EvolveLRM: Closed-Loop Logical-Reasoning Post-Training via Cross-Task Validation Gain

Anonymous ACL submission

Abstract

Logical-reasoning post-training is increasingly a control problem: which update improves reasoning beyond its source format? Current pipelines separately choose evaluation slices, synthetic data, source tasks, and SFT or verifiable RL recipes, often optimizing local accuracy while hiding negative transfer. We propose **EvolveLRM**, a planner-controlled closed-loop framework that turns *evaluation* → *data generation/filtering* → *training* → *re-evaluation* into an evidence-gated cycle. The gate is *cross-task validation gain*: candidate data sources and training recipes are retained when they improve held-out or out-of-distribution reasoning slices after accounting for compute cost, instability, and finite-probe uncertainty. An Experience Bank stores execution traces, imported skills, and system knowledge as action priors for memory-guided planning. On Qwen2.5-7B with the same CLUB source, static SFT maximizes source-format accuracy (19.0) but lowers General Avg from 32.41 to 22.33; verifier-guided GRPO reaches comparable source accuracy (18.8) while raising General Avg to 47.38, and ExCLUB-scale GRPO further raises it to 52.59. System studies over Qwen2.5 0.5B–7B and planner contexts show that strategy memory plus experience retrieval helps the loop reach training dispatch. EvolveLRM reframes reasoning self-evolution as evidence-gated improvement rather than open-loop source maximization.

1 Introduction

Logical reasoning is central for language models that must go beyond factual recall. Prompting and decoding methods such as chain-of-thought reasoning, self-consistency, and tree-structured search show that explicit intermediate computation improves multi-step problem solving (Wei et al., 2022; Wang et al., 2022; Yao et al., 2023). Reasoning-oriented post-training and verifiable reinforcement learning further show that models can acquire stronger reasoning behavior from checkable outcomes (Guo et al., 2025; Li et al., 2025; Liu et al., 2025a). Yet gains on one benchmark or data distribution often fail to transfer to new task families, harder difficulty levels, or different reasoning formats, and apparent reasoning gains can reflect memorization or brittle chain-of-thought behavior (Zhao et al., 2025; Xie et al., 2025).

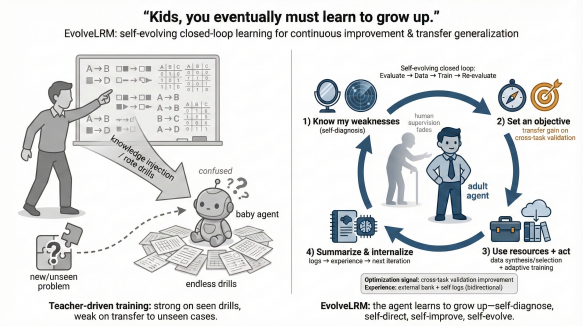

Figure 1: Motivation. Source-only tuning can improve one reasoning format while reducing transfer. EvolveLRM instead closes the loop: diagnose weak slices, construct solver-checked data, update the model, and keep the action only when re-evaluation shows cross-task gain.

The practical pipeline for improving reasoning models remains fragmented. Researchers choose a benchmark slice, design or adopt a data generator, select a post-training recipe, and evaluate the resulting checkpoint. This workflow has produced strong components, but it is often blind to the central control question: *which next training action is likely to improve reasoning beyond the source task?* A source-only objective can reward a checkpoint that solves one puzzle format while losing mathematical, symbolic, or instruction-following robustness. Benchmarks such as BBH, LogicBench, ZebraLogic, LogicGame, SynLogic, Enigmata, and REASONING GYM have made evaluation broader and more rule-checkable (Suzgun et al., 2023; Parmar et al., 2024; Lin et al., 2025; Gui

et al., 2025; Liu et al., 2025b; Chen et al., 2025; Stojanovski et al., 2025); LogicEvolve provides CLUB/ExCLUB, deterministic verifier-backed artifacts with task metadata, solver/evaluator checks, and evolved pools (Du et al., 2026). We use them as representative adapters, while EvolveLRM only requires deterministic scores, provenance, and reward hooks. The missing layer is not another benchmark, but an explicit gate for negative transfer.

We propose **EvolveLRM**, a closed-loop learning framework for logical reasoning models. As shown in Figure 2, EvolveLRM refactors the manual pipeline into an executable cycle of evaluation, diagnosis, data generation/filtering, training, serving, and re-evaluation. A top-level **AutoPlanner** coordinates **AutoEvaluator**, **AutoDataMaker**, **AutoTrainer**, and **AutoServer**; it observes the validation profile, retrieves memory, selects the next actions, validates their schemas, and updates its state after re-evaluation.

The central signal in EvolveLRM is **cross-task validation gain**. Instead of prioritizing data because it improves the same task from which it was drawn, EvolveLRM favors candidate data and recipes that improve held-out or out-of-distribution validation slices. This distinction matters for logical reasoning: a model can become better at a source format while losing robustness on other symbolic or mathematical tasks. We therefore formalize evolution as budgeted action selection under cross-task utility, derive why gain-based selection targets transferable updates, and implement the loop with persistent memory and an Experience Bank that stores both literature-derived strategies and self-iteration traces.

Concretely, in our Qwen2.5-7B study, SFT-Full attains the best in-domain CLUB Acc (19.0) yet loses 10.08 General-Avg points. GRPO on the same source reaches comparable CLUB Acc (18.8) while improving Logic Avg from 8.63 to 19.90 and General Avg from 32.41 to 47.38. The decisive factor is therefore not only the data source, but the update operator's pressure—exactly what an evidence-driven planner should discover.

Our contributions are threefold. We introduce a planner-driven closed-loop framework that replaces disconnected tuning with executable evaluation, data construction, post-training, serving, and re-evaluation roles; formulate cross-task validation gain and Training Gain as operational objectives with finite-probe bounds; and instantiate the framework on Qwen2.5 models with CLUB/ExCLUB adapters, where static SFT gains +18.2 CLUB points but loses 10.08 General-Avg points while verifier-guided GRPO yields positive transfer.

2 Related Work

Logical reasoning benchmarks and diagnosis. Logical-reasoning evaluation has moved from fixed single-task datasets to broader suites that stress compositional rules, multi-step search, memorization risk, and distribution shift. BBH collects difficult BIG-Bench tasks and shows that chain-of-thought helps unevenly across task families (Suzgun et al., 2023). LogicBench and ZebraLogic expose symbolic, constraint-based, and multi-hop failures (Parmar et al., 2024; Lin et al., 2025), while LogicGame, SynLogic, Enigmata, and REASONING GYM add scalable or verifier-backed reasoning data (Gui et al., 2025; Liu et al., 2025b; Chen et al., 2025; Stojanovski et al., 2025; Xie et al., 2025). LogicEvolve is relevant here because it uses a multi-agent loop over task metadata, generation, solving, and evaluation to build the deterministic symbolic reasoning artifacts CLUB and ExCLUB (Du et al., 2026). EvolveLRM operates one level higher: it treats such artifacts as replaceable sensors and training sources inside a closed learning loop, asking which data and recipes improve cross-task validation utility.

Reasoning-time, curriculum, and training-time improvement. Reasoning-time methods improve task solving without changing model parameters. Chain-of-thought prompting elicits intermediate steps (Wei et al., 2022), self-consistency marginalizes over sampled paths (Wang et al., 2022), Tree of Thoughts searches over intermediate states (Yao et al., 2023), and Self-Refine revises outputs using model feedback (Madaan et al., 2023). Curriculum and self-paced learning order examples by difficulty or model readiness (Bengio et al., 2009; Kumar et al., 2010); EvolveLRM instead scores candidate slices by transfer gain, cost, and failure risk, so a hard slice can be selected early if it improves held-out validation. Training-time approaches include rationale bootstrapping, self-training, preference optimization, and verifier-guided RL: STaR bootstraps rationales (Zelikman et al., 2022); Self-Improve and Quiet-STaR learn from self-generated reasoning (Huang et al., 2023; Zelikman et al., 2024); ReST, DPO, self-rewarding models, SPIN, and DeepSeek-R1 demonstrate generated data, preference/reward optimization, self-

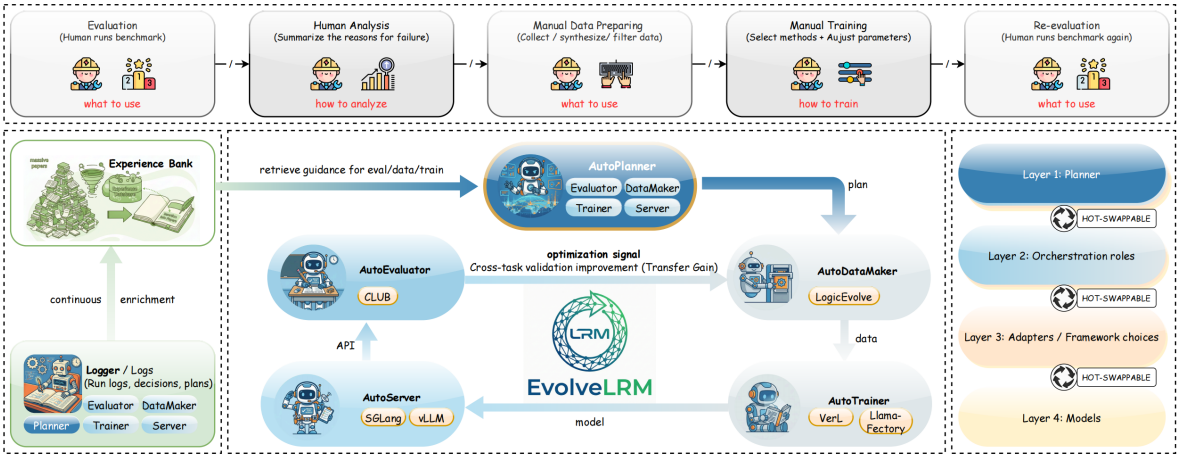

Figure 2: EvolveLRM as a planner-controlled loop. Evaluation builds a cross-task profile; the planner retrieves memory and selects executable data/training actions; modular roles run the action; re-evaluation decides whether the update improved transferable reasoning rather than only the source format.

produced rewards, self-play, and outcome-checked RL (Gulcehre et al., 2023; Rafailov et al., 2023; Yuan et al., 2024; Chen et al., 2024; Guo et al., 2025). EvolveLRM treats SFT, preference-style methods, and RL as replaceable actions, but the completed runs instantiate SFT and GRPO.

Self-improvement and self-evolving agents. Self-improvement research studies how models acquire experience, refine it, and re-evaluate themselves over time (Tao et al., 2024; Gao et al., 2025). Self-Instruct bootstraps instruction data from a seed set (Wang et al., 2023), while Constitutional AI reduces human preference labeling with principle-guided AI feedback (Bai et al., 2022). Recent systems such as R-Zero, AlphaEvolve, Alita, and Agent-World extend self-evolution to reasoning, algorithm discovery, agentic reasoning, and environment synthesis (Huang et al., 2025; Novikov et al., 2025; Qiu et al., 2025; Dong et al., 2026). R-Zero learns reasoning from self-play, AlphaEvolve evolves algorithms through generated code, and Alita scales agentic reasoning with minimal predefinition. EvolveLRM differs in target and signal: it post-trains a single reasoning policy on solver-verified symbolic data and uses cross-task validation gain as the planner's selection criterion, not self-play reward or code execution.

Relative to update operators such as STaR, ReST, DPO, self-rewarding, and verifier-guided RL, and relative to self-evolving meta-loops for code, reasoning, or agents, EvolveLRM contributes the selection layer. The same planner can compose these methods as adapters, but the action must pass candidate validation and improve a cross-task objective rather than only source accuracy or generic preference.

3 EvolveLRM

EvolveLRM is a planner-controlled closed loop for logical-reasoning post-training. Its premise is that a model should not be updated merely because a source task is weak; it should be updated when a candidate data source and training recipe improve a diagnosed validation objective covering held-out tasks, harder difficulty levels, and out-of-distribution (OOD) reasoning formats.

3.1 Architecture

The system contains a top-level **AutoPlanner** and four replaceable execution roles. **AutoEvaluator** runs rule-checkable benchmarks and returns a multi-task performance profile. **AutoDataMaker** constructs or selects data for planner-chosen failure slices through data-source adapters, including LogicEvolve-compatible CLUB/ExCLUB generators. **AutoTrainer** executes the selected recipe, currently supporting SFT and GRPO-style verifier-guided RL through adapters. **AutoServer** deploys the current checkpoint so evaluation and data construction share a consistent endpoint. Structured inputs and logs make the loop auditable and let adapters change without changing the planner objective.

The planner is a proposal generator, not a free-form controller. At each round, it receives the model state, evaluation profile, available benchmarks, data sources, recipes, and retrieved memory; selects evaluation, data, and training actions

Algorithm 1 EvolveLRM round k

Require: model θ_k, state s_k, memory $\mathcal{M}_k$, candidates $\mathcal{F}_k$

1: AUTOSERVER deploys θ_k; PLANNER selects (B_k^e, T_k^e, n_k^e).
2: AUTOEVALUATOR measures $\widehat{U}_{k,i}$ and diagnoses q_k by Eq. 6.
3: PLANNER chooses (B_k^d, T_k^d, n_k^d) and recipe r_a.
4: AUTODATAMAKER builds $\mathcal{D}_a^{(m)}$; AUTOTRAINER updates $\theta_{k,a}^{(m)} = \mathcal{U}_{r_a}(\theta_k, \mathcal{D}_a^{(m)})$.
5: Re-evaluate $\theta_{k,a}^{(m)}$ and estimate $\widehat{G}_k(a;m) = J_k(\theta_{k,a}^{(m)}) - J_k(\theta_k)$.
6: **Gate:** continue iff gain exceeds cost, risk, and probe uncertainty.
7: Otherwise shrink the update, change recipe, expand candidates, or roll back.
8: Update memory with profile, provenance, outcome, and failures.

from adapter-provided candidate sets; and writes re-evaluation evidence back to memory. EvolveLRM thus turns the human workflow—diagnose, curate data, train, and re-test—into a repeatable control process. Algorithm 1 summarizes one round.

Role interface. The roles share a minimal contract. A state s_k contains the model identifier, serving endpoint, previous profile, available task generators, recipes, and memory summaries. A planner action a_k specifies an evaluation request e_k, a data request d_k, and a training request t_k. The executor returns

$$s_{k+1} = \Phi(s_k, e_k, d_k, t_k; \xi_k), \tag{1}$$

where ξ_k denotes stochasticity from model sampling, generation, and training. EvolveLRM assumes only that each role returns structured evidence: verifier scores, provenance, task metadata, and cost/stability signals.

Constrained planner policy. Let $h_k = (s_k, \mathcal{M}_k, \mathcal{H}_k)$ denote the context, where $\mathcal{M}_k$ contains retrieved strategy and experience entries and $\mathcal{H}_k$ contains prior records. An executable action is

$$a_k = (B_k^e, T_k^e, n_k^e, B_k^d, T_k^d, n_k^d, r_k), \tag{2}$$

where the three blocks select evaluation slices, data slices, and the training recipe. Each component is checked for membership, non-empty task lists, and positive sizes. Invalid responses trigger retries; if none succeeds, the round fails closed rather than launching an ill-specified job. This induces a planner distribution over the feasible set $\mathcal{F}_k$,

$$\Pi_k(a \mid h_k) \propto \mathbf{1}\{a \in \mathcal{F}_k\}\, p_\psi(a \mid h_k), \tag{3}$$

where p_ψ is the LLM planner's proposal distribution after memory injection.

3.2 Cross-Task Objective

Let $\mathcal{T} = \{T_1, \ldots, T_N\}$ be the task space. Each task T_i defines a distribution $P_i(x, y)$ and a deterministic verifier $v_i(\hat{y}, y) \in [0, 1]$. For model parameters θ, task utility is

$$U_i(\theta) = \mathbb{E}_{(x,y)\sim P_i}\left[v_i(f_\theta(x), y)\right]. \tag{4}$$

In round k, the planner constructs a validation slice $\mathcal{C}_k$ and a diagnosed weight vector q_k over weak, held-out, unstable, or higher-difficulty tasks:

$$\begin{aligned} J_k(\theta) &= \sum_{i\in\mathcal{C}_k} q_{k,i} U_i(\theta), \\ q_{k,i} \geq 0, &\qquad \sum_{i\in\mathcal{C}_k} q_{k,i} = 1. \end{aligned} \tag{5}$$

This objective is the quantity the loop tries to improve. A task whose own accuracy is low receives priority only when training on it is expected to improve J_k, not merely its own score.

The diagnosed weights can be instantiated from the current evaluation profile. Let $\widehat{U}_{k,i}$ be the measured utility on task i, let d_i encode difficulty or OOD priority, and let $\rho_{k,i}$ mark slices with unstable or regressing behavior across rounds. A soft weighting rule is

$$\begin{aligned} q_{k,i} &= \frac{\exp z_{k,i}}{\sum_{j\in\mathcal{C}_k} \exp z_{k,j}}, \\ z_{k,i} &= \alpha(1 - \widehat{U}_{k,i}) + \beta_d d_i + \beta_r \rho_{k,i}. \end{aligned} \tag{6}$$

Here q_k depends only on the current measured profile $\{\widehat{U}_{k,i}(\theta_k)\}$, not on a candidate θ; $J_k(\cdot)$ is therefore fixed while comparing actions in round k.

A candidate evolution action is $a = (\mathcal{S}_a, \pi_a, b_a, r_a) \in \mathcal{A}_k$, where $\mathcal{S}_a$ is a source slice, π_a is a generation or filtering policy, b_a is the budget, and r_a is the recipe. A short probe or historical estimate gives an updated model $\theta_{k,a}^{(m)}$ under $m \leq b_a$ samples, and the true cross-task gain is

$$G_k(a; m) = J_k(\theta_{k,a}^{(m)}) - J_k(\theta_k). \tag{7}$$

The operational score further accounts for compute and instability. Let $C_{a,m}$ aggregate data volume, rollout count, sequence budget, and launched jobs, and let $R_{a,m}$ increase with invalid planner JSON, unavailable adapters, empty files, reward

mismatch, non-zero exits, out-of-memory events, or re-evaluation drops. The score is

$$\widehat{S}_k(a;m) = \widehat{G}_k(a;m) - \lambda C_{a,m} - \beta R_{a,m}. \quad (8)$$

The planner can either select the best-scoring action or allocate budget softly,

$$\begin{aligned} w_{k,a} &\propto p_{k,a} \exp(\widehat{S}_k(a;m)/\tau), \\ \sum_{a \in \mathcal{A}_k} w_{k,a} &= 1, \end{aligned} \quad (9)$$

where $p_{k,a}$ is a prior induced by strategy memory and the Experience Bank.

3.3 Training Gain

Training Gain measures whether a source slice is sample-efficient for transfer. Given a desired improvement threshold $\gamma > 0$,

$$\begin{aligned} n_\gamma(a) &= \min\{m : G_k(a;m) \geq \gamma\}, \\ \mathrm{TG}_\gamma(a) &= \frac{1}{n_\gamma(a) + \epsilon_0}, \end{aligned} \quad (10)$$

with $\mathrm{TG}_\gamma(a) = 0$ if the threshold is never reached. High Training Gain means that a source data type reaches positive cross-task improvement quickly and cheaply; this is different from source-task accuracy, because a source can be easy to fit yet harmful for J_k.

For finite budgets, the planner also needs a marginal version. For $0 < m_1 < m_2 \leq B$, define

$$\begin{aligned} \Delta\mathrm{TG}_k &= \frac{\Delta G_k}{\Delta C(a; m_1, m_2) + \epsilon_0}, \\ \Delta G_k &= G_k(a; m_2) - G_k(a; m_1), \\ \Delta C &= \mathrm{Cost}(a, m_2) - \mathrm{Cost}(a, m_1). \end{aligned} \quad (11)$$

A positive marginal gain supports increasing the same data family or recipe; a negative marginal gain is evidence of saturation or over-specialization.

3.4 Training Recipes as Update Operators

Each action a induces an update operator $\mathcal{U}_{r_a}$, so that $\theta_{k,a}^{(m)} = \mathcal{U}_{r_a}(\theta_k, \mathcal{D}_a^{(m)})$. EvolveLRM currently exposes SFT-Full, low-rank adaptation (LoRA) SFT, and Group Relative Policy Optimization (GRPO) through trainer adapters. For SFT, the source data $\mathcal{D}_a = \{(x, y)\}$ is converted into instruction-format examples and optimized by

$$\mathcal{L}_{\mathrm{sft}}(\theta; \mathcal{D}_a) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_a} \log \pi_\theta(y \mid x). \quad (12)$$

Full SFT updates all parameters, while LoRA restricts the update to a low-rank adapter subspace. In both cases, the loss imitates the source answer distribution, so its effect on J_k depends on whether that source is a good proxy for the validation mixture.

For verifiable RL, each prompt x yields a group of sampled responses $\{y_j\}_{j=1}^G$. A deterministic task verifier produces rewards $R_j = v(y_j, y^\star)$ and group-relative advantages

$$A_j = \frac{R_j - \frac{1}{G}\sum_{\ell=1}^G R_\ell}{\mathrm{std}(\{R_\ell\}_{\ell=1}^G) + \epsilon_c}. \quad (13)$$

The GRPO-style objective maximizes a clipped policy update with a Kullback–Leibler (KL) stabilizer,

$$\begin{aligned} \mathcal{J}_{\mathrm{grpo}}(\theta) = \mathbb{E}_{x,\{y_j\}} \frac{1}{G} \sum_{j=1}^G \mathrm{clip}(A_j, r_j(\theta); \epsilon_c) \\ - \lambda_{\mathrm{KL}} D_{\mathrm{KL}}(\pi_\theta(\cdot \mid x) \,\|\, \pi_{\mathrm{ref}}(\cdot \mid x)), \end{aligned} \quad (14)$$

where $r_j(\theta) = \pi_\theta(y_j \mid x)/\pi_{\theta_k}(y_j \mid x)$ and the full clipped surrogate is given in Appendix A. The key distinction is that GRPO does not merely clone a generated solution; it uses verifier feedback to separate correct and incorrect sampled reasoning attempts under the same prompt.

3.5 Why GRPO Transfers and SFT Overfits

Let $\mathcal{L}_V(\theta) = -J_k(\theta)$ be the validation loss, let $g_V = \nabla_\theta \mathcal{L}_V(\theta_k)$, and let $g_a = \nabla_\theta \mathcal{L}_a(\theta_k)$ be the training gradient induced by action a. For one step $\theta' = \theta_k - \eta g_a$, define $\Delta\mathcal{L}_V = \mathcal{L}_V(\theta') - \mathcal{L}_V(\theta_k)$ and $\Delta J_k = J_k(\theta') - J_k(\theta_k)$. If $\mathcal{L}_V$ is H-smooth,

$$\begin{aligned} \Delta\mathcal{L}_V &\leq -\eta\langle g_V, g_a\rangle + \tfrac{H\eta^2}{2}\|g_a\|^2, \\ \Delta J_k &\geq \eta\langle g_V, g_a\rangle - \tfrac{H\eta^2}{2}\|g_a\|^2. \end{aligned} \quad (15)$$

Thus an action improves the cross-task objective when its training gradient is aligned with the validation-loss gradient and the update is not too large.

Two mechanisms make GRPO more likely than static SFT to align with g_V. First, verifier reward is a task-level scalar: it updates the policy on the correct/incorrect axis rather than forcing the model toward one source-token distribution. Second, within-group normalization focuses learning on discriminable prompts. When every sampled completion is wrong, the group provides little signal; when every completion is correct, it provides little exploration pressure. The useful region is the mixed

group, where the model has found both failed and successful reasoning paths for the same prompt.

This gives Eq. 15 a concrete interpretation. SFT concentrates probability mass on the observed source-format answer, so the gradient can mostly encode how to reproduce that format. If the validation mixture uses different schemas, search depths, or mathematical constraints, that gradient can be small or nearly orthogonal to directions that reduce $\mathcal{L}_V$. GRPO keeps more exploration alive because rewards compare sampled paths within a prompt and the KL term limits collapse toward one template. The negative-transfer pattern in Table 1 is the empirical case that the gain gate is necessary.

Finite validation probes quantify when the planner should act. Because verifier scores are bounded in $[0, 1]$, Hoeffding's inequality and a union bound give, with probability at least $1 - \delta$,

$$\varepsilon_k = 2 \sum_{i \in \mathcal{C}_k} q_{k,i} \sqrt{\frac{\log(2(|\mathcal{A}_k| + 1)|\mathcal{C}_k|/\delta)}{2n_i}}. \quad (16)$$

The factor of 2 arises because G_k is a difference of two utility estimates. The loop should proceed when

$$\max_{a \in \mathcal{A}_k} G_k(a; m) > 2\varepsilon_k + \lambda \operatorname{Cost}(a, m) + \beta \operatorname{Risk}(a, m), \quad (17)$$

and should otherwise shrink the update, switch recipe, expand candidates, or roll back. Thus monotonicity is not assumed; it is enforced as a decision criterion over measured cross-task evidence.

3.6 Memory-Guided Loop

EvolveLRM uses an Experience Bank to make planning cumulative rather than one-shot. The implementation stores three memory types in one SQLite table: `execution` rows for evaluation profiles, data files, training outcomes, and failures; `skill` rows for imported stage-specific experience; and `system_knowledge` rows for benchmark descriptions, method notes, and the core strategy. At decision time, the planner retrieves stage-specific entries as priors rather than hard rules; every selected action still has to pass candidate validation and later satisfy the measured gain gate.

Memory enters the action prior in Eq. 9. A simple abstraction is

$$\log p_{k,a} = \mu^\top \phi(a, h_k) + \sum_{m \in R_k(a)} \omega_m \operatorname{sim}(a, m), \quad (18)$$

where $\phi(a, h_k)$ encodes strategy features, $R_k(a)$ are retrieved experience entries relevant to the current stage, and $\operatorname{sim}(a, m)$ measures semantic or metadata overlap. In the present implementation, μ and ω_m are not learned. The strategy features are rendered as a prompt block, and retrieval is stage-keyed over the Experience Bank. The equation characterizes the effect of memory on the planner's proposal distribution; learning these weights from cross-run logs is future work.

The final profile updates memory and either triggers another round or stops when no candidate satisfies Eq. 17. The target remains transferable reasoning gain, not a single benchmark score.

4 Experiments

We evaluate EvolveLRM under a constrained but realistic setting: all training runs use a single node with 8× NVIDIA H100 GPUs. The experiments ask whether a closed-loop signal can avoid negative transfer and identify transferable data/recipe actions under this fixed adapter budget, rather than claiming large-scale leaderboard state of the art. The main evidence comes from completed Qwen2.5-7B post-training evaluations (Yang et al., 2024); two system studies examine executability under smaller model scales and different planner contexts. We report single-run results and treat system studies as loop-reliability evidence rather than final accuracy leaderboards.

4.1 Setup

Benchmarks. The in-domain benchmark is CLUB, a deterministic symbolic reasoning suite constructed by LogicEvolve (Du et al., 2026). CLUB and the larger ExCLUB pool expose the interface EvolveLRM needs: task metadata, verifier scores, provenance, and reward functions. We report exact CLUB Acc and partial Cell Acc. For transfer, we evaluate logic-heavy benchmarks (ZebraLogic, LogicGame, LogicGame-CoT, NYT Connections, ReasoningBench, and ComplexLogic), math/code benchmarks (MATH-500 (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), and LiveCodeBench (Jain et al., 2025)), and instruction/knowledge benchmarks (IFEval (Zhou et al., 2023), ComplexSub, and MMLU (Hendrycks et al., 2020)). ReasoningBench spans six reasoning types, while ComplexLogic/ComplexSub stress multi-step instruction following, calculation, code, and function-style constraints.

Models and summaries. The main study uses Qwen2.5-7B-Base. We compare no post-training, SFT-Full, SFT-LoRA, GRPO on the CLUB-scale pool, and GRPO on larger ExCLUB pools. **Logic Avg** averages ZebraLogic, LogicGame, LogicGame-CoT, NYT, ReasoningBench, and two ComplexLogic splits. **General Avg** averages MATH-500, GSM8K, IFEval, ComplexSub, LiveCodeBench, and MMLU.

Training recipes. The SFT baselines use the same CLUB-scale instruction data but differ in parameterization: SFT-Full updates all parameters, whereas SFT-LoRA applies a rank-8 adapter before merging. Both use supervised likelihood training for three epochs with the Qwen chat template. GRPO uses CLUB or ExCLUB verifier rewards, multiple completions per prompt, and a KL-stabilized policy update. The comparison therefore isolates different pressures on related solver-checked sources: imitation for SFT versus reward-weighted exploration for GRPO.

The ExCLUB pool covers 103 configured task types, of which 57 retained data after generator and solver/evaluator filtering; generator success is 66.9%, solver success is 18.4%. ExCLUB-30K caps each retained type at 1,000 examples, while ExCLUB-240K preserves the larger imbalanced pool.

Protocol and scope. All training rows are single-run checkpoints evaluated under fixed workbook protocols. External benchmarks use single-pass evaluation rather than pass@k leaderboards; the CoT variant is marked explicitly for LogicGame. Appendix J records decoding differences that should not be mixed across platforms. The completed rows are action evaluations under a single 8×H100 budget, not multi-seed or FLOP-matched leaderboard claims.

4.2 Main Results

Table 1 shows that the choice of training recipe matters more than source-format accuracy alone suggests. SFT-Full achieves the highest CLUB exact accuracy, improving from 0.8 to 19.0, but it reduces Logic Avg from 8.63 to 5.12 and General Avg from 32.41 to 22.33. SFT-LoRA shows the same transfer degradation. By contrast, GRPO on the CLUB-scale data obtains nearly the same exact CLUB accuracy as SFT-Full (18.8) while achieving the best Cell Acc (32.93), Logic Avg (19.90), and a much higher General Avg (47.38). This supports the framework's central premise: a source-format gain is insufficient unless it also improves cross-task validation utility.

Method	Data	CLUB Acc	Cell Acc	RBench	Logic Avg	General Avg
Base	–	0.8	4.46	29.89	8.63	32.41
SFT-Full	CLUB-7.7K	19.0	25.75	10.78	5.12	22.33
SFT-LoRA	CLUB-7.7K	12.8	23.59	3.11	2.41	20.46
GRPO	CLUB-7.7K	18.8	**32.93**	**39.44**	**19.90**	47.38
GRPO	ExCLUB-30K	4.2	19.29	14.33	14.00	49.44
GRPO	ExCLUB-240K	5.7	23.45	38.11	9.43	**52.59**

Table 1: Qwen2.5-7B post-training results. Scores are percentages. Static SFT reaches the highest source CLUB Acc (19.0) but lowers General Avg by 10.08 points; CLUB-scale GRPO nearly matches source accuracy (18.8) while improving both transfer averages.

Scaling GRPO to larger ExCLUB data changes the profile. The ExCLUB-240K checkpoint no longer maximizes CLUB exact accuracy, but it gives the best General Avg (52.59), MATH-500 (68.20), GSM8K (65.13), and MMLU (72.16). The expanded appendix table reports the full per-benchmark breakdown; the relevant point for action selection is already visible here: ExCLUB-30K improves several general benchmarks, while ReasoningBench rises sharply only in the 240K run.

The transfer trade-off is also visible in deltas from the base model. SFT-Full gains +18.2 exact CLUB points but loses 10.08 points on General Avg; SFT-LoRA has the same sign pattern. GRPO on CLUB-scale data gains nearly the same exact CLUB accuracy while improving both transfer summaries, and GRPO on ExCLUB-240K gives the largest general-reasoning gain despite modest in-domain exact accuracy. If the validation mixture is approximated by the reported averages, then the completed action scores rank SFT as negative-gain actions and GRPO as positive-gain actions: CLUB-scale GRPO improves Logic Avg by +11.27 and General Avg by +14.97, while ExCLUB-240K improves General Avg by +20.18. This is the empirical counterpart of Eq. 5: the best source-format score is not necessarily the best evolution action.

4.3 System Studies

Model scale. We ran the closed-loop driver on Qwen2.5 base models from 0.5B to 7B. These runs are not a homogeneous benchmark sweep because smaller models often failed before stable downstream training; they are pilot evidence about execution viability. As shown in Table 2, only the 7B model produced non-zero diagnostic accuracy under the current verifier and planner prompts, jus-

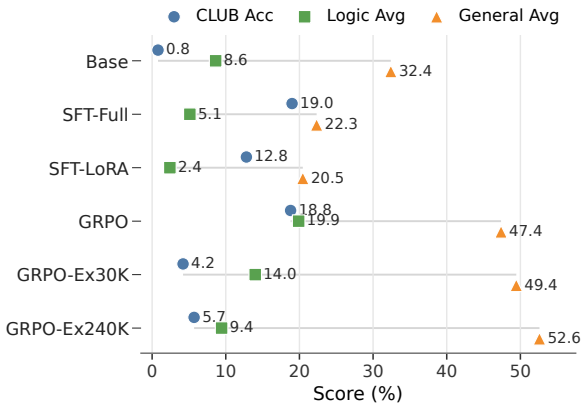

Figure 3: Qwen2.5-7B result profile across source, logic-transfer, and general-transfer metrics. SFT combines source-format gains with transfer losses; GRPO gives a more balanced positive-transfer profile.

Model	Runs	Slices	Acc	Cell
0.5B	4	0	–	–
1.5B	7	3	0.00	0.00
3B	6	29	0.00	0.00
7B	1	4	**11.88**	**14.12**

Table 2: Model-scale pilot. Slices are heterogeneous execution diagnostics; only the 7B run produced non-zero exact and cell accuracy under the current prompts.

tifying the main focus on 7B.

The negative small-model results have a positive read: the planner correctly stops at diagnosis rather than launching a training action with no measurable gain. This is the intended behavior of Eq. 17; small-model viability requires a richer bootstrapping curriculum or a base scale that can produce non-degenerate verifier feedback.

Planner context. We ablate the information provided to AutoPlanner: no auxiliary context, strategy memory only, experience retrieval only, and the full setting. Table 3 reports the deepest stage reached by at least one run. Human-in-the-loop diagnostic runs confirm that the backend can finish when choices are coherent. Among autonomous settings, reduced-context planners stop at serving or evaluation, whereas the full setting reaches training start after evaluation and data construction. This supports the memory prior in Eq. 9: retrieved experience is most reliable when combined with strategy guidance and candidate validation.

4.4 Discussion

The main result is not that one recipe dominates every metric, but that EvolveLRM exposes which

Planner	Context	Deepest stage
Human	No aids	Training finished
LLM	None	Server
LLM	Strategy memory	Server
LLM	Experience retrieval	Evaluation
LLM	Strategy + experience	**Training started**

Table 3: Planner-context study. Stages indicate the deepest point reached in the closed-loop pipeline; run counts vary by variant (1–5; see Appendix 15).

action transfers. Cell Acc makes this visible: SFT-Full reaches the best CLUB Acc (19.0) but only 25.75 Cell Acc, whereas CLUB-scale GRPO is nearly tied on CLUB Acc (18.8) and reaches 32.93 Cell Acc. This suggests that SFT learns the answer format, while verifier-weighted exploration learns more of the constraint structure.

Data scale is also not a monotone proxy for gain. ExCLUB-30K and ExCLUB-240K are drawn from the same generated pool, but their gains differ by benchmark: the 30K run is already strong on MATH-500 and GSM8K, while the 240K run is much stronger on ReasoningBench and General Avg. This is exactly the case for marginal Training Gain in Eq. 11: the planner should ask which additional slice still moves J_k, not whether the source pool is larger.

Finally, the system studies show why this is a control problem. A planner without usable context can start serving yet fail before evaluation or data construction; with strategy and experience, it can reach training dispatch, but the evidence is still limited by run count. Appendix Table 13 further shows task-level reward spread from 0.000 to 0.967, which motivates diagnosis below aggregate reward. We do not claim multi-seed confidence intervals; instead, Eq. 17 requires measured gain to exceed cost, risk, and finite-probe uncertainty before an action is scaled or retained.

5 Conclusion

EvolveLRM frames reasoning post-training as a planner-controlled, evidence-driven loop whose target is cross-task validation gain. On Qwen2.5-7B, CLUB-scale GRPO lifts Logic Avg from 8.63 to 19.90 and General Avg from 32.41 to 47.38, while static SFT reaches the highest CLUB Acc (19.0) but lowers General Avg to 22.33. Scaling GRPO to ExCLUB-240K further raises General Avg to 52.59.

Limitations

The experimental evidence is limited by compute constraints. All runs were conducted on a single 8× NVIDIA H100 node, so we do not provide multi-seed confidence intervals, long-horizon closed-loop trajectories, automatic rollback curves, or a complete task-by-task Training Gain heatmap. This limitation also defines the next research question: how should a planner's gain gate and rollback policy behave over many rounds, many seeds, and multiple nodes when the validation profile itself changes after each update? We report the model scales and hardware setting, but exact aggregate GPU-hour accounting across exploratory and failed runs was not available. The SFT and GRPO runs also differ in objective and effective sampling, so Table 1 should be read as action-level comparison under a fixed adapter budget rather than strict FLOP-matched optimization. We do not compare against every competitive training-time method, such as DPO/RPO variants, broader ReST/RFT recipes, or self-rewarded training without external verifiers. The model-scaling and planner-context studies are therefore reported as execution evidence rather than final accuracy comparisons.

The current system also exposes engineering limitations. Reduced-context planners can produce invalid structured outputs, and data-generation or training adapters may fail on some task/recipe combinations. These failures do not invalidate the completed offline training results, but they show that stronger schema constraints, adapter coverage, retry logic, and rollback policies are necessary before claiming fully autonomous multi-round self-improvement. In that sense, the failure modes are part of the research artifact: they identify the control boundaries that a deployable self-evolution system must make typed and auditable.

The main risks come from automating the loop that chooses data and training actions. A poorly constrained planner could over-optimize a narrow verifier, amplify artifacts in generated puzzle data, waste substantial compute, or apply the same machinery to unsafe task families. The reported system mitigates these risks by restricting experiments to rule-verifiable symbolic and reasoning benchmarks, keeping data provenance and structured logs, using cross-task validation rather than source accuracy alone, and treating the framework as a research prototype rather than a deployed autonomous training service. The work does not use human-subject data, personal profiles, surveillance data, or application-specific decision data.

References

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, page 41–48.

Jiangjie Chen, Qianyu He, Siyu Yuan, Aili Chen, Zhicheng Cai, Weinan Dai, Hongli Yu, Jiaze Chen, Xuefeng Li, Qiying Yu, Hao Zhou, and Mingxuan Wang. 2025. Enigmata: Scaling logical reasoning in large language models with synthetic verifiable puzzles. In *Advances in Neural Information Processing Systems*, volume 38, pages 3613–3661.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, and 1 others. 2026. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*.

Jinhua Du, Xuancheng Huang, Hanyu Lai, Aohan Zeng, and Jie Tang. 2026. LogicEvolve: Advancing logical reasoning toward self-evolution. OpenReview submission.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

Jiayi Gui, Yiming Liu, Jiale Cheng, Xiaotao Gu, Xiao Liu, Hongning Wang, Yuxiao Dong, Jie Tang, and Minlie Huang. 2025. LogicGame: Benchmarking rule-based reasoning abilities of large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1474–1491.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, and 1 others. 2023. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. 2025. R-Zero: Self-evolving reasoning LLM from zero data. *arXiv preprint arXiv:2508.05004*.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 1051–1068.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.

M Kumar, Benjamin Packer, and Daphne Koller. 2010. Self-paced learning for latent variable models. *Advances in neural information processing systems*, 23.

Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, and 1 others. 2025. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*.

Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. 2025. ZebraLogic: On the scaling limits of LLMs for logical reasoning. *arXiv preprint arXiv:2502.01100*.

Hanmeng Liu, Zhizhang Fu, Mengru Ding, Ruoxi Ning, Chaoli Zhang, Xiaozhang Liu, and Yue Zhang. 2025a. Logical reasoning in large language models: A survey. *arXiv preprint arXiv:2502.09100*.

Junteng Liu, Yuanxiang Fan, Jiang Zhuo, Han Ding, Yongyi Hu, Chi Zhang, Yiqi Shi, Shitong Weng, Aili Chen, Shiqi Chen, Mozhi Zhang, Pengyu Zhao, and Junxian He. 2025b. SynLogic: Synthesizing verifiable reasoning data at scale for learning logical reasoning and beyond. In *Advances in Neural Information Processing Systems*, volume 38, pages 100976–100997.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-Refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, and 1 others. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*.

Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. 2024. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13679–13707.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, and 1 others. 2025. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. 2025. Reasoning Gym: Reasoning environments for reinforcement learning with verifiable rewards. In *Advances in Neural Information Processing Systems*, volume 38.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and 1 others. 2023. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. 2024. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 13484–13508.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. 2025. On memorization of large language models in logical reasoning. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 2742–2785.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D Goodman. 2024. Quiet-STaR: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. STaR: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Chengshuai Zhao, Zhen Tan, Pingchuan Ma, Dawei Li, Bohan Jiang, Yancheng Wang, Yingzhen Yang, and Huan Liu. 2025. Is chain-of-thought reasoning of llms a mirage? a data distribution lens. *arXiv preprint arXiv:2508.01191*.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, and 1 others. 2024a. SGLang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024b. LLaMA-Factory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*.

A Proofs and Full Derivations

This appendix records the auxiliary mathematical details omitted from the main text to keep the control loop readable.

AUC-style Training Gain. In addition to threshold Training Gain in Eq. 10, an AUC-style score can integrate positive gain per unit cost:

$$\mathrm{TG}_{\mathrm{auc}}(a) = \frac{1}{B}\int_0^B \frac{[G_k(a;m)]_+}{\mathrm{Cost}(a,m)+\epsilon_0}\,dm. \quad (19)$$

We did not use this score for the reported selection decisions; it is a natural extension when multi-budget trajectories are available.

GRPO clipped surrogate. For a sampled response y_j, let $r_j(\theta) = \pi_\theta(y_j \mid x)/\pi_{\theta_k}(y_j \mid x)$. The clipped term summarized in Eq. 14 can be written as

$$\begin{aligned} \ell_j(\theta) &= \min(u_j(\theta), c_j(\theta)), \\ u_j(\theta) &= r_j(\theta)A_j, \\ c_j(\theta) &= \mathrm{clip}(r_j(\theta), 1-\epsilon_c, 1+\epsilon_c)A_j. \end{aligned} \quad (20)$$

The KL term in Eq. 14 stabilizes the update against reward over-optimization.

Finite-probe gain bound. For task i with validation size n_i, Hoeffding's inequality gives

$$\Pr\left(|\widehat{U}_i(\theta) - U_i(\theta)| \geq \epsilon_i\right) \leq 2\exp(-2n_i\epsilon_i^2). \quad (21)$$

Taking a union bound over validation tasks and candidate actions yields the radius in Eq. 16. If $\hat{a} = \arg\max_a \widehat{G}_k(a; m)$, then

$$G_k(\hat{a}; m) \geq \max_{a \in \mathcal{A}_k} G_k(a; m) - 2\varepsilon_k. \tag{22}$$

Therefore an action whose estimated gain gap exceeds probe uncertainty, cost, and risk is near-optimal among executable candidates.

B Implementation and Reproducibility Details

EvolveLRM relies on explicit implementation contracts, data interfaces, training adapters, logging mechanisms, and experimental artifacts. The executable system is organized around role interfaces, a constrained Planner protocol, memory and experience-bank mechanisms, data and reward schemas, SFT and GRPO adapters, a monitoring stack, and diagnostic results from the completed experiments.

The repository is organized around one implementation package, `codes/EvolveLRM/`, and one experiment package, `experiments/`. The most relevant code paths are the following: `AutoPlanner.py` implements the top-level loop; `Evaluator/AutoEvaluator.py`, `DataMaker/AutoDataMaker.py`, `Trainer/AutoTrainer.py`, and `Server/AutoServer.py` implement the four executable roles; `Memory/MemoryManager.py` implements persistent memory; `Utils/Logger.py` defines the structured event stream; and `Visualization/index.html` is a static log viewer used to monitor long-running experiments.

Two distinctions are important. First, EvolveLRM is not a single training recipe. It is a control layer that selects evaluation slices, data sources, task budgets, and training adapters. Second, several system studies in Section 4 are execution diagnostics. They measure where the current loop reaches, which failures occur, and why strategy and experience memory matter, rather than serving as exhaustive final-score comparisons.

C Role Interfaces and Control Flow

EvolveLRM uses a deliberately narrow interface between the Planner and the executable roles. Each role receives a JSON-like `specific_parameters` dictionary, resolves an adapter through `identity_id`, runs a framework-specific command, and writes structured events through a shared logger. This makes the system modular: replacing CLUB with another verifier, or replacing `verl` with another RL backend, changes an adapter rather than the Planner objective.

One evolution round has five phases. First, AutoServer deploys the current checkpoint and returns an API endpoint. Second, AutoPlanner chooses benchmarks, tasks, and evaluation sizes; AutoEvaluator then runs rule-verifiable evaluation through the LogicEvolve-compatible CLUB adapter and stores the resulting performance profile in memory. Third, AutoPlanner chooses datasets, tasks, and generation budgets; AutoDataMaker constructs task-specific data and samples each generated file to the requested size. Fourth, AutoPlanner chooses a training stage and method, integrates generated files into a round-specific training directory, stops the server to free GPU memory, and invokes AutoTrainer. Fifth, the resulting checkpoint becomes the next model state, and the round-level report is exported.

The implementation distinguishes `identity_id` from the run id. The run id is a logging prefix, for example a model key plus timestamp. By contrast, `identity_id` selects the adapter file, such as `default.py`, under a role-specific adapter directory. This choice avoids hard-coding framework variants into the Planner: a new serving backend or training recipe can be added by adding a new adapter file and selecting it with the same role contract.

C.1 Loop State

The state carried across rounds is a dictionary with the current model name/path, serving information, evaluation results, generated-data locations, training results, and memory summary. The Planner also maintains `public_info`, a nested report keyed by round. Each round stores the model identity before training, the evaluation results grouped by benchmark, the generated data paths grouped by dataset, and the training result with saved checkpoint name and path. At the end of execution, this report is written to a JSON file under the reports directory.

This state representation is intentionally redundant. The JSONL logger supports event-level debugging, the SQLite memory supports retrieval and cross-run comparison, and the exported report supports compact post-hoc aggregation. During the experiments, this redundancy was useful because failures could occur at different levels: mal-

Role	Input contract	Output contract
AutoServer	model name/path, host, port, GPU count, optional remote API	OpenAI-compatible endpoint and process id
AutoEvaluator	endpoint, benchmark, task list, per-task evaluation size	list of parsed verifier-result records
AutoDataMaker	endpoint, dataset, task list, per-task generation size	sampled JSONL files under the raw-data tree
AutoTrainer	model path, integrated data path, stage, method, save path	training log summary and saved checkpoint path
MemoryManager	run id, round id, role outputs, imported skills	queryable execution, skill, and system-knowledge rows

Table 4: Executable role contracts used by the implementation. The Planner never calls a framework directly; it calls these contracts and lets adapters translate them.

formed Planner JSON, server boot failure, invalid task size, missing data adapter, reward-script errors, and GPU memory pressure all leave different traces.

D Planner Decision Protocol

The Planner uses a language model, but the executed action is not free-form text. Each decision point is constrained by an explicit candidate set obtained from adapters or configuration. The response is parsed as JSON, checked for required fields, checked for non-empty selections, checked for membership in the candidate set, and retried up to `max_llm_retries` times. The default value used in the completed runs is five retries. If the Planner still cannot produce a valid response, the round closes as a failed execution rather than launching an undefined training job.

The Planner receives three kinds of context. The first is current state: model id, available frameworks, candidate tasks, and previous round outcomes. The second is strategy memory, loaded from a persistent memory category associated with `core_memory.md`. The third is stage-specific experience retrieved from the Experience Bank. Evaluation decisions retrieve evaluation skills; data-construction decisions retrieve data-construction skills; training and rollback decisions retrieve training skills. Retrieved experience is explicitly framed as a prior, not a rule. The final action still has to pass candidate validation and is later judged by measured cross-task gain.

Human-plan mode uses the same validation layer. With `-human-plan`, Planner prompts are printed to the terminal and the human enters a JSON response ending with `<\end>`. Evaluator, DataMaker, Trainer, and Server still use the real model and framework adapters. Thus the ablation compares only the planning layer, not a different execution backend.

Decision	Validation before execution
Evaluation benchmark	Selected benchmark must appear in the adapter-returned benchmark list.
Evaluation task	Selected tasks must belong to the chosen benchmark.
Evaluation size	Each selected size must be a positive integer and must match the selected tasks.
Data dataset	Selected dataset must appear in the adapter-returned dataset list.
Data task	Selected tasks must belong to the chosen dataset.
Data size	Each generation size must be a positive integer.
Training stage/method	The pair must match a configured method, e.g., `sft-full`, `sft-lora`, or `rl-grpo`.
Rollback/adjustment	Chosen checkpoint must be one of the stored historical model states.
Training-result parse	Parsed result must contain a non-empty `train_result`.

Table 5: Planner decisions and hard validation checks. Invalid JSON, empty fields, unknown candidates, and zero sizes trigger retries instead of execution.

D.1 Why Hard Validation Matters

The failure logs show that planning errors are not merely cosmetic. A valid-looking natural language answer can select a non-existent task, choose an empty size, or combine an incompatible training stage and method. Without validation, such responses would become shell commands or training configurations. EvolveLRM therefore treats LLM planning as proposal generation and uses the adapter layer as the executable type system. This design is also why the system studies report the deepest reached stage: reaching *evaluation*, *data construction*, or *training started* means that all pre-

ceding constrained decisions were executable.

D.2 Prompt Schemas and Invalid Action Handling

Each Planner prompt follows the same design pattern: it first states the role of the selector, then injects optional strategy and experience blocks, then lists candidates and historical evidence, and finally gives a strict JSON schema. This pattern keeps the prompt readable while preserving a machine-checkable output boundary. The implementation does not assume that the model follows the schema perfectly. It extracts the last JSON-like object from the response, checks it against the expected fields, and records a retry reason when validation fails.

The retry logs are useful in two ways. During a live run, they tell the operator whether the model is failing because it misunderstands the task or because the candidate set is too sparse. During analysis, they help distinguish model capability from framework fragility. For example, if a run reaches evaluation but repeatedly fails task-size selection, the bottleneck is not serving or verifier execution; it is structured planning under candidate constraints. If a run selects valid evaluation tasks but later fails because data preprocessing cannot find an adapter, the bottleneck is adapter coverage. This is why the system reports both final training outcomes and deepest reached stages.

D.3 Action Granularity

The Planner does not choose a single monolithic command. It chooses a sequence of small actions: which benchmark to evaluate, which tasks inside that benchmark, how many instances per task, which dataset to generate from, which data tasks to request, how large those requests should be, and which training recipe should consume the generated data. This granularity matters because the correct response to failure differs by stage. A weak evaluation profile calls for a different data slice; an invalid data-generation output calls for a retry or a different generator; a negative transfer result calls for changing the training recipe or rollback. A single command-level Planner would blur these decisions and make causal diagnosis harder.

E Memory and Experience Bank

Memory is implemented as a SQLite database with one table, `memories`. Rows are typed as `execution`, `skill`, or `system_knowledge`. Execution rows store evaluation, data-generation, training, and run-summary records. Skill rows store distilled experience entries imported from the external skill bank. System-knowledge rows store benchmark descriptions, training-method notes, and core strategy entries.

The Experience Bank is imported from a directory tree of skill files. Each skill entry contains a stage, a skill statement, and an explanation. Importing creates `memory_type=skill` rows whose category is the stage, such as `Evaluation`, `Data_Construction`, `Training`, `Inference`, or `System`. During planning, the system over-queries the top-ranked rows for a stage and randomly samples the number configured by `experience_top_k`. The default configuration uses five evaluation skills, eight data-construction skills, and five training skills.

This randomized top-K retrieval is a pragmatic compromise. Deterministic top-K retrieval repeatedly exposes the Planner to the same few entries; unrestricted random sampling makes the prompt noisy. Sampling from a high-importance pool gives repeated runs slight diversity while preserving the stage relevance of retrieved experience.

E.1 Execution Memory

After evaluation, `store_eval_result` writes a row containing the round id, benchmark, tasks, per-task sizes, model name, and result records. After data construction, `store_data_make_result` writes the dataset, selected tasks, generation sizes, model name, and generated files. After training, `store_train_result` writes the stage, method, model name, and parsed metrics. These rows support two later behaviors: the Planner can include historical outcomes in the next prompt, and a post-hoc analyzer can aggregate runs without re-reading every raw log line.

E.2 Strategy Memory

The core strategy file is imported into the same database as system knowledge. At Planner initialization, the file basename becomes the category key. If rows with that category exist, the Planner renders them as an explicit block of adaptive evolution principles. If the database has no matching rows, the block collapses to an empty string and the run proceeds. This design makes strategy memory a soft dependency: ablations can disable it with `-no-strategy`, and misconfigured memory does not crash the execution loop.

Prompt family	Required JSON content	Common rejected response
Benchmark selection	list of {benchmark, reason} objects	benchmark name not in adapter list, or empty selection
Task selection	list of {task, reason} objects for one benchmark or dataset	task invented by the Planner, repeated task, or all tasks selected by default
Task-size selection	nested benchmark/dataset, task, and positive integer size	string-valued size, zero size, missing selected task, or malformed nesting
Training selection	train_stage and train_method	incompatible pair such as a valid stage with an unavailable method
Adjustment decision	adjust_or_not, model_name, and model_path	checkpoint name not present in recorded history
Training-result parse	non-empty train_result summary	prose summary without JSON or empty result field

Table 6: Prompt families and failure cases rejected before execution. The LLM is allowed to propose, but only validated actions are executed.

Field	Purpose
memory_type	Separates execution logs, skills, and system knowledge.
module	Role that produced the row, such as Evaluator or Trainer.
run_id	Connects rows to a single evolution run.
category	Enables stage-specific retrieval, e.g., Evaluation.
title	Human-readable short descriptor.
content	JSON or text payload.
tags	Round, benchmark, method, and task tags.
importance	Retrieval priority used by ordered queries.

Table 7: Memory schema used by MemoryManager. The same table stores short-lived execution evidence and long-lived strategy knowledge.

Stage	Data representation
Raw data	Puzzle, question schema, answer, optional clues and source metadata.
SFT input	ShareGPT format with system and conversations.
SFT target	Assistant JSON block containing the verified solution.
GRPO input	Parquet records with data_source, prompt, and reward_model.
GRPO reward	Rule verifier receives the sampled response and serialized ground truth.

Table 8: Data schemas used by the training adapters. The same raw verified examples can support imitation learning or reward-based RL.

F Data Construction and Verification

AutoDataMaker uses a LogicEvolve-compatible adapter for CLUB and ExCLUB. The adapter first registers the current model API with the configured benchmark package, then invokes generation for each selected subtask. The generation command uses the selected dataset, subtask name, generation count, model name, and API endpoint. Generated task files are returned as JSONL paths. AutoDataMaker then samples the requested number of examples from each returned file using evenly spaced indices over the generated list. This keeps a requested slice representative of the returned file order while avoiding additional stochasticity at the sampling step.

The generated and selected examples are copied into a normalized raw-data tree before training. The integration step creates a round-specific training-data directory, copies each selected task file into that directory, and records source and destination paths in the log. The Trainer therefore consumes one integrated dataset name rather than many scattered generator outputs.

F.1 CLUB and ExCLUB Formatting

CLUB and ExCLUB examples are stored as puzzle-style records with a puzzle description, a question schema, and a verified answer. For SFT, preprocessing wraps each example into a ShareGPT-style two-turn conversation. The user message contains the puzzle, an instruction to solve it, and a JSON output template. The assistant message contains a JSON block whose solution field is the verified answer and whose reasoning field is left as an empty rationale slot. For RL, preprocessing creates prompt and reward_model fields. The reward model stores style=rule and a serialized ground truth record, so the external verifier can score sampled completions.

The implementation supports several CLUB-style task families, including maze, sudoku, string reasoning, traditional and extensible ZebraLogic, knights-and-knaves, line-forming games, tetris-like

Quantity	Value
Configured ExCLUB task types	103
Task types with retained data	57
Generator success rate	66.9%
Solver success rate	18.4%
Generator-and-solver success rate	17.3%
Valid raw JSON objects	241,784
Per-type cap in ExCLUB-30K	1,000
Capped ExCLUB examples	31,716
ExCLUB-30K train / validation	25,372 / 6,344
ExCLUB-240K train / validation	193,427 / 48,357

Table 9: Diagnostics for the ExCLUB generated-data pool used by the EvolveLRM training configurations. Success rates are generator-level logs; retained-example counts are after merge and sampling.

tasks, code-style puzzles, and train-game-card puzzles. Each family has a prompt grid that specifies the required output format. For tasks with explicit clues, preprocessing inserts both the puzzle and clue list into the prompt template. The uniform JSON solution template is important because downstream exact-match and cell-level scoring assume structured answers.

F.2 Generated-Data Pool Diagnostics

The ExCLUB data used by the reported GRPO variants comes from a July 25, 2025 generation pool. The raw pool contains 241,784 valid JSON objects and no invalid JSON rows under the merge script. Coverage is uneven: 57 task types have retained examples out of 103 configured task types. To reduce domination by high-volume tasks, one training configuration caps each task type at 1,000 examples, yielding 31,716 examples. The larger configuration keeps the full 241,784-example pool and therefore preserves the original task imbalance.

These diagnostics explain the difference between the two ExCLUB results in Table 12. ExCLUB-30K trades coverage for balance by applying a per-task cap. ExCLUB-240K uses substantially more data but also inherits stronger imbalance and was trained with the same basic schedule family, so its result should be read as a fixed-budget large-pool run rather than a saturated optimum for the source format.

F.3 Provenance and Leakage Controls

The main CLUB-scale training configuration excludes the CLUB v1 small evaluation split before training, leaving 7,722 training examples and 1,936 held-out examples. This filtering is not a theoretical guarantee against every form of benchmark familiarity, but it prevents the most direct train-test overlap for the in-domain source format. For ExCLUB, the generation batch and approximate pool size are reported because the pool is an evolved data source rather than a fixed public benchmark split. The evaluation therefore separates two claims: source-format accuracy on CLUB and transfer behavior on broader logic and general reasoning benchmarks.

The raw-to-training conversion preserves task provenance through fields such as `source`, `ability`, or `data_source`. This allows later audits to ask whether one task family dominates a training mixture or whether a failure is concentrated in a particular generated source. The current experiments use this provenance mainly for sanity checks and dataset summaries. A natural extension is to compute a full task-by-task Training Gain heatmap from the same provenance, but that experiment was outside the available compute budget.

F.4 Verifier Contract

The verifier contract is intentionally stricter than free-form answer judging. The model is asked to emit a JSON solution with a task-specific schema. Exact accuracy requires the whole structured solution to match the verifier. Cell accuracy gives partial credit when some structured fields are correct. This distinction is useful for diagnosing source-format learning: a model may learn the output scaffold and some constraints without solving the entire puzzle. In the reported CLUB-scale comparison, GRPO improves cell accuracy more strongly than static SFT, suggesting that reward-based exploration learns more of the constraint structure rather than only imitating answer format.

G Training Adapters

AutoTrainer exposes three completed adapters: full-parameter SFT, LoRA SFT, and GRPO. The Planner selects the pair $(\text{stage}, \text{method})$, and AutoTrainer maps it to a framework. SFT uses LLaMA-Factory (Zheng et al., 2024b); GRPO uses `verl` (Sheng et al., 2025). Before any training command is launched, AutoTrainer checks that the raw model exists, constructs a saved-model path if one was not provided, checks the raw-data path, runs the stage-specific preprocessing script, configures Weights & Biases logging, and resolves the adapter file.

The SFT adapters cap `max_samples` at 10,000 in the LLaMA-Factory configuration. This means the

Adapter	Main settings	Implementation notes
SFT-Full	full finetuning, Deepspeed ZeRO-3, batch size 1, grad accum. 2, LR $1e{-5}$, 3 epochs, bf16	Updates all weights; used to test source-format imitation.
SFT-LoRA	rank 8, all targets, batch size 1, grad accum. 8, LR $1e{-4}$, 3 epochs, bf16	Trains adapter, exports merged model, removes temporary adapter directory.
GRPO	train batch 64, response length 8192, prompt length 32768, rollout $n = 5$, actor LR $1e{-6}$, KL loss coef. 0.001, 15 epochs	Uses rule reward function for CLUB or ExCLUB through the `verl` PPO entry point.

Table 10: Training adapters used in the completed experiments. These are implementation hyperparameters, not abstract recommendations for all settings.

CLUB-size SFT experiments use the available 7.7K filtered examples, while larger ExCLUB SFT-style experiments require explicit care about sample limits and sequence length. The GRPO adapter instead reads Parquet train and validation files produced by the RL preprocessing step. The default split is random with a 20% validation fraction.

G.1 Reward Function Resolution

For RL, AutoTrainer resolves the reward function from `data_kind`. If `data_kind=club`, it uses `club_reward_fn`; if `data_kind=exclub`, it uses `exclub_reward_fn`. The paths are derived from the configured CLUB/ExCLUB verifier package. Adding a new data kind requires explicitly providing `reward_function_name` and `reward_function_path`. This prevents the Planner from silently applying a CLUB reward script to an incompatible data source.

G.2 Resource Notes

The completed experiments were run under a single-node $8\times$ H100 budget. In this setting, GRPO on CLUB-size data was feasible and informative. Full ExCLUB SFT-style training exposed sequence-length memory pressure: long examples around 24K tokens trigger quadratic attention memory growth and can out-of-memory even when shorter 2K-token variants run normally. This motivates reporting completed single-run evidence rather than broad multi-seed sweeps.

G.3 Why SFT and GRPO Share Raw Data but Differ in Pressure

The SFT adapters convert verified examples into target answers and optimize likelihood. This is valuable for teaching the model the task format, but it does not distinguish among multiple sampled reasoning attempts for the same prompt. GRPO, by contrast, samples several completions and uses the verifier reward to compare them within a group. As a result, GRPO can penalize plausible but wrong completions and reinforce correct structured outputs without requiring a handcrafted rationale. This difference explains why two methods can see the same CLUB-scale source and produce opposite transfer profiles: the optimization pressure, not only the data source, changes the learned behavior.

The implementation keeps these pressures comparable by routing both SFT and RL through the same raw-data integration step. The difference enters at preprocessing and adapter execution. SFT writes a single JSON file registered in LLaMA-Factory's dataset catalog. GRPO writes train and validation parquet files with serialized ground truth. The Planner therefore treats SFT-Full, SFT-LoRA, and GRPO as alternative update operators over a shared data-selection decision.

H Serving, Logging, and Human Monitoring

AutoServer deploys local models through an SGLang adapter (Zheng et al., 2024a) and returns an OpenAI-compatible endpoint. For multi-GPU serving, the adapter sets tensor parallelism based on model size and GPU count; for the 0.5B model it uses a single tensor-parallel shard, while larger models use tensor parallelism when multiple GPUs are available. Before training, AutoPlanner closes the server process so that training can reclaim GPU memory.

All roles share a `DefaultLogger`. Each event is one JSON object with run id, process id, timestamp, module, task, function, input, return code, output, error, auxiliary fields, and combined stdout/stderr. Subprocess output is also captured in same-name `.log` files when commands are launched through the logging wrapper. This was essential for diagnosing long runs, because the highest-level failure may be a Planner exception while the root cause is buried inside a framework log.

Logged field	Interpretation
`run_id`	Prefix and timestamp identifying one self-evolution run.
`module`	Planner, Server, Evaluator, DataMaker, Trainer, or Memory.
`task`	Coarse stage such as `plan-evaluator` or `train`.
`function`	Method or adapter function that emitted the event.
`input`	Structured input dictionary or command string.
`returncode`	Subprocess exit status when available.
`output`	Main result, parsed summary, or command template.
`error`	Exception or error string.
`out+err`	Captured raw stdout/stderr for long framework commands.

Table 11: Structured event fields. These fields are consumed both by manual inspection and by the static visualization page.

H.1 Visualization Tool

The project includes a zero-dependency static visualization page under `Visualization/index.html`. It can be opened directly in a browser and does not require a local server. The page loads individual `.jsonl` logs, raw `.log` files, or an entire log directory through a file picker or drag-and-drop. The JSONL view displays a top statistics bar, per-round pipeline overview, filter chips by module and status, and expandable event cards. Error cards are highlighted, completed events are separated from start/in-progress events, and large fields such as `output_and_error` can be expanded or copied. The raw-log view collapses progress-bar noise and highlights ERROR/WARN/INFO lines.

This visualization is not part of the optimization objective, but it is part of the research workflow. It allows a human operator to monitor whether a run is stuck in server boot, failing JSON validation, looping over evaluator retries, generating empty data files, or entering training. For long self-evolution experiments, this reduces the gap between a black-box automation and an auditable experimental system.

H.2 Operator Audit Workflow

For long runs, the visualization tool supports a practical audit sequence. First, load the run JSONL and check the per-round pipeline bar. If a stage is missing, filter by the corresponding module and inspect the last event before the gap. Second, inspect retry events emitted by the Planner. Repeated retries at the same decision point usually mean that either the prompt schema is too hard for the Planner or the candidate list is incompatible with the current state. Third, if a subprocess was launched, open the captured raw log and collapse progress lines. This usually reveals whether the framework itself failed, whether the adapter produced an invalid command, or whether the process was still running when the high-level log stopped updating.

This workflow was used while interpreting the model-scale and Planner-context studies. For example, a run that ends after server startup is different from a run that evaluates successfully but cannot choose generation sizes. Both are failures from the perspective of autonomous improvement, but they imply different future work: serving stability, planner schema constraints, adapter coverage, or data generator robustness.

I Experimental Artifacts

The completed benchmark workbook is `experiments/result.xlsx`. The plotting script regenerates the result-profile figure and CSV summaries under `experiments/paper_analysis/`. The model-scaling and Planner-context execution reports are stored in the corresponding experiment run directories and aggregated into the CSV summaries used in the reported analyses. Auxiliary experiment records document data filtering, split construction, decoding settings, and ExCLUB generation coverage; the EvolveLRM comparisons use only the records that define the training configurations reported here.

I.1 Artifact Provenance, Terms, and Intended Use

The study uses public research artifacts and generated symbolic-reasoning data. The base models are Qwen2.5 checkpoints (Yang et al., 2024). SFT adapters rely on LLaMA-Factory (Zheng et al., 2024b); RL training uses `verl` (Sheng et al., 2025); local serving uses SGLang (Zheng et al., 2024a). The checked-out copies of `verl` and SGLang used in the implementation carry Apache-2.0 licenses, and all external models, benchmarks, and software are used as research artifacts under their respective public licenses or terms. Any released EvolveLRM code or data package should keep those upstream notices intact and should be used for research on auditable reasoning-model training rather than for autonomous deployment in safety-critical applica-

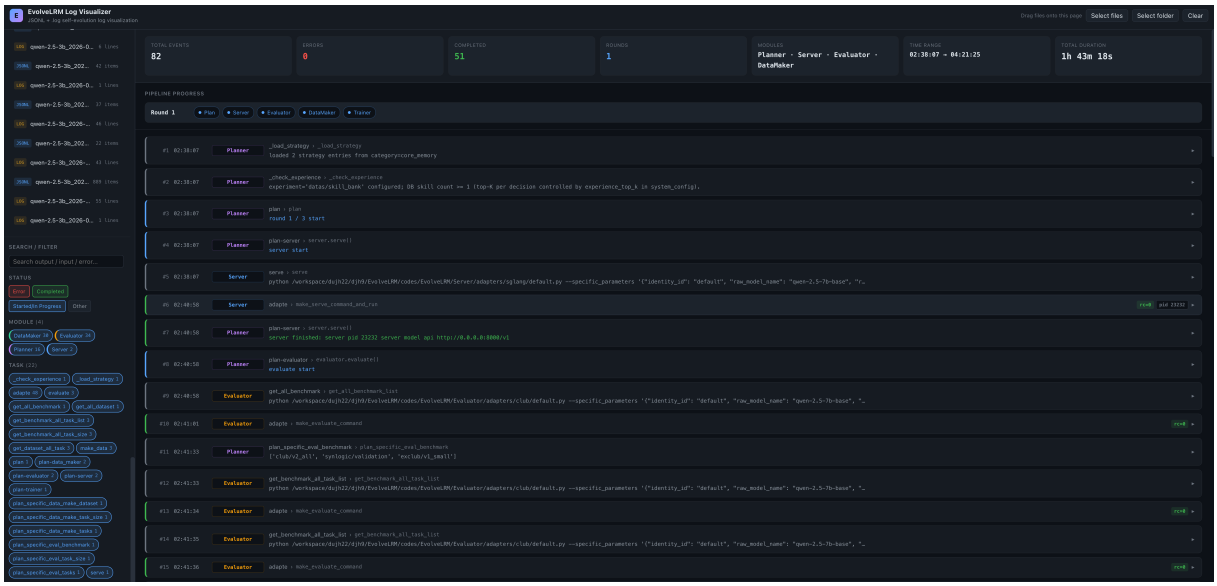

Figure 4: Static monitoring interface for an EvolveLRM run. The view shows loaded log files, run-level statistics, per-round pipeline progress, module/status filters, and expandable structured events.

tions.

CLUB and ExCLUB are deterministic, programmatically generated symbolic-reasoning artifacts from LogicEvolve (Du et al., 2026). Their intended role in this paper is to provide verifiable sensors and training sources for logical-reasoning post-training. We do not repackage personal, social-media, biomedical, legal, or user-interaction data. The generated examples consist of puzzle descriptions, task schemas, and verified structured answers. We checked the generated pools through JSON validity, solver/evaluator verification, train/evaluation split filtering, task-provenance aggregation, and manual inspection of representative records. These checks are designed to detect malformed outputs, verifier mismatch, direct train-test overlap, and accidental non-puzzle content; no fields are intended to name or uniquely identify individual people.

I.2 AI Assistant Use

AI assistants were used as research and writing aids for code navigation, drafting, editing, consistency checking, and LaTeX/table/figure preparation. The authors remain responsible for the research claims, experimental design, data interpretation, paper text, and final submission decisions. AI assistants are not listed as authors and did not make autonomous scientific decisions.

I.3 Data Configurations

The result workbook uses the following data configurations. Config 0 denotes the base model without post-training. Config 1 uses CLUB v1 after filtering the CLUB v1 small evaluation split, leaving 7,722 training examples and 1,936 test examples. Config 2 applies an input-length filter to Config 1. Config 3 uses ExCLUB generated on July 25, 2025 with at most 1,000 examples per task, giving 31,716 examples split into 25,372 training and 6,344 validation examples. Config 4 uses the full 241,784-example ExCLUB generated pool, split into 193,427 training and 48,357 validation examples. Negative config IDs correspond to external reference models or external training-data mixtures and are excluded from the primary EvolveLRM comparisons.

Table 12 summarizes the pattern across source and transfer metrics. SFT-Full and SFT-LoRA improve CLUB source-format accuracy but reduce both summarized transfer metrics and several individual general benchmarks. CLUB-size GRPO improves source and transfer metrics together. ExCLUB-240K GRPO gives the strongest general average and the highest MATH-500, GSM8K, and MMLU values among the reported checkpoints, even though it does not maximize CLUB exact accuracy. This is the empirical case for using cross-task gain rather than source accuracy as the control signal.

J Metric Definitions and Interpretation

CLUB Acc is exact structured accuracy on the in-domain CLUB evaluation. Cell Acc is a partial-

Method	CLUB Acc	Cell Acc	Logic Avg	General Avg	Reasoning Bench	MATH 500	GSM8K	MMLU
Base	0.8	4.46	8.63	32.41	29.89	47.60	1.52	66.24
SFT-Full	**19.0**	25.75	5.12	22.33	10.78	19.60	0.00	63.52
SFT-LoRA	12.8	23.59	2.41	20.46	3.11	19.20	0.53	61.80
GRPO	18.8	**32.93**	**19.90**	47.38	**39.44**	62.80	42.53	69.24
GRPO-Ex30K	4.2	19.29	14.00	49.44	14.33	65.33	55.04	69.00
GRPO-Ex240K	5.7	23.45	9.43	**52.59**	38.11	**68.20**	**65.13**	**72.16**

Table 12: Expanded result summary from the analysis CSV. Values are percentages. Logic Avg and General Avg are the non-weighted summaries used in Section 4.

credit structured score that counts correct fields even when the complete answer is not exactly correct. Logic Avg is the unweighted average over the logic-oriented transfer columns used in the analysis CSV: ZebraLogic, LogicGame, LogicGame-CoT, NYT, ReasoningBench, and two ComplexLogic splits. General Avg is the unweighted average over MATH-500, GSM8K, IFEval, ComplexSub, LiveCodeBench, and MMLU. These averages are not meant to replace individual benchmarks. They are compact summaries for comparing whether a training recipe improves beyond its source format.

The qualitative pattern is visible even without the averages. SFT-Full reaches the best CLUB exact score, but ReasoningBench drops from 29.89 to 10.78 and MATH-500 drops from 47.60 to 19.60. SFT-LoRA shows a similar decline. CLUB-size GRPO nearly matches SFT-Full on exact CLUB accuracy while improving ReasoningBench, MATH-500, GSM8K, LiveCodeBench, and MMLU relative to the base. ExCLUB-240K GRPO is weaker on exact CLUB than CLUB-size GRPO but produces the strongest General Avg. These results support modeling EvolveLRM as a cross-task validation-gain system rather than a source-task maximizer.

J.1 Evaluation Protocol Notes

The primary comparisons use the workbook and derived CSV summaries as the common evaluation source. Decoding records show that CLUB scores can change with output budget and sampling settings: local training-model evaluations used a 1,024-token output budget with temperature 0.7, while some platform evaluations used an 8,192-token budget with deterministic decoding. Scores from different decoding protocols are therefore not mixed as if they were identical measurements. The reported comparisons should be read as fixed-protocol single-run results, not as cross-platform leaderboard claims.

J.2 Execution Diagnostic Interpretation

The model-scale and Planner-context studies are not conventional benchmark tables because the compared runs do not all reach the same terminal state. Some fail before evaluation, some evaluate but do not construct data, and one full-context setting reaches training dispatch. Treating these as final accuracy rows would hide the main signal. The relevant question is whether the closed-loop stack can progress through serving, planning, evaluation, generation, preprocessing, and training. Reporting the deepest reached stage therefore makes the system limitation visible rather than burying it behind missing scores.

J.3 Single-Run Evidence

All reported training results are single-run outcomes under the available compute budget. The artifact paths and logs identify the full execution trace, but the experiments do not provide multi-seed confidence intervals. The results demonstrate that the completed GRPO recipes can improve transfer, while variance estimates, long-horizon trajectories, and full task-pair Training Gain heatmaps remain future work.

K Additional System Diagnostics

K.1 GRPO Reward Validation

The CLUB GRPO run records task-level validation reward means during training. These values are not benchmark accuracies: they are verifier rewards computed on the RL validation split. They are useful because they show that the reward signal is task-dependent rather than uniformly saturated. Some task families provide strong reward signal, such as traditional ZebraLogic and line-forming games, while maze remains flat in the recorded run.

Validation task	Reward mean@1
Train-game-card puzzle	0.081
Code	0.347
Knights-and-knaves	0.610
Maze	0.000
Sudoku	0.432
Line-forming games	0.852
String reasoning	0.256
Extensible ZebraLogic	0.262
Traditional ZebraLogic	**0.967**
Tetris	0.739

Table 13: Task-level validation reward diagnostics from the CLUB GRPO run. The table diagnoses the RL training signal and is separate from the final benchmark results.

The model-scale pilot clarifies why the main training comparison uses Qwen2.5-7B. Smaller models did not produce reliable non-zero verifier signals in the selected slices. For a gain-driven loop, this is a serious bottleneck: if every evaluated slice has zero accuracy, the estimated validation objective becomes flat and the Planner cannot distinguish promising training actions from uninformative ones. The result does not imply that smaller models cannot be improved; it says that the current verifier, task selection, and bootstrapping strategy do not yet provide enough signal for autonomous multi-round improvement at those sizes.

Table 15 should be interpreted as a systems diagnostic. Human planning can complete the pipeline when the operator chooses compatible actions, but it is slow and not scalable. LLM-only planning reaches server startup but does not reliably pass evaluation selection and execution. Experience alone reaches evaluation more often than the no-context setting, but can still fail at task-size JSON validation. The full setting combines top-level strategy with stage-specific experience and reaches training dispatch in the completed run. This supports the design choice in Section 3.6: memory should bias the action prior, while hard validation and measured gain determine whether the action is useful.

L Observed Failure Modes and Mitigations

The failures are informative because EvolveLRM is an end-to-end closed-loop system, not a single offline training command. A system can improve a checkpoint only if it can serve the model, evaluate it, construct data, preprocess that data, choose a compatible training adapter, launch training, and recover a usable checkpoint. The present implementation makes these stages explicit and auditable. At the same time, the failures show why the study does not claim fully autonomous long-horizon self-improvement: stronger schema constraints, broader adapter coverage, retry policies, and rollback policies remain necessary.

L.1 Planned Engineering Remedies

The current implementation already separates proposal, validation, execution, logging, and memory. The next engineering step is to make these boundaries stricter. Planner outputs can be constrained by JSON schema decoding or tool-call APIs rather than post-hoc parsing. Adapter availability can be exposed to the Planner as a typed capability table, preventing choices that cannot be preprocessed. DataMaker can return richer status codes, such as empty generation, verifier mismatch, task timeout, or partial success, rather than only file paths. Trainer can expose resource estimates before launching a long job. Finally, rollback can be made automatic when a new checkpoint improves source accuracy but reduces the cross-task validation objective.

These remedies preserve the cross-task validation objective. They are systems work needed to make the same objective reliable across more rounds, more models, and more data sources.

M Reproduction Checklist

The following checklist summarizes the intended reproduction flow. It is written at the level of repository artifacts rather than machine-specific absolute paths.

Environment. Install the package from `codes/EvolveLRM/`. On a new Linux machine, run the setup script from the project root to initialize external frameworks and environments. Check that the framework paths in the configuration files point to the local checkout. For training monitoring, configure Weights & Biases credentials before launching SFT or GRPO jobs.

Memory initialization. Import the core memory file as system knowledge, then import the skill bank. The strategy and experience features are soft dependencies: if they are absent, the corresponding prompt blocks are empty; if they are present, the Planner uses them in each stage-specific decision.

Model	Reports	Eval reports	Slices	Mean Acc	Mean Cell
Qwen2.5-0.5B	4	0	0	–	–
Qwen2.5-1.5B	7	1	3	0.00	0.00
Qwen2.5-3B	6	4	29	0.00	0.00
Qwen2.5-7B	1	1	4	**11.88**	**14.12**

Table 14: Model-scale execution diagnostics. The table counts generated reports, reports containing evaluation results, evaluated slices, and mean exact accuracy on those slices.

Variant	Planner	Context	Runs	Server	Eval	Data	Train start
1	Human	no aids	3	2	1	1	1
2	LLM	none	2	1	0	0	0
3	LLM	strategy	2	1	0	0	0
4	LLM	experience	5	5	1	0	0
5	LLM	strategy + experience	1	1	1	1	**1**

Table 15: Planner-context diagnostics. Columns report how many runs reached major milestones. The full-context LLM setting reached training dispatch in the completed ablation runs.

Model-scale runs. Use the model-scaling configuration directory under `experiments/`. Launch one model key at a time with `python codes/EvolveLRM/main.py -model <model-key>`. The model key becomes the log prefix, which makes aggregation across machines easier.

Planner ablation. Use the flags `-human-plan`, `-no-strategy`, `-no-experience`, and `-identity-suffix` to reproduce the five Planner settings. The Planner-ablation helper script under `experiments/` runs all settings sequentially. The human setting requires entering valid JSON responses manually at each decision point.

Result aggregation. Run the plotting script under `experiments/` to regenerate CSV summaries and the result-profile figure. The CSV summaries used for Tables 12, 14, and 15 are kept under `experiments/paper_analysis/`.

Monitoring. Open `codes/EvolveLRM/Visualization/index.html` in a browser and load the run's JSONL and raw log files. During a run, the most useful first check is the pipeline overview by round. If the pipeline stops early, expand the latest red event and inspect both the parsed error and the captured stdout/stderr. For server issues, also inspect the redundant server log.

Expected outputs. A successful run produces JSONL logs, optional raw framework logs, stored memories, an exported report, generated raw-data files, preprocessed train files, and a saved checkpoint. The exact set depends on the deepest reached stage. For example, a run that fails during evaluation may still have server logs and Planner retries but no generated data or checkpoint; this is expected and is how the system studies measure execution progress.

Failure mode	Observed cause	Mitigation in current system
Invalid Planner JSON	LLM response missing required fields, using unknown candidates, or returning zero sizes.	JSON extraction, membership checks, positive-size checks, and retry limit.
Flat evaluation signal	Small models scored zero on evaluated slices.	Report as execution evidence; concentrate main training comparison on 7B; future work should add bootstrapping.
Server/resource conflict	Serving process holds GPU memory needed for training.	Planner closes server before invoking Trainer.
Empty generated file	Data generator returns an empty JSONL file for a selected task.	DataMaker logs an error and skips empty files.
Missing adapter	Planner selects a stage/method whose preprocessing adapter does not exist.	Adapter existence check fails early with explicit path.
Long-context memory pressure	ExCLUB examples can trigger high activation and attention memory.	Report completed checkpoints; use filtering, shorter cutoffs, or curriculum length in future runs.
Reward mismatch	RL data kind lacks a compatible reward function.	AutoTrainer resolves known CLUB/ExCLUB reward functions and rejects unknown data kinds unless explicit paths are provided.

Table 16: Failure modes surfaced by implementation and experiment logs. These are engineering limitations rather than hidden evaluation changes.