EVOHGS: CO-EVOLVING LANGUAGE MODELS WITH HYBRID GENETIC SEARCH FOR VEHICLE ROUTING PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Vehicle Routing Problems (VRPs) are a fundamental class of combinatorial optimization problems aimed to find high-quality routes for vehicles under operational constraints. Despite decades of study in operations research, VRPs remain computationally challenging due to their NP-hardness, with additional difficulty introduced by the diverse constraint structures that arise across different variants. Hybrid Genetic Search (HGS) has emerged as a state-of-the-art metaheuristic framework for many VRP variants. However, its performance still depends heavily on manually designed search operators and empirical parameter tuning, limiting its ability to adapt automatically to complex and diverse problem settings. To address the limitation, we propose **EvoHGS**, a large language model (LLM)-guided evolutionary framework that automatically improves HGS across the VRP family. EvoHGS integrates LLM-based heuristic generation with evolutionary search, allowing candidate HGS operators to be generated, evaluated, selected, and refined through an iterative optimization process. To further align generation with downstream search performance, EvoHGS uses feedback collected during evolution to finetune the LLM via reinforcement learning. This enables the model to progressively produce operators that are better adapted to the search dynamics of HGS across VRP variants. Experiments on multiple VRP variants show that EvoHGS consistently outperforms the original HGS baseline at the scale used for evolution, remains competitive when the evolved operators are applied to larger instances, and demonstrates promising zero-shot generalization to unseen VRP variants.

1 INTRODUCTION

Vehicle Routing Problems (VRPs) represent a fundamental class of combinatorial optimization problems with broad applications in logistics, transportation, and supply chain management (Toth & Vigo, 2014). Because of their NP-hard nature, large-scale VRP instances (e.g., those involving hundreds of nodes) are commonly addressed by metaheuristics rather than exact optimization. Among metaheuristic methods, Hybrid Genetic Search (HGS) has emerged as one of the most effective, combining population-based exploration with local improvement and adaptive control mechanisms (Vidal, 2022). As a result, HGS has achieved state-of-the-art (SOTA) performance on numerous benchmark instances (Vidal, 2022; Vidal et al., 2012). Despite its strong empirical performance, the implementation of search mechanisms in HGS still depends heavily on human expertise and extensive empirical tuning. This limitation becomes more pronounced across the diverse family of VRP variants, where differing constraint structures often hinder the transferability of mechanisms that perform well in one variant to others. Thus, there remains a critical need to enhance the adaptability and automation of HGS search mechanisms, with the goal of improving solution quality and reducing the burden of manual design (Wouda & Lan, 2024).

Recent progress in large language models (LLMs) offers a promising pathway toward this goal (Yang et al., 2023; Huang et al., 2025; Jiang et al., 2025b;a; Zhang et al., 2025). LLMs can generate structured programs, propose algorithmic modifications, and participate in search loops for automatic heuristic design (Romera-Paredes et al., 2024; Novikov et al., 2025). Existing methods have explored LLMs as algorithm generators under an evolutionary inference-time reasoning paradigm, where candidate programs (output of the LLM) are sampled, evolved, evaluated, and selected based on

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.

performance (Liu et al., 2024b; Ye et al., 2024; Zheng et al., 2025; Shi et al.; Xu et al., 2025). More recent methods further incorporate accumulated feedback or reinforcement learning to improve the generation over time (Šurina et al., 2025; Liu et al., 2025; Zhu et al., 2026; Huang et al., 2026). This body of research shows that LLMs can actively drive algorithm design.

However, improving HGS requires a paradigm that differs from general LLM-based automatic heuristic design. Existing LLM-based methods often generate standalone heuristics or programs, whose quality can be assessed independently after execution (Romera-Paredes et al., 2024; Liu et al., 2026; Ye et al., 2024). In contrast, HGS is an intricate pipeline in which components such as recombination, local improvement, and selection are tightly coupled (Vidal, 2022; Vidal et al., 2012). Modifying just one internal component is not an isolated event. It creates a ripple effect that drastically alters the global search trajectory and population dynamics. Consequently, the true value of an LLM-generated modification cannot be assessed in a stand-alone manner. It must be integrated into the solver and evaluated through a complete search process. Beyond the complexity of evaluation, modifying HGS imposes strict implementation constraints. Rather than generating a loose script, the LLM must search for modifications that perfectly match the solver's interface. These modifications must compile cleanly within a hybrid C++/Python framework (Wouda & Lan, 2024) and preserve algorithmic invariants. Given the complex interplay of internal components and strict implementation requirements, traditional prompt-only methods (even with inference-time reasoning) struggle to produce valid, executable improvements (Jiang et al., 2024; Gehring et al., 2025). While large language models possess broad programming capabilities, their general prior knowledge alone is insufficient to reliably modify such a specialized framework. Consequently, relying solely on prompts (Ye et al., 2024) inevitably results in code that fails to compile or disrupts search population dynamics. This failure stems from a fundamental mismatch in distribution between the broad space of generalized code generated by LLMs and the highly constrained distribution of valid solver modifications. Overcoming this limitation requires explicitly aligning the model's generation distribution with the target framework. Moreover, optimizing an operator for a single VRP variant does not ensure its benefits to other variants (Šurina et al., 2025; Zhu et al., 2026). An effective modification under one combination of constraints may offer limited gains or even degrade performance when constraints change. This motivates a further requirement for improving a unified solver such as HGS (Vidal et al., 2014): operator design should account for diverse constraints, and the resulting improvements should be evaluated beyond the variants used during optimization.

To address the challenges, we propose **EvoHGS**, a training-driven LLM framework that co-evolves an LLM policy and a pool of executable HGS operator modifications. EvoHGS leverages Supervised Fine-Tuning (SFT) alongside Reinforcement Learning (RL). The SFT phase is crucial for adapting the LLM to output syntax-correct code modifications that seamlessly integrate into the HGS framework. Furthermore, by exposing the model to SFT data spanning six VRP variants, the LLM learns cross-variant constraint representations. As a result, EvoHGS acts as a general-purpose solver enhancer for HGS, rather than a narrow tool specialized for a single problem class. RL uses the results from the solver as feedback to improve the LLM, this allows the LLM generate modifications that make HGS performance better and better. During co-evolution, the LLM-based policy generates ideas for code modifications and their implementations. By evaluating candidates (both ideas and implementations) in terms of compilability, constraint feasibility, and objective performance, feedback is provided for further policy updates.

Our main contributions are summarized as follows:

(1) We target the automatic improvement of the HGS solver, currently the most advanced solver for VRP. The evolved algorithmic components generated by our method outperform the original HGS across six distinct VRP variants, advancing the state-of-the-art in VRP solvers.

(2) We propose **EvoHGS**, a training-driven framework where the LLM drives evolutionary search over HGS component modifications: SFT on modifications spanning multiple VRP variants enables the LLM to generate compilable code and confers generalization across the VRP family, while RL within the evolutionary loop co-evolves the LLM with the pool of modifications to further improve performance.

(3) We evaluate **EvoHGS** across multiple VRP variants and manifest consistent improvements over the original HGS baseline and a wide range of state-of-the-art baselines. The method generalizes effectively across problem scales and, more importantly, across VRP variants.

2 Related Work

LLM-based Automatic Heuristic Design LLMs have recently emerged as powerful tools for automatic heuristic design (Romera-Paredes et al., 2024; Novikov et al., 2025; Lange et al., 2025; Liu et al., 2026; Wu et al., 2025). In the field of combinatorial optimization, EoH (Liu et al., 2024b), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025) and MCTS-AHD (Zheng et al., 2025) leverage LLMs to iteratively generate and refine heuristics based on evaluation feedback. These methods iteratively generate and evaluate candidate heuristics, but keep the search strategy and LLM itself fixed throughout search. To overcome this limitation, more recent work incorporates RL to update the LLM using search feedback (Liu et al., 2025; Zhu et al., 2026). EvoTune (Šurina et al., 2025) and CALM (Huang et al., 2026) follow this direction by combining evolutionary search with RL-based fine-tuning. These methods co-evolve algorithmic populations with LLM parameters, using accumulated feedback to steer generation toward higher-performing regions. Unlike existing LLM-based heuristic designs for VRPs, we directly enhance components of state-of-the-art HGS solver rather than generate new heuristics or standalone methods.

Conventional Solvers and Hybrid Genetic Search. VRPs are commonly tackled by metaheuristic algorithms such as Lin–Kernighan–Helsgaun heuristic (LKH3) (Helsgaun, 2017), Adaptive Large Neighborhood Search (ALNS) (Shaw, 1998; Ropke & Pisinger, 2006) and Hybrid Genetic Search (HGS) (Vidal et al., 2012; Vidal, 2022). Among these, HGS is particularly notable for combining population-based exploration with efficient local search neighborhoods and advanced diversity control across feasible and infeasible subpopulations. Its generality across a broad family of VRP variants was established through a unified solution framework (Vidal et al., 2014), some modern open-source implementations such as PyVRP (Wouda & Lan, 2024) have been developed based on its scalability and modularity. Despite its effectiveness, HGS depends on carefully designed internal components, which are typically tuned through domain expertise. Recent work has begun using LLMs to automate this design process (Kikuta et al., 2024; Li et al., 2025; Tran et al., 2025; Hottung et al., 2025; Zhu et al., 2026; Malik et al., 2026). RFTHGS (Zhu et al., 2026) uses reinforcement learning (RL) to adapt large language models (LLMs) for improving HGS components, while PyVRP+ (Malik et al., 2026) applies metacognitive prompting to evolve multiple HGS components through structured cycles. However, RFTHGS trains the LLM with RL before search and targets CVRP only, while PyVRP+ only prompts a fixed LLM. In contrast, EvoHGS first applies SFT, which injects constraint knowledge shared across VRP variants and adapts the LLM to the solver interface to reduce invalid code; RL inside the evolutionary loop then further improves performance.

Neural Solvers for VRPs. Neural solvers either construct solutions autoregressively (Kool et al., 2018; Kwon et al., 2020; Zhou et al., 2024; Berto et al., 2025) or iteratively refine them with learned operators (d O Costa et al., 2020; Ma et al., 2023; Hottung & Tierney, 2022) (§ A). In contrast, our approach does not learn a solver from scratch; instead, it improves an existing high-performance solver, HGS, by modifying its internal behavior via LLM while preserving its overall structure.

3 Problem Statement

Hybrid Genetic Search (HGS) represents the state-of-the-art for solving typical VRPs (Santini et al., 2023). In this paper, we introduce a new avenue for leveraging large language models (LLMs) to further improve performance. A VRP instance requires constructing a set of routes that serve all customers while satisfying problem-specific constraints and minimizing the total cost. In this work, we consider a family of VRPs characterized by common practical constraints, including capacity limits, open routes, backhaul requirements, duration constraints, and time windows. [1] Let $\mathcal{T}$ denote the set of VRPs with input space $\mathcal{X}$ and solution space $\mathcal{S}$. Instead of optimizing the full HGS framework, we restrict our attention to the crossover operators, which are central to any genetic search framework, while keeping all other components fixed. Let $\mathcal{H}$ denote the space of feasible crossover operators, i.e., executable implementations that satisfy structural and interface constraints and can be successfully integrated into HGS. A crossover operator is then defined as an implementation $c \in \mathcal{H}$. Given a fixed set of VRP instances $\mathcal{D} \subset \mathcal{X}$ and an instance-dependent objective function $f_x : \mathcal{S} \to \mathbb{R}$, where lower values indicate better solutions, let $\mathfrak{S}(c, x)$ denote the solution returned by

[1] Definitions of the VRP variants used in this paper are given in § C.1.

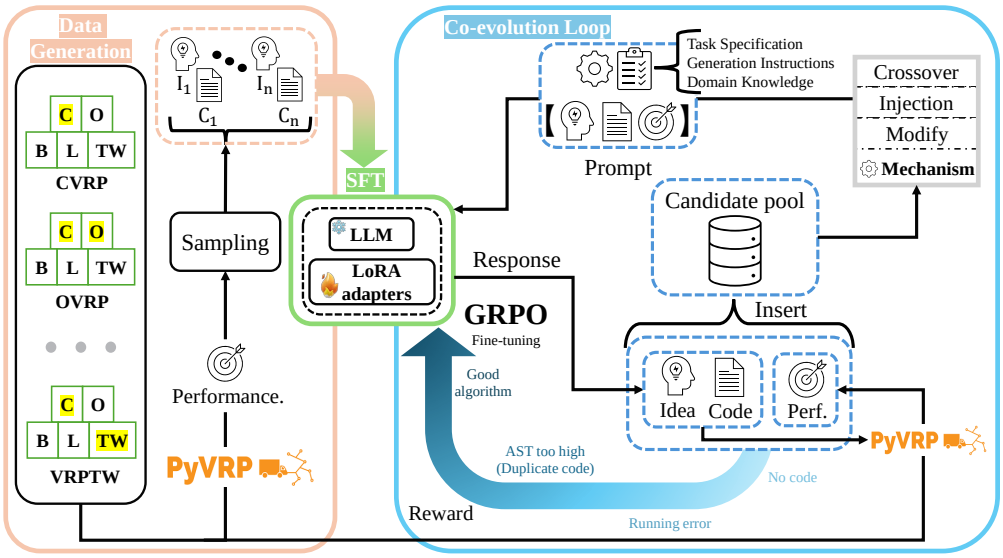

Figure 1: **The framework of EvoHGS**. During data generation, we (1) generate VRP training instances following the MVMoE (Zhou et al., 2024) setup; (2) derive HGS-compatible operator implementations through ReEvo (Ye et al., 2024); and (3) extract conceptual modification ideas through LLM-based code analysis. See § B.1 for comprehensive details.

HGS with crossover operator c on instance x. We define the performance of c as:

$$g(c) = -\frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} f_x(\mathfrak{S}(c, x)) \,. \tag{1}$$

Larger values of $g(c)$ correspond to better performance, considering the common cost minimization setting in VRPs. The goal of **EvoHGS** is to identify the optimal crossover operator within the feasible space $\mathcal{H}$, $c^* = \arg\max_{c \in \mathcal{H}} g(c)$. In particular, let $c^{(0)} \in \mathcal{H}$ denote the original HGS crossover operator. EvoHGS co-evolves HGS-compatible crossover operators and the LLM policy that generates them, aiming to find operators $c \in \mathcal{H}$ such that $g(c) > g(c^{(0)})$.

4 Methodology

As illustrated in Figure 1, EvoHGS automates the design of crossover operators across multiple VRP variants through a closed-loop evolutionary search, integrated with LLM training for generating effective HGS-compatible code modifications. Specifically, we first initialize the LLM via supervised fine-tuning (SFT) (§ 4.1) to establish a strong structural prior; this enables the model to generate conceptually coherent ideas and translate them into HGS-compatible functional code that satisfies all syntactic and interface constraints. During the search phase (§ 4.2.1), the LLM iteratively synthesizes new candidate implementations of crossover operators in an evolutionary manner, guided by top-performing preceding implementations. These candidates are integrated into HGS through incremental compilation and evaluated on the fixed VRP instance set $\mathcal{D}$. Crucially, we also introduce synchronous reinforcement learning (RL) updates (§ 4.2.2), allowing the LLM to co-evolve with the operators. The co-evolving process enables the LLM to continuously refine its generation, leading to more effective implementations based on experience with previously designed crossover operators.

4.1 Supervised Fine-Tuning (SFT)

Redesigning and optimizing HGS workflows requires training on a specific VRP dataset to inject domain knowledge into the LLM, motivating us to initialize it with an SFT stage. Unlike many language reasoning tasks that rely on costly human annotations, our SFT process is built on an automated data pipeline driven by evolutionary search. Each SFT example pairs a prompt with a

target output consisting of a modification idea and its HGS-compatible crossover implementation evolved with ReEvo (Ye et al., 2024) (see Figure 1 and Appendix B.1 for details).

With this dataset, we fine-tune the LLM to generate valid crossover implementations $c \in \mathcal{H}$. Specifically, we use an instruction prompt that directs the model to act as an expert algorithm designer, translating ideas for modifications into executable implementations. This input prompt is conditioned on a precise problem specification. To support generalization across different problems, we consider a family of VRP variants, $\mathcal{T} = \{\text{CVRP}, \text{OVRP}, \text{OVRPTW}, \text{VRPB}, \text{VRPL}, \text{VRPTW}\}$, and treat each task $\tau \in \mathcal{T}$ as the context. The prompting can be defined by $q = \phi(\tau)$. Here, $\phi(\cdot)$ maps the task to a comprehensive textual prompt that outlines the specific constraints of τ, incorporates shared codebase knowledge (such as the strict interface definitions and essential data structures required for seamless integration), and provides the original HGS crossover operator as a baseline (see § B.2 for details).

The LLM first generates a textual description of the intended modification r^{idea}, followed by the corresponding code implementation r^{code}. We denote the target response as a sequence $y = (y_1, \ldots, y_n)$. The SFT objective is

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{(q,y)\in\mathcal{D}_{\text{SFT}}} \sum_{i\in\mathcal{J}_{\text{sup}}} \log P_\theta(y_i \mid q, y_{<i}), \tag{2}$$

where $\mathcal{J}_{\text{sup}}$ represent the token indices for r^{idea} and r^{code}, while $\mathcal{D}_{\text{SFT}}$ represents the multi-task SFT dataset. Instead of merely memorizing token sequences, the SFT objective forces the LLM to align VRP specifics (e.g., constraints) with the code's structural logic, thereby internalizing the strict structural and interface requirements of HGS. As our training data spans various problem scales and routing constraints, the model captures transferable algorithmic patterns across different VRP variants. To make this process computationally practical, we apply LoRA (Hu et al., 2022) that reduces memory and optimization cost without degrading LLM capacity. Ultimately, the SFT stage equips LLM to reliably produce syntactically valid and HGS-compatible implementations, establishing a stable initialization for subsequent co-evolution loop (§ 4.2).

4.2 Co-evolution Loop

Building upon the SFT-initialized LLM, EvoHGS introduces a closed loop where **Evolutionary Search** and **Reinforcement Learning** alternately operate to **co-evolve** the LLM policy alongside a pool of candidate operators. The evolutionary search generates new candidates that are executed to provide performance feedback, using which the RL module directly optimizes the LLM. In return, the improved LLM generates higher-quality operators that are returned to the candidate pool (an evolving collection of the best-performing crossover operators found so far).

4.2.1 Evolutionary Operator Generation

EvoHGS evolves HGS's crossover operators through a set of **evolutionary mechanisms** Ω: **E1** (crossover), **M1** (injection) and **M2** (modification). At each iteration, EvoHGS first samples an evolutionary mechanism from Ω and selects the required parent crossover operator(s) from the current candidate pool. The LLM then generates a modification idea conditioned on the selected evolutionary mechanisms and the parent candidate operator(s), and converts it into executable code. The **M2** evolutionary mechanisms also serve to initialize the search by mutating the original HGS operator.

Each mechanism corresponds to a distinct type of change. **E1** combines key modifications from two parent candidates into a single coherent offspring. **M1** preserves the core structure of a parent while introducing a fresh implementation for variety. This prevents the model from repeating the various ideas explored in recent iterations. **M2** modifies a single part of the parent candidate while keeping its overall structure unchanged. At iteration t, EvoHGS maintains a pool of feasible crossover operators

$$\mathcal{P}_t = \{c_t^{(1)}, \ldots, c_t^{(n_t)}\} \subseteq \mathcal{H}, \qquad n_t \le L_p.$$

Each crossover operator $c \in \mathcal{P}_t$ is associated with a natural-language modification description *idea*(c), its implementation *code*(c), and its evaluated performance $g(c)$ on $\mathcal{D}$. The pool is ordered by $g(\cdot)$ in descending order. At each iteration, an evolutionary mechanism m_t is chosen uniformly at random among **E1**, **M1** and **M2**. Specifically, during initialization, EvoHGS exclusively applies **M2** to promote exploration and rapidly construct the initial candidate pool.

Selection and filtering. Parent selection depends on performance. Let $c_t^{(k)}$ denote the k-th ranked candidate in $\mathcal{P}_t$. For single-parent mechanisms (**M1**, **M2**), a parent is sampled by probability:

$$\Pr\left(c_t^{(k)}\right) = \frac{k^{-1}}{\sum_{j=1}^{n_t} j^{-1}}, \qquad k = 1, \ldots, n_t. \tag{3}$$

For the two-parent mechanism **E1**, the first parent (c_1) is selected based on performance, exactly like the single-parent mechanisms. For the second parent, EvoHGS balances exploitation and exploration: it either uses the same performance-based rule or, with probability p_{div}, actively seeks a structurally different partner. This diversity-driven selection favors candidates that are dissimilar to c_1: $\Pr_{\text{div}}(c \mid c_1) \propto 1 - s_{\text{AST}}(c, c_1)$, where $s_{\text{AST}}(\cdot, \cdot)$ measures the structural similarity between two operators based on their Abstract Syntax Trees (ASTs), with details provided in § C.3. Prior to evaluation, EvoHGS filters out candidate operators that fail to compile or violate interface constraints. It then applies AST-based deduplication to eliminate structural clones.

Pool Update. After evaluation, crossover operators are deduplicated via AST to retain only the top-performing instance of each unique logic. To prevent search stagnation, the pool is collapsed and rebuilt once no new global best has been found for several rounds.[2] As Figure 3 shows, such resets restore exploration during stagnation without compromising the best operator found so far.

4.2.2 Reinforcement Learning

The evolutionary search (§ 4.2.1) identifies candidate operators, yet the resulting successful cases remain external samples in the candidate pool. To internalize these discoveries and update the model's underlying generation logic, EvoHGS employs a Reinforcement Learning (RL) phase. This involves a co-evolutionary cycle: the evolutionary mechanisms **evolve improved offspring** from the current pool, while the RL module exploits the discoveries to update the model's generation policy, thereby enabling the LLM to produce better code for the next iteration.

At each iteration t, the LLM samples a batch of I candidate operators $\{o_i\}_{i=1}^{I}$, which are instantiated and evaluated on instances $x \in \mathcal{D}$. Each candidate o_i is assigned a reward $r(o_i)$ based on its feasibility and performance. The LLM's policy is then updated using Group Relative Policy Optimization (GRPO) (Shao et al., 2024), shifting generation policy toward higher-reward operators.

Constraint Mapping and Penalties. Invalid candidates receive a discrete penalty $\psi(o) < 0$ according to the type and severity of the failure, including **format**, **compilation** and **runtime** failures, while an **anti-plagiarism** penalty applies to candidates whose AST distance from the original HGS operator falls below a threshold η (deatailed values and threshold are provided in § C.3).

Reward Function. Candidates that pass all validity checks receive a continuous performance reward based on their improvement over the baseline operator $c^{(0)}$:

$$\Delta_{\text{base}}(c_{\text{new}}, c^{(0)}) = g(c_{\text{new}}) - g(c^{(0)}), \tag{4}$$

where $g(\cdot)$ denotes the objective performance defined in Eq. (1). The final reward $r(o)$ is defined as:

$$r(o) = \begin{cases} \psi(o), & \text{if } o \in \mathcal{I}, \\ -\alpha + \beta\, \Delta_{\text{base}}, & \text{if } \Delta_{\text{base}} \le 0, \\ 1 + \Delta_{\text{base}}, & \text{if } \Delta_{\text{base}} > 0, \end{cases} \tag{5}$$

where α and β are hyperparameters controlling the reward scaling, and $\mathcal{I}$ represents the set of penalized (invalid or plagiarized) candidates. By strictly anchoring rewards to relative performance, candidates that fall short of the baseline remain penalized, whereas successful modifications receive positive rewards proportional to their performance gains. We apply group-level advantage normalization (Shao et al., 2024) to stabilize policy updates across reward scales (§ C.3).

5 Experiments

In this section, we evaluate the generalization capabilities of EvoHGS across two critical dimensions: cross-scale generalization to larger problem instances (§ 5.1) and zero-shot transfer to unseen routing

[2] More details can be found in § C.3.

Table 1: **Performance on six VRP variants with 100-, 500-, and 1000-node instances.** All models are trained or evolved only on instances with $n = 100$. Gaps (%) are computed relative to HGS-PyVRP. Bold indicates the lowest gap per column and negative gaps are underlined.

Method	CVRP						OVRP						VRPB					
	$n = 100$		$n = 500$		$n = 1000$		$n = 100$		$n = 500$		$n = 1000$		$n = 100$		$n = 500$		$n = 1000$	
	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓
HGS-PyVRP	15.785	*	37.743	*	51.313	*	10.004	*	23.212	*	32.895	*	14.314	*	33.833	*	47.632	*
OR-Tools#	16.265	3.04	40.994	8.61	55.904	8.95	10.189	1.85	25.817	11.22	37.561	14.18	14.682	2.57	37.301	10.25	53.120	11.52
POMO	15.975	1.20	41.286	9.39	63.182	23.13	10.413	4.09	27.300	17.61	47.595	44.69	16.686	16.57	45.822	35.44	73.554	54.42
MVMoE/4E	15.946	1.02	40.519	7.36	88.649	72.76	10.384	3.80	31.428	35.40	83.853	154.91	16.722	16.82	49.539	46.42	110.564	132.12
MVMoE/4E-L	15.972	1.18	44.190	17.08	71.863	40.05	10.402	3.98	28.531	22.91	49.762	51.28	16.652	16.33	47.118	39.27	80.029	68.02
RF-POMO	16.051	1.69	40.874	8.30	60.302	17.52	10.429	4.25	26.524	14.27	41.953	27.54	14.940	4.37	40.543	19.83	62.190	30.56
RF-MoE-L	16.018	1.48	40.633	7.66	61.304	19.47	10.384	3.80	26.480	14.08	42.830	30.20	14.856	3.79	39.462	16.64	81.922	71.99
RF-transformer	16.008	1.41	42.263	11.98	62.051	20.93	10.384	3.80	28.303	21.93	44.462	35.16	14.831	3.61	41.741	23.37	62.781	31.80
ReEvo	15.781	-0.02	37.800	0.15	51.327	**0.03**	9.995	-0.10	23.246	0.15	32.916	0.07	14.304	-0.07	33.874	0.12	47.656	0.05
OpenEvolve	15.805	0.13	37.954	0.56	51.589	0.54	10.001	-0.03	23.248	0.15	32.913	**0.06**	14.315	0.01	33.890	0.17	47.702	0.15
EvoTune	15.822	0.24	38.006	0.70	51.573	0.51	10.014	0.10	23.299	0.37	33.004	0.33	14.329	0.10	33.933	0.30	47.753	0.25
EvoHGS	15.760	**-0.16**	37.763	**0.05**	51.346	0.06	9.992	**-0.13**	23.212	**0.00**	32.928	0.10	14.299	**-0.11**	33.814	**-0.06**	47.517	**-0.24**

Method	VRPTW						VRPL						OVRPTW					
	$n = 100$		$n = 500$		$n = 1000$		$n = 100$		$n = 500$		$n = 1000$		$n = 100$		$n = 500$		$n = 1000$	
	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓	Obj	Gap↓
HGS-PyVRP	25.178	*	94.474	*	182.073	*	15.695	*	37.045	*	50.445	*	14.092	*	48.522	*	84.314	*
OR-Tools#	26.181	3.98	100.598	6.48	191.920	5.41	16.128	2.76	39.924	7.77	54.805	8.64	14.409	2.25	51.615	6.37	88.959	5.51
POMO	26.370	4.73	116.823	23.66	243.990	34.01	15.857	1.03	40.437	9.16	62.275	23.45	14.983	6.32	66.231	36.50	139.080	64.95
MVMoE/4E	26.220	4.14	118.077	24.98	263.258	44.59	15.814	0.76	46.354	25.13	94.024	86.39	14.897	5.71	71.671	47.71	177.306	110.29
MVMoE/4E-L	26.254	4.27	118.418	25.34	263.453	44.70	15.833	0.88	43.435	17.25	71.679	42.09	14.931	5.95	63.024	29.89	143.593	70.31
RF-POMO	31.072	23.41	137.045	45.06	282.031	54.90	15.912	1.38	40.062	8.14	59.173	17.30	17.392	23.42	67.871	39.88	126.630	50.19
RF-MoE-L	30.981	23.05	135.774	43.72	270.820	48.74	15.908	1.36	39.961	7.87	62.139	23.18	17.304	22.79	66.301	36.64	122.933	45.80
RF-transformer	32.942	30.84	145.432	53.94	311.695	71.19	15.889	1.24	41.417	11.80	61.072	21.07	18.040	28.02	71.476	47.31	142.664	69.21
ReEvo	25.150	-0.11	94.502	0.03	182.075	0.00	15.712	0.11	37.154	0.29	50.576	0.26	14.069	**-0.16**	48.571	0.10	84.341	0.03
OpenEvolve	25.262	0.34	95.044	0.60	182.965	0.49	15.720	0.16	37.216	0.46	50.701	0.51	14.114	0.15	49.034	1.06	85.078	0.91
EvoTune	25.201	0.09	94.733	0.27	182.389	0.17	15.704	0.06	37.092	0.13	50.482	**0.07**	14.096	0.02	48.713	0.40	84.617	0.36
EvoHGS	25.126	**-0.21**	94.433	**-0.04**	181.880	**-0.11**	15.672	**-0.14**	37.041	**-0.01**	50.487	0.08	14.071	-0.15	48.467	**-0.11**	84.076	**-0.28**

Note: * denotes the HGS-PyVRP reference objective used for gap computation. OR-Tools# uses a 300 s time limit. All LLM-based heuristic design methods are evaluated under the same fixed budget of 1000 queries per variant. LLM entries are means over evolution runs; mean ± SD and aggregation protocol are in § D.1.

constraints (§ 5.2). To this end, our main comparison benchmarks six representative VRP variants across sizes of 100, 500, and 1000 nodes (50 test instances per setting) to assess scale scalability. Furthermore, we evaluate structural generalization on ten additional unseen variants (detailed in § C.1). We use Qwen3-8B (Yang et al., 2025) as the backbone model for EvoHGS. SFT was conducted on a single 48-GB NVIDIA RTX A6000 GPU, and evaluations used two such GPUs. Aside from a one-time 8,h SFT overhead, an EvoHGS run requires an average of 6.8,h. This is highly competitive with ReEvo (7.5,h) and significantly faster than OpenEvolve (30.8,h) (see § C.5). Further details are provided in § C, and the source code will be made publicly available upon acceptance.

Baselines. To evaluate EvoHGS, we compare it against three categories of baselines: (1) **Traditional VRP solvers** such as OR-Tools (Perron & Furnon) and HGS-PyVRP (Wouda & Lan, 2024); (2) **Neural combinatorial optimization (NCO)** methods including POMO (Kwon et al., 2020), MVMoE (Zhou et al., 2024), and RouteFinder-based models (Berto et al., 2025); and (3) **LLM-based automated heuristic design (LLM-AHD)** approaches, including methods without LLM fine-tuning, such as ReEvo (Ye et al., 2024) and OpenEvolve (Sharma, 2025) and methods with LLM fine-tuning such as EvoTune (Šurina et al., 2025). Notably, AlphaEvolve (Novikov et al., 2025) has not released its official source code, we therefore include OpenEvolve (Sharma, 2025), an open-source community implementation inspired by AlphaEvolve. Among the HGS-specific methods, RFTHGS (Zhu et al., 2026) is designed for CVRP only; we compare against it on CVRPLIB in § F.2, where neither our reproduction nor its released operators achieve a statistically significant gain over HGS-PyVRP, even though its original training is computationally heavy, requiring eight GPUs. For solver-based baselines, HGS-PyVRP is run with a maximum of **1000 iterations**, while OR-Tools is evaluated with a **300 s time limit**. EvoHGS adopts the same HGS evaluation budget of **1000 iterations** for each candidate strategy generated. To ensure a fair comparison among LLM-based AHD methods, ReEvo and OpenEvolve are evaluated using GPT-5-nano (Singh et al., 2025), whereas EvoTune and EvoHGS use Qwen3-8B (Yang et al., 2025) under the same open-weight model setting; all LLM-based AHD methods are evaluated under a fixed budget of **1000 queries**.

Variant	Gap (%) ↓	Compilable (%) ↑
HGS-PyVRP	*	–
*LLM-based AHD: **GPT-5-nano***		
w/o SFT, w/ Evo	-0.12	46.0
*LLM-based AHD: **Qwen3-8B***		
w/o SFT, w/ Evo	-0.08	25.1
w/ SFT, w/ Evo	-0.13	85.3
Ours (SFT + Evo + GRPO)	**-0.17**	**92.6**

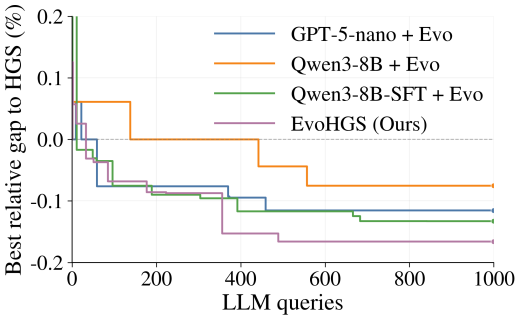

Figure 2: **Ablation study on CVRP. Left:** Best-run performance of each configuration under a fixed budget of 1000 LLM queries. **Right:** Evolution of the best gap across different LLM variants. **Compilable** denotes the percentage of generated candidates that can be successfully compiled and evaluated. * indicates the reference objective used for gap computation. Notation follows Table 1.

5.1 Performance on In-Distribution VRP Variants

Table 1 reports the main comparison on six VRP variants used during evolution, namely CVRP, OVRP, VRPB, VRPTW, VRPL, and OVRPTW. Our primary evaluation focuses on evolving the selective route exchange (SREX) crossover operator within HGS-PyVRP, a mature operator repeatedly refined by experts (Nagata & Kobayashi, 2010; Blocho & Nalepa, 2017) and hence a stringent test for EvoHGS (§ C.3). On CVRPLIB, varying the HGS budget from 800 to 1200 iterations, EvoHGS lowers the mean gap at every budget with runtime comparable to HGS-PyVRP (§ F.2). Beyond crossover, applying EvoHGS to population management (POP), which leaves more room for improvement, yields larger gains of up to 0.83% over HGS-PyVRP on the unseen variant OVRPBL (§ F.3). The NCO baselines are trained on 100-node instances; similarly, the LLM-based heuristic design methods use 100-node instances during evolutionary search. Evaluation on 500- and 1000-node instances therefore measures whether the learned or evolved operators remain effective beyond the scale used during training or search. We use the HGS-PyVRP solver (Wouda & Lan, 2024), limited to 1000 iterations, as our baseline. The performance of each evaluated method is reported as the percentage gap of its objective value relative to this baseline. Accordingly, a smaller gap denotes better performance, and a negative gap indicates an improvement over HGS-PyVRP.

The comparison in Table 1 **reveals two key observations** that highlight the distinctive strengths of EvoHGS. **First**, the results expose a fundamental limitation of directly learning to construct VRP solutions from scratch. NCO baselines can perform competitively near the training scale (100 nodes), but their gaps deteriorate sharply on larger instances, especially under additional routing constraints. **Second**, among LLM-based heuristic design methods, the central challenge is not only generating syntactically plausible code, but generating modifications that remain useful after being integrated into a mature HGS pipeline. Existing baselines such as ReEvo, OpenEvolve and EvoTune can occasionally find beneficial changes, but their gains are inconsistent across tasks and scales, as code-level modifications may interfere with HGS population diversity, local search, or constraint handling. This issue becomes more pronounced on constrained variants such as VRPTW and VRPL, where a useful modification must improve search behavior while still respecting time-window feasibility or route-length limits. In contrast, by co-evolving the LLM with HGS-compatible modifications, EvoHGS achieves better performance across most VRP variants.[3]

Ablation study. The left panel of Figure 2 ablates the main components of EvoHGS on CVRP instances, reporting the best of three runs per configuration (means and paired test statistics are in § D.2). The comparison shows that a stronger base LLM does not automatically yield better HGS modifications. Although GPT-5-nano outperforms the base Qwen3-8B model without SFT, a substantial number of its generated candidates fail to compile, wasting a large portion of the evaluation budget on invalid outputs. With SFT, the much smaller Qwen3-8B raises its compilable rate from 25.1% to 85.3% and attains a better performance than GPT-5-nano, indicating that modifying a mature solver such as HGS first requires the model to learn the expected code structure, interface conventions, and implementation constraints, which general-purpose pretraining alone does not

[3]Statistical significance tests are provided in § D.1.

Table 2: **Zero-shot generalization on unseen VRP variants.** Six representative variants, ordered by number of constraints (from the ten unseen variants in Table 10). Notation follows Table 1.

Method	Variant	$n = 100$		$n = 500$		$n = 1000$		Variant	$n = 100$		$n = 500$		$n = 1000$	
		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓
HGS-PyVRP	OVRPB	10.405	*	25.797	*	36.935	*	OVRPL	9.779	*	23.289	*	32.925	*
OR-Tools#		10.528	1.182	27.946	8.330	40.665	10.099		9.993	2.188	25.768	10.645	37.483	13.844
POMO		12.571	20.814	36.001	39.555	58.927	59.542		10.220	4.512	27.360	17.481	47.531	44.361
MVMoE		12.621	21.298	41.048	59.120	104.156	181.999		10.191	4.213	31.368	34.690	83.098	152.384
ReEvo		10.406	0.010	25.874	0.299	37.087	0.411		9.787	0.077	23.337	0.206	33.061	**0.414**
EvoHGS		10.403	**-0.019**	25.799	**0.008**	36.951	**0.044**		9.776	**-0.032**	23.295	**0.025**	33.077	0.463
HGS-PyVRP	VRPLTW	25.076	*	92.931	*	178.989	*	OVRPLTW	14.203	*	51.067	*	88.828	*
OR-Tools#		26.429	5.396	98.819	6.336	189.466	5.853		14.669	3.281	54.278	6.288	93.516	5.278
POMO		26.150	4.282	115.650	24.448	240.991	34.640		15.089	6.239	68.663	34.456	143.316	61.341
MVMoE		26.090	4.043	116.660	25.534	259.759	45.126		15.018	5.739	74.790	46.455	185.088	108.367
ReEvo		25.101	0.099	93.162	0.249	179.491	0.281		14.206	0.020	51.255	0.368	89.092	0.297
EvoHGS		25.065	**-0.046**	92.773	**-0.170**	178.998	**0.005**		14.191	**-0.085**	50.990	**-0.150**	88.667	**-0.181**
HGS-PyVRP	VRPBLTW	27.499	*	102.475	*	196.410	*	OVRPBLTW	15.760	*	54.231	*	97.088	*
OR-Tools#		28.802	4.738	109.914	7.259	207.133	5.459		16.158	2.525	57.554	6.127	101.766	4.818
POMO		31.587	14.867	139.492	36.122	288.241	46.755		18.197	15.462	78.738	45.190	161.601	66.447
MVMoE		31.595	14.896	140.898	37.495	310.670	58.174		18.128	15.026	85.590	57.825	207.802	114.035
ReEvo		27.509	0.035	102.835	0.351	196.790	0.193		15.760	-0.001	54.381	0.276	97.353	0.273
EvoHGS		27.450	**-0.177**	102.344	**-0.128**	196.134	**-0.140**		15.759	**-0.007**	54.087	**-0.265**	96.709	**-0.391**

Note: OR-Tools# uses a 300 s time limit.

provide. SFT provides this essential solver compatibility and establishes a reliable initialization for subsequent optimization. Building on this foundation, GRPO further leverages feedback from complete HGS evaluations to refine the policy toward high-performing modifications, delivering the best gap and highest compilable rate; it alone yields a statistically significant performance improvement ($p = 0.049$; § D.2), confirming the value of RL-based LLM co-evolution.

5.2 Zero-shot Generalization across VRP Variants

We further evaluate EvoHGS on ten unseen VRP variants that recombine the constraints of the seen ones (listed in Table 10). The setting is strictly zero-shot: the same evolved operator is applied as-is, without any further search, fine-tuning, or in-context adaptation. Table 2 reports on six representative variants; the comprehensive results for all ten evaluated variants are detailed in § E. EvoHGS outperforms HGS-PyVRP across all ten unseen variants at $n = 100$, providing evidence of zero-shot generation beyond the evolutionary training distribution. On larger instances, EvoHGS remains competitive. Overall, EvoHGS yields the lowest gaps across nearly all configurations. It significantly outperforms ReEvo, whose operators are worse than HGS-PyVRP in most unseen settings, suggesting a trade-off between variant-specific optimization and zero-shot transfer. OR-Tools and NCO fail to remain competitive in this setting. Since HGS-PyVRP is a mature, extensively tuned solver, the improvements are small in magnitude, yet statistically significant overall.[4] Finally, the ablation in § E suggests that SFT is the primary contributor to this zero-shot transferability.

6 Conclusion

We introduced **EvoHGS**, an LLM-guided evolutionary framework for automatically improving Hybrid Genetic Search across the VRP family by tightly coupling algorithmic discovery with language model adaptation. EvoHGS is capable of co-evolving (1) a pool of HGS operator modifications that are progressively refined through evolutionary search and (2) an LLM that learns via SFT and RL from solver feedback to generate more effective operators. The generated operators are integrated inside a strong metaheuristic (HGS) solver, thus preserving its overall search structure. The feedback from full HGS execution is then used to update the LLM, so that subsequent generations follow increasingly promising directions in the operator search space. Empirically, EvoHGS consistently outperforms the HGS-PyVRP, effectively transferring to unseen variants without additional evolution or fine-tuning. These results suggest that LLM-guided co-evolution offers a practical path toward discovering reusable, transferable improvement patterns for general solvers rather than variant-specific heuristics. Future work includes jointly evolving more interacting HGS operators and extending **EvoHGS** to additional components in HGS framework.

[4] Significance tests are reported in § E and Table 11.

AI USE STATEMENT

In this work, we used generative AI tools for generating synthetic data sets and implementing methods, both as core components of the proposed EvoHGS framework: Qwen3-14B-Instruct generates the SFT dataset (§ B.1); Qwen3-8B, fine-tuned via SFT (§ 4.1) and GRPO (§ 4.2.2), generates the HGS modifications explored by the evolutionary search (§ 4.2.1); and the LLM-based baselines use GPT-5-nano (ReEvo, OpenEvolve) or Qwen3-8B (EvoTune) (§ 5). Additionally, we used generative AI tools to create software code, as described above, and to edit the manuscript for readability. We have reviewed all AI-assisted work: all LLM-generated code is automatically checked for validity before evaluation (§ 4.2.1) and was verified and tested for correctness by the authors, and all AI-edited text was reviewed by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies the automated design of algorithmic components to evolve overall solver performance for vehicle routing problems, using synthetic instances and established benchmark datasets. It does not involve human-subject experiments or the collection of private or sensitive personal data. Our experiments focus on routing cost and do not explicitly assess broader considerations such as workload fairness or service equity. We report computational costs to make the resource requirements of the proposed approach transparent. We do not anticipate specific ethical concerns arising from this research.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we document the methodology and experimental settings of EvoHGS in the main text and appendices. § 4 describes the SFT objective (§ 4.1), evolutionary search mechanism (§ 4.2.1) and GRPO-based policy updates (§ 4.2.2). § B.1 details the automated SFT data construction pipeline and § B.2 provides the prompt templates used in the co-evolution loop. § 5 and § C specify the evaluation protocol, instance generation and data splits, training hyperparameters, generation and solver budgets, implementation details, statistical testing procedures, and computational costs. § D–F provide additional results, component ablations, and comparisons assessing transfer across variants and solver components. All associated artifacts, including the source code and the fine-tuned LLM weights, will be released upon acceptance to fully support experimental reproducibility.

REFERENCES

Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, André Hottung, Niels A Wouda, Leon Lan, Junyoung Park, Kevin Tierney, and Jinkyoo Park. Routefinder: Towards foundation models for vehicle routing problems. Transactions on Machine Learning Research, 2025(9):1–19, 2025.

Jieyi Bi, Yining Ma, Jianan Zhou, Wen Song, Zhiguang Cao, Yaoxin Wu, and Jie Zhang. Learning to handle complex constraints for vehicle routing problems. Advances in Neural Information Processing Systems, 37:93479–93509, 2024.

Jieyi Bi, Zhiguang Cao, Jianan Zhou, Wen Song, Yaoxin Wu, Jie Zhang, Yining Ma, and Cathy Wu. Towards efficient constraint handling in neural solvers for routing problems. arXiv preprint arXiv:2602.16012, 2026.

Miroslaw Blocho and Jakub Nalepa. Lcs-based selective route exchange crossover for the pickup and delivery problem with time windows. In European Conference on Evolutionary Computation in Combinatorial Optimization, pp. 124–140. Springer, 2017.

Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. Transportation science, 39(1):104–118, 2005.

Paulo R d O Costa, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In Asian conference on machine learning, pp. 465–480. PMLR, 2020.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 39, pp. 26931–26938, 2025.

Han Fang, Zhihao Song, Paul Weng, and Yutong Ban. Invit: A generalizable routing problem solver with invariant nested view transformer. In International Conference on Machine Learning, pp. 12973–12992, 2024.

Chengrui Gao, Haopu Shang, Ke Xue, Dong Li, and Chao Qian. Towards generalizable neural solvers for vehicle routing problems via ensemble with transferrable local policy. In Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, pp. 6914–6922, 2024.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. In International Conference on Machine Learning, pp. 19034–19055. PMLR, 2025.

Keld Helsgaun. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. Roskilde: Roskilde University, 12:966–980, 2017.

André Hottung and Kevin Tierney. Neural large neighborhood search for routing problems. Artificial Intelligence, 313:103786, 2022.

André Hottung, Paula Wong-Chung, and Kevin Tierney. Neural deconstruction search for vehicle routing problems. Transactions on Machine Learning Research.

André Hottung, Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, Daniel Wetzel, Michael Römer, Haoran Ye, Davide Zago, Michael Poli, Stefano Massaroli, et al. Vrpagent: Llm-driven discovery of heuristic operators for vehicle routing problems. arXiv preprint arXiv:2510.07073, 2025.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. Iclr, 1(2):3, 2022.

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.

Ziyao Huang, Weiwei Wu, Kui Wu, Jianping Wang, and Wei-Bin Lee. Calm: Co-evolution of algorithms and language model for automatic heuristic design. In International Conference on Learning Representations, 2026.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In 29th International Conference on Software Engineering (ICSE'07), pp. 96–105. IEEE, 2007.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya B Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. Ledex: Training llms to better self-debug and explain code. Advances in Neural Information Processing Systems, 37:35517–35543, 2024.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. Large language models as end-to-end combinatorial optimization solvers. 2025a.

Xia Jiang, Yaoxin Wu, Chenhao Zhang, and Yingqian Zhang. Droc: Elevating large language models for complex vehicle routing via decomposed retrieval of constraints. In 13th International Conference on Learning Representations, ICLR 2025. OpenReview. net, 2025b.

Daisuke Kikuta, Hiroki Ikeuchi, Kengo Tajiri, and Yuusuke Nakano. Routeexplainer: An explanation framework for vehicle routing problem. In Pacific-asia conference on knowledge discovery and data mining, pp. 30–42. Springer, 2024.

Minsu Kim, Junyoung Park, and Jinkyoo Park. Sym-nco: Leveraging symmetricity for neural combinatorial optimization. Advances in Neural Information Processing Systems, 35:1936–1949, 2022.

Minsu Kim, Sanghyeok Choi, Hyeonah Kim, Jiwoo Son, Jinkyoo Park, and Yoshua Bengio. Ant colony sampling with gflownets for combinatorial optimization. Proceedings of Machine Learning Research, 258:469–477, 2025.

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! arXiv preprint arXiv:1803.08475, 2018.

Wouter Kool, Joep Olde Juninck, Ernst Roos, Kamiel Cornelissen, Pim Agterberg, Jelke van Hoorn, and Thomas Visser. Hybrid genetic search for the vehicle routing problem with time windows: A high-performance implementation. In 12th dimacs implementation challenge workshop, pp. 131–135. DIMACS, Rutgers University Piscataway, NJ, 2022.

Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. Advances in neural information processing systems, 33:21188–21198, 2020.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. arXiv preprint arXiv:2509.19349, 2025.

Kai Li, Fei Liu, Zhenkun Wang, Xialiang Tong, Xiongwei Han, Mingxuan Yuan, and Qingfu Zhang. Ars: Automatic routing solver with large language models. arXiv preprint arXiv:2502.15359, 2025.

Sirui Li, Zhongxia Yan, and Cathy Wu. Learning to delegate for large-scale vehicle routing. Advances in neural information processing systems, 34:26198–26211, 2021.

Fei Liu, Xi Lin, Zhenkun Wang, Qingfu Zhang, Tong Xialiang, and Mingxuan Yuan. Multi-task learning for routing problem with cross-problem zero-shot generalization. In Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 1898–1908, 2024a.

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. 2024b.

Fei Liu, Rui Zhang, Xi Lin, Zhichao Lu, and Qingfu Zhang. Fine-tuning large language model for automated algorithm design. arXiv preprint arXiv:2507.10614, 2025.

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.

Fu Luo, Xi Lin, Yaoxin Wu, Zhenkun Wang, Tong Xialiang, Mingxuan Yuan, and Qingfu Zhang. Boosting neural combinatorial optimization for large-scale vehicle routing problems. In The Thirteenth International Conference on Learning Representations, 2025.

Yining Ma, Zhiguang Cao, and Yeow Meng Chee. Learning to search feasible and infeasible regions of routing problems with flexible neural k-opt. Advances in Neural Information Processing Systems, 36:49555–49578, 2023.

Manuj Malik, Jianan Zhou, Shashank Reddy Chirra, and Zhiguang Cao. Pyvrp+: Llm-driven metacognitive heuristic evolution for hybrid genetic search in vehicle routing problems. In Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems, 2026.

Yuichi Nagata and Shigenobu Kobayashi. A memetic algorithm for the pickup and delivery problem with time windows using selective route exchange crossover. In International Conference on Parallel Problem Solving from Nature, pp. 536–545. Springer, 2010.

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

Laurent Perron and Vincent Furnon. Or-tools. URL https://developers.google.com/optimization/.

Ruizhong Qiu, Zhiqing Sun, and Yiming Yang. Dimes: A differentiable meta solver for combinatorial optimization problems. Advances in Neural Information Processing Systems, 35:25531–25546, 2022.

Eduardo Queiroga, Yuri Frota, Ruslan Sadykov, Anand Subramanian, Eduardo Uchoa, and Thibaut Vidal. On the exact solution of vehicle routing problems with backhauls. European Journal of Operational Research, 287(1):76–89, 2020.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. Nature, 625(7995):468–475, 2024.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation science, 40(4):455–472, 2006.

Alberto Santini, Michael Schneider, Thibaut Vidal, and Daniele Vigo. Decomposition strategies for vehicle routing heuristics. INFORMS Journal on Computing, 35(3):543–559, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv preprint arXiv:2402.03300, 2024.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github. com/codelion/openevolve, 5, 2025.

Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In International conference on principles and practice of constraint programming, pp. 417–431. Springer, 1998.

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

Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, Zhiguang Cao, and Jie Zhang. Generalizable heuristic generation through llms with meta-optimization. In The Fourteenth International Conference on Learning Representations.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. arXiv preprint arXiv:2601.03267, 2025.

Zhiqing Sun and Yiming Yang. Difusco: Graph-based diffusion solvers for combinatorial optimization. Advances in neural information processing systems, 36:3706–3731, 2023.

Anja Šurina, Amin Mansouri, Lars CPM Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning. In Second Conference on Language Modeling, 2025.

Paolo Toth and Daniele Vigo. Vehicle routing: problems, methods, and applications. SIAM, 2014.

Cong Dao Tran, Quan Nguyen-Tri, Huynh Thi Thanh Binh, and Hoang Thanh-Tung. Large language models powered neural solvers for generalized vehicle routing problems. In Towards Agentic AI for Science: Hypothesis Generation, Comprehension, Quantification, and Validation, 2025.

Thibaut Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. Computers & Operations Research, 140:105643, 2022.

Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. Operations Research, 60 (3):611–624, 2012.

Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. A unified solution framework for multi-attribute vehicle routing problems. European Journal of Operational Research, 234(3):658–673, 2014.

Niels A. Wouda and Leon Lan. Pyvrp: A high-performance vehicle routing problem solver package. INFORMS Journal on Computing, 36(4):943–955, 2024.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2, pp. 3228–3239, 2025.

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems. IEEE transactions on neural networks and learning systems, 33(9): 5057–5069, 2021.

Zhenxing Xu, Yizhe Zhang, Weidong Bao, Hao Wang, Ming Chen, Haoran Ye, Wenzheng Jiang, Hui Yan, and Ji Wang. Autoep: Llms-driven automation of hyperparameter evolution for metaheuristic algorithms. arXiv preprint arXiv:2509.23189, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. arXiv preprint arXiv:2505.09388, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In The Twelfth International Conference on Learning Representations, 2023.

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. Advances in neural information processing systems, 37:43571–43608, 2024.

Zikang Yu, Jinbiao Chen, and Jiahai Wang. Combination-of-experts with knowledge sharing for cross-task vehicle routing problems. In The Fourteenth International Conference on Learning Representations, 2026.

Ni Zhang, Zhiguang Cao, Jianan Zhou, Cong Zhang, and Yew-Soon Ong. An agentic framework with llms for solving complex vehicle routing problems. arXiv preprint arXiv:2510.16701, 2025.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In Forty-second International Conference on Machine Learning, 2025.

Jianan Zhou, Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Towards omni-generalizable neural methods for vehicle routing problems. In International conference on machine learning, pp. 42769–42789. PMLR, 2023.

Jianan Zhou, Zhiguang Cao, Yaoxin Wu, Wen Song, Yining Ma, Jie Zhang, and Xu Chi. Mvmoe: Multi-task vehicle routing solver with mixture-of-experts. In International Conference on Machine Learning, pp. 61804–61824. PMLR, 2024.

Rongjie Zhu, Cong Zhang, and Zhiguang Cao. Refining hybrid genetic search for cvrp via reinforcement learning-finetuned llm. In International Conference on Learning Representations, 2026.

A Extended Related Work on Neural Solvers

Despite extensive prior work (Sun & Yang, 2023; Kim et al., 2025), neural solvers for VRPs largely fall into two paradigms. Construction solvers learn to build solutions from scratch in an autoregressive manner. Attention-based encoder-decoder models (Kool et al., 2018) formulate routing as sequential decision-making and learn policies via reinforcement learning. Subsequent work improves solver along multiple directions, including multi-start augmentation (Kwon et al., 2020; Kim et al., 2022), cross-task knowledge sharing (Yu et al., 2026; Qiu et al., 2022), complex constraints (Bi et al., 2024; 2026), and generalization across distributions (Fang et al., 2024; Gao et al., 2024), scales (Zhou et al., 2023; Luo et al., 2025) and constraints (Zhou et al., 2024; Berto et al., 2025). Meanwhile, Improvement solvers start from an initial solution and iteratively refine it via learned operators that mirror classical local search primitives (Shaw, 1998; Bräysy & Gendreau, 2005) such as k-opt (d O Costa et al., 2020; Wu et al., 2021; Ma et al., 2023) and large neighborhood search (Hottung & Tierney, 2022; Hottung et al.). In contrast, EvoHGS does not learn a solver from scratch; instead, it improves an existing high-performance solver, HGS, by modifying its internal behavior via an LLM while preserving its overall structure.

B Method Details

This appendix details the two methodological components summarized in the main text. Appendix B.1 describes the automated pipeline that builds the SFT dataset of § 4.1, from training-instance generation and HGS-compatible code evolution to the synthesis of modification ideas and the structure of the training examples. Appendix B.2 presents the prompt templates from which EvoHGS assembles each query throughout the co-evolution loop in § 4.2.

B.1 SFT Data Construction

This appendix details our automated pipeline for constructing the Supervised Fine-Tuning (SFT) dataset, which teaches the model to map conceptual modifications to executable, HGS-compatible code. Each dataset sample is a prompt–output pair where the prompt specifies the VRP variant and HGS interface constraints, while the target output provides a conceptual idea followed by its C++ implementation. Focusing primarily on the crossover operator, the core challenge is ensuring HGS-compatibility. We address this through a three-stage pipeline: (1) generating VRP training instances as an evaluation testbed, (2) evolving executable code using ReEvo, and (3) extracting the underlying design ideas via LLM-based code analysis.

VRP Training Instance Generation. We generate training instances for six representative VRP variants: CVRP, OVRP, VRPB, VRPTW, VRPL, and OVRPTW. These are detailed in Appendix C.1 and follow the MVMoE (Zhou et al., 2024) setup. For each variant, we create 50 instances at three different scales, specifically 50, 200, and 400 nodes.

HGS-Compatible Code Evolution. To generate HGS-compatible crossover implementations, we adapt the ReEvo (Ye et al., 2024) framework using Qwen3-14B-Instruct. This specific model is selected to maintain a favorable balance between code-generation quality and computational feasibility, since the evolutionary search requires evaluating thousands of iteratively refined candidates. Each generated operator is compiled within the full solver and evaluated on the problem instances detailed in Appendix B.1. Ultimately, this automated search process yields a comprehensive dataset comprising 14,400 training and 2,700 validation examples across six representative VRP tasks (specifically, 2,400 training and 450 validation examples per task).

Prompt Used for CVRP-HGS. Using CVRP as a representative example, we structure our prompts based on the ReEvo framework. However, we make one critical modification: instead of generating isolated Python heuristics, we strictly constrain the LLM to output a complete, compilable C++17 source file (`selective_route_exchange.cpp`) that directly implements PyVRP's selective route exchange (SREX) operator.

Task Description: This component assigns the LLM the role of an optimization expert and defines the core objective.

```
[Problem Setting & Task]
# ROLE: Expert C++ Optimization Engineer for Vehicle Routing Problems

You are a senior C++ optimization engineer with expertise in
    algorithmic
optimization, particularly for Vehicle Routing Problems (VRP). Your
    task
is to analyze and improve the 'selective_route_exchange.cpp' file's
    crossover algorithm.

## TASK OVERVIEW
You are given the file 'selective_route_exchange.cpp' (full listing
below). Your task is to modify this crossover algorithm to create
children with smaller cost while keeping the interface intact.

## CONSTRAINTS
Vehicle capacity must remain feasible, and each route starts and ends
    at the depot.

## OPERATION
Identify a specific logic seed within the following algorithm and
    perform one
small, reliable modification to it. Preserve the parent algorithm's
    main
control-flow invariant. Do not add new unbounded 'while' / retry loops
    .
```

Exported Signature: This block strictly defines the C++ interface that must remain untouched throughout the evolution.

```
[Exported Signature to Preserve]
// Exact exported signature to preserve. Do not rename it or add
    parameters.
pyvrp::Solution pyvrp::crossover::selectiveRouteExchange(
    std::pair<Solution const *, Solution const *> const &parents,
    ProblemData const &data,
    CostEvaluator const &costEvaluator,
    std::pair<size_t, size_t> const &startIndices,
    size_t const numMovedRoutes)
```

External Knowledge and Hard Constraints: This section imposes rigid rules on output formatting, evaluation metrics, and verifiable API usage.

```
[External Knowledge and Hard Constraints]
## DOMAIN KNOWLEDGE: CROSSOVER OPERATIONS

The Selective Route Exchange (SREX)is a crossover operation for the
    Vehicle
Routing Problem Family. The algorithm:
1. Selects routes from two parent solutions
2. Exchanges these routes to create offspring
3. Aims to preserve beneficial route structures while creating new
    combinations

### Key Optimization Areas to Consider:
- Route selection strategy (which routes to exchange)
- Client-to-route assignment decisions
- Proximity/distance calculations between routes or clients
- Handling of task specific constraints
- Diversity generation in offspring solutions

## Essential Fields and Methods for SREX
```

```
**ProblemData Key Methods:**
- `centroid()` - Returns `std::pair<Coordinate, Coordinate>` center of
    all client locations
- `numLocations()` - Returns `size_t` total number of locations (
   depots + clients)

**Solution Key Methods:**
- `numRoutes()` - Returns `size_t` number of routes in the solution
- `routes()` - Returns `std::vector<Route> const &`

**Route Key Methods:**
- `centroid()` - Returns `std::pair<Coordinate, Coordinate>` center of
    route's client locations
- `vehicleType()` - Returns `VehicleType` (`size_t`) vehicle type
   index
- `begin()` / `end()` - Iterator support for visiting clients in route
- `size()` - Returns `size_t` number of clients in route
- `visits()` - Returns the route visits as a client list

**Route Construction:**
- `Route(data, visits, vehicleType)` - Constructor taking `std::vector
   <Client>` visits and vehicle type

**Cost Evaluation:**
- `costEvaluator.penalisedCost(solution)` - Returns the penalised
   objective used to compare offspring solutions

**Selection and Offspring Structure Used by SREX:**
- `parents` - Passed as `std::pair<Solution const *, Solution const
   *>`
- `startIndices` - Passed as `std::pair<size_t, size_t>`
- `numMovedRoutes` - Passed as `size_t`
- `selectiveRouteExchange(...)` - Returns one `Solution` offspring

## HARD RULES (Mandatory Verification Checklist)

1. Keep the function signature and namespace exactly the same:
   `pyvrp::crossover::selectiveRouteExchange(...)`

2. The file must still compile under the current project build flags
   with the current `#include` lines.
   You may NOT remove `#include` directives.
   Treat compiler warnings as build-breaking: do not introduce unused
   variables, unused parameters, or other warning-level issues.

3. Do not change any public headers, class interfaces, or external
   behaviour except for the improved offspring quality.

4. DO NOT fabricate or use non-existent or unmentioned attributes or
   methods.
   Verify every method you use exists in the provided code or parent
   implementations.
   In particular, do not invent APIs like `data.client(...)`, `data.
   clients(...)`,
   `route.visits()`, or new helper methods that are not in the
   provided code.

5. Wrap the code you output in fenced code blocks using ```cpp and
   ```.
```

**Evolutionary Search Prompts.** Rather than redundantly listing the templates for each evolutionary step (Population Initialization, Short-Term Reflection, Crossover, and Elitist Mutation), we summarize our core adaptation: the fundamental shift from isolated Python snippets to full-file
```

C++17 generation. Across all prompts, the LLM must process, compare, and output complete source files. For instance, the initialization seed is explicitly the **original, unmodified baseline** `selective_route_exchange.cpp` file. Consequently, during crossover and mutation, rather than generating partial patches, the LLM is strictly instructed to physically integrate textual reflection hints into a newly generated, fully compliant C++17 file enclosed in ```` ```cpp ... ``` ````.

Modification Idea Synthesis. While the adapted ReEvo framework discovers high-performing C++ implementations, the raw code lacks the design rationale required for an **idea-driven evolutionary loop**. To bridge this gap, we deploy Qwen3-14B-Instruct to reverse-engineer these ideas by systematically comparing each evolved crossover operator against the original HGS baseline. The prompt used for this extraction is designed as follows:

````
Compare the modified C++ implementation against the original baseline
    below.
Identify only meaningful semantic or behavioral modifications,
    ignoring
formatting, comments, renaming, and trivial refactoring.

Output plain text only. Do not output reasoning, analysis, JSON,
    markdown,
bullets, code fences, or <think> tags.

Write one sentence per meaningful modification. Put each modification
    sentence
on its own line. Each sentence must be a forward-looking patch
    instruction in
imperative mood, starting with a concrete action verb such as Add,
    Replace,
Reinitialize, Sort, Filter, Move, Restrict, Compute, or Adjust.
    Mention the
concrete changed entity, condition, data structure, operation, or
    logic block
immediately after the opening verb.

Do not begin sentences with "The modified code" or "The baseline code
    ".
Do not use retrospective phrasing such as "The patch added..." or
    passive
summaries such as "X is reinitialized". Do not make vague claims such
    as
"improves performance" or "improves solution quality".

After the modification list, add one blank line and then one
    concluding
paragraph of at most 3 sentences. The concluding paragraph must
    summarize the
concrete code area and modification. Its final sentence must begin
    with
exactly one of: Thus, Therefore, In summary, and must state the
    concrete code
modification. Do not write anything after that final sentence.

Baseline code:
```cpp
{original_code}
```

Modified code:
```cpp
{modified_code}
```
````

Structuring the Training Examples. With the modification ideas successfully extracted, we assemble the instruction-tuning pairs for the SFT dataset $\mathcal{D}_{\text{SFT}}$. We structure each training instance using a prompt format. To conserve space and maintain narrative flow, the complete text of these prompt templates is deferred to Appendix B.2. The logical structure is defined as follows:

Input: `[Task Description]` (Listing 1) + `[Domain Knowledge]` (Listing 2) + `[Generation Instructions]` (Listing 3) + `[Mechanism Selection (M2)]` (Listing 6)

Output: `[Modification Idea]` (derived in § B.1) + `[Evolved C++ Implementation]` (generated in § B.1)

By placing the modification idea in the target output, it requires the LLM to explicitly formulate a logical rationale before writing any code, ensuring that the final C++ implementation is both algorithmically sound and strictly compliant with the HGS framework. Specifically, the seed code used in M2 is consistent with that in § B.1, both being the original HGS implementation.

B.2 Prompts Used in EvoHGS

This appendix details the prompt templates used by EvoHGS to iteratively evolve PyVRP's selective route exchange (SREX) crossover operator. To ensure the LLM generates logically sound and syntactically valid C++ code, each prompt is dynamically assembled from five functional components.

Task Description This block sets the initial context. It assigns the LLM the persona of an expert C++ optimization engineer and clearly defines the end goal: to modify the `selective_route_exchange.cpp` algorithm to yield lower-cost routes without violating the specific constraints of the target VRP variant.

Listing 1: Template of the task description prompt.

```
# ROLE: Expert C++ Optimization Engineer for Vehicle Routing Problems

You are a senior C++ optimization engineer with expertise in
    algorithmic
optimization, particularly for Vehicle Routing Problems (VRP).
Your task is to analyze and improve the 'selective_route_exchange.cpp'
file's crossover operator.

## TASK OVERVIEW
You are given the file 'selective_route_exchange.cpp' (full listing
    below).
Your task is to modify this crossover algorithm to create children
    with
smaller cost while keeping the interface intact.

## CONSTRAINTS
Required route properties:
<variant-specific constraints>
```

Domain Knowledge To prevent the LLM from hallucinating non-existent variables or unsupported methods, this block provides a strict, safe API reference. It outlines the essential fields and methods available within the PyVRP framework, grounding the generated code in the actual codebase.

Listing 2: Template of the domain-knowledge prompt.

```
## Extra Information
## DOMAIN KNOWLEDGE: CROSSOVER OPERATIONS

The Selective Route Exchange (SREX)is a crossover operation for the
    Vehicle Routing Problem Family. The algorithm:
1. Selects routes from two parent solutions
2. Exchanges these routes to create offspring
```

Table 3: **Task-specific constraint blocks used in EvoHGS prompts.** Each block specifies the constraints associated with a particular VRP variant and is inserted into the corresponding task description.

VRP variant	Constraint block
CVRP	Vehicle capacity must remain feasible, and each route starts and ends at the depot.
OVRP	Vehicle capacity must remain feasible; routes are open (not required to return to the depot).
VRPTW	Vehicle capacity must remain feasible; each customer must be served within its time window.
OVRPTW	Vehicle capacity, open-route validity, and customer time windows must all be respected.
VRPL	Vehicle capacity must remain feasible; each route must satisfy the maximum route-length constraint.
VRPB	Vehicle capacity must remain feasible; linehaul customers must be served before backhaul customers.

```
3. Aims to preserve beneficial route structures while creating new
    combinations

### Key Optimization Areas to Consider:

- Route selection strategy (which routes to exchange)
- Client-to-route assignment decisions
- Proximity/distance calculations between routes or clients
- Handling of task specific constraints
- Diversity generation in offspring solutions

## Essential Fields and Methods for SREX

**ProblemData Key Methods:**
- `centroid()` - Returns `std::pair<Coordinate, Coordinate>` center of
     all client locations
- `numLocations()` - Returns `size_t` total number of locations (
    depots + clients)

**Solution Key Methods:**
- `numRoutes()` - Returns `size_t` number of routes in the solution
- `routes()` - Returns `std::vector<Route> const &`

**Route Key Methods:**
- `centroid()` - Returns `std::pair<Coordinate, Coordinate>` center of
     route's client locations
- `vehicleType()` - Returns `VehicleType` (`size_t`) vehicle type
    index
- `begin()` / `end()` - Iterator support for visiting clients in route
- `size()` - Returns `size_t` number of clients in route
- `visits()` - Returns the route visits as a client list

**Route Construction:**
- `Route(data, visits, vehicleType)` - Constructor taking `std::vector
    <Client>` visits and vehicle type

**Cost Evaluation:**
- `costEvaluator.penalisedCost(solution)` - Returns the penalised
    objective used to compare offspring solutions

**Selection and Offspring Structure Used by SREX:**
- `parents` - Passed as `std::pair<Solution const *, Solution const
    *>`
- `startIndices` - Passed as `std::pair<size_t, size_t>`
- `numMovedRoutes` - Passed as `size_t`
- `selectiveRouteExchange(...)` - Returns one `Solution` offspring
```

Generation Instructions This is the core of the Chain-of-Thought (CoT) mechanism. It mandates a strict reasoning sequence (analyzing before coding), enforces the exact formatting of the generated modification idea, and imposes hard compilation constraints to ensure structural validity.

Listing 3: Template of the common generation-instruction prompt.

```
## THINKING PROCESS REQUIREMENTS

Keep the thinking concise and high density.
Do not write long or repetitive analysis.

1. First, thoroughly analyze the current implementation to understand:
- The algorithm's purpose and workflow
- Key decision points and heuristics
- Performance bottlenecks or optimization opportunities
- Any constraints that must be preserved

2. For your chosen modification:
- Justify why it's likely to improve solution quality
- Verify it maintains the function signature and behavior
- Double-check for compatibility with the rest of the codebase
- Consider edge cases and verify robustness

############################################################
## IDEA FORMAT REQUIREMENTS

Before the final code block, write exactly one short introductory
    sentence that begins with:
'The idea is:'

In the Idea section:

1. After the introductory sentence, list all meaningful planned
    modifications as forward-looking patch instructions.
- Put each modification instruction on its own line.
- Start each instruction with a concrete action verb such as 'Add', '
    Replace', 'Reinitialize', 'Sort', 'Filter', 'Move', 'Restrict', '
    Compute', or 'Adjust'.
- Mention the concrete changed entity, logic block, condition, data
    structure, or operation immediately after the opening action verb.
- Keep each instruction concise and specific.
- Do not use retrospective phrasing such as "The modified code..." or
    "The patch added...".
- Do not include formatting-only changes, renaming, comments, or
    trivial refactoring.
- Keep the entire Idea section concise and high density.

2. Then add one blank line followed by exactly one concluding sentence
     that summarizes the overall modification.
- The concluding sentence must be the last sentence of the Idea
    section.
- It must clearly identify the part of the code being changed and the
    concrete modification.
- The concluding sentence must begin with exactly: 'Thus,'
- That concluding sentence must state the concrete code modification.
- Do not write any sentence after it.
- Output the code block immediately after the Idea section.
- Do not insert any extra explanation, verification text, or summary
    between the Idea section and the code block.

############################################################
## HARD RULES (Mandatory Verification Checklist)

1. Keep the function signature and namespace exactly the same:
   'pyvrp::crossover::selectiveRouteExchange(...)'
```

```
2. The file must still compile under the current project build flags
    with the current '#include' lines.
   You may NOT remove '#include' directives.
   Treat compiler warnings as build-breaking: do not introduce unused
    variables, unused parameters, or other warning-level issues.

3. Do not change any public headers, class interfaces, or external
    behaviour except for the improved offspring quality.

4. DO NOT fabricate or use non-existent or unmentioned attributes or
    methods.
   Verify every method you use exists in the provided code or parent
    implementations.
   In particular, do not invent APIs like 'data.client(...)', 'data.
    clients(...)',
   'route.visits()', or new helper methods that are not in the
    provided code.

5. Wrap the code you output in fenced code blocks using ```cpp to open
     and ''' to close.

6. Before finalizing, double-check that your modification:
- Does not introduce new parameters
- Does not change the function's contract
- Is focused on improving solution quality, not runtime
- Is fully compatible with the existing codebase
- Uses only documented methods and attributes
- Would still compile if every newly referenced method were checked
    against the provided headers
- Does not leave behind unused helper arguments, unused locals, or
    warning-triggering code paths
- Preserves a valid client partition in the offspring: every client
    must still appear at most once and no kept client may be silently
    dropped
- Does not rely on a new unbounded repair/search loop: every new loop
    must have an explicit finite bound or a strictly improving move
    that matches the applied state update
- Does not add post-hoc pruning or filtering that "repairs" the
    offspring by silently deleting clients or depending on emptiness
    checks outside the existing partition construction

############################################################
## Key considerations for high-quality solutions:
- More efficient route structures (fewer vehicles, shorter routes)
- Better client assignment to routes based on spatial relationships
- Improved handling of the active capacity constraints in this VRP
    variant
- Preservation of high-quality route segments during crossover
- Better diversity in the generated offspring
```

Mechanism Selection To drive the evolutionary search, one of three distinct evolutionary mechanisms is dynamically appended to the prompt. As described in 4.2.1, each mechanism steers the LLM toward a different modification strategy.

Listing 4: E1: Crossover prompt.

```
## MECHANISM OBJECTIVE: E1

## E1 SELECTION CONTEXT
- Perform Crossover, not code-level splicing.
- Combine the idea from Parent A and Parent B.
- Do not use either parent as the default skeleton with only minor
    edits.
```

```
## PARENT A
{{parent_a_content}}

## PARENT B
{{parent_b_content}}

Generate one new algorithm that integrates both parent ideas into a
    single
coherent child. If the two ideas conflict, resolve the conflict
    explicitly
while preserving a non-trivial contribution from each parent. The
    child must
be meaningfully different from both parents in its decision logic, not
     just
in wording, comments, or formatting. Then modify the code according to
     that
unified plan.
```

Listing 5: M1: Injection prompt.

```
## MECHANISM OBJECTIVE: M1

## EXPLORED IDEA BLACKLIST
Exclude the following components that have already been explored:
{{recent_ideas_blacklist}}

You do not need a globally new idea, but the injected component must
    be
meaningfully different from these recent idea directions.
```

Listing 6: M2: Modification prompt.

```
## MECHANISM OBJECTIVE: M2

Identify a specific logic seed within the following algorithm and
    perform one
small, reliable modification to it. Preserve the parent algorithm's
    main
control-flow invariant. Do not add new unbounded 'while' / retry loops
    .

## ALGORITHM TO MODIFY
{{parent_algorithm_code}}
```

Candidate Parent Rendering Finally, this block injects the selected parent candidates into the prompt. It provides the LLM with the concrete context—performance metrics, underlying ideas, and current C++ implementations—needed to execute the specified evolutionary mechanism.

Listing 7: Template of the parent rendering prompt.

```
## Algorithm k
- Performance: {candidate.performance}
- Idea: {candidate.idea}
- Code:
```cpp
{candidate.code}
```
```

C Experimental Setup

This appendix specifies the experimental setup of § 5: including the VRP variants and their instance generation (Appendix C.1), the training problem set and the SFT configuration, the implementation

of the co-evolutionary framework (Appendix C.3), the statistical testing protocol shared by all significance tests (Appendix C.4) and the computational cost of operator design (Appendix C.5).

Following standard practice in automatic heuristic design, all LLM-based experiments are conducted independently three times. Table 1 reports the mean across the archived runs, with all LLM-based methods evaluated at 1000 HGS iterations; Appendix D.1 reports the corresponding sample standard deviations. Figure 2 reports the best of three runs per configuration. The separately repeated CVRP and OVRP experiments in Appendices D.1 and F.3, as well as the zero-shot ablations and POP experiments, report the mean and standard deviation over three independent runs at 1000 HGS iterations. For reproducibility, all HGS-based evaluations use PyVRP version 0.12.2 at commit `ea0c421`.

C.1 VRP Variants Considered

A vehicle routing problem (VRP) is defined on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_0, v_1, \ldots, v_n\}$ consists of a depot v_0 and customer nodes, and $\mathcal{E}$ represents edges with associated travel costs. A solution is a set of routes that start from the depot and together visit all customers. Let $\mathcal{S}$ denote the set of feasible solutions. The objective is to find

$$s^* = \arg\min_{s \in \mathcal{S}} f(s),$$

where $f(s)$ is the total travel cost. We use the capacitated VRP (CVRP) as the base setting. Each customer v_i has a demand d_i, and each vehicle has capacity Q. A solution is feasible if every customer is visited exactly once and the total demand assigned to each route does not exceed Q. Following the experimental setup of (Zhou et al., 2024), we construct 16 VRP variants by augmenting the CVRP with different combinations of additional constraints, as summarized in Table 4. The node coordinates are sampled uniformly from the unit square $[0, 1]^2$.

Table 4: **The 16 VRP variants and their constraints.** Each variant combines capacity (C) with a subset of open route (O), backhaul (B), duration limit (L), and time window (TW).

Variant	C	O	B	L	TW	Variant	C	O	B	L	TW
CVRP	✓					VRPBL	✓		✓	✓	
OVRP	✓	✓				VRPBTW	✓		✓		✓
VRPB	✓		✓			VRPLTW	✓			✓	✓
VRPL	✓			✓		OVRPBL	✓	✓	✓	✓	
VRPTW	✓				✓	OVRPBTW	✓	✓	✓		✓
OVRPTW	✓	✓			✓	OVRPLTW	✓	✓		✓	✓
OVRPB	✓	✓	✓			VRPBLTW	✓		✓	✓	✓
OVRPL	✓	✓		✓		OVRPBLTW	✓	✓	✓	✓	✓

We next describe the data-generation procedure for each constraint.

- **Capacity (C):** Following the experimental setup in (Zhou et al., 2024), we extend the dataset scale to 500 and 1,000 nodes. The vehicle capacity Q is set to $Q = \{40, 50, 80, 100, 100, 150\}$ for $n = \{50, 100, 200, 400, 500, 1000\}$, respectively. For each customer i, the linehaul demand d_i is sampled from the integer set $\{1, 2, \ldots, 9\}$.
- **Open Route (O):** The open-route constraint changes the route structure by allowing each vehicle to finish at any customer node instead of returning to the depot. It is implemented by setting a binary indicator $o = 1$, without introducing additional instance data. When combined with other constraints, the feasibility checks are adjusted dynamically:
  - With duration limit (L), the route length is computed without the return-to-depot distance.
  - With time windows (TW), arrival-time calculations omit the depot return segment.
  - With capacity (C) and backhaul (B), the constraint logic remains unchanged, while the depot return requirement is removed.
- **Backhaul (B):** Customer demands are first sampled from the integer set $\{1, 2, ..., 9\}$. Then, 20% of customers are randomly selected as backhaul customers, while the remaining customers are treated as linehaul customers. For selected backhaul customers, the sampled

demand is used as pickup demand, and their linehaul demand is set to 0. For linehaul customers, the sampled demand is used as delivery demand, and their backhaul demand is set to 0. As a result, each customer has only one type of demand. Routes may contain both linehaul and backhaul customers. **As in the classical VRPB**, each route must serve all linehaul customers before any backhaul customer, thereby forbidding a transition from a backhaul customer back to a linehaul customer within the same route (Queiroga et al., 2020).

- **Duration Limit (L):** Following (Liu et al., 2024a) and (Zhou et al., 2024), we set the maximum route length to $L = 3$ for each vehicle, which is suitable for instances generated in the unit square and helps maintain route feasibility.
- **Time Window (TW):** Following (Li et al., 2021) and (Zhou et al., 2024), we set the depot time window to $[e_0, l_0] = [0, 3]$ and assign a fixed service time $s_i = 0.2$ to each customer v_i, while the depot has zero service time. For each customer v_i, we first sample the time-window center as $\gamma_i \sim \mathcal{U}(e_0 + c_{0i},\, l_0 - c_{i0} - s_i)$, where $c_{0i} = c_{i0}$ denotes the travel time between the depot v_0 and customer v_i. We then sample the half-width as $h_i \sim \mathcal{U}\left(\frac{s_i}{2},\, \frac{l_0}{3}\right) = \mathcal{U}(0.1, 1)$, and define the customer time window by clipping the interval around γ_i to the depot horizon: $[e_i, l_i] = [\max(e_0, \gamma_i - h_i), \min(l_0, \gamma_i + h_i)]$.

C.2 Training Problem Set and Supervised Fine-Tuning

Training problem set. As mentioned in § 5 and following the experimental setup of (Zhou et al., 2024), our EvoHGS framework is evolved on six VRP variants: CVRP, OVRP, VRPB, VRPL, VRPTW, and OVRPTW. For each variant, we consider three instance scales ($n \in \{100, 500, 1000\}$). Specifically, for $n = 100$, we generate 100 instances in total: 50 are reserved for the evolutionary optimization process, while the remaining 50 serve as the test set. For the larger scales ($n = 500$ and $n = 1000$), 50 instances are generated for each, exclusively used for performance evaluation.

Supervised fine-tuning. We perform SFT on the Qwen3-8B base model. To balance computational efficiency with model performance, we utilize QLoRA with bfloat16 precision and a maximum sequence length of 8192 tokens. The LoRA adapter is configured with a rank of 32, an alpha of 64, and a dropout rate of 0.05, applied to the query, key, value, output, gate, up, and down projection layers. The model is trained for 2 epochs using the 8-bit paged AdamW optimizer. We use a per-device batch size of 2 with 8 gradient accumulation steps, resulting in an effective batch size of 8. For hyperparameters, we set the learning rate to 1.5×10^{-4} with a cosine learning-rate decay schedule and a warmup ratio of 0.03, and use a weight decay of 0.01. Gradients are clipped with a maximum norm of 1.0. We enable gradient checkpointing and TF32, use a random seed of 42, and evaluate the model every 50 optimizer steps. Empirically, we found the 150-step checkpoint to be a pragmatic balance that retains necessary exploration capabilities. Therefore, we adopted this specific checkpoint as our final SFT model.

C.3 Implementation Details of the Co-evolutionary Framework

Choice of the Evolution Target. We evolve PyVRP's default selective route exchange (SREX) crossover because crossover is a key component of HGS and SREX has received substantial attention from human experts. Since its introduction (Nagata & Kobayashi, 2010), SREX has been repeatedly refined (Blocho & Nalepa, 2017), and it remains in modern high-performance HGS implementations (Kool et al., 2022). Recent automatic algorithm design work independently chose SREX as its optimization target (Zhu et al., 2026). PyVRP provides SREX as a replaceable module, so all other HGS components remain fixed during evolution (Wouda & Lan, 2024). SREX therefore offers a stringent test of whether EvoHGS can still improve a mature operator after extensive expert optimization. Population management, which leaves more room for improvement, is evolved in § F.3.

Hyperparameters and Setup. The EvoHGS framework operates with a standardized set of hyperparameter configurations, which can be grouped into three core phases:

- **LLM Generation:** We sample $K = 8$ candidates per prompt using a temperature of 0.8, top-p of 0.95, and a maximum generation length of 8192 tokens.

- **Evolutionary Search:** The candidate pool maintains a target size of $L_p = 8$. The patience threshold for triggering a collapse is set to $T_p = 10$. Evolutionary mechanisms are sampled with probabilities of 0.25 (**E1**), 0.35 (**M1**), and 0.40 (**M2**). Each feasible candidate is evaluated for 1000 HGS iterations to obtain its objective performance.
- **RL Optimization:** For the reward function (Eq. 5), the scaling parameters are empirically set to $\alpha = 0.2$ and $\beta = 0.2$. Discrete penalties for invalid candidates are strictly assigned as $\psi = -1.0$ (format failure), -0.9 (compilation failure), -0.8 (runtime failure), and -0.5 (plagiarism). The policy is updated via QLoRA (bfloat16 precision) with a learning rate of 5×10^{-5} and a PPO clipping range of 0.2, implemented using the open-source `veRL` framework (Sheng et al., 2025).

AST-Based Duplicate Filtering. To prevent the solver from wasting computational resources on trivial syntactic variations, we implement a strict deduplication pipeline inspired by tree-based code clone detection (Jiang et al., 2007). First, each generated C++ file is parsed by Clang (C++20) into a JSON-formatted abstract syntax tree (AST), isolating only complete function definitions. Next, we normalize these trees to capture true semantics: we strip comments and source locations, apply deterministic α-renaming to standardize variable names, replace concrete literals with generic tokens, and resolve implicit type conversions. Finally, the normalized ASTs are serialized into canonical JSON strings. Any candidate whose structural similarity reaches a threshold of $\rho = 0.95$ with an existing pool member or the original HGS baseline is penalized or discarded.

Pool Resilience and Recovery. To prevent search stagnation, we implement a **probabilistic collapse** mechanism driven by n_{nb}, the count of rounds without a new global best. Once n_{nb} reaches a threshold T_p, a pool collapse is triggered with probability $P_{\text{collapse}} = \max(0, \min(1, \frac{n_{\text{nb}} - T_p + 1}{T_p}))$. During collapse, the pool is reset to contain only the current **global best** and the **original seed**. The system then enters a recovery phase where new feasible candidates are generated via **M2** and injected until the target size L_p is restored. Figure 3 marks the five pool resets of a representative CVRP run. Consistent with the trigger condition, all resets fall on plateaus of the best-so-far curve: none occurs during the rapid early progress up to query 149, two occur before the next improvement at query 373, and three before the one at query 884. Each later improvement follows a reset: the new best at query 373 appears shortly after the second reset, and the final improvements at queries 884 and 996 follow the last reset. Because the pool retains the global best, a reset discards no progress and only replaces stagnant members with fresh **M2** candidates. A single run cannot isolate the causal effect of resets, but this pattern indicates that they restore exploration during stagnation at no cost to the best operator found.

GRPO Advantage Normalization. During the RL phase, we stabilize the learning process to ensure the model updates consistently, regardless of the absolute scale of the routing improvements. For each rollout group of K candidates generated from the same prompt, we compute the normalized advantage A_i based on their raw rewards $\{r_i\}_{i=1}^{K}$:

$$A_i = \begin{cases} \dfrac{r_i - \bar{r}}{\sigma_r}, & \text{if } \sigma_r > 10^{-12} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

where $\bar{r}$ and σ_r denote the mean and standard deviation of the group's rewards.

Ablation setup. For the API-based ablation study, GPT-5-nano is queried via an OpenAI-compatible API using the identical sampling configuration.

C.4 Statistical Testing Protocol

This subsection introduces the statistical test used throughout the paper. Since the gaps to HGS-PyVRP are small, we test whether an observed improvement is systematic rather than an artifact of instance-level variation. Two methods are always compared on the same instances, so the comparison is paired: for each instance i, we compute the difference $d_i = \text{Obj}_{\text{comparator},i} - \text{Obj}_{\text{method},i}$, where $d_i > 0$ indicates a lower (better) objective for the method. Unless stated otherwise, we conduct a one-sided paired Wilcoxon signed-rank test on the d_i, which does not assume normally distributed

differences. The null hypothesis asserts that the distribution of d_i has no positive location shift relative to zero (i.e., the method does not systematically improve upon its comparator), while the alternative hypothesis is that the location of d_i is greater than zero. Thus, smaller p-values provide stronger evidence that the method tends to achieve lower objective values on the matched instances. We report the standardized test statistic z and the p-value; * and ** denote $p < 0.05$ and $p < 0.01$, and all p-values are reported unadjusted and interpreted on a per-comparison basis. Appendix E reports the pooled instance-level comparison on the ten unseen variants and Appendix D.1 the per-task paired comparisons on the seen CVRP and OVRP tasks; the tests embedded in Tables 8, 11, 15, and 16 follow the same protocol.

C.5 Computational Cost

Table 5 reports end-to-end design-search times on CVRP at $n = 100$. These methods have different computational cost profiles: OpenEvolve and ReEvo use hosted GPT-5-nano without task-specific training, whereas EvoHGS uses a local Qwen3-8B that additionally requires a one-time SFT of about 8 hours on one NVIDIA RTX A6000; this cost is excluded from the search time, so a lower recurring search time does not imply a lower total cost for a single campaign. The EvoHGS search is dominated by rollout generation (about 3 hours 30 minutes), followed by GRPO updates (about 2 hours) and candidate evaluation (about 50 minutes), whereas ReEvo spends about 4 hours 55 minutes on LLM requests and reflection and about 2 hours 35 minutes on candidate compilation and evaluation. After design, the selected operator is compiled into HGS, and solving new instances requires no LLM inference or model updates.

Table 5: **End-to-end operator-design cost on CVRP at $n = 100$.** Times are averaged over three independent runs of 1,000 LLM queries each.

Method	LLM	One-time SFT	Search time (hh:mm)
OpenEvolve	GPT-5-nano (hosted)	—	30:45
ReEvo	GPT-5-nano (hosted)	—	07:30
EvoHGS	Qwen3-8B (local)	$\approx$ 8 h	**06:50**

Search time is the recurring per-campaign cost and excludes the one-time SFT. Including SFT, the first EvoHGS campaign costs about 14 hours 50 minutes, while later campaigns reuse the same fine-tuned model. A dash indicates that the method requires no task-specific training. Times were measured on the hardware reported in § 5 and are not normalized across differing accelerators.

D Additional Results on Seen Variants

This appendix complements the results on the six seen variants of § 5.1. Appendix D.1 reports the run-to-run dispersion behind the main comparison together with paired significance tests, Appendix D.2 reports means, standard deviations, and statistical tests for the component ablation and compares independent generation with evolutionary search, and Appendix D.3 traces how the crossover operator changes over one representative EvoHGS run.

D.1 Full Statistics and Paired Tests for the Main Comparison

Table 6 gives the full statistics behind Table 1: for each LLM-based method and variant, the mean and sample SD over three independent evolution runs. The SREX operator from each run is evaluated with a fixed seed (seed 0) on 50 test instances per setting under a budget of 1,000 HGS iterations, the same budget as the HGS-PyVRP reference. Because the evaluation seed is fixed across runs, the SD reflects variability across independently evolved operators rather than variability across solver seeds. A zero SD indicates identical aggregate outcomes, not an absence of uncertainty. Gaps are computed from the ratio of aggregate mean objectives rather than by averaging instance-wise gaps.

Paired comparisons on seen tasks. Table 7 adds paired comparisons for the single-task SREX results on CVRP and OVRP at $n = 100$. The mean gaps and SDs are the archived run statistics of Tables 1 and 6, whereas the tests (Appendix C.4) are computed on three separately repeated runs. EvoHGS improves significantly over HGS-PyVRP on both tasks, whereas ReEvo does so on neither;

Table 6: **Full statistics of the LLM-based comparison on six VRP variants.** Objectives and gaps (%) are means over three independent evolution runs, complementing Table 1.

Variant	Method	$n = 100$		$n = 500$		$n = 1000$	
		Obj	Gap	Obj	Gap	Obj	Gap
CVRP	ReEvo	15.781 ±0.022	-0.025 ±0.141	37.800 ±0.018	0.151 ±0.047	51.327 ±0.013	**0.028** ±0.025
	OpenEvolve	15.805 ±0.039	0.130 ±0.250	37.954 ±0.261	0.559 ±0.692	51.589 ±0.434	0.538 ±0.846
	EvoTune	15.822 ±0.010	0.237 ±0.065	38.006 ±0.098	0.697 ±0.259	51.573 ±0.156	0.507 ±0.305
	EvoHGS	15.760 ±0.002	**-0.156** ±0.013	37.763 ±0.064	**0.051** ±0.169	51.346 ±0.031	0.064 ±0.059
OVRP	ReEvo	9.995 ±0.001	-0.095 ±0.012	23.246 ±0.001	0.146 ±0.003	32.916 ±0.001	0.066 ±0.003
	OpenEvolve	10.001 ±0.005	-0.028 ±0.052	23.248 ±0.009	0.152 ±0.038	32.913 ±0.015	**0.057** ±0.047
	EvoTune	10.014 ±0.012	0.101 ±0.123	23.299 ±0.105	0.372 ±0.453	33.004 ±0.130	0.334 ±0.396
	EvoHGS	9.992 ±0.003	**-0.125** ±0.027	23.212 ±0.021	**-0.001** ±0.091	32.928 ±0.021	0.101 ±0.064
VRPB	ReEvo	14.304 ±0.000	-0.071 ±0.001	33.874 ±0.018	0.123 ±0.052	47.656 ±0.049	0.051 ±0.103
	OpenEvolve	14.315 ±0.007	0.009 ±0.051	33.890 ±0.022	0.168 ±0.065	47.702 ±0.033	0.147 ±0.069
	EvoTune	14.329 ±0.010	0.103 ±0.070	33.933 ±0.096	0.296 ±0.284	47.753 ±0.069	0.254 ±0.144
	EvoHGS	14.299 ±0.001	**-0.109** ±0.009	33.814 ±0.020	**-0.055** ±0.060	47.517 ±0.019	**-0.240** ±0.040
VRPTW	ReEvo	25.150 ±0.004	-0.110 ±0.014	94.502 ±0.036	0.030 ±0.038	182.075 ±0.053	0.001 ±0.029
	OpenEvolve	25.262 ±0.158	0.335 ±0.628	95.044 ±0.858	0.603 ±0.908	182.965 ±1.164	0.490 ±0.639
	EvoTune	25.201 ±0.017	0.092 ±0.068	94.733 ±0.146	0.274 ±0.154	182.389 ±0.229	0.174 ±0.126
	EvoHGS	25.126 ±0.008	**-0.207** ±0.033	94.433 ±0.065	**-0.044** ±0.069	181.880 ±0.067	**-0.106** ±0.037
VRPL	ReEvo	15.712 ±0.000	0.107 ±0.002	37.154 ±0.004	0.293 ±0.011	50.576 ±0.005	0.259 ±0.010
	OpenEvolve	15.720 ±0.048	0.162 ±0.305	37.216 ±0.251	0.461 ±0.678	50.701 ±0.440	0.508 ±0.872
	EvoTune	15.704 ±0.013	0.055 ±0.082	37.092 ±0.029	0.127 ±0.079	50.482 ±0.002	**0.074** ±0.004
	EvoHGS	15.672 ±0.005	**-0.143** ±0.032	37.041 ±0.040	**-0.010** ±0.107	50.487 ±0.037	0.083 ±0.073
OVRPTW	ReEvo	14.069 ±0.000	**-0.163** ±0.002	48.571 ±0.009	0.102 ±0.018	84.341 ±0.013	0.032 ±0.015
	OpenEvolve	14.114 ±0.052	0.151 ±0.372	49.034 ±0.611	1.056 ±1.258	85.078 ±0.906	0.907 ±1.075
	EvoTune	14.096 ±0.007	0.022 ±0.047	48.713 ±0.081	0.395 ±0.167	84.617 ±0.080	0.360 ±0.095
	EvoHGS	14.071 ±0.004	-0.151 ±0.028	48.467 ±0.066	**-0.113** ±0.137	84.076 ±0.091	**-0.282** ±0.108

Each entry reports the mean with the sample SD in smaller type to its right; a zero SD denotes identical observed aggregate results. Gaps (%) are relative to HGS-PyVRP; lower is better. The lowest mean gap in each column is in bold, and negative gaps, i.e., improvements over HGS-PyVRP, are underlined.

against ReEvo, EvoHGS is significant on CVRP and falls marginally short of the 0.05 threshold on OVRP.

Table 7: **Instance-wise paired Wilcoxon comparisons of the single-task SREX operators on CVRP and OVRP at $n = 100$.** Each method is tested against HGS-PyVRP (Org), and EvoHGS additionally against ReEvo.

Method	Mean Gap ↓	vs. Org (z/p)	vs. ReEvo (z/p)
CVRP			
ReEvo	$\underline{-0.025}$ ± 0.141	$-0.246/0.597$	—
EvoHGS	$\underline{\mathbf{-0.156}}$ ± 0.013	$2.756/\mathbf{0.003}^{**}$	$2.360/\mathbf{0.009}^{**}$
OVRP			
ReEvo	$\underline{-0.095}$ ± 0.012	$0.977/0.164$	—
EvoHGS	$\underline{\mathbf{-0.125}}$ ± 0.027	$2.637/\mathbf{0.004}^{**}$	$1.632/0.051$

Gap (%) is relative to HGS-PyVRP (Org); the lowest mean gap per variant is in bold, and negative gaps are underlined. z/p: standardized Wilcoxon signed-rank test statistic and p-value; $^{*}p < 0.05$, $^{**}p < 0.01$. An em dash marks a non-applicable comparison.

D.2 Component Ablation and Independent Generation

Whereas Figure 2 plots the best of three runs per configuration, Table 8 reports the mean and sample SD over the same runs (the full-model runs being the three EvoHGS CVRP runs underlying Table 1), together with the paired tests of Appendix C.4 on run-averaged instance-level objectives; each configuration is tested against HGS-PyVRP and against the preceding configuration in its block. In the Qwen3-8B block, each added component lowers the mean gap, but only the improvement from adding GRPO is statistically significant relative to the preceding configuration, while the SFT step is not. In the GPT-5-nano block, embedding generation in evolutionary search lowers the mean gap relative to independent generation, yet neither configuration improves significantly over HGS-PyVRP and the increment itself is not significant, so this comparison does not establish that evolutionary selection alone accounts for the gains of the full framework. The gaps here are those recorded on the evolution instances during the search, whereas Tables 6 and 7 re-evaluate the selected operators on held-out test instances, so the full-model values differ slightly although the runs are the same. The corresponding zero-shot decomposition is reported in Appendix E (Table 11).

Table 8: **Component ablation (Qwen3-8B) and independent-generation comparison (GPT-5-nano) on CVRP.** Mean gap with the sample SD over three runs; each configuration is tested against HGS-PyVRP (Org) and against the preceding configuration in its block.

Configuration	Mean Gap ↓	vs. Org (z/p)	vs. prior configuration (z/p)
Qwen3-8B: CVRP component ablation			
Evolution	$\underline{-0.070}$ ± 0.009	$0.951/0.171$	—
SFT + evolution	$\underline{-0.103}$ ± 0.018	$1.984/\mathbf{0.024}^{*}$	$1.211/0.113$
SFT + evolution + GRPO	$\underline{\mathbf{-0.156}}$ ± 0.013	$2.785/\mathbf{0.003}^{**}$	$1.656/\mathbf{0.049}^{*}$
GPT-5-nano: independent generation vs. evolution, without SFT			
Independent generation	$\underline{-0.072}$ ± 0.020	$1.211/0.113$	—
Evolutionary search	$\underline{\mathbf{-0.099}}$ ± 0.023	$1.308/0.095$	$0.536/0.296$

Gap (%) is relative to HGS-PyVRP (Org); the lowest mean gap in each block is in bold, and negative gaps are underlined. z/p: Wilcoxon statistic and p-value; $^{*}p < 0.05$, $^{**}p < 0.01$. An em dash marks a non-applicable comparison.

D.3 Representative Operator Evolution

Table 9 and Figure 3 trace the best-so-far crossover operator of one EvoHGS CVRP run at $n = 100$, listing for each representative operator the LLM query at which it was obtained, its gap relative to HGS-PyVRP, its core design idea, and the change from its predecessor. Among the three archived

EvoHGS CVRP runs underlying Table 1, we show the one whose final operator yields the median improvement over the original HGS-PyVRP, i.e., the run whose final gap relative to the original solver ranks in the middle of the three. The example is therefore representative rather than best-case: of the other two runs, one ends with a larger improvement and one with a smaller improvement, and the EvoHGS curve in Figure 2, which plots the best of the same three runs, ends at a lower gap. The gaps are recorded on the evolution instances during the search and are therefore not directly comparable with the test-set entries of Tables 1 and 6. Across the trajectory, the surviving operators converge on three structural rules: pairing routes with compatible vehicle types, keeping paired route sizes balanced, and preserving a plausible route count in the offspring. Later operators also widen the route-pairing window and drop earlier sorting and penalty terms once these stop contributing, so the final operator is simpler than the intermediate ones. The trajectory describes a single run and does not establish that the same design sequence recurs across runs or variants.

Table 9: **Representative evolution of the crossover operator during one EvoHGS CVRP run at $n = 100$.** Each row is a best-so-far operator, with the query at which it was found, its gap (%) relative to HGS-PyVRP, and its design change.

Query	Gap (%)	Core idea	Incremental change
3	$\underline{-0.001}$	Size-driven route alignment with empty-route filtering	Introduced directional size-based alignment, route-size sorting, and empty-route filtering
106	$\underline{-0.014}$	Bounded alignment with type compatibility and angle sorting	Added a bounded search range, increased the weight of route-size matching, introduced a vehicle-type mismatch penalty, and applied angle-based sorting
136	$\underline{-0.032}$	Route-count balance with hierarchical route sorting	Added a route-count deviation penalty and prioritized vehicle type, route size, and angle in route sorting
149	$\underline{-0.061}$	Deeper compatibility search with mean-size balancing	Added a penalty for deviation from the mean paired-route size and simplified the sorting criteria
373	$\underline{-0.089}$	Dynamic-window route pairing with hard type compatibility	Introduced a dynamic pairing window, enforced strict vehicle-type compatibility, and simplified the scoring function
884	$\underline{-0.132}$	Wide fixed-window size matching with structural tie-breaking	Replaced the dynamic window with a wider fixed window, focused the score on route-size differences, and added structural tie-breaking
996	$\underline{\mathbf{-0.153}}$	Route-count-aware final offspring selection	Prioritized route-count consistency before cost comparison and used route-size variance only as a tie-breaker

Query denotes the index, among the 1000 LLM queries of the run, at which the representative operator was obtained. Gaps are relative to HGS-PyVRP; more negative values indicate larger improvements. The lowest gap is in bold, and negative gaps, i.e., improvements over HGS-PyVRP, are underlined. Entries come from a single run (the one with the median final improvement over HGS-PyVRP among the three archived runs of Table 1), so the gaps are not averaged over repetitions and carry no dispersion estimate.

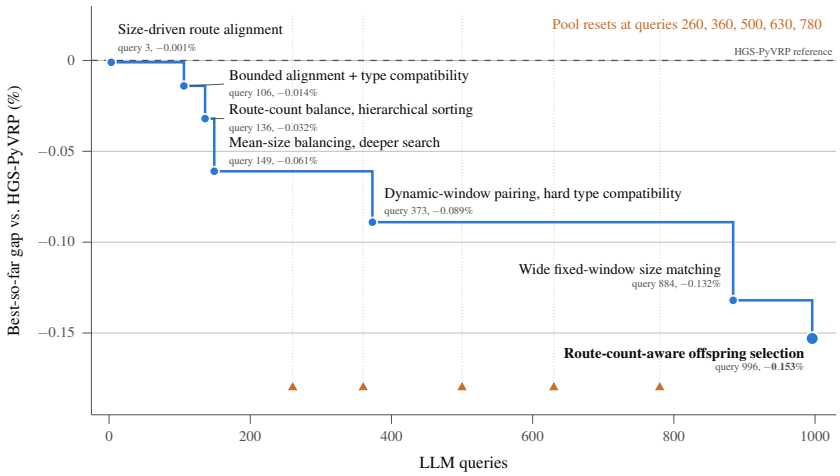

Figure 3: **Best-so-far gap of the crossover operator over the LLM queries of one EvoHGS CVRP run at $n = 100$.** Each blue marker is a representative operator of Table 9, labeled with its core idea, query index, and gap (%) relative to HGS-PyVRP (dashed line); lower is better. Orange triangles mark pool resets (Appendix C.3).

E Additional Results on Zero-Shot Generalization

This appendix complements the zero-shot results in § 5.2 by providing the full results on all ten unseen variants (Table 10), a pooled significance test over these variants, and an ablation that identifies the source of the transfer.

Table 10: **Zero-shot generalization on all ten unseen VRP variants.** Full version of Table 2. Notation follows Table 1.

Method	Variant	$n = 100$		$n = 500$		$n = 1000$		Variant	$n = 100$		$n = 500$		$n = 1000$	
		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓
HGS-PyVRP	OVRPB	10.405	*	25.797	*	36.935	*	OVRPL	9.779	*	23.289	*	32.925	*
OR-Tools#		10.528	1.182	27.946	8.330	40.665	10.099		9.993	2.188	25.768	10.645	37.483	13.844
POMO		12.571	20.814	36.001	39.555	58.927	59.542		10.220	4.512	27.360	17.481	47.531	44.361
MVMoE		12.621	21.298	41.048	59.120	104.156	181.999		10.191	4.213	31.368	34.690	83.098	152.384
ReEvo		10.406	0.010	25.874	0.299	37.087	0.411		9.787	0.077	23.337	0.206	33.061	**0.414**
EvoHGS		10.403	**-0.019**	25.799	**0.008**	36.951	**0.044**		9.776	**-0.032**	23.295	**0.025**	33.077	0.463
HGS-PyVRP	VRPBL	14.499	*	33.934	*	48.054	*	VRPBTW	27.474	*	112.945	*	191.144	*
OR-Tools#		14.899	2.759	37.575	10.730	53.957	12.284		28.644	4.259	117.039	3.625	202.231	5.800
POMO		16.986	17.153	45.824	35.039	74.296	54.610		31.587	14.971	140.662	24.540	281.385	47.211
MVMoE		16.954	16.930	48.123	41.814	108.990	126.806		31.595	15.001	142.160	25.867	303.645	58.857
ReEvo		14.503	0.026	34.085	0.445	48.262	0.433		27.498	0.089	113.855	0.806	191.803	0.345
EvoHGS		14.485	**-0.097**	33.982	**0.141**	48.059	**0.011**		27.447	**-0.097**	112.747	**-0.175**	191.039	**-0.055**
HGS-PyVRP	VRPLTW	25.076	*	92.931	*	178.989	*	OVRPBL	10.410	*	25.363	*	37.044	*
OR-Tools#		26.429	5.396	98.819	6.336	189.466	5.853		10.530	1.153	27.574	8.717	40.740	9.977
POMO		26.150	4.282	115.650	24.448	240.991	34.640		12.569	20.744	35.312	39.226	59.093	59.521
MVMoE		26.090	4.043	116.660	25.534	259.759	45.126		12.613	21.160	40.321	58.976	101.994	175.333
ReEvo		25.101	0.099	93.162	0.249	179.491	0.281		10.408	-0.022	25.451	0.348	37.175	0.355
EvoHGS		25.065	**-0.046**	92.773	**-0.170**	178.998	**0.005**		10.403	**-0.065**	25.376	**0.050**	37.053	**0.024**
HGS-PyVRP	OVRPBTW	15.760	*	54.565	*	97.607	*	OVRPLTW	14.203	*	51.067	*	88.828	*
OR-Tools#		16.101	2.164	57.959	6.220	102.087	4.590		14.669	3.281	54.278	6.288	93.516	5.278
POMO		18.197	15.462	78.912	44.620	162.067	66.040		15.089	6.239	68.663	34.456	143.316	61.341
MVMoE		18.128	15.026	85.831	57.300	208.454	113.564		15.018	5.739	74.790	46.455	185.088	108.367
ReEvo		15.768	0.052	54.722	0.287	97.769	0.166		14.206	0.020	51.255	0.368	89.092	0.297
EvoHGS		15.748	**-0.073**	54.444	**-0.222**	97.134	**-0.485**		14.191	**-0.085**	50.990	**-0.150**	88.667	**-0.181**
HGS-PyVRP	VRPBLTW	27.499	*	102.475	*	196.410	*	OVRPBLTW	15.760	*	54.231	*	97.088	*
OR-Tools#		28.802	4.738	109.914	7.259	207.133	5.459		16.158	2.525	57.554	6.127	101.766	4.818
POMO		31.587	14.867	139.492	36.122	288.241	46.755		18.197	15.462	78.738	45.190	161.601	66.447
MVMoE		31.595	14.896	140.898	37.495	310.670	58.174		18.128	15.026	85.590	57.825	207.802	114.035
ReEvo		27.509	0.035	102.835	0.351	196.790	0.193		15.760	-0.001	54.381	0.276	97.353	0.273
EvoHGS		27.450	**-0.177**	102.344	**-0.128**	196.134	**-0.140**		15.759	**-0.007**	54.087	**-0.265**	96.709	**-0.391**

Note: OR-Tools# uses a 300 s time limit.

Pooled comparison on unseen variants. We first examine whether the zero-shot improvements reported in Table 10 are systematic when all unseen settings are considered jointly. Pooling the ten unseen variants of § 5.2 at $n \in \{100, 500, 1000\}$ with 50 instances each gives 1,500 matched comparisons with HGS-PyVRP. With $\Delta_i = (\mathrm{Obj}_{\mathrm{HGS},i} - \mathrm{Obj}_{\mathrm{EvoHGS},i})/\,\mathrm{Obj}_{\mathrm{HGS},i} \times 100$ for instance i, where $\Delta_i > 0$ indicates a lower objective for EvoHGS, we apply the test of Appendix C.4 to the Δ_i. The null hypothesis asserts that the distribution of Δ_i has no positive location shift relative to zero (i.e., EvoHGS does not systematically improve upon HGS-PyVRP), while the alternative hypothesis is that the location of Δ_i is greater than zero. The test rejects the null hypothesis ($z = 2.778$, $p = 0.003$), but the median Δ_i is only 0.007%: the pooled improvement is statistically significant yet small in magnitude, and it does not establish an improvement for every variant and size. The small magnitude is expected: HGS-PyVRP is a mature, extensively tuned solver whose remaining headroom on these instances is limited, as also noted by RFTHGS (Zhu et al., 2026), so the relevant question is whether a zero-shot operator improves it consistently rather than by a large margin, and the pooled test answers this in the affirmative, distinguishing a systematic improvement from run-to-run noise.

Ablation on zero-shot transfer. Table 11 isolates the sources of zero-shot transfer on OVRPB, OVRPBL, and OVRPBLTW, which progressively add backhaul, route-length, and time-window constraints to the seen OVRP setting. Four configurations build on one another: evolutionary search with the base LLM (**w/o SFT, w/ Evo**); the same search started from the SFT checkpoint (**w/ SFT, w/ Evo**); and the full method, in which GRPO fine-tune the model on the six seen tasks either independently (**EvoHGS (single-task)**) or jointly (**EvoHGS (multi-task)**) from that checkpoint. ReEvo is included for comparison. Entries are means over three independent evolution runs; the single-task and multi-task means at $n = 100$ are those of Tables 2 and 13. Zero-shot transfer is attributable primarily to SFT: without it, neither evolution with the base LLM nor ReEvo improves significantly over HGS-PyVRP on any variant, whereas evolution from the SFT checkpoint does so on all three. GRPO then adds variant-dependent gains rather than a uniform advantage: single-task training improves further on the two harder variants, OVRPBL and OVRPBLTW, while multi-task training improves over single-task only on OVRPB. Both EvoHGS configurations outperform ReEvo on all three variants; while the difference between the multi-task configuration and ReEvo on OVRPBL is not statistically significant.

Table 11: **Zero-shot ablation and ReEvo comparison on three unseen VRP variants.** Mean gap (%) with the sample SD over three independent runs, with paired tests against HGS-PyVRP (Org), the preceding ablation configuration, and ReEvo.

Method	Mean Gap ↓	vs. Org (z/p)	vs. prior ablation (z/p)	vs. ReEvo (z/p)
		OVRPB		
ReEvo	0.010 ± 0.002	$-1.509/0.934$	—	—
w/o SFT, w/ Evo	0.005 ± 0.009	$-1.082/0.860$	—	—
w/ SFT, w/ Evo	$\underline{-0.027} \pm 0.011$	$1.981/\mathbf{0.024}^{*}$	$2.439/\mathbf{0.007}^{**}$	—
EvoHGS Single-Task	$\underline{-0.019} \pm 0.005$	$1.335/0.091$	$0.486/0.314$	$3.207/< \mathbf{0.001}^{**}$
EvoHGS Multi-Task	$\underline{\mathbf{-0.038}} \pm 0.008$	$2.282/\mathbf{0.011}^{*}$	$2.137/\mathbf{0.016}^{*}$	$3.817/< \mathbf{0.001}^{**}$
		OVRPBL		
ReEvo	$\underline{-0.022} \pm 0.004$	$1.168/0.121$	—	—
w/o SFT, w/ Evo	$\underline{-0.026} \pm 0.012$	$1.022/0.154$	—	—
w/ SFT, w/ Evo	$\underline{-0.038} \pm 0.006$	$1.667/\mathbf{0.048}^{*}$	$0.316/0.376$	—
EvoHGS Single-Task	$\underline{\mathbf{-0.065}} \pm 0.011$	$3.990/< \mathbf{0.001}^{**}$	$3.898/< \mathbf{0.001}^{**}$	$2.325/\mathbf{0.010}^{*}$
EvoHGS Multi-Task	$\underline{-0.063} \pm 0.006$	$3.475/< \mathbf{0.001}^{**}$	—	$1.343/0.090$
		OVRPBLTW		
ReEvo	$\underline{-0.0012} \pm 0.0016$	$0.235/0.407$	—	—
w/o SFT, w/ Evo	$\underline{-0.0009} \pm 0.0014$	$0.201/0.420$	—	—
w/ SFT, w/ Evo	$\underline{-0.0051} \pm 0.0005$	$1.870/\mathbf{0.031}^{*}$	$1.095/0.137$	—
EvoHGS Single-Task	$\underline{\mathbf{-0.0073}} \pm 0.0018$	$2.821/\mathbf{0.002}^{**}$	$2.585/\mathbf{0.005}^{**}$	$1.895/\mathbf{0.029}^{*}$
EvoHGS Multi-Task	$\underline{-0.0065} \pm 0.0009$	$2.132/\mathbf{0.017}^{*}$	—	$1.982/\mathbf{0.024}^{*}$

Gap (%) is relative to HGS-PyVRP (Org); the lowest mean gap per variant is in bold, and negative gaps are underlined. z/p: Wilcoxon statistic and p-value; $^{*}p < 0.05$, $^{**}p < 0.01$. "vs. prior ablation" refers to the immediately preceding configuration in the ablation sequence. An em dash marks a comparison that was not evaluated.

F Extended Experiments

This appendix extends EvoHGS beyond the single-task, synthetic, crossover-only setting of the main text along three directions. Appendix F.1 replaces the per-variant search with a multi-task search in which one shared model is fine-tuned jointly on the six seen variants, and evaluates the resulting operators on seen and unseen variants. Appendix F.2 applies the operator evolved on synthetic CVRP instances to the CVRPLIB benchmark without further adaptation and adds a comparison with RFTHGS. Appendix F.3 changes the evolution target from the selective route exchange (SREX) crossover to the population management (POP) operator. In HGS (Vidal, 2022), POP maintains separate feasible and infeasible subpopulations and ranks individuals by a biased fitness that combines solution cost with the contribution to population diversity; this fitness drives both parent selection and survivor selection, during which a subpopulation that exceeds its size limit is reduced by discarding clones and the individuals with the worst biased fitness. POP thus governs the balance between intensification and diversification of the whole search rather than the construction of a single offspring, which makes it a more tightly coupled target than SREX.

F.1 Multi-Task Evolutionary Search

In the multi-task setting, EvoHGS fine-tunes a single shared language model across the six VRP variants $\tau \in \mathcal{T}$ (CVRP, OVRP, VRPB, VRPTW, VRPL, and OVRPTW). Within each evolutionary round, candidate generation, evaluation, and pool maintenance remain independent per variant; because the candidate pools are disjoint, a collapse triggered by stagnation in one variant (e.g., CVRP) does not disturb the search in another (e.g., VRPTW). The evaluation data from all variants are then merged into a single batch for one shared GRPO update. To prevent variants with intrinsically larger reward scales from dominating this update, we rescale rewards per task while leaving the rule-based penalties untouched: invalid candidates ($c \in \mathcal{I}_\tau$), which receive the penalties $\psi(c)$ for compilation or abstract-syntax-tree errors, retain their raw negative rewards r_c^{raw} defined in § 4.2.2, whereas the rewards of valid candidates are standardized within each variant's pool $\mathcal{P}_\tau$ using z-scores and passed

through a $\tanh$ projection:

$$r_c^{\text{train}} = \begin{cases} \tanh\left(\frac{r_c^{\text{raw}} - \mu_\tau}{\sigma_\tau}\right), & c \notin \mathcal{I}_\tau \text{ and } \sigma_\tau > 10^{-12}, \\ 0, & c \notin \mathcal{I}_\tau \text{ and } \sigma_\tau \leq 10^{-12}, \\ r_c^{\text{raw}}, & c \in \mathcal{I}_\tau. \end{cases}$$

where μ_τ and σ_τ are the mean and standard deviation of the raw rewards over the valid subset of $\mathcal{P}_\tau$. The shared model thus learns equitably from valid improvements across all variants, while syntax violations retain strict, unscaled penalties.

Results on seen variants. Table 12 compares the multi-task operator with HGS-PyVRP and the single-task operators of Table 1. The shared search retains most of the benefit over HGS-PyVRP but loses the variant-specific edge: single-task attains the lower gap in most settings, including every variant at the evolution scale $n = 100$, and multi-task is lower only at some larger sizes. Appendix E provides the corresponding zero-shot ablations.

Table 12: **Multi-task EvoHGS on six seen VRP variants, compared with HGS-PyVRP and single-task EvoHGS.** Objectives and gaps (%) at $n \in \{100, 500, 1000\}$.

Variant	Method	$n = 100$		$n = 500$		$n = 1000$	
		Obj.	Gap	Obj.	Gap	Obj.	Gap
CVRP	HGS-PyVRP	15.785	$*$	37.743	$*$	51.313	$*$
	EvoHGS Single-Task	15.760	**-0.16**	37.763	**0.05**	51.346	**0.06**
	EvoHGS Multi-Task	15.761	-0.15	37.790	0.12	51.353	0.08
OVRP	HGS-PyVRP	10.004	$*$	23.212	$*$	32.895	$*$
	EvoHGS Single-Task	9.992	**-0.13**	23.212	**0.00**	32.928	0.10
	EvoHGS Multi-Task	10.003	-0.01	23.229	0.07	32.862	**-0.10**
VRPB	HGS-PyVRP	14.314	$*$	33.833	$*$	47.632	$*$
	EvoHGS Single-Task	14.299	**-0.11**	33.814	**-0.06**	47.517	**-0.24**
	EvoHGS Multi-Task	14.316	0.02	33.830	-0.01	47.633	0.00
VRPTW	HGS-PyVRP	25.178	$*$	94.474	$*$	182.073	$*$
	EvoHGS Single-Task	25.126	**-0.21**	94.433	-0.04	181.880	**-0.11**
	EvoHGS Multi-Task	25.171	-0.03	94.430	**-0.05**	181.931	-0.08
VRPL	HGS-PyVRP	15.695	$*$	37.045	$*$	50.445	$*$
	EvoHGS Single-Task	15.672	**-0.14**	37.041	**-0.01**	50.487	0.08
	EvoHGS Multi-Task	15.706	0.07	37.050	0.01	50.426	**-0.04**
OVRPTW	HGS-PyVRP	14.092	$*$	48.522	$*$	84.314	$*$
	EvoHGS Single-Task	14.071	**-0.15**	48.467	**-0.11**	84.076	**-0.28**
	EvoHGS Multi-Task	14.080	-0.09	48.578	0.12	84.221	-0.11

Evolution uses $n = 100$ instances. HGS-PyVRP and single-task EvoHGS rows repeat the Table 1 entries; $*$ marks the reference. Gaps (%) are relative to HGS-PyVRP; lower is better. The lowest gap in each column is in bold, and negative gaps, i.e., improvements over HGS-PyVRP, are underlined. All entries reported in the table are mean values over three independent runs.

Results on unseen variants. Table 13 repeats this comparison, with ReEvo, on the ten unseen variants of Table 10, whose single-task and ReEvo rows are reused. Both EvoHGS operators improve over HGS-PyVRP in most settings, and ReEvo never attains the lowest gap. The split follows the constraint type: multi-task is generally lower on the variants without time windows (OVRPB, OVRPL, VRPBL, OVRPBL), whereas single-task is lower in almost every setting with time windows. Multi-task training thus shifts transfer toward the open-route, backhaul, and route-length combinations closest to the jointly trained tasks rather than yielding a uniform zero-shot advantage.

Table 13: **Multi-task EvoHGS on ten unseen VRP variants, compared with HGS-PyVRP, ReEvo, and single-task EvoHGS.** Objectives and gaps (%) at $n \in \{100, 500, 1000\}$ under zero-shot transfer.

Method	Variant	$n = 100$		$n = 500$		$n = 1000$		Variant	$n = 100$		$n = 500$		$n = 1000$	
		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓		Obj.	Gap↓	Obj.	Gap↓	Obj.	Gap↓
HGS-PyVRP	OVRPB	10.405	∗	25.797	∗	36.935	∗	OVRPL	9.779	∗	23.289	∗	32.925	∗
ReEvo		10.406	0.010	25.874	0.299	37.087	0.411		9.787	0.077	23.337	0.206	33.061	0.414
EvoHGS Single-Task		10.403	-0.019	25.799	0.008	36.951	0.044		9.776	**-0.032**	23.295	0.025	33.077	0.463
EvoHGS Multi-Task		10.401	**-0.038**	25.787	**-0.040**	36.924	**-0.030**		9.782	0.030	23.288	**-0.008**	32.959	**0.104**
HGS-PyVRP	VRPBL	14.499	∗	33.934	∗	48.054	∗	VRPBTW	27.474	∗	112.945	∗	191.144	∗
ReEvo		14.503	0.026	34.085	0.445	48.262	0.433		27.498	0.089	113.855	0.806	191.803	0.345
EvoHGS Single-Task		14.485	-0.097	33.982	**0.141**	48.059	**0.011**		27.447	**-0.097**	112.747	**-0.175**	191.039	**-0.055**
EvoHGS Multi-Task		14.484	**-0.103**	33.993	0.174	48.166	0.233		27.483	0.032	113.104	0.141	191.092	-0.027
HGS-PyVRP	VRPLTW	25.076	∗	92.931	∗	178.989	∗	OVRPBL	10.410	∗	25.363	∗	37.044	∗
ReEvo		25.101	0.099	93.162	0.249	179.491	0.281		10.408	-0.022	25.451	0.348	37.175	0.355
EvoHGS Single-Task		25.065	**-0.046**	92.773	**-0.170**	178.998	**0.005**		10.403	**-0.065**	25.376	0.050	37.053	0.024
EvoHGS Multi-Task		25.089	0.049	92.897	-0.037	179.055	0.037		10.403	-0.063	25.367	**0.019**	37.020	**-0.063**
HGS-PyVRP	OVRPBTW	15.760	∗	54.565	∗	97.607	∗	OVRPLTW	14.203	∗	51.067	∗	88.828	∗
ReEvo		15.768	0.052	54.722	0.287	97.769	0.166		14.206	0.020	51.255	0.368	89.092	0.297
EvoHGS Single-Task		15.748	**-0.073**	54.444	**-0.222**	97.134	**-0.485**		14.191	**-0.085**	50.990	**-0.150**	88.667	**-0.181**
EvoHGS Multi-Task		15.757	-0.022	54.483	-0.150	97.423	-0.188		14.197	-0.046	50.992	-0.146	88.760	-0.076
HGS-PyVRP	VRPBLTW	27.499	∗	102.475	∗	196.410	∗	OVRPBLTW	15.760	∗	54.231	∗	97.088	∗
ReEvo		27.509	0.035	102.835	0.351	196.790	0.193		15.760	-0.001	54.381	0.276	97.353	0.273
EvoHGS Single-Task		27.450	**-0.177**	102.344	**-0.128**	196.134	**-0.140**		15.759	**-0.007**	54.087	-0.265	96.709	**-0.391**
EvoHGS Multi-Task		27.458	-0.151	102.455	-0.019	196.211	-0.101		15.759	-0.007	54.074	**-0.289**	96.957	-0.135

∗ marks the reference. Gaps (%) are relative to HGS-PyVRP; lower is better. The lowest gap in each column is in shown in bold, and negative gaps, i.e., improvements over HGS-PyVRP, are underlined. All entries reported in the table are mean values over three independent runs.

F.2 CVRPLIB Benchmark

We evaluate operators evolved on synthetic CVRP instances with $n = 100$ (§ 5) directly on the CVRPLIB X set, without further evolution or fine-tuning; gaps are computed relative to the HGS-PyVRP baseline at the same budget.

We compare RFTHGS, ReEvo, and EvoHGS at 800, 1000, and 1200 HGS iterations under two protocols. First, we retrain RFTHGS ourselves: its released VERL configuration requires eight GPUs, so we ported it to TRL/Unsloth to train Qwen3-14B on a single NVIDIA RTX A6000, matching the hardware of EvoHGS's SFT (EvoHGS's RL fine-tuning uses two such GPUs), keeping all other comparable settings unchanged. Of ten operators sampled from checkpoint 50, only three compiled; Table 14 compares them with the three evolved SREX operators of ReEvo and EvoHGS. EvoHGS-SREX attains the lowest mean gap at every budget and is never worse than RFTHGS in any size group, whereas RFTHGS improves over HGS-PyVRP only on the smallest instances. Second, to remove the dependence on our reproduction, we evaluate the crossover snippets released in the RFTHGS OpenReview supplementary material and reproduce their gaps under seed 0, obtaining results broadly in line with the original paper. ReEvo and EvoHGS are represented at each budget by the operator with the median gap among their three runs, so that no comparison is driven by a best run, and every method is tested against HGS-PyVRP with the paired test of Appendix C.4 within each size group. As Table 15 shows, no RFTHGS comparison is statistically significant. EvoHGS-POP achieves statistical significance in roughly half of the comparisons, whereas ReEvo-POP and EvoHGS-SREX do so in fewer cases. These results characterize the evaluated implementations and protocols; they do not imply that RFTHGS cannot improve HGS under other training or evaluation settings.

Runtime. In Table 15, every method stays within 2 s of HGS-PyVRP on instances with fewer than 400 customers; larger differences, of both signs, appear only on larger instances. For EvoHGS-SREX, since local search dominates the cost of an HGS iteration, offspring that preserve good route segments and assign clients more coherently may start closer to a local optimum and require fewer local-search moves; offspring that violate fewer constraints may also reduce repair and penalty computations. For EvoHGS-POP, which yields the largest speedups (up to 37.6 s), population management is dominated by pairwise diversity distances, whose number grows quadratically with the population size; a policy that discards individuals more aggressively or keeps a smaller population makes each iteration cheaper. Consistent with this, the speedups of EvoHGS-POP occur mainly on the largest instances ($n \in [800, 1000]$), where every budget shows a reduction.

Table 14: **Checkpoint-sampled RFTHGS operators versus the evolved ReEvo and EvoHGS crossover operators on the CVRPLIB X set.** Mean gap (%, lower is better) relative to HGS-PyVRP at the same budget over three operators per method, with the sample SD in smaller type; The "Mean" colomn averages across the five size groups. The lowest value in each column within a budget block is in bold, and negative gaps (improvements over HGS-PyVRP) are underlined.

Budget	Method	$[100, 200)$	$[200, 400)$	$[400, 600)$	$[600, 800)$	$[800, 1000]$	Mean
800	RFTHGS	-0.05 $\pm$0.01	0.20 $\pm$0.05	0.01 $\pm$0.01	0.13 $\pm$0.05	0.18 $\pm$0.03	0.09
	ReEvo-SREX	-0.08 $\pm$0.07	0.14 $\pm$0.01	0.03 $\pm$0.05	0.14 $\pm$0.01	-0.13 $\pm$0.12	0.02
	EvoHGS-SREX	**-0.13** $\pm$0.09	**-0.07** $\pm$0.08	**-0.08** $\pm$0.03	**0.04** $\pm$0.03	**-0.15** $\pm$0.12	**-0.08**
1000	RFTHGS	-0.05 $\pm$0.01	0.20 $\pm$0.02	0.04 $\pm$0.02	0.16 $\pm$0.04	0.19 $\pm$0.01	0.11
	ReEvo-SREX	-0.12 $\pm$0.11	0.19 $\pm$0.04	**0.03** $\pm$0.01	0.10 $\pm$0.01	**-0.14** $\pm$0.13	0.01
	EvoHGS-SREX	**-0.14** $\pm$0.09	**-0.01** $\pm$0.08	**0.03** $\pm$0.01	**0.02** $\pm$0.02	-0.13 $\pm$0.13	**-0.05**
1200	RFTHGS	-0.03 $\pm$0.03	0.20 $\pm$0.01	0.03 $\pm$0.02	0.11 $\pm$0.03	0.18 $\pm$0.00	0.10
	ReEvo-SREX	-0.07 $\pm$0.13	0.16 $\pm$0.08	**0.01** $\pm$0.05	0.09 $\pm$0.01	-0.11 $\pm$0.10	0.02
	EvoHGS-SREX	**-0.12** $\pm$0.03	**0.00** $\pm$0.08	0.03 $\pm$0.01	**0.03** $\pm$0.09	**-0.14** $\pm$0.12	**-0.04**

Budget is the number of HGS iterations. RFTHGS uses the three compilable candidates among ten sampled at checkpoint 40; ReEvo-SREX and EvoHGS-SREX use their three evolved crossover operators.

Table 15: **Paired comparisons of the released RFTHGS crossover snippets, alongside the ReEvo and EvoHGS operators, against HGS-PyVRP on the CVRPLIB X set.** Each cell reports ΔGap (%) / ΔTime (s), with the paired-test p-value in parentheses.

Budget	Method	$n \in [100, 200)$	$n \in [200, 400)$	$n \in [400, 600)$	$n \in [600, 800)$	$n \in [800, 1000]$
800	RFTHGS	-0.01 / -0.04 (0.765)	-0.09 / -1.34 (0.492)	+0.02 / +0.04 (0.629)	**-0.10** / -4.12 (0.480)	-0.06 / -1.41 (0.508)
	EvoHGS-SREX	-0.12 / +0.07 (0.089)	-0.07 / +0.39 (0.357)	-0.07 / -1.04 (0.163)	+0.03 / +0.41 (0.913)	-0.15 / -0.53 (0.104)
	ReEvo-POP	**-0.20** / -0.37 (**0.021***)	-0.17 / +0.14 (**0.030***)	**-0.26** / -1.74 (**0.003****)	**-0.10** / -4.27 (0.079)	**-0.27** / -1.61 (**0.046***)
	EvoHGS-POP	-0.15 / +0.61 (**0.036***)	**-0.22** / +1.43 (< 0.001)	-0.16 / +0.68 (**0.047***)	-0.07 / +0.82 (0.154)	-0.24 / -9.13 (**0.025***)
1000	RFTHGS	-0.03 / +0.06 (0.881)	-0.04 / -1.37 (0.994)	+0.04 / -0.50 (0.421)	-0.07 / -7.31 (0.530)	-0.04 / +5.66 (0.575)
	EvoHGS-SREX	-0.13 / +0.71 (**0.045***)	0.00 / +1.34 (0.405)	+0.02 / -5.65 (0.781)	+0.02 / -11.46 (0.779)	-0.13 / -1.74 (0.104)
	ReEvo-POP	-0.12 / -0.09 (0.086)	-0.06 / +0.27 (0.122)	**-0.28** / -0.45 (**0.002****)	-0.05 / -6.36 (0.319)	-0.13 / -11.83 (0.091)
	EvoHGS-POP	**-0.16** / +0.15 (**0.024***)	**-0.13** / +1.82 (**0.020***)	-0.12 / -0.40 (**0.046***)	**-0.09** / -5.89 (0.068)	**-0.18** / -12.70 (0.055)
1200	RFTHGS	-0.06 / -0.02 (0.469)	+0.03 / -1.34 (0.502)	+0.04 / -0.05 (0.687)	-0.05 / -7.64 (0.583)	-0.01 / +8.62 (0.878)
	EvoHGS-SREX	-0.11 / +0.16 (0.079)	0.00 / -0.59 (0.676)	+0.03 / +3.48 (0.796)	+0.02 / +8.90 (0.700)	-0.13 / -21.20 (0.081)
	ReEvo-POP	-0.11 / -0.51 (**0.026***)	**-0.11** / -1.18 (0.173)	-0.10 / -2.64 (0.176)	-0.06 / -0.58 (0.107)	-0.16 / -17.42 (0.091)
	EvoHGS-POP	**-0.13** / +0.98 (**0.043***)	-0.11 / +1.79 (0.086)	**-0.13** / -2.20 (**0.035***)	**-0.08** / +3.68 (**0.025***)	**-0.18** / -37.63 (0.065)

Budget is the number of HGS iterations; Δ is method minus HGS-PyVRP at the same budget, so negative ΔGap favors the method. ReEvo and EvoHGS entries use the operator with the median gap among their three runs; RFTHGS entries are its released snippets. SREX and POP denote the crossover and population-management targets. Bold marks the lowest ΔGap in each column of a budget block and $p < 0.05$; negative ΔGap values are underlined; $^*p < 0.05$, $^{**}p < 0.01$.

F.3 Evolution of Population Management

To assess whether EvoHGS extends beyond crossover evolution, we switch the evolution target from selective route exchange (SREX) to population management (POP), keeping all other experimental settings and HGS components fixed; ReEvo uses the same target and budget, and both methods run with `MaxIterations = 1000`. Table 16 evaluates the resulting operators on the seen CVRP and OVRP tasks and, for zero-shot transfer, applies the POP design evolved on OVRP at $n = 100$ without further adaptation to OVRPB, OVRPBL, and OVRPBLTW, which progressively add backhaul, route-length, and time-window constraints. Gaps are computed as $100(f_{\text{method}} - f_{\text{HGS}})/f_{\text{HGS}}$ and reported as the mean and sample SD across the three aggregate run gaps, and the last two columns report the paired tests described in Appendix C.4 at $n = 100$. EvoHGS-POP attains the lowest mean objective in every seen and unseen setting and improves significantly over HGS-PyVRP on all five variants. Its improvement over ReEvo-POP is also statistically significant on all variants except OVRPBL, where its lower mean gap does not establish a significant improvement.

Table 16: **Population-management (POP) evolution on seen tasks and zero-shot transfer to unseen variants.** Objectives and gaps (%) relative to HGS-PyVRP report the mean and sample SD (smaller type) over three independent runs. Here, z and p denote the standardized test statistic and p-value from the one-sided paired Wilcoxon signed-rank test ($n = 100$). Bold text indicates the lowest mean gap per column block or statistical significance ($p < 0.05$); underlined gaps indicate strictly positive improvements over HGS-PyVRP.

Variant	Method	$n = 100$		$n = 500$		$n = 1000$		vs. Org	vs. ReEvo-POP
		Obj	Gap	Obj	Gap	Obj	Gap	(z/p)	(z/p)
Seen tasks									
CVRP	HGS-PyVRP	15.785	—	37.743	—	51.313	—	—	—
	ReEvo-POP	15.769 ±0.005	-0.103 ±0.033	37.705 ±0.078	-0.101 ±0.207	51.215 ±0.080	-0.190 ±0.157	$1.453/0.073$	—
	EvoHGS-SREX	15.760 ±0.002	-0.156 ±0.013	37.763 ±0.064	0.051 ±0.169	51.346 ±0.031	0.064 ±0.059	—	—
	EvoHGS-POP	15.758 ±0.004	**-0.169** ±0.025	37.684 ±0.071	**-0.156** ±0.189	51.098 ±0.092	**-0.419** ±0.179	$2.602/\mathbf{0.005}^{**}$	$2.186/\mathbf{0.014}^{*}$
OVRP	HGS-PyVRP	10.004	—	23.212	—	32.895	—	—	—
	ReEvo-POP	9.993 ±0.007	-0.108 ±0.070	23.190 ±0.042	-0.093 ±0.179	32.820 ±0.031	-0.229 ±0.094	$1.913/\mathbf{0.028}^{*}$	—
	EvoHGS-SREX	9.992 ±0.003	-0.125 ±0.027	23.212 ±0.021	-0.001 ±0.091	32.928 ±0.021	0.101 ±0.064	—	—
	EvoHGS-POP	9.987 ±0.002	**-0.167** ±0.017	23.155 ±0.020	**-0.246** ±0.088	32.693 ±0.049	**-0.613** ±0.150	$3.414/<\mathbf{0.001}^{**}$	$2.681/\mathbf{0.004}^{**}$
Unseen variants: zero-shot transfer of the POP design evolved on OVRP									
OVRPB	HGS-PyVRP	10.405	—	25.797	—	36.935	—	—	—
	ReEvo-POP	10.412 ±0.003	0.063 ±0.032	25.786 ±0.031	-0.044 ±0.119	36.893 ±0.107	-0.113 ±0.291	$0.143/0.443$	—
	EvoHGS-POP	10.404 ±0.001	**-0.011** ±0.007	25.661 ±0.014	**-0.529** ±0.055	36.681 ±0.022	**-0.688** ±0.060	$1.760/\mathbf{0.039}^{*}$	$2.400/\mathbf{0.008}^{**}$
OVRPBL	HGS-PyVRP	10.410	—	25.363	—	37.044	—	—	—
	ReEvo-POP	10.406 ±0.005	-0.040 ±0.050	25.343 ±0.045	-0.077 ±0.179	36.975 ±0.102	-0.186 ±0.274	$1.259/0.104$	—
	EvoHGS-POP	10.403 ±0.001	**-0.073** ±0.006	25.238 ±0.022	**-0.492** ±0.087	36.738 ±0.037	**-0.827** ±0.101	$1.763/\mathbf{0.039}^{*}$	$1.134/0.128$
OVRPBLTW	HGS-PyVRP	15.760	—	54.231	—	97.088	—	—	—
	ReEvo-POP	15.764 ±0.005	0.027 ±0.034	54.160 ±0.188	-0.132 ±0.346	97.046 ±0.312	-0.043 ±0.321	$-0.525/0.700$	—
	EvoHGS-POP	15.758 ±0.002	**-0.013** ±0.014	53.897 ±0.087	**-0.616** ±0.160	96.302 ±0.271	**-0.809** ±0.279	$1.844/\mathbf{0.033}^{*}$	$2.726/\mathbf{0.003}^{**}$

z/p: Wilcoxon statistic and p-value; $^{*}p < 0.05$, $^{**}p < 0.01$. EvoHGS-SREX rows repeat the archived single-task means of Table 1 (SDs from Table 6) and are not tested. An em dash marks a non-applicable comparison.

G Broader Impacts

EvoHGS demonstrates how LLM-guided evolutionary search can systematically improve heuristic optimization algorithms, with potential positive impacts in logistics efficiency and transportation. In addition, it points toward a broader paradigm where LLM-generated components may be iteratively refined through feedback-driven evolutionary processes, while also raising considerations around the computational cost of co-evolutionary training and the need for auditability of LLM-generated operators.

H Licenses of External Assets

The licenses and URLs of all external codebases, models, and software used in this work are summarized in Table 17. We acknowledge the original authors of these resources and ensure that their licenses and terms of use are properly respected.

Table 17: **Summary of licenses for external assets used in this work.**

Resource	Type	License	URL
Qwen3-8B	Model	Apache-2.0 License	https://huggingface.co/Qwen/Qwen3-8B
Qwen3-14B-Instruct	Model	Apache-2.0 License	https://huggingface.co/OpenPipe/Qwen3-14B-Instruct
verl	Code	Apache-2.0 License	https://github.com/verl-project/verl
PyVRP	Code	MIT License	https://github.com/PyVRP/PyVRP/releases/tag/v0.12.2
OR-Tools	Code	Apache-2.0 License	https://developers.google.com/optimization
POMO	Code	Available online	https://github.com/yd-kwon/POMO/tree/master
MvMOE	Code	Available online	https://github.com/RoyalSkye/Routing-MVMoE
RouteFinder	Code	MIT License	https://github.com/ai4co/routefinder
ReEvo	Code	MIT License	https://github.com/ai4co/reevo
EvoTune	Code	MIT License	https://github.com/CLAIRE-Labo/EvoTune
OpenEvolve	Code	Apache-2.0 License	https://github.com/codelion/open-evolve