-0.85 | Max 8 |
| ROC-AUC 0.65-0.75 | Max 6 |
| ROC-AUC 0.55-0.65 | Max 4 |
| ROC-AUC < 0.55 | Max 3 |

**Regression Tasks (RMSE as % of range):**
| RMSE < 10% | Max 9 |
| RMSE 10-25% | Max 8 |
| RMSE 25-50% | Max 6 |
| RMSE > 50% | Max 4 |

Output format:
PROCESS_SCORE: [0 to 10]
```

D EXAMPLE COACH EVALUATIONS

This section presents real coach evaluation responses from training runs, illustrating how the coach provides dense, context-aware feedback that goes beyond simple correctness checking.

D.1 MATHCHAT: PROBLEM SOLVER EVALUATION

The following coach response evaluates a Problem Solver agent attempting an AMC geometry problem about counting rectangles inscribed in a regular dodecagon. This example demonstrates how the coach rewards metacognitive monitoring—the agent's ability to test and revise its own hypotheses.

```
PROCESS_SCORE: 0.8

**Evaluation:**

1. **Reasonableness of the Action:** The agent's action,
   which is to provide a step-by-step reasoning process,
   is perfectly aligned with its role as Problem Solver.
   The reasoning itself is logical and demonstrates a
   good attempt to solve the problem.

2. **Quality of Reasoning:**
   * **Initial Insight:** The agent has a crucial insight
     early on: a rectangle inscribed in a circle (which
     the dodecagon is) must have its diagonals as
     diameters of the circle. This is the correct path
     to the solution.
   * **Flawed Premise:** The agent then makes a common
     mistake, assuming that for the quadrilateral to be
     a rectangle, the two diameters must be perpendicular.
     This is incorrect; the quadrilateral formed by the
     endpoints of *any* two distinct diameters is always
     a rectangle.
   * **Excellent Self-Correction:** The agent then
     intelligently uses the example provided in the asy
     code to test its own hypothesis. It correctly
     identifies the vertices of the example rectangle
     and determines that its diagonals are not
     perpendicular. This leads it to correctly conclude
     that its premise was wrong. This act of testing a
     hypothesis against evidence and revising its
     thinking is a sign of a very strong reasoning
     process.
   * **Getting Stuck:** After correctly abandoning the
     "perpendicular diameters" idea, the agent
     unfortunately also abandons the "diameters" idea
     altogether. It switches to a much more complex
     method of analyzing perpendicular sides and gets
     stuck. It failed to realize that its original
     insight was correct and only the "perpendicular"
     constraint needed to be removed.

**Conclusion:**

The agent demonstrated a strong, logical thought process.
It identified the most promising approach, made a mistake,
but then impressively corrected that mistake based on the
provided evidence. Its main failure was not revisiting
the original, correct approach after removing the flawed
constraint. Despite not reaching the final answer, the
quality of the step-by-step reasoning and self-correction
is high.
```

This example illustrates how the coach rewards metacognitive monitoring: when the agent tests its hypothesis against evidence and revises its thinking, this is explicitly praised as "a sign of a very strong reasoning process." The coach also distinguishes between the quality of the initial insight (correct), the intermediate error (flawed premise), the recovery attempt (excellent self-correction), and the final outcome (got stuck). This granular feedback enables agents to learn which reasoning behaviors to reinforce.

D.2 DSBENCH: DATA ENGINEER EVALUATION

The following coach response evaluates a Data Engineer agent preprocessing a healthcare dataset for stroke prediction. This example demonstrates how the coach identifies subtle implementation bugs that would silently corrupt the pipeline.

```
PROCESS_SCORE: 0.6

**Evaluation of agent_data_engineer:**

**1. Reasoning and Plan:**
The agent's thought process is logical and structured. It
correctly identifies the key responsibilities: loading
data, performing EDA, handling missing data, encoding
categoricals, scaling numericals, and engineering new
features. The agent correctly reasons about the need for
deterministic preprocessing and considers the appropriate
order of operations.

**2. Code Implementation and Preprocessing:**

* **Strengths:**
  - The use of sklearn.pipeline.Pipeline and
    sklearn.compose.ColumnTransformer is a best practice.
    It encapsulates preprocessing steps cleanly and helps
    prevent data leakage.
  - The choice of preprocessing steps is appropriate:
    mean imputation for numerical, constant fill for
    categorical, StandardScaler for numerical, and
    OneHotEncoder for categorical.
  - The agent correctly includes handle_unknown='ignore'
    in the OneHotEncoder, which is crucial for handling
    potential new categories in the test set.
  - The agent correctly sets random_state=42 to ensure
    the process is deterministic.
  - The feature engineering ideas (age_group, bmi_over_age)
    are plausible and relevant for a stroke prediction
    task.

* **Weaknesses / Bugs:**
  - **Critical Flaw:** The agent defines numerical_cols
    and categorical_cols *before* creating the new
    features (age_group, bmi_over_age). Consequently, the
    ColumnTransformer is configured to only process the
    *original* columns. The newly created features will
    be dropped by the transformer, completely nullifying
    the feature engineering step. The column lists should
    have been defined *after* the new columns were added
    to the DataFrame.
  - **Redundancy:** The agent performs imputation twice.
    First, it imputes directly on the DataFrame using
    .fillna(). Then, it includes SimpleImputer again
    within the pipelines of the ColumnTransformer. While
    not a fatal error, it is inefficient and indicates a
    slight misunderstanding of how to use the pipelines
    effectively.
```

```
**3. Summary Output:**
The agent correctly structures the final output according
to the specified PREPROCESSING_SUMMARY format.

**Conclusion:**
The agent demonstrates a strong conceptual understanding
of the data engineering role and utilizes modern, robust
tools (sklearn pipelines). However, the implementation
contains a critical logical error in the order of
operations that prevents the engineered features from
being correctly processed. This is a significant bug that
would negatively impact the final model. The plan was
excellent, but the execution was flawed.
```

This example demonstrates the coach's ability to identify subtle implementation bugs that would silently corrupt the ML pipeline. The "Critical Flaw"—defining column lists before creating new features, causing engineered features to be silently dropped—is exactly the kind of bug that passes syntax checks but produces incorrect results. The coach also evaluates domain-specific best practices: using `ColumnTransformer` for clean preprocessing, setting `handle_unknown='ignore'` for robust handling of unseen categories, and ensuring deterministic behavior with `random_state`. The conclusion "the plan was excellent, but the execution was flawed" provides actionable feedback that helps the agent learn to verify implementation details match high-level intentions.

D.3 MATHCHAT: STRATEGY VS. IMPLEMENTATION

The following example evaluates a Problem Solver on a grid coloring combinatorics problem, demonstrating how the coach distinguishes between excellent high-level strategy and flawed low-level modeling.

```
PROCESS_SCORE: 0.6

This is a very strong attempt at a complex combinatorial
problem. The agent's overall strategy is excellent, but
there is a critical flaw in its understanding of the
grid's structure that prevents it from reaching the
correct conclusion.

**Evaluation of the Agent's Reasoning:**

1. **Problem Decomposition (Excellent):** The agent
   correctly identifies the 12 line segments and astutely
   categorizes them into 8 "outer" segments (on the
   perimeter) and 4 "internal" segments (shared between
   squares). This is a key insight that simplifies the
   problem.

2. **Strategic Approach (Excellent):** The agent devises
   a powerful strategy:
   - First, consider all possible colorings of the 4
     internal segments.
   - Second, for each internal coloring, calculate the
     number of ways the 8 outer segments can be colored
     to satisfy the constraints for each of the four
     squares.
   - Finally, sum the results over all possible internal
     colorings.
   This is a systematic and valid way to solve the
   problem.

3. **Modeling the Grid Structure (Flawed):** This is
   where the agent makes a critical error. It incorrectly
   assumes the grid can be split into two independent
   columns.
```

```
   - **Agent's flawed model:** It states that the
     top-left and bottom-left squares share two internal
     edges, and the top-right and bottom-right squares
     share the other two internal edges. This would mean
     the left column is completely independent of the
     right column.
   - **Correct model:** The internal edges connect all
     four squares. For example, the top-middle vertical
     segment is shared by the top-left and top-right
     squares, linking the two columns.

4. **Execution of the Flawed Model (Good):** Given its
   incorrect assumption, the agent's subsequent reasoning
   is logical and well-executed. It correctly sets up
   the calculation based on its flawed model. If the
   initial model had been correct, this method would
   have yielded the right answer for that model.

**Conclusion:**

The agent demonstrated a sophisticated approach to
problem-solving. The strategy of breaking down the
problem based on the more constrained "internal" edges
is exactly the right way to tackle this. However, the
fundamental misunderstanding of the grid's connectivity
is a major flaw that makes the final answer incorrect.
The high-level strategy is excellent, but the low-level
implementation is based on a false premise.
```

This example shows the coach's nuanced evaluation: "Problem Decomposition (Excellent)," "Strategic Approach (Excellent)," but "Modeling the Grid Structure (Flawed)." The coach explicitly notes that the agent's execution of its flawed model was internally consistent—"if the initial model had been correct, this method would have yielded the right answer." This distinction between strategic quality and modeling accuracy provides precise feedback for improvement.

D.4 DSBENCH: ANALYST PIPELINE FAILURE

The following example evaluates an Analyst agent that fundamentally misunderstands how to apply a trained ML pipeline to test data—a critical conceptual error in production ML systems.

```
PROCESS_SCORE: 0.1

**Critique of the Agent's Action:**

1. **Reproducing the Pipeline (Major Failure)**: The
   agent's primary responsibility is to apply the *exact*
   same preprocessing and use the *exact* same trained
   model from the previous steps. The agent fails
   critically on this point:
   - **LabelEncoder Error**: The code le = LabelEncoder()
     and le.fit_transform(test_df['...']) creates a *new*
     encoder and fits it *only on the test data*. This is
     incorrect. The encoder must be the one that was
     fitted on the training data to ensure consistent
     mapping of categories to numbers.
   - **TfidfVectorizer Error**: The code initializes a
     new, unfitted vectorizer. Calling .transform()
     without first calling .fit() on the training data
     will cause a NotFittedError. The agent should have
     used the vectorizer object that was already fitted
     by the Data Engineer/Modeler.
   - **Model Retraining Error**: The code re-initializes
     a new RandomForestClassifier and then attempts to
     retrain it with model.fit(X_train, y_train). This
```

```
     is a severe misunderstanding of the agent's role.
     The Analyst should use the *already trained* model
     from the Modeler. Furthermore, the variables
     X_train and y_train are not defined within the
     Analyst's script, meaning the code would crash
     with a NameError.

2. **Generating Predictions (Failure)**: Due to the
   errors mentioned above, the provided script is not
   executable. It would fail at the tfidf.transform
   step, and even if it didn't, it would fail at the
   model.fit step. Therefore, it cannot generate any
   predictions.

3. **Creating Submission (Partial Success)**: The logic
   for creating the final submission DataFrame and
   printing it in the required format is syntactically
   correct. However, this is useless since the
   predictions themselves cannot be generated.

**Conclusion:**

The agent demonstrated a fundamental misunderstanding of
how to apply a trained machine learning pipeline to new,
unseen data. The concepts of using pre-fitted transformers
(like LabelEncoder and TfidfVectorizer) and a pre-trained
model were completely missed. The resulting code is
non-functional and conceptually wrong.
```

This example demonstrates the coach identifying fundamental conceptual errors about ML deployment. The agent's mistake—refitting encoders on test data and retraining the model—would cause category mapping inconsistencies and invalidate the entire pipeline. This is precisely the kind of error that distinguishes production-ready ML practitioners from novices. The coach's detailed breakdown (LabelEncoder error, TfidfVectorizer error, Model Retraining error) provides specific, actionable feedback about each conceptual gap.

E TRAINING ALGORITHM DETAILS

This appendix provides additional technical details for the training algorithm described in Section 2.3.

E.1 WHY REINFORCE++ OVER GRPO

Group-Relative Policy Optimization (GRPO) normalizes advantages within groups of samples sharing the same prompt, which assumes identical input states across the group. This assumption is valid for single-agent settings where all samples from the same prompt begin in the same state. However, in end-to-end multiagent training, each agent's input context depends on the stochastic output of upstream agents. Consider two rollouts of the same math problem: the Problem Solver may generate different reasoning chains, causing the Code Executor to observe different inputs despite originating from the same initial prompt. This state divergence makes within-group comparisons ill-defined for GRPO.

An alternative approach—training each agent separately with frozen upstream context—would restore the same-state assumption. However, this requires separate rollout passes for each agent, significantly reducing efficiency and preventing agents from co-adapting during training. Instead, we train all agents end-to-end from complete trajectories using REINFORCE++, which applies global batch normalization across all collected experiences rather than per-prompt normalization.

E.2 ADVANTAGE ESTIMATION

The advantage estimation proceeds in two steps. First, for each action a_t with coach reward $r_t^{\text{coach}} \in [0, 1]$, we compute the KL-penalized reward:

$$r_t = r_t^{\text{coach}} - \beta \cdot D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \tag{2}$$

where $\beta = 0.01$ is the KL penalty coefficient and D_{KL} measures divergence from the frozen reference policy. The advantage for each action is then computed as undiscounted return-to-go:

$$A_t = \sum_{\tau \geq t} r_\tau \tag{3}$$

which propagates downstream rewards back to earlier actions within each agent's trajectory. This choice of $\gamma = 1$ (no discounting) reflects the finite-horizon nature of multiagent workflows where all actions contribute equally to the final outcome.

Second, we collect advantages A_t from all agents and all experiences in the current batch, then compute global statistics:

$$\mu = \frac{\sum_{i,t} A_t^{(i)} \cdot m_t^{(i)}}{\sum_{i,t} m_t^{(i)}}, \quad \sigma^2 = \frac{\sum_{i,t} (A_t^{(i)} - \mu)^2 \cdot m_t^{(i)}}{\sum_{i,t} m_t^{(i)}} \tag{4}$$

where $m_t^{(i)}$ are action masks indicating valid actions. Advantages are then normalized: $\hat{A}_t = (A_t - \mu)/\sqrt{\sigma^2 + \epsilon}$ with $\epsilon = 10^{-8}$ for numerical stability.

E.3 POLICY GRADIENT OBJECTIVE

The policy gradient objective follows PPO with clipped surrogate:

$$\mathcal{L}_{\text{policy}} = -\mathbb{E}\Big[\min\Big(\rho_t(\theta)\hat{A}_t, \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big)\Big] \tag{5}$$

where $\rho_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\text{old}}(a_t|s_t)$ is the probability ratio between the current and old policy, and $\epsilon = 0.2$ is the clipping range. The clipping prevents excessively large policy updates that could destabilize training, which is particularly important when training multiple agents simultaneously where instability in one agent can cascade through the pipeline.

E.4 STRATEGIC MULTI-OBJECTIVE JUDGING

Our current coach evaluates each action independently without memory of previous training epochs or access to aggregate performance statistics. Each action is scored based solely on its intrinsic quality, without considering whether the training process would benefit more from improving reliability versus refinement at that stage.

A strategic coach with memory could balance objectives dynamically. For example, such a coach could:

- Assign higher rewards to successful task completion early in training when the pipeline is unreliable
- Shift focus to quality improvements once success rate stabilizes
- Restore emphasis on reliability if it degrades

Such a coach could access metrics like rolling success rate, quality trends, and per-agent failure modes to make context-aware reward decisions across the training trajectory. Implementing strategic coaching in our framework would be straightforward: the coach prompt already accepts arbitrary metadata fields that could include per-epoch statistics. We leave implementation and empirical evaluation of strategic multi-objective judging to future work.

F REWARD BACKPROPAGATION

This appendix formalizes *Reward Backpropagation*, an approach for outcome-aware process reward decomposition.

F.1 MOTIVATION

Process rewards without outcome information are inherently inefficient: without knowing what actually mattered, a coach must critique everything that *could* be improved—some feedback may be valid, some may be mere stylistic preference. This bottom-up approach asks each agent to "improve my part to my best effort" without knowing which improvements actually matter for the final result.

Outcome-aware evaluation inverts this. Given a specific outcome (e.g., incorrect predictions, or surprisingly good results despite upstream errors), it becomes much easier to trace backward and identify which agent made which mistake, or whose action saved the team. This top-down approach asks: "given this outcome, who is the real contributor to success or failure?" The linear backpropagation structure provides an ordered flow of attribution—not a chaotic discussion—where each step receives credit or blame only for the residual not yet explained by downstream actions.

Reward Backpropagation formalizes this shift from local, bottom-up optimization to global-aware, top-down attribution.

F.2 METHOD OVERVIEW

Given a trajectory $\tau = (s_1, s_2, \ldots, s_T)$ and final outcome $R_T \in \mathbb{R}^k$ (potentially vector-valued), Reward Backpropagation proceeds in two phases:

Forward Pass (Process Rewards). For each step t, query a coach to produce a process reward p_t evaluating local decision quality without knowledge of the final outcome.

Backward Pass (Outcome Decomposition). Starting from R_T, run a backward chain of coach calls:

1. At step T, the coach attributes part of R_T to step T, producing contribution Δ_T and passing residual r_{T-1} backward
2. At step $t < T$, the coach receives residual r_t, attributes part to step t (producing Δ_t), and passes $r_{t-1} = r_t - a_t$ backward
3. This continues to step 1, yielding per-step outcome-aware contributions $\{\Delta_t\}$

Implementation Note. The notation above is *conceptual* rather than literal. In practice, the "residual" r_t and "subtraction" $r_{t-1} = r_t - a_t$ are not scalar or vector arithmetic but rather *natural language attribution*. Each backward step asks the coach to explain, in natural language, how the output of agent $t-1$ (which forms part of the input to agent t) contributed to the outcome being evaluated. For example, the coach might produce feedback such as: "Agent $t-1$'s output contained attribute X, which caused issue Y at agent t. I consider X to be a problematic decision, worth -3 out of 10." The "subtraction" thus represents conceptual decomposition—the coach articulates what portion of the outcome can be attributed to each step, with the residual being the *remaining unexplained outcome* passed backward for further attribution. This natural language grounding makes the process interpretable and allows rich, multi-dimensional feedback beyond what scalar rewards can capture.

F.3 COMBINING PROCESS AND OUTCOME SIGNALS

The final step-level reward combines both signals:

$$f_t = \alpha \cdot p_t + \beta \cdot \Delta_t \tag{6}$$

where $\alpha, \beta \geq 0$ are hyperparameters. Process reward p_t captures local quality; outcome contribution Δ_t captures how the final result validates or refutes that quality. Steps whose local quality did not affect the outcome receive small $|\Delta_t|$.

F.4 CONSISTENCY CONSTRAINTS

We encourage approximate decomposition consistency:

$$\sum_{t=1}^{T} \Delta_t \approx v(R_T) \tag{7}$$

where $v : \mathbb{R}^k \to \mathbb{R}$ scalarizes the outcome vector. This can be enforced via prompting or post-hoc normalization.

F.5 BENEFITS

Reward Backpropagation provides: (1) causally plausible credit assignment—locally good actions in failed trajectories receive negative Δ_t; (2) interpretable narratives about which steps mattered; (3) robustness to process-reward hacking since Δ_t anchors rewards to actual outcomes. The approach is compatible with REINFORCE++ and can be distilled into a lightweight Process Reward Model for efficiency.

G DISTRIBUTED TRAINING IMPLEMENTATION

This appendix provides detailed implementation specifications for the distributed training architecture described in Section 2.

G.1 PARALLEL AGENT INITIALIZATION

Each agent is initialized via `ThreadPoolExecutor` to parallelize model loading across agents:

```
with ThreadPoolExecutor(max_workers=len(agent_configs)) as executor:
    futures = [executor.submit(self._init_agent, agent_id, config, args)
               for agent_id, config in agent_configs]
    agents = [f.result() for f in futures]
```

Each agent instantiates independent Ray actor groups for:

- **vLLM Engines**: `num_engines` × `tensor_parallel_size` Ray actors for autoregressive generation with PagedAttention and KV-cache optimization
- **Actor Policy**: `PPORayActorGroup` with `actor_num_nodes` × `actor_num_gpus_per_node` workers for trainable policy parameters
- **Reference Model**: Same structure as actor, frozen copy for KL divergence computation
- **Critic Model**: Value function estimation distributed across ranks (optional, not used with REINFORCE++)

When `colocate_all_models=True`, placement groups bundle multiple models on shared GPUs with fractional allocation (`num_gpus_per_actor=0.2`), reducing peak memory by enabling actor, reference, and vLLM models to share GPU memory.

G.2 DISTRIBUTED ROLLOUT GENERATION

Prompts are sharded evenly across `world_size` workers:

```
chunk_size = (num_prompts + world_size - 1) // world_size
for rank in range(world_size):
    start_idx, end_idx = rank * chunk_size, min((rank+1) * chunk_size, num_prompts)
    chunked[rank] = prompts[start_idx:end_idx]
```

Each rank receives its shard and executes workflows via Ray remote calls:

```
all_refs = []
for rank in range(world_size):
    ref = generate_samples_remote.remote(world, chunked[rank], rank, world_size)
    all_refs.append(ref)
results = ray.get(all_refs)  # Barrier: wait for all ranks
```

Within each workflow, LLM engines are assigned cyclically: if fewer engines than ranks, engine i handles rank $i \mod$ num_engines; otherwise engines are striped across ranks. Requests are batched across available engines:

```
batch_size = (len(prompts) + len(llms) - 1) // len(llms)
for i, llm in enumerate(llms):
    batch = prompts[i * batch_size : (i + 1) * batch_size]
    refs.append(llm.generate.remote(batch, sampling_params))
```

G.3 EXPERIENCE PREPARATION AND AGENT ROUTING

Completed trajectories are processed by `ExperienceMaker`, which computes per-action rewards, KL divergence, and advantages. For multiagent training, each action is routed to its originating agent:

```
sharded_data = [[None] * world_size for _ in range(num_agents)]
for rank in range(world_size):
```

```
    for agent_id in range(num_agents):
        agent_data = all_results[rank][agent_id]  # Extract agent's batch
        sharded_data[agent_id][rank] = ray.put(agent_data)
```

This enables each agent to train on its own trajectory segments while maintaining batch synchronization across the distributed system.

Variable Turn Count Handling. When agents have variable turn counts (e.g., Code Executor in MathChat with up to 5 turns), different workers may produce different numbers of samples, causing NCCL deadlocks during gradient synchronization. The `filter_agents_data` option truncates all workers to the minimum sample count:

```
# Count actual samples per rank for each agent
actual_samples_per_rank = [[count_samples(rank, agent)
                            for agent in range(num_agents)]
                           for rank in range(world_size)]

# Truncate to minimum across ranks to ensure equal gradient syncs
min_samples = [min(rank[agent] for rank in actual_samples_per_rank)
               for agent in range(num_agents)]
```

This is critical for MathChat where `coder_max_turns=5` causes variable action counts across workers.

Global Batch Normalization (REINFORCE++). Advantages are normalized across the entire batch, not per-prompt:

```
all_advantages = torch.cat([exp.advantages for exp in experiences])
mean = (all_advantages * action_masks).sum() / num_actions
var = ((all_advantages - mean).pow(2) * action_masks).sum() / num_actions
rstd = var.clamp(min=1e-8).rsqrt()
normalized = (all_advantages - mean) * rstd
```

G.4 Weight Synchronization

After each training step, updated weights must be broadcast from DeepSpeed actors to vLLM engines. The synchronization mechanism depends on `colocate_all_models`:

Standard NCCL Broadcast. When models are on separate GPUs, a dedicated NCCL process group connects DeepSpeed rank 0 with all vLLM engine ranks:

```
# Process group: [DeepSpeed rank 0, vLLM engine 0, engine 1, ...]
world_size = vllm_num_engines * vllm_tensor_parallel_size + 1

# Broadcast each parameter (ZeRO-3 requires AllGather first)
for name, param in model.named_parameters():
    with deepspeed.zero.GatheredParameters([param], enabled=zero_stage==3):
        torch.distributed.broadcast(param.data, src=0, group=model_update_group)
```

CUDA IPC (Co-located Models). When `colocate_all_models=True` (default for both MathChat and DSBench), the system automatically selects CUDA IPC for zero-copy parameter sharing, which is faster than NCCL for models on the same GPU:

```
# Automatically enabled when colocate_all_models=True and backend="nccl"
weight = param.data.clone()
ipc_handle = reduce_tensor(weight)  # Get CUDA IPC handle
ipc_handle = {get_physical_gpu_id(): ipc_handle}
torch.distributed.all_gather_object(ipc_handle_list, ipc_handle)
# vLLM engines reconstruct tensor from IPC handle without data copy
```

G.5 MEMORY OPTIMIZATION

Prefix Caching. vLLM reuses KV cache across requests with common prefixes (e.g., system prompts shared across agents). Cache is reset after weight updates: `engine.reset_prefix_cache.remote()`.

Sleep/Wake Mode. vLLM engines can offload to CPU during training to free GPU memory:

```
def sleep(self, level=1):
    self.llm.sleep(level=level)  # Offload to CPU
def wake_up(self):
    self.llm.wake_up()           # Reload to GPU
```

DeepSpeed ZeRO-3. Optimizer states, gradients, and parameters are partitioned across GPUs. Parameters are gathered on-demand during forward/backward passes, enabling training of models larger than single-GPU memory.

G.6 SYNCHRONIZATION POINTS

Table 5 summarizes the key synchronization barriers in the training loop.

Table 5: Communication patterns and synchronization points in distributed training.

Component	Protocol	Frequency
Prompt sharding	Ray `put`/`get`	Per iteration
Rollout collection	Ray `remote`/`get`	Per iteration
Reference model inference	Ray `remote`	Per trajectory
Gradient reduction	NCCL `all_reduce`	Per training step
Weight broadcast	NCCL `broadcast`	Per training step
Metric aggregation	NCCL `all_reduce`	Per training step

G.7 CONFIGURATION PARAMETERS

Table 6 lists key hyperparameters controlling distributed training behavior.

Table 6: Key configuration parameters for distributed training.

Parameter	Default	Description
`actor_num_nodes`	1	Nodes for policy training
`actor_num_gpus_per_node`	2	GPUs per node for policy
`vllm_num_engines`	2	vLLM inference engines
`vllm_tensor_parallel_size`	1	Tensor parallelism per engine
`vllm_gpu_memory_utilization`	0.7	GPU memory for KV cache
`colocate_all_models`	True	Share GPUs across model types
`zero_stage`	3	DeepSpeed ZeRO level
`vllm_sync_backend`	nccl	Weight sync backend

G.8 TASK-SPECIFIC CONFIGURATION

Table 7 compares the configurations used for MathChat and DSBench experiments.

Key differences: DSBench uses longer generation limits (16K vs 4K) to accommodate multi-step data science pipelines, trains for more episodes (30 vs 8) due to smaller dataset size, and uses a more capable coach model (Gemini 3 Pro) for complex ML evaluation. Both use `filter_agents_data=True` to handle variable turn counts across workers.

G.9 COMPUTATIONAL COST

Hardware. All experiments use a single node with 8× NVIDIA H100 GPUs.

Table 7: Configuration differences between MathChat and DSBench.

Parameter	MathChat	DSBench
rollout_batch_size	32	16
n_samples_per_prompt	2	2
num_episodes	8	30
generate_max_len	4096	16384
prompt_max_len	24576	24576
coach_model	gemini-2.5-flash	gemini-3-pro-preview
coder_max_turns	5	N/A
filter_agents_data	True	True
vllm_gpu_memory_utilization	0.7	0.6

Training Time. Wall-clock training time is approximately 8–12 hours for 106 training steps on MathChat, with coach API calls being the primary bottleneck (each coach evaluation adds $\sim$2–5 seconds latency). Each rollout generates $\sim$3–9 coach calls (3 agents $\times$ 1–3 turns per agent), totaling $\sim$3,000–10,000 coach calls per training run. DSBench training (30 episodes $\times$ 128 rollouts) incurs similar per-episode costs but with longer per-task execution times due to multi-turn tool use.

API Cost. Using Gemini 2.5 Flash at approximately \$0.075 per 1M input tokens and \$0.30 per 1M output tokens, the estimated API cost is \$50–150 per training run.