MEEVO: METACOGNITIVE EVOLUTION COMBINED WITH NATURAL EVOLUTION FOR AUTOMATIC HEURISTIC DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) have advanced Automatic Heuristic Design (AHD) by enabling heuristic generation through reasoning and code synthesis. In LLM-based AHD, the LLM reasons about algorithm design and generates executable heuristic code. Existing architectures adopt two main paradigms: Natural Evolution applies crossover and mutation to this code to explore diverse strategies, but discards the reasoning traces behind the design decisions, weakening knowledge inheritance; Metacognitive Evolution retains these reasoning traces and refines them through reflection, but lacks population-level recombination, limiting exploration. These limitations reduce search efficiency, stability, and solution quality on complex problems. To address this gap, we propose MeEvo, an AHD framework that cyclically couples Natural Evolution and Metacognitive Evolution with operator balance that shifts from exploration to exploitation. Natural Evolution explores heuristic code while recording LLM-generated reasoning traces, fitness values, errors and best heuristic into a shared history; Metacognitive Evolution then reflects on this history to generate improved heuristics that feed into the next Natural Evolution cycle. This design enables population-driven exploration and reflection-driven refinement to reinforce each other. Experiments on five optimization problems show that MeEvo achieves stronger performance and lower variance than tested LLM-based AHD architectures, especially on complex constrained tasks.

Code and experiment results are available at https://anonymous.4open.science/r/MeEvo-9835.

Keywords: Automatic Heuristic Design, Large Language Model, Natural Evolution, Metacognitive Evolution, Evolutionary Computation

1 INTRODUCTION

Many real-world problems, from logistics and resource allocation to network design, require optimization heuristics: rules of thumb that find good solutions efficiently (Kim & Yum, 1993). Designing such heuristics demands extensive expertise and is inherently problem-specific; the No Free Lunch theorem (Wolpert & Macready, 1997) further guarantees that no single heuristic works best for all problems, requiring repeated manual effort for each new task. Automatic Heuristic Design (AHD) (Sabar et al., 2014) aims to automate this process, with classical approaches ranging from Genetic Programming (Espejo et al., 2009) to reinforcement learning (Yi et al., 2022); however, these remain constrained by human-defined search spaces (Liu et al., 2024).

Large Language Models (LLMs) (Zhao et al., 2026) have transformed AHD by combining evolutionary computation with neural code generation (Zhang et al., 2024). LLM-generated heuristics inherently comprise two coupled components: *reasoning traces*, the LLM's explanations of its design choices that record design knowledge, and executable heuristic code subject to fitness evaluation. Existing LLM-based AHD architectures pursue two main paradigms, each with distinct strengths. **Natural Evolution (NE)** architectures (FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), MCTS-AHD (Zheng et al., 2025)) apply population-driven crossover and mutation to explore the heuristic code space broadly, generating diverse algorithmic strategies

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.

through recombination and variation. Some generate reasoning traces alongside code (EoH) or employ auxiliary reflection to guide search (ReEvo), but they do not consolidate accumulated reasoning traces and performance outcomes through a separate history-level reflection process. **Metacognitive Evolution (ME)** (MeLA (Qiu et al., 2026)) adopts a complementary strategy: it analyzes records of heuristic design, execution errors, and fitness outcomes to identify effective reasoning patterns, diagnose failures, and guide subsequent heuristic generation. Through iterative reflection, the LLM progressively refines its design logic across generations, but without population-driven crossover and mutation it cannot discover fundamentally different algorithmic strategies outside its accumulated reasoning trajectory. The two paradigms therefore have complementary limitations: Natural Evolution provides population-level exploration of heuristic code but lacks systematic history-level consolidation, whereas Metacognitive Evolution uses accumulated search records for directed refinement but lacks the code diversity provided by crossover and mutation. Neither paradigm alone exploits both forms of search.

This paper introduces **MeEvo**, a framework that combines Natural Evolution and Metacognitive Evolution through three contributions:

(1) Cyclic alternation of Natural Evolution and Metacognitive Evolution. We alternate Natural Evolution and Metacognitive Evolution cyclically, enabling them to communicate through shared history (detailed in §3.1) rather than co-evolving in parallel as in prior architectures.

(2) Hierarchical operator balance without external coordination. We achieve exploration-exploitation balance at two levels without external coordination: the alternation rhythm between the two processes governs macro-level balance, while within Natural Evolution, the crossover-to-mutation ratio adapts autonomously from exploration-dominated to exploitation-focused as the search progresses.

(3) Comprehensive experimental validation across diverse optimization problems. We validate MeEvo on five problems spanning classical benchmarks, black-box optimization, and real-world constrained tasks, with ablation studies, parameter sensitivity analysis, and statistical significance testing demonstrating the framework's effectiveness and generalizability.

Fig. 1 illustrates this architecture.

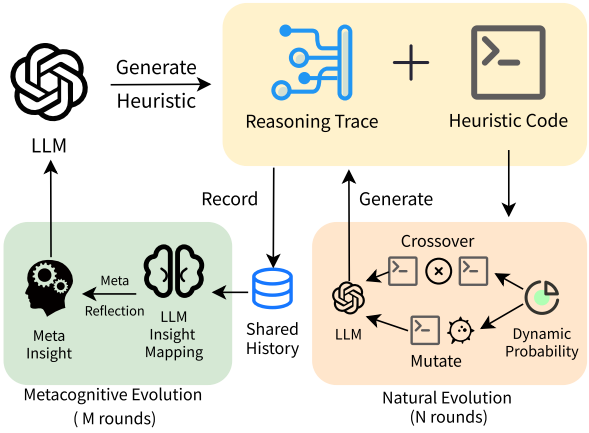


Figure 1. Architecture of MeEvo combining Natural Evolution and Metacognitive Evolution. Reasoning traces and heuristic code are recorded in shared history, through which the two phases alternate and cooperate.

2 Literature Review

Detailed literature taxonomy and paradigm comparison are provided in **Appendix E.1–E.3**.

The literature reveals two interrelated gaps. **Gap 1: Population-driven code search and history-level metacognitive reflection have not been systematically combined as alternating phases.** EC-based architectures—FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024) and MCTS-AHD (Zheng et al., 2025)—apply population-driven selection,

crossover or mutation to explore code space, whereas MeLA (Qiu et al., 2026) analyzes accumulated design and evaluation records through metacognitive reflection but lacks population-driven diversity. Recent hybrid architectures, including EoH-MR (Qi et al., 2025) and PyVRP$^+$ (Malik et al., 2026), incorporate reflection as an auxiliary module for code generation; within the scope of the architectures reviewed here, Natural Evolution and Metacognitive Evolution have not been treated as distinct, cyclically coupled phases. **Gap 2: No architecture achieves exploration–exploitation balance through structural phase alternation.** Existing mechanisms—the Navigator rule engine in HiFo-Prompt (Chen et al., 2025), policy sampling and island migration in EvoPH (Liu et al., 2025), and meta-cognitive monitoring in EoH-MR (Qi et al., 2025)—coordinate search through auxiliary mechanisms; whether the alternation rhythm itself can govern this balance remains unexplored.

MeEvo addresses these two gaps through the design presented next.

3 METHODOLOGY

3.1 SYSTEM OVERVIEW

MeEvo cyclically couples two complementary AHD paradigms: population-based **Natural Evolution** and history-conditioned **Metacognitive Evolution**. The latter iteratively reflects on accumulated reasoning and evaluation history; it is not a conventional evolutionary process with its own population, crossover, or mutation. The two phases operate as follows:

Natural Evolution. The LLM applies population-driven crossover and mutation to heuristic code. Each variant includes a *reasoning trace* explaining its design choices, which is recorded for later metacognitive analysis.

Metacognitive Evolution. The LLM analyzes *shared history*—reasoning traces, fitness values, execution errors, and the best heuristic—to determine whether search is improving or stagnating, identify reasoning patterns and algorithmic components linked to strong performance, and diagnose repeated failures. It summarizes these findings as *Meta Insights*—concise guidance on what to retain, change, or try next—which are then fed to LLM to translated into code that seeds the next Natural Evolution cycle.

MeEvo alternates the phases rather than running them in parallel. Natural Evolution requires a stable parent pool for reliable crossover, whereas Metacognitive Evolution benefits from history accumulated over several generations. MeEvo therefore conducts macro-level exploration through Natural Evolution for N generations, then applies macro-level exploitation through Metacognitive Evolution for M generations whose heuristics re-enter the next cycle.

3.2 ITERATION STEPS

The inputs are problem P, population sizes N_0 and N_{pop}, evaluation budget L, correction limit COR, parent pool size K, and cycle counts N and M. Each heuristic is executed on the target instances to compute its objective (lower is better).

Algorithm 1 formalizes the loop. Automated Problem Analysis (APA) guides initial generation, after which Natural Evolution selects the top-K feasible heuristics, applies balanced LLM crossover or mutation (Eq. 1), and records traces. Metacognitive Evolution then reflects on shared history and maps each Meta Insight to N_{pop} heuristics. Evaluation and error correction are detailed in **Appendix B.2**; only execution failures receive $f(h) = \infty$.

See **Appendix A.1** for notation, **Appendix B.1**–**Appendix B.3** for auxiliary procedures, and **Appendix C.1** for representative execution errors.

3.3 NATURAL EVOLUTION: POPULATION-DRIVEN CROSSOVER AND MUTATION

Operator Balance. The crossover and mutation probabilities shift linearly with search progress:

$$p_c(\text{Eval}) = \alpha + \beta \times \left(1 - \frac{\text{Eval}}{L}\right) \tag{1}$$

Algorithm 1 Main Framework of Cyclic Alternation

Input: P, N_0, N_{pop}, L, COR, N, M, K
Output: Best heuristic h^*, fitness $f(h^*)$
$Eval \leftarrow 0$, $H \leftarrow \emptyset$, $F \leftarrow \emptyset$, $TH \leftarrow \emptyset$, $ERR \leftarrow \emptyset$, $f(h^*) \leftarrow \infty$
$APA \leftarrow$ ANALYZE(P)
Generate N_0 initial heuristics via INITHEURISTIC; evaluate, correct errors, and update $H, F, TH, ERR, h^*, Eval$
while $Eval < L$ **do** ▷ Budget-limited cyclic loop
    // — Natural Evolution —
    for $i \leftarrow 1$ **to** N **do**
        $\mathcal{P} \leftarrow$ top-K feasible heuristics from H sorted by $f(h)$; compute p_c, p_m via Eq. 1
        Generate N_{pop} offspring via LLMCROSSOVER/LLMMUTATE; evaluate and correct errors; update $H, F, TH, ERR, h^*, Eval$
    end for
    if $Eval \geq L$ **then break**
    // — Metacognitive Reflection and Generation —
    $Meta \leftarrow$ METACOGNITIVEREFLECTION(TH, F, ERR, h^*)
    for $j \leftarrow 1$ **to** M **do**
        Generate N_{pop} heuristics from $Meta$; evaluate, correct errors, and update $H, F, TH, ERR, h^*, Eval$
    end for
end while
return $h^*, f(h^*)$

$$p_m(\text{Eval}) = 1 - p_c(\text{Eval}) \tag{2}$$

Here, $Eval$ and L are the current and maximum evaluation counts, α is the terminal crossover probability, and β is the adaptation amplitude, with $\alpha+\beta \leq 1$. Since $p_c + p_m = 1$, each offspring uses one operator. For example, With α=0.5 and β=0.2, p_c decreases from 0.7 to 0.5, adapting the internal operator mixture as the search progresses. This schedule does not identify crossover exclusively with exploration or mutation exclusively with exploitation; it adjusts the relative use of two complementary variation mechanisms. Operator details and parameter sensitivity are reported in **Appendix F.1** and **Appendix G.2**.

3.4 METACOGNITIVE EVOLUTION PHASE: REFLECTION-GUIDED GENERATION

Natural Evolution provides macro-level exploration through stochastic code variation, whereas Metacognitive Evolution provides macro-level exploitation through *directed reflection* on shared history. It analyzes shared history to diagnose the search state, preserve effective reasoning and code components, and identify concrete changes, summarizing them as a Meta Insight for the next generation.

This phase has no separate metacognitive population or fitness function. Each round feeds the Meta Insight to LLM, which generates N_{pop} candidates and reasoning traces evaluated with the same task objective $f(h)$. Feasible candidates and traces enter shared history and the candidate pool; subsequent Natural Evolution top-K selection and global-best updates determine retention. Thus, "evolution" denotes history-conditioned improvement in a distinct phase, not a second population-based evolutionary algorithm.

Two-Step Reflection-to-Code Pipeline. The process occurs in two steps. First, the metacognitive LLM distills the shared history (reasoning history TH, fitness history F, error records ERR, and current best h^*) into a Meta Insight:

$$\text{Meta} = \mathcal{R}(TH, F, ERR, h^*) \tag{3}$$

Here, $\mathcal{R}$ produces a structured report containing *Convergence Diagnosis*, *Limitation Analysis*, and *Pathway Recommendation*, which respectively characterize search state, ineffective strategies, and specific improvements.

Second, the insight mapping LLM translates the Meta Insight into executable code with a reasoning trace:

$$h_{t+1} = \mathcal{G}(\text{Meta}, APA), \quad Th_{t+1} = \mathcal{T}(\text{Meta}) \tag{4}$$

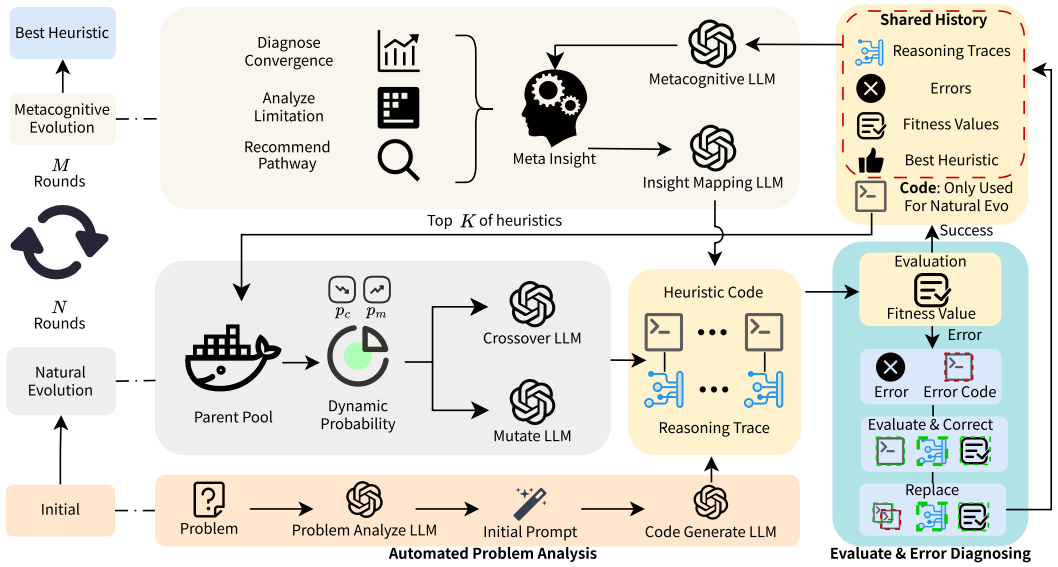


Figure 2. Overall procedure of MeEvo. The Natural Evolution search phase and Metacognitive Evolution reflection-and-generation phase alternate cyclically, coupled through shared history.

Here, $\mathcal{G}$ generates code and $\mathcal{T}$ its reasoning trace. Meta Insight specifies *what* should change and *why*, while mapping determines *how* to implement it. This separation grounds reflection in accumulated history and provides an interpretable signal for generation. Mapping preserves strong components of the best heuristic while redesigning identified weaknesses. Reasoning history remains immutable and supplies forward-looking guidance without modifying recorded traces. Further details are provided in **Appendix F**.

3.5 Inherited Components and Prompt Design

MeEvo extends three design principles from MeLA (Qiu et al., 2026): Automated Problem Analysis (APA), Error Diagnosis (ED), and metacognitive reflection through structured Meta Insights. All LLM calls use defined roles and output formats; complete prompts are in **Appendix C.2** and **Appendix C.3**. **Fig. 2** shows the overall procedure.

4 Experiments

4.1 Experimental Setup

We evaluate MeEvo on five optimization problems, categorized by evaluation protocol: two downstream-solver benchmarks (TSP-ACO, BPP-ACO), two real-world constrained problems (ACS, WSN), and one constructive-heuristic benchmark (TSP-Construct), using two LLM backbones, DeepSeek-V4-Flash and MIMO-v2.5-Pro, selected for their strong open-source performance, API availability, and low inference cost. Related settings are provided in **Appendix G.1**. The problems are:

TSP with ACO framework (TSP-ACO). The LLM generates an $(n \times n)$ heuristic matrix that guides an Ant Colony Optimization process (Ye et al., 2024). The matrix is clipped to $[10^{-9}, 10^{6}]$ and normalized before use by ACO (30 ants, 100 iterations, ρ=0.9, α_p=1, β_h=1). We optimize on n=50 with 5 instances.

Bin Packing with ACO framework (BPP-ACO). Given n items with demands $d_i \sim$ Uniform$[20, 100]$ and bin capacity C=150, the goal is to minimize the number of bins B subject to $\sum_{i \in B_j} d_i \leq C$ for all j (Ross et al., 2000). The LLM generates an $(n \times n)$ heuristic matrix from the demand vector. ACO uses 20 ants and 15 iterations, with ρ=0.9, α_p=1, and β_h=1; optimization uses n=500 and 5 instances.

Adaptive Curriculum Sequencing (ACS). This real-world e-learning problem generates personalized learning paths balancing learner characteristics and pedagogical constraints. Using the OULAD dataset (Martins et al., 2021) with 120 materials, 20 concepts, and 30 students, the decision variable is $X \in \{0,1\}^{30\times 120}$. The objective minimizes concept coverage gaps, time-bound violations, and preference mismatches:

$$f(X) = \varepsilon_1\omega_1 \sum_i \max(0, \text{covered}_i - |\mathbf{g}_i|) \quad + \varepsilon_2\omega_2 \sum_i (1 - \text{coverage}_i)$$
$$+\varepsilon_3 \sum_i \mathbf{1}\{\text{time}_i \notin [t_i^{\min}, t_i^{\max}]\} + \varepsilon_4 \sum_i \text{pref_mismatch}_i. \tag{5}$$

Here ε_1=1, ε_2=10^4, ε_3=1000, ω_1=0.25, ω_2=1, and ε_4=0.25; $[t_i^{\min}, t_i^{\max}]$ is the student-specific attention span range, and ψ_1=3, ψ_2=6, and ψ_3=1 are material priority limits. The guiding radius rg decays from 2.5 to 0.1. Evaluation uses 20 search agents and 50 iterations.

Wireless Sensor Network (WSN) Deployment. We optimize placement and power of 50 Convergence Nodes (CNs) to cover 200 fixed Sensing Nodes (SNs) while minimizing energy. Each CN j has position $(x_j, y_j) \in [0, 50]^2$ and power $P_j \in [0, 30]$ dBm. Path loss is $PL_{ij} = 55.0 + 10 \times 2.5\log_{10}(d_{ij}) + \beta_{ij}$, where $\beta_{ij} = \beta_{ij}^x + \beta_{ij}^y$ is position-dependent fading: β_{ij}^x=8.0 (resp. β_{ij}^y=20.0) when SN i and CN j lie on opposite sides of the x=25 (resp. y=25) midline, and 0 otherwise. Constraints require received signal ≥ -85 dBm, CN connectivity within 20.0, and capacity ≤ 15 SNs per CN. The objective is:

$$f(\mathbf{x}) = \sum_j 10^{P_j/10} + 10 \times \text{uncovered_SNs}$$
$$+ 1000 \times \text{disconnected} + 100 \times \max(0, \text{std}(P) - 1). \tag{6}$$

The guiding radius rg decays from 2.5 to 0.1; evaluation uses 50 search agents and 100 iterations.

Traveling Salesperson Problem, Constructive (TSP-Construct). Given n city coordinates $\{c_1, \ldots, c_n\} \subset \mathbb{R}^2$ normalized to $[0, 1]$, we seek the shortest tour visiting all cities exactly once:

$$\min \sum_{i=1}^{n-1} d(\pi_i, \pi_{i+1}) + d(\pi_n, \pi_1), \tag{7}$$

where π is a permutation and $d(\cdot,\cdot)$ is Euclidean distance. The LLM generates a *constructive heuristic*, called iteratively to select the next city. A neighborhood matrix $N \in \mathbb{Z}^{n\times n}$ lists nodes sorted by distance from each node, and candidates at step t are

$$\tilde{U}_t = \{j \in N_{\pi_t} : j \notin \{\pi_1, \ldots, \pi_t\}\}. \tag{8}$$

Training uses 64 instances with n=50 and evaluation budget L=100; evaluation uses $n \in \{50, 100, 200\}$ with 64 instances each.

Each problem is independently run **5** times. For TSP-Construct, we report the mean (standard deviations are small across architectures). For the remaining problems, we report mean, standard deviation, and best value following the protocol of prior work (Ye et al., 2024). All problems are minimization tasks (lower is better).

TSP-ACO, BPP-ACO, ACS, and WSN (**TABLE 1**) follow an indirect protocol: the LLM generates heuristic information that guides a downstream solver (ACO) or directly optimizes solution components. We compare against ACO (Dorigo et al., 2006), Genetic Algorithm (GA) (Holland, 1992), Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995), and Sand Cat Swarm Optimization (SCSO) (Seyyedabbasi & Kiani, 2023) as traditional baselines, plus LLM architectures. For ACS and WSN, the traditional baselines are evaluated under fixed budgets adopted from prior metaheuristic studies on these problems (Martins et al., 2021; Machado et al., 2021; Chen et al., 2024): 30 agents and 500 iterations for ACS, and 100 agents and 1000 iterations for WSN. These budgets substantially exceed the LLM-based AHD protocol (ACS: 20 agents, 50 iterations; WSN: 50 agents, 100 iterations). Each traditional baseline is independently executed 5 times. For every run, we retain the final best-so-far objective at budget exhaustion, and report its mean and standard deviation across the five runs. This protocol compares terminal solution quality under the specified fixed budgets.

Table 1. Comparative Performance Analysis on TSP-ACO, BPP-ACO and real-world problems. For ACS and WSN, traditional baselines are evaluated with substantially larger fixed search budgets (ACS: 30 search agents, 500 iterations; WSN: 100 search agents, 1000 iterations). Values report the mean and standard deviation of the final best-so-far objective over 5 independent runs.

Method	TSP-ACO		BPP-ACO		ACS		WSN	
	Obj (↓)	Gap (%)	Obj (↓)	Gap (%)	Obj (↓)	Gap (%)	Obj (↓)	Gap (%)
ACO	5.98 ± 0.19	4.62	208.54 ± 1.37	2.09	1793.72 ± 108.32	210.25	1294.70 ± 80.32	2448.62
GA	-	-	-	-	2634.25 ± 218.92	355.63	1922.46 ± 0.13	3684.37
PSO	-	-	-	-	2662.31 ± 80.34	360.48	1800.28 ± 286.34	3443.86
SCSO	-	-	-	-	1673.53 ± 112.04	189.46	417.02 ± 9.88	720.91
LLM backbone: *DeepSeek-V4-Flash*								
Funsearch	5.923 ± 0.02	3.62	206.60 ± 0.15	1.14	1242.75 ± 639.42	114.94	96.38 ± 27.32	89.72
EoH	5.852 ± 0.01	2.38	205.48 ± 0.75	0.59	1253.97 ± 462.30	116.89	119.71 ± 15.77	135.65
ReEvo	5.833 ± 0.01	2.05	205.48 ± 0.97	0.59	654.95 ± 22.85	13.28	130.06 ± 24.47	156.02
MCTS-AHD	5.812 ± 0.02	1.68	205.28 ± 0.72	0.49	1213.48 ± 434.93	109.89	83.36 ± 6.83	64.09
MeLA	5.828 ± 0.01	1.96	204.80 ± 0.28	0.25	588.50 ± 9.07	1.79	58.40 ± 2.02	14.96
MeEvo	$\mathbf{5.716 \pm 0.00}$	**0.00**	$\mathbf{204.28 \pm 0.10}$	**0.00**	$\mathbf{578.16 \pm 5.21}$	**0.00**	$\mathbf{50.80 \pm 0.76}$	**0.00**
LLM backbone: *MIMO-v2.5-Pro*								
Funsearch	6.013 ± 0.12	5.20	207.20 ± 1.33	1.43	687.24 ± 34.78	18.87	99.23 ± 45.71	95.33
EoH	5.847 ± 0.03	2.29	204.88 ± 0.43	0.29	1273.18 ± 525.10	120.21	104.89 ± 10.43	106.48
ReEvo	5.838 ± 0.02	2.13	204.68 ± 0.52	0.20	1239.17 ± 592.81	114.33	125.94 ± 31.69	147.91
MCTS-AHD	5.807 ± 0.01	1.59	204.44 ± 0.04	0.08	1392.67 ± 519.74	140.88	92.82 ± 5.34	82.72
MeLA	5.824 ± 0.01	1.89	204.60 ± 0.20	0.16	609.47 ± 15.21	5.42	59.37 ± 3.16	16.87
MeEvo	$\mathbf{5.718 \pm 0.01}$	**0.04**	$\mathbf{204.40 \pm 0.24}$	**0.06**	$\mathbf{585.94 \pm 11.14}$	**1.35**	$\mathbf{52.00 \pm 2.26}$	**2.36**

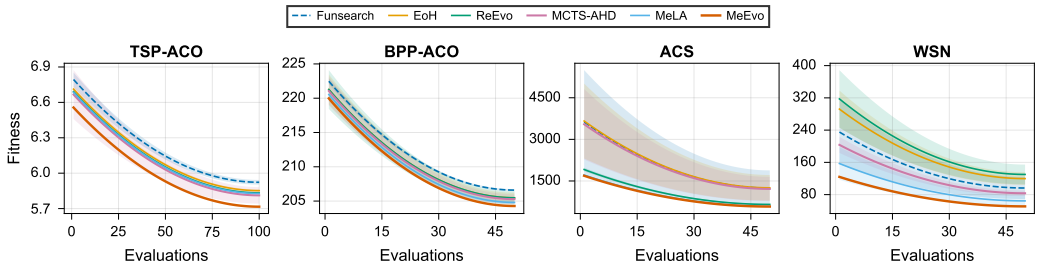


Figure 3. Convergence curves. On TSP-ACO and BPP-ACO, all architectures converge to similar values. On ACS and WSN, MeEvo achieves lower final objectives.

TSP-Construct evaluates constructive heuristics. Optimal tours are from the LKH solver (Lin & Kernighan, 1973; Zheng et al., 2025). We compare against Nearest-greedy (Rosenkrantz et al., 1977), POMO (Kwon et al., 2020), Genetic Programming(GP) (Koza, 1994), and LLM architectures (FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), MCTS-AHD (Zheng et al., 2025), HiFo-Prompt (Chen et al., 2025), MeLA (Qiu et al., 2026)). Gap is relative to LKH optimal.

Parameters are detailed in **Appendix G.1**, and the rationale for α, β, and K is provided in **Appendix G.2**. N and M respectively define the number of Natural Evolution generations and reflection-guided generation rounds per cycle, and are analyzed in the ablation study (§4.3).

4.2 Results and Analysis

TSP-ACO, BPP-ACO, ACS, and WSN. TABLE 1 presents results on four problems. MeEvo achieves the best mean performance across all problems with both LLM backbones, with gaps to the next best architecture below 1.68% on TSP-ACO and 0.25% on BPP-ACO. Despite larger computational budgets, traditional baselines struggle: even SCSO reaches gaps of 189.46% (ACS) and 720.91% (WSN). Among LLM-based architectures, MeEvo outperforms MeLA on both backbones (e.g., MeLA gaps of 1.79% on ACS and 14.96% on WSN with DeepSeek-V4-Flash) and shows superior stability ($\pm$5.21 on ACS, $\pm$0.76 on WSN). Best heuristic codes are provided in **Appendix D.1**.

Table 2. Comparative Performance Analysis on TSP-Construct Across Different Problem Sizes

Method	$n = 50$		$n = 100$		$n = 200$	
	Obj (↓)	Gap (%)	Obj (↓)	Gap (%)	Obj (↓)	Gap (%)
Optimal	5.678	-	7.767	-	10.658	-
Greedy Construct	7.003	23.34	9.542	22.85	13.094	22.86
POMO	**5.696**	**0.32**	**8.002**	**3.03**	12.875	20.80
GP	6.312	11.17	8.748	12.63	**12.134**	**13.85**
LLM backbone: *DeepSeek-V4-Flash*						
Funsearch	6.803	19.81	9.182	18.22	12.850	20.57
EoH	6.485	14.21	8.894	14.51	12.582	18.05
ReEvo	6.622	16.63	8.893	14.50	12.362	15.99
MCTS-AHD	6.403	12.77	8.762	12.81	12.271	15.13
HiFo-Prompt	6.392	12.57	8.749	12.64	12.264	15.07
MeLA	6.423	13.12	8.801	13.31	12.338	15.76
MeEvo(ours)	6.334	11.55	8.687	11.84	12.213	14.59
LLM backbone: *MIMO-v2.5-Pro*						
Funsearch	6.927	22.00	9.034	16.31	12.737	19.51
EoH	6.473	14.00	8.832	13.71	12.682	18.99
ReEvo	6.732	18.56	8.816	13.51	12.536	17.62
MCTS-AHD	6.403	12.77	8.697	11.97	12.290	15.31
HiFo-Prompt	6.392	12.57	8.693	11.92	12.285	15.27
MeLA	6.423	13.12	8.795	13.24	12.413	16.47
MeEvo(ours)	6.343	11.71	8.753	12.69	12.209	14.55

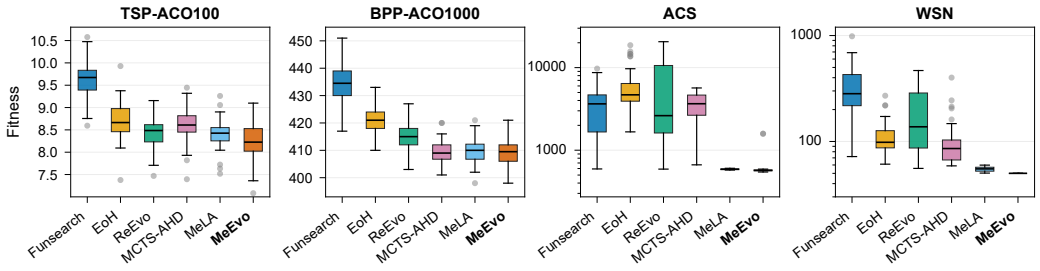


Figure 4. Performance of Optimal Heuristic Generated by Different Architectures.

FIG. 3 shows convergence behavior: MeEvo reaches the lowest final objectives on ACS and WSN, while methods are similar on TSP-ACO and BPP-ACO.

TSP-Construct. TABLE 2 compares constructive heuristics across three problem sizes. POMO achieves near-optimal performance at n=50 (Gap 0.32%) but drops to 20.80% at n=200. GP achieves Gaps of 11.17%, 12.63%, and 13.85%, surpassing all LLM-based AHD architectures at n=50 and n=200, with stronger generalization than POMO, but incurs approximately 2 million TSP evaluations during evolution (μ=1000 × G=200 × 10 training instances). LLM-based methods lag behind GP and POMO on this task because the LLM must directly generate a complete constructive procedure without a downstream solver. MeEvo remains competitive with LLM-based methods across sizes and generalizes reliably to larger instances, unlike deep RL methods that collapse under distribution shift.

FIG. 4 tests the best heuristic from each architecture (see **Appendix G.1**): for TSP-ACO (trained at n=50) and BPP-ACO (trained at n=500), it is evaluated on 64 instances at n=100 and n=1000; for ACS and WSN, it is independently executed 30 times. In TSP-ACO, the best heuristic generated by MeEvo exhibits the lowest median value. In ACS and WSN, the MeEvo heuristic shows stable performance with minimal deviation and repeatedly reaches its optimal fitness, supporting generalization and reproducibility.

4.3 Ablation Analysis

Using DeepSeek-V4-Flash (5 runs), we compare Natural Evolution only $(N, /)$, reflection-guided generation only $(/, M)$, fine-grained interleaving $(1, 1)$, and reflection-heavy coupling $(1, 2)$ against the default $(2, 1)$ (**TABLE 3**).

Single-phase baselines. $(N, /)$ collapses on ACS (1293.54) and WSN (98.19), whereas $(/, M)$ reaches 588.50 and 58.40; the latter is nevertheless outperformed by every coupled configuration.

Coupled phases. Both coupled configurations lower the objective relative to either single-phase baseline on all four tasks. Between them, $(1, 1)$ gives lower values on TSP-ACO, BPP-ACO and WSN, whereas $(1, 2)$ gives a lower ACS value. The default $(2, 1)$ yields the lowest objective in every problem.

Table 3. Ablation results for different (N, M) cycle configurations.

(N, M)	TSP-ACO Obj. (↓)	BPP-ACO Obj. (↓)	ACS Obj. (↓)	WSN Obj. (↓)
$(N, /)$ – Natural Evo	5.83	205.28	1293.54	98.19
$(/, M)$ – ME reflection-guided only	5.83	204.80	588.50	58.40
$(1, 1)$ – Interleaved reflection	5.75	204.44	584.24	53.37
$(1, 2)$ – Reflection-guided heavy	5.77	204.68	581.28	55.24
$(2, 1)$ – MeEvo	**5.72**	**204.28**	**578.16**	**50.80**

To isolate the cyclic NE–ME architecture from inherited support components, we evaluate three controlled variants under the same backbone, budgets, and five-run protocol: *No APA* removes only Automated Problem Analysis, *No ED* removes Error Diagnosis and its repair attempts, and *NE–ME only* removes both while retaining common interface requirements.

Table 4. Ablation of inherited support components. All variants use the same complete problem description, interface requirements, backbone, and evaluation protocol; No APA disables only the automated analysis output, while No ED disables error diagnosis and its associated repair attempts.

Setting	**TSP-ACO** Obj. (↓)	**BPP-ACO** Obj. (↓)	**ACS** Obj. (↓)	**WSN** Obj. (↓)
Full MeEvo	**5.72**	**204.28**	**578.16**	**50.80**
No APA	5.75	205.28	584.29	52.92
No ED	5.74	205.44	586.91	53.82
NE–ME only (No APA + No ED)	5.76	205.84	590.74	55.17

Effect of APA. With the same complete problem description, removing APA increases the objective by 0.52%, 0.49%, 1.06%, and 4.17% on TSP-ACO, BPP-ACO, ACS, and WSN, respectively (**Table 4**). APA has limited direct effect and mainly provides convenient, portable problem-aware initialization.

Effect of ED. Removing ED and repair increases the objective by 0.35%, 0.57%, 1.51%, and 5.94% on TSP-ACO, BPP-ACO, ACS, and WSN, respectively; effects are larger on ACS and WSN, while NE–ME remains active.

Core NE–ME architecture. Removing both components increases the objective by 0.70%, 0.76%, 2.18%, and 8.60%; retained performance supports NE–ME as the main contributor, with APA and ED providing complementary initialization and error handling.

Computational cost and statistical significance analyses are reported in **Appendix G.3** and **Appendix G.4**.

5 Conclusion

We introduced MeEvo, which cyclically couples Natural Evolution on heuristic code with Metacognitive Evolution over shared history. Phase alternation provides macro-level exploration and exploitation, while Natural Evolution adapts its internal operator balance. Across five problems and two LLM backbones, it outperformed existing LLM-based AHD architectures, achieved lower variance on constrained ACS and WSN, and remained competitive with LLM-based methods on TSP-Construct while generalizing reliably to larger instances. Ablations show that both phases and sufficient exploration before reflection are beneficial, with significance tests supporting systematic gains.

REFERENCES

Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. Hifo-prompt: Prompting with hindsight and foresight for llm-based automatic heuristic design. *arXiv preprint arXiv:2508.13333*, 2025.

Long Chen, Zishang Qiu, Ying Wu, and Zhenzhou Tang. Optimizing k-coverage in energy-saving wireless sensor networks based on the elite global growth optimizer. *Expert Systems with Applications*, 256:124878, 2024.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

Pedro G Espejo, Sebastián Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(2):121–144, 2009.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv e-prints*, pp. arXiv–2309, 2023.

John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

Kashif Hussain, Mohd Najib Mohd Salleh, Shi Cheng, and Yuhui Shi. On the exploration and exploitation in popular swarm-based metaheuristic algorithms. *Neural Computing and Applications*, 31(11):7665–7683, 2019.

James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pp. 1942–1948. ieee, 1995.

Jae-Hwan Kim and Bong-Jin Yum. A heuristic method for solving redundancy optimization problems in complex systems. *IEEE Transactions on Reliability*, 42(4):572–578, 1993.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

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.

Emily R Lai. Metacognition: A literature review. *Always learning: Pearson research report*, 24: 1–40, 2011.

Wenjun Li, Dexun Li, Kuicai Dong, Cong Zhang, Hao Zhang, Weiwen Liu, Yasheng Wang, Ruiming Tang, and Yong Liu. Adaptive tool use in large language models with meta-cognition trigger. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13346–13370, 2025.

Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

Wenjie Lin and Jin Wei-Kocsis. Think, reflect, create: Metacognitive learning for zero-shot robotic planning with llms. *arXiv e-prints*, pp. arXiv–2505, 2025.

Fei Liu, Xialiang Tong, 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. *arXiv preprint arXiv:2401.02051*, 2024.

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

Yihong Liu, Junyi Li, Wayne Xin Zhao, Hongyu Lu, and Ji-Rong Wen. Experience-guided reflective co-evolution of prompts and heuristics for automatic algorithm design. *arXiv preprint arXiv:2509.24509*, 2025.

Marcelo de Oliveira Costa Machado, Natalie Ferraz Silva Bravo, André Ferreira Martins, Heder Soares Bernardino, Eduardo Barrere, and Jairo Francisco de Souza. Metaheuristic-based adaptive curriculum sequencing approaches: a systematic review and mapping of the literature. *Artificial Intelligence Review*, 54(1):711–754, 2021.

Manuj Malik, Jianan Zhou, Shashank Reddy Chirra, and Zhiguang Cao. PyVRP$^+$: Llm-driven metacognitive heuristic evolution for hybrid genetic search in vehicle routing problems. *arXiv preprint arXiv:2604.07872*, 2026. Accepted at AAMAS 2026.

Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pp. 50–60, 1947.

André Ferreira Martins, Marcelo Machado, Heder Soares Bernardino, and Jairo Francisco de Souza. A comparative analysis of metaheuristics applied to adaptive curriculum sequencing. *Soft Computing*, 25(16):11019–11034, 2021.

Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. Genetic programming: An introductory tutorial and a survey of techniques and applications. *Univ. Essex School of Computer Science and Eletronic Engineering Technical Report No. CES-475*, pp. 1–112, 2007.

Fubo Qi, Tianyu Wang, Ruixiang Zheng, and Mian Li. A memetic and reflective evolution framework for automatic heuristic design using large language models. *Applied Sciences*, 15(15):8735, 2025.

Zishang Qiu, Xinan Chen, Long Chen, Wenjie Yi, and Ruibin Bai. Mela: A metacognitive llm-driven architecture for automatic heuristic design. *Expert Systems with Applications*, 330:133022, 2026. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2026.133022. URL https://www.sciencedirect.com/science/article/pii/S0957417426019330.

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.

Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.

Peter Ross, Sonia Schulenburg, JG Marin-Blázquez, and Emma Hart. Hyper-heuristics: learning to combine simple heuristics in bin-packing problem. In *Proceedings of the Genetic and Evolutionary Computation COnference, GECCO*, volume 2, 2000.

Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation*, 19(3):309–325, 2014.

Amir Seyyedabbasi and Farzad Kiani. Sand cat swarm optimization: A nature-inspired algorithm to solve global optimization problems. *Engineering with computers*, 39(4):2627–2651, 2023.

Yuqing Wang and Yun Zhao. Metacognitive prompting improves understanding in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1914–1926, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.

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

Wenjie Yi, Rong Qu, Licheng Jiao, and Ben Niu. Automated design of metaheuristics using reinforcement learning within a novel general search framework. *IEEE Transactions on Evolutionary Computation*, 27(4):1072–1084, 2022.

Rui Zhang, Fei Liu, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Understanding the importance of evolutionary search in automated heuristic design with large language models. In *International Conference on Parallel Problem Solving from Nature*, pp. 185–202. Springer, 2024.

Shaofeng Zhang, Shengcai Liu, Ning Lu, Jiahao Wu, Ji Liu, Yew-Soon Ong, and Ke Tang. Llm-driven instance-specific heuristic generation and selection. *arXiv preprint arXiv:2506.00490*, 2025.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Zican Dong, Yupeng Hou, Beichen Zhang, Yingqian Min, Junjie Zhang, Peiyu Liu, et al. A survey of large language models. *Frontiers of Computer Science*, 20(12):2012627, 2026.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*, 2025.

AI USE STATEMENT

This study develops and evaluates an LLM-based automatic heuristic design framework. DeepSeek-V4-Flash and MIMO-v2.5-Pro are the computational backbones of the proposed method and the objects of experimental evaluation, rather than auxiliary tools used to formulate research questions, design the experimental methodology, or make scientific decisions. Within the framework, the models perform problem analysis, heuristic generation, crossover and mutation, execution-error diagnosis and repair, and reflection over accumulated search history. These operations constitute the implementation of the method and the interpretation of fitness and execution feedback during search; we made the research decisions and drew the final scientific conclusions.

We have not used generative AI tools to generate synthetic data sets, develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in writing proofs, propose or refine research hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean or reformat datasets, or support qualitative and thematic data analysis. The remaining required-disclosure tasks are not applicable to this work.

Separately, during manuscript preparation, generative AI assistance was limited to language polishing and LaTeX formatting. The heuristic code and experimental artifacts produced within the LLM-based workflow were verified through execution and benchmark evaluation under the protocols described in this paper, and the reported results and manuscript text were checked by the authors for correctness and consistency. We take responsibility for the final content of this work, including text, claims, code, and other artifacts produced with the aid of generative AI.

APPENDIX

A CONSOLIDATED NOTATION AND SYMBOL DEFINITIONS

A.1 NOTATION TABLE

Table A1. Consolidated Notation and Symbol Definitions

Symbol	Definition
	Problem and evaluation
P	Target optimization problem
APA	Automated problem analysis description
$f(h), F$	Individual fitness and fitness set
Err, ERR	Individual and population error records
$Eval, L$	Primary evaluation counter and maximum budget
	Population and evolution
N_0	Initial population size
N_{pop}	Offspring count per generation
h, H	Individual heuristic and population set
h^*	Global best heuristic (elitist)
$\mathcal{P}$	Parent pool (top-K feasible heuristics)
K	Parent pool size for truncation selection
COR	Error correction attempt limit
	Reasoning traces and shared history
Th, TH	Reasoning trace and reasoning history
$Meta$	Meta Insight (reflection output)
$\mathcal{R}$	Metacognitive reflection function (Eq. (3))
$\mathcal{G}, \mathcal{T}$	Insight-to-code and insight-to-trace mapping functions (Eq. (4))
	Cyclic alternation architecture
N, M	Natural Evolution generations / reflection-guided generation rounds per cycle
p_c, p_m	Crossover and mutation probabilities
α	Baseline crossover probability at convergence (Eq. (1))
β	Adaptation amplitude (Eq. (1))
	Baseline method parameters (GP & ACO)
μ	GP population size
G	GP generations
r_c, r_m	GP subtree crossover and mutation rates
k_{tour}	GP tournament selection size
$D_{\max}, d_{\max}$	GP max tree depth (evolution / initialization)
e	GP elitism count
γ	GP parsimony coefficient (penalizes tree size)
n_{ants}, n_{iter}	ACO colony size and iterations
ρ	ACO pheromone evaporation rate
α_p, β_h	ACO pheromone / heuristic exponents (distinct from MeEvo α, β)
β^x, β^y	WSN environmental fading coefficients (distinct from MeEvo β)

B AUXILIARY ALGORITHMS

B.1 NATURAL EVOLUTION PROCEDURE

Algorithm B1 Natural Evolution: Population-Driven Crossover and Mutation

procedure NATURALEVO($H, F, Eval, L, N_{pop}, K$)
  // Step 1: Select top-K parents by fitness
  $\mathcal{H}_v \leftarrow \{h \in H \mid f(h) \neq \infty\}$; $\mathcal{P} \leftarrow$ top-K of $\mathcal{H}_v$ sorted by $f(h)$ ascending
  if $|\mathcal{P}| < 2$ **then**
    return $\emptyset, \emptyset$ ▷ Insufficient parents
  end if
  // Step 2: Compute dynamic probabilities
  $p_c \leftarrow \alpha + \beta \times (1 - Eval/L)$; $p_m \leftarrow 1 - p_c$ ▷ Crossover and mutation probabilities (Eq. (1))
  // Step 3: Generate offspring via LLM; $H_{off} \leftarrow \emptyset$, $TH_{off} \leftarrow \emptyset$
  for $n \leftarrow 1$ **to** N_{pop} **do**
    $r \leftarrow$ uniform$(0, 1)$
    if $r < p_c$ **then**
      $h_a, h_b \leftarrow$ random sample from $\mathcal{P}$, 2 distinct
      $h_{off}, Th_{off} \leftarrow$ LLMCROSSOVER(
          h_a, h_b, APA)
    else
      $h_a \leftarrow$ random sample from $\mathcal{P}$, 1
      $h_{off}, Th_{off} \leftarrow$ LLMMUTATE(
          h_a, APA)
    end if
    $H_{off} \leftarrow H_{off} \cup \{h_{off}\}$
    $TH_{off} \leftarrow TH_{off} \cup \{Th_{off}\}$
  end for
  return H_{off}, TH_{off}
end procedure

B.2 EVALUATION AND ERROR CORRECTION

The budget L counts the primary search evaluations. When execution fails, Error Diagnosis invokes up to COR repair attempts; these candidates are tested as replacements for the faulty heuristic rather than retained as additional population members. Their evaluations do not increment $Eval$, and unselected correction candidates are not recorded as separate individuals. The additional LLM calls and wall-clock cost for diagnosis and correction are included in the resource accounting reported in **Appendix G.3**.

Algorithm B2 Evaluation and Error Correction

procedure EVALANDCORRECT(h, Th, COR)
  $ErrSet \leftarrow \emptyset$
  $f(h), Err \leftarrow$ EVALUATE(h)
  if $Err = \emptyset$ **then**
    return $h, f(h), Th, ErrSet$
  else
    $ErrSet \leftarrow ErrSet \cup \{Err\}$
    $h, f(h), Th_R, Err_R \leftarrow$
      REPLACEERR(h, Err, Th, COR)
    $Th \leftarrow Th \cup Th_R$
    $ErrSet \leftarrow ErrSet \cup Err_R$
    return $h, f(h), Th, ErrSet$
  end if
end procedure

B.3 METACOGNITIVE REFLECTION PROCEDURE

Algorithm B3 Replacement of Faulty Heuristics

procedure REPLACEERR($h_{Err}, Err, Th_{Err}, COR$)
  $h_{Rep} \leftarrow h_{Err}$, $f(h_{Rep}) \leftarrow \infty$
  $TH_R \leftarrow \emptyset$, $ERR_R \leftarrow \emptyset$
  for $k \leftarrow 1$ **to** COR **do**
    $h_C, Th_C \leftarrow$ CORRECTERRORS(
      h_{Err}, Err, Th_{Err})
    $TH_R \leftarrow TH_R \cup \{Th_C\}$
    $f(h_C), Err_C \leftarrow$ EVALUATE(h_C)
    if $Err_C = \emptyset$ **and** $f(h_C) < f(h_{Rep})$ **then**
      $h_{Rep} \leftarrow h_C$, $f(h_{Rep}) \leftarrow f(h_C)$
    else if $Err_C \neq \emptyset$ **then**
      $ERR_R \leftarrow ERR_R \cup \{Err_C\}$
    end if
  end for
  return $h_{Rep}, f(h_{Rep}), TH_R, ERR_R$
end procedure

Algorithm B4 Metacognitive Reflection

procedure METACOGNITIVEREFLECTION(TH, F, ERR, h^*)
  $S_f \leftarrow$ ANALYZEFITNESS(F, h^*)
  $S_o \leftarrow$ RETAINOPTIMAL(TH, F, h^*)
  $S_n \leftarrow$ SEARCHBETTER($TH, F, ERR,$
      S_f, S_o)
  $Meta \leftarrow$ CONSTRUCTPROMPT(
      S_f, S_o, S_n, ERR, h^*)
  return $Meta$
end procedure

C Prompt Details

C.1 Execution Errors and Problem Descriptions

C.2 System-Level Role Definitions

C.3 User-Level Prompt Definitions

Representative execution errors

ValueError: operands could not be broadcast together with shapes (50,) (50,150)
TypeError: “numpy.float64” object is not callable
IndexError: too many indices for array: array is 1-dimensional, but 2 were indexed
SyntaxError: “[” was never closed
ValueError: probabilities contain NaN

Initial Description
Implement a function that performs adaptive course material recommendation using the SCSO metaheuristic optimization algorithm.
Heuristic operation success rate : EOH - 5% ReEvo - The operation rate is too low to proceed to the crossover and mutation stages.
Complete Description
Implement a function that performs adaptive course material recommendation using the SCSO metaheuristic optimization algorithm. The function must:
1. Strict Requirements for Parameter Access:
- All parameters must be accessed using OBJECT.ATTRIBUTE notation only
- NEVER use dictionary-style access like data_al[’lb’] or data_al[0]
- Required parameter accesses that must use dot notation:
* data_al.lb - lower bounds
* data_al.ub - upper bounds
* data_al.dim - problem dimension
* data_al.SearchAgents - population size
* data_al.MaxIter - maximum iterations
2. Mandatory Implementation Structure:
# 1. Parameter initialization (MUST use dot notation)
lb = np.array(data_al.lb)
ub = np.array(data_al.ub)
dim = data_al.dim
SearchAgents_no = data_al.SearchAgents
# 2. Position initialization
ub_array = np.array(ub)
lb_array = np.array(lb)
position initialization logic
3. Core Algorithm Requirements:
- Must implement both exploration and exploitation layers
- Must include boundary constraint handling
- Must use cosine-based position updates
- Must maintain roulette wheel selection
4. Input/Output Specifications:
Input Parameters:
- data_al: Algorithm config object with dot-accessible attributes
- data_pb: Problem data object
- Positions: Current population positions
- Best_pos: Current best solution
- Best_score: Current best fitness
- rg: Current search radius
Returns:
- Updated Positions array only
- NO other return values allowed
Heuristic operation success rate : EOH - 75.49% ReEvo - 40.71%

Figure 5. Representative execution errors and incomplete problem descriptions that result in no runnable heuristics for EoH and ReEvo.

(1) Problem Analysis Expert *Function:* `problem_role()`

Role: "You are an expert in optimization algorithm design, metaheuristics, and computational complexity analysis."
Task: Analyze the given optimization problem and produce a structured technical report.
Analysis framework:
(1) *Problem Classification*—paradigm (combinatorial/continuous), search space (dimensionality, variable types, bounds), complexity class;
(2) *Solution Approaches*—heuristic function role (inputs/computation/outputs), constraints (hard/soft with variable names), algorithm families (2–3 recommended with justification);
(3) *Failure Mode Analysis*—2–3 specific failure modes with trigger conditions and vulnerable code regions.
Output: Numbered report (sections 1–3) with specific references to code elements.

---

(2) System Role—with problem context *Function:* `system_role(problem_analysis)`

Role: "You are an expert in evolutionary computing, swarm intelligence, and algorithm optimization."
Task: Using the embedded problem context, iteratively improve the provided algorithm. Reference the problem analysis to guide optimization choices.
Output format: A reasoning block {thought_process: (1) Error or weakness diagnosis, (2) Solution rationale, (3) Impact assessment} followed by a complete Python code block with #EVOLVE-START/#EVOLVE-END markers.
Constraints: Modify ONLY code between #EVOLVE-START/#EVOLVE-END; keep function name `heuristics_v2` and input signature unchanged; returned `Positions` must match baseline shape/type; no new dependencies beyond numpy.

---

(3) System Role—without problem context *Function:* `system_role()`

Role: "You are an expert in evolutionary computing and algorithm optimization."
Task: Iteratively improve the provided algorithm by modifying only the designated region.
Output format and constraints: Identical to Variant (2), but without embedded problem context. Used when problem analysis is unavailable.

---

(4) Error Diagnosis Expert *Function:* `error_role()`

Role: "You are an expert Python code debugger specializing in numerical and scientific computing."
Task: Fix the reported error in the provided algorithm code.
Output format: A reasoning block {thought_process: (1) Error diagnosis—classify root cause (syntax/type mismatch/shape error/index out of bounds/division by zero/logic flaw); (2) Solution rationale; (3) Impact assessment—confirm no breakage of other functionality} followed by corrected Python code.
Constraints: Make the MINIMAL change necessary; do not refactor unrelated code; keep function name and signature unchanged.

---

(5) Metacognitive Reflection Expert *Function:* `e_learning_role()`

Role: "You are a self-reflective AI optimizer conducting metacognitive analysis of your own evolutionary search process."
Task: Examine the optimization history and identify strategic improvement directions.
Analysis framework:
(1) *Cognitive Limitations*—systematic biases or blind spots (e.g., over-reliance on single search mechanism, insufficient diversity maintenance);
(2) *Potential Biases*—optimization strategies repeated despite ineffectiveness, convergence to familiar patterns;
(3) *Optimization Pathways*—specific algorithmic changes with named techniques (e.g., "replace uniform random initialization with opposition-based learning").
Constraints: Reference actual strategies from evolution history; keep analysis under 80 words; focus on structural changes, not parameter tuning.

Figure 6. System-level role definitions for the five LLM call types in MeEvo. Each role is implemented as an XML-structured system prompt that defines the LLM's expertise persona, task specification, analysis framework, and output constraints. The corresponding user-level prompts are shown in **Fig. 7**.

(1) Problem Analysis Prompt *Function:* `problem_prompt(problem)`

Task: Analyze the following optimization problem in depth.
Input: {problem}—the complete Python source code of the optimization problem.
Analysis requirements:
(1) *Optimization Paradigm*—classify as combinatorial or continuous with evidence;
(2) *Search Space*—dimensionality, variable types, bounds;
(3) *Constraints*—enumerate all hard/soft constraints with variable names;
(4) *Heuristic Function Role*—inputs, computation, outputs;
(5) *Suitable Techniques*—3–5 proven techniques with justification;
(6) *Failure Modes*—2–3 concrete failure modes with triggers.
Output: Numbered list (1–6), 2–4 sentences per section with specific code references.

---

(2) Initial Generation Prompt *Function:* `return_promoter_init(...)`

Role: Expert algorithm optimizer specializing in evolutionary computing and metaheuristics.
Inputs: {problem} (problem context), {init_code} (baseline), {init_eval} (history), {code} (current code).
Task: Transform the algorithm by selecting TWO complementary techniques from:
— *Exploration*: Levy flight, Cauchy mutation, opposition-based learning, chaotic initialization;
— *Exploitation*: local search (2-opt, VND), gradient-informed update, Nelder-Mead simplex;
— *Diversity*: crowding distance, fitness sharing, restart mechanism, adaptive population;
— *Constraint handling*: repair operator, feasibility rules, penalty adaptation.
Output: Reasoning block {thought_process} + complete Python code. Modify ONLY between #EVOLVE-START and #EVOLVE-END.

---

(3) Error Correction Prompt *Function:* `error_prompt(str_error, code_str)`

Task: Fix the reported error with minimal changes.
Inputs: {str_error} (error message), {code_str} (faulty code).
Requirements:
(1) *Root Cause*—classify as syntax/type/shape/index/division/logic error with specific line;
(2) *Fix*—minimal correction within #EVOLVE-START/#EVOLVE-END;
(3) *Verification*—confirm no new issues introduced.
Output: {error_analysis} + {fix_strategy} + corrected Python code block.

---

(4) Metacognitive Reflection Prompt *Function:* `e_learning_prompt(...)`

Task: Perform metacognitive analysis of the evolutionary optimization process.
Inputs: {thoughts} (evolution reasoning records), {errors} (errors encountered), {code} (current best), {history} (convergence history).
Analysis framework:
(1) *Convergence Diagnosis*—classify into PREMATURE_CONVERGENCE / STEADY_IMPROVEMENT / OSCILLATION / STAGNATION;
(2) *Component Analysis*—identify 2–3 strongest algorithmic components with code references;
(3) *Weakness Diagnosis*—1–2 fundamental weaknesses requiring structural redesign;
(4) *Strategy Recommendation*—concrete change direction with named techniques.
Constraints: Do NOT repeat error-causing strategies; response under 100 words; reference actual code elements.

Figure 7. Pre-defined user-level prompts for the first four LLM call types in MeEvo: (1) Problem Analysis, (2) Initial Generation, (3) Error Correction, and (4) Metacognitive Reflection.

(5) Metacognition-Guided Generation Prompt *Function:* `metacognition_prompt(...)`

Role: Expert algorithm optimizer performing an iteration of evolutionary improvement.
Inputs: {problem}, {metacognition} (metacognitive insights), {init_code} (original baseline), {init_eval}, {code} (current code).
Task: (1) Preserve strong components from the original baseline; (2) Redesign weak areas identified in metacognitive insights with fundamentally different approaches; (3) Modify ONLY between #EVOLVE-START/#EVOLVE-END; (4) Align with problem structure.
Output: {reasoning} (1–3 sentences on what insight guided the change) + complete Python code.

---

(6) Crossover Prompt *Function:* `crossover_prompt(...)`

Role: Expert algorithm optimizer performing crossover between two parent solutions.
Inputs: {parent1_code}, {parent2_code}, {problem_description}.
Task: (1) Analyze each parent's optimization technique and strengths; (2) Select superior component from each parent; (3) Synthesize a coherent hybrid resolving incompatibilities; (4) Introduce at least ONE novel enhancement.
Output: {reasoning} (2–4 sentences on component provenance and novel enhancement) + complete Python code.

---

(7) Mutation Prompt *Function:* `mutation_prompt(...)`

Role: Expert algorithm optimizer performing an innovative mutation on a parent solution.
Inputs: {parent_code}, {problem_description}.
Task: (1) Analyze parent's primary optimization mechanism; (2) Identify 2–3 mutation opportunities; (3) Implement the most promising mutation that fundamentally changes the heuristic's search behavior.
Mutation guidance:
— *Continuous*: Levy flight, Cauchy mutation, adaptive step size, differential evolution, chaotic perturbation;
— *Combinatorial*: 2-opt/3-opt local search, swap/inversion mutation, simulated annealing acceptance, VND.
Output: {reasoning} (2–4 sentences on mutation type and expected impact) + complete Python code.

Figure 8. Pre-defined user-level prompts for the remaining three LLM call types in MeEvo: (5) Metacognition-Guided Generation, (6) Crossover, and (7) Mutation. Each prompt specifies the task, inputs, output format, and constraints. The corresponding system-level role definitions are shown in **Fig. 6**.

D BEST HEURISTIC CODES GENERATED BY MEEVO

D.1 BEST HEURISTIC IMPLEMENTATIONS

The following are the best heuristic codes generated by MeEvo (using DeepSeek-V4-Flash) for the four optimization problems. These codes correspond to the best objective values reported in **Appendix Table G8**.

TSP-ACO (Best Obj: 5.712)

```python
def heuristics_v2(distance_matrix):
    dist = np.array(distance_matrix, dtype=float)
    n = dist.shape[0]
    eps = 1e-10
    inv_dist = 1.0 / (dist + eps)
    inv_max = np.max(inv_dist)
    inv_norm = inv_dist / inv_max if inv_max > 0 else inv_dist
    max_starts = min(250, max(40, n * 8))
    freq = np.zeros((n, n))
    all_costs = []
    tours = []
    for _ in range(max_starts):
        start = np.random.randint(n)
        unvisited = set(range(n))
        path = [start]
        unvisited.remove(start)
        current = start
        while unvisited:
            unvisited_list = np.array(list(unvisited))
            dists = dist[current, unvisited_list]
            probs = 1.0 / (dists + eps) ** 2.5
            probs /= probs.sum()
            next_city = np.random.choice(unvisited_list, p=probs)
            path.append(next_city)
            unvisited.remove(next_city)
            current = next_city
        tour = path
        tour_len = len(tour)
        improved = True
        max_iter = 200
        it = 0
        while improved and it < max_iter:
            improved = False
            it += 1
            for i in range(tour_len - 1):
                for j in range(i + 2, tour_len - (1 if i == 0 else 0)):
                    a, b = tour[i], tour[(i+1) % tour_len]
                    c, d = tour[j], tour[(j+1) % tour_len]
                    if dist[a, c] + dist[b, d] < dist[a, b] + dist[c, d] - 1e-9:
                        tour = tour[:i+1] + tour[i+1:j+1][::-1] + tour[j+1:]
                        improved = True
                        break
                if improved:
                    break
        cost = sum(dist[tour[i], tour[(i+1) % tour_len]] for i in range(tour_len))
        all_costs.append(cost)
        tours.append(tour)
    sorted_idx = np.argsort(all_costs)
    rank_weights = np.zeros(max_starts)
    for rank, idx in enumerate(sorted_idx):
        rank_weights[idx] = 1.0 / (rank + 1) ** 1.8
    for idx, tour in enumerate(tours):
        w = rank_weights[idx]
        for i in range(n):
            u = tour[i]
            v = tour[(i+1) % n]
            freq[u, v] += w
            freq[v, u] += w
    max_freq = np.max(freq)
    if max_freq > 0:
        norm_freq = freq / max_freq
    else:
        norm_freq = freq
    freq_pow = np.sqrt(norm_freq)
    alpha = 1.0
    heuristic = inv_norm * (1.0 + alpha * freq_pow)
    noise = 1.0 + 0.005 * np.random.randn(n, n)
    noise = (noise + noise.T) / 2.0
```

```
    heuristic = heuristic * noise
    K = max(7, min(45, n // 2))
    epsilon = 1e-6
    for i in range(n):
        row = heuristic[i, :]
        top = np.argsort(row)[-K:]
        mask = np.zeros(n, dtype=bool)
        mask[top] = True
        heuristic[i, ~mask] = epsilon
    heuristic = (heuristic + heuristic.T) / 2.0
    heuristic = np.clip(heuristic, epsilon, None)
    return heuristic
```

BPP-ACO (Best Obj: 204.20)

```
def heuristics_v2(node_attr, node_constraint):
    demand = np.asarray(node_attr, dtype=float)
    cap = np.asarray(node_constraint).flatten()[0]
    n = len(demand)
    i_idx = np.arange(n)[:, None]
    j_idx = np.arange(n)[None, :]
    sum_matrix = demand[i_idx] + demand[j_idx]
    feasible = (sum_matrix <= cap) & (i_idx != j_idx)
    n_feasible = feasible.sum(axis=1)
    prod_matrix = demand[i_idx] * demand[j_idx]
    waste = cap - sum_matrix
    heu = np.where(feasible, prod_matrix / (waste + 1e-10), 1e-10)
    crit_bonus = 1.0 / (n_feasible[i_idx] + n_feasible[j_idx] + 1.0)
    heu = heu * (1.0 + crit_bonus)
    np.fill_diagonal(heu, 0.0)
    return heu
```

ACS (Best Obj: 564.44)

```
def heuristics_v2(Positions, Best_pos, Best_score, rg):
    SearchAgents_no = Positions.shape[0]
    dim = Positions.shape[1]
    lb_array = np.zeros((SearchAgents_no, dim))
    ub_array = np.ones((SearchAgents_no, dim))
    rand_adjust = lb_array + (ub_array - lb_array) * np.random.rand(*Positions.shape)
    Positions = np.where((Positions < lb_array) | (Positions > ub_array), rand_adjust,
     Positions)
    if SearchAgents_no < 4:
        Positions = Positions + 0.1 * rg * np.random.randn(*Positions.shape) * (Best_pos -
     Positions)
        return np.clip(Positions, lb_array, ub_array)
    if np.any(np.isnan(Best_pos)) or np.isnan(Best_score):
        median_pos = np.median(Positions, axis=0)
        Positions = 0.5 * Positions + 0.5 * median_pos + 0.05 *
     np.random.randn(*Positions.shape)
        return np.clip(Positions, lb_array, ub_array)
    binary_Positions = (Positions > 0.5).astype(float)
    binary_Best = (Best_pos > 0.5).astype(float)
    archive_size = max(3, int(SearchAgents_no * (0.2 + 0.5 * (rg / 2.5))))
    def select_diverse_subset(pop, k):
        n = pop.shape[0]
        if n <= k:
            return np.arange(n)
        selected = [np.argmax(np.sum(np.abs(pop - binary_Best), axis=1))]
        for _ in range(k - 1):
            dists_to_selected = np.min([np.sum(np.abs(pop - pop[s]), axis=1) for s in
     selected], axis=0)
            dists_to_selected[selected] = -1
            selected.append(np.argmax(dists_to_selected))
        return np.array(selected)
    archive_indices = select_diverse_subset(binary_Positions, min(archive_size,
     SearchAgents_no))
    Archive = binary_Positions[archive_indices]
    idx = np.arange(SearchAgents_no)
    a1 = np.random.randint(0, len(Archive), size=SearchAgents_no)
    a2 = np.random.randint(0, len(Archive), size=SearchAgents_no)
    mask_same = (a1 == a2)
    a2 = np.where(mask_same, (a2 + 1) % len(Archive), a2)
    F = 0.5 + 0.4 * (rg / 2.5)
    binary_diff = Archive[a1] - Archive[a2]
    structural_diff = 4 * binary_diff * (1 - np.abs(binary_diff))
    mutant = Best_pos + F * structural_diff
```

```
    temperature = 0.05 + 0.95 * (rg / 2.5)
    uncertainty = np.abs(binary_Positions - Archive[a1]) * np.abs(binary_Positions - Archive[a2])
    uncertainty = np.clip(uncertainty + 0.05, 0, 1)
    CR_base = 0.5 + 0.4 * (1 - rg / 2.5)
    CR = CR_base + 0.35 * uncertainty * temperature
    CR = np.clip(CR, 0.2, 0.95)
    j_rand = np.random.randint(0, dim, size=SearchAgents_no)
    crossover_mask = np.random.rand(*Positions.shape) < CR
    crossover_mask[np.arange(SearchAgents_no), j_rand] = True
    trial = np.where(crossover_mask, mutant, Positions)
    proximity_to_threshold = 1 - np.abs(2 * trial - 1)
    push_strength = 0.5 * (1 - rg / 2.5) * proximity_to_threshold
    push_direction = (trial > 0.5).astype(float)
    trial = trial + push_strength * (push_direction - trial)
    current_binary = (trial > 0.5).astype(float)
    coord_importance = np.abs(trial - 0.5)
    direction_to_best = 2 * (binary_Best - 0.5)
    current_direction = 2 * (current_binary - 0.5)
    alignment = direction_to_best * (1 - 2 * (current_direction == direction_to_best).astype(float))
    explore_weight = rg / 2.5
    exploit_weight = 1 - explore_weight
    guided_score = coord_importance * (explore_weight * np.random.randn(*trial.shape) + exploit_weight * alignment)
    num_refine = max(1, int(dim * 0.05 * (2.5 - rg) / 2.5))
    if num_refine > 0:
        top_coords = np.argpartition(-np.abs(guided_score), num_refine, axis=1)[:, :num_refine]
        for i in range(SearchAgents_no):
            for j in top_coords[i]:
                if np.random.rand() < exploit_weight:
                    target = binary_Best[j]
                else:
                    target = 1 - current_binary[i, j]
                trial[i, j] = 0.5 + (target - 0.5) * (0.5 + 0.5 * np.random.rand())
    Positions = np.clip(trial, lb_array, ub_array)
    invalid_mask = ~np.isfinite(Positions)
    if np.any(invalid_mask):
        replacement = 0.5 + 0.2 * np.random.randn(*Positions.shape)
        Positions = np.where(invalid_mask, replacement, Positions)
        Positions = np.clip(Positions, lb_array, ub_array)
    return Positions
```

WSN (Best Obj: 50.00)

```
def heuristics_v2(Positions, Best_pos, Best_score, rg):
    SearchAgents_no = Positions.shape[0]
    dim = Positions.shape[1]
    num_cn = dim // 3
    lb = np.zeros(dim)
    ub = np.tile([50.0, 50.0, 30.0], 50)
    Positions = np.clip(Positions, lb, ub)
    F = 0.5 + 0.5 * (rg / 10.0)
    CR = 0.7 + 0.3 * (rg / 10.0)
    Cauchy_scale = 0.1 * (rg / 10.0)
    best_resh = Best_pos.reshape(num_cn, 3)
    def compute_connectivity_components(coords):
        dists = np.linalg.norm(coords[:, None] - coords[None, :], axis=2)
        adj = dists <= 20.0
        visited = np.zeros(num_cn, dtype=bool)
        components = []
        for i in range(num_cn):
            if not visited[i]:
                stack = [i]
                comp = []
                while stack:
                    node = stack.pop()
                    if visited[node]:
                        continue
                    visited[node] = True
                    comp.append(node)
                    neighbors = np.where(adj[node] & ~visited)[0]
                    stack.extend(neighbors.tolist())
                components.append(np.array(comp))
        return components
    def repair_connectivity(trial):
        trial_resh = trial.reshape(num_cn, 3)
        coords = trial_resh[:, :2]
```

```
        components = compute_connectivity_components(coords)
        if len(components) <= 1:
            return trial
        largest_idx = np.argmax([len(c) for c in components])
        largest_comp = components[largest_idx]
        other_nodes = np.concatenate([c for i, c in enumerate(components) if i !=
    largest_idx])
        large_coords = coords[largest_comp]
        for node in other_nodes:
            node_coord = coords[node]
            dists = np.linalg.norm(large_coords - node_coord, axis=1)
            nearest_idx = largest_comp[np.argmin(dists)]
            trial_resh[node, :2] += 0.6 * (coords[nearest_idx] - node_coord)
            trial_resh[node, 2] = trial_resh[node, 2] * 0.4 +
    np.mean(trial_resh[largest_comp, 2]) * 0.6
        trial = np.clip(trial_resh.ravel(), lb, ub)
        return trial
    def repair_power(trial):
        trial_resh = trial.reshape(num_cn, 3)
        p = trial_resh[:, 2]
        std_p = np.std(p)
        if std_p > 1.0:
            mean_p = np.mean(p)
            p = mean_p + (p - mean_p) * (1.0 / std_p)
            trial_resh[:, 2] = p
            trial = np.clip(trial_resh.ravel(), lb, ub)
        return trial
    for i in range(SearchAgents_no):
        r1, r2 = np.random.randint(SearchAgents_no, size=2)
        while r1 == i:
            r1 = np.random.randint(SearchAgents_no)
        while r2 == i or r2 == r1:
            r2 = np.random.randint(SearchAgents_no)
        mutant = Positions[i] + F * (Best_pos - Positions[i]) + F * (Positions[r1] -
    Positions[r2])
        cauchy_noise = np.random.standard_cauchy(dim) * Cauchy_scale
        mutant += cauchy_noise
        j_rand = np.random.randint(dim)
        trial = np.where(np.random.rand(dim) < CR, mutant, Positions[i])
        trial[j_rand] = mutant[j_rand]
        trial = repair_connectivity(trial)
        trial = repair_power(trial)
        if rg < 2.0 and np.random.rand() < 0.3:
            pert = np.random.randn(dim) * (rg / 30.0)
            trial = Best_pos + pert
            trial = np.clip(trial, lb, ub)
            trial = repair_connectivity(trial)
            trial = repair_power(trial)
        Positions[i] = trial
    return Positions
```

E Extended Literature Review

E.1 Natural Evolution for AHD: Crossover, Mutation, and Population-Driven Search

Combining LLMs with evolutionary computation has produced a new generation of AHD architectures (Liu et al., 2024). A central thread applies *Natural Evolution* operators (crossover, mutation, and selection acting on a population) to explore the heuristic code space. FunSearch (Romera-Paredes et al., 2024) constrained LLM-generated code within a predefined skeleton and used selection to retain top performers. EoH (Liu et al., 2024) introduced LLM-mediated crossover and mutation, co-evolving natural-language thoughts alongside heuristic code. ReEvo (Ye et al., 2024) enhanced these operators with dual-level reflection that analyzes parent pairs to guide crossover and accumulates expertise across iterations to guide mutation. MCTS-AHD (Zheng et al., 2025) replaced the population with a Monte Carlo tree, preserving underperforming candidates as revisitable nodes for more thorough exploration. EoH-S (Liu et al., 2026) extended EoH by evolving a set of heuristics rather than a single best candidate, maintaining population diversity through heuristic-set-level selection. Zhang et al. (2025) instead addressed instance heterogeneity by partitioning problem classes into feature-based sub-classes and generating tailored heuristics for each—a direction orthogonal to population-level evolution. A parallel line of work evolves the prompts guiding code generation (OPRO (Yang et al., 2024), EvoPrompt (Guo et al., 2023), HiFo-Prompt (Chen et al., 2025), EvoPH (Liu et al., 2025)) rather than the code itself; these architectures address a different challenge and are summarized in **TABLE E2**.

Despite their success, Natural Evolution architectures primarily apply evolutionary operators to heuristic code. Reasoning traces may be produced during generation or used locally to guide an operator, as in EoH and ReEvo, but accumulated reasoning traces and performance outcomes are not consolidated through a separate history-level reflection process. Population-driven exploration can therefore discover diverse strategies without explicitly synthesizing design knowledge across evolutionary cycles.

E.2 Metacognitive Evolution for AHD: Reflection-Driven Refinement

A complementary thread targets the LLM's reasoning process itself through *metacognitive reflection*: the capacity to monitor, evaluate, and regulate one's own cognitive processes (Lai, 2011). This is distinct from reasoning enhancement techniques such as Chain-of-Thought (CoT) (Wei et al., 2022) and Tree-of-Thought (ToT) (Yao et al., 2023), which structure *how* the model thinks but do not govern *whether* its current strategy is effective. Metacognition is a higher-order regulatory layer: the ability to step back and assess whether the current reasoning trajectory is productive (Lin & Wei-Kocsis, 2025; Wang & Zhao, 2024; Li et al., 2025).

MeLA (Qiu et al., 2026) instantiated metacognition for AHD through Metacognitive Evolution. Rather than evolving heuristic code through crossover and mutation, or task descriptions through forward-looking optimization, MeLA metacognitively reflects on accumulated reasoning traces, execution errors, and fitness outcomes. Its metacognitive search engine analyzes these records to reinforce effective reasoning patterns, preserve high-performing algorithmic components from the best heuristic, and hypothesize novel improvement directions, then distills these insights to guide subsequent heuristic generation. However, MeLA operates as a single-process system: heuristic generation and metacognitive reflection are coupled within one loop, without the diversity that population-driven crossover and mutation provide. While metacognitive reflection progressively deepens the LLM's design logic, it cannot independently discover fundamentally novel algorithmic strategies outside its accumulated reasoning trajectory.

Emerging hybrid architectures represent early steps toward combining these two paradigms. EoH-MR (Qi et al., 2025) augments EoH with CMA-ES local optimization and a meta-cognitive reflection module that monitors population diversity, fitness improvement velocity, and stagnation status, then uses these signals to dynamically adjust operator selection probabilities (e.g., promoting exploratory mutation when diversity collapses) and prune low-yield prompt templates. PyVRP$^+$ (Malik et al., 2026) introduces a Reason-Act-Reflect cycle for metacognitive heuristic evolution. However, metacognitive reflection remains an auxiliary module serving code generation rather than a dedicated history-conditioned phase. In EoH-MR, meta-cognitive monitoring operates on the pop-

ulation as a guidance signal; in PyVRP$^+$, the Reason-Act-Reflect cycle refines individual solutions but does not alternate between population-driven exploration and metacognitive reflection as distinct phases. Consequently, neither provides a cyclic mechanism that bidirectionally couples population-driven code exploration with history-level reflection.

The progression from Natural Evolution architectures to Metacognitive Evolution, with recent hybrid efforts beginning to bridge them (**TABLE E2**), reveals a consistent pattern. Even in the most advanced architectures, population-driven crossover and mutation and reflection-driven metacognition have not been combined as distinct, cyclically coupled phases in a unified framework.

Table E2. Evolution Paradigm Comparison in LLM-based AHD

Method	Evolution Paradigm	Key Mechanism	Dedicated History-Level Reflection
FunSearch (Romera-Paredes et al., 2024)	Natural Evolution	Selection + skeleton constraint	No
EoH (Liu et al., 2024)	Natural Evolution	Crossover + mutation (5 strategies)	No
ReEvo (Ye et al., 2024)	Natural Evolution	Dual-level reflection-enhanced crossover/mutation	No
MCTS-AHD (Zheng et al., 2025)	Natural Evolution	Tree search (population alternative)	No
EoH-S (Liu et al., 2026)	Natural Evolution	Heuristic-set-level selection	No
Zhang et al. (2025)	Instance-specific AHD	Subclass partitioning + tailored heuristic generation	No
EvoPrompt (Guo et al., 2023)	Prompt-level	EC operators on task descriptions	No
OPRO (Yang et al., 2024)	Prompt-level	Training-set evaluation	No
HiFo-Prompt (Chen et al., 2025)	Prompt-level	Navigator rule engine	No
EvoPH (Liu et al., 2025)	Prompt-level	Policy sampling + Island migration	No
MeLA (Qiu et al., 2026)	Metacognitive Evolution	Metacognitive reflection	Yes
EoH-MR (Qi et al., 2025)	NE + auxiliary reflection	Meta-cognitive monitoring + CMA-ES	No
PyVRP$^+$ (Malik et al., 2026)	NE + auxiliary reflection	Reason-Act-Reflect cycle	No
MeEvo (Ours)	**NE + ME (cyclic alternation)**	**Phase alternation + operator balance**	**Yes**

E.3 Exploration-Exploitation Mechanisms in AHD

A central challenge in optimization is balancing *exploration* (searching broadly for new promising regions) against *exploitation* (refining solutions in known good regions) (Kennedy & Eberhart, 1995; Hussain et al., 2019). In classical algorithms, this balance is controlled by numerical parameters, such as crossover and mutation rates in Genetic Algorithms (GA) (Holland, 1992), inertia weights in Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995), but the optimal setting varies by problem and changes during the search.

LLM-based AHD introduces a new possibility: using the model's reasoning capabilities for higher-level, semantic exploration-exploitation decisions rather than relying solely on low-level numerical parameters. Existing architectures adopt distinct strategies (**TABLE E3**): HiFo-Prompt uses a Navigator rule engine; EvoPH uses policy sampling and island migration; EoH-MR uses meta-cognitive monitoring. All rely on external coordination mechanisms.

MeEvo differs from parallel co-evolution architectures such as EvoPH (Liu et al., 2025): rather than running two populations simultaneously with explicit coupling mechanisms (policy sampling, island migration), MeEvo cyclically alternates between Natural Evolution and Metacognitive Evolution. The rationale and implications of this design are detailed in Section 3.1 of the main paper.

Table E3. Comparison of Cyclic Alternation Architectures in LLM-based AHD

Method	Process 1	Process 2	Relationship	Exploration–Exploitation Control
HiFo-Prompt (Chen et al., 2025)	Prompt design	Code generation	Single-process + dual-module	Navigator rule engine
EvoPH (Liu et al., 2025)	Prompt template	Code generation	Parallel co-evolution	Policy sampling + Island migration
MeEvo (Ours)	**Natural Evolution (crossover+mutation)**	**Metacognitive Evolution (reflection)**	**Cyclic alternation**	**Hierarchical (phase alternation + operator balance)**

F EXTENDED METHOD DETAILS

F.1 NATURAL EVOLUTION OPERATORS

In the Natural Evolution process, the LLM acts as an evolutionary operator, applying population-driven crossover and mutation directly to heuristic code (see **Appendix B.1** for the full procedure). Crossover recombines heterogeneous parent strategies; mutation introduces local code modifications. Together, these operators support fine-grained exploration–exploitation balance within the broader Natural Evolution search. The reasoning traces produced alongside each generation are recorded with their fitness outcomes in shared history for subsequent metacognitive reflection. The entire population is replaced by offspring each generation. Each iteration executes three steps:

For example, with N=2 and M=1 (the default configuration validated in Section 4.3 of the main paper): Natural Evolution runs two generations of crossover and mutation, each producing N_{pop} offspring and recording their reasoning traces into shared history. Metacognitive Evolution then performs one reflection over this accumulated history and uses the resulting Meta Insight to generate one batch of N_{pop} improved heuristics, which are added to the population H. The next Natural Evolution cycle selects parents from this enriched pool, so metacognitively refined heuristics participate in crossover alongside existing ones. This NE→ME→NE rhythm repeats until the evaluation budget L is exhausted.

The *shared history* comprises four record types accumulated across all generations: reasoning traces TH, fitness values F, execution errors ERR, and the best heuristic h^*.

Reasoning traces should be understood as functionally transmitted design information rather than copied genetic strings. A trace produced in one generation is recorded in TH and made available to the reflection operator; the resulting Meta Insight conditions offspring generation, which produces a new heuristic code and reasoning trace. Transmission is therefore indirect, following the path $Th \rightarrow TH \rightarrow$ Meta $\rightarrow (h_{t+1}, Th_{t+1})$, while recorded traces remain in the shared history and are not individually selected, recombined, mutated, or discarded as fragments. Each Meta Insight is consequently grounded in the accumulated history supplied to its reflection round, rather than being a one-to-one transformation of a single parent trajectory.

F.1.1 PARENT SELECTION

Infeasible heuristics ($f(h) = \infty$) are excluded. The remaining feasible heuristics $\mathcal{H}_v$ are sorted by ascending fitness, and the top K form the parent pool $\mathcal{P}$. If fewer than two feasible parents exist, the iteration is skipped.

F.1.2 LLM-MEDIATED CROSSOVER

With probability p_c, two distinct parents $h_a, h_b \in \mathcal{P}$ are selected, and LLMCROSSOVER prompts the LLM to: (1) analyze each parent's distinct algorithmic approach, (2) identify complementary strengths, (3) synthesize a hybrid offspring that combines components in novel ways, and (4) introduce at least one enhancement not present in either parent. By recombining heterogeneous strategies, crossover broadens the heuristic space beyond single-parent variation. The LLM outputs a reasoning trace Th_{off} that records which parent contributed each component and why those components were combined.

F.1.3 LLM-MEDIATED MUTATION

With probability $p_m = 1 - p_c$, a single parent is sampled, and LLMMUTATE prompts the LLM to modify the heuristic through local code changes. Illustrative examples are provided for guidance: adaptive step-size tuning, Cauchy perturbation, and chaotic local search for continuous optimization; 2-opt/3-opt, swap, and inversion operators for combinatorial optimization. The LLM is not restricted to these named techniques and is instructed to focus on refining search mechanisms rather than merely tuning numerical parameters; this semantic-level guidance incurs additional token cost relative to random operators (see **Appendix G.3**).

F.2 METACOGNITIVE EVOLUTION DETAILS

F.2.1 CONVERGENCE-AWARE REFLECTION AND EVOLUTIONARY OPERATORS

The Meta Insight performs three tasks. *Convergence Diagnosis* classifies the search state (premature convergence, steady improvement, oscillation, or stagnation) by analyzing fitness trends in F and error patterns in ERR. *Limitation Analysis* identifies cognitive biases and repeated ineffective strategies. *Pathway Recommendation* proposes specific improvement pathways with named algorithmic techniques. These three tasks provide directed, history-conditioned guidance that complements the stochastic variation produced by crossover and mutation in Natural Evolution. Shared history (TH, F, ERR, h^*) carries reasoning traces and fitness outcomes across cycles.

G Extended Experimental Details

G.1 Experimental Settings and Supplementary Results

Challenges in Real-World Problems. A critical question is whether LLM-based AHD generalizes beyond classical benchmarks. We include two real-world problems, ACS and WSN, to test this: they demand reasoning about domain-specific constraints beyond generic algorithmic templates, testing whether AHD architectures remain effective when LLMs lack domain-specific knowledge. Existing architectures perform poorly here without structured problem analysis; incomplete problem descriptions yield near-zero runnable heuristics for architectures such as EoH and ReEvo (see **Appendix C.1**), making evolutionary search infeasible.

Appendix Table G4 summarizes the parameter settings for all five optimization problems. Settings for TSP-ACO, BPP-ACO and TSP-Construct follow previous studies (Ye et al., 2024; Zheng et al., 2025); ACS and WSN parameters were determined through extensive testing.

Table G4. Parameter settings for all five optimization problems.

Parameter	TSP-ACO	BPP-ACO	ACS	WSN	TSP-Construct
Initial Population N_0	20	20	20	20	20
Offspring per Gen. N_{pop}	10	10	10	10	10
Max Solutions L	100	50	50	50	100
Independent Runs	5	5	5	5	5
Problem-specific					
Number of items/cities n	50	500	–	–	50 / 100 / 200
Item demand d_i	–	Uniform$[20, 100]$	–	–	–
Bin capacity C	–	150	–	–	–
Materials	–	–	120	–	–
Concepts	–	–	20	–	–
Students	–	–	30	–	–
Number of CN	–	–	–	50	–
Number of SN	–	–	–	200	–
Capacity	–	–	–	15	–
Connection Distance	–	–	–	20	–
SN Connect Param.	–	–	–	-85	–
Path Loss Const. PL_0	–	–	–	55.0	–
γ_{PL} (Path loss exponent)	–	–	–	2.5	–
Search Agents	–	–	20†	50†	–
Max Iterations	–	–	50†	100†	–
Initial rg	–	–	2.5	2.5	–
Min rg	–	–	0.1	0.1	–

†For ACS and WSN, traditional baselines (ACO, Genetic Algorithm (GA), Particle Swarm Optimization (PSO), and Sand Cat Swarm Optimization (SCSO)) use increased budgets (ACS: 30 agents, 500 iterations; WSN: 100 agents, 1000 iterations), also independently run 5 times.

Appendix Table G5 lists the penalty coefficients and constraint parameters specific to the ACS and WSN problems.

Appendix Table G6 details the parameters of all baseline methods and AHD architectures.

Appendix Table G7 shows the LLM backbone configurations. All five problems are evaluated on both backbones.

For Genetic Programming (GP) in TSP-Construct, we implement tree-based genetic programming with ramped half-and-half initialization following prior work (Koza, 1994; Poli et al., 2007); all GP parameters are listed in **Appendix Table G6**. Briefly, a population of μ=1000 expression trees is evolved over G=200 generations, using tournament selection (k_{tour}=7), one-point subtree crossover (r_c=0.9), and uniform subtree mutation (r_m=0.1) (Poli et al., 2007). Tree depth is

Table G5. Constraint parameters for ACS and WSN problems.

Parameter	Description	Value
	ACS	
ε_1	Penalty factor for exceeding the number of concepts required for learning	1
ε_2	Penalty factor for not covering the number of learning concepts	10^4
ε_3	Penalty per time-bound violation (outside $[t_i^{\min}, t_i^{\max}]$)	1000
ω_1	Coefficient for ε_1	0.25
ω_2	Coefficient for ε_2	1
ε_4	Penalty factor for preference mismatch	0.25
ψ_1	High priority material quantity limit	3
ψ_2	Medium priority material quantity limit	6
ψ_3	Challenging material quantity limit	1
	WSN	
coverage	Penalty per uncovered sensor node	10
connectivity	Penalty for disconnected network	1000
power_std	Penalty for power standard deviation	100
β^x	Cross-midline x-fading (opposite sides of x=25)	8.0
β^y	Cross-midline y-fading (opposite sides of y=25)	20.0

Table G6. Parameters of Baseline Methods and AHD Architectures

Method	**Parameters**
	Baseline Methods
Greedy Construct	–
GP	μ=1000, G=200, r_c=0.9, r_m=0.1, k_{tour}=7, $D_{\max}$=17, $d_{\max}$=6, e=1, γ=0.005
ACO (TSP-ACO)	$n_{ants} = 30$, $n_{iter} = 100$, $\alpha_p = 1$, $\beta_h = 1$, $\rho = 0.9$
ACO (BPP-ACO)	$n_{ants} = 20$, $n_{iter} = 15$, $\alpha_p = 1$, $\beta_h = 1$, $\rho = 0.9$
SCSO	rG decays from 2 to 0
GA	crossover = 0.8, mutation = 0.1
PSO	$w = 0.7$, c_1=c_2=1.5
	AHD Architecture
Funsearch	$k = 2$
EoH	$p = 5$
ReEvo	crossover rate = 1, mutation rate = 0.5
MCTS-AHD	$NI = 4$, $H = 10$, $k = 2$, $p_0 \in \{2, 3, 4, 5\}$, $\lambda_0 = 0.1$, $\alpha_{MCTS} = 0.5$
HiFo-Prompt	Multiple parameters; see (Chen et al., 2025)
MeLA	$COR = 2$
MeEvo	$COR = 2$, $\alpha = 0.5$, $\beta = 0.2$, $K = 5$, $N = 2$, $M = 1$

Table G7. LLM backbone configurations. All five problems are evaluated on both backbones.

LLM Model	Temperature	Description
DeepSeek-V4-Flash	1.0	High exploration, diverse generation
MIMO-v2.5-Pro	0.5	Lower randomness, more stable code

bounded by $D_{\max}$=17 during evolution and $d_{\max}$=6 at initialization to curb bloat (Koza, 1994); one elite individual is preserved per generation (e=1), and a parsimony coefficient γ=0.005 penalizes tree size in the fitness function to favor compact expressions. The function set comprises $\{+, -, \times, \div, \max, \min, \sin, \cos, \sqrt{\cdot}, \text{neg}, \text{abs}\}$ with ephemeral random constants in $[-5, 5]$; the terminal set supplies eight real-valued TSP heuristic features (e.g., current-to-candidate distance, unvisited-city statistics, normalized path length). The population is evolved on 10 randomly sampled n=50 TSP training instances; the best-evolved individual is evaluated on 64 held-out test instances at each of $n \in \{50, 100, 200\}$.

POMO, GP, and HiFo-Prompt are evaluated only on TSP-Construct. POMO (Kwon et al., 2020) is a deep reinforcement learning (RL) model trained end-to-end for TSP city selection and cannot generate the heuristic matrices (TSP-ACO, BPP-ACO) or continuous search operators (ACS, WSN) required by the other four problems. GP (Koza, 1994) evolves symbolic expression trees designed for scalar constructive heuristics; adapting to matrix-valued or continuous-space outputs demands a fundamentally different function set. HiFo-Prompt (Chen et al., 2025) requires extensive per-problem manual parameter configuration (**Appendix Table G6**), rendering cross-problem deployment impractical.

Table G8. Best values across different architectures.

Method	TSP-ACO	BPP-ACO	ACS	WSN
	Best Obj.	Best Obj.	Best Obj.	Best Obj.
Funsearch	5.902	206.00	652.27	76.19
EoH	5.842	204.60	642.75	98.38
ReEvo	5.823	**204.20**	628.06	94.61
MCTS-AHD	5.792	204.40	642.50	76.10
MeLA	5.812	204.40	579.32	55.76
MeEvo (DeepSeek)	**5.712**	**204.20**	**568.38**	**50.00**
MeEvo (MIMO)	**5.712**	**204.20**	**564.44**	**50.00**

G.2 Parameter Sensitivity Analysis

We distinguish two parameter categories. α, β, and K are optimization hyperparameters internal to Natural Evolution; their sensitivity is examined below. N and M govern the frequency and allocation of the cyclically coupled search and reflection-guided generation phases and are ablated in Section 4.3 of the main paper to validate the architectural design. We evaluate variants on two representative problems, TSP-ACO (constrained combinatorial) and ACS (real-world, multi-constraint), using DeepSeek-V4-Flash, with all other parameters held at their default values (**Appendix Table G6**). Each configuration is repeated 5 times; we report the mean objective.

Appendix Table G9 reports the sensitivity of all three optimization hyperparameters. Panel (a) examines α and β. The default configuration (α=0.5, β=0.2) is compared against a fixed-probability baseline (β=0), wider amplitude (β=0.4), narrower amplitude (β=0.1), a higher baseline (α=0.6, β=0.1), and a reversed direction (α=0.7, $\beta = -0.2$, i.e., p_c increases over time). Panel (b) varies K which controls the selection pressure in Natural Evolution. The default (K=5) is compared against smaller pools ($K \in \{2, 3\}$) and larger pools ($K \in \{8, 10\}$).

The results in **Appendix Table G9** support three conclusions regarding the internal operator balance. First, the adaptive schedule outperforms a fixed crossover probability (β=0), confirming that adjusting the crossover-to-mutation ratio during Natural Evolution is beneficial. Second, β=0.2 provides the best balance: smaller adjustment is insufficient, whereas a wider adjustment over-oscillates and degrades ACS performance. Third, the baseline α=0.5 is important, because increasing it to 0.6 sharply degrades ACS performance (from 578.16 to 590.73); the reversed direction (α=0.7,

Table G9. Sensitivity analysis of optimization hyperparameters.

Panel A: Adaptive operator balance hyperparameters α and β				
(α, β)	p_c Range	TSP-ACO Obj. (↓)	ACS Obj. (↓)	Notes
$(0.5, 0.0)$	[0.5, 0.5]	5.736	584.74	Fixed (no adaptation)
$(0.5, 0.1)$	[0.5, 0.6]	5.728	579.42	Narrow amplitude
$(0.5, 0.4)$	[0.5, 0.9]	5.730	579.12	Wide amplitude
$(0.6, 0.1)$	[0.6, 0.7]	5.742	590.73	Higher baseline
$(0.7, -0.2)$	[0.5, 0.7]	5.724	582.18	Reversed ($p_c \uparrow$)
$(0.5, \mathbf{0.2})$	[0.5, 0.7]	**5.716**	**578.16**	**Default**
Panel B: Parent pool size K				
K	—	TSP-ACO Obj. (↓)	ACS Obj. (↓)	Notes
2	—	5.772	591.83	Minimum viable
3	—	5.754	584.15	Small pool
8	—	5.724	582.83	Large pool
10	—	5.793	602.43	Very large pool
5	—	**5.716**	**578.16**	**Default**

$\beta = -0.2$) also underperforms the default. Thus, Natural Evolution supplies macro-level exploration while maintaining its own adaptive operator balance, and Metacognitive Evolution supplies macro-level exploitation through reflection.

Regarding parent pool size (Panel B), **Appendix Table G9** reveals a U-shaped sensitivity. $K{=}2$ underperforms due to insufficient genetic material for meaningful crossover (TSP rises to 5.772, ACS to 591.83). $K{=}10$ performs worst overall (TSP: 5.793, ACS: 602.43), as a large pool admits lower-fitness individuals that dilute selection pressure and steer the LLM toward weaker design patterns. $K{=}5$ achieves the optimal trade-off: it maintains sufficient diversity for effective recombination without relaxing selection pressure. This is consistent with truncation selection theory in evolutionary computation, where moderate selection pressure balances exploration and exploitation.

G.3 Computational Cost Analysis

Appendix Table G10 reports approximate per-generation token consumption and wall-clock time for each method using DeepSeek-V4-Flash. The token accounting includes LLM calls used for error diagnosis and repair; the corresponding replacement evaluations are excluded from the primary budget L. Mean and standard deviation are computed over the two generation budgets ($L \in \{50, 100\}$), and $\sim$ denotes values derived from approximate measurements. MeEvo uses approximately 65,000 tokens and 5.3 seconds per generation. Although this is the highest mean token cost, it is only 5% above ReEvo (62,000) and 8% above MCTS-AHD (60,000), while its standard deviation (9,000) is substantially lower than those of ReEvo (37,000), MCTS-AHD (23,000), and MeLA (23,000).

The corresponding DeepSeek-V4-Flash results in **Table 1** show that this overhead has different value across tasks. On the comparatively simple downstream-solver problems, TSP-ACO and BPP-ACO, MeEvo's advantage over MeLA is small (the latter has gaps of 1.96% and 0.25%, respectively). On the more constrained ACS and WSN problems, however, MeEvo obtains the lowest mean objectives and improves on MeLA by 1.79% and 14.96%, respectively. Thus, the additional per-generation cost is best interpreted as a task-dependent quality–cost trade-off: it is most justified when history-conditioned reflection produces meaningful gains on the harder constrained problems, rather than as a universal efficiency advantage.

Table G10. Per-generation LLM token cost and wall-clock time (*DeepSeek-V4-Flash*). Mean and standard deviation are computed over $L \in \{50, 100\}$; $\sim$ denotes values derived from approximate measurements.

Method	Mean	Std.	Time (s)
FunSearch	$\sim$15000	$\sim$1240	$\sim$1.5
EoH	$\sim$16000	$\sim$5800	$\sim$1.6
ReEvo	$\sim$62000	$\sim$37000	$\sim$2.3
MCTS-AHD	$\sim$60000	$\sim$23000	$\sim$3.3
MeLA	$\sim$45000	$\sim$23000	$\sim$4.2
MeEvo	$\sim$65000	$\sim$9000	$\sim$5.3

G.4 Statistical Significance Analysis

We assess statistical significance using the one-sided Mann–Whitney U test (Mann & Whitney, 1947), comparing MeEvo against the second-best architecture within each LLM backbone on four problems (n=5 each; **Appendix Table G11**). TSP-Construct is excluded because inter-architecture gaps are narrow (0.5–1.0 pp; **Table 2** of the main paper) and n=5 cannot separate architectures under these conditions.

Table G11. Mann–Whitney U Test: MeEvo vs. Second-Best Baseline (per LLM backbone)

Problem	2nd-Best Baseline	**MeEvo** Mean(↓) ± Std(↓)	**Baseline** Mean(↓) ± Std(↓)	U Stat.	p-value[†]
LLM backbone: *DeepSeek-V4-Flash*					
TSP-ACO	MCTS-AHD	5.716 ± 0.003	5.812 ± 0.016	0	0.004
BPP-ACO	MeLA	204.28 ± 0.10	204.80 ± 0.28	1	0.008
ACS	MeLA	578.16 ± 5.21	588.50 ± 9.07	3	0.028
WSN	MeLA	50.80 ± 0.76	58.40 ± 2.02	0	0.004
LLM backbone: *MIMO-v2.5-Pro*					
TSP-ACO	MCTS-AHD	5.718 ± 0.010	5.807 ± 0.010	0	0.004
BPP-ACO	MCTS-AHD	204.40 ± 0.24	204.44 ± 0.035	10	0.345
ACS	MeLA	585.94 ± 11.14	609.47 ± 15.21	2	0.016
WSN	MeLA	52.00 ± 2.26	59.37 ± 3.16	0	0.004

[†]One-sided exact p-value. With n_1=n_2=5, the null distribution of U is discrete on $\binom{10}{5}$=252 equally likely configurations; U=0 indicates complete separation (all 5 MeEvo runs below all 5 baseline runs).

MeEvo achieves significant improvement in 7 of 8 comparisons at α=0.05 (**Appendix Table G11**), with complete separation (U=0, p=0.004) in four cases. The only non-significant result is BPP-ACO with MIMO (p=0.345), where the strong ACO structure leaves limited room for differentiation. These results confirm that MeEvo's advantage is systematic.