DAPF: A DYNAMIC AGENT-DRIVEN AUTOMATIC POST-TRAINING FRAMEWORK

Anonymous authors
Paper under double-blind review

ABSTRACT

Domain-specific fine-tuning of large language models requires repeated decisions about training configurations, data schedules, and recovery from unsuccessful experiments. Automating these decisions requires connecting training feedback to actionable interventions while accounting for limited computational resources. We introduce DAPF, an agent-driven framework for closed-loop post-training of language and vision-language models. Given a target model, task domain, and validation performance target, DAPF iteratively generates hypotheses, designs and executes experiments, diagnoses training outcomes, and selects subsequent interventions. Its attribution module combines loss-pattern analysis with agent-generated diagnoses to propose targeted adjustments and update estimates of intervention effectiveness from experimental feedback. A budget-aware selector prioritizes expected validation improvement per unit cost and chooses the least-cost action predicted to reach the target when available. An append-only evidence graph records experimental decisions and supports recovery of model checkpoints and agent state. We evaluate DAPF on seven benchmarks using Qwen3-1.7B for text tasks and Qwen2.5-VL-3B for multimodal reasoning. Across three independent runs, DAPF achieves higher mean test accuracy than FT-Agent on every benchmark, with an average absolute gain of 1.74 percentage points. Reported average GPU runtime decreases from 1.5 to 1.3 hours, while the average number of effective optimization rounds decreases from 4.9 to 3.8. These results suggest that feedback-guided intervention selection can improve fine-tuning performance and computational efficiency within a bounded search space.

1 INTRODUCTION

Domain-specific fine-tuning requires repeated decisions about hyperparameters, training schedules, and recovery from failed experiments. LLM agents increasingly automate experiment proposal, execution, and interpretation (Yano et al., 2025; Li et al., 2026; A-EVO-Lab, 2026; Yu et al., 2026). A central challenge is connecting the evidence from each experiment to a useful next intervention under limited compute.

Automation alone does not explain why a training run improves. Changes to hyperparameters, data scheduling, and computational effort can confound comparisons. Experiment records must therefore connect hypotheses, configurations, training traces, and evaluation results. Long-running workflows must also preserve the state required to resume an interrupted optimization loop.

We introduce DAPF, an agent-driven framework that takes a base model, a target domain, and a validation-performance requirement. Its six stages cover requirement analysis, hypothesis generation, experiment design, training execution, evaluation, and iterative decision-making. Loss-pattern analysis and agent-generated diagnoses guide candidate interventions, while a budget-aware policy prioritizes predicted validation improvement per unit cost. An append-only evidence graph records decisions and artifacts, and a watchdog supervises training execution.

We evaluate DAPF against manual fine-tuning, manual curriculum learning, and FT-Agent on seven benchmarks. An attribution-ablated variant examines the contribution of diagnostic feedback. The study focuses on feedback-guided optimization within a bounded search space.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2 Related Work

Automated machine-learning research. MLE-Bench (Chan et al., 2025) and DSBench (Jing et al., 2024) evaluate agents on machine-learning and data-science workflows. R&D-Agent (Xu et al., 2025) organizes experiment development through modular agents. A related line uses language models for neural architecture search, including LLMatic (Nasir et al., 2023), CoLLM-NAS (Li et al., 2025), PEL-NAS (Zhu et al., 2025b), LLM-NAS (Zhu et al., 2025a), and closed-loop architecture design (Khalid et al., 2026). DAPF instead studies post-training decisions for fixed model backbones.

Self-improving language models and agents. SELF (Lu et al., 2023) and SEAL (Zweiger et al., 2025) explore adaptation through model-generated feedback or training data. Agent0 (Xia et al., 2025) uses interacting agents for curriculum and execution, while EvolveR (Wu et al., 2025) accumulates reusable experience. SIA (Hebbar et al., 2026) considers changes to both external agent components and model weights. These approaches motivate feedback-driven adaptation; our focus is recording and selecting interventions within a constrained fine-tuning workflow.

Automated post-training. LaMDAgent (Yano et al., 2025), FT-Dojo (Li et al., 2026), A-Evolve-Training (A-EVO-Lab, 2026), and AutoTrainees (Yu et al., 2026) address agent-based post-training or its evaluation. DAPF brings training diagnosis, explicit compute accounting, experiment provenance, and recovery into one loop. Our experiments compare DAPF with FT-Agent and manual fine-tuning baselines.

3 Method

The input of DAPF is a user requirement tuple $\mathcal{I} = (\theta_0, D, r_{\text{tar}})$, where θ_0 denotes the base model, D represents the target fine-tuning domain, and r_{tar} is the required target validation accuracy. Let $R(\theta) = \text{Metric}(f_\theta, \mathcal{D}_{\text{val}})$ denote validation-set performance of model parameter θ. Our objective is to find θ^* under finite computational budget B_0 satisfying $R(\theta^*) \geq r_{\text{tar}}$. All development-stage choices, including hypothesis selection, strategy selection, stopping, and checkpoint selection, use only the training and validation splits. After the strategy and selected checkpoint are frozen, we evaluate the held-out test set once, solely for final reporting. Formally, the overall problem is formulated as a constrained stochastic optimization:

$$\min_{\pi,\tau} \quad \mathbb{E}_\pi \left[\sum_{t=0}^{\tau-1} c(a_t) \right], \tag{1}$$

$$\text{s.t.} \quad \Pr\left(R(\theta_\tau) \geq r_{\text{tar}}\right) \geq 1 - \delta, \tag{2}$$

$$\sum_{t=0}^{\tau-1} c(a_t) \leq B_0. \tag{3}$$

Here a_t is the experiment strategy at round t, $c(a_t)$ denotes its computational cost, π is the strategy-selection policy, and τ is the stopping timestep. This objective guides strategy selection under a finite compute budget. The framework runs through six sequential modules.

3.1 Six-Stage Agentic Closed-Loop Pipeline

DAPF uses six stages (Figure 1). Each stage records its inputs, outputs, and links to preceding artifacts in the evidence chain.

1. **Requirement analysis.** The analyst checks the model, domain, target, and resource constraints, requests missing information, and writes structured requirements to a `UserRequirementNode`.
2. **Hypothesis generation.** The researcher proposes candidates, filters infeasible options, and selects one falsifiable hypothesis. The `HypothesisNode` records a primary variable change, its predicted effect, and a numerical falsification criterion.

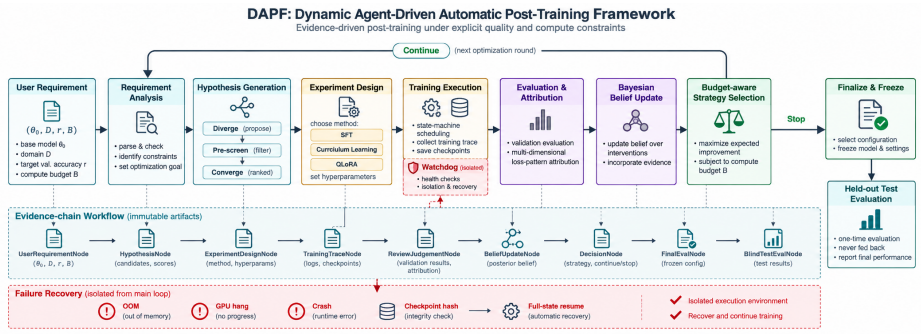

Figure 1: **DAPF workflow.** Six stages connect requirements, hypotheses, experiments, and validation-based decisions. The watchdog supervises execution; an evidence chain records the optimization history. Held-out testing occurs only after the selected configuration is frozen and does not inform subsequent interventions.

3. **Experiment design.** The experimenter converts the hypothesis into an executable configuration, including tuning parameters and any curriculum settings. An `ExperimentDesignNode` records the configuration before execution.
4. **Training execution.** A state machine launches training under an isolated watchdog. It monitors non-finite loss, divergence, stalls, and budget consumption. Logs and checkpoint references are stored in a `TrainingTraceNode`, including for failed runs.
5. **Evaluation and critique.** The critic compares training traces and validation results with the hypothesis, identifies loss patterns, and records diagnoses and their confidence in a `ReviewJudgementNode`.
6. **Decision and update.** Validation performance, remaining budget, and operational stop conditions determine whether to continue. Otherwise, intervention estimates are updated and the next feasible action is selected. At termination, the model and configuration are frozen; a separate record stores one held-out test evaluation that is never fed back to the agents.

3.2 Evidence-chain Workflow: Immutable Evidence Graph

Instead of storing only final checkpoints and flat text logs, DAPF maintains an append-only immutable directed evidence graph, where each stage output becomes a graph node. Nodes are never modified in-place; updates are realized by appending new derived nodes. Each node carries a unique identifier, timestamp, cryptographic content hash for integrity validation, payload fields, and directed edges encoding causal or derivation relationships between nodes.

Node types include: UserRequirementNode, HypothesisNode, ExperimentDesignNode, TrainingTraceNode, ReviewJudgementNode, BeliefUpdateNode, DecisionNode, FinalEvalNode, and BlindTestEvalNode. Binary artifacts (model checkpoints, full datasets) are stored externally; inside the graph we only store file paths and SHA-256 hashes as pointers, avoiding bloating the graph with large binary payloads. Directed edges encode semantic relations: `derive`, `inspire_experiment`, `produce_by`, `analyze_from`, `revise_next_hypothesis`, `terminate_select`, and `postfreeze_evaluate`.

Causal assertions from the critic are embedded inside `ReviewJudgementNode` with explicit confidence levels: L1 for correlation observation, L2 for candidate causal conjecture, and L3 for intervention evidence from controlled probe experiments. When low-confidence attribution is produced, the decision module may schedule intervention probe experiments to assess the proposed explanation, whose traces are also appended into the evidence graph as sub-graph branches.

3.3 Attribution-driven Intervention & Belief Update

Given `ReviewJudgementNode` from critic module, attribution-driven intervention decomposes training outcome or failure into multi-dimensional contributing factors: learning rate configuration,

data quality, method choice (LoRA / curriculum / QLoRA), compute resource failure, and system fault. Each factor is assigned a diagnostic score and confidence value.

We maintain a persistent belief table for candidate intervention actions (e.g., reduce_lr, increase_lora_r, filter_dataset, switch_to_curriculum, run_probe). For each action we store the expected performance gain $\mathbb{E}[\text{gain}]$, estimated failure probability P_{fail}, and belief confidence. After each round, new evidence from the evidence graph updates intervention estimates, generating an immutable `BeliefUpdateNode` recording pre-update belief, referencing evidence node identifiers, and post-update belief values. Belief states are persisted for crash recovery.

Importantly, attribution outputs are treated as conjectures rather than ground-truth causal facts. Low-confidence attributions can trigger intervention probe experiments: we fork from an existing checkpoint, enforce single-variable intervention while keeping all other conditions fixed, and run short training rounds to collect further evidence for belief calibration.

3.4 Budget-aware Dynamic Strategy Selection

At round t, DAPF maintains the current model θ_t, the validation performance $R_t = R(\theta_t)$, and the remaining computational budget

$$B_t^{\text{rem}} = B_0 - \sum_{j=0}^{t-1} c(a_j). \tag{4}$$

The remaining performance gap to the user requirement is defined as

$$g_t = \max\left(r_{\text{tar}} - R_t,\, 0\right). \tag{5}$$

Given the current training trace and attribution results, the framework constructs a candidate strategy set $\mathcal{A}_t$, including hyper-parameter adjustment, curriculum configuration, recovery actions, and early termination. Since each strategy consumes computational resources, only budget-feasible actions are retained:

$$\mathcal{A}_t^{\text{feas}} = \{a \in \mathcal{A}_t \mid \hat{c}_t(a) \le B_t^{\text{rem}}\}, \tag{6}$$

where $\hat{c}_t(a)$ denotes the estimated cost of executing action a.

For each feasible action, the attribution module estimates its expected validation improvement, denoted by $\widehat{\Delta}_t(a)$. To avoid rewarding improvement beyond the required target, we define the effective gain as

$$G_t(a) = \min\left(\widehat{\Delta}_t(a),\, g_t\right). \tag{7}$$

DAPF then selects the action with the highest expected effective improvement per unit cost:

$$a_t = \arg\max_{a \in \mathcal{A}_t^{\text{feas}}} \frac{G_t(a)}{\hat{c}_t(a) + \epsilon}, \tag{8}$$

where $\epsilon > 0$ is a small constant for numerical stability.

When one or more actions are predicted to satisfy the remaining target gap, DAPF instead selects the least-cost sufficient action:

$$a_t = \arg\min_{a \in \mathcal{A}_t^{\text{feas}}} \hat{c}_t(a) \quad \text{s.t.} \quad \widehat{\Delta}_t(a) \ge g_t. \tag{9}$$

After executing a_t, the model and remaining budget are updated as

$$\theta_{t+1} = \text{Train}(\theta_t, a_t, \mathcal{D}_{\text{train}}), \qquad B_{t+1}^{\text{rem}} = B_t^{\text{rem}} - c(a_t). \tag{10}$$

The iterative process terminates when $R_t \ge r_{\text{tar}}$, the budget is exhausted, or no feasible action remains.

3.5 Fault-tolerant Training Scientist

The fault-tolerant subsystem persists model checkpoints and loop state to support recovery after interrupted execution. Recovery starts from the latest valid persisted state. It consists of four components:

- **Watchdog & process-group control**: each training round runs inside an isolated POSIX process group supervised by a watchdog thread. OOM, segmentation fault or hanging are detected; on fault the watchdog aborts the child training process and atomically records the current `ExperimentDesignNode` and partial `TrainingTraceNode` before exiting, marking system-fault attribution inside review node. The main agent loop does not crash together with training child processes.
- **Checkpoint chain index**: an append-only index mapping each stored checkpoint SHA-256 hash to its corresponding `TrainingTraceNode` identifier, reconstructing the chronological chain of model states across iterations.
- **Full-state persistence**: besides the immutable evidence-chain graph, we persist belief state snapshot, remaining GPU budget state, and checkpoint-chain index after each completed iteration decision, using atomic file rename to avoid partial-write corruption.
- **Crash-resume logic**: upon restart, the system loads evidence-chain graph, validates node content hashes for integrity, recovers belief table and remaining budget, locates the last valid completed trace node, loads its corresponding checkpoint, and resumes iteration from the recovered state.

3.6 Loss-Pattern Attribution

After each training round, the critic invokes a deterministic classifier over the loss history:

$$\begin{aligned}
\texttt{diverged} &: \ell_T > 5 \cdot \ell_1 \ \vee \ \exists_t \, \neg\text{isfinite}(\ell_t), \\
\texttt{overfitting} &: \ell_T^{\text{train}} < 0.7 \cdot \ell_1^{\text{train}} \ \wedge \ \ell_T^{\text{val}} > 1.1 \cdot \min_t \ell_t^{\text{val}}, \\
\texttt{plateau_early} &: \text{Var}(\ell_{T-2:T}) < 10^{-3}, \\
\texttt{oscillating} &: \sigma(\ell) > 0.5 \cdot \mu(\ell),
\end{aligned}$$

where ℓ_t denotes the training loss at step t. Each pattern maps deterministically to a set of hyper-parameter fixes (e.g., diverged $\Rightarrow$ lr $\times$ 0.3, warmup $+ = 0.1$; over-fitting $\Rightarrow$ weight-decay $\times 2$, epochs -1).

When the deterministic classifier cannot confidently identify a pattern, a secondary LLM diagnosis is invoked with the same schema, and its output is used only if its reported confidence exceeds 0.6.

4 Experiments

4.1 Setup

Model and data. We employ two base models in our experiments: `Qwen3-1.7B` for text-only tasks and `Qwen2.5-VL-3B` for multi-modal reasoning. For text-only evaluation we adopt GSM8K (Cobbe et al., 2021), MathQA (Amini et al., 2019), HumanEval (Chen et al., 2021), OpenBookQA (Mihaylov et al., 2018), SciQ (Welbl et al., 2017), and BioProBench. For multi-modal evaluation we use ScienceQA (Lu et al., 2022). Training adopts LoRA (Hu et al., 2022) with r=16, α=32, dropout 0.05 applied to all attention and MLP projections, yielding 17.4M trainable parameters (1.003% of full model parameters).

Hardware. All experiments are conducted on eight NVIDIA RTX A6000 (48 GB) GPUs with bf16 mixed precision. Gradient checkpointing is selectively enabled depending on model size and input sequence length to balance memory footprint and training throughput.

Evaluation. Validation results guide experiment selection, checkpoint selection, and stopping. After the training strategy and checkpoint are frozen, we evaluate the held-out test set once for reporting. Test results are not fed back into the optimization loop. We report task performance, GPU runtime, effective optimization rounds, token consumption, and error-attribution counts. Test scores are summarized as the mean and standard deviation over three independent runs; efficiency measurements are averaged across runs.

Table 1: Reported test scores (mean ± standard deviation over three runs) on seven benchmarks. Bold marks the highest reported mean. The ablation removes attribution-driven reasoning. Text tasks use Qwen3-1.7B; ScienceQA uses Qwen2.5-VL-3B.

Benchmark	Manual SFT	Manual CL	FT-Agent	DAPF w/o attribution	DAPF
GSM8K	0.6492 ± 0.021	0.6018 ± 0.027	0.6514 ± 0.016	0.6436 ± 0.014	$\mathbf{0.6571 \pm 0.011}$
MathQA	0.6372 ± 0.018	0.6396 ± 0.020	0.6402 ± 0.015	0.6225 ± 0.013	$\mathbf{0.6468 \pm 0.010}$
HumanEval	0.4924 ± 0.025	0.4697 ± 0.029	0.5152 ± 0.018	0.5214 ± 0.016	$\mathbf{0.5303 \pm 0.013}$
OpenBookQA	0.6320 ± 0.016	0.6170 ± 0.019	0.6260 ± 0.014	0.6280 ± 0.012	$\mathbf{0.6500 \pm 0.009}$
SciQ	0.8940 ± 0.012	0.8874 ± 0.014	0.9060 ± 0.010	0.8864 ± 0.009	$\mathbf{0.9150 \pm 0.007}$
BioProBench	0.4690 ± 0.022	0.4531 ± 0.024	0.4385 ± 0.020	0.4720 ± 0.018	$\mathbf{0.4850 \pm 0.015}$
ScienceQA	0.7532 ± 0.015	0.7600 ± 0.017	0.7867 ± 0.013	0.7910 ± 0.011	$\mathbf{0.8017 \pm 0.008}$

4.2 Baselines and Attribution Ablation

We compare four training approaches and an attribution-ablated DAPF variant. The comparisons use the same model backbones, training datasets, evaluation protocols, and maximum compute budgets.

Manual SFT Baseline: Standard supervised fine-tuning with fixed training hyperparameters and a cosine learning rate schedule, following conventional static fine-tuning pipelines without adaptive data scheduling or iterative self-optimization.

Manual Curriculum SFT: A manually designed curriculum learning baseline built upon the standard SFT recipe. This group adopts unified difficulty quantification metrics and ordered data sampling strategies to organize progressive training, excluding automated agent iteration and dynamic strategy adjustment.

FT-Agent Closed-loop Training: An LLM-agent baseline that iteratively proposes training configurations, executes fine-tuning, and uses experiment results to select subsequent configurations.

DAPF Closed-loop Training: DAPF is the proposed agentic framework for closed-loop iterative post-training. It integrates a six-stage workflow that iteratively diagnoses training outcomes, attributes performance changes, and selects subsequent interventions. To evaluate the contribution of the attribution module, we introduce *DAPF w/o attribution*, a control variant that bypasses attribution-driven reasoning and directly invokes an external LLM for decision-making and action selection. All other training settings are kept consistent between the two variants.

The approaches differ in data scheduling and decision-making: the manual baselines use fixed recipes, whereas the agent-based methods adapt configurations using experiment feedback.

4.3 Main Results

Table 1 presents the main comparison.

DAPF has higher mean test scores than Manual SFT on all seven benchmarks, with absolute gains of 0.79–4.85 percentage points. Relative to FT-Agent, the average gain is 1.74 percentage points. The evaluated tasks cover mathematical reasoning, code generation, commonsense, science, biology, and multimodal question answering.

Manual CL improves on Manual SFT for MathQA and ScienceQA but has lower mean scores on the other five benchmarks. The effect of the fixed curriculum therefore varies across the evaluated tasks. DAPF also exceeds the attribution-ablated variant on all seven benchmarks, supporting the utility of diagnostic feedback in these experiments.

4.4 Token Consumption and Error Attribution Comparison

Table 2 reports average token consumption of 32K for FT-Agent and 27K for DAPF, with average error-attribution counts of 2.1 and 1.7, respectively. These measurements summarize resource consumption and diagnostic activity across runs.

Table 2: Average token consumption and error-attribution counts.

	FT-Agent	**DAPF**
Avg Token Cost	32k	27k
Error Attribution Count	2.1	1.7

DAPF uses fewer tokens on average in these runs.

4.5 Efficiency Analysis

Figure 2 summarizes computational overhead metrics across four strategies: Manual SFT, Manual CL, FT-Agent and DAPF. The plots report average GPU runtime and effective optimization rounds. Token consumption is reported separately in Table 2.

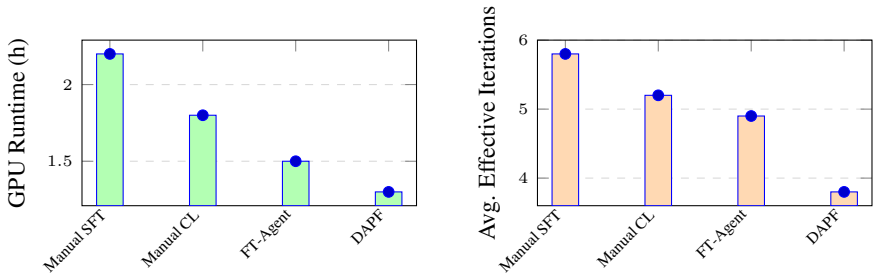

Figure 2: Computational overhead comparison across four strategies: Manual SFT, Manual CL, FT-Agent and DAPF. We report GPU runtime in hours and the average number of effective iterative rounds for each approach. All values are averaged over 3 independent repetitions.

5 Limitations

DAPF operates within a predefined search space and is evaluated on two model backbones and seven benchmarks. Its effectiveness depends on the available intervention set and the quality of training and validation feedback. Errors in agent-generated diagnoses and information loss between stages can propagate through successive optimization rounds. Broader evaluation on larger models, additional domains, and distributed training environments remains future work.

6 Discussion and Conclusion

DAPF connects experiment generation, training execution, diagnosis, and intervention selection in a closed-loop post-training workflow. Across the seven reported benchmarks, it achieves higher mean test accuracy than FT-Agent while reducing reported average GPU runtime and iteration count. The attribution ablation is consistent with diagnostic feedback helping within the tested settings. Additional component ablations, calibrated intervention estimates, and evaluation on larger models are needed to establish the contribution and generality of each mechanism.

AI Use Statement

Generative AI tools assisted with literature retrieval, English-language editing, manuscript review and restructuring, LaTeX formatting, and creation of the framework illustration. DAPF itself uses

LLM agents for requirement interpretation, hypothesis generation, experiment configuration, and critique, as described in the method and appendix. These uses are distinct from the final held-out evaluation protocol. The authors are responsible for checking AI-assisted text, references, figures, and claims against the implemented system and experimental records.

REFERENCES

A-EVO-Lab. A-Evolve-Training: Autonomous post-training of a 30b model, 2026.

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations (ICLR)*, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, and Alex Ray. Evaluating large language models trained on code, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Rei Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

Prannay Hebbar, Yogendra Manawat, Samuel Verboomen, Alesia Ivanova, Selvam Palanimalai, Kunal Bhatia, and Vignesh Baskaran. Sia: Self improving ai with harness & weight updates, 2026.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Xinya Jing, Xinya Du, Pengfei Li, Tianwei Zhang, and Yang Liu. Dsbench: How far are data science agents to becoming data science experts?, 2024.

Waleed Khalid, Dmitry Ignatov, and Radu Timofte. From memorization to creativity: Llm as a designer of novel neural architectures. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2026.

Qizheng Li, Yifei Zhang, Xiao Yang, Xu Yang, Zhuo Wang, Weiqing Liu, and Jiang Bian. FT-Dojo: Towards autonomous llm fine-tuning with language agents, 2026.

Zhe Li, Zhiwei Lin, and Yongtao Wang. Collm-nas: Collaborative large language models for efficient knowledge-guided neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2025.

Jianqiao Lu, Wanjun Zhong, Wenyong Huang, Yufei Wang, Fei Mi, Baojun Wang, Weichao Wang, Lifeng Shang, and Qun Liu. Self: Self-evolution with language feedback, 2023.

Pan Lu, Swaroop Mishra, Tony Xia, Arian Koukos, James Georgiou, and Noah A. Clark. Scienceqa: Science question answering with multi-modal contexts. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Muhammad U. Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. Llmatic: Neural architecture search via large language models and quality diversity optimization. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 2023.

Johannes Welbl, Nelson F. Liu, and Matt Gardner. Sciq: A 13.7k-question corpus for question answering and question generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

Dai Wu, Ming Li, Hao Zhao, and Yu Chen. Evolver: Self-evolving llm agents through an experience-driven lifecycle, 2025.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning, 2025.

Yang Xu, Shikai Fang, Yang Xiao, et al. R&D-Agent: An LLM-agent framework towards autonomous data science, 2025.

Taro Yano, Yoichi Ishibashi, and Masafumi Oyamada. LaMDAgent: An autonomous framework for post-training pipeline optimization via llm agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025.

Zhaojian Yu, Penghao Yin, Xiao-Ping Zhang, et al. AutoTrainees: Teaching language models to improve language models autonomously, 2026.

Hengyi Zhu, Grace Li Zhang, and Shaoyi Huang. Llm-nas: Llm-driven hardware-aware neural architecture search, 2025a.

Hengyi Zhu, Grace Li Zhang, and Shaoyi Huang. Pel-nas: Search space partitioned architecture prompt co-evolutionary llm-driven hardware-aware neural architecture search, 2025b.

Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-adapting language models (SEAL), 2025.

A Running Example

This appendix provides a real-world command-line execution log of DAPF on the SciQ dataset, demonstrating the single-round automatic fine-tuning workflow. Given target validation accuracy 0.85, the agent automatically selects hyper-parameters, executes training, evaluates the candidate on the validation set, and terminates early once the validation target is satisfied. The frozen configuration then receives exactly one held-out test evaluation for reporting only.

Listing 1: CLI running example: single-round automatic fine-tuning on SciQ with target validation accuracy 0.85

```
python3 /workspace/run_finetuning.py \
  --model /data/qwen3-1p7b \
  --dataset /data/multi_bench/sciq/train.jsonl \
  --target 0.85
============================================================
DAPF: Single-round fine-tuning
  model: /data/qwen3-1p7b
  dataset: /data/multi_bench/sciq/train.jsonl (sciq)
  target: 0.85
  max_rounds: 1
  budget: 3600s
============================================================
ROUND 1 / 1
[0/3] LLM deciding epochs...
[LLM epoch] 2 epochs (With 2000 samples and a 0.85 target validation
    accuracy, the rules specify 1-2 epochs, so 2 is chosen for better
    convergence.)
[1/3] Training (lr=2.0e-04, r=16, epochs=2)...
[train] exit=0 time=662s
[2/3] Validating...
[validation] exit=0 time=40s
  validation accuracy: 0.89
[3/3] Validation target reached; freezing configuration.
```

```
DEVELOPMENT SUMMARY (validation only)
  Round 1: lr=2.0e-04, r=16, epochs=2, train=662s, val_acc=89.00\%,
      validation=40s
```

The reported test-set result for this frozen run is generated separately after the above development log and is not supplied to the agent.

B DATASETS

Domain	Benchmark	Task
Mathematics	GSM8K	Multi-step arithmetic word problems.
Mathematics	MathQA	Mathematical question answering with annotated operations.
Code	HumanEval	Python function generation with unit tests.
Commonsense	OpenBookQA	Question answering grounded in core scientific facts.
Science	SciQ	Science questions with supporting context and distractors.
Biology	BioProBench	Biological reasoning on protein, gene, and molecular tasks.
Vision	ScienceQA	Multimodal science question answering with image context.

Benchmark	Train	Valid/Test	Training annotation
GSM8K	7473	1319	No CoT
MathQA	29837	4420	No CoT
HumanEval	–	–	No CoT
OpenBookQA	–	–	No CoT
SciQ	–	–	No CoT
BioProBench	–	–	No CoT
ScienceQA	12726	4241	Structured CoT; expert validated

Table 3: Benchmark tasks, dataset sizes, and training annotations. CoT denotes chain-of-thought annotations.

C ADJUSTER SOURCE CODE

The adjuster component consumes the critic report and applies recommended hyper-parameter adjustments, while enforcing hard safety bounds to prevent out-of-range configurations generated by LLM suggestions.

Listing 2: Adjuster: apply critic recommendations and clip hyper-parameters inside safe ranges

```
"""Apply␣critic␣recommendations␣to␣the␣next␣round's␣config."""
from __future__ import annotations
import copy
from .types import CriticReport

# Hard caps so LLM can't blow up the config
SAFE_RANGES = {
   "lr": (1e-6, 1e-2),
   "lora_r": (4, 256),
   "lora_alpha": (4, 256),
   "lora_dropout": (0.0, 0.5),
   "batch_size": (1, 16),
   "grad_accum": (1, 64),
   "epochs": (1, 10),
   "extra_steps": (0, 500),
}


def _clip(key: str, value):
   if key not in SAFE_RANGES:
      return value
```

```
    lo, hi = SAFE_RANGES[key]
    if isinstance(value, (int, float)):
        return max(lo, min(hi, value))
    return value


def adjust(base_config: dict, critic: CriticReport, last_eval: dict) ->
    dict:
    """Apply␣critic's␣recommendations␣to␣base_config.␣Returns␣new␣config␣
        for␣next␣round."""
    cfg = copy.deepcopy(base_config)
    adj = critic.recommended_adjustments or {}

    # LR multiplier (relative)
    if "lr_multiplier" in adj:
        try:
            mult = float(adj["lr_multiplier"])
            cfg["lr"] = _clip("lr", cfg.get("lr", 2e-4) * mult)
        except (TypeError, ValueError):
            pass

    # Dropout delta (additive)
    if "dropout_delta" in adj:
        try:
            d = float(adj["dropout_delta"])
            cfg["lora_dropout"] = _clip("lora_dropout",
                cfg.get("lora_dropout", 0.05) + d)
        except (TypeError, ValueError):
            pass

    # If critic says should_early_stop, mark a flag (the orchestrator
        reads it)
    if critic.should_early_stop:
        cfg["_early_stop_recommended"] = True

    # If critic says we plateaued, halve LR next round
    if not critic.should_continue and not critic.should_early_stop:
        cfg["lr"] = _clip("lr", cfg.get("lr", 2e-4) * 0.5)

    # Record the recommendation for the next round's prompt
    cfg["_last_critic"] = {
        "summary": critic.summary,
        "convergence_progress": critic.convergence_progress,
        "should_continue": critic.should_continue,
        "adjustments": adj,
    }
    return cfg
```

D CRITIC MODULE SOURCE CODE

The LLM-driven critic module analyses training trajectory metrics, produces diagnosis and hyper-parameter adjustment suggestions. It invokes an OpenAI-compatible LLM endpoint and includes a heuristic fallback path when LLM invocation fails.

Listing 3: Critic module: LLM-based trajectory analysis with heuristic fallback

```
"""LLM␣critic:␣analyze␣trajectory␣and␣recommend␣adjustments.
Calls␣<configured-api-endpoint>␣with␣claude-haiku␣(configurable).
Falls␣back␣to␣heuristic-only␣analysis␣if␣LLM␣fails.
"""
from __future__ import annotations
import json
import urllib.error
```

```
import urllib.request
import re
from .types import Trajectory, CriticReport, Round

SYSTEM_PROMPT = (
    "You are an expert ML training critic. Given a training trajectory"
    "and goal, output STRICT JSON describing diagnosis and recommended
        adjustments.\n"
    "Rules:\n"
    "Output ONLY valid JSON. No prose.\n"
    "Be specific about why the model is/isn't converging.\n"
    "recommended_adjustments values MUST be in safe ranges:\n"
    "lr_multiplier: 0.1 to 5.0\n"
    "dropout_delta: -0.1 to 0.2\n"
    "extra_steps: 0 to 200\n"
    "freeze_embedding: true or false\n"
    "should_continue=true if there's room to improve; false if plateau/
        diverged.\n"
    "Output schema:\n"
    "{\n"
    '"summary": "one sentence about current state",\n'
    '"convergence_progress":0.0-1.0,\n'
    '"bottleneck_hypothesis":"why loss isn\'t decreasing",\n'
    '"recommended_adjustments":{\n'
    '"lr_multiplier": 1.0,\n'
    '"dropout_delta": 0.0,\n'
    '"extra_steps": 0,\n'
    '"freeze_embedding":false\n'
    "},\n"
    '"should_continue":true,\n'
    '"should_early_stop": false,\n'
    '"reasoning":"..."\n'
    "}\n"
)


def call_llm(system, user, max_tokens=1500, max_retries=3):
    """Direct OpenAI-compatible call. Returns parsed JSON or raises."""
    payload = {
        "model": MODEL,
        "messages": [
            {"role": "system", "content": system},
            {"role": "user", "content": user},
        ],
        "max_tokens": max_tokens,
        "temperature": 0.2,
    }
    req = urllib.request.Request(
        f"{API_BASE}/chat/completions",
        data=json.dumps(payload).encode(),
        headers={
            "Content-Type": "application/json",
            "Authorization": f"Bearer {API_KEY}",
        },
    )
    last_err = None
    for attempt in range(max_retries):
        try:
            with urllib.request.urlopen(req, timeout=60) as resp:
                data = json.loads(resp.read())
                msg = data["choices"][0]["message"]
                content = msg.get("content") or msg.get("reasoning_content")
                    or ""
                return _extract_json(content)
        except urllib.error.HTTPError as e:
```

```
            last_err = e
            if e.code == 429:
                import time
                time.sleep(5 * (2 ** attempt))
            else:
                raise
    raise RuntimeError(f"LLM failed after {max_retries} retries: {last_err}")


def _extract_json(text: str) -> dict:
    """Robust JSON extraction - shared JSON extraction utility."""
    t = text.strip()
    if t.startswith("```"):
        lines = t.splitlines()
        if lines[0].startswith("```"):
            lines = lines[1:]
        if lines and lines[-1].strip() == "```":
            lines = lines[:-1]
        t = "\n".join(lines)
    try:
        return json.loads(t)
    except Exception:
        pass

    candidates = []
    depth = 0
    start = None
    for i, ch in enumerate(t):
        if ch == "{":
            if depth == 0:
                start = i
            depth += 1
        elif ch == "}":
            depth -= 1
            if depth == 0 and start is not None:
                candidates.append(t[start:i + 1])
                start = None
    for cand in sorted(candidates, key=len, reverse=True):
        try:
            return json.loads(cand)
        except Exception:
            fixed = cand
            if fixed.count('"') % 2 == 1:
                fixed += '"'
            try:
                return json.loads(fixed)
            except Exception:
                continue
    raise ValueError(f"Could not extract JSON (len={len(t)})")


def fallback_critic(round, heuristic_signals) -> CriticReport:
    """No-LLM fallback: derive report from heuristic signals."""
    rep = CriticReport()
    fs = heuristic_signals.any_stop_signal()
    if fs:
        rep.should_continue = False
        rep.should_early_stop = True
        rep.summary = "heuristic stop signal: " + (
            "diverging" if heuristic_signals.diverging else
            "nan_detected" if heuristic_signals.nan_detected else
            "overfitting" if heuristic_signals.overfitting else
            "gradient_exploding" if heuristic_signals.grad_exploding else
            "stop signal"
```

```
        )
        rep.bottleneck_hypothesis = "see␣heuristic␣signals"
        rep.recommended_adjustments = {
            "lr_multiplier": 0.5,
            "dropout_delta": 0.0,
            "extra_steps": 0,
            "freeze_embedding": False
        }
    else:
        loss_trend = round.trajectory.trend("train_loss")
        rep.summary = f"fallback:␣train_loss␣trend={loss_trend},␣
            last_val_acc={round.trajectory.last('val_acc'):.3f}"
        if loss_trend == "decreasing":
            rep.convergence_progress = 0.7
            rep.should_continue = True
            rep.recommended_adjustments = {
                "lr_multiplier": 1.0,
                "extra_steps": 50,
                "dropout_delta": 0.0,
                "freeze_embedding": False
            }
        else:
            rep.convergence_progress = 0.3
            rep.should_continue = True
            rep.recommended_adjustments = {
                "lr_multiplier": 0.5,
                "extra_steps": 30,
                "dropout_delta": 0.0,
                "freeze_embedding": False
            }
    rep.raw_response = "fallback␣(no␣LLM)"
    return rep


def analyze(round, heuristic_signals, task: dict) -> CriticReport:
    """Main␣entry:␣try␣LLM,␣fall␣back␣to␣heuristic."""
    summary = round.trajectory.summary()
    user_p = (
        f"Task␣goal:␣{task.get('goal',␣'unknown')}\n"
        f"Target␣metric:␣{task.get('target_value',␣'?')}\n"
        f"Round:␣{round.round_id}\n"
        f"Trajectory␣summary:␣{json.dumps(summary)}\n"
        f"Last␣10␣train_loss:␣{round.trajectory.train_loss[-10:]}\n"
        f"Last␣10␣val_acc:␣{round.trajectory.val_acc[-10:]}\n"
        f"Last␣10␣lr:␣{round.trajectory.learning_rate[-10:]}\n"
        f"Last␣10␣grad_norm:␣{round.trajectory.grad_norm[-10:]}\n"
        f"Heuristic␣signals␣(any␣True␣means␣early␣stop␣risk):␣"
        f"nan={heuristic_signals.nan_detected},␣"
        f"diverging={heuristic_signals.diverging},␣"
        f"overfitting={heuristic_signals.overfitting},␣"
        f"plateau={heuristic_signals.val_acc_plateau},␣"
        f"grad_exploding={heuristic_signals.grad_exploding},␣"
        f"grad_vanishing={heuristic_signals.grad_vanishing},␣"
        f"lr_floor={heuristic_signals.lr_floor_reached}\n"
    )
    try:
        data = call_llm(SYSTEM_PROMPT, user_p)
        rep = CriticReport(
            summary=data.get("summary", ""),
            convergence_progress=float(data.get("convergence_progress", 0.0)
                ),
            bottleneck_hypothesis=data.get("bottleneck_hypothesis", ""),
            recommended_adjustments=data.get("recommended_adjustments", {}),
            should_continue=bool(data.get("should_continue", True)),
            should_early_stop=bool(data.get("should_early_stop", False)),
```

```
            raw_response=json.dumps(data),
        )
        return rep
    except Exception as e:
        return fallback_critic(round, heuristic_signals)
```

E PROMPT TEMPLATES AND INVOCATION PARAMETERS

DAPF delegates requirement interpretation, hypothesis generation, experiment configuration, and result review to separate roles. Each role returns structured JSON only, enabling validation, retries, and auditing. The templates below are reproducible examples of the current implementation.

E.1 RESEARCHER: FALSIFIABLE HYPOTHESIS CARD

```
You are the DAPF Researcher. Given requirements and previous rounds,
output ONE falsifiable, single-variable hypothesis card as a JSON object
    only.

Required fields:
  hypothesis_id, round_id, parent_goal, statement, category,
  change, prediction, falsification, evidence, priority,
  selection_reason, risk_flags, status.

Rules:
  1. Change exactly one main axis (for example learning rate or dropout).
  2. Prediction must name a metric, direction, and magnitude (e.g. >=3%).
  3. Falsification must be a numeric threshold.
  4. Return valid JSON only; do not add markdown or commentary.
```

E.2 EXPERIMENTER: EXECUTABLE EXPERIMENT CONFIGURATION

```
You are the DAPF Experimenter. Convert the selected hypothesis into one
executable experiment.json. The runner is:
python3 -m autorun.training.lm_finetune \
  --experiment <experiment.json> --requirements <requirements.json>

The JSON must contain:
  baseline_checkpoint, data (absolute train/eval paths),
  training (max_seq_len, epochs, per_device_batch_size,
  gradient_accumulation_steps, learning_rate, scheduler,
  lora_r, lora_alpha, lora_dropout, target_modules,
  precision, seed, gradient_checkpointing, shuffle),
  resources, evaluation, artifacts, design_notes, status.

Rules:
  1. All paths are absolute.
  2. Keep the baseline checkpoint fixed unless the requirement says
      otherwise.
  3. Preserve the single-variable change in the hypothesis.
  4. Return JSON only.
```

E.3 CRITIC: RESULT REVIEW

```
You are the DAPF Critic. Review the observed experiment without modifying
     it.
Return review.json with:
  hypothesis_id, experiment_id, verdict,
  execution_quality, limitations, assessment, recommendation,
  evidence (path and claim), status.
```

```
verdict is one of: confirmed, weakened, refuted, inconclusive.
Rules:
  1. Failed or timed-out execution is inconclusive.
  2. A single seed cannot establish statistical significance.
  3. Every claim must cite an artifact path.
  4. Return JSON only.
```

E.4 Dynamic Inputs, Sampling, and Retries

Role	Dynamic inputs	Temp.	Tokens	Failure handling
Researcher	requirements (truncated to 3000 characters), round_id, and history (truncated to 2000 characters)	0.4	2000	JSON extraction and retry
Experimenter	requirements, hypothesis, dataset, and absolute paths	0.2	2500	Fill default fields, validate, and retry
Critic	hypothesis, experiment (truncated to 3000 characters), and result	0.1	1800	JSON extraction and retry

The LLM call is retried at most eight times. The first valid JSON object is extracted from each response and then checked for required fields. Responses that cannot be parsed or that omit required fields do not proceed to training.

F State Machine and Decision Rules

A complete iteration follows the sequence `INIT` → `HYPOTHESIZE` → `SPECIFY` → `RUNNING` → `REVIEW` → `DECIDE` → `PERSIST`. Training failures transition from `RUNNING` to `RECOVER`; after recovery, the constrained configuration is executed again.

```
if target_is_met(metrics):
   stop("goal_met")
elif has_nan_or_gradient_explosion(metrics) or critic_requests_early_stop
    :
   stop("fatal_or_early_stop")
elif plateau_for_three_rounds(history):
   stop("plateau")
elif critic_verdict == "refuted" and not used_lr_retry:
   continue_with(learning_rate = learning_rate / 2)
elif critic_verdict == "refuted" and rounds_with_critic_stop >= 2:
   stop("critic_stop")
elif budget_exhausted(budget):
   stop("budget_exhausted")
elif round_id >= max_rounds:
   stop("max_rounds")
else:
   continue_with(constrained_intervention())
```

The controller uses a bounded set of interventions. Its main automatic parameter adjustments are learning-rate scaling and incremental LoRA dropout changes. It also propagates early-stop decisions and checkpoint state between rounds.

G JSON Records and Artifact Layout

The following directory layout associates each optimization round with its configurations, metrics, and artifacts:

```
workspace/
  round_XX/
```

```
requirements.json
hypothesis.json
experiment/experiment.json
execution/
  stdout.log
  stderr.log
  result.json
  metrics.json
  training_summary.json
  review.json
  decision.json
checkpoints/{best.pt,last.pt}
```

Record	Question it must answer
requirements.json	What are the target metric, direction, threshold, data, and resource budget?
hypothesis.json	Which single axis changed, and what are the prediction and numeric falsification condition?
experiment.json	How is the run executed, with which model, data, hyperparameters, and random seed?
metrics.json	What are the training/validation metrics, and was the final target reached?
review.json	Does the evidence support confirmed, weakened, refuted, or inconclusive?
decision.json	What is the next action or the reason for stopping?

H Training Protocol

The training script uses supervised fine-tuning (SFT) with LoRA. Labels for the input-prompt portion are set to -100, so cross-entropy is computed only for the answer portion; sequences are truncated to `max_seq_len`. The default configuration uses bf16, AdamW, a fixed random seed of 42, gradient accumulation, and gradient checkpointing, and saves `best.pt`, `last.pt`, `metrics.json`, and `training_summary.json`.

Item	Recorded content (use the final experiment.json as the source of truth)
Base model	Qwen3-1.7B (fixed absolute checkpoint path)
Trainable parameters	LoRA r, α, dropout, and target modules
Optimizer	AdamW; record learning rate and scheduler
Numerics and stability	bf16, seed=42, and gradient checkpointing
Data loading	Absolute train/eval paths, batch size, gradient accumulation, and shuffle
Artifacts	Best/last checkpoints, metric JSON, training summary, and runtime information

I Evaluation Protocol and Known Limitations

Generation uses greedy decoding (`do_sample=False`, `max_new_tokens=128`, maximum input length 1024). Multiple-choice answers are parsed from the first A–D token; yes/no questions are parsed as yes/no; ordering questions are parsed from "Correct order:" or the final line; GSM8K compares the final numeric value using `math.isclose`; generation tasks use LCS-based ROUGE-L F1 with a threshold of 0.3. Each sample records the question ID, question type, ground truth, prediction, tail of the raw output, and correctness.

The general evaluation routine uses a permissive ROUGE-L threshold for generation tasks. Its set-based comparison for ordering questions measures element agreement rather than sequence correctness.

J Fault Tolerance, Recovery, and Reproducibility

Training subprocesses have a 3600-second timeout; exit code, OOM, stalled logs, and missing metric files are recorded as faults. Failed or timed-out experiments are marked `inconclusive` and are

not treated as evidence that the target was achieved. Metrics and summaries are written atomically, while failure logs and budget consumption are retained; configuration errors and missing data are treated as unrecoverable and stop the run to avoid unbounded retries.

The reproducibility checklist is:

1. Record the complete git commit, Python/PyTorch/Transformers versions, PEFT version, GPU model, and CUDA version.
2. Save all six JSON records: requirements, hypothesis, experiment, metrics, review, and decision.
3. Fix the random seed (42 by default) and record the dataset version, absolute paths, and preprocessing command.
4. Record per-round duration, memory/OOM events, checkpoint paths, and the final evaluation command.
5. For externally downloaded HumanEval data, record the source URL, file hash, and local-copy information.

K Supplementary Results and Reproducibility Notes

All reported results, including the Manual Curriculum SFT (Manual CL) baseline, are aggregated over three independent runs using three pre-specified random seeds. We report the mean test accuracy and standard deviation across the three runs. For all methods, the model backbone, dataset splits, evaluation protocol, and maximum computational budget are held fixed, while the intended methodological differences are preserved. In particular, Manual CL uses curriculum-based data ordering, whereas DAPF variants differ in their decision mechanisms. Per-seed configurations, metrics, and checkpoints are retained for reproducibility.