RECURSIVE SELF-IMPROVEMENT THROUGH DEFECT-DRIVEN VERIFIABLE ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-improvement (RSI) requires a model to identify what to learn next and obtain reliable supervision for capabilities it has not yet mastered. We introduce a defect-driven RSI framework that turns recurring model failures into verifiable learning environments. We find that many failures are prompt-recoverable: the model fails unaided but succeeds with defect-specific structural guidance. This observation motivates using unaided performance and assisted verification success to target capabilities that remain deficient but are accessible under guidance. We use initialized executable environments to adapt training-problem distributions, generate and verify guided solution trajectories, and train the learner without guidance in its inputs. After each round, we reassess the updated learner to adjust data allocation and use it for controlled rewriting of subsequent problem descriptions, closing the loop between capability improvement and curriculum adaptation. In experiments with SIRL-Qwen2.5-7B on three optimization modeling benchmarks, we obtain relative pass@1 gains of 34.02% on MAMO ComplexLP and 12.62% on OptMATH-bench-193 over the base model. Across rounds, we observe an overall increase in the proportion of unaided successes, with defect-specific guidance, we obtain 6.49 times as many verified trajectories as with unaided generation under the same sampling budget. Together, we use defect-specific structural guidance to link learning-target selection, dynamic capability assessment, and training-data generation, and we use the constructed environments to support execution and verification throughout this recursive improvement process.

1 INTRODUCTION

Recursive self-improvement (RSI) seeks to improve models through feedback and supervision generated within the improvement process itself (Zelikman et al., 2022; Singh et al., 2023). This requires resolving two coupled questions: what should the current model learn next, and how can it obtain reliable supervision for capabilities it has not yet mastered? Tasks already solved reliably may offer limited learning signal, whereas tasks far beyond the learner's reach rarely yield correct self-generated solutions. Useful RSI data must therefore expose a capability that remains deficient while preserving a route to reliable supervision.

Our central observation is that some failures are *prompt-recoverable*: the model fails unaided but succeeds with defect-specific structural guidance. Recoverability thus provides an operational signal for identifying tasks that remain challenging without assistance but admit verified model-generated solutions under guidance. Consider traveling salesman problem (TSP) formulations. A model may impose one incoming and one outgoing arc per city but omit subtour elimination, allowing disconnected cycles. We supply defect-specific structural guidance on the missing connectivity constraints, enabling the model to construct a valid formulation without providing the instance's optimal tour or objective value.

We build an RSI framework using executable environments initialized by a strong external model under human review. We use these environments to generate problems exposing diagnosed defects and verify guided solution trajectories for supervised training without auxiliary guidance in the inputs. After each update, we reassess the learner to adapt the curriculum and use it for controlled rewriting of subsequent problem descriptions. Building on capability-adaptive task and environment construction (Huang et al., 2025a; Zeng et al., 2025; Shi et al., 2026), we combine unaided mastery

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

and assisted verification success to guide the evolving training distribution toward capabilities that remain deficient but accessible under guidance.

We evaluate SIRL-Qwen2.5-7B (Chen et al., 2025a) on three optimization modeling benchmarks, with relative pass@1 gains over the base model of 34.02% on MAMO ComplexLP and 12.62% on OptMATH-bench-193. We observe an increasing proportion of unaided successes and a declining proportion of assistance-only successes across rounds. With structural assistance, we obtain 6.49 times as many verified trajectories under a fixed sampling budget.

Our contributions are threefold:

- **A defect-driven framework for environment-mediated RSI.** We translate diagnosed modeling defects into executable environments with targeted instance generation and programmatic solution verification, providing a basis for repeated capability assessment and verified supervision collection.
- **A prompt-recoverability mechanism for recursive self-improvement.** We use defect-specific structural guidance to screen seed problems for recoverable failures, inform recoverability-guided data allocation, and elicit solution trajectories for newly generated instances. In doing so, we connect learning-target selection, dynamic capability assessment, and training-data generation within the executable environments, which support execution and verification throughout the loop.
- **Improved unaided modeling and efficient supervision generation.** Across three optimization modeling benchmarks, we improve SIRL-Qwen2.5-7B's pass@1 from 54.52% to 73.07% on MAMO ComplexLP and from 35.58% to 40.07% on OptMATH-bench-193. We observe an increasing proportion of unaided successes across rounds and obtain 6.49 times as many verified trajectories with defect-specific guidance as with unaided generation under the same sampling budget.

2 Related Work

Self-improvement and guided supervision. Self-improvement methods obtain training signals from model-generated outputs and feedback. Correctness-filtered self-training alternates solution sampling, selection, and supervised updates, as in STaR and ReST-EM (Zelikman et al., 2022; Singh et al., 2023). Additional information during generation can expand the available supervision: STaR's rationalization supplies the correct answer after an unsuccessful attempt and removes that hint from the corresponding training input (Zelikman et al., 2022), while Distilling Step-by-Step uses teacher-generated rationales as additional supervision for smaller models (Hsieh et al., 2023). Adaptation can also be learned explicitly: SEAL optimizes generated training data and update directives using downstream performance (Zweiger et al., 2025). Building on guided self-training, we use the learner's response to defect-specific structural guidance to identify what it should learn next. Across rounds, we reuse the same guidance to assess the learner's evolving capabilities and elicit solution trajectories for newly generated instances, which we verify before training.

Adaptive curricula and executable environments. Adaptive data construction addresses which tasks to present as a learner's capabilities change. At the task level, failure-driven synthesis targets incorrect examples or diagnosed weaknesses, as in LLM2LLM, WarriorMath, and Socratic-Zero (Lee et al., 2024; Chen et al., 2025b; Wang et al., 2025). Self-play also couples task proposal with solving: Absolute Zero uses code execution to validate proposed tasks and answers (Zhao et al., 2025), while R-Zero rewards questions according to uncertainty estimated from the Solver's answer consistency (Huang et al., 2025a). At the environment level, adaptation operates over distributions of generated instances or the programs defining them. RLVE advances a difficulty range when accuracy at its upper bound exceeds a threshold (Zeng et al., 2025), while EvoEnv synthesizes environment programs with validity checks and solver-relative difficulty calibration (Shi et al., 2026). We use defect-driven executable environments to repeatedly assess the evolving learner, generate new instances that meet specified structural and difficulty requirements, and collect guided trajectories with execution and structural verification. We then use the assisted–unassisted performance gap measured in these assessments to adjust subsequent data allocation.

Learning for optimization modeling. Optimization modeling exposes formulation defects, such as missing constraints, as explicit mathematical structures that can recur across parameterized instances. ORLM develops training corpora through instruction-based synthesis (Huang et al., 2024), and OptMATH combines controllable instance generation with back-translation and rejection-based validation (Lu et al., 2025). Learning and data generation are also coupled through execution feedback: AutoOR combines verified synthesis with RL and curriculum training (Motwani et al., 2026), while EvoOptiGraph uses graph-derived weakness signals to guide structural instance evolution (Kang et al., 2026). Building on these advances, we turn diagnosed formulation defects into executable learning environments built from the problems' core mathematical formulations, including their variables, objectives, and constraints. We use these shared formulations for instance generation, solver-derived reference answers, and defect-specific checks to provide a consistent mathematical basis for verifying candidate solutions.

3 METHODOLOGY

We turn recurring modeling defects into executable learning environments and use capability feedback to organize a recursive curriculum. We use prompt-recoverable failures to target capabilities that remain deficient but admit verified solutions under guidance. During initialization, we use an external model to diagnose failures and extract the core mathematical formulations used to build generators, structural guidance, and verification procedures under human review. In each recursive round, we assess the current learner with and without guidance, allocate training examples across environments and structural strata, and collect verified solution trajectories. We train the learner without auxiliary guidance in its inputs and reassess it before the next round.

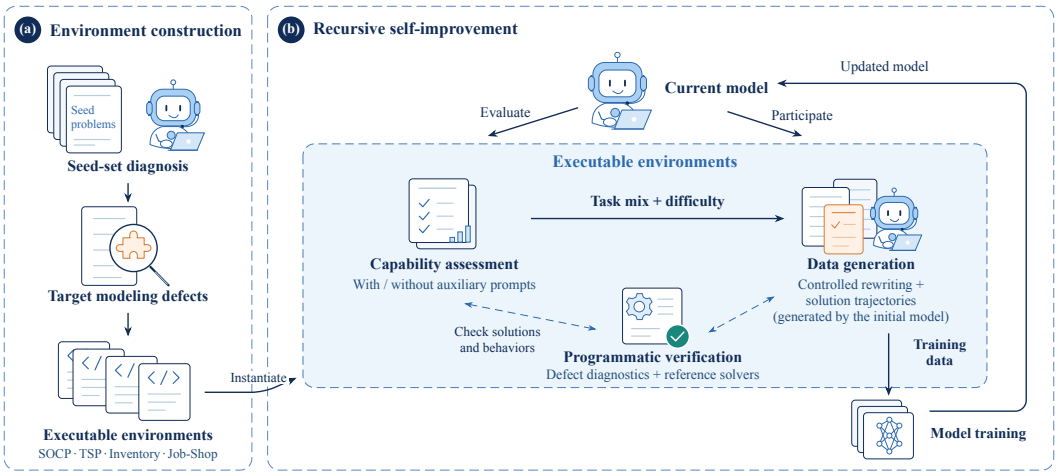


Figure 1: Defect-driven environment-mediated recursive self-improvement. (a) Recurring modeling defects motivate reusable executable environments grounded in the problems' variables, objectives, and constraints. (b) Within these environments, failures recoverable with guidance identify learning opportunities. The current learner's assessments and controlled problem rewriting shape subsequent training data, while a fixed initial model supplies guided trajectories for verification. Training on verified trajectories without auxiliary guidance updates the learner, whose changing capabilities guide the next round.

3.1 DEFECT-DRIVEN EXECUTABLE ENVIRONMENTS

Initialization and diagnosis. We screen the initial model on seed problems with K_{probe} unaided attempts per problem. We use an external model to examine failed responses for recurring formulation defects and supply defect-specific structural guidance without instance-specific answers. On problems with zero observed success, we compare matched guidance with unaided resampling, generic prompting, and mismatched guidance, giving each condition K_{recover} new attempts. We record an *observed prompt-recoverable failure* when matched guidance yields at least one verified

solution after zero observed unaided successes. We review the extracted defects, guidance, and environments during initialization and subsequently reuse them.

A shared mathematical specification. For each diagnosed problem family, we use the external model to extract its variables, objective, and constraints. This core formulation grounds parameterized instance generation, reference solving, and candidate-solution verification. Each environment $e \in \mathcal{E}$ contains a generator G_e, verification procedures V_e, structural guidance $P_{a,e}$, and L_e generation strata. Defect-exposure rules screen generated instances to select problems that can expose the targeted defects in the model's formulations. We use each environment's solver to compute reference answers independently. We therefore ground verification in the intended formulation, not merely in whether a candidate program executes.

We instantiate environments for job-shop scheduling, inventory transshipment, SOCP, and TSP.

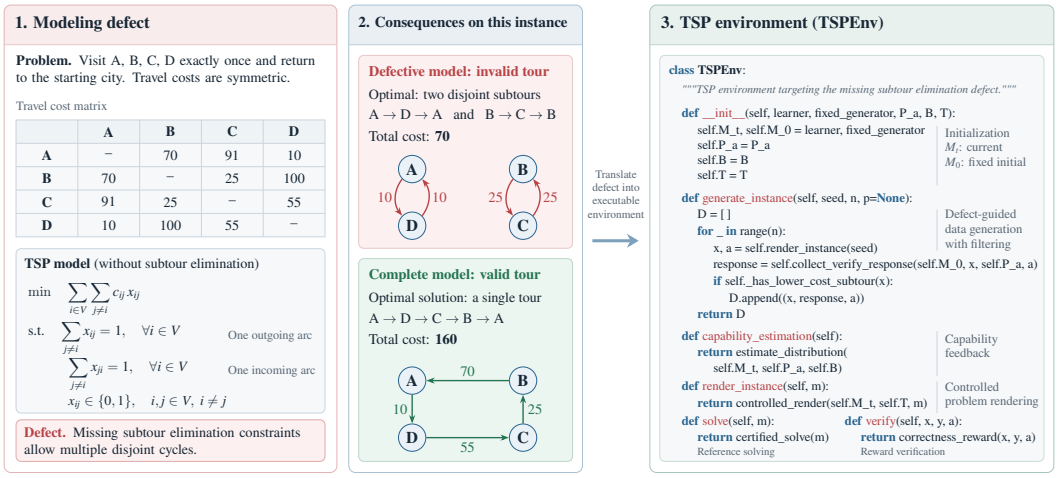


Figure 2: A TSP environment grounded in its core formulation. Degree constraints alone admit the two disconnected cycles $A \to D \to A$ and $B \to C \to B$ at cost 70, whereas the complete formulation yields a valid optimal tour $A \to D \to C \to B \to A$ at cost 160. We use this gap to filter generated instances that expose missing subtour elimination. The resulting environment assesses capability with structural guidance, renders controlled problem descriptions, computes reference solutions, and verifies candidate responses.

3.2 Capability Assessment and Prompt Recoverability

Before round t, we evaluate the current learner π_{θ_t} on fixed DEV instances from each environment e and structural stratum ℓ, with m attempts per instance in each condition. The capability assessment mechanism uses the same solution-correctness verifier in both conditions: execution against the reference answer and task-specific structural checks. Only the provision of defect-specific guidance changes. We keep instances, prompts, the sampling seed, and m fixed across rounds. Let $n_{e,\ell}$ be the number of attempts per condition and $c^{\text{plain}}_{e,\ell,t}$, $c^{\text{aux}}_{e,\ell,t}$ the corresponding verified-success counts. Define

$$p^{\text{plain}}_{e,\ell,t} = \frac{c^{\text{plain}}_{e,\ell,t}}{n_{e,\ell}}, \qquad p^{\text{aux}}_{e,\ell,t} = \frac{c^{\text{aux}}_{e,\ell,t}}{n_{e,\ell}}, \qquad \Delta_{e,\ell,t} = p^{\text{aux}}_{e,\ell,t} - p^{\text{plain}}_{e,\ell,t}. \tag{1}$$

Unaided success measures independent mastery, while assisted success measures solvability with structural guidance. Their difference estimates the aggregate assisted advantage under a common acceptance criterion. Unlike the observed problem-level recovery event in initialization, this is a stratum-level net difference, not the fraction of individual failures recovered. We use its positive part as a curriculum priority signal.

3.3 RECOVERABILITY-GUIDED CURRICULUM ALLOCATION

In each round, we apportion a verified-example quota B across environments and their structural strata. This controls accepted training-set size, rather than generation attempts or compute.

One priority signal. Define

$$r_{e,\ell,t} = [\Delta_{e,\ell,t}]_+, \qquad \bar{r}_{e,t} = \frac{1}{L_e}\sum_{\ell=1}^{L_e} r_{e,\ell,t}, \qquad Z_t = \sum_{e\in\mathcal{E}} \bar{r}_{e,t}, \tag{2}$$

where $[z]_+ = \max(0, z)$. Large positive gaps prioritize regions with additional verified success under guidance. When both conditions fail, or unaided performance catches up with assisted performance, this priority decreases. Averaging over strata prevents an environment from receiving more weight merely because it contains more equally recoverable strata.

Joint allocation with exploration. For $Z_t > 0$, set

$$q_t(e,\ell) = (1-\epsilon)\frac{r_{e,\ell,t}/L_e}{Z_t} + \frac{\epsilon}{|\mathcal{E}|L_e} \tag{3}$$

with exploration fraction $0 < \epsilon < 1$, set to 0.20 in all reported recursive rounds. The first component prioritizes recoverable regions, whereas the second is uniform over environments and then over their strata. If $Z_t = 0$, the rule falls back to $q_t(e,\ell) = 1/(|\mathcal{E}|L_e)$. Thus, $\sum_e \sum_\ell q_t(e,\ell) = 1$, and strata without a positive measured gap retain exploration probability.

Two-level quota execution. Environment and conditional stratum distributions follow directly from the joint rule:

$$q_t(e) = \sum_\ell q_t(e,\ell), \qquad q_t(\ell \mid e) = \frac{q_t(e,\ell)}{q_t(e)}. \tag{4}$$

For $Z_t > 0$, $q_t(e) = (1-\epsilon)\bar{r}_{e,t}/Z_t + \epsilon/|\mathcal{E}|$. We first apportion B under $q_t(e)$, then each task quota $B_{e,t}$ under $q_t(\ell \mid e)$, using the largest-remainder rule at both steps. The resulting integer quotas satisfy $\sum_e B_{e,t} = B$ and $\sum_\ell B_{e,\ell,t} = B_{e,t}$. Quotas are frozen during collection and recomputed after each learner update.

3.4 VERIFIED SUPERVISION AND RECURSIVE UPDATES

Controlled rewriting and trajectory collection. In each round, we freeze the quotas $B_{e,\ell,t}$ and synthesize new problems from the corresponding mathematical specifications. We use the environment generator to determine each mathematical instance and have the current learner rewrite its descriptions and contextual backgrounds while preserving variables, objectives, and constraints. We withhold numerical instance parameters during rewriting and insert them afterward. For SFT, we keep a copy of the initial model π_{θ_0} fixed to generate new guided trajectories for these problems throughout all rounds. This choice is intended to limit propagation of training-induced errors into subsequent supervision.

Training without auxiliary inputs. Accepted trajectories form D_t. We condition trajectory generation on guidance but omit it from the SFT input:

$$\tau \sim \pi_{\theta_0}(\cdot \mid x, P_{a,e}), \qquad \mathcal{L}^{(t)}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(x,\tau)\sim D_t} \log \pi_\theta(\tau \mid x). \tag{5}$$

Updating π_{θ_t} on D_t yields $\pi_{\theta_{t+1}}$. In our implementation, these updates optimize LoRA parameters while the base-model weights remain frozen. In the next round, we reassess this learner, recompute allocation, and use the learner for controlled rewriting. The core formulations, diagnostic rules, reference solvers, guidance, DEV instances, and allocation hyperparameters remain fixed. The recursion lies in the dependence of the next training distribution on the updated learner: capability feedback changes the curriculum, and controlled rewriting changes its linguistic realization. The trajectory-generation policy remains fixed.

Algorithm 1 Recursive self-improvement guided by capability assessment

Input: verifiable environments $\mathcal{E}$, fixed DEV, initial learner π_{θ_0}, quota B, exploration ϵ, rounds T.
Keep a copy of π_{θ_0} for guided SFT trajectory generation.
For $t = 0, \ldots, T-1$:
  Evaluate π_{θ_t} on DEV with and without guidance using the same correctness verifier.
  Compute positive gaps and the joint distribution, then apportion and freeze quotas $B_{e,\ell,t}$.
  While quotas remain unfilled and the candidate-batch limit is not reached:
    Synthesize new instances for strata with unfilled verified-example quotas, and use π_{θ_t} for controlled rewriting.
    Sample guided trajectories from π_{θ_0}, then verify and retain up to each quota.
  If any quota remains unfilled: report the shortfall and pause before training.
  Form D_t from accepted (x, τ) pairs, omitting auxiliary guidance from inputs.
  Fine-tune π_{θ_t} on D_t to obtain $\pi_{\theta_{t+1}}$.
Output: learner π_{θ_T} after T completed updates.

4 Main Results

4.1 Experimental Setup

Models and probing data. We use SIRL-Qwen2.5-7B (Chen et al., 2025a) as the base model for our main experiments. We use this initial checkpoint to generate SFT supervision. To identify recurring modeling defects, we curate 627 probing instances from OptMATH-Train (Lu et al., 2025). These instances cover representative operations research problems, including second-order cone programming (SOCP), the traveling salesman problem (TSP), and scheduling tasks such as job-shop scheduling. The resulting diagnoses guide the construction of corresponding executable training environments. We also compare against general-purpose language models and models trained specifically for optimization modeling.

Evaluation benchmarks and metrics. We report unaided optimization modeling performance on three benchmarks: NL4Opt (Ramamonjison et al., 2023), MAMO ComplexLP (Huang et al., 2025b), and OptMATH-bench-193 (Lu et al., 2025). For MAMO ComplexLP, we use the corrected 203-instance version released in the SIRL repository, which includes revisions to problem statements and reference answers. For OptMATH-bench-193, we use the 193-instance version distributed in the same repository.[1] Further details on benchmark curation are provided by SIRL (Chen et al., 2025a) and its accompanying repository documentation. We measure pass@1 accuracy (P@1) as the percentage of problems solved correctly using one sampled attempt per problem. For each benchmark, we evaluate with seeds $42, 43, 44$ and report the mean and population standard deviation of the three unrounded run-level P@1 scores. Macro AVG is the equally weighted mean of the three reported benchmark means.

4.2 Performance Comparison

We compare SIRL-Qwen2.5-7B+Env with the base policy and other models on three benchmarks. Our model achieves mean P@1 scores of 93.47%, 73.07%, and 40.07% on NL4Opt, MAMO ComplexLP, and OptMATH-bench-193, respectively. Relative to SIRL-Qwen2.5-7B, the largest gains occur on MAMO ComplexLP and OptMATH-bench-193: mean P@1 increases from 54.52% to 73.07% and from 35.58% to 40.07%, corresponding to relative improvements of 34.02% and 12.62%, respectively. On NL4Opt, mean P@1 increases by 0.41 percentage points, from 93.06% to 93.47%. Our model obtains the highest reported accuracy on NL4Opt and MAMO ComplexLP. DeepSeek-V4.1-Flash remains the strongest reported model on OptMATH-bench-193, at 42.49%. Overall, improvements over the base policy are particularly pronounced on MAMO ComplexLP and OptMATH-bench-193.

[1] The benchmark files and correction notes are available at `https://github.com/Cardinal-Operations/SIRL/tree/main/test_data`. We use `MAMO_ComplexLP_fixed.jsonl` and `OptMATH_Bench_193.jsonl`.

Table 1: Performance comparison on three optimization modeling benchmarks. Benchmark scores are mean P@1 $\pm$ population standard deviation (%) over evaluation seeds $42, 43, 44$. Macro AVG is the equally weighted average of the three benchmark means and is reported without a standard deviation. Bold indicates the highest reported value in each column.

Types	Models	Pass@1 (%)			Macro AVG
		NL4Opt	MAMO ComplexLP	OptMATH-bench-193	
Baseline	SIRL-Qwen2.5-7B	93.06 ± 0.33	54.52 ± 1.23	35.58 ± 1.29	61.05
General LLMs	DeepSeek-V4.1-Flash	88.03 ± 0.84	69.46 ± 1.06	$\mathbf{42.49} \pm 2.24$	66.66
	Qwen3.8-Flash	84.49 ± 0.67	60.26 ± 3.35	26.25 ± 1.71	57.00
Offline learning	ORLM-LLaMA-3-8B	81.77 ± 1.02	39.90 ± 2.24	10.71 ± 2.09	44.13
	OptMATH-Qwen2.5-7B	77.96 ± 0.88	23.65 ± 0.80	26.42 ± 2.64	42.68
	OptMATH-Qwen2.5-32B	85.71 ± 0.58	53.20 ± 0.80	35.41 ± 1.29	58.11
Iterative DPO	StepORLM-Qwen3-8B	93.06 ± 1.00	55.50 ± 1.63	13.82 ± 0.24	54.13
Ours	SIRL-Qwen2.5-7B+Env	$\mathbf{93.47} \pm 0.33$	$\mathbf{73.07} \pm 2.59$	40.07 ± 0.49	**68.87**

Model sources: SIRL-Qwen2.5-7B (Chen et al., 2025a); DeepSeek-V4.1-Flash (DeepSeek-AI, 2026); Qwen3.8-Flash (Qwen Team, 2026); ORLM-LLaMA-3-8B (Huang et al., 2024); OptMATH-Qwen2.5-7B and OptMATH-Qwen2.5-32B (Lu et al., 2025); and StepORLM-Qwen3-8B (Zhou et al., 2025).

4.3 Round-wise Recursive Improvement

We examine how model capability and curriculum composition evolve from R_0 to R_{15} under the capability assessment and allocation mechanism. We combine measurements at six checkpoints with paired evaluations on the same 204-instance DEV set. State curves use execution success, whereas highest-stratum assisted pass and allocation use joint execution–structure acceptance.



(a) Checkpoint measurements

Round	Assisted pass (%)	Mean difficulty	Benchmark acc. (%)
R_0	64.58	—	45.05
R_3	85.42	0.487	50.29
R_6	81.25	0.458	52.10
R_9	**91.67**	0.438	56.20
R_{12}	83.33	0.478	55.87
R_{15}	87.50	0.482	**56.57**

Assisted pass: joint execution–structure acceptance on 48 highest-stratum DEV instances. Mean difficulty: normalized difficulty of the newly generated training data used to obtain R_t. Benchmark accuracy: mean over MAMO ComplexLP and OptMATH-bench-193, with R_0 using Table 1's baseline.

(b) From assisted to independent success

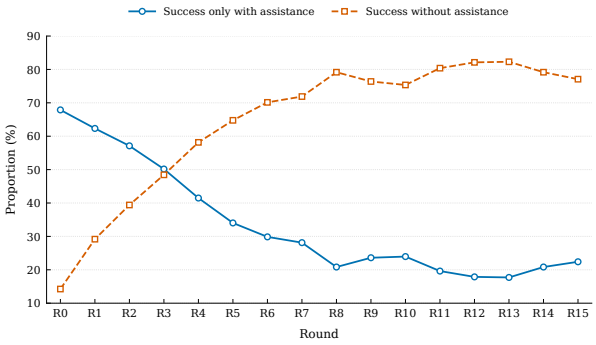


Figure 3: **Round-wise recursive improvement.** Right: paired execution-only outcomes on the same 204 DEV instances, macro-averaged over four environments. Assistance-only means unaided failure and assisted success on the same instance. Left: highest-stratum joint acceptance, mean difficulty of newly generated training data, and three-seed benchmark means. Mean difficulty weights newly generated training examples within each environment, then environments equally.

From prompt-recoverable failures to independent success. Unaided success rises from 14.24% at R_0 to 29.17%, 39.41%, and 48.44% at R_1, R_2, and R_3, respectively. At R_4, unaided success reaches 58.16%, overtaking assistance-only success at 41.49%. By R_{15}, unaided success is 77.08%, assistance-only success has fallen from 67.88% to 22.40%, and both-fail outcomes have fallen from

17.88% to 0.52%. Improvement is not monotonic: unaided success peaks at 82.29% at R_{13} before declining over the final two rounds.

Curriculum allocation tracks evolving capability. Assisted success at the highest predefined stratum rises from 64.58% at R_0 to 87.50% at R_{15}, with a peak of 91.67% at R_9. The mean difficulty of newly generated training data decreases from 0.487 at R_3 to 0.458 at R_6 and 0.438 at R_9, then increases to 0.478 at R_{12} and 0.482 at R_{15}. Under the capability assessment mechanism, repeated reassessment updates allocation as the learner changes.

Connection to downstream performance. The two-benchmark unassisted mean increases from 45.05% at R_0 to 56.57% at R_{15}, a gain of 11.52 percentage points. It reaches 50.29% at R_3, 52.10% at R_6, and 56.20% at R_9, followed by a small decrease to 55.87% at R_{12} and a recovery at R_{15}. Together, the diagnostic and benchmark trends show improved independent problem solving over the recursive run using the assessment and allocation mechanism.

4.4 Effect of Capability-Based Allocation

We compare our capability-based allocation with uniform allocation, which assigns equal budgets to the four environments and uniform weights to strata within each environment. Both variants complete 15 rounds with 128 verified examples per round, with all other generation, training, and evaluation settings unchanged.

Table 2: Allocation ablation at R15. Scores are mean unaided P@1 $\pm$ standard deviation (%) over three evaluation seeds. Red arrows indicate gains over uniform allocation in percentage points, while Macro AVG averages the three benchmark means.

Allocation	NL4Opt	MAMO ComplexLP	OptMATH-bench-193	Macro AVG
Uniform	93.33 ± 0.51	68.97 ± 1.21	39.38 ± 1.12	67.23
Our mechanism	$\mathbf{93.47} \pm 0.33$ (↑ 0.14)	$\mathbf{73.07} \pm 2.59$ (↑ 4.10)	$\mathbf{40.07} \pm 0.49$ (↑ 0.69)	**68.87** (↑ 1.64)

We observe higher mean accuracy on all three benchmarks , with the largest gain on MAMO ComplexLP and a 1.64-point gain in Macro AVG. This comparison supports using capability assessments to direct a fixed training budget toward recoverable deficiencies.

4.5 Effect of Structural Assistance

We examine whether structural assistance improves the collection efficiency of verified self-generated trajectories on difficult problems. We hold the model, problems, and joint-verification rules fixed and vary only the provision of structural hints. Each condition uses 200 problems per task and 3 samples per problem, totaling 2,400 attempts across four tasks. We measure collection efficiency as the fraction of trajectories passing joint verification.

Table 3: Effect of structural assistance on the yield of verified self-generated trajectories under a fixed sampling budget.

Task type	Unaided pass rate (%)	Assisted pass rate (%)	Gain (pp)	Rate ratio
Job-shop	8.50	83.00	+74.50	9.76×
Inventory transshipment	17.33	64.17	+46.83	3.70×
SOCP	22.00	79.67	+57.67	3.62×
TSP	1.50	93.50	+92.00	62.33×
Overall	12.33	80.08	+67.75	6.49×

We observe improved verification pass rates across all four tasks in Table 3. The overall rate rises from 12.33% to 80.08%, yielding 6.49 times as many usable trajectories for the same number of sampling attempts. On TSP, the pass rate increases from 1.50% to 93.50%. We thus use structural assistance to obtain substantially more verified self-generated supervision from difficult problems under a fixed sampling budget.

4.6 Effect of Trajectory Generator Updates

We compare generating guided trajectories with the fixed initial model against using the updated learner from the preceding round. Both variants synthesize fresh problems and trajectories at every round, use 128 verified examples per update, and share the same SFT settings. Each variant adapts its allocation using its own DEV assessments.

Table 4: Trajectory-generator ablation. Equal-weight mean unaided accuracy (%) on MAMO ComplexLP and OptMATH-193. $R0^*$ is the shared base checkpoint. Both variants generate fresh supervision each round.

Round	Fixed initial-model trajectories	Updated-model trajectories
$R0^*$	45.05	45.05
R1	49.27	49.11
R2	50.82	50.92
R3	50.29	46.43
R4	50.54	38.59
R5	50.45	31.17

We observe similar gains through R2 (50.82% versus 50.92%), followed by diverging performance: the fixed-generator variant retains 50.45% accuracy at R5, whereas the updated-generator variant declines from R3, falls below the base model by R4, and reaches 31.17% at R5. This decline coincides with changes in supervision: completion-token counts are 18.9–22.4% lower in R3–R5, and Job-shop accounts for 75.8% of R5 training examples, compared with 51.6% under the fixed generator. These observations are consistent with feedback between evolving supervision and adaptive task allocation reinforcing specialization at the expense of broader benchmark performance. The comparison supports retaining the initial model as the trajectory generator within our recursive training loop.

5 Conclusion

We presented a framework for recursive self-improvement through defect-driven verifiable environments. We construct environments with a strong external model under human review and use prompt recoverability to connect defect diagnosis, adaptive difficulty allocation, and assisted trajectory generation. We assess updated models to obtain capability feedback for data allocation and use them for controlled rewriting of environment-generated problems, allowing the curriculum to track their changing capabilities. We use verified guided trajectories to train without auxiliary guidance in the inputs. In experiments on three optimization modeling benchmarks, we observe improved average accuracy, with the largest gains on MAMO ComplexLP and OptMATH-bench-193 relative to the base model. With structural assistance, we further increase verified trajectory yield to 6.49 times that of unaided generation under the same sampling budget, making difficult problems a more productive source of training data.

Future Work. A further direction is to extend this feedback loop by transferring defect diagnosis and environment construction from the external model to the learner itself. The learner would reflect on its own errors, build or refine executable environments, generate training data validated by programmatic verifiers, and use it for further training. This extension would aim to jointly improve reflection, problem generation, and problem solving, so that each model update also strengthens the processes that diagnose weaknesses and construct subsequent learning tasks. This would extend the scope of self-improvement from adaptive data construction within reviewed environments to the creation and refinement of the environments themselves.

AI USE STATEMENT

In addition to the base model used in our training framework, we used GPT-5.6 to analyze solution trajectories, identify the base model's core modeling defects, and construct structured executable environments targeting these defects. We validated every component of the constructed environments using comprehensive test cases. The base model's role in generating training trajectories and the updated learner's role in controlled problem rewriting are described in Section 3.4.

We also used AI tools to assist with code implementation, and tested each component with test cases to check its correctness. For manuscript preparation, we used AI tools to assist with translation and to check selected passages for grammatical accuracy and consistency with our intended meaning. We reviewed the AI-assisted outputs and take responsibility for the final text, claims, code, and other artifacts in this work.

REPRODUCIBILITY STATEMENT

We have submitted the code materials needed to reproduce the corresponding experiments in the supplementary ZIP archive. Experimental settings and evaluation protocols are described in Section 4.1. Additional details on initial diagnosis, structural-assistance prompts, capability assessment, data allocation, generation controls, and training configurations are provided in Appendices A, B, and C.

REFERENCES

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed RL: Grounding large language models for authentic optimization modeling, 2025a. URL https://arxiv.org/abs/2505.11792. arXiv preprint arXiv:2505.11792.

Yue Chen, Minghua He, Fangkai Yang, Pu Zhao, Lu Wang, Yu Kang, Yifei Dong, Yuefeng Zhan, Hao Sun, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. WarriorMath: Enhancing the Mathematical Ability of Large Language Models with a Defect-aware Framework, 2025b. URL https://arxiv.org/abs/2508.01245.

DeepSeek-AI. Introducing DeepSeek-V4.1-Flash: Smarter, faster, more efficient. Official model release, 2026. URL https://www.deepseek.com/en/news/deepseek-v4-1-flash/.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-acl.507. URL https://aclanthology.org/2023.findings-acl.507/.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-Zero: Self-Evolving Reasoning LLM from Zero Data, 2025a. URL https://arxiv.org/abs/2508.05004.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A Customizable Framework in Training Large Models for Automated Optimization Modeling, 2024. URL https://arxiv.org/abs/2405.17743.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710. Association for Computational Linguistics, 2025b. doi: 10.18653/v1/2025.findings-naacl.146. URL https://aclanthology.org/2025.findings-naacl.146/.

Qingcan Kang, Mingyang Liu, Xiaojin Fu, Shixiong Kai, Tao Zhong, and Mingxuan Yuan. EvoOptiGraph: Weakness-Driven Coevolution via Graph-Based Structural Generation for Optimization Modeling, 2026. URL https://arxiv.org/abs/2606.26578.

Nicholas Lee, Thanakul Wattanawong, Sehoon Kim, Karttikeya Mangalam, Sheng Shen, Gopala Anumanchipalli, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. LLM2LLM: Boosting LLMs with Novel Iterative Data Enhancement, 2024. URL https://arxiv.org/abs/2403.15042.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A Scalable Bidirectional Data Synthesis Framework for Optimization Modeling, 2025. URL https://arxiv.org/abs/2502.11102.

Sumeet Ramesh Motwani, Chuan Du, Aleksander Petrov, Christopher Davis, Philip Torr, Antonio Papania-Davis, and Weishi Yan. AutoOR: Scalably Post-training LLMs to Autoformulate Operations Research Problems, 2026. URL https://arxiv.org/abs/2604.16804.

Qwen Team. Qwen3.8-Flash. Qwen Cloud official model documentation, 2026. URL https://www.qwencloud.com/models/qwen3.8-flash. Accessed September 20, 2026.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203. PMLR, 2023. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

Yucheng Shi, Zhenwen Liang, Kishan Panaganti, Dian Yu, Wenhao Yu, and Haitao Mi. Learning to Build the Environment: Self-Evolving Reasoning RL via Verifiable Environment Synthesis, 2026. URL https://arxiv.org/abs/2605.14392.

Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond Human Data: Scaling Self-Training for Problem-Solving with Language Models, 2023. URL https://arxiv.org/abs/2312.06585.

Shaobo Wang, Zhengbo Jiao, Zifan Zhang, Yilang Peng, Xu Ze, Boyu Yang, Wei Wang, Hu Wei, and Linfeng Zhang. Socratic-Zero : Bootstrapping Reasoning via Data-Free Agent Co-evolution, 2025. URL https://arxiv.org/abs/2509.24726.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping Reasoning With Reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, Chenyang Zhao, Yulia Tsvetkov, Simon Shaolei Du, Natasha Jaques, Hao Peng, Pang Wei Koh, and Hannaneh Hajishirzi. RLVE: Scaling Up Reinforcement Learning for Language Models with Adaptive Verifiable Environments, 2025. URL https://arxiv.org/abs/2511.07317.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute Zero: Reinforced Self-play Reasoning with Zero Data, 2025. URL https://arxiv.org/abs/2505.03335.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models, 2025. URL https://arxiv.org/abs/2509.22558. arXiv preprint arXiv:2509.22558.

Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-Adapting Language Models, 2025. URL https://arxiv.org/abs/2506.10943.

A Initial Diagnosis of Structurally Recoverable Failures

This precursor experiment motivates defect-driven environment construction; it is separate from the round-wise capability assessment and the training-trajectory yield ablation. We fix SIRL-Qwen2.5-7B (Chen et al., 2025a) and screen a capability-probing set $\mathcal{D}_{\text{probe}}$ disjoint from the final evaluation instances. With screening-budget hyperparameter $K_{\text{probe}} = 6$, the observed zero-success set is

$$\mathcal{D}_0 = \left\{ x \in \mathcal{D}_{\text{probe}} : \sum_{k=1}^{K_{\text{probe}}} V(x, \tau_k) = 0 \right\}, \qquad \tau_k \sim \pi_{\theta_0}(\cdot \mid x), \tag{6}$$

where V is a common binary solution-correctness verifier. This yields 513 candidate problems across four tasks.

Controlled recovery. Shared errors in failed trajectories guide candidate-defect identification and hint construction. Each candidate receives $K_{\text{recover}} = 3$ *new* samples under each of four conditions: unaided resampling, generic prompting, mismatched structural assistance, and matched structural assistance. Matched hints address the implicated relations or operations without supplying final answers. Recovery P@3 counts problems with at least one solution-correct trajectory among the three new samples; the initial six samples are excluded.

Table 5: Initial solution recovery P@3 (%). Aggregates weight tasks by their candidate counts. All four conditions have the same additional sampling budget.

Task	N	Unaided resampling	Generic prompt	Mismatched assistance	Matched assistance
Job-shop	116	1.72	0.00	1.72	**96.55**
Inventory transshipment	75	9.33	16.00	16.00	**100.00**
SOCP	163	0.00	0.00	0.00	**84.05**
TSP	159	2.52	1.89	22.64	**83.65**
Weighted aggregate	513	2.53	2.92	9.75	**89.08**

Matched assistance recovers $457/513$ problems (89.08%), compared with 9.75% for the strongest aggregate control. The matched condition leads in every task, supporting targeted recoverability beyond additional resampling or untargeted guidance under this protocol.

Correction of the target structure. Let $S_d(x, \tau)$ indicate passage of the structural check for defect d, and define joint acceptance $J_d(x, \tau) = V(x, \tau) S_d(x, \tau)$. Joint recovery@3 requires at least one of the three assisted trajectories to pass *both* checks; successes on different trajectories do not suffice. Defect coverage uses the entire task-specific candidate set as its denominator, whereas joint recovery uses the corresponding defect subset.

Table 6: Representative defects and their recovery under matched assistance. Both numerical columns are percentages, with different denominators as defined above.

Task	Target structural error	Coverage	Joint recovery@3
Job-shop	Incomplete encoding of the two possible same-machine processing orders and associated disjunctive constraints.	92.24	96.26
Inventory transshipment	Inventory lower bound $I_t \geq L_t$ encoded as equality $I_t = L_t$.	74.67	100.00
SOCP	Incorrect handling of the cone-threshold domain.	96.32	85.35
TSP	Local degree constraints permit multiple subtours instead of a single global tour.	88.05	95.00

The recurring defects and joint recovery provide behavioral evidence for organizing environments around shared structural operations.

B Structural Assistance Prompt Templates

We reproduce the four task-specific structural-assistance templates below, preserving their wording and placeholder syntax; line wrapping is adjusted for readability. They provide modeling rela-

tions and implementation guidance without supplying an instance's optimal objective or solution. The TSP template includes an explicit MTZ construction and control-flow scaffold. The displayed SOCP prompt is specific to the explicitly nonnegative squared variant; guidance is routed by instance semantics. These templates document task-specific assistance.

B.1 JOB-SHOP SCHEDULING

```
Private structural guidance (do not quote this guidance in the response):

- This instance contains {job_count} jobs and {operation_count}
operations.
  Its canonical structure contains {precedence_count} consecutive
  precedence relations and {conflict_pair_count} unordered same-machine
  conflict pairs.

- Identify each operation by one globally unique key:
  (job_id, operation_position). An operation position alone is not
  unique.

- Use the same operation key consistently for processing times,
start-time variables, precedence relations, machine-conflict relations,
and completion bounds. Create variables only for operations that actually
appear in the problem, not for a rectangular job-by-position Cartesian
product.

- Preserve every stated within-job precedence relation. For each
same-machine conflict pair, model the two alternative processing orders
so that the operations cannot overlap.

- Before solving, verify that there is one start-time variable and one
makespan completion bound per operation. Check that all
{precedence_count} precedence relations and all {conflict_pair_count}
conflict pairs are represented. Every referenced operation must exist and
use the same key structure at variable creation and lookup.

- Minimize the stated makespan and provide a complete executable model.
  Derive the instance data and relation endpoints from the question
  yourself.
```

B.2 INVENTORY TRANSSHIPMENT

```
Answer-free relation ledger for minimum-inventory lower bounds and
permitted surplus:
- Treat every stated need as a lower bound on final inventory; unused
surplus may stay put.
- For each location i use: current[i] + sum_j flow[j,i] - sum_j flow[i,j]
>= minimum[i].
- A cost in row i and destination j belongs to the ordered arc (i,j), not
to (j,i).
- Create a nonnegative transfer variable for every ordered pair of
distinct locations.
- Intermediate routing is allowed and each used directed leg contributes
its own unit cost.
- Per-location current-minus-minimum ledger (positive means available
slack):
  location {location_id}: {current_inventory} - {minimum_inventory} =
  {net_surplus}
This ledger contains no optimum, transfer plan, objective value, or
reference implementation.
```

B.3 SECOND-ORDER CONE PROGRAMMING

The prompt below applies to `explicit_t_nonnegative_squared`, whose statement explicitly specifies $t \geq 0$. For free-threshold squared variants, the routed guidance identifies the literal nonconvex double cone and selects its $t \geq 0$ branch according to the statement's intended stan-

dard convex SOCP semantics. Final acceptance requires $t \geq 0$; the environment targets semantic reconciliation and recovery of the intended convex formulation.

```
Private modeling checklist (do not quote it in the final response):
1. Reconcile variable-domain prose with the displayed cone relation
before writing code.
2. Distinguish a convex Lorentz cone from the two-sheet set produced by
squaring a free threshold.
3. When prose and a displayed inequality conflict, identify which
interpretation is consistent with the stated standard convex SOCP.
4. Do not invent affine links that the complete statement does not
provide.
5. Preserve the model's own natural response and coding style.

Private semantic ledger (no optimum, solution, or code is supplied):
- The threshold domain is already nonnegative, so the displayed squared
inequality is a valid Lorentz-cone encoding.
- Canonical compilation baseline: declare every primary x and vector
auxiliary y unrestricted, declare every scalar threshold t nonnegative,
transcribe every printed affine equality literally, create every printed
component exactly once, and add one correctly grouped cone constraint per
block.
- Additional modeling focus: Audit variable domains, affine signs,
component coverage, block grouping, objective sense, and result
extraction together.
- State the decisions briefly in <think>/<model>, then solve the given
coefficients yourself.
```

B.4 TRAVELING SALESMAN PROBLEM

```
[Private answer-free TSP relation assistance; do not mention this
assistance in the response.]
The selected travel arcs must form one Hamiltonian cycle containing every
city. One incoming and one outgoing arc per city are necessary but are
not sufficient: those degree equations alone allow two or more
disconnected subtours. Explicitly add a global subtour-elimination
mechanism and audit that all cities belong to the same closed route. The
instance has {{city_count}} cities.
Do not use or guess an optimal value or an optimal tour; derive them only
by solving the model.
Use an answer-free MTZ relationship ledger. Choose the first listed city
as a depot. Create binary x[i,j] for every ordered pair i != j. Add
exactly one outgoing and one incoming equation for every city. For every
non-depot city create an order variable u[i] with bounds 1 through n-1.
For every ordered pair of distinct non-depot cities add u[i] - u[j] +
n*x[i,j] <= n-1. Minimize the complete arc-cost sum and optimize the same
model object. These MTZ rows are the relation family that turns the
degree-only cycle cover into one global tour.
Use this control-flow scaffold without copying any optimum: define cities
and the complete cost dictionary from the question; create x for all i !=
j; create u only for cities[1:]; add the two degree loops; add the MTZ
nested loop only over cities[1:] with i != j; set the minimization
objective; optimize; and print `Just print the best solution:` followed
by model.ObjVal.
```

B.5 TEMPLATE PARAMETERS

Placeholders encode instance metadata rather than reference solutions. The inventory ledger has one entry per location. Single and double braces below reproduce the respective template syntax.

Table 7: Parameters used in the structural-assistance templates.

Placeholder	Meaning
{job_count}	Number of jobs.
{operation_count}	Total number of operations.
{precedence_count}	Number of precedence relations between consecutive operations within jobs.
{conflict_pair_count}	Number of unordered pairs of operations using the same machine.
{location_id}	Location identifier.
{current_inventory}	Current inventory at the location.
{minimum_inventory}	Minimum required final inventory at the location.
{net_surplus}	Current inventory minus the minimum requirement.
{{city_count}}	Number of cities in the TSP instance.

C Capability Assessment and Data Allocation

C.1 Fixed assessment data and shared verification

We use a fixed DEV set of 204 mathematical instances for capability assessment in 17 equally populated environment–stratum cells: Job-shop D0–D5 (72), inventory D0–D3 (48), SOCP D1–D4 (48), and TSP N4/N5/N6 (36). Each cell contains 12 instances. An assessment uses one unaided and one assisted attempt per instance, producing 408 responses. Both conditions use the same execution–structure correctness verifier; only defect-specific guidance differs. Instances, guidance templates, sampling seed, and attempts per condition remain fixed across rounds. Verified-success counts determine the empirical rates and assisted–unassisted gaps in Eq. 1.

Table 8: Configuration of the capability assessment and allocation mechanism.

Parameter	Role	Value
B	Verified examples per update	128
m	DEV attempts per instance in each condition	1
σ_{DEV}	Fixed assessment sampling seed	42
ϵ	Uniform exploration fraction	0.20
DEV instances	Fixed mathematical instances across all strata	204
Instances per cell	Fixed instances per environment–stratum cell	12

C.2 Definitions of round-wise measurements

Evaluation populations and checkpoint alignment. Figure 3 reports the recoverability-guided run from R_0 to R_{15}. All checkpoints use the same 204 DEV instance IDs, with one unaided and one assisted response per instance and sampling seed 42, giving 408 responses per checkpoint. R_0 reuses the initial model's recorded DEV evaluation. The full DEV set supplies the state proportions; highest-stratum assisted pass uses only 48 instances, 12 from each environment's highest predefined stratum. Downstream evaluation instead uses MAMO ComplexLP (203 instances) and OptMATH-bench-193 (193 instances). DEV informs allocation, so its curves describe development-set behavior rather than independent generalization.

Paired execution states and environment macro-averaging. Let $u_t(x), a_t(x) \in \{0, 1\}$ indicate execution success for the unaided and assisted responses to the same DEV instance x. These indicators use raw_accepted, without the additional structural acceptance test. Define three mutually exclusive, exhaustive states:

$$\begin{aligned} h_{\text{both fail},t}(x) &= (1 - u_t(x))(1 - a_t(x)), \\ h_{\text{assistance-only},t}(x) &= (1 - u_t(x))a_t(x), \\ h_{\text{unaided success},t}(x) &= u_t(x). \end{aligned} \tag{7}$$

Unaided success includes instances whose assisted response fails. For state c, the percentage is

$$P_c(t) = \frac{100}{4} \sum_{e \in \mathcal{E}} \frac{1}{|D_e|} \sum_{x \in D_e} h_{c,t}(x). \tag{8}$$

Thus each environment has equal weight despite different DEV sizes. With 12 instances per stratum, averaging within an environment also equals an unweighted average over its strata. The right

panel plots assistance-only and unaided success; both-fail is the remaining percentage. Assistance-only is not assisted accuracy minus unaided accuracy: that difference also subtracts unaided-success/assisted-failure cases. In addition, the allocation gaps in Eq. 1 use joint execution–structure acceptance, not these execution-only states.

Table 9 gives the reported percentages used to plot Figure 3.

Table 9: Exact reported percentages underlying the round-wise state curves. States use paired execution-only outcomes on the fixed DEV set and equal weighting across environments. Entries are rounded to two decimal places, so a row may sum to 100.01%.

Round	Both fail	Assistance-only	Unaided success	Round	Both fail	Assistance-only	Unaided success
R_0	17.88	67.88	14.24	R_8	0.00	20.83	79.17
R_1	8.51	62.33	29.17	R_9	0.00	23.61	76.39
R_2	3.47	57.12	39.41	R_{10}	0.69	23.96	75.35
R_3	1.39	50.17	48.44	R_{11}	0.00	19.62	80.38
R_4	0.35	41.49	58.16	R_{12}	0.00	17.88	82.12
R_5	1.22	34.03	64.76	R_{13}	0.00	17.71	82.29
R_6	0.00	29.86	70.14	R_{14}	0.00	20.83	79.17
R_7	0.00	28.13	71.88	R_{15}	0.52	22.40	77.08

Highest-stratum assisted pass and benchmark accuracy. Let $j_t^{\mathrm{aux}}(x)$ indicate joint execution–structure acceptance (`accepted`) of the assisted response. For the highest-stratum DEV subset $D_{e,\mathrm{top}}$,

$$A_{\mathrm{top}}(t) = \frac{100}{4} \sum_{e \in \mathcal{E}} \frac{1}{12} \sum_{x \in D_{e,\mathrm{top}}} j_t^{\mathrm{aux}}(x). \tag{9}$$

This statistic differs from the state curves in both its population and acceptance rule. Benchmark accuracy is the equally weighted mean of the two benchmarks' three-seed unaided pass@1 means. Its R_0 value uses the main-table baseline: $(54.52 + 35.58)/2 = 45.05\%$.

Mean difficulty of newly generated training data. For each environment, assign its predefined generation strata normalized difficulty levels $d_{e,\ell} = (\ell - 1)/(L_e - 1)$, where $\ell = 1, \ldots, L_e$ follows their listed order. In particular, SOCP D1 and TSP N4 map to zero; numerical labels are not divided by their maximum.

Table 10: DEV composition and normalized generation levels. Every stratum has 12 DEV instances; Top denotes the last predefined stratum.

Environment	L_e	DEV	Top	Normalized levels (in stratum order)
Job-shop D0–D5	6	72	D5	$0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1$
Inventory D0–D3	4	48	D3	$0, \frac{1}{3}, \frac{2}{3}, 1$
SOCP D1–D4	4	48	D4	$0, \frac{1}{3}, \frac{2}{3}, 1$
TSP N4–N6	3	36	N6	$0, \frac{1}{2}, 1$

Let $\mathcal{T}_t$ contain the newly synthesized and verified examples used in round t to obtain checkpoint π_{θ_t}, with $n^{\mathrm{tr}}_{e,\ell,t}$ examples from each environment–stratum cell and $n^{\mathrm{tr}}_{e,t} = \sum_\ell n^{\mathrm{tr}}_{e,\ell,t} > 0$. Define

$$\bar{d}_{e,t} = \frac{\sum_{\ell=1}^{L_e} n^{\mathrm{tr}}_{e,\ell,t} d_{e,\ell}}{n^{\mathrm{tr}}_{e,t}}, \qquad \bar{d}_t = \frac{1}{4} \sum_{e \in \mathcal{E}} \bar{d}_{e,t}. \tag{10}$$

Mean difficulty weights the new training examples within each environment, then weights environments equally. It is computed separately for each newly generated batch, using environment macro-averaging rather than a single sample-weighted average across all 128 examples. In the zero-based update notation of Section 3.4, $\mathcal{T}_t = D_{t-1}$:

$$\pi_{\theta_{t-1}} \xrightarrow{\text{assessment and synthesis}} \mathcal{T}_t \xrightarrow{\mathrm{SFT}} \pi_{\theta_t}. \tag{11}$$

The R_t row pairs $\bar{d}_t$ with post-training evaluations of π_{θ_t}. R_0 precedes the first round of data synthesis and training, so mean difficulty is shown as a dash.

The reported mean difficulties are $\bar{d}_3 = 0.487$, $\bar{d}_6 = 0.458$, $\bar{d}_9 = 0.438$, $\bar{d}_{12} = 0.478$, and $\bar{d}_{15} = 0.482$, rounded to three decimals. Each round uses newly synthesized training data. Changes in $\bar{d}_t$ describe the normalized difficulty mix of each round's new training data, not a calibrated change in absolute task difficulty.

C.3 Capability-guided allocation and integer quotas

The positive gaps in Eq. 2 determine allocation priorities. Equation 3 combines these priorities with uniform exploration over environments and their structural strata. The reference configuration is $B = 128$ and $\epsilon = 0.20$. Hyperparameters remain fixed across rounds, while capability estimates and quotas are recomputed after each learner update. The exploration fraction is the allocation tuning parameter; success rates are computed directly from verified-success counts.

For an integer total N and normalized weights q_i, assign $\lfloor Nq_i \rfloor$ first, then distribute the remaining units in descending order of fractional remainders, breaking ties by a fixed environment or stratum order. Apply this procedure to $(B, q_t(e))$, then to $(B_{e,t}, q_t(\ell \mid e))$. If all measured gaps are nonpositive, the underlying distribution is uniform over environments and, conditionally, over strata. After apportionment, collection serves only unfilled positive quotas, following Appendix C.6.

C.4 Round-wise assessment signals and realized training counts

Table 11 links each checkpoint's assessment to the newly generated training data used in the following update. Each entry reports unaided pass, assisted pass, the mean positive stratum gap, and the actual number of verified examples used to train the next checkpoint. Both pass rates use joint execution–structure acceptance. For condition $c \in \{\mathrm{plain}, \mathrm{aux}\}$, the environment-level rate is $\bar{p}^{c}_{e,t} = L_e^{-1} \sum_\ell p^{c}_{e,\ell,t}$. The third quantity is $\bar{r}_{e,t} = L_e^{-1} \sum_\ell [p^{\mathrm{aux}}_{e,\ell,t} - p^{\mathrm{plain}}_{e,\ell,t}]_+$, with the positive part taken before averaging. It therefore need not equal the difference between the first two quantities.

Table 11: Assessment signals and actual next-round training counts. Each cell is unaided pass (%) / assisted pass (%) / mean positive stratum gap (%) / training examples. The last quantity is the realized count of newly generated, verified examples used in the next update, not an expected fractional quota.

Assessment → training	Job-shop	Inventory	SOCP	TSP
$R_0 \to R_1$	2.78 / 91.67 / 88.89 / 42	2.08 / 52.08 / 50.00 / 27	52.08 / 58.33 / 16.67 / 13	0.00 / 97.22 / 97.22 / 46
$R_1 \to R_2$	1.39 / 100.00 / 98.61 / 50	0.00 / 77.08 / 77.08 / 40	62.50 / 58.33 / 12.50 / 12	52.78 / 97.22 / 44.44 / 26
$R_2 \to R_3$	5.56 / 97.22 / 91.67 / 51	6.25 / 93.75 / 87.50 / 49	62.50 / 72.92 / 16.67 / 15	83.33 / 97.22 / 13.89 / 13
$R_3 \to R_4$	8.33 / 98.61 / 90.28 / 56	31.25 / 100.00 / 68.75 / 45	62.50 / 77.08 / 16.67 / 16	91.67 / 100.00 / 8.33 / 11
$R_4 \to R_5$	12.50 / 98.61 / 86.11 / 66	66.67 / 97.92 / 31.25 / 28	64.58 / 85.42 / 25.00 / 24	88.89 / 94.44 / 5.56 / 10
$R_5 \to R_6$	13.89 / 100.00 / 86.11 / 78	83.33 / 97.92 / 14.58 / 19	64.58 / 83.33 / 22.92 / 25	97.22 / 58.33 / 0.00 / 6
$R_6 \to R_7$	27.78 / 100.00 / 72.22 / 79	91.67 / 97.92 / 6.25 / 13	66.67 / 83.33 / 22.92 / 30	94.44 / 75.00 / 0.00 / 6
$R_7 \to R_8$	19.44 / 100.00 / 80.56 / 85	85.42 / 100.00 / 14.58 / 21	85.42 / 95.83 / 10.42 / 16	97.22 / 94.44 / 0.00 / 6
$R_8 \to R_9$	36.11 / 100.00 / 63.89 / 97	97.92 / 97.92 / 0.00 / 7	85.42 / 89.58 / 8.33 / 18	97.22 / 91.67 / 0.00 / 6
$R_9 \to R_{10}$	41.67 / 100.00 / 58.33 / 76	83.33 / 100.00 / 16.67 / 26	83.33 / 91.67 / 10.42 / 19	97.22 / 91.67 / 0.00 / 7
$R_{10} \to R_{11}$	23.61 / 100.00 / 76.39 / 94	93.75 / 100.00 / 6.25 / 14	89.58 / 95.83 / 6.25 / 14	94.44 / 66.67 / 0.00 / 6
$R_{11} \to R_{12}$	41.67 / 100.00 / 58.33 / 92	100.00 / 100.00 / 0.00 / 6	93.75 / 93.75 / 6.25 / 16	86.11 / 80.56 / 5.56 / 14
$R_{12} \to R_{13}$	40.28 / 100.00 / 59.72 / 95	93.75 / 100.00 / 6.25 / 16	100.00 / 85.42 / 0.00 / 6	94.44 / 94.44 / 2.78 / 11
$R_{13} \to R_{14}$	43.06 / 100.00 / 56.94 / 95	100.00 / 100.00 / 0.00 / 6	91.67 / 87.50 / 6.25 / 16	94.44 / 94.44 / 2.78 / 11
$R_{14} \to R_{15}$	43.06 / 100.00 / 56.94 / 80	89.58 / 97.92 / 8.33 / 17	97.92 / 81.25 / 0.00 / 7	86.11 / 100.00 / 13.89 / 24

Every row totals 128 training examples. The realized environment counts match the allocation rule with $\epsilon = 0.20$ and largest-remainder apportionment. As TSP's mean positive gap falls from 97.22% at R_0 to 8.33% at R_3, its next-round count falls from 46 to 11. By contrast, Job-shop retains a positive gap of 90.28% at R_3 and receives 56 examples in R_4. Zero-gap environments can still receive examples through exploration. These records show the realized response of data allocation to assessment feedback.

C.5 Structural strata

Table 12 lists preset counts, probabilities, and gap bounds, which are configurable generation hyperparameters.

Table 12: Structural generation presets. Inventory probabilities precede defect-conditioned overrides.

Task	Stratum	Preset
Job-shop	D0	2 jobs, 2 operations/job; minimal complete structure.
	D1	3 jobs, 2 operations/job.
	D2	3–4 jobs, 2–3 operations/job.
	D3	2–4 jobs, 3–7 operations/job; deep chains, non-unit durations.
	D4	4–7 jobs, 2 operations/job; unit durations, wide structures, hotspot machines.
	D5	4–7 jobs, 3–7 operations/job; wide, deep, and mixed extrapolation.
Inventory	D0	4 nodes; 1–2 shortage nodes; base multihop probability 40%.
	D1	5 nodes; 2–3 shortage nodes; 33%.
	D2	6 nodes; 1–4 shortage nodes; 67%.
	D3	7 nodes; 2–3 shortage nodes; 80%.
SOCP	D1	2 variables, 0 equalities, 2 cone constraints.
	D2	3 variables, 1 equality, 3 cone constraints.
	D3	4 variables, 1 equality, 3 cone constraints.
	D4	5 variables, 2 equalities, 4 cone constraints.
TSP	N4/N5/N6	4/5/6 cities; symmetric closed-tour instances exposing subtours.

Within-stratum variation. Job-shop difficulty also depends on conflict topology, machine revisits, durations, and defects; defect construction can change the final machine count. D3 and D4 are different structural modes: realized DEV operation counts are 9–23 and 8–14, respectively, so D4 is not uniformly harder. Inventory's multihop per-leg-charging defect forces multihop structure; base probabilities are not realized DEV proportions. Directed costs, surplus retention, node balance, and index consistency vary independently. SOCP varies the presentation of the intended standard convex formulation: norm form, free-threshold squared form with an explicit convex intent, and explicitly nonnegative squared form. Guidance follows the variant's semantics, and acceptance requires $t \geq 0$.

TSP gap. Let $z^\star_{\mathrm{TSP}}$ be the optimal single closed-tour cost and $z^\star_{\mathrm{CC}}$ the optimal cycle-cover cost under one incoming and one outgoing arc per city, allowing multiple cycles. The generator requires an optimal cycle cover with at least two cycles and

$$\Delta_{\mathrm{subtour}} = \frac{z^\star_{\mathrm{TSP}} - z^\star_{\mathrm{CC}}}{z^\star_{\mathrm{TSP}}}, \qquad \delta_{\min} \leq \Delta_{\mathrm{subtour}} \leq \delta_{\max}, \quad (\delta_{\min}, \delta_{\max}) = (0.02, 0.62). \tag{12}$$

The denominator is the complete-tour optimum. This gap measures objective distortion from omitting subtour elimination, not model-solving difficulty.

C.6 Generation limits and failure handling

With quotas frozen, we synthesize new candidate problems for cells with unfilled verified-example quotas in each round. A deficit is the quota minus the number of retained examples. Each request multiplies that cell's deficit by μ_{early} through batch b_{switch}, then by μ_{late}. This changes candidate effort, not accepted quotas. Table 13 records the configured limits.

Table 13: Generation-control hyperparameters. The batch cap is per round, not per problem.

Parameter	Scope	Value
K_{prep}	Outer preparation attempts per candidate slot	1
K_{rewrite}	Maximum description-rewriting attempts	2
K_{solve}	Fixed-assisted samples per eligible problem	1
b_{switch}	Last batch with the initial request multiplier	48
μ_{early}; μ_{late}	Early and late candidate-request multipliers	2; 16
K_{gen}	Maximum candidate batches per round	160

Thus, batches 1–48 request twice the remaining deficit and batches 49–160 sixteen times it. Internal mathematical-instance construction may retry; these limits do not bound total model or solver calls. At an unresolved shortfall, the program saves target/actual counts by task and stratum, candidates, responses, verification results, and rejection reasons; writes `GENERATION_SHORTFALL_Rxx.json`; raises an exception; and enters `paused_for_review`.

It does not train, advance, transfer quotas, or substitute old data. Training starts only when all planned verified quotas are filled.

C.7 Training configuration and evaluation statistics

Sequential adapter training. Training starts from the original SIRL-Qwen2.5-7B checkpoint. We initialize a new LoRA adapter for R_1 and resume the preceding round's final adapter and AdamW state in subsequent rounds. Base-model weights remain frozen. The learning-rate schedule follows global optimizer updates across rounds rather than restarting at each round. Each round trains for one epoch on 128 newly synthesized examples, with completion-only loss on assistant responses; problem tokens are masked from the loss. The maximum training sequence length is 12,288 tokens, including the problem and response.

Table 14: Training configuration for the recursive SFT run.

Setting	Configuration
Adaptation	LoRA; rank 64, alpha 128, dropout 0.05, bias `none`
Target modules	`q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`
Optimizer	Fused AdamW; $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon_{\text{opt}} = 10^{-8}$; weight decay 0.01
Gradient clipping	Maximum gradient norm 1.0
Learning rate	Scale 8×10^{-6}, floor 10^{-6}; 4-update warmup multiplier with global exponential decay (Eq. 13)
Hardware and batch	4 A800 GPUs; micro-batch 1 per GPU; gradient accumulation 2; effective batch 8
Per-round training	128 examples; 1 epoch; 16 optimizer updates
Total updates	$15 \times 16 = 240$ optimizer updates
Sequence and loss	Maximum 12,288 tokens; no packing; completion-only loss
Numerical settings	BF16, TF32, SDPA, and gradient checkpointing
Cross-round state	LoRA parameters and AdamW state continued; global learning-rate progress retained
Final checkpoint	Prespecified R_{15} after 15 training rounds

Global learning-rate schedule. Let g be the number of completed optimizer updates immediately before the current update. The 15-round run uses $g = 0, \ldots, 239$, with round $t \in \{1, \ldots, 15\}$ covering $g = 16(t-1), \ldots, 16t-1$. The learning rate is

$$\eta(g) = \max\left\{10^{-6},\ 8 \times 10^{-6} \min\left(1, \frac{g+1}{4}\right)(0.98)^{g/16}\right\}. \tag{13}$$

The warmup multiplier reaches one at $g = 3$; the exponential factor applies throughout, including warmup. The scale 8×10^{-6} is the nominal learning-rate scale, and 10^{-6} is the floor. The schedule does not reset at round boundaries.

Evaluation mean and standard deviation. For benchmark b, checkpoint t, and evaluation seed $s \in \mathcal{S} = \{42, 43, 44\}$, let $c_{b,t,s}$ be the number of correct responses among N_b problems. Define

$$A_{b,t,s} = 100\frac{c_{b,t,s}}{N_b}, \qquad \bar{A}_{b,t} = \frac{1}{3}\sum_{s\in\mathcal{S}} A_{b,t,s}, \qquad \sigma_{b,t} = \sqrt{\frac{1}{3}\sum_{s\in\mathcal{S}}(A_{b,t,s} - \bar{A}_{b,t})^2}. \tag{14}$$

Table 1 reports $\bar{A}_{b,t} \pm \sigma_{b,t}$ in percent. We compute both from unrounded run-level accuracies and use divisor 3 for the standard deviation (`ddof=0`).

Prespecified final model. The trajectory $R_0, \ldots, R_{15}$ contains the initial checkpoint and 15 updated checkpoints. Intermediate benchmark scores describe this trajectory and are not used to select the final model.