FROM DEMONSTRATIONS TO SOLVER-GUIDED EXPLORATION: LEARNING TO FORMULATE CONSTRAINT PROGRAMS FROM NATURAL LANGUAGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Translating natural-language descriptions into executable constraint programming (CP) models requires more than generating code that compiles. A model may return a solution while omitting a required constraint, using an incorrect domain, or optimizing the wrong objective. Human experts address such errors by inspecting solver feedback and revising the formulation, whereas existing LLM-based CP modeling systems often generate a model once or follow a fixed repair procedure. We introduce CPARL, a solver-grounded learning framework that combines CP-specific supervised fine-tuning with agentic reinforcement learning. The supervised stage first teaches direct modeling and solver-assisted revision through general demonstrations, followed by a second round targeting errors identified on a separate development set. Building on these capabilities, the reinforcement learning stage learns to choose between direct generation and solver-assisted revision, and to control model edits, solver operations, and termination. For tool-assisted trajectories, we propose Solver-Event Group Relative Policy Optimization (SE-GRPO), which groups decisions between successive solver evaluations and assigns credit using incremental solver evidence, anchored by semantic validation. We further introduce NL4CP, a new CP modeling benchmark built from formally specified problems, and evaluate CPARL on NL4CP and four public benchmarks. Using a Qwen3.5-27B backbone, CPARL achieves 35.0% micro and 44.6% macro pass@1 accuracy, exceeding all evaluated baselines fine-tuned for optimization modeling and directly prompted DeepSeek-V4-Pro and GLM-5.1 in both micro and macro averages. Ablation studies examine the contributions of the training stages, supervision mixture, and solver-event credit assignment. We analyze code executability and executable-but-invalid failures.

1 INTRODUCTION

Constraint programming (CP) is a declarative paradigm for combinatorial decision and optimization problems (Rossi et al., 2008). It has been applied to scheduling (Zeballos, 2010), routing (Backer et al., 2000), resource allocation (Zhang et al., 2013), configuration (Sawyer et al., 2012), and software verification (Gotlieb, 2012). CP modeling languages provide global constraints such as `AllDifferent`, `Cumulative`, and `Circuit`, allowing recurring combinatorial structures to be expressed directly and handled by specialized propagation algorithms. Given a natural-language description and instance data, however, a modeler must still choose decision variables and their domains, express every required constraint, and define an objective when the task involves optimization. A small omission can change the problem entirely. For example, a staff-scheduling model without a coverage constraint may compile and return a schedule, but that schedule does not meet the stated staffing requirements. This difficulty was identified in early work on CP usability (Puget, 2004) and automated modeling (O'Sullivan, 2010). It remains central in modular natural-language-to-CP translation (Tsouros et al., 2023), conversational CP (Freuder, 2024), and recent solver-integrated modeling systems (Shi et al., 2025).

Large language models (LLMs) offer a possible interface from natural language to executable CP code. CP-Bench evaluates generation across MiniZinc, CPMpy, and OR-Tools (Michailidis et al.,

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.

2025); TEXT2ZINC focuses on data-compatible MiniZinc models (Singirikonda et al., 2025); and CPEVAL studies MiniZinc and PyCSP3 generation with problem-specific checkers (Song & Cohen, 2025). Their results expose an important gap between executability and correctness. Generated models may compile while admitting invalid solutions, excluding valid ones, or optimizing a different objective; they are also sensitive to paraphrases and distracting context (Pellegrino & Mauro, 2025). Our error analysis finds the same failure modes, including missing constraints, incorrect domains, faulty reification, and index mismatches (see App. F).

CP experts address these problems through iterative interaction with solvers such as Gecode, Choco, and OR-Tools CP-SAT. They compile an initial formulation, inspect errors and candidate solutions, examine propagation or conflicts, and revise the model accordingly. Recent LLM systems automate parts of this process. Song & Cohen (2025) repeatedly compile and repair generated models and use checker feedback to align their outputs, while Shi et al. (2025) combine supervised fine-tuning and retrieval with solver-scored search and validation-triggered correction. In both cases, the available operations and the rules for invoking them are fixed by the system designer. A single repair procedure is unlikely to be equally useful for a syntax error, a missing constraint, a weak decomposition, and an incorrect objective. Moreover, a structural edit may reveal its effect only after later propagation or validation. This raises our central question: *can a single LLM agent learn to construct and improve CP models by choosing which solver evidence to request and how to revise them?*

Our key idea is to train an LLM agent to decide when to use a solver and how to use its feedback to improve a CP model. Given a problem description, the agent can reason through it and submit a complete CP program directly (*direct generation*). Alternatively, it can consult a solver during model construction and revise its code in response to the feedback (*solver-assisted revision*). We introduce CPARL, a solver-grounded learning framework that develops both abilities in two stages: CP-specific supervised fine-tuning (SFT), followed by agentic reinforcement learning (Agentic RL). Stage I uses two rounds of SFT. The first teaches model construction and solver interaction from successful demonstrations. The second adds examples addressing errors identified on a separate development set, providing further practice on modeling decisions the agent still gets wrong. Stage II then lets the agent construct CP programs on its own and learn from solver feedback and the correctness of their outputs. It trains the agent to choose whether to use a solver and, when doing so, which operations to request, how to revise the model, and when to stop.

A central challenge is deciding what feedback should guide learning. A program may compile and return a solution while still omitting a required constraint or using the wrong objective. CPARL therefore runs the completed CP program and checks its output against the task's constraints or reference answer. It uses the same checks whether the program is generated directly or revised with solver assistance. The resulting pass/fail feedback trains direct model generation and the initial choice of whether to use a solver. This feedback alone, however, gives little guidance about which earlier edits or solver calls helped. For *solver-assisted revision*, we introduce Solver-Event Group Relative Policy Optimization (SE-GRPO). It groups the model edits and tool decisions made between successive evaluations of the current model, then scores each group using the change in available evidence. For example, adding a missing constraint can help the program produce a valid solution and earn a reward. Repeating a solver call that returns the same information earns no additional reward. Successful compilation is useful, but the main goal is to learn which modeling decisions help solve the stated problem correctly.

We make four contributions: (i) we introduce CPARL, a two-stage framework that combines SFT and RL to learn CP model construction and solver use, and propose SE-GRPO, a new RL algorithm that groups model edits and tool choices between consecutive solver checks and uses changes in solver feedback to improve these decisions; (ii) we construct CPARL-SFT-8K, with 8,156 demonstrations for general and targeted SFT, CPARL-DEV-1K, a 1,000-problem development set for error analysis, targeted data construction, and model selection, and CPARL-RL-POOL for learning through the agent's own solver interactions; (iii) we introduce NL4CP, a new natural-language CP modeling benchmark built from formally specified XCSP3 competition problems. It covers satisfaction and optimization, pairing newly written descriptions with independent checks of solutions and objective values against the source specifications; and (iv) we compare CPARL with seven general-purpose LLMs and five LLMs fine-tuned for optimization modeling on NL4CP and four public benchmarks. CPARL achieves higher mean micro and macro pass@1 accuracy than all five fine-tuned models, DeepSeek-V4-Pro, and GLM-5.1. Our ablation studies show that the targeted second SFT round and agentic RL each improve CP modeling accuracy.

2 Related Work

LLMs for CP Modeling. Recent work studies how LLMs translate natural-language problems into CP programs. CP-Bench (Michailidis et al., 2025) evaluates this ability across MiniZinc, CPMpy, and OR-Tools. TEXT2ZINC (Singirikonda et al., 2025) focuses on generating MiniZinc models that work with the supplied instance data, while CPEVAL (Song & Cohen, 2025) uses problem-specific checkers to test the solutions produced by generated programs. These studies show that a program can run successfully yet fail to solve the stated problem. Solver feedback can help identify and correct such errors. ConstraintLLM (Shi et al., 2025) combines supervised fine-tuning, retrieval, and tree search to generate and repair CP models. However, predefined rules determine when it checks and repairs a model. CPARL learns these decisions: whether to use a solver, which operation to call, how to revise the model, and when to stop. Our SE-GRPO algorithm uses changes in solver feedback after each check to reward the preceding edits and tool choices.

Learning from Solver Feedback. Recent work improves LLM-based optimization modeling through synthetic training data and solver-based feedback. ORLM (Huang et al., 2025) expands and augments seed examples for SFT on mathematical formulations and solver code. OptMATH (Lu et al., 2025) generates descriptions from optimization instances, then filters reconstructed models by agreement with the original optimal objective values. ReSocratic (Yang et al., 2025) converts stepwise modeling demonstrations into questions and executable code for SFT; OptiBench evaluates the resulting numerical answers. SIRL (Chen et al., 2025) uses RL rewards for executable code and matching reference objective values, adding a bonus for advanced modeling structures. StepORLM (Zhou et al., 2026) combines solver outcomes with a learned evaluator of reasoning steps to construct preferences for weighted direct preference optimization (DPO). CALM (Tang et al., 2026) curates SFT data by correcting a model's own reasoning with brief expert hints; STORM then learns through GRPO with interactive code execution and a binary final-answer correctness reward. SAGE (Zhao et al., 2026) trains on solver-verified examples of different modeling strategies, then applies GRPO with larger weights for strategy tokens and rewards for format, correctness, and solver efficiency. OR-R1 (Ding et al., 2026) performs RL on unlabeled test problems, using majority-voted objective values from generated programs as proxy labels. Our SE-GRPO algorithm uses changes in solver feedback to assign credit to the CP model edits and tool choices between successive checks. STORM uses a final-answer reward, while SAGE applies fixed section weights to a trajectory-level advantage. See App. A for further discussion.

3 Background and Problem Formulation

Definition 1 (CP model)**.** In this work, a *constraint programming (CP) model* is a finite-domain model $M = (X, D, C, f)$, where X is a set of decision variables, D gives their allowed values, C is a set of constraints, and f is an optional objective. A solution assigns each variable a value from its domain and satisfies every constraint. Satisfaction tasks seek any such solution; optimization tasks seek one that minimizes or maximizes f, as specified by the problem.

Definition 2 (CP modeling)**.** *CP modeling* is the process of translating a problem's requirements into a CP model: choosing the variables and their domains, expressing the constraints, and defining an objective when needed. In this work, the input is a natural-language problem description q and instance data I. An LLM automates these modeling decisions by generating an executable CPMpy program implementing M. It invokes a solver and reports the result in the required format.

Definition 3 (Semantic correctness)**.** A model is *semantically correct* if it allows every valid assignment and excludes every invalid one, while preserving the specified objective for optimization tasks. Let $\mathcal{F}_q(I)$ and $\mathcal{F}_M(I)$ be the sets of assignments allowed by the task and by the model, respectively, compared on the task variables after projecting out any auxiliary variables. For optimization, let f_q and f_M be the specified and implemented objective functions. Correctness requires

$$\mathcal{F}_M(I) = \mathcal{F}_q(I), \qquad f_M(\mathbf{x}) = f_q(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{F}_q(I), \tag{1}$$

where $\mathbf{x}$ assigns a value to each task variable. The second equality applies only to optimization tasks, which must also preserve the specified minimization or maximization direction.

For example, if tasks must occupy distinct time slots, omitting this constraint permits invalid schedules; adding an unnecessary restriction can exclude valid ones. Different code and constraint encodings are allowed as long as they preserve the solution set and objective in Definition 3.

Definition 4 (Executability and validation)**.** A program is *executable* under an evaluation protocol if it parses, constructs a model, obtains a solver result within the execution limits, and returns output in the required format. *Validation* checks that output against the task's constraints or reference answers. The program *passes validation* when the available task checker accepts its output (App. C).

An executable program can still represent the wrong problem. Passing validation provides evidence of correctness, but does not prove the full semantic equivalence in Definition 3. Solvers also use *propagation* to remove variable values inconsistent with the encoded constraints before or during search. Different encodings of the same constraints can eliminate different numbers of candidate values. However, an incorrect constraint can also remove values by excluding valid solutions, so stronger propagation alone does not establish correctness.

4 CPARL: Learning from Demonstrations and Solver Interaction

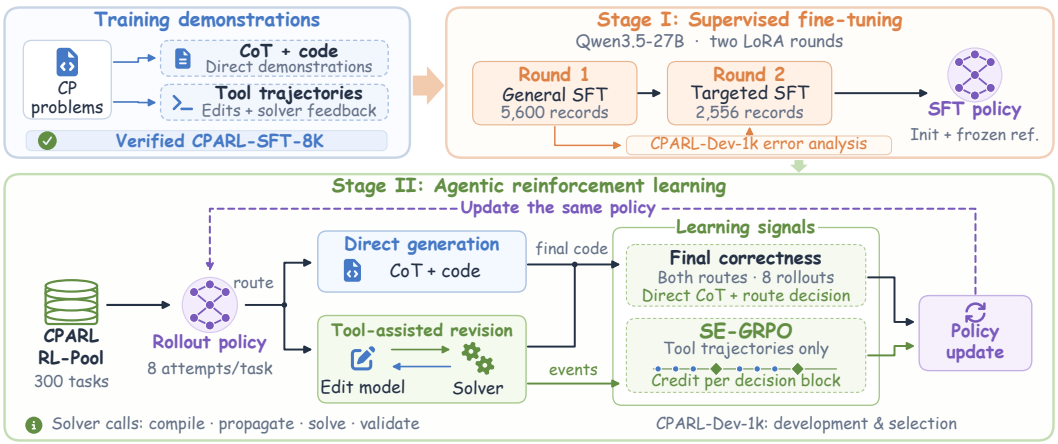

Figure 1: Overview of CPARL. In Stage I (top), verified examples show the LLM how to build CP programs, both directly and with solver feedback. A second SFT round targets errors that remain after the first. In Stage II (bottom), the LLM chooses between these approaches and learns from its own attempts. Final pass/fail results help it learn to generate correct programs directly and decide when to request feedback. During revision, SE-GRPO uses changes in solver feedback to reward the model edits and tool choices between successive checks. This helps distinguish useful repairs from repeated calls that provide no new information. Both signals update the same LLM.

CPARL trains an LLM in two stages to construct CP programs and decide when and how to use solver feedback (Figure 1). We first describe the two SFT rounds for learning from demonstrations (Sec. 4.1). Next, we explain how the LLM learns from its own attempts with Agentic RL and SE-GRPO, and how the trained LLM generates CP programs for new tasks (Sec. 4.2). Finally, we introduce our NL4CP benchmark (Sec. 4.3).

4.1 Stage I: Two-Round Mixed-Route SFT

We first train the LLM to build CP programs and revise them using solver feedback. Round 1 provides examples of both skills. We then test the trained LLM on development problems and use the mistakes it makes to prepare additional examples for Round 2.

Round 1: Learning to model with and without solver feedback. The first round uses 5,600 CP modeling problems adapted from CSPLib, the CPMpy example library, NLP4LP, and the XCSP3 competition website (see App. D.1 for data sources and problem selection). All three demonstration formats begin with the same input: the natural-language problem description q and instance data I. An *answer-only demonstration* uses a complete CPMpy program as the target output, without CoT reasoning or tool interactions. For each problem, we prepare a *chain-of-thought (CoT) demonstration* whose target output is a short explanation of the modeling steps followed by a complete CPMpy program. We also prepare a *tool demonstration* to teach program construction and revision through

interaction. At each step, the LLM receives the problem, its current program, and feedback from previous calls; the target output is an action that writes or edits code, requests a solver operation, or submits the final program. Solver responses are supplied as input to later decisions, not generated as part of the LLM's target output (see App. D.4 for record structure and supervision). We execute the final CoT and tool programs and check their solutions against the problem's constraints. For optimization, we also compare objective values with the reference. Only demonstrations that pass enter the training set (see App. D.1 for program checks and expert review).

We mix the two types of demonstrations to teach both the steps for building a complete model and how to revise it after receiving solver feedback. Each problem has both a CoT demonstration and a tool demonstration, but each training run uses only one of them. In our selected mixture, we use tool demonstrations for the 2,240 problems estimated to be most difficult (40%). For the remaining 3,360 problems (60%), we use CoT demonstrations (see App. D.3 for difficulty assessment and demonstration selection). We chose this ratio by evaluating candidate first-round models on CPARL-DEV-1K. To check whether combining the two types helps, we compare *Mixed SFT R1* with models trained using only CoT, tool, or answer-only demonstrations. The mixture achieves higher mean micro and macro accuracy than all three (Sec. 5.3, Table 2). Table 6 also reports all eleven tool proportions from 0% to 100% in increments of 10 percentage points.

Round 2: Targeting remaining modeling errors. After Round 1, we inspect the selected model's failures on CPARL-DEV-1K to identify recurring modeling mistakes. We adapt the problems on which it failed, changing both their descriptions and mathematical formulations, and prepare demonstrations showing how to model the revised problems correctly. We then continue SFT on 2,556 verified demonstrations: 1,022 tool demonstrations and 1,534 CoT demonstrations. Together, the demonstrations from both rounds make up the 8,156-record CPARL-SFT-8K corpus. These additional examples come from development-set error analysis, and App. D.3 describes their construction. For comparison, *One-stage-SFT R1&R2* combines the examples from both rounds into one training set, randomizes their order, and trains on them in a single stage. Training first on the original examples and then on the additional examples (*Mixed-SFT R1+R2*) gives higher mean accuracy on all five benchmarks (Sec. 5.3, Table 2).

Learning from the demonstrated steps. In both SFT rounds, we train the LLM to predict the next demonstrated output using the information available at that point. It learns the initial choice of whether to use solver feedback, followed by the reasoning, code, and any tool actions shown in the demonstration. For example, a demonstration may show a solver error message followed by a code edit. The message is input to the LLM; the edit is the training target. Throughout each demonstration, task inputs, supplied program states, and solver responses provide context. We compute the loss only on the target outputs.

For a batch $\mathcal{B}$, let $z_{i,j}$ be token j in demonstration i. The LLM assigns it probability $p_\theta(z_{i,j} \mid z_{i,<j})$ after reading the preceding tokens $z_{i,<j}$. The mask $m_{i,j}$ selects which tokens to learn: it is 1 for target outputs and 0 for input context. The weight $w_{i,j}$ sets each selected token's contribution. The Round 1 loss is $\mathcal{L}_{\mathrm{SFT}} = -\frac{\sum_{i\in\mathcal{B}}\sum_j m_{i,j}w_{i,j}\log p_\theta(z_{i,j}|z_{i,<j})}{\sum_{i\in\mathcal{B}}\sum_j m_{i,j}w_{i,j}}$. The denominator makes this a weighted average over the target tokens. In Round 1, the initial-choice field and tool-action control fields receive weight 4; code and other output tokens receive weight 1. We fine-tune Qwen3.5-27B with LoRA in both rounds (see App. H.1 for token masks, weights, and training settings).

From demonstrations to interaction. We continue training the final SFT model in Stage II, where it learns from the results of its own modeling attempts (see App. D.6 for a comparison of the two training stages). It uses the same formats for program edits and solver calls as in the SFT demonstrations. We also keep an unchanged version of the SFT model as a reference. Given the same context, the RL loss penalizes differences between the trainable model's next-token probabilities and those of this fixed reference. This discourages large changes in behavior while the model learns to obtain better results (see App. H.4 for the reference penalty).

4.2 STAGE II: SOLVER-GROUNDED AGENTIC RL

Stage II continues from the SFT model, learning from the LLM's own attempts. A final check tells us whether an attempt succeeded, but not which edits or solver calls helped. We therefore introduce

Solver-Event Group Relative Policy Optimization (SE-GRPO), our RL algorithm for solver-assisted attempts. It uses changes in solver feedback to score the edits and tool choices made between checks.

Learning by comparing attempts. We generate several attempts per problem to compare modeling decisions. In each attempt, the LLM freely chooses to write a complete program with CoT reasoning and no intermediate feedback, or to build and revise one through solver calls. Both routes use the same final checker after submission. We then update the LLM and generate new attempts.

Since the LLM has already learned from SFT demonstrations, we focus RL on mistakes that remain. We construct CPARL-RL-POOL by adapting problems the second-round SFT model failed to solve on CPARL-DEV-1K (see App. D.5 for problem construction and sampling). Our main setting uses 300 problems, with eight attempts per problem each time it is sampled. A preliminary 1,000-problem setting shows a small additional gain under a different training schedule (see Sec. 5.4 for results and App. G.4 for the two schedules).

Choosing what to edit and check. During a solver-assisted attempt, the LLM uses the problem, current program, and previous feedback to decide what to edit or check next. It can check whether the program constructs a solver model (`compile`), inspect which candidate values can be eliminated (`propagate`), search for a solution (`solve`), or check the result against the task's requirements (`validate`). An attempt ends when it submits with `finalize` or reaches its interaction limit (see App. H.2 for editing actions, observations, and failure handling).

Grouping decisions that a check can evaluate. Between solver calls, the LLM may make several edits, but the next check only reveals their combined effect. We therefore evaluate these edits together with the checking call as one *solver-event block*. For example, adding a missing `AllDifferent` constraint to keep tasks in distinct time slots and then propagating the model form one block; validating that model forms the next. The first block includes initial program generation; `finalize` also closes a block. Let $a_{i,t}$ be decision t in attempt i, and $t_{i,k}$ the index of the decision closing block k. With $t_{i,0} = -1$, the block is $B_{i,k} = (a_{i,t_{i,k-1}+1}, \ldots, a_{i,t_{i,k}})$.

Turning feedback into a training score. To score a block, we compare each new solver result with the previous result of the same kind in that attempt. The *reward* combines changes in validation (ΔV), whether the program runs (ΔE), and propagation's removal of candidate values ($\Delta\Phi$). Penalties for model growth (ΔC) and interaction cost (κ) discourage unnecessary code and calls:

$$r_{i,k} = \lambda_{\text{val}}\Delta V_{i,k} + \lambda_{\text{exec}}\Delta E_{i,k} + \lambda_{\text{prop}} g^{\text{val}}_{i,k}\Delta\Phi_{i,k} - \lambda_{\text{size}}[\Delta C_{i,k}]_+ - \lambda_{\text{cost}}\kappa(B_{i,k}). \tag{2}$$

We select the nonnegative λ weights using performance on CPARL-DEV-1K; $[x]_+ = \max(x, 0)$ penalizes only increases in model size. Repeated checks without new evidence earn no progress reward but still incur a cost. Incorrect constraints can remove valid solutions, so the gate $g^{\text{val}}_{i,k}$ allows propagation credit only when a later validation check succeeds on the same program version (see App. H.3 for reward definitions and weighting).

Learning from later checks and other attempts. In the example above, evidence that the repair works may arrive at the next validation call. To credit the earlier edit for that later result, a block's *return* adds its own reward and subsequent rewards, with γ controlling how much later rewards count (see App. H.8 for a worked repair example). Early blocks have more opportunities to collect later rewards. We therefore compare first blocks with first blocks, second blocks with second blocks, and so on, among solver-assisted attempts at the same problem. The relative score, called the *advantage*, measures how a block's return compares with that group's average. For attempt i with K_i blocks,

$$R_{i,k} = \sum_{j=k}^{K_i} \gamma^{j-k} r_{i,j}, \qquad \widehat{A}_{i,k} = \frac{R_{i,k} - \mu_{q,k}}{\max(\sigma_{q,k}, \epsilon_A)}. \tag{3}$$

Here $\mu_{q,k}$ and $\sigma_{q,k}$ are the mean and population standard deviation of returns from attempts that reach block k on problem q; ϵ_A keeps the denominator positive. Blocks at the same position may contain different edits and solver operations.

Updating the LLM from these comparisons. Clipped GRPO (Shao et al., 2024) encourages decisions from blocks with above-average returns and discourages those from blocks with below-average returns. To avoid giving longer edits more weight, we average over LLM-generated tokens within each block, then over blocks within each attempt, and finally over attempts. Let G_{tool} be the number

of solver-assisted attempts for one problem, $\mathcal{U}_{i,k}$ the LLM-generated token positions in $B_{i,k}$, and $\ell_{i,k,u}$ the clipped token objective using $\widehat{A}_{i,k}$:

$$\mathcal{L}^{\text{tool}}_{\text{SE-GRPO}} = -\frac{1}{G_{\text{tool}}} \sum_{i=1}^{G_{\text{tool}}} \frac{1}{K_i} \sum_{k=1}^{K_i} \frac{1}{|\mathcal{U}_{i,k}|} \sum_{u \in \mathcal{U}_{i,k}} \ell_{i,k,u} + \beta \mathcal{K}_{\text{ref}}. \tag{4}$$

The term $\beta\mathcal{K}_{\text{ref}}$ discourages large departures from the frozen SFT policy (see App. H.4 for the token-level objective and comparisons when few attempts reach a block).

Learning whether to use solver feedback. Final correctness provides feedback for programs generated without solver calls and lets us compare the two approaches. We compare each attempt's final pass/fail result with the average across all eight attempts at the same problem. Using this outcome advantage, L_{CoT} trains reasoning and code on the CoT route, while L_{route} trains the initial choice across both routes (see App. H.5 for the outcome advantage and token masks). The complete objective is

$$\mathcal{L}_{\text{mixed}} = \mathcal{L}^{\text{tool}}_{\text{SE-GRPO}} + L_{\text{CoT}} + 0.1 L_{\text{route}} + \beta_{\text{add}} \mathcal{K}_{\text{new}}. \tag{5}$$

All terms update the same LLM, with the objective averaged over problems. $\mathcal{K}_{\text{new}}$ applies the SFT reference penalty only to tokens not already covered by the tool term (see App. H.5 and H.7 for training settings and the algorithm).

Inference. On a new problem, the trained LLM chooses whether to use solver feedback. We evaluate one attempt, ending at submission or the interaction limit. The LLM's parameters remain fixed, and no training rewards are computed. Submitted programs undergo the same execution and correctness checks on both routes (see App. H.2 for execution and stopping rules).

4.3 NL4CP: A New CP Modeling Benchmark

Why a new benchmark? A program may run successfully while omitting a required constraint. To assess CP modeling, we therefore need to check its returned solution against the problem's requirements. We build NL4CP from XCSP3 competition problems, whose variables, constraints, and objectives are formally specified. For each problem, we write a natural-language description and check that it preserves the original requirements and instance data. This adds newly written tasks with an independent reference for checking solutions (see App. E.1–E.3 for source selection, task format, and description checks).

Task and evaluation. The LLM generates a complete CPMpy program from a self-contained description with instance data. The formal instance and evaluation references are hidden. We run the program and check its solution against the original domains and constraints. For optimization, we recompute the objective from the solution and compare it with the benchmark reference (see App. E.4 for evaluation rules). We evaluate 147 tasks; App. E.1 lists source and retained counts.

5 Experiments

Our experiments test how well CPARL builds correct CP programs, which training choices help, and what errors remain: **RQ1:** How does CPARL compare with general-purpose LLMs and LLMs fine-tuned for optimization modeling (Sec. 5.2)? **RQ2:** How does accuracy change with mixed CoT and tool demonstrations, two SFT rounds, and subsequent RL (Sec. 5.3)? **RQ3:** How does RL accuracy vary with solver feedback, credit assignment, and optional tool use (Sec. 5.4)? **RQ4:** Is the main obstacle running the code or producing correct solutions, and how does training change these failures (Sec. 5.5)?

5.1 Experimental Setup

Datasets. We evaluate on CP-Bench (Michailidis et al., 2025), TEXT2ZINC (Singirikonda et al., 2025), CPEVAL (Song & Cohen, 2025), IndusCP (Shi et al., 2025), and our NL4CP (Sec. 4.3). All methods generate CPMpy programs using the same input format and checker within each benchmark (see App. B.1 for instance counts and interfaces).

Baselines. We directly prompt DeepSeek-V4-Pro, Qwen-3.6-Max-Preview, Qwen3.5-27B, GLM-5.1, GPT-5.4, Gemini-3.1-Pro-Preview, and Claude-4.6-Sonnet, without CP-specific training or intermediate solver feedback. We also evaluate models fine-tuned for general optimization modeling: ORLM-LLaMA3-8B (Huang et al., 2025), OptMATH-Qwen2.5-7B/32B (Lu et al., 2025), and SIRL-Qwen2.5-7B/32B (Chen et al., 2025). CPARL and its training variants share the Qwen3.5-27B backbone, whose direct-prompt score provides the unadapted reference (see App. C.3 for models and decoding settings, and App. G.3 for training controls).

Metrics. Pass@k is the percentage of instances solved correctly within k attempts, as verified by benchmark checkers. Code Executability Rate (CER@k) measures the percentage with at least one program that runs within execution limits and returns correctly formatted solver output. We report $k = 1, 4$, with micro and macro averages (see App. C.2 for definitions and aggregation).

5.2 RQ1: Overall Effectiveness

	Direct Prompt							General Fine-Tuned					Ours
Dataset	DeepSeek V4-Pro	Qwen 3.6-Max	Qwen 3.5-27B	GLM 5.1	GPT 5.4	Gemini 3.1-Pro	Claude 4.6-Sonnet	ORLM LLaMA3-8B	OptMATH Qwen2.5-7B	OptMATH Qwen2.5-32B	SIRL Qwen2.5-7B	SIRL Qwen2.5-32B	*CPARL* Qwen3.5-27B
CP-Bench	$53.9_{\pm 4.5}$	$66.5_{\pm 2.9}$	$2.3_{\pm 0.6}$	$52.3_{\pm 3.4}$	$60.4_{\pm 9.7}$	$69.1_{\pm 4.3}$	$50.5_{\pm 6.6}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$2.0_{\pm 0.0}$	$2.0_{\pm 0.0}$	$13.9_{\pm 0.0}$	$56.2_{\pm 1.5}$
TEXT2ZINC	$38.7_{\pm 1.7}$	$32.4_{\pm 2.2}$	$7.9_{\pm 3.7}$	$37.5_{\pm 3.4}$	$40.9_{\pm 2.7}$	$49.3_{\pm 2.5}$	$47.5_{\pm 1.2}$	$0.0_{\pm 0.0}$	$0.9_{\pm 0.0}$	$0.9_{\pm 0.0}$	$0.0_{\pm 0.0}$	$15.5_{\pm 0.0}$	$43.3_{\pm 2.7}$
CPEVAL	$69.1_{\pm 2.8}$	$73.8_{\pm 4.2}$	$15.6_{\pm 3.1}$	$67.8_{\pm 6.0}$	$70.9_{\pm 3.4}$	$75.9_{\pm 3.2}$	$71.3_{\pm 2.4}$	$0.0_{\pm 0.0}$	$6.3_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$24.4_{\pm 0.9}$	$64.4_{\pm 5.7}$
IndusCP	$23.8_{\pm 1.8}$	$33.2_{\pm 1.6}$	$2.6_{\pm 1.6}$	$23.6_{\pm 3.2}$	$34.9_{\pm 1.7}$	$32.4_{\pm 1.3}$	$33.6_{\pm 0.8}$	$0.0_{\pm 0.0}$	$0.3_{\pm 0.5}$	$0.0_{\pm 0.0}$	$5.2_{\pm 0.8}$	$6.7_{\pm 0.6}$	$25.1_{\pm 0.5}$
NL4CP	$33.3_{\pm 1.1}$	$35.0_{\pm 4.3}$	$1.4_{\pm 1.8}$	$29.0_{\pm 0.4}$	$36.9_{\pm 4.3}$	$42.2_{\pm 0.8}$	$41.8_{\pm 2.5}$	$0.0_{\pm 0.0}$	$1.4_{\pm 0.0}$	$0.7_{\pm 0.0}$	$0.7_{\pm 0.0}$	$8.4_{\pm 0.4}$	$33.9_{\pm 2.4}$
Micro Avg.	$33.8_{\pm 1.6}$	$40.0_{\pm 0.9}$	$4.0_{\pm 0.8}$	$32.6_{\pm 1.8}$	$41.4_{\pm 2.2}$	$43.3_{\pm 1.1}$	$41.2_{\pm 1.0}$	$0.0_{\pm 0.0}$	$0.9_{\pm 0.2}$	$0.4_{\pm 0.0}$	$3.1_{\pm 0.5}$	$10.1_{\pm 0.3}$	$35.0_{\pm 0.9}$
Macro Avg.	$43.8_{\pm 1.8}$	$48.2_{\pm 1.2}$	$6.0_{\pm 1.1}$	$42.0_{\pm 1.5}$	$48.8_{\pm 2.9}$	$53.8_{\pm 1.6}$	$48.9_{\pm 1.6}$	$0.0_{\pm 0.0}$	$1.8_{\pm 0.1}$	$0.7_{\pm 0.0}$	$1.6_{\pm 0.2}$	$13.8_{\pm 0.2}$	$44.6_{\pm 1.5}$

Table 1: Pass@1 accuracy (%): mean $\pm$ sample standard deviation over five decoding evaluations.

The framework improves pass@1 and pass@4 accuracy. CPARL trains Qwen3.5-27B with two SFT rounds and SE-GRPO, allowing direct generation or solver interaction. Macro pass@1 rises from 6.0% to 44.6%, with gains of at least 22.5 points on every benchmark (Table 1); macro pass@4 rises from 9.2% to 58.1% (Table 12). The gains span all five benchmarks under both attempt budgets.

NL4CP remains challenging even with four attempts. On NL4CP, CPARL raises the backbone's pass@1 from 1.4% to 33.9%, versus 8.4% for SIRL-Qwen2.5-32B, the best fine-tuned baseline (Table 1). CPARL reaches 47.6% pass@4, while Gemini-3.1-Pro leads at 61.9% (Table 12). Even the best model leaves over a third of instances unsolved. This makes NL4CP a useful testbed for assessing progress in generating correct CP programs from natural-language descriptions.

CPARL outperforms fine-tuned baselines and competes with general-purpose models. CPARL exceeds all five optimization fine-tuned baselines in pass@1 on every benchmark, and DeepSeek-V4-Pro and GLM-5.1 in both Micro Avg. and Macro Avg. (Table 1). Although Gemini-3.1-Pro leads in Macro Avg. pass@4 (63.8% versus CPARL's 58.1%; Table 12), CPARL leads on TEXT2ZINC (60.9% versus 58.2%; Table 12). This higher pass@4 comes despite lower CER@4 (73.6% versus 92.7%; Table 13), showing that executability does not ensure correctness.

5.3 RQ2: Demonstration Mixtures and Staged Training

CoT-only leads among single-format controls. *Answer-only SFT R1* learns to output a CPMpy program without reasoning or tools. *CoT-only-SFT R1* adds modeling steps before code, without tool calls, gaining 1.4 micro and 2.6 macro points over *Answer-only* (Table 2). *Tool-only-SFT R1* learns to write code, call the solver, and revise using feedback, but scores 1.2 micro and 2.3 macro points lower than *Answer-only* (Table 2). Thus, tool interaction alone does not ensure better modeling.

Mixed supervision outperforms single-format controls. *Mixed SFT R1* uses 5,600 demonstrations (40% tools, 60% CoT). It exceeds *Answer-only*, *CoT-only*, and *Tool-only SFT R1* in both Micro Avg. and Macro Avg., gaining 3.7 macro points over the strongest control, *CoT-only* (Table 2).

More tool demonstrations do not necessarily improve accuracy. As shown in Table 6, the 40% tool / 60% CoT mixture performs best among eleven ratio settings, with 29.2% micro and 36.2% macro accuracy. At 50% tools, these averages fall to 27.7% and 33.4%; tool-only training scores 23.7% and 27.6%. These results support keeping reasoning demonstrations alongside tool trajectories, rather than replacing them with more tool interactions. See App. D.3 for mixture construction.

Two-round SFT outperforms one-round SFT on the same data. *Mixed-SFT R1+R2* continues *Mixed SFT R1* on 2,556 demonstrations targeting development-set errors, keeping the same mixture. In contrast, *One-stage-SFT R1&R2* combines and shuffles all 8,156 demonstrations from both rounds for one SFT round. Two-round SFT scores 6.1 macro points higher than one-round SFT, with gains on all five benchmarks (Table 2). Compared with *Mixed SFT R1* alone, the second round adds 4.9 macro points (Table 2). See App. G.3 for training settings and evaluation details.

5.4 RQ3: How RL Training Choices Affect Accuracy

All four RL variants in Table 2 start from *Mixed-SFT R1+R2* and use the same 300-task training pool and schedule.

	Variant	CP-Bench	TEXT2ZINC	CPEVAL	IndusCP	NL4CP	Micro Avg.	Macro Avg.
SFT	*Answer-only SFT R1*	$42.6_{\pm 1.7}$	$25.5_{\pm 0.9}$	$38.5_{\pm 5.0}$	$19.7_{\pm 0.9}$	$23.1_{\pm 1.4}$	$24.9_{\pm 0.7}$	$29.9_{\pm 0.9}$
	CoT-only-SFT R1	$41.9_{\pm 0.6}$	$28.2_{\pm 2.4}$	$45.8_{\pm 2.4}$	$19.7_{\pm 0.8}$	$27.0_{\pm 1.0}$	$26.3_{\pm 0.7}$	$32.5_{\pm 0.5}$
	Tool-only-SFT R1	$39.3_{\pm 1.5}$	$18.2_{\pm 0.9}$	$35.9_{\pm 3.1}$	$19.6_{\pm 1.5}$	$24.9_{\pm 1.4}$	$23.7_{\pm 1.1}$	$27.6_{\pm 0.8}$
	Mixed SFT R1	$46.2_{\pm 3.7}$	$35.5_{\pm 5.5}$	$51.0_{\pm 7.2}$	$22.2_{\pm 0.2}$	$26.3_{\pm 3.9}$	$29.2_{\pm 1.0}$	$36.2_{\pm 2.0}$
	Mixed-SFT R1+R2	$50.2_{\pm 0.6}$	$39.7_{\pm 1.4}$	$62.0_{\pm 3.9}$	$24.0_{\pm 0.9}$	$29.7_{\pm 1.0}$	$32.5_{\pm 0.4}$	$41.1_{\pm 0.3}$
	One-stage-SFT R1&R2	$42.2_{\pm 2.5}$	$35.8_{\pm 1.9}$	$52.6_{\pm 0.9}$	$20.4_{\pm 0.5}$	$23.8_{\pm 2.0}$	$27.6_{\pm 0.5}$	$35.0_{\pm 0.5}$
RL	***Full CPARL***	$56.2_{\pm 1.5}$	$43.3_{\pm 2.7}$	$64.4_{\pm 5.7}$	$25.1_{\pm 0.5}$	$33.9_{\pm 2.4}$	$35.0_{\pm 0.9}$	$44.6_{\pm 1.5}$
	GRPO	$61.2_{\pm 3.5}$	$40.9_{\pm 2.1}$	$54.4_{\pm 3.0}$	$24.9_{\pm 1.2}$	$31.3_{\pm 1.5}$	$34.0_{\pm 1.3}$	$42.5_{\pm 1.6}$
	Trajectory-level credit	$57.4_{\pm 2.5}$	$38.0_{\pm 1.5}$	$67.5_{\pm 3.0}$	$25.1_{\pm 0.6}$	$30.5_{\pm 2.0}$	$34.2_{\pm 0.6}$	$43.7_{\pm 1.1}$
	Tool-only SE-GRPO	$56.6_{\pm 4.3}$	$39.5_{\pm 1.9}$	$57.2_{\pm 5.0}$	$24.9_{\pm 0.5}$	$31.6_{\pm 1.8}$	$33.6_{\pm 0.8}$	$41.9_{\pm 1.6}$

Table 2: Training variants (pass@1 accuracy, %). Mean ± sample SD. See App. G.3 for control settings and evaluation details.

***Full CPARL* outperforms *GRPO*.** Both *Full CPARL* and *GRPO* can write code directly or revise it using solver feedback. *Full CPARL* uses SE-GRPO to score each block of edits and tool calls from feedback changes. Final correctness trains its direct code generation and decision to use tools. *GRPO* assigns one advantage per attempt using final correctness alone. *Full CPARL* scores 1.0 point higher than *GRPO* in Micro Avg. pass@1 (Table 2) and improves all five benchmarks over *Mixed-SFT R1+R2* (+3.5 macro points; Table 2).

Block-level credit improves accuracy. *Trajectory-level credit* keeps *Full CPARL*'s rewards, routes, and losses but uses one advantage per tool trajectory. *Full CPARL* gains 0.8 micro and 0.9 macro points over this control (Table 2). These gains support separate credit for each decision block.

Keeping both routes helps. *Tool-only SE-GRPO* forces tools in training and evaluation, retaining event rewards and block credit but removing direct-generation and route losses. CPARL achieves 35.0% micro accuracy versus 33.6% for *Tool-only SE-GRPO*, a 1.4-point gain (Table 2).

5.5 RQ4: From Executable Programs to Correct Solutions

Runnable code need not yield correct solutions. Gemini-3.1-Pro reaches 70.5% macro CER@1 but 53.8% pass@1 (Table 15). CPARL's CER@4 exceeds pass@4 by 25.3 points on IndusCP and 9.5 on NL4CP (Tables 12 and 13), leaving these instances unsolved despite runnable candidates in four attempts. A solver can solve a model that misrepresents the problem. These gaps motivate validating modeled constraints and objectives against the task, beyond checking execution.

Training reduces execution errors. As shown in Figure 2, execution errors account for 67.0% of Qwen3.5-27B's error labels, compared with 30.7% for CPARL. This shift suggests that training helps the model avoid basic programming mistakes. Timeout/budget and constraint errors account for 27.2% and 15.8% of CPARL's labels, respectively. These errors highlight two next steps: using the execution budget effectively and translating problem requirements into correct constraints.

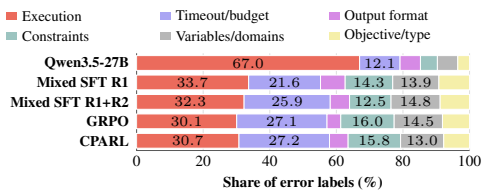

Figure 2: Error-label shares (one evaluation/model). See App. F for label definitions.

6 Conclusion

CPARL combines two SFT rounds with solver-guided RL to learn CP modeling and tool use. SE-GRPO assigns credit to model edits and tool choices using changes in solver feedback. Across five benchmarks, CPARL improves pass@1 and pass@4 over the base model. We also introduce NL4CP to evaluate CP modeling from natural-language descriptions. Future work will explore more diverse training tasks and better task selection.

AI USE STATEMENT

We used generative AI tools to help prepare training problems, generate and refine CoT and tool-interaction demonstrations, and draft natural-language descriptions for NL4CP. These workflows include human review and execution-based checks of the associated programs (see App. D.1–D.3 and E.3 for data construction and verification). AI tools also assisted with language polishing. The authors take full responsibility for the content and accuracy of the final manuscript.

REPRODUCIBILITY STATEMENT

Sec. 4 and App. H specify the training objectives, interaction protocol, and algorithms. App. D and E document training-data and benchmark construction, while App. B, C, and G give the execution and metric definitions, evaluation settings, and reported results. Full evaluation prompts and rollout examples appear in App. J and I.

Code and datasets: `https://anonymous.4open.science/r/cparl2027/`.

REFERENCES

Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of heuristics*, 6(4): 501–523, 2000.

Quentin Cappart, Tias Guns, Michele Lombardi, Gilles Pesant, and Dimos Tsouros. Combining constraint programming and machine learning: From current progress to future opportunities. *Journal of Artificial Intelligence Research*, 84, 2025.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. Or-r1: Automating modeling and solving of operations research optimization problem via test-time reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 228–236, 2026. doi: 10.1609/aaai.v40i1.36983.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. ReTool: Reinforcement learning for strategic tool use in LLMs. *arXiv preprint arXiv:2504.11536*, 2025a.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Advances in Neural Information Processing Systems*, volume 38, 2025b.

Eugene C. Freuder. Conversational modeling for constraint satisfaction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 22592–22597, 2024. doi: 10.1609/aaai.v38i20.30268.

Arnaud Gotlieb. Tcas software verification using constraint programming. *The Knowledge Engineering Review*, 27(3):343–360, 2012.

Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

Daya Guo, Dejian Yang, Haowei Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

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. *Operations Research*, 73(6):2986–3009, 2025. doi: 10.1287/opre.2024.1233.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Christophe Lecoutre and Nicolas Szczepanski. Pycsp3: modeling combinatorial constrained problems in python. *arXiv preprint arXiv:2009.00326*, 2020.

Zhong Li, Zihan Guo, Xiaohan Lu, Juntao Wang, Jie Song, Chao Shen, Jiageng Wu, and Mingyang Sun. OptArgus: A multi-agent system to detect hallucinations in LLM-based optimization modeling. *arXiv preprint arXiv:2605.11738*, 2026a.

Zhong Li, Qi Huang, Yuxuan Zhu, Mohammad Mohammadi Amiri, Niki van Stein, Thomas Bäck, Matthijs van Leeuwen, Zaiwen Wen, and Lincen Yang. MM-OptBench: A solver-grounded benchmark for multimodal optimization modeling. *arXiv preprint arXiv:2605.12154*, 2026b.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *Forty-third International Conference on Machine Learning*, 2026c. URL `https://openreview.net/forum?id=0Q7adIsv6p`.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 40769–40802. PMLR, 2025.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. ReFT: Reasoning with reinforced fine-tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7601–7614. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.410.

Kostis Michailidis, Dimos Tsouros, and Tias Guns. Cp-bench: Evaluating large language models for constraint modelling. In *ECAI 2025*, pp. 861–868. IOS Press, 2025.

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming – CP 2007*, pp. 529–543. Springer, 2007. doi: 10.1007/978-3-540-74970-7_38.

Barry O'Sullivan. Automated modelling and solving in constraint programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pp. 1493–1497, 2010.

Alessio Pellegrino and Jacopo Mauro. When words change the model: Sensitivity of llms for constraint programming modelling. *arXiv preprint arXiv:2511.14334*, 2025.

Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In *International Conference on Principles and Practice of Constraint Programming*, pp. 5–8. Springer, 2004.

Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. Using constraint programming to manage configurations in self-adaptive systems. *Computer*, 45(10):56–63, 2012.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Weichun Shi, Minghao Liu, Wanting Zhang, Langchen Shi, Fuqi Jia, Feifei Ma, and Jian Zhang. Constraintllm: A neuro-symbolic framework for industrial-level constraint programming. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15999–16019. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.809.

Akash Singirikonda, Serdar Kadioglu, and Karthik Uppuluri. Text2zinc: A cross-domain dataset for modeling optimization and satisfaction problems in minizinc. *arXiv preprint arXiv:2503.10642*, 2025.

Yuliang Song and Eldan Cohen. Do llms understand constraint programming? zero-shot constraint programming model generation using llms. In *International Conference on Learning and Intelligent Optimization*, pp. 16–31. Springer, 2025.

Zhengyang Tang, Zihan Ye, Chenyu Huang, Xuhan Huang, Chengpeng Li, Sihang Li, Guanhua Chen, Ming Yan, Zizhuo Wang, Hongyuan Zha, Dayiheng Liu, and Benyou Wang. CALM before the STORM: Unlocking native reasoning for optimization modeling. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026. URL https://openreview.net/forum?id=VFZWSzWgtS.

Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv preprint arXiv:2308.01589*, 2023.

Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. Streamllm: Enhancing constraint programming with large language model-generated streamliners. In *2025 IEEE/ACM 1st International Workshop on Neuro-Symbolic Software Engineering (NSE)*, pp. 17–22. IEEE Computer Society, 2025.

Zhuohan Wang, Ziwei Zhu, Ziniu Li, Congliang Chen, Zhihang Lin, MingZhe Yang, Yizhou Han, Yufeng Lin, Angyang Gu, Xinglin Hu, Ruoyu Sun, and Tian Ding. Certified evaluation for LLMs in optimization modeling: From graph isomorphism to formulation isomorphism. In *ICML 2026 Workshop CTB*, 2026. URL https://openreview.net/forum?id=jEudMU2Ppd.

Quan Wei, Siliang Zeng, Chenliang Li, Zhongruo Wang, William Brown, Oana Frunza, Wei Deng, Anderson Schneider, Yuriy Nevmyvaka, Yang Katie Zhao, Alfredo Garcia, and Mingyi Hong. Reinforcing multi-turn reasoning in LLM agents via fine-grained reward structure and credit assignment. *arXiv preprint arXiv:2505.11821v3*, 2026. URL https://arxiv.org/abs/2505.11821v3.

Wantong Xie, Yi-Xiang Hu, Jieyang Xu, Feng Wu, and Xiang-Yang Li. MURKA: Multi-reward reinforcement learning with knowledge alignment for optimization tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve LLMs for optimization modeling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=fsDZwS49uY.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. *Advances in Neural Information Processing Systems*, 36:45548–45580, 2023.

L. J. Zeballos. A constraint programming approach to tool allocation and production scheduling in flexible manufacturing systems. *Robotics and Computer-Integrated Manufacturing*, 26(6):725–743, 2010. doi: 10.1016/j.rcim.2010.04.005.

Long Zhang, Yi Zhuang, and Wei Zhu. Constraint programming based virtual cloud resources allocation model. *International Journal of Hybrid Information Technology*, 6(6):333–344, 2013.

Ruiqing Zhao, Fengzhi Li, Yuan Zuo, Rui Liu, Yansong Liu, Yunfei Ma, Fanyu Meng, and Junlan Feng. Strategy-aware optimization modeling with reasoning LLMs. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, 2026. URL https://openreview.net/forum?id=xEN1n4vu1j.

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. Steporlm: A self-evolving framework with generative process supervision for operations research language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ZrgxU8WMmG.

Supplementary Material

From Demonstrations to Solver-Guided Exploration: Learning to Formulate Constraint Programs from Natural Language

Table of Contents

A Extended Related Work
- A.1 Automated Constraint Programming Modeling
- A.2 Evaluation of Generated Constraint and Optimization Models
- A.3 Solver-Grounded Generation and Revision
- A.4 Post-Training with Verifiable Solver Feedback

B Evaluation Benchmarks and CPMpy Interfaces
- B.1 Benchmarks and Evaluation Units
- B.2 Common Generation Language
- B.3 Empirical Language Screening
- B.4 Benchmark-Specific CPMpy Interfaces
- B.5 Scope of the Unified Evaluation

C Evaluation Protocol and Metrics
- C.1 Execution and Validation
- C.2 Metrics and Aggregation
- C.3 Model and Run Configuration

D Construction of CPARL Training Data
- D.1 Corpus Scope and Five-Tuple Construction
- D.2 Internal Development Set CPARL-DEV-1K
- D.3 From Five-Tuples to CPARL-SFT-8K
- D.4 Trajectory Ordering and Supervision
- D.5 Stage II Problem Pool and On-Policy Rollout Generation
- D.6 Shared Interface and Distinct Learning Signals

E Construction of NL4CP
- E.1 Formal Source Pool and Instance Selection
- E.2 Model-Visible Information and Hidden References
- E.3 Reverse Construction and Quality Control
- E.4 Independent Validation and Benchmark Composition

F Error Analysis of the Baseline and Trained Variants

G Additional Experimental Results and Diagnostics
- G.1 Multi-Sample Performance: Correctness and Executability
- G.2 Single-Attempt Outcome and Failure Decomposition
- G.3 Essential Controlled Comparisons
- G.4 Controlled RL Experiments on a Larger Training Pool
- G.5 RL Training Dynamics

H Complete Methodological Details
- H.1 Stage I: Trajectory Supervised Fine-Tuning
- H.2 Interaction Environment and Action Space
- H.3 Solver-Event Blocks and Rewards
- H.4 Solver-Event GRPO
- H.5 Mixed-Route Objective and Training Configuration
- H.6 Checkpoint Selection and Sweep Interpretation
- H.7 Training Algorithm
- H.8 Worked SE-GRPO Example

I CPARL Rollout Examples
- I.1 Tool-Assisted Modeling Repair
- I.2 Direct Solution via the CoT Route

J Evaluation Prompts
- J.1 CPARL Evaluation Prompt
- J.2 Direct Prompt

A Extended Related Work

A.1 Automated Constraint Programming Modeling

Early work on automated CP modeling encompasses constraint acquisition, reformulation, filtering-algorithm synthesis, and solver selection (Puget, 2004; O'Sullivan, 2010). Conversational modeling (Freuder, 2024) and integration with machine learning (Cappart et al., 2025) extend this agenda while retaining its central requirement: preserving the intended variables, domains, constraints, and objectives. Holy Grail 2.0 decomposes natural-language translation into semantic extraction, relation identification, formulation, code generation, and compilation-based refinement (Tsouros et al., 2023). Such pipelines expose intermediate errors but prescribe the stage order. CPARL instead learns when to produce a model directly and when to gather solver evidence and revise it.

A.2 Evaluation of Generated Constraint and Optimization Models

CP-Bench covers satisfaction and optimization across modeling interfaces (Michailidis et al., 2025); TEXT2ZINC separates execution success from solution accuracy and checks compatibility with instance data (Singirikonda et al., 2025); CPEVAL uses problem-specific verifiers (Song & Cohen, 2025). Paraphrase studies expose sensitivity to equivalent descriptions and distracting context (Pellegrino & Mauro, 2025), while IndusCP adds industrial tasks (Shi et al., 2025). These settings motivate evaluation beyond compilability.

Related mathematical-programming benchmarks broaden different axes. OptiBench includes linear and nonlinear problems with tabular inputs and executable numerical answers (Yang et al., 2025); MIPLIB-NL supplies 223 industrial specifications with aligned formulations and separated models and data (Li et al., 2026c). ORGEval checks formulation equivalence through graph isomorphism under a checkable structural condition (Wang et al., 2026), whereas MM-OptBench grounds textual and visual inputs in exact-solver-verified sources (Li et al., 2026b). Outcome agreement alone need not establish formulation equivalence. Our CP evaluation distinguishes executability from checker acceptance, and NL4CP adds independently constructed tasks involving finite domains, global constraints, and reification. The limits of each retained oracle are explicit in App. B.5.

A.3 Solver-Grounded Generation and Revision

SATLM executes generated declarative specifications with SAT/SMT solvers (Ye et al., 2023); its deductive reliability remains conditional on faithful specification. CPEVAL uses predefined compile-and-repair steps (Song & Cohen, 2025), and ConstraintLLM combines retrieval, tree search, and validation-triggered correction (Shi et al., 2025). StreamLLM instead proposes streamlining constraints for an existing model to reduce search (Voboril et al., 2025). OptArgus audits consistency between descriptions, formulations, and solver implementations (Li et al., 2026a). These systems establish solvers as execution engines and repair triggers. CPARL learns a policy over revisions, solver operations, and termination, conditioned on the model and available diagnostics, rather than fixing their sequence.

A.4 Post-Training with Verifiable Solver Feedback

ORLM (Huang et al., 2025) expands and augments seed examples for supervised instruction tuning. OptMATH (Lu et al., 2025) back-translates optimization instances into descriptions, reconstructs formulations and code, and filters examples by agreement with the source objective values. CALM (Tang et al., 2026) uses brief expert hints to correct a model's own reasoning trajectories for SFT. Its STORM model is subsequently trained through GRPO with interactive code execution and a binary final-answer reward; it can revise code after observing solver output. SAGE (Zhao et al., 2026) combines solver-verified multi-strategy supervision with rewards for format, solver outcomes, and correctness-gated efficiency. Its segment-weighted GRPO scales one trajectory-level advantage using fixed weights for strategy, modeling, and checking tokens. Our two SFT rounds teach direct generation and solver interaction, then target errors identified on a separate development set. SE-GRPO assigns distinct advantages to decision blocks using changes in solver evidence, rather than predefined reasoning-section weights.

SIRL (Chen et al., 2025) rewards output format, execution, and agreement with reference objective values. Its second RL stage adds a bonus for correct outputs whose generated solver models contain advanced modeling structures, such as Big-M constraints. MURKA (Xie et al., 2025) aligns specialized agents with solver-grounded objectives. StepORLM (Zhou et al., 2026) combines solver outcomes and a generative process reward model to construct weighted preferences for DPO: solver correctness takes priority, and process scores distinguish trajectories with matching outcomes. Its iterative loop also refines the process evaluator. OR-R1 (Ding et al., 2026) uses majority-voted objective values as proxy labels for test-time RL on unlabeled problems, together with format and code-validity rewards. These objectives differ from CPARL's use of validation, execution, and propagation changes to assign credit within CP model revision.

Learning from demonstrations followed by RL also appears in ReFT for reasoning (Luong et al., 2024), ReTool for code-interpreter use (Feng et al., 2025a), and DeepSeek-R1 for reasoning post-training (Guo et al., 2025). CPARL grounds both stages in CP model construction and solver verification. Likewise, finer credit assignment extends response-level GRPO (Shao et al., 2024) in turn-level methods and GiGPO (Wei et al., 2026; Feng et al., 2025b). Our contribution is CP-specific: demonstrations and exploration share a modeling interface; verifiable solver evaluations delimit credit blocks; evidence is bound to the evaluated model; and block balancing prevents long edits from dominating short tool decisions. SE-GRPO governs the tool branch within the mixed-route policy, while direct content and the initial route decision additionally learn from terminal correctness.

B Evaluation Benchmarks and CPMpy Interfaces

B.1 Benchmarks and Evaluation Units

All methods generate CPMpy directly. Table 3 summarizes the retained units, generated artifacts, and correctness checks. CP-Bench supplies official CPMpy reference models (Michailidis et al., 2025); TEXT2ZINC supplies descriptions, DZN data, and reference outputs (Singirikonda et al., 2025); CPEVAL provides concrete CSPLib instances and Python verifiers (Song & Cohen, 2025); IndusCP provides industrial descriptions and SAT or objective outcomes (Shi et al., 2025). NL4CP adds independently checked tasks (App. E). Counts refer to our evaluation units and can differ from the source experiments.

Benchmark	Current units	Generated CPMpy contract	Current correctness check
CP-Bench	101 release entries	Complete executable program that prints a JSON assignment	Inject the assignment into the official CPMpy model; re-solve and preserve the reference objective
TEXT2ZINC	110 verified rows	`solve_instance (data_dict)` after deterministic DZN parsing	Enforce output keys and shapes; compare `_objective` for optimization or the full output for satisfaction
CPEVAL	64 instances from 29 classes	`solve_instance (data_dict)` for each concrete pickle instance	Apply the released problem-specific verifier to the returned dictionary
IndusCP	478 cases from 137 problems	Declare `constraints` and `model`; the wrapper solves with OR-Tools	Match released SAT or objective outcomes over at most five instances
NL4CP	147 retained from 212 candidates	Complete CPMpy model under the released schema	Independent feasibility, objective, and output checks

Table 3: Evaluation units and CPMpy interfaces used in our harness. The wrappers adapt instance and output boundaries, not generated source code. Counts can differ from those reported in the source papers because they refer to the retained units evaluated here.

B.2 Common Generation Language

Fixing CPMpy (Guns, 2019) across demonstrations, RL, inference, and baselines controls for language familiarity: the source suites otherwise use MiniZinc (Nethercote et al., 2007),

PyCSP3 (Lecoutre & Szczepanski, 2020), CPMpy, or direct OR-Tools code. CPMpy retains declarative variables, global constraints, and objectives while supporting structured revision and solver diagnostics. Benchmark-specific function boundaries remain as listed in Table 3.

B.3 Empirical Language Screening

A preliminary Qwen3.7-Max generation study favored CPMpy over PyCSP3 on every tested suite (Table 4). These results selected the common representation; they are separate from the main model comparison.

Language	CP-Bench	CPEVAL instance	CPEVAL problem	TEXT2ZINC
CPMpy	71.3	73.0	62.1	37.6
PyCSP3	60.4	59.4	41.4	12.0

Table 4: Preliminary accuracy (%) when Qwen3.7-Max directly generates two candidate modeling languages. CPEVAL-instance averages over instances; CPEVAL-problem requires every instance of a problem to pass. One CPMpy API failure was excluded from the instance-level capability count but counted as failure under the strict problem-level rule. This preliminary study selects the representation and is separate from the main model comparison.

Among CPMpy failures, semantic verification or output mismatch accounted for 65.5% on CP-Bench and 82.4% on TEXT2ZINC; CPEVAL semantic failures (47.1%) exceeded execution errors (35.3%). PyCSP3 execution errors accounted for 61.5% of CPEVAL and 54.7% of TEXT2ZINC failures. This preference is not universal: with DeepSeek-V3.2, CPMpy exceeded MiniZinc on CPEVAL problem-level accuracy (51.7% versus 27.6%), whereas MiniZinc performed better on its native TEXT2ZINC interface (18.6% versus 1.0%). We therefore claim a practical common representation, not general language superiority. CPMpy's OR-Tools-backed root propagation also provides domain tightening and infeasibility rather than the complete constraint-level explanations available in some CP-native systems.

B.4 Benchmark-Specific CPMpy Interfaces

Deterministic adapters load released instance data, execute the submitted artifact, normalize its output, and invoke the retained checker. They neither translate reference formulations into CPMpy nor repair a submission or infer missing constraints. For CPEVAL, output contracts expose the variable keys read by the released verifier when public templates use placeholders; these specify the required output schema, not a reference formulation. Hidden reference models are available only to the checkers. Every method shares one wrapper version within each suite; unsupported source syntax is excluded before sampling. Parser, bootstrap, serialization, and output-field details are maintained as engineering specifications for the code release. For IndusCP, one CPMpy program is generated per problem and reused across its instances. Accuracy is computed over 478 instances, whereas tool calls and generation-and-interaction time are averaged over 137 problem-level trajectories. Shared trajectory costs are counted once; post-generation verification is excluded from these costs.

B.5 Scope of the Unified Evaluation

The harness fixes language, inputs, output contracts, and checkers within a suite; equivalent formulations may still differ in global constraints, propagation, and runtime. It does not reproduce the source systems: CP-Bench omits consensus and self-verification; IndusCP retains 137 rather than 140 problems and omits CARM, tree search, and self-correction; TEXT2ZINC uses all 110 verified rows rather than the 63 NLP4LP problems in its main experiment; CPEVAL excludes Problem 41, generates per concrete instance, and omits self-improvement and Two-Step protocols. Results characterize this unified setting.

Checker strength also differs. CP-Bench and CPEVAL verify assignments; IndusCP primarily matches SAT or a reference objective, and optimized TEXT2ZINC rows compare the objective after checking output shape. Pass@1 accuracy therefore denotes acceptance by the stated oracle, not

proven formulation equivalence. App. F specifies the analysis of recorded execution and modeling errors across the Qwen3.5-27B baseline and our SFT and RL variants.

C Evaluation Protocol and Metrics

C.1 Execution and Validation

Complete candidate artifacts run in isolated subprocesses with fixed timeouts and without credentials or network access. Syntax/import errors, construction failures, timeouts, exceptions, malformed outputs, failed checks, and reference mismatches count as incorrect attempts. Input adapters are preflighted before each evaluation set is frozen: unsupported source formats are declared exclusions; an unexpected adapter failure invalidates the affected evaluation rather than becoming a model error. Acceptance requires both the output contract and the benchmark-specific checker in Table 3.

C.2 Metrics and Aggregation

Instance outcomes. Main comparisons use fixed instance sets Ω_d of size N_d for benchmark d. In decoding run r, let $v_j^{(r)} = 1$ denote checker acceptance and $e_j^{(r)} = 1$ denote successful parsing, model construction, solver return within the execution limits, and output-schema compliance. The same executability definition applies to every model and to both CER@1 and CER@4. Timeouts and output-format failures have $e_j^{(r)} = 0$. Code Executability Rate (CER) counts attempts with $e_j^{(r)} = 1$; Executable-but-Invalid Rate (EIR) counts attempts with $e_j^{(r)} = 1$ and $v_j^{(r)} = 0$. Thus, an EIR failure produces a properly formatted output but does not pass the correctness checker. All rates use the full set of attempts as their denominator. Because $v_j^{(r)} \le e_j^{(r)}$, the three single-attempt rates are

$$\begin{aligned} \mathrm{pass@1}_d^{(r)} &= \frac{100}{N_d}\sum_j v_j^{(r)}, \qquad \mathrm{CER@1}_d^{(r)} = \frac{100}{N_d}\sum_j e_j^{(r)}, \\ \mathrm{EIR}_d^{(r)} &= \frac{100}{N_d}\sum_j e_j^{(r)}(1 - v_j^{(r)}) = \mathrm{CER@1}_d^{(r)} - \mathrm{pass@1}_d^{(r)}\,. \end{aligned} \tag{6}$$

Sums range over Ω_d. Non-executable, executable but invalid, and accepted outcomes exhaust the same set of attempts. The decomposition must be computed within each run before summarizing means and sample standard deviations; the standard deviation of EIR cannot be obtained by subtracting the standard deviations of CER and pass@1 accuracy.

Four-attempt outcomes. For attempt k on instance j, let $v_{j,k}$ and $e_{j,k}$ denote checker acceptance and executability under the same definitions. Following the pass@k convention (Chen et al., 2021), the four-attempt metrics are

$$\begin{aligned} \mathrm{pass@4}_d &= \frac{100}{N_d}\sum_{j\in\Omega_d} \max_{1\le k\le 4} v_{j,k}, \\ \mathrm{CER@4}_d &= \frac{100}{N_d}\sum_{j\in\Omega_d} \max_{1\le k\le 4} e_{j,k}. \end{aligned} \tag{7}$$

Thus, pass@4 requires at least one checker-accepted program among four attempts; CER@4 requires at least one executable program. These are success rates over four attempts, not averages of four pass@1 or CER@1 scores. App. G.1 reports the existing measurements.

Solver use and aggregation. Route fractions and Calls are measured over all attempted generation episodes, including failures and timeouts. Calls counts agent-requested solver operations; terminal checking and solving the final submitted program are not additional agent requests. Direct trajectories therefore contribute zero agent-requested calls. Malformed route decisions are reported separately rather than silently assigned to a legal route. Shared trajectory costs are counted once. Hard token, call and time limits describe the evaluation budget; equal limits do not imply equal realized cost. No end-to-end speedup claim is made from call counts alone. Across benchmarks, micro rates weight by N_d (900 instances in total); macro rates give the five benchmarks equal weight. Explicit

problem-level accuracy, such as the language screen, requires all instances of a problem to pass and weights problems equally.

Results are displayed to one decimal place; aggregates and derived rates are calculated before display rounding.

Repeated decoding. The CPARL, RL-ablation, Qwen3.5-27B direct-prompt, and other external-baseline summaries use five evaluations; SFT checkpoints and SFT controls use three. Each benchmark score and each micro/macro aggregate is computed within a run before its mean and sample standard deviation are calculated. These deviations describe variation across evaluation settings and seeds, not independent training runs. Checkpoint sweeps retain one evaluation per checkpoint as a separate record of training progress. Evaluation counts are stated for each result.

C.3 Model and Run Configuration

App. J provides the full CPARL evaluation and Direct Prompt templates, including their system instructions and user inputs.

We directly prompt DeepSeek-V4-Pro, Qwen-3.6-Max-Preview, GLM-5.1, GPT-5.4, Gemini-3.1-Pro-Preview, Claude-4.6-Sonnet, and the unadapted Qwen3.5-27B backbone. We also evaluate models fine-tuned for general optimization modeling: ORLM-LLaMA3-8B (Huang et al., 2025), OptMATH-Qwen2.5-7B/32B (Lu et al., 2025), and SIRL-Qwen2.5-7B/32B (Chen et al., 2025). Internal controls use the same backbone and CPMpy interface (App. G.3).

All benchmark evaluations use a unified execution and verification environment with Python 3.12.12, CPMpy 0.10.1, and OR-Tools 9.14.6206. NL4CP solutions are checked with the XCSP3 solution checker 2.6.0.

Direct Prompt and General Fine-Tuned methods share an output-token limit. The 300-task RL report specifies training seed 42 and evaluation worker seed 1234. *Full CPARL*, *terminal-correctness GRPO*, and *Trajectory-level credit* choose routes freely; *Tool-only SE-GRPO* fixes the tool route in training and evaluation. The controls start from the same second-round SFT model and use the same reported problem pool, rollout count, and update schedule. Equal configured budgets do not imply equal realized calls, tokens, or time. Key settings are in App. H.1 and H.5; checkpoint comparison and selection are described in App. H.6.

CPARL and its RL ablation variants are each evaluated five times using the same checkpoint across evaluations. The first evaluation uses temperature 0 and worker seed 1234; the remaining four use temperature 0.4 and worker seeds 2345, 3456, 4567, and 5678, respectively. Pass@1 reports the mean and sample standard deviation over all five evaluations. Pass@4 uses the four temperature-0.4 evaluations, counting an instance as solved if at least one attempt passes its checker.

These final evaluations allow at most eight assistant turns, including initial model construction, with one action per turn. Generation is capped at 16,384 new tokens per turn and 65,536 per trajectory, subject to a 32,768-token context with 256 tokens reserved. Interaction tool budgets are 8, 10, and 20 seconds for compile, propagate, and solve, respectively. Final verification allows 300 seconds of solving per instance. Sandboxed execution receives an additional 30-second startup allowance.

D Construction of CPARL Training Data

CPARL-SFT-8K contains 8,156 fixed demonstration records used in two successive LoRA SFT rounds (5,600 and 2,556). CPARL-RL-Pool supplies 300 problems for subsequent on-policy RL. The two SFT rounds are distinct from the framework's SFT and RL stages.

D.1 Corpus Scope and Five-Tuple Construction

The first SFT round contains 5,600 problems adapted from instances and examples in CSPLib, the CPMpy example library, NLP4LP, and the XCSP3 competition website, with source counts given in Table 5. A team of four Ph.D. researchers and postdoctoral fellows in optimization, assisted by GPT-5.6, selects modeling problems and screens both their descriptions and mathematical formulations against the evaluation tasks, seeking structural differences rather than paraphrases. Programs

are executed and checked against the stated constraints and reference objective. Human experts and AI assistants also inspect agreement among the description, formulation, code, and answer. Expert-assisted demonstrations undergo the same checks; successful execution alone is insufficient. Each construction record contains five logical components: an identifier, a self-contained description, a CP formulation, executable CPMpy code, and a reference answer. The formulation specifies variables, domains, constraints, and any objective; the answer supports feasibility and objective checks.

Table 5: First-round SFT source distribution and public resource references.

Resource	Problems	Current public reference	Published license
CSPLib	1,448	CSPLib website	CC BY 4.0
CPMpy examples	1,340	v1.0.0 (July 18, 2026)	Apache-2.0
NLP4LP	230	May 13, 2024 update	CC BY-NC 4.0
XCSP3 instances	2,582	2026 competition entry	MIT
Total	5,600	—	—

Counts refer to the adapted training problems grouped by their original source.

D.2 Internal Development Set CPARL-Dev-1K

We construct an internal development set, CPARL-Dev-1K, to select the first-round mixture, construct second-round SFT data, and select the final RL checkpoint.

CPARL-Dev-1K comprises 1,000 problems adapted from CSPLib and XCSP competition instances: 650 CSPs and 350 COPs. It was constructed before the first-round SFT evaluation by changing problem contexts, constraints, and parameter ranges. Each record contains a natural-language description, a CP formulation, CPMpy code, and a reference answer. Human reviewers verify the constraints and, where applicable, the optimization objective. We screen descriptions and CP formulations against the five evaluation benchmarks to exclude duplicates, retaining problems with low textual similarity and non-identical CP formulations.

At construction time, CPARL-Dev-1K is selected to differ semantically and structurally from the initial SFT corpus. Failures of first-round SFT supply problems adapted into the second round; failures of the resulting mixed SFT checkpoint supply adaptations for RL. The selected mixture uses 40% tool and 60% CoT demonstrations; the selected RL checkpoint is step 135. The development set is excluded from benchmark aggregates.

D.3 From Five-Tuples to CPARL-SFT-8K

CoT Training Data Construction. DeepSeek-V4-Pro receives each problem description and makes up to four solution attempts. Each attempt produces reasoning and a candidate program. We retain the first attempt whose CPMpy program passes the execution-based correctness checks. Human experts, assisted by AI, then organize the retained reasoning into a CoT of at most five steps. If all four attempts fail, Codex and human experts solve the problem, verify the program, and prepare a CoT under the same length limit. This produces one CoT demonstration for each of the 5,600 problems. The four-attempt acceptance procedure is separate from the pass@1 metric used to compare trained models.

Tool-Trajectory Data Construction. For the same 5,600 problems, DeepSeek-V4-Pro makes up to four tool-interaction attempts, each limited to eight assistant turns (not eight individual tool calls). We retain the first correct trajectory with authentic execution feedback; if all fail, human experts construct and verify one through actual tool execution.

Difficulty ranking. We rank all $N = 5{,}600$ problems using four features. Constraint count C is the number of distinct constraint rules in the CP formulation: repeated applications of one indexed rule count once, while variable domains and the optimization objective are excluded. Variable scale

V counts scalar decision variables after expanding arrays, excluding constants, ordinary program temporaries, and derived expressions. Description length L counts characters after normalizing line endings, collapsing whitespace, and trimming leading and trailing whitespace. Code length K counts characters in the verified CPMpy code after removing code fences, Python comments, and blank lines; retained lines are joined by a single newline. We use GPT-5.6 to extract C and V from the supplied formulation and code, and a program to compute lengths and the global ranking.

For feature $f \in \{C, V, L, K\}$, let $x_{i,f}$ be its value for problem i. We normalize each feature by its ascending rank across the full corpus, assigning the average rank to ties:

$$P_{i,f} = \frac{\mathrm{rank}_{\mathrm{avg}}(x_{i,f}) - 1}{N - 1}, \qquad D_i = 100\left(0.30P_{i,C} + 0.30P_{i,V} + 0.20P_{i,L} + 0.20P_{i,K}\right). \tag{8}$$

The weights are fixed heuristic choices. Problems are ordered by decreasing D_i, without rounding scores before sorting. Ties are broken by decreasing C, then V, then K, and finally by lexicographically increasing problem ID. For each tool proportion from 0% to 100% in increments of 10%, the corresponding prefix of this ordering selects tool demonstrations; the remaining problems use CoT demonstrations. The score is a routing heuristic, not a guarantee of solver complexity.

Ranking prompt. We supply the following prompt with the feature definitions and scoring rule above. Its input contains all 5,600 records; its final output contains only their ordered IDs.

Difficulty Ranking Prompt

```
You are a constraint-programming modeling expert. Rank all 5,600 input
  problems from hardest to easiest using the feature definitions and scoring
  rule above.

Each record contains a problem ID, a natural-language description, a CP
  mathematical model, and verified CPMpy code:

problem_id: {problem_id}
description: {description}
cp_model: {cp_model}
code: {code}

Extract C and V using only the supplied records. Compute L, K, percentile
  ranks, and weighted scores programmatically across all 5,600 problems, using
   weights 0.30, 0.30, 0.20, and 0.20 for C, V, L, and K, respectively. Apply
  the specified tie-breaking rule.

Resolve missing or conflicting information with human review before producing
  the ranking; do not guess.

Return only one JSON array of exactly 5,600 problem IDs, ordered from hardest
  to easiest, with every input ID appearing exactly once. Do not include
  scores, explanations, or additional fields.
```

First-round mixtures. We consider eleven first-round mixtures from the outset, with tool proportions of 0%, 10%, . . ., 100%. Each mixture contains one demonstration for each of the same 5,600 problems: the hardest problems receive tool demonstrations, and the remainder receive CoT demonstrations. A route-decision field supervises this choice. Qwen3.5-27B is trained separately on each mixture using LoRA SFT. Based on the performance of all eleven models on CPARL-DEV-1K, we select the 40% tool / 60% CoT mixture for second-round training. Table 6 reports pass@1 accuracy on the five evaluation benchmarks for all eleven mixtures.

Second-round targeted training. Human experts, assisted by AI, inspect the selected checkpoint's CPARL-Dev-1k failures and construct 2,556 adaptations addressing recurring error causes and problem types. Both descriptions and mathematical formulations are changed; these tasks do not originate from the evaluation benchmarks. We independently rank the 2,556 new problems using the same feature definitions, weights, tie-breaking rule, and ranking prompt as in the first round, replacing the corpus size in the prompt and Equation 8 with $N = 2{,}556$. Feature ranks are normalized

Tool / CoT	CP-Bench	TEXT2ZINC	CPEVAL	IndusCP	NL4CP	Micro Avg.	Macro Avg.
100% / 0%	$39.3_{\pm 1.5}$	$18.2_{\pm 0.9}$	$35.9_{\pm 3.1}$	$19.6_{\pm 1.5}$	$24.9_{\pm 1.4}$	$23.7_{\pm 1.1}$	$27.6_{\pm 0.8}$
90% / 10%	$35.3_{\pm 1.5}$	$18.8_{\pm 2.3}$	$35.4_{\pm 4.8}$	$19.0_{\pm 0.8}$	$24.3_{\pm 0.8}$	$22.9_{\pm 0.4}$	$26.6_{\pm 1.3}$
80% / 20%	$38.6_{\pm 1.7}$	$22.1_{\pm 1.9}$	$42.2_{\pm 4.7}$	$20.2_{\pm 0.3}$	$25.2_{\pm 1.8}$	$24.9_{\pm 0.5}$	$29.7_{\pm 1.0}$
70% / 30%	$37.3_{\pm 2.1}$	$24.9_{\pm 2.3}$	$42.2_{\pm 7.8}$	$21.6_{\pm 1.3}$	$24.5_{\pm 3.8}$	$25.7_{\pm 1.2}$	$30.1_{\pm 2.8}$
60% / 40%	$38.3_{\pm 1.1}$	$27.6_{\pm 1.4}$	$44.8_{\pm 2.4}$	$20.6_{\pm 0.3}$	$22.5_{\pm 0.7}$	$25.4_{\pm 0.6}$	$30.7_{\pm 1.0}$
50% / 50%	$42.6_{\pm 5.5}$	$31.5_{\pm 3.4}$	$45.8_{\pm 2.4}$	$22.1_{\pm 0.8}$	$25.2_{\pm 3.0}$	$27.7_{\pm 1.2}$	$33.4_{\pm 1.5}$
40% / 60%	$46.2_{\pm 3.7}$	$35.5_{\pm 5.5}$	$51.0_{\pm 7.2}$	$22.2_{\pm 0.2}$	$26.3_{\pm 3.9}$	$\mathbf{29.2_{\pm 1.0}}$	$\mathbf{36.2_{\pm 2.0}}$
30% / 70%	$40.9_{\pm 3.2}$	$32.4_{\pm 3.2}$	$49.0_{\pm 3.3}$	$22.5_{\pm 1.6}$	$26.5_{\pm 1.8}$	$28.3_{\pm 1.5}$	$34.3_{\pm 1.9}$
20% / 80%	$38.9_{\pm 0.6}$	$30.6_{\pm 2.6}$	$49.5_{\pm 4.8}$	$21.6_{\pm 1.6}$	$23.1_{\pm 0.7}$	$26.9_{\pm 0.5}$	$32.8_{\pm 1.2}$
10% / 90%	$39.3_{\pm 2.1}$	$29.4_{\pm 2.3}$	$43.2_{\pm 3.9}$	$21.1_{\pm 0.8}$	$23.4_{\pm 1.4}$	$26.1_{\pm 0.3}$	$31.3_{\pm 0.2}$
0% / 100%	$41.9_{\pm 0.6}$	$28.2_{\pm 2.4}$	$45.8_{\pm 2.4}$	$19.7_{\pm 0.8}$	$27.0_{\pm 1.0}$	$26.3_{\pm 0.7}$	$32.5_{\pm 0.5}$

Table 6: SFT supervision ratios (Tool / CoT), with pass@1 accuracy in percent. Entries report mean $\pm$ sample standard deviation over three evaluations (App. C.2). Benchmark sizes are 101, 110, 64, 478, and 147 in column order (900 total); micro averages pool instances and macro averages weight benchmarks equally.

within this new batch. The hardest 1,022 problems receive tool demonstrations, and the remaining 1,534 receive CoT demonstrations. Continued LoRA SFT uses 1,022 tool trajectories and 1,534 CoT demonstrations (40.0% and 60.0%). Table 7 gives the resulting corpus counts.

Corpus composition and observed lengths. Table 7 summarizes the selected training records. Problems with an optimization objective are counted as COPs; feasibility-only problems are counted as CSPs. Table 8 reports broad modeling categories, assigned by matching available problem labels and description patterns; unmatched records are grouped as other. Each record belongs to one category. Tool-trajectory length counts assistant messages, including initialization and final submission, but not user messages or tool observations. CoT length counts the numbered reasoning steps.

SFT round	Tool	CoT	CSP	COP	Total
First (selected mixture)	2,240	3,360	2,461	3,139	5,600
Second	1,022	1,534	568	1,988	2,556
Total	3,262	4,894	3,029	5,127	8,156

Table 7: Training-record composition of CPARL-SFT-8K. Route (Tool/CoT) and task type (CSP/COP) are separate partitions of the same records. Counts exclude unused candidate demonstrations and alternative first-round mixtures.

Modeling category	First round	Second round
Scheduling and timetabling	920	426
Assignment and load balancing	260	284
Packing and knapsack	211	284
Selection, covering, and facility location	151	426
Routing, transport, and flows	169	142
Graph and network constraints	319	142
Puzzles and combinatorial design	1,120	284
Sequencing and permutations	378	284
Production and resource planning	347	0
Arithmetic and set partitioning	432	284
Other constraint models	1,293	0
Total	5,600	2,556

Table 8: Training records by broad modeling category.

Mean micro/macro pass@1 increases from 29.2%/36.2% to 32.5%/41.1%, with gains on all five benchmarks (Table 2). *One-stage-SFT R1&R2* in Table 2 uses the same 8,156-record corpus and

Assistant turns	First-round tool records	Second-round tool records
3	617	0
4	659	680
5	331	342
6	373	0
7	141	0
8	119	0
Total	2,240	1,022
Mean	4.61	4.33

Table 9: Observed assistant-turn counts in the retained tool demonstrations.

Reasoning steps	First-round CoT records	Second-round CoT records
1	18	0
2	415	0
3	2,630	1,534
4	287	0
5	10	0
Total	3,360	1,534
Mean	2.96	3.00

Table 10: Observed numbers of numbered reasoning steps in the retained CoT demonstrations. Shorter one- and two-step records are included in both counts and means.

reaches 27.6%/35.0%. This distinguishes staged continuation from pooling the corpus. The two SFT checkpoints use the same five benchmark populations (900 instances, including 147 NL4CP instances). The SFT scores summarize three evaluations; micro and macro averages are calculated per run before summarization (App. C.2).

D.4 Trajectory Ordering and Supervision

Records preserve the initial route decision and the causal order of policy actions, model states, and authentic feedback; CoT records contain no invented tool responses. Policy-owned route decisions, reasoning, code, edits, requests, and terminal submission are supervised. Environment-owned contracts, observations, diagnostics, delimiters, and padding are conditioning context with zero loss. At tokenization, message ownership determines the policy-token mask used in the SFT loss in Sec. 4.1. The submitted final code must match the active validated model. Stage I, Stage II, and inference use the same message and action contract; field schemas, literal examples, and legacy-format handling belong to the accompanying engineering documentation.

D.5 Stage II Problem Pool and On-Policy Rollout Generation

For the main configuration, the pool contains 300 reformulated failures of the mixed SFT checkpoint on CPARL-Dev-1k: 180 satisfaction and 120 optimization problems. All are used for RL; failed SFT responses are not targets. All 300 problems undergo manual review, focusing on whether their reference code runs in the RL execution environment and correctly represents the stated constraints and optimization objectives, where applicable. The RL pool contains no exact duplicates of problems in either SFT round. A task supplies the visible description and instance data, while its reference, checker, output mapping, and limits remain environment-only. Reference solutions, objectives, checker internals, and scalar reward components are absent from policy observations; requested tools return normalized diagnostics.

Each epoch shuffles all 300 problems. An update uses eight distinct problems, with eight fresh rollouts per problem from a frozen behavior snapshot. Routes are sampled naturally; no 60:40 quota is imposed during RL. The direct route submits code to the terminal checker, while the tool route obtains diagnostics through actual execution. Once the group is complete, we compute the common binary outcome advantage across both routes and retain the original event ledger and advantages for

the tool trajectories. Each collected group is used for one update and is not replayed after a policy refresh. Ordinary policy failures are retained.

Policy-caused invalid edits, compilation errors, timeouts, premature termination, and failed validation receive their prescribed returns; only infrastructure faults are discarded and resampled. The reported 225 updates allocate 14,400 rollout slots per condition. The SFT continuation and RL pool share development parents.

D.6 Shared Interface and Distinct Learning Signals

Both stages support direct formulation and solver-assisted revision. CoT demonstrations teach concise modeling; tool demonstrations teach actions and feedback-conditioned revision, including when to use them. SFT imitates these decisions at demonstrated observations (Sec. 4.1); RL compares new attempts from the current policy using the mixed objective (Equation 5). Shared terminal outcomes train direct content and route choice, while tool-event credit evaluates solver-assisted decisions. The full SFT-to-RL comparison measures the change after Stage II as a whole; the same-reward trajectory-credit control in App. G.3 addresses the role of event/depth-relative attribution.

E Construction of NL4CP

NL4CP tests recovery of a formally specified constraint problem from natural language. Its hidden source instances supply an independent semantic reference for evaluating generated CPMpy models.

E.1 Formal Source Pool and Instance Selection

The source pool contains 212 candidates from the 2024 and 2025 XCSP competitions[1], covering 62 named families: 138 CSPs and 74 COPs (65 minimization and 9 maximization). Formal XML specifies indexed variables, domains, global/table constraints, constants, and objectives; competition artifacts support checking assignments and objective values. Quality control retains 147 of the 212 candidates; all retained tasks are evaluated. Figure 3 describes the source pool's application coverage, not a balanced quota or the final evaluated distribution.

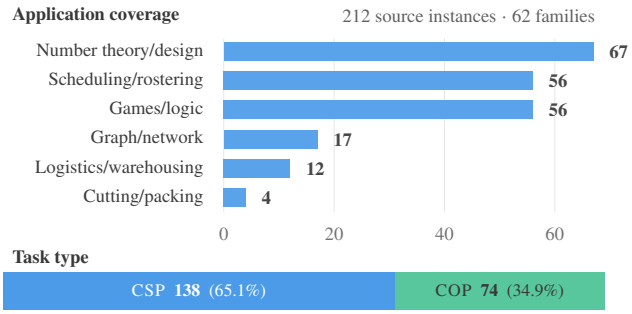

Figure 3: NL4CP source-pool composition; quality control retains 147 of 212 candidates.

E.2 Model-Visible Information and Hidden References

Each tested model receives an identifier and a self-contained description specifying variable families, index ranges, domains, constraints, numerical data, any objective, and the required output. No external data file is supplied. Original variable identifiers and indexing conventions permit deterministic mapping to the hidden formal instance. XML, reference solutions, and reference objective values remain evaluator-only. All methods generate CPMpy directly: evaluation uses neither LLM-based translation into another language nor source-code similarity to a reference program. File layouts and release manifests are maintained as engineering documentation.

[1] http://www.xcsp.org/competitions/

E.3 Reverse Construction and Quality Control

Construction proceeds from the formal instance as semantic authority. We select tasks across CSP/COP types, competition years, scales, and families; deterministically parse their formal components; and assemble an instance-specific fidelity checklist. An LLM writes a natural-language task preserving identifiers, scopes, constants, and output requirements. Human auditors compare it against the checklist and XML, emphasizing quantified index conditions, global constraints, objective direction, and boundary values. An accepted description is bound to its hidden source and evaluator, and a reference CPMpy program is executed through the test interface. Neither generated models nor outputs are manually repaired during evaluation. Reference outputs must pass, and deliberately corrupted assignments/objectives test rejection of representative errors.

E.4 Independent Validation and Benchmark Composition

For CSPs, the evaluator maps the returned variables to the hidden XML and uses the competition checker to test domains and constraints. For COPs, all reference solutions are verified optima. The evaluator additionally recomputes the objective independently from the feasible assignment and compares it with the reference optimal value.

Release materials are intended to include natural-language tasks, provenance and licensing, deterministic evaluators, and reference CPMpy models, with access rules for hidden XML and solutions/bounds. Evaluated systems must not receive scoring references. The reported results cover all 147 tasks retained from the 212 source candidates after quality control.

F Error Analysis of the Baseline and Trained Variants

We compare the supplied error annotations for the Qwen3.5-27B direct-prompt baseline, *Mixed SFT R1*, *Mixed-SFT R1+R2*, *terminal-correctness GRPO*, and *full CPARL*. The baseline precedes CP-specific training; both RL versions use the main 300-task pool at step 135. Each model contributes one full pass@1 evaluation at worker seed 1234, with error labels assigned to failed attempts.

Figure 2 shows the composition of the six main error-label categories. For model m and category c, its share is $100n_{m,c}/\sum_{k\in\mathcal{C}} n_{m,k}$, where $n_{m,c}$ is the count in Table 11 and $\mathcal{C}$ contains the six main categories, excluding the two supplementary rows. Each bar sums to 100% before rounding. These are shares of labels, not mutually exclusive shares of failed instances: one attempt can receive multiple labels. Table 11 separately retains incidence rates over all 900 instances.

Table 11 groups the failures into six types:

- **Program execution:** no usable program, syntax/import/API errors, or runtime exceptions.
- **Timeout or budget:** the recorded execution or interaction limit is reached.
- **Output format:** missing fields, incorrect shapes, or an invalid output interface.
- **Constraints:** missing or incorrect restrictions, including logical links and global-constraint semantics.
- **Variables and domains:** incorrect variables, bounds, indices, or mappings from instance data.
- **Objective or task type:** an incorrect objective expression or direction, or confusion between satisfaction and optimization.

An attempt may contain several errors and contribute to multiple categories. Other documented causes and cases with insufficient evidence are listed separately. Missing archived material is not itself a model error.

Table 11 shows a large reduction in program-execution errors: from 71.7% for the backbone to 35.3% after SFT R1, 29.7% after SFT R1+R2, and 26.9% for CPARL. The largest decrease occurs in the first SFT round, suggesting that demonstrations address many program-construction and execution errors.

Relative to terminal-correctness GRPO, CPARL has fewer execution and timeout/budget errors, as well as fewer errors in each of the three semantic modeling categories. The largest count reductions are in variables and domains (123 to 102) and timeout/budget failures (230 to 214).

Error type	Qwen3.5-27B	*Mixed SFT R1*	*Mixed-SFT R1+R2*	*GRPO*	*CPARL*
Program execution	645 (71.7)	318 (35.3)	267 (29.7)	255 (28.3)	242 (26.9)
Timeout or budget	117 (13.0)	204 (22.7)	214 (23.8)	230 (25.6)	214 (23.8)
Output format	59 (6.6)	70 (7.8)	48 (5.3)	35 (3.9)	44 (4.9)
Constraints	49 (5.4)	135 (15.0)	103 (11.4)	136 (15.1)	124 (13.8)
Variables and domains	59 (6.6)	131 (14.6)	122 (13.6)	123 (13.7)	102 (11.3)
Objective or task type	34 (3.8)	87 (9.7)	73 (8.1)	69 (7.7)	61 (6.8)
Other documented causes	7 (0.8)	8 (0.9)	7 (0.8)	16 (1.8)	18 (2.0)
Insufficient evidence	6 (0.7)	3 (0.3)	6 (0.7)	3 (0.3)	3 (0.3)

Table 11: Error counts and micro percentages, shown as count (%), from one full pass@1 evaluation per model at worker seed 1234. Each percentage uses all 900 test instances, including successful attempts. Categories may overlap.

Execution and timeout/budget errors remain CPARL's most frequent labels (26.9% and 23.8%), while constraints are its largest semantic error category (13.8%). This distribution identifies both budget-efficient solver interaction and faithful CP formulation as remaining challenges: an executable program can still encode the wrong constraints, domains, or objective.

G Additional Experimental Results and Diagnostics

App. G.1 reports the measured four-attempt correctness and executability results. App. G.2 gives the single-attempt execution measurements and failure decomposition. App. G.3 records the SFT comparisons and the reported RL controls: same-reward trajectory-level credit and *tool-only SE-GRPO*. App. G.4 reports the preliminary 1,000-task RL-pool comparison. Metric definitions and aggregation rules appear in App. C.2.

G.1 Multi-Sample Performance: Correctness and Executability

Pass@4 measures whether at least one of four generated programs passes the benchmark checker; CER@4 measures whether at least one returns a solver result within the execution limits with valid output format, using the same execution definition as CER@1. Tables 12 and 13 report these metrics, respectively, for seven directly prompted LLMs, five models fine-tuned for general optimization modeling, and CPARL, using the same four attempts per model. Benchmark sizes are 101, 110, 64, 478, and 147 in row order (900 total). Micro averages pool instances; macro averages weight the five benchmarks equally.

Each entry summarizes one four-attempt set; no means or standard deviations across repeated four-attempt sets are reported. CPARL uses the four temperature-0.4 evaluations of step 135 in App. C.3; its temperature-0 evaluation is excluded. External baselines also use temperature 0.4 and the same per-attempt limits as their pass@1 evaluations.

	Direct Prompt							General Fine-Tuned					Ours
Dataset	DeepSeek V4-Pro	Qwen 3.6-Max	Qwen 3.5-27B	GLM 5.1	GPT 5.4	Gemini 3.1-Pro	Claude 4.6-Sonnet	ORLM LLaMA3-8B	OptMATH Qwen2.5-7B	OptMATH Qwen2.5-32B	SIRL Qwen2.5-7B	SIRL Qwen2.5-32B	*CPARL* Qwen3.5-27B
CP-Bench	74.3	75.3	5.0	69.3	71.3	76.2	61.4	0.0	0.0	2.0	2.0	13.9	66.3
TEXT2ZINC	49.1	45.5	13.6	51.8	48.2	58.2	52.7	0.0	0.9	0.9	0.0	15.5	60.9
CPEVAL	82.8	82.8	18.8	82.8	81.3	79.7	79.7	0.0	6.3	0.0	0.0	25.0	82.8
IndusCP	39.5	46.4	2.7	36.4	46.2	43.1	38.9	0.0	1.1	0.0	6.5	8.6	33.1
NL4CP	51.0	51.7	6.1	45.6	55.8	61.9	57.1	0.0	1.4	0.7	0.7	9.5	47.6
Micro Avg.	49.6	53.0	6.0	46.8	53.3	54.3	49.0	0.0	1.3	0.4	3.8	11.3	46.1
Macro Avg.	59.3	60.3	9.2	57.2	60.6	63.8	58.0	0.0	1.9	0.7	1.8	14.5	58.1

Table 12: Pass@4 accuracy (%): at least one checker-accepted solution among four attempts.

Gemini-3.1-Pro has the highest micro and macro pass@4 scores, 54.3% and 63.8%; SIRL-Qwen2.5-32B leads the general fine-tuned group at 11.3% and 14.5% (Table 12). Their macro CER@4 scores are higher, at 84.2% and 27.2%, respectively (Table 13). Thus, a runnable candidate within four attempts does not ensure a checker-accepted solution.

	Direct Prompt							General Fine-Tuned					Ours
Dataset	DeepSeek V4-Pro	Qwen 3.6-Max	Qwen 3.5-27B	GLM 5.1	GPT 5.4	Gemini 3.1-Pro	Claude 4.6-Sonnet	ORLM LLaMA3-8B	OptMATH Qwen2.5-7B	OptMATH Qwen2.5-32B	SIRL Qwen2.5-7B	SIRL Qwen2.5-32B	*CPARL* Qwen3.5-27B
CP-Bench	93.1	94.1	14.9	88.1	96.0	96.0	84.2	1.0	0.0	6.9	20.8	25.7	74.3
TEXT2ZINC	90.0	85.5	17.3	90.0	90.0	92.7	88.2	0.0	2.7	1.8	0.0	38.2	73.6
CPEVAL	93.8	93.8	20.3	93.8	96.9	96.9	93.8	0.0	25.0	1.6	3.1	34.4	90.6
IndusCP	52.3	62.3	6.1	51.9	63.4	61.7	54.0	0.0	2.5	1.1	14.2	15.1	58.4
NL4CP	68.7	76.9	11.6	63.3	76.9	73.5	77.6	0.0	21.8	4.8	5.4	22.5	57.1
Micro Avg.	67.1	73.3	10.3	65.4	74.9	73.8	68.2	0.1	7.0	2.5	11.0	21.7	64.1
Macro Avg.	79.6	82.5	14.0	77.4	84.6	84.2	79.5	0.2	10.4	3.2	8.7	27.2	70.8

Table 13: CER@4 (%): at least one executable program among the same four attempts.

CPARL reaches 46.1% micro and 58.1% macro pass@4 accuracy, compared with 6.0% and 9.2% for the unadapted backbone (Table 12). Its macro CER@4 is 70.8% (Table 13), leaving a macro-averaged 12.7% of tasks with a runnable candidate but no checker-accepted solution, and 29.2% with no runnable candidate.

G.2 Single-Attempt Outcome and Failure Decomposition

All outcome rates use the executability definition in App. C.2. An attempt is checker-accepted, executable but rejected (EIR), or non-executable; $\mathrm{EIR} = \mathrm{CER@1} - \mathrm{pass@1}$. Timeouts and output-format failures belong to the non-executable group. Table 14 reports CER@1, and Table 15 reports the corresponding outcome rates where available.

	Direct Prompt						General Fine-Tuned					Full Method
Dataset	DeepSeek V4-Pro	Qwen 3.6-Max	GLM 5.1	GPT 5.4	Gemini 3.1-Pro	Claude 4.6-Sonnet	ORLM LLaMA3-8B	OptMATH Qwen2.5-7B	OptMATH Qwen2.5-32B	SIRL Qwen2.5-7B	SIRL Qwen2.5-32B	CPARL Qwen3.5-27B
CP-Bench	$73.1_{\pm 5.2}$	$84.8_{\pm 2.7}$	$68.7_{\pm 5.2}$	$84.8_{\pm 10.5}$	$87.9_{\pm 5.9}$	$71.1_{\pm 6.6}$	$1.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$6.9_{\pm 0.0}$	$20.8_{\pm 0.0}$	$25.7_{\pm 0.0}$	$65.5_{\pm 2.7}$
TEXT2ZINC	$69.1_{\pm 3.5}$	$57.3_{\pm 3.9}$	$65.6_{\pm 4.2}$	$77.1_{\pm 2.1}$	$78.7_{\pm 2.6}$	$80.0_{\pm 1.4}$	$0.0_{\pm 0.0}$	$2.7_{\pm 0.0}$	$1.8_{\pm 0.0}$	$0.0_{\pm 0.0}$	$38.2_{\pm 0.0}$	$58.7_{\pm 2.8}$
CPEVAL	$77.2_{\pm 3.1}$	$83.1_{\pm 2.8}$	$75.9_{\pm 7.2}$	$84.1_{\pm 2.3}$	$90.9_{\pm 3.4}$	$84.1_{\pm 2.8}$	$0.0_{\pm 0.0}$	$25.0_{\pm 0.0}$	$1.6_{\pm 0.0}$	$3.1_{\pm 0.0}$	$34.4_{\pm 0.0}$	$80.0_{\pm 4.7}$
IndusCP	$30.9_{\pm 2.9}$	$45.5_{\pm 2.8}$	$33.1_{\pm 4.5}$	$46.3_{\pm 3.3}$	$42.7_{\pm 1.4}$	$46.9_{\pm 1.6}$	$0.0_{\pm 0.0}$	$1.1_{\pm 0.7}$	$0.8_{\pm 0.5}$	$10.4_{\pm 1.0}$	$12.4_{\pm 1.1}$	$50.4_{\pm 3.6}$
NL4CP	$46.0_{\pm 1.6}$	$51.6_{\pm 2.7}$	$41.5_{\pm 1.6}$	$53.1_{\pm 3.8}$	$52.2_{\pm 1.7}$	$59.9_{\pm 2.6}$	$0.0_{\pm 0.0}$	$21.8_{\pm 0.0}$	$4.8_{\pm 0.0}$	$5.4_{\pm 0.0}$	$21.4_{\pm 0.4}$	$44.8_{\pm 5.7}$
Micro Avg.	$46.1_{\pm 1.8}$	$55.0_{\pm 1.6}$	$45.5_{\pm 2.4}$	$58.2_{\pm 2.5}$	$57.2_{\pm 1.4}$	$58.4_{\pm 1.5}$	$0.1_{\pm 0.0}$	$6.3_{\pm 0.4}$	$2.3_{\pm 0.3}$	$9.0_{\pm 0.5}$	$20.1_{\pm 0.6}$	$54.3_{\pm 2.5}$
Macro Avg.	$59.3_{\pm 1.2}$	$64.5_{\pm 1.2}$	$57.0_{\pm 1.7}$	$69.1_{\pm 2.5}$	$70.5_{\pm 2.0}$	$68.4_{\pm 2.1}$	$0.2_{\pm 0.0}$	$10.1_{\pm 0.1}$	$3.2_{\pm 0.1}$	$8.0_{\pm 0.2}$	$26.4_{\pm 0.2}$	$59.9_{\pm 2.2}$

Table 14: Instance-level CER@1 (%). Entries report mean $\pm$ sample standard deviation across decoding evaluations under the common execution and output requirements in App. C.2. CPARL is evaluated five times.

Method	Pass@1 accuracy ↑	Executable but invalid (EIR) ↓	Non-executable $(100 - \mathrm{CER@1})$ ↓
Gemini-3.1-Pro	53.8	16.7	29.5
SIRL-Qwen2.5-32B	13.8	12.7	73.6
Mixed-SFT R1+R2	41.4	13.4	45.2
CPARL	44.9	17.8	37.3

Table 15: Single-attempt outcome decomposition (%), macro-averaged over five benchmarks. CER@1 and EIR use the common execution definition in App. C.2. *Mixed-SFT R1+R2* and CPARL report outcomes from one evaluation at seed 1234. Complete rows sum to 100 before rounding. Baseline comparators are fixed by macro pass@1 accuracy.

Execution and checker labels are paired by instance and decoding run. CER@1 counts attempts that satisfy every execution and output requirement; EIR counts those attempts rejected by the checker. Rates are aggregated within each run before taking their mean and sample standard deviation.

G.3 Essential Controlled Comparisons

Table 2 reports *Answer-only SFT R1*, *CoT-only-SFT R1* (E1), *Tool-only-SFT R1* (E2), and *One-stage-SFT R1&R2*. Table 6 reports all eleven Tool/CoT ratios from 0/100 to 100/0. *Trajectory-level credit* (E3) and *Tool-only SE-GRPO* (E5) each report 15 checkpoint evaluations; Table 2 compares their step-135 results. E4 denotes the existing-output diagnostics in App. G.2. These comparisons address supervision content, staged SFT, solver-evidence attribution, and the complete mixed-route design. The final RL evaluations use the temperature/seed schedule in App. C.3; checkpoint sweeps retain their separate single-evaluation protocol.

Reported SFT measurements. The three supervision controls, four additional mixtures (60%–90% tools), and full-data single-stage control report means and sample standard deviations over three evaluations. These summaries replace the earlier single-count-vector supervision results. For *One-stage-SFT R1&R2*, correct counts across the three runs are $(45, 43, 40)$ for CP-Bench, $(40, 41, 37)$ for TEXT2ZINC, $(33, 34, 34)$ for CPEVAL, $(100, 98, 95)$ for IndusCP, and $(35, 32, 38)$ for NL4CP, yielding totals of 253, 248, and 244 out of 900. Micro and macro scores are computed per run before taking their mean and sample standard deviation ($n - 1$ denominator), using unrounded accuracies. These counts reproduce the table entries. Summary statistics are available for the other seven new conditions. *Answer-only SFT R1* uses the problem description and instance data as input and a complete CPMpy program as the target output, without CoT reasoning or tool interactions. It is a separate supervision control outside the Tool/CoT mixture family.

One-stage-SFT R1&R2. This experiment pools and shuffles all 8,156 records instead of training first on 5,600 records and then on 2,556 targeted records. It compares training schedules with the same total corpus, not a separately collected untargeted corpus. Its mean micro/macro pass@1 accuracy (27.6%/35.0%) is below *Mixed-SFT R1+R2* (32.5%/41.1%).

Shared RL configuration. The report specifies the same second-round SFT initialization and frozen reference for the 300-task conditions. E3 and E5 each train for 225 updates: eight tasks per update, eight rollouts per task, and 14,400 rollout slots over six traversals of the pool (180 CSP and 120 COP tasks). Each run initializes a new LoRA adapter with training seed 42 and uses each batch once; optimizer and decoding settings follow App. H.5. The reported hardware is four H100 80GB GPUs with BF16 and no quantization.

E3: trajectory-level attribution of the same evidence. *Trajectory-level credit* retains event rewards and allows both routes. For each tool trajectory, it sums discounted event rewards, normalizes this return within the same problem's tool group, and assigns the resulting advantage to every block (Equation 14). The tool loss still averages over tokens within blocks, blocks within trajectories, tool trajectories within a problem, and then problems. CoT and route loss weights remain 1 and 0.1, with added clipping 0.2 and reference coefficient 0.02. This compares event/depth-relative credit with a shared trajectory advantage under the same event rewards; *terminal-correctness GRPO* instead changes both rewards and credit.

E5: *Tool-only SE-GRPO*. The environment supplies the `<tool_decision>` field with value `use_tool`; this prefix has zero policy-loss and KL masks. Every training rollout and evaluation attempt uses tools. Model edits, solver operations, and termination remain learned decisions with event/depth credit. The added CoT and route losses and their reference term have weight zero; tool-policy reference regularization remains active. This compares the complete mixed-route design with always using tools.

Checkpoint comparison and reporting. E3 and E5 each report one pass@1 evaluation of all 900 test instances at every 15th update through step 225, with worker seed 1234. Table 2 uses step 135 for all four 300-task conditions, aligned to the development-selected CPARL checkpoint. Table 2 reports means and sample standard deviations over five evaluations at this checkpoint for each condition. Micro and macro scores are computed per evaluation before summarization. Figure 5 and Tables 19–22 show the complete sweeps, including changes in their ordering.

G.4 Controlled RL Experiments on a Larger Training Pool

In this section, the two RL methods, SE-GRPO and GRPO, use the same 1,000-task training pool and Qwen3.5-27B initialization after two SFT rounds on 8,156 records. Both freely choose direct CoT or tool interaction: the 40% tool / 60% CoT ratio describes SFT supervision, not a fixed RL routing quota. Each method trains for two epochs and 250 parameter-update steps, with one evaluation every 25 steps. The report repeats the paper's configured batch of eight tasks and eight fresh on-policy rollouts per task per update. CPARL-Dev-1k selects SE-GRPO step 200 for this supplementary setting.

At step 200, five evaluations of SE-GRPO yield $36.1_{\pm 0.7}\%$ micro and $45.4_{\pm 1.2}\%$ macro pass@1 accuracy; *terminal-correctness GRPO* yields $34.9_{\pm 0.4}\%$ and $44.4_{\pm 0.9}\%$, respectively (Table 17). The mean gaps, calculated before rounding, are 1.2 micro and 1.1 macro percentage points. Micro and macro scores are computed within each evaluation before taking the mean and sample standard

RL setting	Pool tasks	Epochs	Total updates	Eval. interval	SE-GRPO step
Main	300	6	225	15	135
Larger pool	1,000	2	250	25	200

Table 16: Reported RL schedules. Epochs and total updates describe complete training runs; the final column gives the CPARL-Dev-1k-selected SE-GRPO checkpoint within each run. Both algorithms share the reported schedule within a pool setting. Table 17 reports their selected-step comparisons; GRPO is aligned to the SE-GRPO step, not independently selected.

Variant	CP-Bench	TEXT2ZINC	CPEVAL	IndusCP	NL4CP	Micro Avg.	Macro Avg.
SE-GRPO (300 tasks)	$56.2_{\pm 1.5}$	$43.3_{\pm 2.7}$	$64.4_{\pm 5.7}$	$25.1_{\pm 0.5}$	$33.9_{\pm 2.4}$	$35.0_{\pm 0.9}$	$44.6_{\pm 1.5}$
GRPO (300 tasks)	$61.2_{\pm 3.5}$	$40.9_{\pm 2.1}$	$54.4_{\pm 3.0}$	$24.9_{\pm 1.2}$	$31.3_{\pm 1.5}$	$34.0_{\pm 1.3}$	$42.5_{\pm 1.6}$
SE-GRPO (1,000 tasks)	$61.6_{\pm 1.5}$	$42.5_{\pm 0.8}$	$63.8_{\pm 2.3}$	$26.7_{\pm 0.4}$	$32.5_{\pm 4.6}$	$36.1_{\pm 0.7}$	$45.4_{\pm 1.2}$
GRPO (1,000 tasks)	$56.6_{\pm 2.6}$	$43.3_{\pm 3.3}$	$65.3_{\pm 4.0}$	$25.4_{\pm 0.3}$	$31.2_{\pm 1.9}$	$34.9_{\pm 0.4}$	$44.4_{\pm 0.9}$

Table 17: RL training-pool comparison (pass@1 accuracy, %; mean $\pm$ sample standard deviation over five evaluations per checkpoint). The 300-task setting uses step 135 and the 1,000-task setting step 200. Within each setting, GRPO uses the SE-GRPO-selected step. The 300-task SE-GRPO row is the same repeated-evaluation summary as *Full CPARL* in Table 2; historical single-evaluation sweep scores are reported separately.

deviation. Relative to the main 300-task SE-GRPO checkpoint, the larger-pool means increase by 1.1 micro and 0.9 macro points.

Why might the aggregate gain be limited? First, the reported full schedules increase nominal problem visits only from $300 \times 6 = 1{,}800$ to $1{,}000 \times 2 = 2{,}000$, despite more than tripling the number of training tasks. Each task is revisited fewer times, so wider coverage may be offset by less repeated practice with feedback-conditioned revision. These are full-run exposure counts, not counts at the selected checkpoints or evidence of equal token/solver compute. Second, both settings build on the same extensive CP-specific SFT; additional RL tasks may repeat modeling patterns already represented in SFT and the smaller targeted pool. Finally, benchmark gains and losses partly cancel in the aggregate. Five decoding evaluations measure variation at fixed checkpoints rather than variation across independent training runs. The evidence supports only a modest observed improvement under the reported schedule, not a saturation claim or a general conclusion that larger RL pools are ineffective.

G.5 RL Training Dynamics

Figure 4 shows the main 300-task training run. From the first to the last 25 updates, mean raw reward rises from 0.378 to 0.619, while tool selection rises from 41.3% to 56.0%. Mean response length averages 1,674 and 2,581 tokens per rollout in these two windows; mean calls per tool-selected rollout decreases from 2.18 to 1.97 (averaging the per-step ratios). These are descriptive training-batch trends from one run; checkpoint selection remains based on CPARL-Dev-1k (App. H.6).

H Complete Methodological Details

This appendix records the learning objectives, evidence semantics, key training settings, and the complete checkpoint-count tables with their interpretation. Interface schemas and runtime orchestration are documented in the local project materials.

H.1 Stage I: Trajectory Supervised Fine-Tuning

Both SFT rounds supervise the initial route decision and the resulting policy outputs: reasoning and final code for direct modeling, or model writes, revisions, solver requests, and termination for tool use. Tool demonstrations contain authentic responses from execution of the active CPMpy model; the demonstrator does not generate solver feedback. The terminal model must pass the training

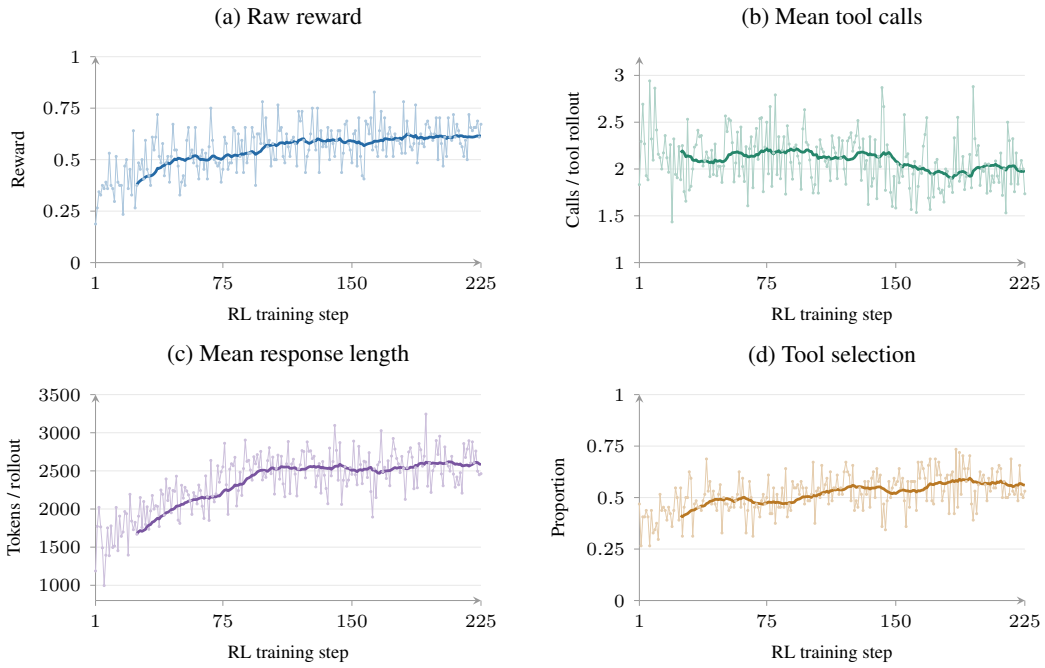

Figure 4: Training dynamics of the main CPARL run over all 225 RL updates (64 rollouts per update). Pale lines and dots show every step; dark lines show 25-step trailing means, starting at step 25. Raw reward is terminal correctness, before advantage normalization and excluding solver-event rewards. Tool calls are averaged only over rollouts selecting tools; response length and tool selection use all 64 rollouts. Length counts model-generated tokens across all turns, excluding prompts and tool observations.

validators. Task data, model metadata, diagnostics, and padding are context only. Both stages share the same observation and action interface.

For serialized record $s_i = (z_{i,1}, \ldots, z_{i,|s_i|})$, $m_{i,j}$ marks policy-generated tokens and $w_{i,j}$ is their supervision weight. Its normalized loss is

$$\mathcal{L}_{\mathrm{SFT}}^{(i)} = -\frac{1}{\sum_j m_{i,j} w_{i,j}} \sum_{j=1}^{|s_i|} m_{i,j} w_{i,j} \log p_\theta(z_{i,j} \mid z_{i,<j}). \tag{9}$$

Writing $D_i = \sum_j m_{i,j} w_{i,j} > 0$, the first-round batch objective in Sec. 4.1 is $\sum_i D_i \mathcal{L}_{\mathrm{SFT}}^{(i)} / \sum_i D_i$. The reduction aggregates weighted token sums across GPUs and accumulation steps, so it does not give every record equal weight. First-round sequences are limited to 8,192 tokens. Control tokens receive weight 4 and other supervised tokens weight 1. Control spans include the entire `<tool_decision>` field and, for non-finalize actions, delimiters, JSON structure, keys, and control-value fields; code payloads retain weight 1. Environment-owned tokens receive zero weight.

Table 18 consolidates the settings currently reported for both SFT rounds and RL. After the second round, the merged SFT parameters initialize the actor and remain frozen as its reference.

H.2 Interaction Environment and Action Space

The interaction is partially observed (Kaelbling et al., 1998): the intended semantics and full solver state are not directly available to the policy. At decision t,

$$o_t = (q, I, M_t, d_t, H_t), \tag{10}$$

contains the task, complete active source, latest diagnostic, and bounded history of actions, model versions, diagnostics, and remaining budgets. Complete source avoids reconstructing the model from truncated conversation. Every diagnostic is bound to its model digest; reward-only semantic gates and scalar reward components are excluded from observations.

Table 18: Reported training settings.

Setting	SFT round 1	SFT round 2	RL
Records / tasks	5,600 records	2,556 records	300 tasks
Training length	1 epoch; 100 updates	1 epoch; 46 updates	6 epochs; 225 updates
Batch	56 records	56 records	8 tasks $\times$ 8 rollouts
Learning rate	5×10^{-5} to 5×10^{-6}; cosine	5×10^{-5} to 5×10^{-6}; cosine	2×10^{-5} to 5×10^{-6}; linear knots
Warmup	3 updates	3 updates	5 updates
LoRA rank / scaling / dropout	64 / 128 / 0.05	64 / 128 / 0.05	16 / 32 / 0
Hardware	8 H100 GPUs	4 H100 80GB GPUs	4 H100 80GB GPUs
Reported time	3,706.5 s	1,960.5 s	12 h 5 min 34.1 s

The solver operations are introduced in Sec. 4.2. The editing actions are `set_model`, `insert`, `delete`, `replace`, and `rename`. They create a complete program or modify named elements; renaming also updates references. A transactional edit can combine related changes, such as revising a variable domain and its channeling constraints. Edits apply atomically after schema, version, and source checks; rejection preserves the model and returns an observable diagnostic. Solver calls preserve source and supply feedback at the next decision. Compilation checks model construction and typing; propagation reports root-domain changes and conflicts; solving returns feasibility, candidate assignments, objective values, and available search statistics. Independent validation supplies evidence of correctness. Validation does not force termination: its counterexamples may guide another revision. Only `finalize` or the declared budget ends the episode.

Compilation failures, rejected edits, solver timeouts, and failed validation remain policy outcomes. Infrastructure failures are retried and excluded from reward computation. A budget end without a completed submission fails the common terminal criterion. Cached deterministic results retain the same logical cost and evidence as fresh execution, so cache state cannot alter the objective or extend the episode. Initial model generation belongs to the first solver-event block. At inference the optimizer and reward ledger are removed; both routes still undergo the same output-format and terminal checks.

H.3 Solver-Event Blocks and Rewards

Only tool trajectories have solver-event blocks. The block definition in Sec. 4.2 groups decisions through a closing `compile`, `propagate`, `solve`, `validate`, or `finalize`. The first block includes initial-model generation. Edits before one check remain in the same block because no intervening solver evidence separates their effects.

The digest-bound ledger records semantic validation V, executability E, propagation Φ, complexity C, and cost κ. Each evidence delta compares with the most recent corresponding evidence in the same rollout. An unchanged repeated check earns no positive delta and still incurs cost. For the nonempty set X_{prim} of primary, output-linked variables, with initial and root-propagated domains D_x^0 and D_x^{root},

$$\bar{\delta}(M) = \frac{1}{|X_{\mathrm{prim}}|} \sum_{x \in X_{\mathrm{prim}}} \frac{|D_x^0| - |D_x^{\mathrm{root}}|}{\max(1, |D_x^0|)}, \qquad \Phi(M) = \min\{\bar{\delta}(M), \tau\}. \tag{11}$$

Pruning can remove valid assignments, so a propagation difference contributes only when later validation succeeds on that exact model digest. A changed model or failed validation does not authorize the earlier propagation evidence. Complexity and interaction cost are

$$C(M) = \log\bigl(1 + \mathbf{w}^\top \mathbf{n}(M)\bigr), \tag{12}$$

$$\kappa(B_{i,k}) = \kappa_{\mathrm{act}} |B_{i,k}| + \sum_{e \in \mathcal{O}_{i,k}} \kappa_e, \tag{13}$$

where $\mathbf{n}(M)$ counts variables, constraints, auxiliary variables, executable syntax nodes, and output statements; $\mathbf{w}$ is fixed globally. $\mathcal{O}_{i,k}$ is the multiset of solver operations. Costs may distinguish operations but are shared across problems. Equation 2 penalizes positive complexity growth only. Its finalized rewards enter the discounted return in Equation 3, retaining both local evidence differences and delayed credit without a learned process reward model.

Reward weights and their selection. Validation measures progress toward a correct solution; execution and validated propagation provide feedback on intermediate progress. Model-growth and interaction penalties discourage unnecessary code and calls. We choose the five λ coefficients using performance on CPARL-DEV-1K. Their effect depends on each component's scale. For example, propagation is a bounded fraction, whereas interaction cost accumulates across actions and solver calls. The weighted contributions in Equation 2, rather than the coefficients alone, determine how strongly each component affects the reward.

H.4 SOLVER-EVENT GRPO

Let G_{tool} be the number of tool trajectories among the eight rollouts sampled from one behavior snapshot for a problem. The tool loss is zero for an empty tool group. A singleton tool group has zero relative advantage while retaining reference regularization. These statistics are separate from the all-eight outcome comparison.

For depth k, let $\mathcal{I}_{q,k} = \{i : K_i \geq k\}$ within the tool subset. When at least two rollouts reach that depth, Equation 3 uses their return mean and population standard deviation, with stop-gradient through the normalized advantage. With too few rollouts at a late depth, the documented fallback is

$$\widehat{A}_i^{\text{traj}} = \text{stopgrad}\left(\frac{R_i^{\text{traj}} - \mu_q^{\text{traj}}}{\max(\sigma_q^{\text{traj}}, \epsilon_A)}\right), \qquad R_i^{\text{traj}} = \sum_{k=1}^{K_i} \gamma^{k-1} r_{i,k}. \tag{14}$$

The fallback mean and population standard deviation use all tool trajectories of the same problem. It applies when $G_{\text{tool}} \geq 2$ but only one trajectory reaches the current depth; a singleton tool group retains zero relative advantage. The reported *Trajectory-level credit* control assigns this advantage to every block. It differs from the measured *terminal-correctness GRPO* control, which uses only the final binary outcome.

For a policy token z_u, use the standard behavior ratio $\rho_{i,u} = \pi_\theta(z_u \mid z_{<u})/\pi_{\theta_{\text{old}}}(z_u \mid z_{<u})$ and clipped surrogate

$$h(\rho, A) = \min\{\rho A, \text{clip}(\rho, 1-\varepsilon, 1+\varepsilon)A\}. \tag{15}$$

Thus $\ell_{i,k,u} = h(\rho_{i,u}, \widehat{A}_{i,k})$ in Equation 4. Reference regularization uses the frozen second-round SFT policy and the nonnegative sampled estimator $d_{i,u}^{\text{ref}} = \exp(\delta_{i,u}) - \delta_{i,u} - 1$, where $\delta_{i,u} = \log \pi_{\theta_{\text{SFT}}}(z_u \mid z_{<u}) - \log \pi_\theta(z_u \mid z_{<u})$. Its reduction follows the policy loss:

$$\mathcal{K}_{\text{ref}} = \frac{1}{G_{\text{tool}}} \sum_{i=1}^{G_{\text{tool}}} \frac{1}{K_i} \sum_{k=1}^{K_i} \frac{1}{|\mathcal{U}_{i,k}|} \sum_{u \in \mathcal{U}_{i,k}} d_{i,u}^{\text{ref}}. \tag{16}$$

Both terms average tokens within blocks, blocks within trajectories, and trajectories within the tool group. Environment responses are masked. This hierarchy gives equal averaging weight to blocks within a trajectory regardless of their token counts. SE-GRPO combines these solver-defined credit boundaries, model-specific evidence differences, and group-relative updates; it does not claim a new generic policy-gradient family (Shao et al., 2024; Wei et al., 2026; Feng et al., 2025b).

H.5 MIXED-ROUTE OBJECTIVE AND TRAINING CONFIGURATION

Outcomes and masks. Let $Y_i = 1$ when attempt i submits a valid answer that passes the final checker, and $Y_i = 0$ otherwise. We compare all $G = 8$ attempts at the same problem using

$$A_i^{\text{out}} = \text{stopgrad}\left[\frac{Y_i - \bar{Y}_q}{\max(s_q, 10^{-6})}\right], \qquad \bar{Y}_q = \frac{1}{G}\sum_{i=1}^{G} Y_i, \tag{17}$$

Here $\bar{Y}_q$ and s_q are the mean and population standard deviation of the binary outcomes. Compilation alone is insufficient. Missing, duplicate, or inconsistent route tags, empty answers, incorrect programs, and exhausted budgets without submission give $Y_i = 0$. Hidden terminal checking is not an

actor-initiated tool call. All-correct and all-incorrect groups have zero added advantage and remain in the group.

Let $\mathcal{D}_i$ contain only the value tokens of the first identifiable legal <tool_decision>, excluding its tags. Let $\mathcal{C}_i$ contain direct-route reasoning, final code, and submission delimiters, excluding $\mathcal{D}_i$; tool trajectories have $\mathcal{C}_i = \varnothing$. These masks exclude environment and checker outputs. With $\mathcal{X}_i = \mathcal{C}_i$ for $X = \mathrm{CoT}$ or $\mathcal{D}_i$ for $X = \mathrm{route}$,

$$L_X = -\frac{1}{G}\sum_{i=1}^{G}\frac{\sum_{u\in\mathcal{X}_i} h(\rho_{i,u}, A_i^{\mathrm{out}})}{\max(1, |\mathcal{X}_i|)}, \qquad X \in \{\mathrm{CoT}, \mathrm{route}\}. \tag{18}$$

Their weights in Equation 5 are 1 and 0.1; clipping uses $\varepsilon = 0.2$ and added reference regularization $\beta_{\mathrm{add}} = 0.02$. Empty masks contribute zero without removing an outcome. On malformed trajectories, identifiable valid decision values and recognizable direct content remain trainable with outcome zero; missing token positions are never fabricated.

The tool objective keeps its policy mask; any initial route tokens it covers receive both event and outcome signals. The added KL uses the same sampled estimator as the tool term, excluding tokens already covered by tool reference regularization. It averages over newly covered tokens and then all eight rollouts. The complete objective averages over problems. The tool loss uses the variable tool-subgroup size, whereas added terms use all eight outcomes; uniformly averaging 64 trajectory losses is generally not equivalent.

For *Trajectory-level credit*, these added losses and their weights are unchanged. For *Tool-only SE-GRPO*, the environment-supplied route prefix is excluded from policy and KL masks; L_{CoT}, L_{route}, and β_{add} are zero, while tool-policy reference regularization remains active.

For four correct rollouts out of eight, $\bar{Y}_q = s_q = 0.5$ and the added advantages are $+1$ and -1. A correct tool and a correct direct rollout receive the same added advantage regardless of tool-event return. This does not equalize total update strength across routes: the tool event objective remains active. No added reward favors a route proportion, reasoning length, or tool count.

Training and inference budgets. The main 300-task RL configuration runs for 225 updates without adapter or optimizer resets; App. G.4 records the larger-pool schedule. Three reporting phases each contain 75 updates and two traversals of the 300-task pool. Every update uses eight distinct tasks and eight fresh rollouts per task, totaling 14,400 rollout slots; each batch is used once. The learning rate warms up through step 5, then interpolates $(5, 2\times10^{-5})$, $(75, 1.5\times10^{-5})$, $(150, 10^{-5})$, and $(225, 5\times10^{-6})$. AdamW uses $(\beta_1, \beta_2) = (0.9, 0.95)$, $\epsilon = 10^{-8}$, no weight decay, and gradient clipping 1.0. Training uses BF16 without quantization and seed 42; first-round SFT uses seed 1234.

Training sampling uses temperature/top-p 1.0/1.0; checkpoint-sweep evaluation uses 0.6/0.95 and one submission per task. The five evaluations of final CPARL follow App. C.3. *Full CPARL*, *terminal-correctness GRPO*, and E3 select routes naturally; E5 fixes the tool route (App. G.3). All four 300-task conditions disable top-k and use repetition penalty 1.0. RL context and initial input limits are 16,384 and 4,096 tokens. Direct assistant output is capped at 8,192 tokens, including the route decision, reasoning, and code. Direct generation has a 300-second limit after slot acquisition, excluding queueing and terminal checking; this is not a solver timeout. The four 300-task sweeps use evaluation worker seed 1234. Tool-route budgets and worker seeds for the final repeated evaluations are given in App. C.3.

The *terminal-correctness GRPO* control shares the stated initialization, tasks, training schedule, and evaluation budgets. It does not force either route: both direct and tool trajectories receive the same terminal-correctness reward rule, with one outcome-based advantage per trajectory. It uses neither event credit nor a separate route loss. Routing is inherited from mixed-route SFT; if initial decision tokens are covered by the ordinary policy loss, they also receive this terminal-outcome credit. Excluding them from that mask would remove that direct gradient, but would not establish a fixed routing behavior under shared parameter updates.

H.6 Checkpoint Selection and Sweep Interpretation

Figure 5 compares *full CPARL* (SE-GRPO), *terminal-correctness GRPO*, *Trajectory-level credit* (E3), and *Tool-only SE-GRPO* (E5). Each is evaluated every 15 updates from 15 to 225 on the same

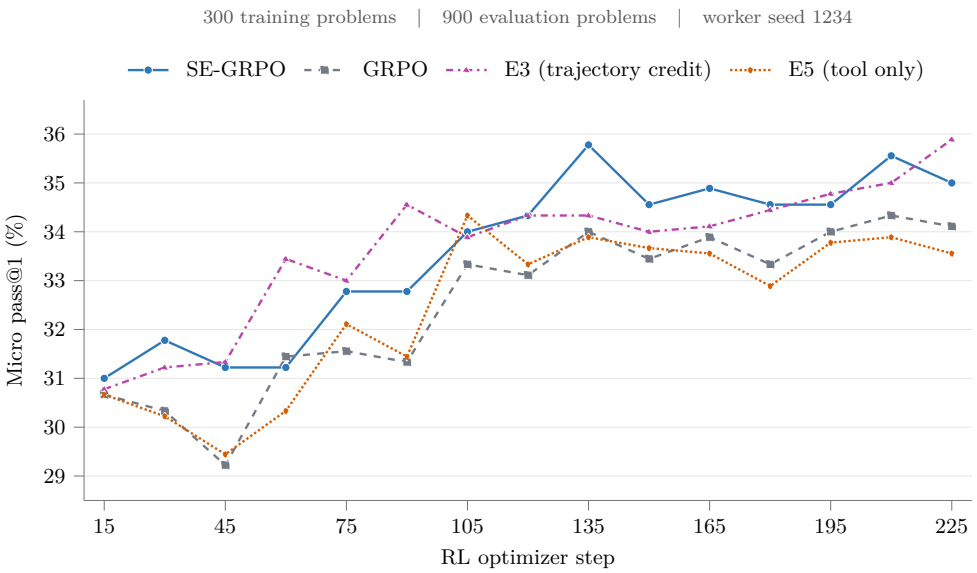

Figure 5: Micro pass@1 across 900 test instances for four 300-task RL conditions. Each point is one evaluation (worker seed 1234). SE-GRPO, GRPO, and E3 freely choose routes; E5 always uses tools. Step 135 is the development-selected CPARL checkpoint and the common comparison point in Table 2.

900 instances: 101 CP-Bench, 110 TEXT2ZINC, 64 CPEVAL, 478 IndusCP, and 147 NL4CP. Tables 19–22 give all per-benchmark correct counts, totals, and micro accuracies. These records remain single-evaluation checkpoint sweeps and are separate from the new five-evaluation summaries.

SE-GRPO exceeds E3 at eight checkpoints, ties at one, and has lower accuracy at six. At the common step 135, it solves 322 tasks versus 309 for E3; at step 225, E3 solves 323 versus 315 for SE-GRPO. SE-GRPO exceeds E5 at 14 of 15 checkpoints (E5 leads at step 105), with 322 versus 305 correct at step 135. It likewise exceeds GRPO at 14 checkpoints, with GRPO ahead at step 60.

Common-step comparisons and repeated evaluations serve different roles. Table 2 reports all four 300-task conditions at step 135, selected on CPARL-DEV-1K for CPARL. It retains only the repeated-evaluation *Full CPARL* summary for SE-GRPO; Table 17 also uses five-evaluation summaries, while Table 19 preserves the single-evaluation sweep. The five evaluations of final CPARL yield 315, 325, 320, 304, and 312 correct out of 900. Their micro/macro means and sample standard deviations are $35.0_{\pm 0.9}\%$ and $44.6_{\pm 1.5}\%$, respectively. The single sweep result (322/900) is reported separately and is not included in this five-evaluation summary. The mean gains over second-round SFT are 2.5 micro and 3.5 macro percentage points, with improvements on all five benchmarks. Table 2 also reports five-evaluation summaries for GRPO, E3, and E5 at step 135; their single-evaluation sweeps remain separate. These evaluations vary decoding at a fixed checkpoint; they are not independent training runs.

H.7 TRAINING ALGORITHM

Algorithm 1 summarizes the learning sequence. All eight attempts for a task come from one frozen behavior snapshot.

Algorithm 1 Two-Stage Training of CPARL

1: Train first-round LoRA SFT on 5,600 mixed-route records (Sec. 4.1)
2: Continue SFT on 2,556 error-targeted records; merge and freeze the reference policy
3: Initialize the actor from this policy and attach a new RL LoRA adapter
4: **for** each RL update **do**
5: Freeze a behavior snapshot; sample eight fresh mixed-route attempts for each of eight tasks

Step	CP-Bench (101)	TEXT2ZINC (110)	CPEVAL (64)	IndusCP (478)	NL4CP (147)	Total	Micro (%)
15	50	35	39	115	40	279	31.0
30	48	36	40	120	42	286	31.8
45	51	35	39	114	42	281	31.2
60	52	37	37	110	45	281	31.2
75	52	39	41	119	44	295	32.8
90	54	40	42	115	44	295	32.8
105	53	43	44	120	46	306	34.0
120	55	45	41	124	44	309	34.3
135	60	47	40	129	46	**322**	**35.8**
150	54	48	40	125	44	311	34.6
165	59	47	43	123	42	314	34.9
180	56	47	43	124	41	311	34.6
195	57	42	42	128	42	311	34.6
210	58	46	43	129	44	320	35.6
225	57	45	43	127	43	315	35.0

Table 19: Single-run *mixed-route SE-GRPO* checkpoint sweep on the main 300-task RL pool. Entries are correct-answer counts over 900 benchmark instances; micro pass@1 is reported in percent. The step-135 sweep result is separate from the five-evaluation summary of the final model in Table 1.

Step	CP-Bench (101)	TEXT2ZINC (110)	CPEVAL (64)	IndusCP (478)	NL4CP (147)	Total	Micro (%)
15	50	37	41	105	43	276	30.7
30	46	35	40	108	44	273	30.3
45	50	34	35	104	40	263	29.2
60	53	35	36	111	48	283	31.4
75	51	37	39	108	49	284	31.6
90	54	41	40	105	42	282	31.3
105	53	42	43	117	45	300	33.3
120	56	40	42	118	42	298	33.1
135	62	45	35	119	45	306	34.0
150	64	45	35	114	43	301	33.4
165	61	44	40	120	40	305	33.9
180	55	42	42	122	39	300	33.3
195	58	40	42	123	43	306	34.0
210	60	45	39	120	45	309	34.3
225	57	42	40	125	43	307	34.1

Table 20: Single-run *terminal-correctness GRPO* checkpoint sweep on the main 300-task RL pool, using the same 900 benchmark instances. Entries are correct-answer counts; micro pass@1 is reported in percent. The highest observed micro accuracy occurs at step 210; Table 17 compares both methods at the SE-GRPO-selected step 135.

6: Check submissions and assign common Y_i; retain failures and empty masks in each group
7: Compute A_i^{out} from all eight outcomes for each task
8: Finalize digest-bound tool rewards, delayed gates, and event-depth/fallback advantages
9: Update with Equation 5, averaging over tasks and counting KL once
10: **end for**

H.8 Worked SE-GRPO Example

Consider four tasks assigned to four slots, with task 1 fixed to slot 1. Omitting `AllDifferent(slot)` permits several tasks to share a slot even though the model compiles and solves. A rollout can contain five event blocks: (1) generate the faulty model and compile; (2) validate and observe a counterexample; (3) insert `AllDifferent` and propagate, holding pruning credit pending; (4) validate the repaired digest, activating its propagation evidence; and (5) submit with `finalize`. The last block repeats no positive evidence but still incurs the prescribed termination cost; it is a required submission, not a redundant diagnostic.

Trajectory-level event-return GRPO gives every block one advantage. SE-GRPO compares blocks at matching depth across tool rollouts, allowing the repair and other decisions to receive different credit. If an intervening edit changes the digest, validation of that new model cannot authorize the

Step	CP-Bench (101)	TEXT2ZINC (110)	CPEVAL (64)	IndusCP (478)	NL4CP (147)	Total	Micro (%)
15	49	33	40	114	41	277	30.8
30	48	32	43	115	43	281	31.2
45	54	33	34	116	45	282	31.3
60	60	37	45	110	49	301	33.4
75	57	34	44	119	43	297	33.0
90	59	40	43	124	45	311	34.6
105	54	46	39	123	43	305	33.9
120	58	43	38	128	42	309	34.3
135	58	41	43	121	46	**309**	**34.3**
150	58	43	39	121	45	306	34.0
165	56	45	39	122	45	307	34.1
180	54	42	40	129	45	310	34.4
195	57	51	36	121	48	313	34.8
210	57	43	38	127	50	315	35.0
225	62	47	41	126	47	323	35.9

Table 21: Single-evaluation *Trajectory-level credit* (E3) checkpoint sweep on the 300-task pool. Entries are correct counts over 900 test instances (worker seed 1234); micro pass@1 is in percent. Bold marks step 135, the checkpoint used for the separate five-evaluation summary in Table 2; the scores here remain single-evaluation results.

Step	CP-Bench (101)	TEXT2ZINC (110)	CPEVAL (64)	IndusCP (478)	NL4CP (147)	Total	Micro (%)
15	51	40	39	102	44	276	30.7
30	43	33	42	110	44	272	30.2
45	50	34	35	107	39	265	29.4
60	50	33	34	110	46	273	30.3
75	53	40	36	111	49	289	32.1
90	57	38	43	102	43	283	31.4
105	54	45	44	119	47	309	34.3
120	54	42	42	119	43	300	33.3
135	59	45	33	120	48	**305**	**33.9**
150	65	43	38	111	46	303	33.7
165	60	41	40	119	42	302	33.6
180	57	39	40	124	36	296	32.9
195	59	42	39	120	44	304	33.8
210	62	43	37	117	46	305	33.9
225	54	39	40	123	46	302	33.6

Table 22: Single-evaluation *Tool-only SE-GRPO* (E5) checkpoint sweep on the 300-task pool. Entries are correct counts over 900 test instances (worker seed 1234); micro pass@1 is in percent. Bold marks step 135, the checkpoint used for the separate five-evaluation summary in Table 2; the scores here remain single-evaluation results.

old pruning evidence. Averaging tokens within each block gives a long initial program and a short repair the same block-level averaging weight.

I CPARL ROLLOUT EXAMPLES

The following examples are from the CPARL evaluation. The panels show a tool-assisted repair and a direct submission; explanatory annotations summarize the recorded interactions.

I.1 TOOL-ASSISTED MODELING REPAIR

Problem Description: CP-Bench / Abbot's Puzzle

Distribute 100 bushels among 100 people: three bushels per man, two per woman, and half per child. There are five times as many women as men. Find the numbers of men, women, and children.

CPARL Agent Response (Condensed Tool Trajectory)

Route: `use_tool` **6 assistant turns; 3 tool calls**

1 Create the initial model

```
<tool_decision>use_tool</tool_decision>
```

`set_model` declares `men`, `women`, and `children` as integers in $[0, 100]$, with these constraints:

```
constraints = [
    men + women + children == 100,
    3 * men + 2 * women + children == 200,
    women == 5 * men,
]
```

`PATCH_OK` The model is stored.

2 Compile

`COMPILE_OK` `compile` succeeds: CPMpy construction and conversion to OR-Tools pass.

3 Solve the initial model

`SOLVE_UNSAT` `solve` returns `UNSATISFIABLE`; no assignment is available.

4 Repair the grain constraint

Repair annotation. Scaling $3m + 2w + \frac{1}{2}c = 100$ by two requires doubling both adult coefficients as well. The agent replaces line 9 of the original program:

```
{"type":"replace","start_line":9,"end_line":9,
 "code":"        6 * men + 4 * women + children == 200,"}
```

`PATCH_OK` The replacement is applied.

5 Solve the repaired model

`SOLVE_SAT` The second `solve` returns `FEASIBLE`: `men=5, women=25, children=70`.

6 Submit the corrected program

`finalize` The agent submits the repaired program; `ANSWER_OK` acknowledges receipt.

Solution check. $(m, w, c) = (5, 25, 70)$ satisfies:

$$m + w + c = 5 + 25 + 70 = 100,$$
$$3m + 2w + \tfrac{1}{2}c = 15 + 50 + 35 = 100, \qquad w = 5m = 25.$$

The three tool calls are one compilation and two solves. `ANSWER_OK` denotes submission receipt; the arithmetic above is an explanatory check, not an additional validation call.

I.2 Direct Solution via the CoT Route

Problem Description: Text2Zinc / Refinery

Instance `text2zinc:2` has 8,000 and 5,000 units of two crude-oil resources. Three processes consume $(3, 5)$, $(1, 1)$, and $(5, 3)$ units per process unit; product sales minus operating costs give net returns of 200, 60, and 206. Maximize total net revenue within both resource limits.

CPARL Agent Response (Recorded Direct Submission)

Route: `no_tool` **1 assistant turn; 0 tool calls**

The saved response contains a route decision and a final program, with no separate step-by-step explanation. The response is reproduced below with line wrapping only.

```
<tool_decision>no_tool</tool_decision>
<action>{"type":"finalize"}</action>

<answer>
```

```python
import cpmpy as cp

def build_model(instance=None, metadata=None):
    Allocated = [8000, 5000]
    Cost = [51, 11, 40]
    Input = [[3, 5], [1, 1], [5, 3]]
    Output = [[4, 3], [1, 1], [3, 4]]
    Price = [38, 33]
    L, O, P = 3, 2, 2

    x = cp.intvar(0, 10000, shape=L, name="x")
    constraints = []
    for o in range(O):
        constraints.append(
            sum(Input[l][o] * x[l] for l in range(L))
            <= Allocated[o])

    revenue = sum(Output[l][p] * Price[p] * x[l]
                  for l in range(L) for p in range(P))
    cost = sum(Cost[l] * x[l] for l in range(L))
    objective = revenue - cost

    model = cp.Model(constraints, maximize=objective)
    variables = {"x": x}
    return model, variables
```

```
</answer>
```

Post-hoc Check (Outside the Agent Trajectory)

Submitted formulation.

$$\begin{aligned} \max \quad & 200x_1 + 60x_2 + 206x_3 \\ \text{s.t.} \quad & 3x_1 + x_2 + 5x_3 \le 8000, \\ & 5x_1 + x_2 + 3x_3 \le 5000, \qquad x_1, x_2, x_3 \in \{0, \dots, 10000\}. \end{aligned}$$

Reported result. Independent execution returns $(x_1, x_2, x_3) = (0, 500, 1500)$ with net revenue **339,000**, matching the reference optimum and using both resources fully.

Scope. The original problem permits fractional process levels. This instance has an integral optimum attained by the submitted integer model; objective agreement does not establish domain equivalence. This check is not a tool call made during generation.

J EVALUATION PROMPTS

We provide the system and user prompts for CPARL evaluation and the Direct Prompt baseline. Both require executable CPMpy code with OR-Tools as the backend. CPARL chooses between direct generation and tool-assisted revision; Direct Prompt submits a complete model in one response without execution feedback. Final submitted programs are checked by the evaluator in both settings.

J.1 CPARL EVALUATION PROMPT

The system prompt includes the routing protocol appended by the evaluation worker. The user template is instantiated with the current observation at each turn: the problem description, model code, latest feedback, and interaction history. Initially, the model and history are empty and no diagnostic is available. The format reminder is appended to every user input. Evaluation budgets are specified in App. C.3.

CPARL Evaluation Prompt (System, CPMpy)

```
You are a constraint programming modeling agent. Your task is to solve CP
  problems by choosing actions from a limited action space based on the
  current observation.

At each step, you will receive an observation in the following form:

o_t = (NL, M_t, D_t, H_t)

Where:
- NL is the natural-language description of the CP problem.
- M_t is the current solver code. The code must use only CPMpy with OR-Tools
  as the backend.
- D_t is the diagnostic information for the current step, namely the latest
  feedback returned by the environment or tools.
- H_t is the interaction history of at most T steps before the current step,
  including previous actions and their results. The value of T is given in the
   sample metadata.

The observation payload must follow the JSON structure below:

{
  "NL": {
    "problem": "NATURAL_LANGUAGE_PROBLEM"
  },
  "M_t": {
    "source": "CURRENT_EXECUTABLE_CPMPY_CODE",
    "numbered_source": "CURRENT_CODE_WITH_LINE_NUMBERS"
  },
  "D_t": {
    "source": "none | compile | propagate | solve",
    "message": "CONCISE_FEEDBACK_MESSAGE"
  },
  "H_t": [
    {
      "turn": 0,
      "action": {},
      "result": {
        "message": "Model replaced"
      }
    }
  ]
}

Field meanings:
- NL.problem is the original CP problem description.
- M_t.source is the current executable CPMpy model code.
- M_t.numbered_source is the same current code with line numbers for edit
  actions.
```

CPARL Evaluation Prompt (System, CPMpy) (continued)

```
- D_t.source identifies the latest feedback source: none, compile, propagate,
  or solve.
- D_t.message is concise feedback containing necessary errors, solver status,
  objective values, or key solution values while avoiding overly long output.
- H_t contains at most the previous T interaction records.
- Each H_t entry summarizes one previous action and the result after that
  action.

Based on the observation, you must accurately output one action from the
  allowed action space.

CPMpy modeling rules:
- Use `import cpmpy as cp`. Do not use PyCSP3, MiniZinc, the native OR-Tools
  Python API, or any other modeling language.
- Unless the problem truly requires a small deviation, prefer the canonical
  structure below:
   1. imports;
   2. optional data/constants;
   3. `def build_model(instance=None, metadata=None):`;
   4. decision variables and auxiliary variables;
   5. `constraints = []`;
   6. append constraints in small semantic blocks;
   7. optional objective expression;
   8. `model = cp.Model(constraints)` or `cp.Model(constraints, minimize=
  objective)` or `cp.Model(constraints, maximize=objective)`;
   9. `variables = {"name": var, ...}`;
   10. `return model, variables`.
- Keep variable names stable and meaningful. Prefer names that are easy to
  patch, such as `x`, `start`, `assign`, `load`, `used`, or names derived
  directly from the problem statement.
- Use finite integer domains and Boolean domains. Create variables with `cp.
  intvar(lb, ub, shape=..., name="...")` or `cp.boolvar(shape=..., name="...")
  `.
- Put most constraints into `constraints += [...]` statements. Whenever
  possible, keep each line or block to one semantic constraint or a small
  coherent group of constraints.
- When CPMpy global constraints express the model more clearly, and the
  installed CPMpy version and OR-Tools backend support them, use global
  constraints such as `cp.AllDifferent`, `cp.Element`, `cp.Table`, `cp.Count`,
   `cp.Cumulative`, `cp.Circuit`, `cp.NoOverlap`, `cp.Increasing`, `cp.
  LexLessEq`, and `cp.LexChainLessEq`.
- Use auxiliary variables, reification, channeling constraints, symmetry
  breaking, and implication constraints when they improve correctness or make
  solving easier.
- If the canonical template would make the model incorrect, do not overfit to
  the template for the sake of form. Modeling accuracy takes priority over
  superficial regularity.
- Unless the task explicitly asks for a standalone script, do not call `model.
  solve()` in the model code. The environment tools are responsible for
  compilation, propagation, solving, and validation.

Allowed action types:

1. set_model:
   Replace the entire current CP model with a new CPMpy model. Use this when
  the model is empty or the current model is fundamentally wrong.
   Format: <action>{"type":"set_model","code":"CPMPY_CODE"}</action>

2. insert:
   Insert new CPMpy code before or after a specified line in the current model
  . Use this to add variables, auxiliary expressions, constraints, objectives,
   helper functions, or return entries without rewriting the whole model.
   Format: <action>{"type":"insert","after_line":1,"code":"CPMPY_CODE"}</
  action>
   Or: <action>{"type":"insert","before_line":1,"code":"CPMPY_CODE"}</action>
```

CPARL Evaluation Prompt (System, CPMpy) (continued)

```
3. delete:
   Delete a specified range of code lines. Use this to remove incorrect,
  redundant, unsupported, or conflicting code.
   Format: <action>{"type":"delete","start_line":1,"end_line":1}</action>

4. replace:
   Replace a specified range of code lines with new CPMpy code. Use this to
  fix incorrect variables, constraints, domains, objectives, or modeling logic
  .
   Format: <action>{"type":"replace","start_line":1,"end_line":1,"code":"
  CPMPY_CODE"}</action>

5. rename:
   Rename a variable, symbol, or identifier in the current model. Use this
  when the modeling logic is correct but the naming is inconsistent or unclear
  .
   Format: <action>{"type":"rename","old":"OLD_IDENTIFIER","new":"
  NEW_IDENTIFIER"}</action>

6. tool:
   Call an environment tool without directly modifying the current model. The
  returned result will appear in D_t of the next observation.

   - compile:
     Check whether the current CPMpy model can be parsed, imported, and
  converted to the OR-Tools backend. Use this before solving to catch syntax
  errors, undefined symbols, malformed CPMpy expressions, unsupported
  constructs, or backend conversion issues.
     Format: <action>{"type":"tool","name":"compile"}</action>

   - propagate:
     Run lightweight propagation or presolve on the current CPMpy model with
  OR-Tools. Use this to detect infeasibility early and, when available, obtain
   variable-level domain tightening information. Do not expect this tool to
  provide PyCSP3-style conflict explanations, first failing constraints, or
  unsatisfiable cores.
     Format: <action>{"type":"tool","name":"propagate"}</action>

   - solve:
     Solve the current CPMpy model using the OR-Tools backend and return the
  solver result. Use this to check feasibility or optimality and obtain
  solution values, objective values, or solver status for the current model.
     Format: <action>{"type":"tool","name":"solve"}</action>

7. finalize:
   End the trajectory and output the final correct CPMpy model.
   Format:
   <action>{"type":"finalize"}</action>
   <answer>FINAL_CPMPY_CODE</answer>

The action process can repeat for at most N rounds. The value of N is given in
   the sample metadata.

Output constraints:
- At every step, output exactly one allowed action in the specified format.
- Do not output explanations, reasoning, Markdown, code fences, or any other
  text outside the required tags.
- The action JSON must be a single valid JSON object.
- The action type must be strictly one of: set_model, insert, delete, replace,
   rename, tool, finalize.
- If the action type is tool, the tool name must be strictly one of: compile,
  propagate, solve.
- Do not invent new action types, new tools, or extra fields.
- Line numbers refer to the current numbered model in M_t.
- For insert, use exactly one of before_line or after_line.
- For delete and replace, start_line and end_line are inclusive endpoints.
```

CPARL Evaluation Prompt (System, CPMpy) (continued)

```
- The final answer must be correct, executable CPMpy code that can be solved
  with the OR-Tools backend.
- You must finalize within at most N action rounds.

STRICT ROUTING PROTOCOL (highest priority):
- The first assistant response must begin with exactly one of <tool_decision>
  use_tool</tool_decision> or <tool_decision>no_tool</tool_decision>. Never
  repeat this tag.
- use_tool: the first action must be set_model; continue with normal edit/tool
   actions, and finish only with finalize plus answer.
- no_tool: finish in that same response with finalize plus answer; do not
  enter the tool loop.
- On every non-final action turn, output exactly one complete block and
  nothing else: <action>{VALID_JSON_OBJECT}</action>.
- The response must begin with the exact literal `<action>` and end with the
  exact literal closing tag `</action>` (including the slash).
- Never end with `<action>`; that is an opening tag, not a closing tag.
- Between the two tags, emit exactly one valid JSON object. Escape all quotes,
   backslashes, and newlines inside every `code` string as JSON.
- Before sending, verify: opening tag present; JSON closed; exact `</action>`
  closing tag present; no prose or Markdown outside the tags.
- For finalization only, output exactly a complete `<action>{"type":"finalize
  "}</action>` block followed by one complete `<answer>FINAL_CPMPY_CODE</
  answer>` block.
```

CPARL Evaluation Prompt (User, Each Turn)

```
<observation>
{observation_text}
</observation>

Choose the next structured CP action. Output only the required tag(s).

MANDATORY FORMAT CHECK: On the first response, begin with exactly one
  tool_decision tag. After choosing use_tool, set_model must be the first
  action. After choosing no_tool, finalize and answer in this response. Never
  repeat tool_decision. A non-final response must start with `<action>`,
  contain exactly one valid JSON object, and end with the exact nine
  characters `</action>`. The slash is required. Never end with `<action>`.
  Example: <action>{"type":"tool","name":"compile"}</action>. For finalization
  , close the finalize action block before the complete `<answer>...</answer>`
   block.
```

J.2 DIRECT PROMPT

The direct baseline receives the same problem description and submits a complete CPMpy model without intermediate tool calls. Its `no_tool` tag is a fixed output marker, not a learned route choice.

Direct Testing Prompt (System, CPMpy)

```
You are a constraint programming modeling agent. Your task is to directly
  translate a natural-language CP problem into a correct, executable CPMpy
  model in a single response.

You must solve the modeling task without tool calls, execution feedback, or
  iterative interaction. Return the complete final model directly.

CPMpy modeling rules:
- Use `import cpmpy as cp`. Do not use PyCSP3, MiniZinc, the native OR-Tools
  Python API, or any other modeling language.
- Use CPMpy with OR-Tools as the backend.
- Unless the problem truly requires a small deviation, prefer the canonical
  structure below:
```

Direct Testing Prompt (System, CPMpy) (continued)

```
  1. imports;
  2. optional data/constants;
  3. `def build_model(instance=None, metadata=None):`;
  4. decision variables and auxiliary variables;
  5. `constraints = []`;
  6. append constraints in small semantic blocks;
  7. optional objective expression;
  8. `model = cp.Model(constraints)` or `cp.Model(constraints, minimize=
  objective)` or `cp.Model(constraints, maximize=objective)`;
  9. `variables = {"name": var, ...}`;
  10. `return model, variables`.
- Keep variable names stable and meaningful. Prefer names such as `x`, `start
  `, `assign`, `load`, `used`, or names derived directly from the problem
  statement.
- Use finite integer domains and Boolean domains. Create variables with `cp.
  intvar(lb, ub, shape=..., name="...")` or `cp.boolvar(shape=..., name="...")
  `.
- Put most constraints into `constraints += [...]` statements. Whenever
  possible, keep each line or block to one semantic constraint or a small
  coherent group of constraints.
- When CPMpy global constraints express the model more clearly, and the
  installed CPMpy version and OR-Tools backend support them, use global
  constraints such as `cp.AllDifferent`, `cp.Element`, `cp.Table`, `cp.Count`,
   `cp.Cumulative`, `cp.Circuit`, `cp.NoOverlap`, `cp.Increasing`, `cp.
  LexLessEq`, and `cp.LexChainLessEq`.
- Use auxiliary variables, reification, channeling constraints, symmetry
  breaking, and implication constraints when they improve correctness or make
  solving easier.
- If the canonical template would make the model incorrect, do not overfit to
  the template for the sake of form. Modeling accuracy takes priority over
  superficial regularity.
- Unless the task explicitly asks for a standalone script, do not call `model.
  solve()` in the model code. The evaluator will execute and check the
  submitted model.

Output requirements:
- Generate the complete final CPMpy code in this response.
- Do not request tools or output intermediate model-editing actions.
- Do not output explanations, reasoning, Markdown, or code fences.
- Use exactly the following output structure:

<tool_decision>no_tool</tool_decision>
<action>{"type":"finalize"}</action>
<answer>FINAL_CPMPY_CODE</answer>

- The `no_tool` tag is a fixed output marker, not a routing decision.
- Replace FINAL_CPMPY_CODE with the complete executable CPMpy code.
- Output each required tag exactly once, with no additional text outside the
  tags.
```

Direct Testing Prompt (User)

```
Problem:
{problem_description}

Generate the complete final CPMpy model directly. Output only the required
  tags and the code inside the answer block.
```