CO-AGENT: COST-AWARE COMBINATORIAL OPTIMIZATION VIA VERIFIER-GUIDED DISTILLATION AND SKILL EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-model agents can generate optimization programs, invoke mathematical solvers, and repair execution failures, but strong performance often comes at the cost of large models and repeated inference. We investigate how small language models can coordinate these capabilities to solve combinatorial optimization (CO) problems under limited computational budgets. We introduce CO-Agent, a framework that combines a compact planner and generator with a solver harness and an evolving repository of reusable optimization skills. The framework integrates three complementary learning mechanisms. First, cost-aware planner optimization learns to allocate generation, solver, and refinement resources from the outcomes of complete agent trajectories. Second, verifier-guided distillation trains the generator using re-executed and verified teacher corrections on states visited by the student. Third, repository-level skill evolution evaluates candidate skills by their marginal contribution to deployed-agent utility under natural retrieval, accounting for retrieval interference and negative transfer. A shared execution–verification loop grounds all three mechanisms in feasibility, solution quality, and computational cost. Experiments on CO-Bench, FrontierCO, and HeuriGym demonstrate that CO-Agent achieves competitive solution quality with lower inference cost, while ablation and continual-evolution studies show the complementary benefits of the three mechanisms and sustained improvements in skill reuse.

1 INTRODUCTION

Combinatorial optimization (CO) underlies many decision-making problems in routing, scheduling, resource allocation, and network design. Solving these problems requires executable procedures that satisfy constraints and produce high-quality solutions within practical computational limits. Large language models (LLMs) have opened new possibilities for automating this process. FunSearch and EoH demonstrate the potential of language models for program search and heuristic discovery (Romera-Paredes et al., 2024; Liu et al., 2024), while ReEvo and MCTS-AHD explore reflective feedback and structured search to improve algorithm generation (Ye et al., 2024; Zheng et al., 2025). These advances motivate agents that can coordinate the complete optimization workflow, from selecting a solving strategy to executing, verifying, and refining its implementation.

A central challenge is achieving this capability at a practical deployment cost. Larger models and additional search can improve performance, but each generation, execution, and repair consumes resources. For a small language model, the challenge is especially pronounced: an unreliable solver proposal can trigger repeated repairs and erase the savings from cheaper inference. We therefore study how to build a capable CO agent around a compact model by combining learned planning and code generation with reusable optimization skills and deterministic tools. This design requires addressing three coupled challenges. First, the value of additional computation is state dependent: model calls, solver invocations, and repair attempts should be allocated according to the current solution, execution feedback, and remaining budget, requiring planning policies that jointly consider solution quality and computational cost (Li et al., 2026b). Second, successful teacher demonstrations may fail to cover the execution errors, infeasible solutions, and weak heuristics actually encountered by a compact student. Although on-policy distillation reduces this training–inference mismatch by learning from student-generated behavior (Agarwal et al., 2024), optimization agents additionally require

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.

execution-grounded supervision that identifies which teacher corrections yield verified improvements in feasibility, objective quality, and cost. Third, reusable skills must be evolved at the repository level rather than judged in isolation. While archetype-based skill learning and heuristic-set evolution demonstrate the value of reusable and complementary procedures (Yang et al., 2026; Liu et al., 2026), inserting a new skill can alter retrieval, consume context, and interfere with existing behaviors. Effective skill evolution must therefore measure its contribution through the behavior of the deployed agent across tasks.

To address these challenges, we introduce CO-Agent, a compact and execution-grounded framework that jointly learns computation allocation, solver generation and repair, and repository-level skill evolution through a shared execution–verification loop. Given a problem description, a numerical instance, and a computational budget, a compact planner selects skills and allocates resources, while a compact generator produces executable solver code. The executor and verifier return feasibility, objective quality, computational cost, and diagnostic feedback, which are fed back into subsequent planning and repair. On top of this loop, three complementary learning mechanisms improve the planner, generator, and skill repository, respectively. Model training and repository evolution use training and development data, whereas static evaluation freezes all learned components while retaining within-task execution, verification, and repair.

Our contributions are as follows:

- **Cost-aware router planner (CARP).** We develop CARP, a budget-aware planner trained in-the-flow to jointly optimize feasibility, solution quality, and computational cost.
- **Verifier-guided on-policy distillation for CO (CO-VGOPD).** We propose CO-VGOPD, which distills verifier-confirmed teacher repairs collected from student-visited execution states.
- **Agent-utility-guided skill evolution (AUSE).** We introduce AUSE, which combines complementary skill evolution with repository-level admission based on marginal utility and negative transfer.
- **Comprehensive evaluation.** Experiments on CO-Bench, FrontierCO, and HeuriGym demonstrate strong optimization performance and favorable quality–cost trade-offs, with ablations and continual-evolution studies validating the contributions of the proposed mechanisms.

2 Related Work

LLMs for CO. LLMs have been applied to optimization modeling, solver-code generation, and direct solution construction, with OR-R1 and LLMCoSolver exploring specialized modeling and end-to-end CO solving (Ding et al., 2026; Jiang et al., 2025). CO-Bench, FrontierCO, and HeuriGym further benchmark diverse real-world CO tasks, challenging instances, and executable heuristic construction (Sun et al., 2026; Feng et al., 2026; Chen et al., 2026b). These studies mainly evaluate individual generated models, solutions, or heuristics, whereas CO-Agent focuses on a persistent compact agent that allocates computation, executes solvers, interprets verification feedback, and repairs decisions under an explicit resource budget.

Agent training through reinforcement learning and distillation. GRPO provides a memory-efficient alternative to PPO by estimating advantages from relative rewards among sampled responses without a separate value model (Shao et al., 2024), while AgentFlow and related cost–aware methods optimize agent behavior and quality–cost trade-offs in interactive execution (Li et al., 2026b). For capability transfer, GKD reduces the training–inference mismatch through student-generated behavior (Agarwal et al., 2024), while sequence-level distillation and OPD respectively distill teacher-generated solutions and teacher supervision on student trajectories (Kim & Rush, 2016; Li et al., 2026a); TGOPD further introduces verifier-based teacher gating between OPD and verifier-grounded GRPO (Zhang et al., 2026). CO-Agent builds on these directions with execution-grounded training for CO: CARP optimizes planning under explicit cost, while CO-VGOPD trains on student-visited failures and retains teacher corrections only after executable verification confirms improvement.

Automated algorithm discovery and skill evolution. LLMs have been used as search operators for algorithm discovery through candidate selection and refinement (Chen et al., 2021; Shinn et al.,

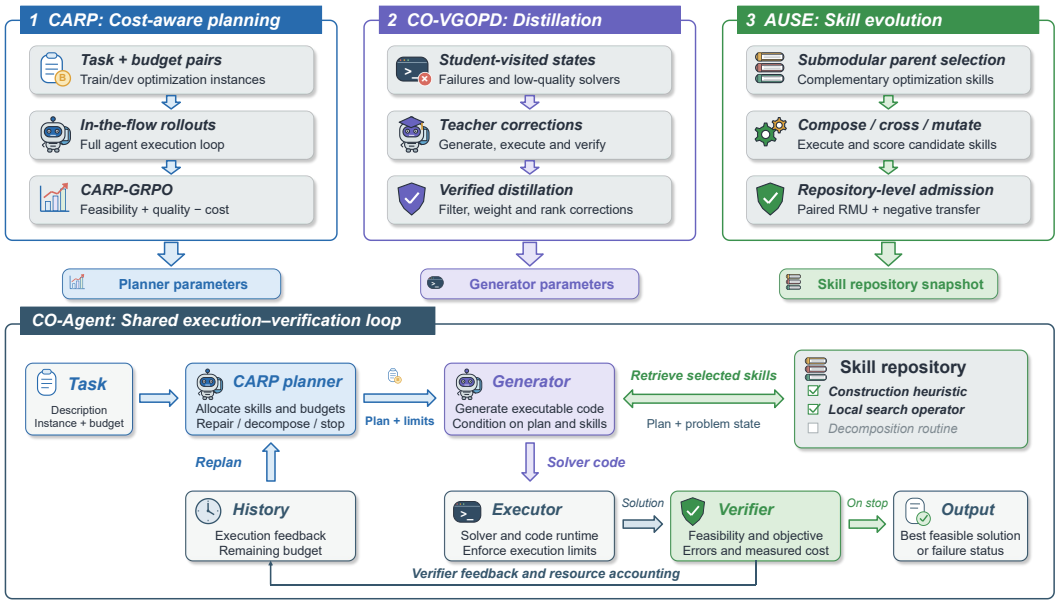


Figure 1: Overview of CO-Agent. CARP learns cost-aware planning from complete execution–verification trajectories; CO-VGOPD distills re-executed and verifier-confirmed teacher corrections on student-visited states; and AUSE evolves complementary skills and admits them according to repository marginal utility under natural retrieval. At deployment, the frozen planner and generator retrieve skills, generate executable solver code, invoke optimization tools, and use deterministic verifier feedback to repair, decompose, continue, or terminate within a computational budget.

2023), evolutionary program search (Romera-Paredes et al., 2024), and heuristic evolution with reflection or tree search (Liu et al., 2024; Ye et al., 2024; Zheng et al., 2025); Chain of Experts further employs multi-agent cooperation for solution debugging (Xiao et al., 2024). HiFo-Prompt further combines hindsight knowledge accumulation with foresight-based population control to adaptively guide LLM-driven heuristic evolution (Chen et al., 2026a). Beyond algorithm discovery, OptSkills, Anything2Skill, Skill1, and SAGE study reusable procedural skills and their acquisition or evolution (Yang et al., 2026; Pan et al., 2026; Shi et al., 2026; Wang et al., 2026). In contrast, AUSE evaluates skills after insertion into the deployed repository under natural retrieval and admits them according to repository-level marginal utility and controlled negative transfer, explicitly accounting for retrieval competition, context cost, and incorrect activation.

3 Problem Formulation

We consider a CO problem specified by a natural-language description d, which defines the objective, constraints, and required input/output format. Let $\mathcal{X}$ be the feasible set and $f(x)$ the objective value of a feasible solution $x \in \mathcal{X}$. Unlike single-shot generation, CO-Agent solves a problem through a verifier-grounded interaction loop. At turn t, a planner P selects a solving plan from the current history $\mathcal{H}_t$ and skill repository $\mathcal{K}$, a generator G produces executable solver code, and the executor E and verifier V return a candidate solution and structured feedback:

$$\begin{aligned} p_t &= P(d, \mathcal{K}, \mathcal{H}_t), \quad & o_t &= G(d, p_t), \\ x_t &= E(o_t), & y_t &= V(o_t, x_t). \end{aligned} \tag{1}$$

Here y_t records execution status, feasibility, verified objective quality, constraint violations, runtime information, and diagnostic signals. The tuple (p_t, o_t, x_t, y_t) is appended to $\mathcal{H}_t$, so later decisions condition on the consequences of earlier ones. A T turns trajectory is $\tau = \{(p_t, o_t, x_t, y_t)\}_{t=1}^{T}$, and its best verified feasible solution is denoted by $x_\tau^\star$.

Solving also consumes model, token, solver, execution, and wall-clock resources. We aggregate them into a trajectory cost $C(\tau)$. Given a computational budget B, the formulation is

$$\max_{\tau} \; f(x_\tau^\star) \qquad \text{s.t.} \quad x_\tau^\star \in \mathcal{X},\; C(\tau) \le B. \tag{2}$$

Thus, the agent must jointly decide how to solve and how much computation to spend. The following section learns these decisions at three complementary levels: planning, solver generation, and reusable skills. Additional trajectory cost definitions are provided in Appendix A.1.

4 CO-AGENT

CO-Agent couples three learning mechanisms through the same execution–verification loop in Eq. (1). Figure 1 summarizes the complete framework.

4.1 COST-AWARE ROUTER PLANNER

The CARP replaces fixed routing with a trainable policy π_ϕ that allocates computation according to the evolving solving state. At turn t, the planner observes the problem, available skills and tools, execution history, and remaining budget:

$$h_t = (d, \mathcal{K}, \mathcal{T}, \mathcal{H}_t, B - C_t)\,, \quad p_t \sim \pi_\phi(\cdot \mid h_t), \tag{3}$$

where $\mathcal{T}$ is tool set, C_t denotes trajectory cost at turn t, and p_t specifies the solving plan, such as generation, repair, decomposition, solver invocation, or termination, together with the selected skill/tool and resource allocation.

Cost-aware trajectory reward. Because the utility of a planning action depends on its downstream execution outcome, CARP is trained from complete solving trajectories. We define the relative budget consumption as $\rho_B(\tau) = \min\left\{\frac{C(\tau)}{B}, \rho_{\max}\right\}$, where $\rho_{\max}$ is a fixed upper bound on the cost-to-budget ratio to prevent expensive trajectories from producing unbounded negative rewards. Then, the trajectory-level reward is defined as

$$R_B(\tau) = \lambda_v v + \lambda_f v \widetilde{f}(x_\tau^\star) - \lambda_c \rho_B(\tau) - \lambda_o \left[\rho_B(\tau) - 1\right]_+, \tag{4}$$

where $v \in \{0, 1\}$ indicates whether a feasible solution is found, $\widetilde{f}(x_\tau^\star) \in [0, 1]$ denotes the normalized verified objective value, and $[x]_+ = \max(x, 0)$. The non-negative coefficients λ_v, λ_f, λ_c, and λ_o control the relative importance of feasibility, solution quality, resource consumption, and budget violation, respectively. This formulation jointly rewards feasibility and verified solution quality while penalizing resource consumption and budget violation.

CARP-GRPO training. For a group of trajectories sampled from the same problem–budget pair, their rewards are normalized into group-relative advantages A_i. Then, CARP maximizes the expected return

$$\begin{aligned}\mathcal{J}_{\text{CARP}}(\phi) = \mathbb{E}\left[\frac{1}{N_g}\sum_{i=1}^{N_g}\frac{1}{T_i}\sum_{t=1}^{T_i}\min\left(\rho_{i,t}A_i,\; \text{clip}\left(\rho_{i,t}, 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}}\right)A_i\right)\right] \\ - \beta D_{\text{KL}}\left(\pi_\phi \| \pi_{\text{ref}}\right),\end{aligned} \tag{5}$$

where N_g denotes the number of trajectories per problem–budget group, T_i is the number of decision turns in trajectory i, $\rho_{i,t} = \pi_\phi(p_{i,t} \mid h_{i,t})/\pi_{\text{old}}(p_{i,t} \mid h_{i,t})$ is the policy ratio, $\text{clip}(\cdot)$ restricts the ratio to $[1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}}]$, and ϵ_{clip} is the clipping coefficient. $D_{\text{KL}}(\pi_\phi \| \pi_{\text{ref}})$ denotes the KL divergence from the current policy π_ϕ to the fixed reference policy π_{ref}, with β controlling the regularization strength. Detailed training algorithm are provided in Appendix A.2.

4.2 CO-SPECIFIC VERIFIER-GUIDED ON-POLICY DISTILLATION

A compact generator should learn from the failures it actually encounters during deployment. Therefore, CO-VGOPD treats the stronger teacher as a repair policy conditioned on student-visited execution states. For a generated solver o, returned solution x, and execution outcome y, we define

the CO-specific verified utility as $U_{\mathrm{CO}}(o,x,y) = \lambda_e z(y) + \lambda_f v(y) + \lambda_q v(y)\widetilde{f}(x) - \lambda_c C(o,x,y)$, where $z(y)$ and $v(y)$ indicate successful execution and feasibility, $\widetilde{f}(x) \in [0,1]$ is the normalized verified objective quality, and $C(o,x,y)$ denotes the normalized deployment-time execution cost of solver o. This utility distinguishes broken programs, infeasible solutions, weak feasible heuristics, and high-quality but unnecessarily expensive solvers.

Execution-grounded repair and weighting. For each on-policy student rollout i, CO-Agent executes the student solver (o_i^s, x_i^s, y_i^s), where o_i^s, x_i^s, and y_i^s denote the student solver, its returned solution, and the corresponding verifier diagnostics, respectively. Based on these execution results, the planner constructs a failure-conditioned repair context p_i^+, from which the teacher generates a repaired solver o_i^T. The repaired solver is then executed and verified in the same optimization environment, yielding (x_i^T, y_i^T). We measure the verified repair gain by $\Delta U_i = U_{\mathrm{CO}}(o_i^T, x_i^T, y_i^T) - U_{\mathrm{CO}}(o_i^s, x_i^s, y_i^s)$, where $U_{\mathrm{CO}}(\cdot)$ is the CO-Agent utility and ΔU_i is the resulting improvement. We retain the repair only if it is executable and $\Delta U_i \geq \delta$, where $\delta \geq 0$ is the minimum required improvement. For each retained repair, its Verified Repair Advantage is defined as $A_i^{\mathrm{VRA}} = \mathrm{clip}\left(\frac{[\Delta U_i]_{*+}}{\sigma_U + \epsilon_U}, 0, A_{\max}\right)$, where $\sigma_U > 0$ is the mini-batch standard deviation of verified utility differences, $\epsilon_U > 0$ is a numerical stability constant, and $A_{\max} > 0$ prevents a small number of unusually large improvements from dominating training. We further let $r_i \in \{0,1\}$ indicate whether the repair converts an initially non-executable or infeasible solver into an executable and feasible one, and assign the sampling weight $w_i = 1 + \alpha_a A_i^{\mathrm{VRA}} + \alpha_r r_i$, where $\alpha_a, \alpha_r \geq 0$ control the contributions of verified improvement and feasibility recovery, respectively.

Verifier-guided OPD. Accepted repairs are distilled under the same failure-conditioned contexts encountered at inference time. When teacher logits are available, we use token-level knowledge distillation. To preserve the execution-induced preference for verified repairs, we additionally contrast each accepted teacher repair with its preceding student solver. The resulting objective is

$$\mathcal{L}_{\text{CO-VGOPD}} = \frac{1}{|\mathcal{D}_{\mathrm{corr}}|} \sum_{i \in \mathcal{D}_{\mathrm{corr}}} w_i \mathcal{L}_{\mathrm{distill}}^{(i)} + \lambda_p \mathcal{L}_{\mathrm{pref}}, \tag{6}$$

where $\mathcal{L}_{\mathrm{distill}}^{(i)} = \frac{1}{|o_i^T|} \sum_{t=1}^{|o_i^T|} D_{\mathrm{KL}}(\pi_T(\cdot \mid c_{i,t}) \,\|\, \pi_\theta(\cdot \mid c_{i,t}))$, denotes the selected token-level distillation loss, $\mathcal{L}_{\mathrm{pref}} = -\sum_{i \in \mathcal{D}_{\mathrm{corr}}} A_i^{\mathrm{VRA}} \log \sigma\left(S_\theta(o_i^T \mid d_i, p_i^+) - S_\theta(o_i^s \mid d_i, p_i^+)\right)$, denotes verifier-induced pairwise preference loss. For any code sequence o, define its length-normalized student score under the repair context as $S_\theta(o \mid d, p) = \frac{1}{|o|} \sum_{j=1}^{|o|} \log \pi_\theta\left(o_j \mid d, p, o_{<j}\right)$. c_i is the failure-conditioned context, and λ_p control the preference contribution. $\sigma(a) = 1/(1+\exp(-a))$ is the sigmoid function and $\mathcal{D}_{\mathrm{corr}}$ denotes the set of verifier-approved correction pairs. By repeatedly collecting failures from the current student and distilling only verifier-confirmed repairs, CO-VGOPD concentrates learning on execution errors, constraint violations, weak heuristics, and inefficient solver behaviors that the compact model itself visits. The complete training algorithm are provided in Appendix A.3.

4.3 Agent-Utility-Guided Skill Evolution

The skill repository should improve the deployed agent rather than merely accumulate individually strong heuristics. Therefore, AUSE performs two-stage selection: diversity-aware evolution within a local population, followed by repository-level admission under natural retrieval. Each skill s records its triggering archetype, algorithmic principle, executable procedure, applicability and failure conditions, and historical utility statistics. Candidate skills are evaluated by running the complete CO-Agent loop on a neighborhood of structurally related problems. Their fitness is measured by the verifier-grounded utility $J(s \mid d)$, obtained by averaging an execution-level utility $U_{\mathrm{evo}}(\tau)$ that jointly accounts for feasibility, normalized objective quality, and trajectory cost.

Complementary parent selection and evolution. Skill selection should not only consider fitness but also diversity. Let $\Phi(d) = \{\chi_h\}$ denote structural or algorithmic attributes of problem d, and let $m(s, \chi_h) \in [0,1]$ measure how strongly skill s covers attribute χ_h. Given the current repository $\mathcal{K}$, AUSE selects a parent subset $\mathcal{R} \subseteq \mathcal{K}$ with at most K_{par} skills by maximizing

$$F_d(\mathcal{R}) = \eta_{\mathrm{cov}} \sum_h \max_{s \in \mathcal{R}} m(s, \chi_h) + \eta_{\mathrm{fit}} \sum_{s \in \mathcal{R}} J(s \mid d), \quad |\mathcal{R}| \leq K_{\mathrm{par}}. \tag{7}$$

where $\eta_{\text{cov}}, \eta_{\text{fit}} \geq 0$ balance structural coverage and verified fitness. The coverage term is monotone submodular and the fitness term is modular, so greedy selection preserves the standard $(1 - 1/e)$ approximation guarantee while balancing verified performance and complementarity (Nemhauser et al., 1978). Then, LLM-based meta-operators compose, crossover, mutate, or repair the selected parents, and every offspring must be executed and verified before entering the candidate population.

Repository-level admission. High local fitness does not guarantee that a skill improves the deployed system: inserting a new skill may alter retrieval, consume additional context, or activate on inappropriate tasks. Therefore, AUSE evaluates each candidate $s^\star$ through a counterfactual repository test on a held-out admission neighborhood, comparing the naturally retrieved agent before and after inserting the candidate. We define its Repository Marginal Utility (RMU) together with its Negative Transfer Rate (NTR) as

$$\begin{aligned} \mathrm{RMU}(s^\star \mid \mathcal{K}) &= \frac{1}{L_{\text{adm}}} \sum_{\ell=1}^{L_{\text{adm}}} \left[U_{\text{evo}}\left(\tau^{\text{nat}}_{\ell,\mathcal{K}\cup\{s^\star\}}\right) - U_{\text{evo}}\left(\tau^{\text{nat}}_{\ell,\mathcal{K}}\right) \right], \\ \mathrm{NTR}(s^\star \mid \mathcal{K}) &= \frac{1}{L_{\text{adm}}} \sum_{\ell=1}^{L_{\text{adm}}} \mathbb{I}\left[U_{\text{evo}}\left(\tau^{\text{nat}}_{\ell,\mathcal{K}\cup\{s^\star\}}\right) < U_{\text{evo}}\left(\tau^{\text{nat}}_{\ell,\mathcal{K}}\right) \right]. \end{aligned} \tag{8}$$

A candidate is admitted only when the paired-bootstrap $\mathrm{LCB}_{95\%}(\mathrm{RMU})$ exceeds the minimum improvement threshold δ_{RMU} and its $\mathrm{NTR}(s^\star \mid \mathcal{K})$ does not exceed $\nu_{\max}$. This makes the repository, rather than the isolated skill, the unit of improvement: a candidate survives only when its insertion yields a reliable end-to-end gain without excessive negative transfer to other tasks. Full evolution operators, utility definitions, statistical tests, and algorithmic details are provided in Appendix A.4.

5 Experiments

5.1 Experimental Setup

The experiments answer four research questions: **(1) Can compact language models become strong general-purpose CO agents? (2) What enables compact models to overcome the limitations of model scale? (3) Can cost-aware planning improve the quality–cost trade-off? (4) Can solver skills evolve continually to improve agent-level performance?**

Benchmarks. We evaluate on three complementary benchmark suites. CO-Bench (Sun et al., 2026) measures broad CO capability across heterogeneous problem families and serves as the primary benchmark for solver generation and feasibility. FrontierCO (Feng et al., 2026) stresses large and difficult instances for which solution quality under limited computation is particularly important. HeuriGym (Chen et al., 2026b) evaluates heuristic discovery on low-exposure problems and therefore provides a useful test of algorithmic generalization rather than memorization. This combination probes breadth, scale, and low-exposure heuristic design.

Baselines. For RQ1, We organize baselines into three groups. Base LLM and Reasoning LLM generate a solver in a single call at multiple model scales. LLM Agent includes Best-of-N, GreedyRefine, FunSearch, ReEvo, EoH, HiFo-Prompt, MSTC-AHD and Chain of Experts. For RQ2, we additionally compare SFT, GRPO, vanilla OPD, TGOPD, and sequence-level distillation. For RQ3, we compare fixed allocation, rule router, and an outcome-only Flow-GRPO planner. For RQ4, we compare static skills, random evolution, a fitness-only genetic algorithm, HiFo-Prompt, and OptSkills.

5.2 RQ1: Can Compact Models Become Strong CO Agents?

Main results. Table 1 shows that the 4B CO-Agent achieves the strongest overall performance across all three benchmarks. Here, Gap measures the relative distance to the reference solution, with lower values indicating better solution quality, while FR denotes the feasible-solution rate, with higher values indicating more reliable solver generation. On CO-Bench, which emphasizes broad solver generation across heterogeneous problem families, CO-Agent achieves the lowest gap of 4.7 and the highest FR of 92.0%, reducing the gap by 2.7 points over QwQ-32B while using one eighth of the model size. On FrontierCO, which stresses large and difficult instances under limited computation, it achieves a gap of 26.4 and an FR of 95.2%. On HeuriGym, which targets low-exposure heuristic

Table 1: Main comparison across CO benchmarks. Lower is better for Gap; higher is better for FR.

Category	Method	CO-Bench		FrontierCO		HeuriGym	
		Gap ↓	FR ↑	Gap ↓	FR ↑	Gap ↓	FR ↑
Base LLM	Qwen2.5-7B	24.8±1.9	46.5±2.3	63.5±2.0	62.0±2.1	70.0±2.5	44.0±1.8
	Qwen2.5-32B	19.5±2.2	57.0±1.9	53.0±2.3	72.5±2.4	62.0±2.5	53.0±2.5
	Qwen2.5-Coder-32B	16.0±1.7	64.0±2.7	47.5±2.4	76.5±2.0	57.0±2.0	58.0±2.8
Reasoning	Qwen3-4B	12.3±1.6	72.4±2.0	45.1±1.3	77.6±1.8	53.2±1.2	62.0±1.7
	DeepSeek-R1-8B	11.4±1.3	74.2±1.4	41.5±1.5	80.5±1.5	50.5±1.1	64.5±2.1
	Qwen3-8B	10.6±1.6	76.9±1.3	40.8±1.4	81.5±2.0	49.0±1.6	66.5±1.5
	Gemma3-12B	9.9±1.1	78.5±1.7	39.0±1.6	82.8±1.7	47.2±1.4	69.0±1.4
	Qwen3-30B-A3B	9.0±1.5	80.2±1.8	35.8±1.2	85.0±1.5	44.5±1.2	72.5±1.4
	Qwen3-32B	8.2±1.1	82.6±1.5	34.2±1.2	86.5±1.7	42.7±1.4	75.5±2.0
	QwQ-32B	7.4±1.4	83.9±1.3	31.8±1.5	88.2±1.5	40.8±1.5	78.5±1.8
LLM Agent	BestOfN	9.7±0.7	80.5±1.1	39.2±1.4	86.0±1.5	47.1±1.2	78.0±1.5
	Greedy Refine	8.9±1.1	82.4±1.4	37.5±1.0	87.8±1.2	45.2±1.3	80.5±0.9
	FunSearch	8.4±1.5	83.0±1.1	36.6±0.7	88.4±1.2	44.4±0.9	82.0±0.8
	ReEvo	7.6±1.1	84.3±1.2	34.8±1.4	89.6±1.6	41.9±1.3	84.5±1.1
	EoH	7.2±1.0	84.9±1.4	33.9±0.7	90.0±1.3	41.0±0.8	85.5±1.1
	HiFo-Prompt	6.8±1.4	85.8±1.3	32.7±0.8	90.9±1.4	39.5±1.3	86.8±0.8
	MSTC-AHD	5.5±0.7	89.5±1.2	29.8±1.2	92.5±1.6	35.0±1.4	89.5±1.5
	Chain of Experts	5.2±1.0	90.3±1.2	29.0±1.2	92.9±1.3	34.4±1.2	89.8±1.2
Ours	**CO-Agent (4B)**	**4.7±0.8**	**92.0±0.8**	**26.4±0.5**	**95.2±1.1**	**32.9±0.4**	**90.5±0.8**

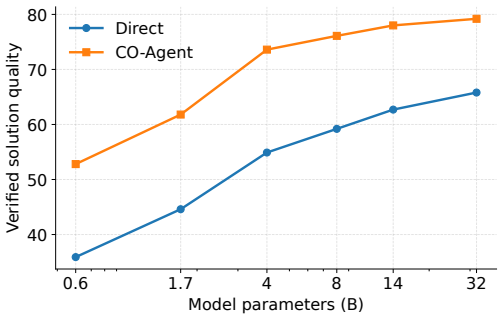


Figure 2: Parameter scaling. CO-Agent shifts the capability curve upward across model sizes; the 4B agent exceeds substantially larger direct-generation models.

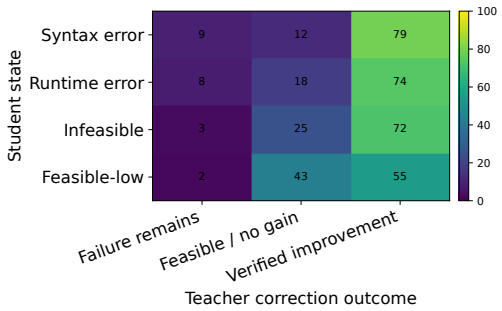


Figure 3: Repair transition matrix. CO-VGOPD converts a large fraction of syntax, runtime, and feasibility failures into verifier-confirmed improvements.

discovery, it obtains a gap of 32.9 and an FR of 90.5%, indicating consistent generalization beyond common optimization patterns. Overall, CO-Agent achieves both lower gaps and higher feasibility across all three evaluation settings.

Parameter scaling analysis. Figure 2 compares performance across model scales. Direct generation improves as the model becomes larger, while CO-Agent consistently achieves higher performance at the same scale. Notably, the 4B CO-Agent already outperforms direct generation with a 32B model. These results show that the gains come primarily from the agentic optimization loop rather than model scaling alone. Larger backbones further improve CO-Agent, but with diminishing returns.

5.3 RQ2: What Compensates for Model Scale?

RQ1 establishes that the compact agent can outperform much larger direct models. We next isolate where the gain comes from, focusing on CO-VGOPD and the interaction between trainable and non-parametric components.

CO-VGOPD versus standard training. Table 2 compares generator-training objectives under the same 4B backbone and same compute budget. CO-VGOPD achieves the best results on all four metrics, reaching 96.7% execution rate, 91.6% FR, a 5.8 gap, and 69.0% repair rate. Compared

Table 2: Generator-training comparison with the same compact backbone and evaluation budget.

Training	Exec ↑	FR ↑	Gap ↓	Repair ↑
Base 4B	79.1	70.4	13.2	18.0
SFT	88.0	78.2	10.8	34.0
GRPO	90.9	82.6	9.1	43.0
Vanilla OPD	91.7	83.4	8.7	45.0
TGOPD	93.0	85.2	7.9	49.0
Sequence Distill	91.2	82.9	8.8	44.0
CO-VGOPD	**96.7**	**91.6**	**5.8**	**69.0**

Table 3: Component ablations. Cost is normalized using the same definition as the full system.

Variant	Gap ↓	FR ↑	Cost ↓
Full CO-Agent	**4.7**	**95.2**	0.66
w/o CARP	4.9	94.6	1.00
w/o CO-VGOPD	7.4	87.9	0.66
w/o AUSE	6.1	91.3	0.64

with TGOPD, it improves FR by 6.4 points and reduces the gap by 2.1 points, while increasing successful repair by 20.0 points. The consistent gains in execution, feasibility, solution quality, and repair indicate that conditioning distillation on student-executed failures and retaining only verifier-confirmed repairs provides a more effective training signal than standard on-policy distillation alone.

What does the teacher repair? Figure 3 traces the state of student-generated solvers before and after teacher correction. The majority of syntax, runtime, and infeasibility failures are converted into verifier-confirmed improvements. Even when the student already produces a feasible but weak solver, more than half of the corrections improve the verified utility. This is the behavior targeted by CO-VGOPD: the teacher is not merely a source of correct final programs, but a repair policy acting on concrete failure states visited by the current student.

System ablations. Table 3 shows that all three core modules contribute complementary benefits. Removing CO-VGOPD causes the largest degradation in both solution quality and feasibility, increasing the gap from 4.7 to 7.4 and reducing FR from 95.2% to 87.9%. Removing AUSE also leads to a clear quality drop, with the gap increasing to 6.1 and FR decreasing to 91.3%, highlighting the contribution of adaptive skill evolution. In contrast, removing CARP has only a modest effect on gap and FR, but increases normalized cost from 0.66 to 1.00, confirming its primary role in computation allocation. Overall, the ablations show that CO-VGOPD and AUSE primarily improve solution quality and feasibility, while CARP improves cost efficiency, and their combination yields the strongest quality–cost trade-off.

5.4 RQ3: Can CARP Improve the Quality–Cost Trade-off?

The previous experiments use sufficiently generous budgets to study raw capability. RQ3 instead evaluates whether the planner can allocate computation efficiently when model tokens, solver time, repair turns, and wall-clock time are explicitly limited.

Quality–cost frontier. Figure 4 is the primary CARP result. A fixed configuration improves quality as more compute is allocated, but wastes budget on easy instances and does not react efficiently to failed solver executions. A hand-written rule router is more efficient, while an outcome-only Flow-GRPO planner benefits from in-the-flow training. CARP dominates all three over the full operating range: at comparable cost it reaches higher verified quality, and at comparable quality it terminates earlier. The result establishes that CARP contributes more than generic planner, conditioning decisions on the remaining budget and directly penalizing trajectory cost changes the learned allocation policy.

5.5 RQ4: Can Solver Skills Evolve Continually with AUSE?

AUSE targets a different source of capability from CO-VGOPD and CARP: the external repository of reusable optimization knowledge. We evaluate both local heuristic discovery and the harder repository-level question of whether an evolved skill improves the deployed agent after natural retrieval.

Skill-evolution quality. Table 4 compares AUSE with static retrieval, random evolution, fitness-only genetic search, and recent LLM-based heuristic/skill baselines. AUSE reaches a deployed score of 72.9, improving over OptSkills by 4.0 points. Its advantage is not limited to local fitness: RMU increases to 4.9, while NTR decreases to 6.1%. Relative to OptSkills, this corresponds to a 55.8%

Table 4: Skill evolution and repository-level admission.

Method	Deployed Score ↑	Held-out Gain ↑	RMU ↑	NTR ↓	Cost ↓	Diversity ↑
Static Skills	64.0	0.0	0.0	0.0	0.00	0.42
Random Evolution	65.4	1.1	0.6	22.4	1.00	0.55
Fitness-only GA	67.6	2.4	1.2	18.3	1.00	0.46
HiFo-Prompt	68.4	3.1	1.8	15.6	1.14	0.58
OptSkills	68.9	3.5	2.0	13.8	0.91	0.61
AUSE w/o Submodular	70.0	4.3	2.8	11.7	0.80	0.62
AUSE w/o RMU	69.6	5.6	3.1	19.8	0.73	0.76
AUSE	**72.9**	**6.8**	**4.9**	**6.1**	**0.72**	**0.79**

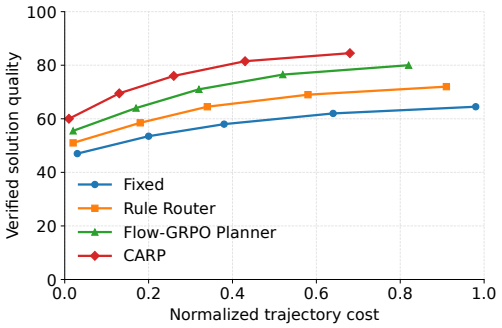


Figure 4: Quality–cost Pareto frontier. CARP consistently moves the frontier toward higher verified quality at lower trajectory cost.

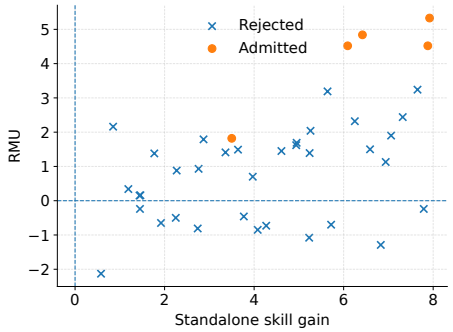


Figure 5: Standalone skill gain versus RMU. Several locally strong skills have negative repository-level effects; AUSE rejects them through repository-level admission.

reduction in negative transfer. The two AUSE ablations clarify the source of the gain. Removing submodular parent selection reduces diversity and final skill quality, while removing RMU admission leads to a high NTR of 19.8% even though locally evolved skills often appear strong.

Why repository-level admission matters. Figure 5 exposes a failure mode hidden by standalone skill evaluation. Among candidate skills with more than three points of standalone gain, approximately 25.9% have negative RMU after they are added to the repository. These skills can interfere with retrieval, consume context, or trigger on inappropriate problems even though they look strong when explicitly activated. AUSE's paired repository-level evaluation rejects such candidates, supporting the distinction between skill quality and -level utility.

6 Conclusion

We presented CO-Agent, a compact agent framework for CO that improves solution quality and computational efficiency through a unified execution–verification loop. Rather than relying on model scale or a fixed solving pipeline, CO-Agent learns to allocate computation with CARP, repair solver failures through CO-VGOPD, and evolve reusable optimization skills with AUSE. These components share an executable utility that jointly captures feasibility, objective quality, and computational cost, turning execution feedback into an active learning signal throughout the agent lifecycle. More broadly, our results show that learned routing, failure-driven adaptation, and skill evolution can complement model scaling for capable and resource-efficient CO agents. Future work will explore larger and more heterogeneous optimization domains, richer tool ecosystems, and continual skill evolution under non-stationary workloads.

AI USE STATEMENT

AI-assisted tools were used to aid and polish the writing and to support the retrieval and discovery of relevant literature. The authors independently verified the relevant content and are fully responsible for the research, results, analyses, and conclusions presented in this paper.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, 2024.

Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. Hifo-prompt: Prompting with hindsight and foresight for llm-based automatic heuristic design. In *International Conference on Learning Representations*, 2026a.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, Samitha Samaranayake, Carla P. Gomes, and Zhiru Zhang. Heurigym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. In *International Conference on Learning Representations*, 2026b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mo Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

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.

Shengyu Feng, Weiwei Sun, Shanda Li, Ameet Talwalkar, and Yiming Yang. Frontierco: Real-world and large-scale evaluation of machine learning solvers for combinatorial optimization. In *International Conference on Learning Representations*, 2026.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. Large language models as end-to-end combinatorial optimization solvers. In *Advances in Neural Information Processing Systems*, 2025.

Yoon Kim and Alexander M. Rush. Sequence-level knowledge distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 1317–1327, 2016.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan-ang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *arXiv preprint arXiv:2604.13016*, 2026a.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. In-the-flow agentic system optimization for effective planning and tool use. In *International Conference on Learning Representations*, 2026b.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 2024.

Fei Liu, Yilu Liu, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Eoh-s: Evolution of heuristic set using llms for automated heuristic design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 37090–37098, 2026. doi: 10.1609/aaai.v40i43.41038.

George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978. doi: 10.1007/BF01588971.

Qianjun Pan, Yutao Yang, Junsong Li, Jie Zhou, Kai Chen, Xin Li, Qin Chen, and Liang He. Anything2skill: Compiling external knowledge into reusable skills for agents. *arXiv preprint arXiv:2606.09316*, 2026.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

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.

Yaorui Shi, Yuxin Chen, Zhengxi Lu, Yuchun Miao, Shugui Liu, Qi Gu, Xunliang Cai, Xiang Wang, and An Zhang. Skill1: Unified evolution of skill-augmented agents via reinforcement learning. *arXiv preprint arXiv:2605.06130*, 2026.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Neural Information Processing Systems*, 2023.

Weiwei Sun, Shengyu Feng, Shanda Li, and Yiming Yang. Co-bench: Benchmarking language model agents in algorithm search for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 33126–33134, 2026. doi: 10.1609/aaai.v40i39.40596.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1529–1550. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.69.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When llms meet complex operations research problems. In *International Conference on Learning Representations*, volume 2024, pp. 48519–48537, 2024.

Haochen Yang, Ke Zhao, Mengyuan Ma, Xingyu Lu, Xiangfeng Wang, and Hong Qian. Optskills: Learning generalizable optimization skills from problem archetypes via cluster-based distillation. *arXiv preprint arXiv:2605.29829*, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, 2024.

Zhiwei Zhang, Zechen Sun, Fei Zhao, Kang Peng, Bin Liang, Huayu Deng, Yao Hu, Kam-Fai Wong, and Mu Chuan. Verify before you distill: Prompt-level teacher gating for on-policy distillation. *arXiv preprint arXiv:2609.02998*, 2026.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373. PMLR, 2025.

A Additional Method Details

This appendix collects the definitions, training objectives, implementation choices, and algorithmic pseudocode omitted from the main text for space. The main paper contains the conceptual flow and equations needed to understand the method; the material below specifies the full procedures used for implementation and reproduction.

A.1 Trajectory Cost Definitions

To explicitly characterize the computational resources consumed by CO-Agent, we record the cost of each trajectory τ along five complementary dimensions: model calls, token usage, solver computation, program execution, and end-to-end wall-clock time. These quantities capture different aspects of the resource consumption and are therefore reported separately before being aggregated into a unified trajectory cost.

Model cost. The model cost measures the number and scale of language-model invocations during a trajectory. Let $\mathcal{M}(\tau)$ denote the set of model calls in trajectory τ. We define the normalized model cost as

$$\widetilde{C}_{\text{model}}(\tau) = \frac{\sum_{i \in \mathcal{M}(\tau)} c_{\text{model}}(m_i)}{C^{\text{ref}}_{\text{model}}}, \tag{9}$$

where m_i is the model used by the i-th invocation, $c_{\text{model}}(m_i)$ is its predefined model-cost unit, and $C^{\text{ref}}_{\text{model}}$ is a reference cost used for normalization. This term captures the computational burden associated with invoking models of different scales, independently of the number of tokens generated in each invocation.

Token cost. The token cost measures the amount of language-model computation induced by the input and output sequences. For a trajectory containing N model calls, we define

$$C_{\text{token}}(\tau) = \sum_{i=1}^{N} \left(T_i^{\text{in}} + T_i^{\text{out}} \right), \tag{10}$$

where T_i^{in} and T_i^{out} denote the number of input and output tokens of the i-th model call, respectively. Its normalized form is

$$\widetilde{C}_{\text{token}}(\tau) = \frac{C_{\text{token}}(\tau)}{C^{\text{ref}}_{\text{token}}}. \tag{11}$$

Unlike model cost, which primarily reflects the number and scale of model invocations, token cost reflects how much language processing is performed within those invocations.

Solver cost. Solver cost measures the computational resources consumed by external optimization solvers invoked during a trajectory. Let $\mathcal{S}(\tau)$ denote the set of solver executions. We record the solver runtime as

$$C_{\text{solver}}(\tau) = \sum_{j \in \mathcal{S}(\tau)} T_j^{\text{solver}}, \tag{12}$$

where T_j^{solver} is the runtime of the j-th solver invocation. The normalized solver cost is

$$\widetilde{C}_{\text{solver}}(\tau) = \frac{C_{\text{solver}}(\tau)}{C^{\text{ref}}_{\text{solver}}}. \tag{13}$$

This term captures the computational burden introduced by optimization procedures such as exact, mathematical-programming, or heuristic solvers.

Execution cost. Execution cost measures the computational resources required to execute programs or solver code generated by the agent, excluding the runtime directly attributed to an external optimization solver. Let $\mathcal{E}(\tau)$ denote the set of program executions. We define

$$C_{\text{exec}}(\tau) = \sum_{k \in \mathcal{E}(\tau)} T_k^{\text{exec}}, \tag{14}$$

where T_k^{exec} is the runtime of the k-th program execution. Its normalized form is

$$\widetilde{C}_{\text{exec}}(\tau) = \frac{C_{\text{exec}}(\tau)}{C_{\text{exec}}^{\text{ref}}}. \tag{15}$$

This term is useful for distinguishing the cost of executing generated code from the cost of the optimization solver itself. For example, a trajectory that invokes a local-search heuristic may incur execution cost without incurring solver cost.

Wall-clock cost. Wall-clock cost measures the actual elapsed time from the beginning to the end of a trajectory:

$$C_{\text{wall}}(\tau) = t_{\text{end}}(\tau) - t_{\text{start}}(\tau). \tag{16}$$

We normalize it as

$$\widetilde{C}_{\text{wall}}(\tau) = \frac{C_{\text{wall}}(\tau)}{C_{\text{wall}}^{\text{ref}}}. \tag{17}$$

Unlike the component-wise computational costs above, wall-clock time captures the actual end-to-end latency experienced during execution and may include waiting, scheduling, I/O, and other overheads. Consequently, it is not necessarily equal to the sum of model, solver, and execution runtimes, particularly when operations are performed concurrently.

Unified trajectory cost. To combine the heterogeneous resource dimensions, we first normalize each cost component and then compute

$$C(\tau) = \alpha_m \widetilde{C}_{\text{model}}(\tau) + \alpha_t \widetilde{C}_{\text{token}}(\tau) + \alpha_s \widetilde{C}_{\text{solver}}(\tau) + \alpha_e \widetilde{C}_{\text{exec}}(\tau) + \alpha_w \widetilde{C}_{\text{wall}}(\tau), \tag{18}$$

where

$$\alpha_m, \alpha_t, \alpha_s, \alpha_e, \alpha_w \geq 0, \qquad \sum_{k \in \{m,t,s,e,w\}} \alpha_k = 1. \tag{19}$$

Normalization is essential because the five raw quantities have different units and scales. The reference quantities C_k^{ref} are fixed before evaluation and are shared across methods within the same experimental setting.

The resulting $C(\tau)$ provides a unified resource measure for cost-aware planning while retaining the underlying cost dimensions for detailed analysis. In particular, CARP can optimize the quality–cost trade-off using the aggregated cost, whereas the individual components are reported separately in the supplementary material to reveal where computational savings originate.

A.2 CARP: Training Procedure

CO-Agent optionally employs the CARP to adaptively allocate computation under a prescribed budget. CARP is implemented as a trainable planning policy π_ϕ operating inside the complete CO-Agent solving loop. At turn t, the planner receives the problem description d, available skills $\mathcal{K}$, optimization tools $\mathcal{T}$, interaction history $\mathcal{H}_t$, and remaining budget $B - C_t$, and samples the next solving operation according to Eq. (3). The selected operation may invoke generation, repair, decomposition, solver execution, or termination, together with the corresponding skill, tool, and resource allocation.

Given a sampled plan p_t, the downstream generator, executor, and verifier produce an executable solver, candidate solution, and verification outcome:

$$o_t = G(d, p_t), \qquad x_t = E(o_t, p_t, \mathcal{T}), \qquad y_t = V(o_t, x_t). \tag{20}$$

The resulting execution information is appended to $\mathcal{H}_t$, and the incurred cost is accumulated into C_t. Thus, CARP observes states induced by the actual CO solving process rather than by offline planner traces. During training, G, E, V, $\mathcal{K}$, and $\mathcal{T}$ are kept fixed, while only π_ϕ is optimized.

For each training problem d, we sample a computational budget B and generate N_g trajectories using the current behavior planner $\pi_{\phi_{\text{old}}}$. All trajectories in a group share the same problem–budget pair, allowing their planning strategies to be compared under an identical resource constraint. For trajectory τ_i, the best verified feasible solution $x_i^\star$ is selected from all solutions obtained during its

Algorithm 1 CARP-GRPO: In-the-Flow Cost-Aware Planner Optimization

Require: Training set $\mathcal{D}$; planner π_ϕ; generator G; executor E; verifier V; skill repository $\mathcal{K}$; tools $\mathcal{T}$; budget distribution q_B; group size N_g; maximum turns $T_{\max}$
Ensure: Optimized planner parameters $\phi^\star$

for each training iteration **do**
  Sample $d \sim \mathcal{D}$ and $B \sim q_B$
  Set behavior planner $\pi_{\phi_{\text{old}}} \leftarrow \pi_\phi$
  for $i = 1, \ldots, N_g$ **do**
    Initialize $\mathcal{H}_{i,1} \leftarrow \emptyset$, $C_{i,1} \leftarrow 0$
    for $t = 1, \ldots, T_{\max}$ **do**
      Construct $h_{i,t} = (d, \mathcal{K}, \mathcal{T}, \mathcal{H}_{i,t}, B - C_{i,t})$
      Sample $p_{i,t} \sim \pi_{\phi_{\text{old}}}(\cdot \mid h_{i,t})$
      Apply deterministic budget check to $p_{i,t}$
      if $p_{i,t}$ requests termination **then**
        break
      end if
      Generate $o_{i,t} \leftarrow G(d, p_{i,t})$
      Execute $x_{i,t} \leftarrow E(o_{i,t}, p_{i,t}, \mathcal{T})$
      Verify $y_{i,t} \leftarrow V(o_{i,t}, x_{i,t})$ and measure incremental cost $\Delta C_{i,t}$
      Update $C_{i,t+1} \leftarrow C_{i,t} + \Delta C_{i,t}$ and append $(p_{i,t}, o_{i,t}, x_{i,t}, y_{i,t})$ to $\mathcal{H}_{i,t}$
      if $C_{i,t+1} \geq B$ or $y_{i,t}$ indicates successful termination **then**
        break
      end if
    end for
    Store trajectory τ_i and compute $R_B(\tau_i)$ using Eq. (4)
  end for
  Compute group advantages A_i using Eq. (21)
  Update ϕ by maximizing Eq. (5)
end for
return $\phi^\star$

execution, and the final reward $R_B(\tau_i)$ is computed using Eq. (4). The normalized group advantage is

$$A_i = \frac{R_B(\tau_i) - \mu_R}{\sigma_R + \epsilon_{\text{adv}}}, \tag{21}$$

where $\epsilon_{\text{adv}} > 0$ is a small numerical constant. The group mean and standard deviation are computed as

$$\mu_R = \frac{1}{N_g} \sum_{i=1}^{N_g} R_B(\tau_i), \qquad \sigma_R = \sqrt{\frac{1}{N_g} \sum_{i=1}^{N_g} \left(R_B(\tau_i) - \mu_R\right)^2}. \tag{22}$$

The same trajectory-level advantage is assigned to every planning turn because the value of an intermediate planning decision is determined by its downstream verified outcome.

The planner is then updated using the clipped group-relative objective in Eq. (5), with optional reference-policy regularization controlled by β. This in-the-flow training procedure directly couples planning decisions with execution and verification outcomes, enabling CARP to learn computation allocation from end-to-end solution quality, feasibility, and cost. The detailed CARP-GRPO training procedure is summarized in Algorithm 1.

A.3 CO-VGOPD: Training Procedure

Algorithm 2 summarizes the complete procedure. Starting from an SFT-initialized generator, each on-policy round first lets the current student interact with the planner, executor, and verifier. Student-generated failures or low-quality solutions induce repair plans through the same history-feedback mechanism used during inference. The teacher produces a correction under the resulting repair context, after which both student and teacher candidates are evaluated by the same executor and verifier. Only verified improvements are retained, and their training contribution is proportional

Algorithm 2 CO-Specific Verifier-Guided On-Policy Distillation (CO-VGOPD)

Require: Training set $\mathcal{D}$; student π_θ; teacher π_T; planner P; executor E; verifier V; skill repository $\mathcal{K}$; number of rounds R; acceptance threshold δ
Ensure: Distilled student π_θ

for $r = 1, \ldots, R$ **do**
  Sample mini-batch $\mathcal{B} \subset \mathcal{D}$ and initialize $\mathcal{D}_{\text{corr}} \leftarrow \emptyset$
  for each $d_i \in \mathcal{B}$ **do**
    Construct current solving context and generate student solver $o_i^s \sim \pi_\theta$
    Execute and verify student: $(x_i^s, y_i^s) \leftarrow (E(o_i^s), V(o_i^s, x_i^s))$
    Augment the interaction history with (o_i^s, x_i^s, y_i^s)
    Construct failure-conditioned repair context p_i^+
    Generate teacher correction $o_i^T \sim \pi_T(\cdot \mid p_i^+)$
    Execute and verify teacher: $(x_i^T, y_i^T) \leftarrow (E(o_i^T), V(o_i^T, x_i^T))$
    Compute verified improvement ΔU_i
    if $z(y_i^T) = 1$ and $\Delta U_i \geq \delta$ **then**
      Compute A_i^{VRA}, repair indicator r_i, and weight w_i
      Add $(p_i^+, o_i^s, o_i^T, w_i)$ to $\mathcal{D}_{\text{corr}}$
    end if
  end for
  Compute $\mathcal{L}_{\text{distill}}$ and $\mathcal{L}_{\text{pref}}$ on $\mathcal{D}_{\text{corr}}$
  Update θ by minimizing $\mathcal{L}_{\text{CO-VGOPD}}$
end for
return π_θ

to the Verified Repair Advantage. Repeating this process shifts the distillation distribution toward execution failures, constraint violations, poor heuristics, and inefficient algorithms that the current student actually encounters, thereby reducing the mismatch between static offline distillation and deployment-time CO solving.

A.4 AUSE: Skill Evolution and Repository Admission

Let $\mathcal{K}$ denote the current skill repository, $\mathcal{K}_0$ the initial repository, $\mathcal{K}_{\text{final}}$ the evolved repository, and $\mathcal{D}_{\text{dep}}$ and $\mathcal{D}_{\text{hold}}$ denote the deployment and held-out evaluation sets, respectively. Let $U_{\text{evo}}(\tau)$ denote the verifier-grounded utility of a complete CO-Agent trajectory τ.

Deployed Score. The Deployed Score measures the overall capability of the deployed agent using the current skill repository under the natural skill retrieval mechanism:

$$\text{DS}(\mathcal{K}, \mathcal{D}_{\text{dep}}) = \frac{1}{|\mathcal{D}_{\text{dep}}|} \sum_{d \in \mathcal{D}_{\text{dep}}} U_{\text{evo}}\left(\tau_{d,\mathcal{K}}^{\text{nat}}\right), \tag{23}$$

where $\tau_{d,\mathcal{K}}^{\text{nat}}$ denotes the complete solving trajectory obtained for problem d using repository $\mathcal{K}$ and natural skill retrieval. A higher Deployed Score indicates stronger end-to-end performance of the deployed agent.

Held-out Gain. The Held-out Gain measures whether the evolved skills generalize beyond the problems used during skill evolution:

$$\text{HG} = S(\mathcal{K}_{\text{final}}, \mathcal{D}_{\text{hold}}) - S(\mathcal{K}_0, \mathcal{D}_{\text{hold}}), \tag{24}$$

where $S(\mathcal{K}, \mathcal{D})$ denotes the average verified solution score obtained by the CO-Agent using repository $\mathcal{K}$ on dataset $\mathcal{D}$. A larger Held-out Gain indicates stronger generalization of the evolved skill repository.

Skill Diversity. The Skill Diversity metric measures the degree of dissimilarity among skills maintained in the repository. Let $e(s)$ denote the representation of skill s and $d(\cdot, \cdot)$ denote a pairwise

distance function. We define repository diversity as

$$\mathrm{Div}(\mathcal{K}) = \frac{1}{|\mathcal{K}|(|\mathcal{K}|-1)} \sum_{\substack{s_i, s_j \in \mathcal{K} \\ i \neq j}} d\left(e(s_i), e(s_j)\right). \tag{25}$$

When cosine distance is used, the pairwise distance is

$$d(e_i, e_j) = 1 - \frac{e_i^\top e_j}{\|e_i\|_2 \|e_j\|_2}. \tag{26}$$

A higher Diversity indicates that the repository contains more complementary rather than redundant skills.

Skill Evolution. For the target description d, we construct a local evaluation neighborhood

$$\mathcal{N}(d) = \left\{ d^{(1)}, \ldots, d^{(L)} \right\}, \tag{27}$$

containing L training or validation problems with related structural properties. When a candidate skill s is explicitly activated on $d^{(\ell)}$, the fixed CO-Agent components produce a solving trajectory $\tau_{\ell,s}$ following the same $d \to p \to o \to x \to y$ interaction loop defined in Eq. (1). Let $x^\star_{\ell,s}$ denote the best verified feasible solution found along $\tau_{\ell,s}$ and let $y^\star_{\ell,s}$ denote its corresponding verifier result. We use the feasibility indicator $v(\cdot)$ and normalized objective $\widetilde{f}(\cdot)$ introduced above and define the skill-level trajectory utility as

$$U_{\mathrm{evo}}(\tau_{\ell,s}) = \eta_{\mathrm{feas}} v(y^\star_{\ell,s}) + \eta_{\mathrm{obj}} v(y^\star_{\ell,s}) \widetilde{f}(x^\star_{\ell,s}) - \eta_{\mathrm{cost}} \frac{C(\tau_{\ell,s})}{B_{\mathrm{evo}}}, \tag{28}$$

where $C(\tau_{\ell,s})$ is the trajectory cost defined in Eq. (18), $B_{\mathrm{evo}} > 0$ is the fixed per-rollout evaluation budget used during skill evolution, and $\eta_{\mathrm{feas}}, \eta_{\mathrm{obj}}, \eta_{\mathrm{cost}} \geq 0$ control the relative importance of feasibility, solution quality, and computational efficiency. If no feasible solution is found, we set $v(y^\star_{\ell,s}) = 0$ and the objective-quality term to zero. The verified fitness of skill s for target problem d is then

$$J(s \mid d) = \frac{1}{L} \sum_{\ell=1}^{L} U_{\mathrm{evo}}(\tau_{\ell,s}). \tag{29}$$

A high-fitness skill alone is not necessarily a good evolutionary parent, because repeatedly selecting near-duplicate heuristics can quickly collapse the search population. We therefore use submodular parent selection to balance empirical fitness and algorithmic complementarity. Let

$$\Phi(d) = \{\chi_1, \ldots, \chi_{H_{\mathrm{attr}}}\} \tag{30}$$

denote H_{attr} structural or algorithmic attributes extracted from d, such as routing structure, capacity constraints, decomposition opportunities, construction heuristics, neighborhood operators, and local-search mechanisms. For a skill s and attribute χ_h, let $m(s, \chi_h) \in [0, 1]$ quantify how strongly s covers that attribute, and let $\omega_h \geq 0$ denote its importance. At evolution round r, let $\mathcal{S}^{(r)}$ denote the current skill population and let $\mathcal{R} \subseteq \mathcal{S}^{(r)}$ be a candidate parent set. We define

$$F_d(\mathcal{R}) = \eta_{\mathrm{cov}} \sum_{h=1}^{H_{\mathrm{attr}}} \max_{s \in \mathcal{R}} m(s, \chi_h) + \eta_{\mathrm{fit}} \sum_{s \in \mathcal{R}} J(s \mid d), \tag{31}$$

subject to

$$|\mathcal{R}| \leq K_{\mathrm{par}}, \tag{32}$$

where K_{par} is the parent-set cardinality budget, $J(s \mid d) \in [0, 1]$ is a non-negative normalized version of Eq. (29), and $\eta_{\mathrm{cov}}, \eta_{\mathrm{fit}} \geq 0$ balance structural coverage and verified fitness. We define the coverage contribution of the empty set as zero. Under non-negative similarities and weights, the weighted max-coverage term is monotone submodular and the fitness term is modular; hence F_d is also monotone submodular. Greedy maximization under the cardinality constraint in Eq. (32) therefore achieves the standard $(1 - 1/e)$ approximation guarantee, while favoring parent skills that are both effective and complementary.

Given the selected parent set, an LLM-based evolutionary meta-operator Γ generates new structured skills. We distinguish Γ from the solver generator G: G produces executable solver code for a fixed solving plan, whereas Γ edits the skill representation itself. Specifically, Γ_{comp} composes complementary algorithmic modules from multiple parent skills, Γ_{cross} exchanges subprocedures between two parent skills, Γ_{mut} mutates algorithm choices, parameters, decomposition strategies, data structures, or local-search operators, and Γ_{repair} revises an unsuccessful offspring using its verifier feedback. Each generated offspring is executed through the complete CO-Agent loop on $\mathcal{N}(d)$ and receives a verified fitness through Eq. (29). We use elitist survival to preserve the best discovered skill and a second submodular selection step to maintain population diversity. Evolution stops after at most R_{evo} rounds or when the best fitness improvement remains below $\epsilon_{\mathrm{evo}} > 0$ for κ_{pat} consecutive rounds.

Local evolutionary fitness is only a first-stage criterion. A skill that performs well when explicitly activated may still degrade the deployed agent by interfering with natural retrieval, increasing context cost, or triggering on inappropriate problems. We therefore evaluate the final evolved candidate $s^\star$ at the repository level before admission. Let $\mathcal{N}_{\mathrm{adm}}(d) = \{\bar{d}^{(1)}, \ldots, \bar{d}^{(L_{\mathrm{adm}})}\}$ denote a held-out admission neighborhood containing L_{adm} related problems. For each $\bar{d}^{(\ell)}$, we run the complete agent under natural skill retrieval with the original repository $\mathcal{K}$ and with the augmented repository $\mathcal{K} \cup \{s^\star\}$, yielding trajectories $\tau^{\mathrm{nat}}_{\ell,\mathcal{K}}$ and $\tau^{\mathrm{nat}}_{\ell,\mathcal{K}\cup\{s^\star\}}$, respectively. The paired repository-level gain is

$$\Delta^{\mathrm{repo}}_{\ell}(s^\star \mid \mathcal{K}) = U_{\mathrm{evo}}\left(\tau^{\mathrm{nat}}_{\ell,\mathcal{K}\cup\{s^\star\}}\right) - U_{\mathrm{evo}}\left(\tau^{\mathrm{nat}}_{\ell,\mathcal{K}}\right), \tag{33}$$

and the Repository Marginal Utility (RMU) is

$$\mathrm{RMU}\left(s^\star \mid \mathcal{K}\right) = \frac{1}{L_{\mathrm{adm}}} \sum_{\ell=1}^{L_{\mathrm{adm}}} \Delta^{\mathrm{repo}}_{\ell}\left(s^\star \mid \mathcal{K}\right). \tag{34}$$

We additionally define the negative-transfer rate (NTR) as

$$\mathrm{NTR}\left(s^\star \mid \mathcal{K}\right) = \frac{1}{L_{\mathrm{adm}}} \sum_{\ell=1}^{L_{\mathrm{adm}}} \mathbb{I}\left[\Delta^{\mathrm{repo}}_{\ell}\left(s^\star \mid \mathcal{K}\right) < 0\right]. \tag{35}$$

Let $\mathrm{LCB}_{95\%}(\mathrm{RMU})$ denote the lower bound of a paired-bootstrap 95% confidence interval computed from the repository gains in Eq. (33). We admit $s^\star$ into the deployed repository only if

$$\mathrm{LCB}_{95\%}\left(\mathrm{RMU}(s^\star \mid \mathcal{K})\right) > \delta_{\mathrm{RMU}}, \qquad \mathrm{NTR}(s^\star \mid \mathcal{K}) \le \nu_{\max}, \tag{36}$$

where $\delta_{\mathrm{RMU}} \ge 0$ is the minimum reliable repository-level improvement and $\nu_{\max} \in [0, 1]$ is the maximum tolerated negative-transfer rate. This final natural-retrieval test prevents locally strong but globally disruptive skills from being added to the repository.

Algorithm 3 summarizes the complete procedure. Starting from problem-relevant elite skills retrieved from $\mathcal{K}$, the algorithm alternates between submodular parent selection, LLM-based composition/crossover/mutation, verifier-grounded offspring evaluation, and elitist population update. The locally best evolved skill is subsequently subjected to repository-level counterfactual evaluation under natural retrieval before admission.

B Experiment details

Models and training. Unless otherwise specified, CO-Agent uses a compact Qwwn3-4B generator and planner. We compare against direct-generation models ranging from 0.6B to 32B parameters to separate gains from agentic structure from gains due to parameter scaling. CO-VGOPD uses a stronger 14B-class teacher only during training; all reported student-side inference costs exclude teacher calls because the teacher is absent at deployment. The primary low-resource training setting uses FP16 and parameter-efficient tuning. Specifically, we use LoRA with rank $r = 16$, scaling factor $\alpha = 32$, and dropout 0.05, applied to the attention and MLP projection layers. Unless otherwise specified, the student models are trained with a maximum sequence length of 4096 tokens, per-device batch size 1, gradient accumulation of 16 steps, and a learning rate of 2×10^{-5}. CARP optimizes only the planner policy using GRPO with group-wise normalized rewards, while the generator parameters are frozen. CO-VGOPD optimizes only the generator using on-policy student trajectories;

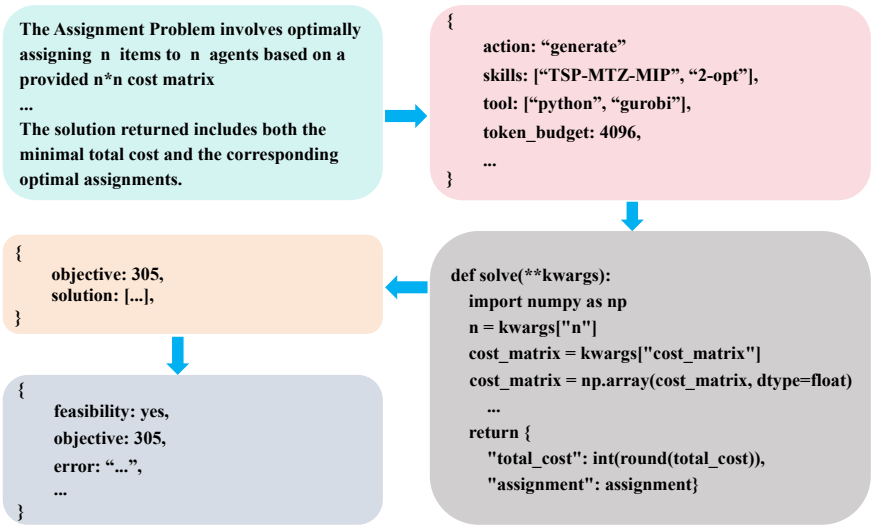


Figure 6: CO-Agent loop.

teacher-generated corrections are retained only when they pass the verifier and improve the verified objective by at least the prescribed improvement threshold δ. The planner remains frozen during CO-VGOPD training. AUSE does not update model parameters during repository evolution; instead, it modifies the external skill repository through skill generation, evaluation, and admission based on repository marginal utility and negative-transfer filtering, while keeping the underlying planner and generator fixed during each isolated module study. Unless otherwise specified, all experiments use five random seeds, and the reported results are averaged across seeds with standard deviations under same resource budget.

CO-Agent Loop. Figure 6 illustrates a representative CO-Agent solving loop. Given a combinatorial optimization problem, the planner first selects the next operation, relevant skill, and computational tool based on the current solving state and remaining budget. The generator then produces an executable solver, which is executed in the optimization environment and evaluated by the verifier for feasibility, objective quality, and execution cost. The resulting solution and verification feedback are appended to the interaction history and used to construct the next planning state. If the current solution is infeasible or insufficiently effective, CO-Agent performs targeted repair and re-enters the loop; otherwise, it terminates and returns the best verified solution obtained within the budget.

Robustness across problems. The aggregate gains are broad. Figure 7 reports problem-level win/tie/loss rates against the strongest non-ours baseline. CO-Agent wins roughly three quarters of the tasks on all three benchmark suites, while losses remain close to one tenth of the evaluation set. This distribution is important because a large mean improvement can otherwise be produced by a few unusually easy problem families.

Figure 8 further decomposes performance by representative CO structure. Improvements appear in routing, packing, scheduling, covering, assignment, facility-location, graph, and knapsack tasks. The strongest improvements occur on families that benefit from multiple rounds of executable repair and solver-aware planning, whereas the gains are smaller but still consistent on simpler constructive problems.

Sample efficiency. The data-scaling experiment in Figure 9 reveals a large gap between static imitation and execution-grounded corrective supervision. With only 256 corrective trajectories, CO-VGOPD reaches 65.4 CO-Bench score, already exceeding SFT trained with 2,048 trajectories (64.2). Thus, on-policy verifier-grounded corrections can be considerably more informative than simply collecting more successful demonstrations. The advantage continues as the training set grows rather than disappearing at larger data scales.

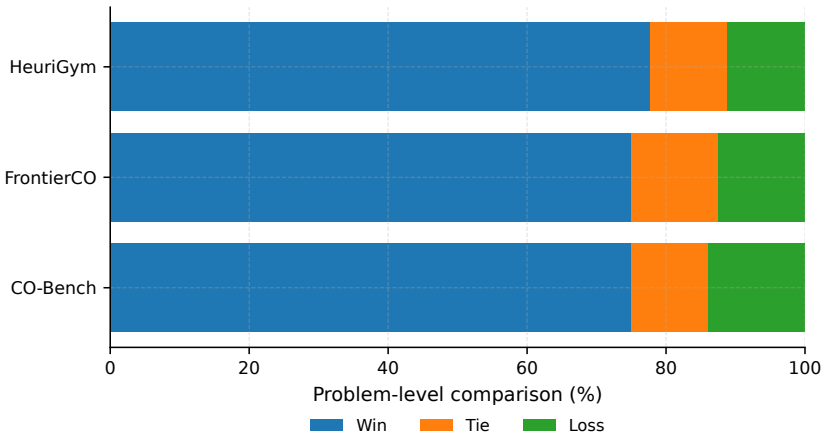


Figure 7: Problem-level wins, ties, and losses against the strongest non-ours baseline. Improvements are broad rather than driven by a small subset of tasks.

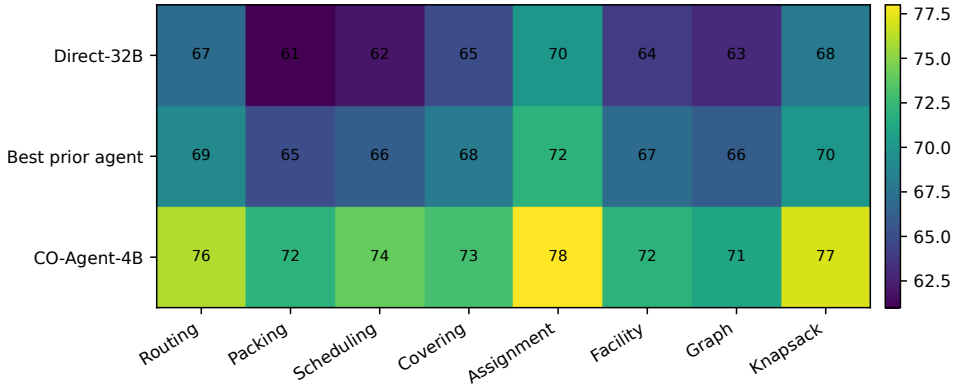


Figure 8: Family-wise normalized performance on representative CO structures. CO-Agent improves consistently across routing, packing, scheduling, covering, assignment, facility-location, graph, and knapsack families.

Effect across model scales. Figure 10 compares base, SFT, and CO-VGOPD students from 0.6B to 8B. CO-VGOPD improves every size, but the absolute and relative gains are largest for the smallest students. This is consistent with our intended use case: verifier-grounded repair is particularly valuable when the compact generator lacks sufficient intrinsic capacity to reliably produce executable and feasible optimization code in a single pass.

Verified Repair Advantage. Finally, Figure 11 bins correction pairs by VRA and measures their realized downstream improvement. The monotone trend indicates that VRA is not merely a training heuristic: it is aligned with the actual value of a teacher correction after execution. Weighting corrections by VRA therefore concentrates learning on repairs that materially improve the solver rather than on superficial textual differences.

What allocation policy is learned? Figure 12 groups instances by difficulty. On easy problems, almost half of the normalized budget is used for the initial model/token allocation and the planner frequently terminates after a short trajectory. Hard problems exhibit a qualitatively different allocation: the repair share grows from 8% to 28%, solver computation increases, average trajectory length grows from 1.8 to 5.1 turns, and early stopping falls from 68% to 15%. CARP therefore does not obtain its efficiency simply by always using less computation; it reallocates computation toward the operations that are useful for each difficulty regime.

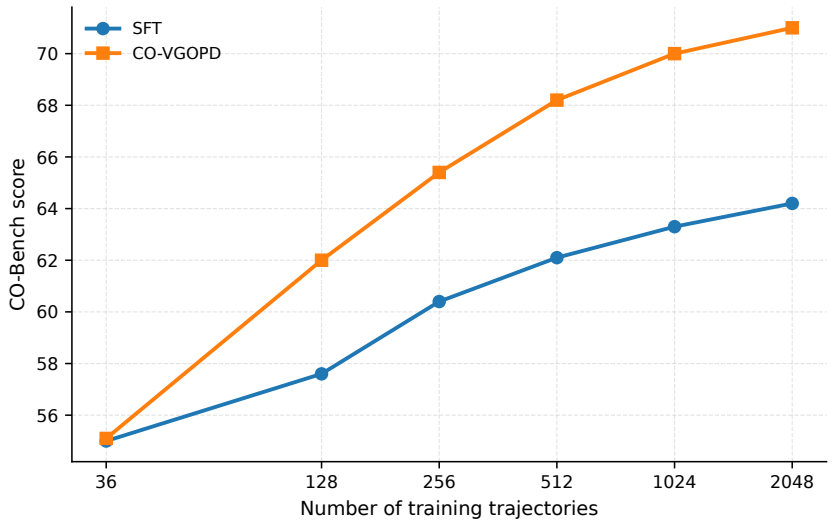


Figure 9: Training-data scaling. Verifier-grounded corrective trajectories provide substantially higher sample efficiency than static SFT demonstrations.

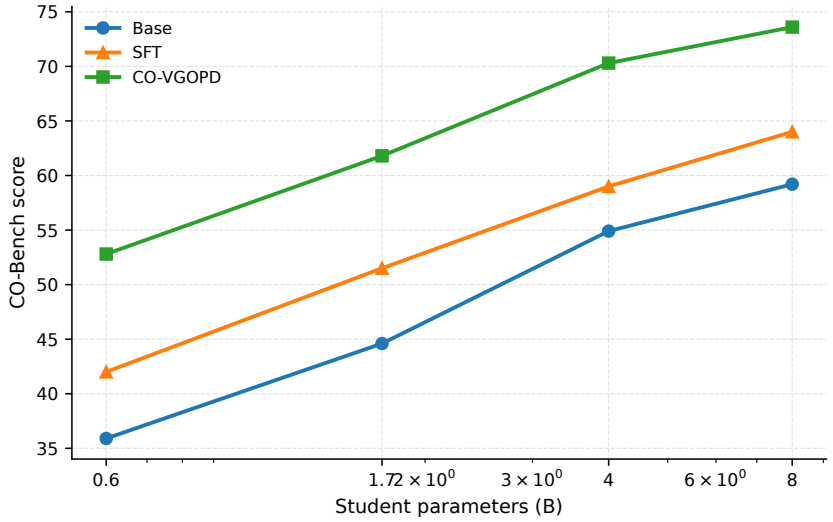


Figure 10: Student scaling. CO-VGOPD consistently improves compact generators, with particularly large gains in the smallest models.

Generalization to unseen budgets. A practical budget-aware planner should operate at budgets not explicitly used during training. Figure 13 evaluates several intermediate and extrapolated budgets. CARP preserves a clear advantage at unseen budget values, including both tight and generous regimes. This suggests that the planner learns a conditional resource-allocation policy rather than memorizing a small set of fixed configurations.

Evolution efficiency. Figure 14 plots the best verified skill fitness against the number of candidate evaluations. AUSE exceeds the final HiFo-Prompt fitness after roughly 150 evaluations, whereas HiFo-Prompt requires the full 400-evaluation budget to reach its final level. This behavior supports the submodular parent objective: selecting complementary high-fitness parents reduces redundant evolutionary trials and accelerates discovery of useful algorithmic combinations.

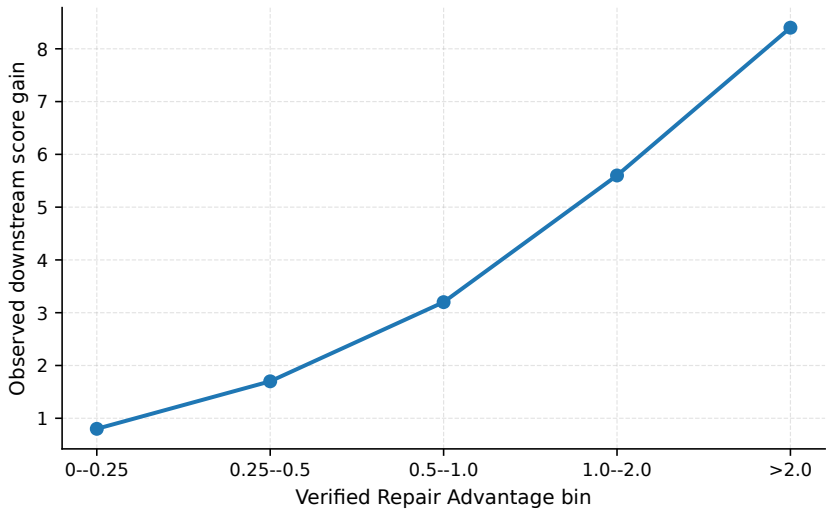


Figure 11: Calibration of Verified Repair Advantage (VRA). Larger VRA values correspond to larger realized downstream improvements.

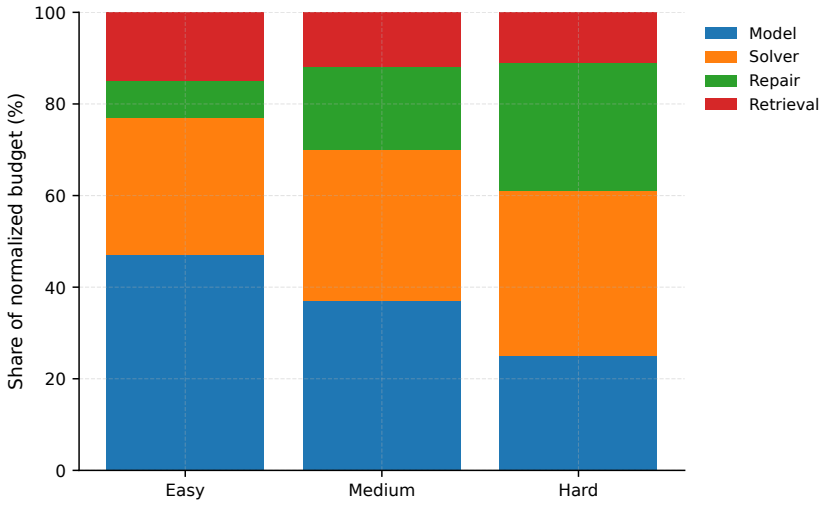


Figure 12: Learned allocation behavior. CARP terminates easy problems early while shifting more computation toward solver execution and repair on difficult problems.

Continual repository growth. We next introduce new problem families sequentially and evaluate the average performance over all families observed so far. Figure 15 shows that naive append initially improves but eventually regresses as repository interference accumulates. Fitness-only evolution is more stable but saturates. AUSE continues to improve throughout the sequence because each new skill must demonstrate positive marginal utility before admission.

Figure 16 confirms the same effect from the perspective of previous-task retention. At the final stage, naive repository growth retains only 80% of its earlier-task performance, compared with 96% for AUSE. Together with the RMU scatter, these results show that continual skill evolution requires not only discovering locally better heuristics but also controlling the system-level externalities created by an expanding skill repository.

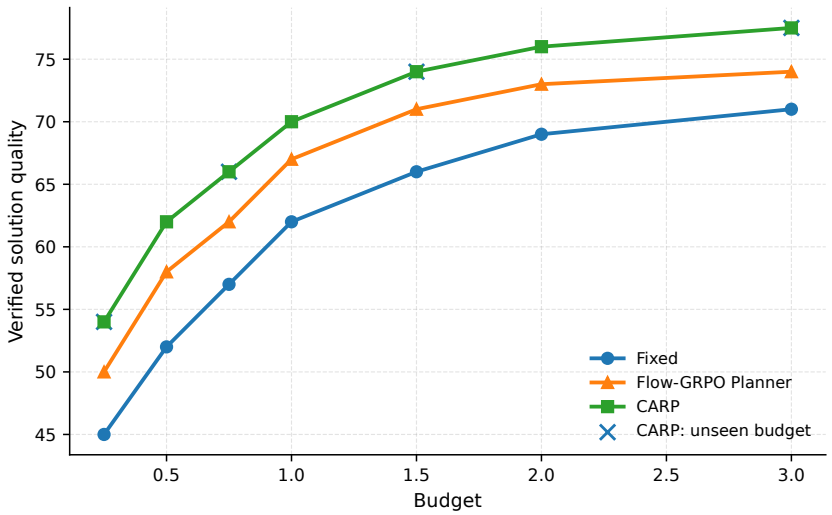


Figure 13: Budget generalization. CARP preserves its advantage on both training budgets and unseen inference budgets.

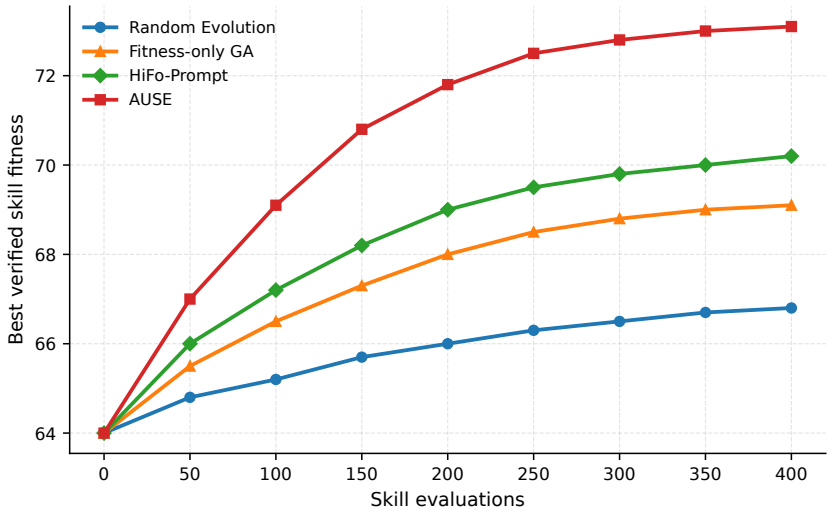


Figure 14: Evolution efficiency under a matched evaluation budget. AUSE reaches stronger skills with substantially fewer evaluations.

Algorithm 3 Agent-Utility-Guided Skill Evolution for CO (AUSE)

Require: Target description d; repository $\mathcal{K}$; planner P; generator G; executor E; verifier V; meta-operator Γ; local neighborhood $\mathcal{N}(d)$; admission neighborhood $\mathcal{N}_{\mathrm{adm}}(d)$; population size N_{pop}; parent budget K_{par}; maximum rounds R_{evo}; threshold ϵ_{evo}; patience κ_{pat}

Ensure: Evolved skill $s^\star$ and updated repository $\mathcal{K}^+$

Extract structural attributes $\Phi(d) = \{\chi_1, \ldots, \chi_{H_{\mathrm{attr}}}\}$

Retrieve N_{pop} relevant elite skills from $\mathcal{K}$ to initialize $\mathcal{S}^{(0)}$

Evaluate each $s \in \mathcal{S}^{(0)}$ on $\mathcal{N}(d)$ and compute $J(s \mid d)$

$n_{\mathrm{stall}} \leftarrow 0$

for $r = 0, \ldots, R_{\mathrm{evo}} - 1$ **do**

  Select parent set

$$\mathcal{R}^{(r)} \leftarrow \mathrm{GreedySubmodularSelect}\left(\mathcal{S}^{(r)}, F_d, K_{\mathrm{par}}\right)$$

  Initialize offspring set $\mathcal{O}^{(r)} \leftarrow \emptyset$

  while $|\mathcal{O}^{(r)}| < N_{\mathrm{pop}}$ **do**

    Sample parent skill(s) from $\mathcal{R}^{(r)}$ and operator $\gamma \in \{\Gamma_{\mathrm{comp}}, \Gamma_{\mathrm{cross}}, \Gamma_{\mathrm{mut}}\}$

    Generate offspring $\hat{s} \leftarrow \gamma(d, \mathcal{R}^{(r)})$

    Evaluate $\hat{s}$ on $\mathcal{N}(d)$ using the complete CO-Agent loop

    if $\hat{s}$ produces execution or feasibility failures **then**

      Repair $\hat{s} \leftarrow \Gamma_{\mathrm{repair}}(d, \hat{s}, \mathcal{Y}_{\hat{s}})$ and re-evaluate it

    end if

    if $\hat{s}$ passes validity checks **then**

      Compute $J(\hat{s} \mid d)$ and add $\hat{s}$ to $\mathcal{O}^{(r)}$

    end if

  end while

  $\mathcal{A}^{(r)} \leftarrow \mathcal{S}^{(r)} \cup \mathcal{O}^{(r)}$

  Preserve elite skill

$$s^{(r)}_{\mathrm{elite}} \leftarrow \arg\max_{s \in \mathcal{A}^{(r)}} J(s \mid d)$$

  Construct next population

$$\mathcal{S}^{(r+1)} \leftarrow \{s^{(r)}_{\mathrm{elite}}\} \cup \mathrm{GreedySubmodularSelect}\left(\mathcal{A}^{(r)} \setminus \{s^{(r)}_{\mathrm{elite}}\}, F_d, N_{\mathrm{pop}} - 1\right)$$

  $J^{(r)}_{\mathrm{best}} \leftarrow \max_{s \in \mathcal{S}^{(r)}} J(s \mid d), \quad J^{(r+1)}_{\mathrm{best}} \leftarrow \max_{s \in \mathcal{S}^{(r+1)}} J(s \mid d)$

  if $J^{(r+1)}_{\mathrm{best}} - J^{(r)}_{\mathrm{best}} < \epsilon_{\mathrm{evo}}$ **then**

    $n_{\mathrm{stall}} \leftarrow n_{\mathrm{stall}} + 1$

  else

    $n_{\mathrm{stall}} \leftarrow 0$

  end if

  if $n_{\mathrm{stall}} \geq \kappa_{\mathrm{pat}}$ **then**

    break

  end if

end for

$s^\star \leftarrow \arg\max_{s \in \mathcal{S}^{(R')}} J(s \mid d)$, where R' is the final completed round

Evaluate $s^\star$ under natural retrieval on $\mathcal{N}_{\mathrm{adm}}(d)$ using $\mathcal{K}$ and $\mathcal{K} \cup \{s^\star\}$

Compute $\mathrm{RMU}(s^\star \mid \mathcal{K})$ and $\mathrm{NTR}(s^\star \mid \mathcal{K})$

if Eq. (36) holds **then**

  $\mathcal{K}^+ \leftarrow \mathcal{K} \cup \{s^\star\}$

else

  $\mathcal{K}^+ \leftarrow \mathcal{K}$

end if

return $s^\star, \mathcal{K}^+$

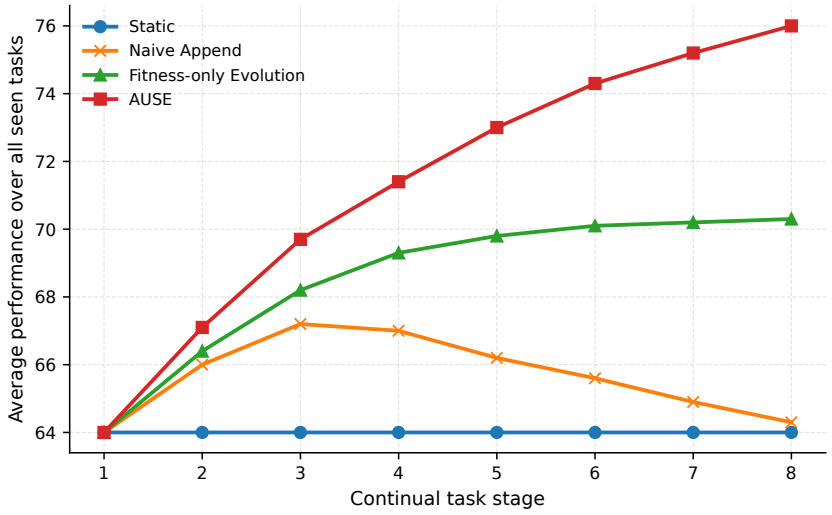


Figure 15: Continual evolution as new problem families arrive. Naive repository growth eventually suffers interference, whereas AUSE continues to improve agent-level performance.

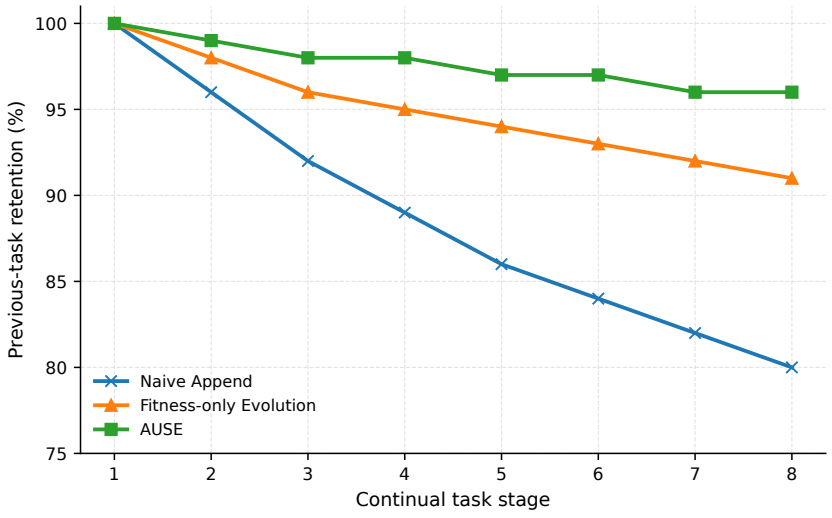


Figure 16: Retention on previously seen tasks. Repository-level admission substantially reduces regression as the skill library grows.