EvoX MetaBench: Composable Benchmarking for Meta-Black-Box Optimization at Scale

Anonymous authors
Paper under double-blind review

Abstract

Meta-black-box optimization (MetaBBO) automates optimizer design through learning or design search. However, method-specific implementations often couple components, which complicates controlled comparisons of their effects and interactions. Such comparisons also incur substantial computational costs from repeated optimization runs across problems and random seeds. To address these challenges, we introduce **`EvoX MetaBench`**, a composable framework and benchmark for MetaBBO at scale. Its task-specific interfaces support component assembly for algorithm selection, algorithm configuration, solution manipulation, and algorithm generation. Researchers can build custom workflows or instantiate baseline recipes assembled from the framework's components, then substitute and recombine compatible components under shared experimental conditions. The benchmark provides over 11,000 problem instances and instance-generation recipes across numerical, combinatorial, and real-world optimization, with explicit training, validation, and test manifests for reproducible comparisons. To support these workflows at scale, the framework builds on EvoX to batch optimization runs on GPUs. Experiments show that GPU execution accelerates LDE training and testing by up to $107.6\times$ and $174.9\times$, respectively, compared with CPU execution. Component studies show that configuration gains depend on population size, useful state information depends on the optimizer, and the link between parent and offspring quality depends on the generation operator.

1 Introduction

Black-box optimization (BBO) seeks high-quality solutions through objective evaluations. The No Free Lunch (NFL) theorems motivate matching search strategies to target problem distributions (Wolpert et al., 1995). Meta-black-box optimization (MetaBBO) (Ma et al., 2026c; Yang et al., 2025a) automates this design process through learning or design search across four tasks: Algorithm Selection (AS) (Wu et al., 2024; Guo et al., 2024), Algorithm Configuration (AC) (Sharma et al., 2019; Xue et al., 2022; Pei et al., 2025; Xu et al., 2026), Solution Manipulation (SM) (Li et al., 2024; Han et al., 2025; Wang et al., 2026a;b), and Algorithm Generation (AG) (Liu et al., 2026; 2024a;b; Stein & Bäck, 2025; Romera-Paredes et al., 2024; Zhang et al., 2026). The performance of these methods depends jointly on state representations, decision mechanisms, underlying optimization procedures, and learning or design-search strategies. Deriving reusable guidance for optimizer design therefore requires understanding which choices contribute to performance, how they interact, and when their benefits transfer across optimizers and problem distributions.

Understanding these component effects and interactions requires controlled comparisons. Platforms such as MetaBox and MetaBox-v2 (Ma et al., 2023; 2025b) provide shared interfaces and protocols for developing and comparing MetaBBO methods. However, overall performance alone cannot isolate component effects when methods differ in several design choices. For example, testing whether the benefit of a state representation (Ma et al., 2025a) depends on the optimizer requires varying their pairing while holding other design choices fixed. Reusable component interfaces can support such substitutions and recombinations under shared evaluation conditions and make these studies a routine part of method development and benchmarking.

Systematic component studies require both task-specific composition and efficient execution. The four MetaBBO tasks differ in their decision objects and execution lifecycles, yet methods within

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.

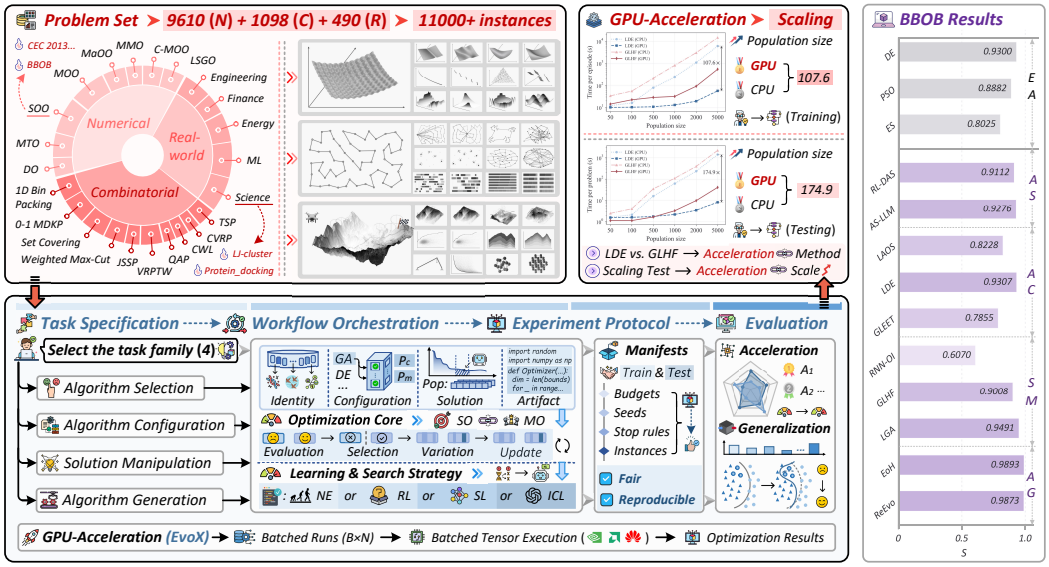

Figure 1: Overview of **`EvoX MetaBench`**. Diverse problem collections support four MetaBBO tasks through composable optimization components and learning strategies. Shared experimental protocols organize training and evaluation, while GPU acceleration enables batched execution. The accompanying results show training and testing speedups across execution configurations and normalized improvement S for selected baselines on 16 BBOB functions.

the same task often share similar workflow structures. These common structures therefore provide a basis for exchanging and recombining components while preserving their task-specific roles. Exploring the resulting designs across optimizers and problem distributions also requires repeated optimization runs during learning, design search, and evaluation (Huang et al., 2025; Tang et al., 2022; Lange, 2023).

To meet these needs, we introduce **`EvoX MetaBench`**, a composable framework and benchmark for fine-grained analysis and design of MetaBBO methods. Its task-specific interfaces allow researchers to assemble custom workflows or use predefined baseline recipes as starting points for comparison and further development. Both routes use shared problem collections and experimental protocols for complete-method comparisons and controlled component studies across numerical, combinatorial, and real-world optimization. The framework builds on EvoX (Huang et al., 2025; Liang et al., 2026; Zheng et al., 2025a) to execute batches of optimization runs on GPUs. Figure 1 summarizes the framework and representative experimental results. Our contributions are threefold:

- **A composable MetaBBO framework.** We define task-specific component interfaces for AS, AC, SM, and AG that support workflow assembly and controlled substitution and recombination of compatible components for method development and fine-grained analysis.
- **Benchmark resources and scalable execution.** We provide shared problem collections, component-assembled baseline recipes, and explicit task manifests under common experimental protocols. Batched GPU execution through EvoX supports repeated optimization runs at scale.
- **Empirical insights into MetaBBO design.** We benchmark representative MetaBBO methods under shared protocols for performance, efficiency, and generalization. Our analysis relates GPU speedups to method design and workload size, and examines transfer to application problems. Controlled studies provide guidance on configuration granularity, state representation, and operator-dependent generation value.

2 Related Work

Meta Black-Box Optimization. MetaBBO automates optimizer design through learning or design search over a distribution of optimization problems. Given a problem distribution $\mathcal{P}$ and a task-

dependent design space Ω, a meta-level mechanism $\mathcal{M}$ maps a task or search context s to a decision $\omega = \mathcal{M}(s) \in \Omega$, whose quality is assessed through the resulting optimization performance (Yang et al., 2025a; Ma et al., 2026c; Liu et al., 2026). Figure 2 illustrates this general workflow. Existing methods can be organized by the object represented by ω. AS chooses complete optimizers or makes portfolio decisions (Kerschke et al., 2019; Tian et al., 2020; Guo et al., 2024; Zhu et al., 2024; Cenikj et al., 2024; Wu et al., 2024); AC controls the parameters, operators, or strategies of a prescribed optimizer (Karafotias et al., 2015; Sharma et al., 2019; Ma et al., 2026a; Tan & Li, 2021; Guo et al., 2025; Zhang & Yi, 2025); SM directly generates, transforms, or selects candidate solutions, populations, or search distributions (Chen et al., 2017; TV et al., 2019; Wang et al., 2026a; Lange et al., 2023a; Li et al., 2024; Yang et al., 2024a; Liu et al., 2024c); and AG produces an algorithm artifact, such as an update rule, algorithm structure, or executable program (Romera-Paredes et al., 2024; Liu et al., 2023; 2024a; Ye et al., 2024; Zheng et al., 2025b; Stein & Bäck, 2025; Chen et al., 2024). Reinforcement Learning (RL) (Li et al., 2025a), Supervised Learning (SL) (Li et al., 2024), Neuroevolution (NE) (Lange et al., 2023a), and In-Context Learning (ICL) (Yang et al., 2024a) provide a complementary classification according to how $\mathcal{M}$ is obtained or adapted.

Despite their different decision objects, MetaBBO methods share a coupled design structure. Their performance depends jointly on the representation of search information, the decision mechanism, the underlying optimizer, and the learning or design-search procedure. Existing methods commonly develop these elements as an integrated system, so complete-method comparisons can vary several design choices simultaneously. Consequently, such comparisons cannot reliably attribute a performance difference to an individual representation, controller, operator, or training rule. A component-level benchmark must therefore expose these elements as independently replaceable units while preserving the decision semantics and execution lifecycle of each MetaBBO task.

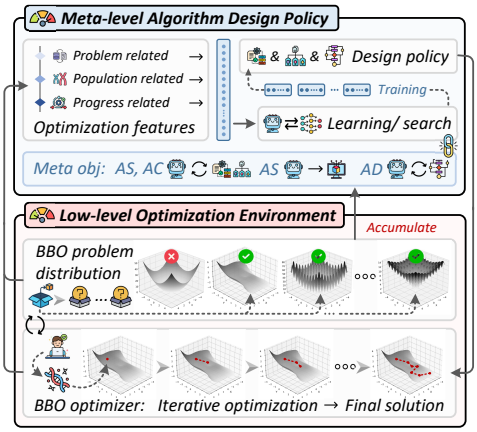

Figure 2: General framework and optimization workflow of MetaBBO.

Benchmarks and Platforms. Existing BBO resources provide complementary foundations for problem construction, evaluation, and scalable execution. BBOB (Hansen et al., 2021) supplies controlled numerical landscapes, while HPO-B (Pineda Arango et al., 2021) provides data and evaluation protocols for hyperparameter optimization. MA-BBOB (Vermetten et al., 2023) extends task construction through affine combinations of BBOB functions, whereas EoB (Wang et al., 2026c) generates objective-function programs through LLM-guided evolution. Complementing these problem resources, COCO (Hansen et al., 2021) standardizes optimizer assessment, PyPop7 (Duan et al., 2024) provides implementations of population-based optimizers, and EvoX (Huang et al., 2025) supports distributed GPU execution. Together, these resources improve the reproducibility and scalability of BBO experiments, but they do not define the nested learning and design-search workflows required by MetaBBO.

MetaBBO platforms extend these foundations from optimizer evaluation to method development and training. Table 1 summarizes representative platforms. PlatMetaX (Yang et al., 2025b) primarily supports RL-based algorithm configuration, while LLM4AD (Liu et al., 2024b) focuses on LLM-based algorithm generation. MetaBox (Ma et al., 2023) provides Agent/Optimizer templates, baseline implementations, and automated training and evaluation workflows, and MetaBox-v2 (Ma et al., 2025b) broadens this coverage across RL, SL, NE, and ICL with parallel training and distributed testing. However, these interfaces primarily integrate complete agents, optimizers, or method-specific templates, within which state construction, action interpretation, and optimization logic remain coupled. Fine-grained replacement and recombination can therefore require coordinated changes to method-specific implementations. **`EvoX MetaBench`** targets this missing level of abstraction through task-specific component interfaces and explicit compatibility rules. As a result, components and their compatible combinations become the units of method construction and controlled benchmarking under shared experimental protocols.

Table 1: Comparison of representative BBO and MetaBBO platforms. Problem counts report collection sizes, including tasks and instance recipes for **`EvoX MetaBench`**. *# Classic* and *# MetaBBO* count built-in methods. Composability distinguishes coupled pipelines from independently replaceable MetaBBO components.

	Resources			Execution and Composition		Task Coverage			
Platform	# Problems	# Classic	# MetaBBO	Parallelism	Composability	AS	AC	SM	AG
COCO	481	0	–	–	–	–	–	–	–
PyPop7	103	70+	–	–	–	–	–	–	–
EvoX	100+	50+	–	GPU	–	–	–	–	–
MetaBox-v1	334	11	8	–	Coupled	0	7	1	0
MetaBox-v2	1,934	13	23	MP, Ray	Coupled	1	16	5	1
`EvoX MetaBench`	**11,000+**	**50+**	**31**	**GPU**	**Decoupled**	**4**	**9**	**8**	**10**

3 EvoX MetaBench

3.1 Composable MetaBBO Workflows

`EvoX MetaBench` organizes MetaBBO methods as compositions of task-specific components connected through explicit interfaces. Researchers can select problem sets, assemble compatible components, and choose an appropriate learning or design-search procedure to construct complete workflows for method development and fine-grained analysis. They can also instantiate predefined baseline recipes for comparative evaluation and use these assemblies as starting points for component substitution. Figure 3 illustrates these two research workflows.

The framework provides built-in, task-specific component libraries for assembling custom methods. Taking AC as an example, researchers select a tensorized optimizer and specify the parameters or operators to control, such as the mutation strategy, scale factor, and crossover rate in DE. The state and reward libraries specify the controller's observations and optimization feedback. The former supports representations composed from over 20 feature components covering fitness, solutions, and the optimization process, while the latter provides commonly used reward formulations for evaluating progress. Controller choices span RL, ES, and LLMs. For example, the framework provides eight native RL implementations, including DQN variants, PPO, DDPG, and TD3. A configuration adapter maps controller outputs to optimizer configurations at the population or individual level. These components together define a complete method that runs within an automated workflow for training or inference. Corresponding libraries support the other tasks through optimizer portfolios and selectors for AS, differentiable mutation and crossover modules drawn from methods such as GLHF and B2Opt for SM, and generation and evolutionary search components from methods such as EoH and ReEvo for AG. Appendix B details these libraries, their assembly procedures, and the learning or design-search paradigms supported by each task.

Composable baselines. **`EvoX MetaBench`** implements 31 baselines as concrete examples of component assembly. Each baseline is assembled from the framework's components and can be instantiated through a predefined recipe without additional implementation code. Each recipe specifies the component choices and the associated learning or design-search procedure. Researchers can use these recipes for comparative evaluation on compatible problem sets or develop new methods by reusing, replacing, or extending selected components within the same workflow. The recipes cover all four MetaBBO tasks: **(i) AS:** RL-DAS (Guo et al., 2024), HHRL-MAR (Zhu et al., 2024), TransOptAS (Cenikj et al., 2024), and AS-LLM (Wu et al., 2024; 2025). **(ii) AC:** DE-DQN (Tan & Li, 2021), DE-DDQN (Sharma et al., 2019), LDE (Sun et al., 2021), RLEPSO (Yin et al., 2021), GLEET (Ma et al., 2024), Ada-UCDE (Zhang et al., 2023), LES (Lange et al., 2023b), LLMCES (Custode et al., 2024), and LAOS (Zhang & Yi, 2025). **(iii) SM:** RNN-OI (Chen et al., 2017), RNN-Opt (TV et al., 2019), LGA (Lange et al., 2023a), GLHF (Li et al., 2024), B2Opt (Li et al., 2025b), EPOM (Han et al., 2025), OPRO (Yang et al., 2024a), and LMEA (Liu et al., 2024c). **(iv) AG:** FunSearch (Romera-Paredes et al., 2024), AEL (Liu et al., 2023), EoH (Liu et al., 2024a), ReEvo (Ye et al., 2024), LLaMEA (Stein & Bäck, 2025), LLMEPS (Zhang et al., 2024a), MEoH (Yao et al., 2025), MCTS-AHD (Zheng et al., 2025b), LHNS (Xie et al., 2025), and PartEvo (Hu & Zhang, 2025). The framework also provides access to EvoX's tensorized evolution-

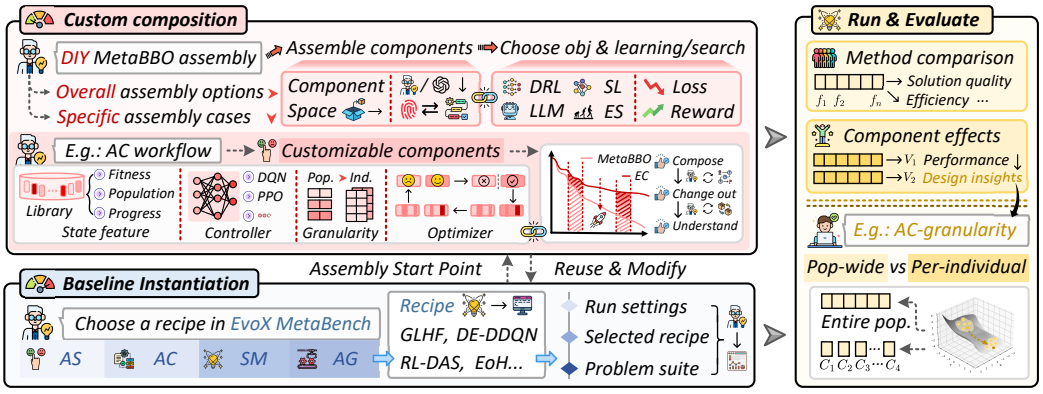

Figure 3: **Two research workflows in `EvoX MetaBench`. Custom composition** combines compatible components, objectives, and learning or design-search procedures for method development and component analysis. **Baseline instantiation** provides predefined recipes for comparative evaluation, which also serve as starting points for component substitution.

ary algorithms for single- and multi-objective optimization as reusable building blocks for MetaBBO methods. Appendix C provides implementation details and technical summaries of the MetaBBO baselines.

Evaluation and analysis. Researchers can use EvoX monitors to record solution and fitness histories, track best-so-far performance, and collect algorithm-provided auxiliary information. These records support convergence analysis, studies of component behavior, and researcher-defined metrics. To compare single-objective minimization performance across problem instances with different objective scales, we use the mean normalized improvement S relative to a fixed reference pool:

$$S = \frac{1}{KR} \sum_{i=1}^{K} \sum_{r=1}^{R} \frac{f_{i,r}^{(0)} - f_{i,r}^{\text{best}}}{f_{i,r}^{(0)} - f_{i}^{\text{ref}}}, \tag{1}$$

where K is the number of problem instances and R is the number of independent runs per instance. The initial best-fitness reference $f_{i,r}^{(0)}$ is shared across methods, and $f_{i,r}^{\text{best}}$ is the best objective value attained in a run. For each instance i, f_i^{ref} is the lowest objective value observed across the reference methods and their prescribed evaluation runs under the same function-evaluation budget. The reference pool and the values of $f_{i,r}^{(0)}$ and f_i^{ref} are fixed for each evaluation protocol and retained when additional methods are evaluated under that protocol. Higher scores indicate greater normalized improvement relative to the fixed reference.

3.2 Problem Collection and Task Composition

`EvoX MetaBench` provides a shared foundation for MetaBBO development and comparison through over 11,000 problem instances and instance-generation recipes across three categories. *Numerical benchmarks* provide controlled landscapes for single- (Hansen et al., 2021; Ahrari et al., 2022), multi- (Huband et al., 2005; 2006), and many-objective (Cheng et al., 2017) optimization across constrained (Kumar et al., 2021), large-scale (Omidvar, 2007), multimodal (Li et al., 2013), multitask (Da et al., 2017), and dynamic settings (Yazdani et al., 2022). *Combinatorial problems* involve structured discrete decisions in routing (Reinelt, 1991), scheduling (Solomon, 1987), packing (Falkenauer, 1996), and related domains (Chu & Beasley, 1998; Burkard et al., 1997; Goemans & Williamson, 1995). *Real-world problems* derive their objectives and constraints from domain models, datasets, simulators, or interactive environments in engineering (Hansen et al., 2021; Xue et al., 2026), energy (Sinha et al., 2003), finance (Metaxiotis & Liagkouras, 2012), science (Doye et al., 1999; Shehadeh & Kůdela, 2025), and machine learning (Pineda Arango et al., 2021; Brockman et al., 2016). The left panel of Figure 4 summarizes the suites and their sizes; Appendix D provides suite-level details.

Category	*Size*	*Collections*
Numerical	9,610+	BBOB, CEC, DTLZ, UF, ZDT, WFG, ...
Combinatorial	1,098	TSP/VRP, JSSP, Knapsack, Max-Cut, ...
Real-world	490+	HPO, NE, UAV, Protein, ...

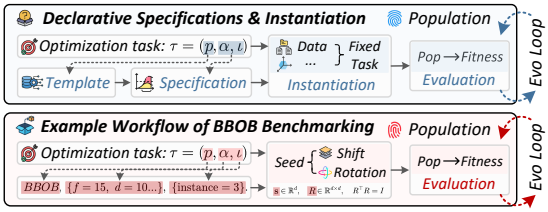

Figure 4: **Problem collection and task instantiation. Left:** Problem suites and collection sizes across numerical, combinatorial, and real-world optimization. **Right:** Instantiation of a task specification into a concrete problem instance, illustrated with BBOB, where the function template, dimension, and official instance identifier jointly determine the landscape.

We describe each task by $\tau = (p, \alpha, \iota)$, where p denotes a problem template and α specifies its structural and evaluation configuration, including the decision space, dimension, objectives, and constraints. The instance descriptor ι provides information such as an official identifier, landscape transformations, or a data and scenario configuration. A family-specific procedure maps this specification to a concrete problem instance with its native decision representation and evaluation semantics. For BBOB, the function template, dimension, and official instance identifier jointly determine the function-specific landscape transformations and objective bias. For a graph-based combinatorial problem, the instance descriptor specifies the graph structure and associated costs. The right panel of Figure 4 illustrates this specification-to-instance process using BBOB.

We organize these task specifications into training, validation, and test manifests to support learning, design search, and evaluation across the four MetaBBO tasks:

$$\mathcal{M}_s = \text{Compose}\left(\mathcal{B}; \mathcal{C}_s\right), \qquad s \in \{\text{train}, \text{val}, \text{test}\}. \tag{2}$$

Here, $\mathcal{B}$ denotes the problem collection, and $\mathcal{C}_s$ specifies the problem templates, structural configurations, instance descriptors, and partition rules used to construct $\mathcal{M}_s$. Each manifest lists task specifications $\tau = (p, \alpha, \iota)$. For example, training and test manifests can use disjoint BBOB function templates or disjoint combinations of objective count and dimension in DTLZ. These choices support evaluation on held-out functions or structural configurations, respectively. Learning and design-search procedures draw tasks from $\mathcal{M}_{\text{train}}$, while validation and test manifests remain fixed for reproducible comparison. Tasks selected from these manifests are instantiated through their family-specific procedures. Appendix D provides a concrete BBOB example of manifest construction and task instantiation.

3.3 GPU-Accelerated Execution with EvoX

MetaBBO training and evaluation require repeated optimization runs across problems and random seeds. **`EvoX MetaBench`** batches these runs in tensor-based workflows. For example, a batch of B runs on the same problem instance with different random seeds and N individuals per run allows objective evaluation over BN candidate solutions. Batching also covers feature construction, controller decisions, and population updates. The assembled components therefore advance multiple runs through GPU tensor operations. EvoX (Huang et al., 2025) provides the tensorized optimizers, problem interfaces, and vectorization utilities underlying this execution. **`EvoX MetaBench`** integrates these capabilities into task-specific component assemblies and their learning, design-search, and evaluation procedures. Components with compatible batched inputs and outputs can reuse the corresponding execution routines when substituted or recombined. Baseline recipes and custom assemblies thus share computational infrastructure for complete-method comparisons and controlled component studies.

4 Experiments

We evaluate **`EvoX MetaBench`** through comparisons of complete methods and controlled component analyses. The experiments address five research questions: **RQ1** examines CPU and GPU efficiency and scaling behavior in training and testing; **RQ2** compares BBOB optimization performance and evaluates transfer to real-world problems; **RQ3** studies how configuration granularity

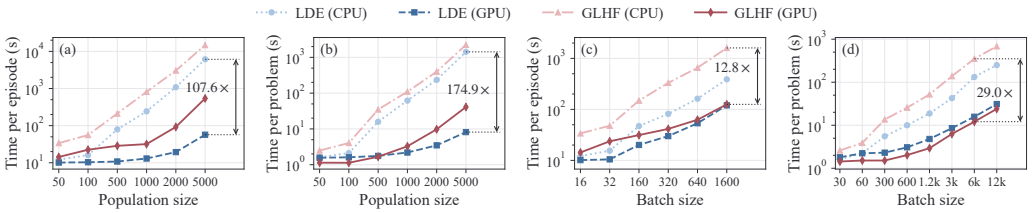

Figure 5: CPU and GPU runtime scaling of LDE and GLHF. (a,b) Training and testing with varying population size; (c,d) training and testing with varying batch size at $N = 50$. Training reports mean time per episode, while testing reports total time per problem for the full batch of runs.

affects low-level optimization performance and the additional gains from meta-training in AC; **RQ4** examines how state information sources interact when combined and the role of representation learning; and **RQ5** investigates the relationship between parent performance and generation value across operators in AG. We first introduce the experimental settings and then address **RQ1–RQ5**.

4.1 Experimental Setup

Baselines. From the baseline collection of **`EvoX MetaBench`**, we select methods representing different learning paradigms within AS, AC, and SM, and different design-search strategies within AG. The evaluated methods comprise RL-DAS, HHRL-MAR, and AS-LLM for AS; DE-DQN, DE-DDQN, LDE, GLEET, and LAOS for AC; RNN-OI, RNN-Opt, LGA, GLHF, and OPRO for SM; and EoH, ReEvo, MCTS-AHD, and LLMEPS for AG. Descriptions of these methods and their component configurations are provided in Appendix C.

Training and Testing. Methods requiring parameter learning are trained on eight 40-dimensional BBOB functions for 100 episodes. The trained methods are evaluated on the remaining 16 functions; selected methods are also tested on the UAV, HPO-B, and LJ Cluster problems. RQ2 uses 10,000 function evaluations per test run; RQ3 and RQ4 use 200 generations per run during training and testing. These comparisons share 30 test seeds. All LLM-based baselines use GLM-5.3-Flash (GLM-5-Team et al., 2026), with the same number of API calls allocated to each AG method for RQ2 design search. Experiments are conducted on a system equipped with two Intel Xeon Platinum 8255C CPUs at 2.50 GHz and an NVIDIA GeForce RTX 2080 Ti GPU. Detailed protocols for RQ2–RQ5 are provided in Appendices E–H, respectively.

4.2 Computational Efficiency and Scaling (RQ1)

We compare CPU and single-GPU runtimes for LDE (RL) and GLHF (SL) across population sizes N and batch sizes B. Population sweeps cover $N = 50$–5000, with $B = 16$ during training and 30 runs per problem during testing. Batch sweeps fix $N = 50$ and cover $B = 16$–1600 for training and $B = 30$–12000 for testing. We report mean time per training episode and total time per test problem; speedups are CPU/GPU runtime ratios.

Figure 5 reveals three patterns. **(1)** GPU acceleration generally increases with population size. At $N = 5000$, LDE achieves $107.6\times$ training and $174.9\times$ testing speedups. **(2)** Larger batches also improve acceleration, although testing gains eventually level off. GLHF reaches $12.8\times$ training acceleration at $B = 1600$ and $29.0\times$ testing acceleration at $B = 6000$; doubling the latter batch size leaves the speedup nearly unchanged. **(3)** Training acceleration also depends on the learning paradigm, as optimization rollouts and model updates offer different opportunities for parallel execution. These findings motivate matching population and batch sizes to each method's computational structure when scaling MetaBBO training and evaluation.

4.3 Optimization Performance and Generalization (RQ2)

We compare 20 methods on 16 BBOB functions, with results grouped by MetaBBO task and evaluation protocol. Figure 6 (left) reports AS, AC, and SM performance alongside classical baselines. HHRL-MAR, LDE, and LGA achieve the highest observed S within AS, AC, and SM, respectively. Figure 6 (right) reports AG performance after target-specific algorithm design, with the

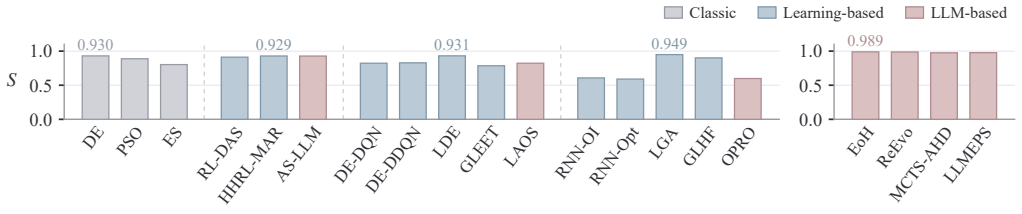

Figure 6: Mean normalized improvement S on 16 BBOB test functions. Left: AS, AC, and SM methods with classical baselines. Right: AG methods with target-specific algorithm design. All scores use the same fixed reference values. Colors distinguish classical, learning-based, and LLM-based methods.

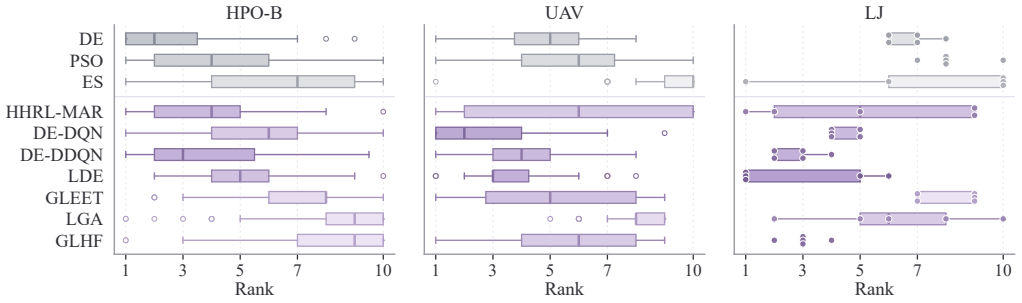

Figure 7: Real-world performance ranked by mean final fitness over 30 runs per instance. Lower ranks indicate better performance. Darker shades indicate better median ranks.

selected algorithms frozen and evaluated on independent test instances of the corresponding functions. EoH achieves the highest observed S among the four AG methods. To examine transfer to real-world problems, we evaluate selected trained methods on HPO-B, UAV, and LJ alongside classical baselines (Figure 7). LGA's leading performance within SM on BBOB does not translate into consistently strong rankings on these problems: its median ranks on HPO-B and UAV are 9 and 8, compared with 5 and 3 for LDE. These results indicate that relative performance on synthetic benchmarks may not persist across application domains. Complete experimental protocols and BBOB results are provided in Appendix E.

4.4 In-depth Analysis (RQ3–RQ5)

Final optimization performance in a bi-level MetaBBO workflow reflects both low-level search and meta-level learning or design search. Comparisons between complete methods can therefore obscure the effects of individual design choices. We use the composability of **`EvoX MetaBench`** to examine these choices through controlled comparisons. Our representative studies investigate configuration granularity and state representations in learning-based AC, and the relationship between algorithm performance and generation value in LLM-based AG.

Configuration Granularity (RQ3). We compare population-shared and per-individual configuration in DE, with learned (Meta) or uniformly random (Classic) configuration selection. The learned variants use DDQN with individual-level states and are trained separately at each population size $N \in \{20, 50, 100, 200\}$ under a 100-episode training budget. To examine learned assignment, we also evaluate Meta-Random on selected BBOB instances. This variant randomly permutes Meta-Ind's proposed configurations across individuals at test time. These comparisons yield three observations. **(1)** Classic-Ind achieves higher S than Classic-Pop at three of the four population sizes (Table 2). This indicates that configuration heterogeneity can improve search without learning. **(2)** Meta-Ind outperforms Meta-Pop and both Classic variants at $N = 20$ and $N = 50$, but does not retain this lead at the two larger population sizes under the same training budget. This suggests considering population size and controller training

Table 2: Configuration granularity results (S) on BBOB. Column bests are shaded and bold; runners-up are underlined.

Method	$N = 20$	$N = 50$	$N = 100$	$N = 200$
Classic-Pop	0.8978	0.8899	0.8437	<u>0.8485</u>
Classic-Ind	0.8973	<u>0.8938</u>	0.8513	**0.8594**
Meta-Pop	<u>0.9125</u>	0.8231	**0.9292**	0.7697
Meta-Ind	**0.9425**	**0.9384**	<u>0.9216</u>	0.8408

budget jointly when developing individual-level control. **(3)** Meta-Ind achieves lower mean final fitness than Meta-Random in seven of eight settings on f_{12} and f_{14}. These results support learned individual-specific assignment. Appendix F provides the full motivation, experimental settings, and detailed analysis.

State Representation (RQ4). We compare state representations for DE, PSO, and ES under DDQN control at $N = 50$. We group ten frequently used features into fitness-space ($\mathbf{F}$), solution-space ($\mathbf{X}$), and process ($\mathbf{P}$) information. The comparison covers all seven non-empty combinations of these groups alongside the frozen pretrained NeurELA (Ma et al., 2025a). All variants follow the same protocol for controller training and evaluation within each optimizer.

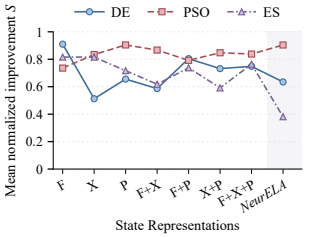

Figure 8: State representations in AC. Mean normalized improvement S over BBOB functions.

The usefulness of state information varies across optimizers (Figure 8). $\mathbf{F}$ leads for DE, $\mathbf{P}$ for PSO, and $\mathbf{F}$ and $\mathbf{X}$ achieve nearly equal scores for ES. Combining all three groups yields lower S than the best single group for each optimizer. Thus, broader information coverage alone does not ensure better control. The relative performance of NeurELA also varies across optimizers: it is competitive with $\mathbf{P}$ for PSO but trails the best handcrafted states for DE and ES. These findings motivate two directions for feature composition and representation learning: matching state information to the optimizer's search mechanism and identifying signals that remain useful across optimizers. Appendix G provides the full motivation, experimental settings, and detailed analysis.

Generation Value (RQ5). We examine whether stronger parent algorithms yield better offspring in LLM-based AG on TSP50. We cross rank-weighted and uniform parent selection with local modification and structural rewriting in design searches with equal proposal budgets. For fixed parents sampled from these searches, each operator generates repeated offspring batches of equal size. Within each batch, we select the best offspring on design instances and evaluate it on held-out test instances. We estimate generation value by averaging these selected offspring's test tour lengths across batches; lower values indicate better offspring. These comparisons yield two findings. **(1)** Parent performance correlates strongly with generation value under local modification (Spearman $\rho = 0.947$), but the association is much weaker under structural rewriting ($\rho = 0.278$; Figure 9). The usefulness of parent performance as a signal of generation potential therefore depends on the operator. **(2)** Despite the strong association under local modification, rank-weighted selection does not consistently outperform uniform selection in complete design searches.

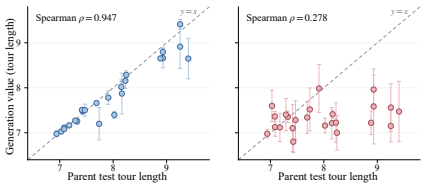

Figure 9: Parent performance and generation value on TSP50 under local modification (left) and structural rewriting (right).

LLM-based operators incur inference costs before candidate evaluation. These findings therefore motivate inexpensive, operator-specific estimates of generation value to guide parent selection. Appendix H provides the full motivation, experimental settings, and detailed analysis.

5 Conclusion

We introduced **`EvoX MetaBench`**, a composable framework and benchmark for MetaBBO analysis and design. The framework combines task-specific workflows, shared experimental protocols, and batched GPU execution to support complete-method comparisons and controlled component studies at scale. Our experiments provide three insights for MetaBBO development. **(1)** GPU speedups depend on both workload size and method design, so execution strategies should match the computational structure of each method. **(2)** Relative performance changes across BBOB and application problems, which motivates validation on target problem families. **(3)** Controlled component studies distinguish the contributions of configuration heterogeneity and learned assignment. They further motivate matching state information to the optimizer and assessing generation value under the intended operator. Future work will extend **`EvoX MetaBench`** with broader baselines, richer component compositions, more diverse black-box optimization scenarios, and more efficient execution to support increasingly comprehensive MetaBBO research.

Reproducibility Statement

The source code is available in EvoX MetaBench. Appendix B describes component interfaces and method assembly, while Appendix D documents the benchmark collection and task construction. The main text defines the evaluation metrics and runtime measurements. Appendices E–H provide study-specific protocols, configurations, and detailed results, including the BBOB function partition, optimization budgets, and prompts for the generation-value study.

Ethics Statement

This work provides a general-purpose framework for optimization research, with experiments conducted on computational benchmarks. Downstream applications should incorporate domain-specific safety constraints, particularly for tasks such as UAV planning. LLM-generated optimization code should be validated before deployment.

References

Ali Ahrari, Saber Elsayed, Ruhul Sarker, Daryl Essam, and Carlos A Coello Coello. Problem definition and evaluation criteria for the cec'2022 competition on dynamic multimodal optimization. In *Proceedings of the IEEE World Congress on Computational Intelligence*, volume 18, pp. 23, 2022.

John E Beasley. Or-library: distributing test problems by electronic mail. *Journal of the operational research society*, pp. 1069–1072, 1990.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Rainer E Burkard, Stefan E Karisch, and Franz Rendl. Qaplib–a quadratic assignment problem library. *Journal of Global optimization*, 10(4):391–403, 1997.

Arina Buzdalova, Vladislav Kononov, and Maxim Buzdalov. Selecting evolutionary operators using reinforcement learning: Initial explorations. In *Genetic and Evolutionary Computation Conference Companion*, pp. 1033–1036, 2014.

Gjorgjina Cenikj, Gašper Petelin, and Tome Eftimov. TransOptAS: Transformer-based algorithm selection for single-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 403–406, 2024.

Fei Chen, Yang Gao, Zhao-qian Chen, and Shi-fu Chen. SCGA: Controlling genetic algorithms with Sarsa(0). In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce*, pp. 1177–1183, 2005.

Jiacheng Chen, Zeyuan Ma, Hongshu Guo, Yining Ma, Jie Zhang, and Yue-Jiao Gong. SYMBOL: Generating flexible black-box optimizers through symbolic equation learning. In *International Conference on Learning Representations*, volume 2024, pp. 41907–41926, 2024.

Yutian Chen, Matthew W. Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P. Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. In *Proceedings of the International Conference on Machine Learning*, volume 70, pp. 748–756, 2017.

Ran Cheng, Miqing Li, Ye Tian, Xingyi Zhang, Shengxiang Yang, Yaochu Jin, and Xin Yao. A benchmark test suite for evolutionary many-objective optimization. *Complex & Intelligent Systems*, 3(1):67–81, 2017.

Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1):63–86, 1998.

Leonardo Lucio Custode, Fabio Caraffini, Anil Yaman, and Giovanni Iacca. An investigation on the use of large language models for hyperparameter tuning in evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1838–1845, 2024.

Bingshui Da, Yew-Soon Ong, Liang Feng, A Kai Qin, Abhishek Gupta, Zexuan Zhu, Chuan-Kang Ting, Ke Tang, and Xin Yao. Evolutionary multitasking for single-objective continuous optimization: Benchmark problems, performance metric, and baseline results. *arXiv preprint arXiv:1706.03470*, 2017.

Jonathan PK Doye, Mark A Miller, and David J Wales. Evolution of the potential energy surface with size for lennard-jones clusters. *The Journal of Chemical Physics*, 111(18):8417–8428, 1999.

Qiqi Duan, Guochen Zhou, Chang Shao, Zhuowei Wang, Mingyang Feng, Yuwei Huang, Yajing Tan, Yijun Yang, Qi Zhao, and Yuhui Shi. Pypop7: A pure-python library for population-based black-box optimization. *Journal of Machine Learning Research*, 25(296):1–28, 2024.

Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2 (1):5–30, 1996.

Chenchen Feng, Minyang Chen, Zhuozhao Li, and Ran Cheng. Unleashing the potential of differential evolution through individual-level strategy diversity. *IEEE Transactions on Evolutionary Computation*, 2026.

Iztok Fister, Dušan Fister, and Iztok Fister, Jr. Reinforcement learning-based differential evolution for global optimization. In *Differential Evolution: From Theory to Practice*, pp. 43–75. 2022.

Zijian Gao, Zeyuan Ma, Yuanting Zhong, Yue-Jiao Gong, and Hongshu Guo. Detect and act: Automated dynamic optimizer through meta-black-box optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 889–898, 2026.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, et al. Glm-5: From vibe coding to agentic engineering, 2026. URL `https://arxiv.org/abs/2602.15763`.

Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

Hongshu Guo, Yining Ma, Zeyuan Ma, Jiacheng Chen, Xinglin Zhang, Zhiguang Cao, Jun Zhang, and Yue-Jiao Gong. Deep reinforcement learning for dynamic algorithm selection: A proof-of-principle study on differential evolution. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 54(7):4247–4259, 2024.

Hongshu Guo, Sijie Ma, Zechuan Huang, Yuzhi Hu, Zeyuan Ma, Xinglin Zhang, and Yue-Jiao Gong. Reinforcement learning-based self-adaptive differential evolution through automated landscape feature learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1117–1126, 2025.

Muqi Han, Xiaobin Li, Kai Wu, Xiaoyu Zhang, and Handing Wang. Enhancing zero-shot black-box optimization via pretrained models with efficient population modeling, interaction, and stable gradient approximation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 33238–33274, 2025.

Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tušar, and Dimo Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36(1):114–144, 2021.

Qinglong Hu and Qingfu Zhang. Partition to evolve: Niching-enhanced evolution with llms for automated algorithm discovery. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, pp. 154771–154817, 2025.

Zhenzhen Hu, Wenyin Gong, Witold Pedrycz, and Yanchi Li. Deep reinforcement learning assisted co-evolutionary differential evolution for constrained optimization. *Swarm and Evolutionary Computation*, 83:101387, 2023.

Beichen Huang, Ran Cheng, Zhuozhao Li, Yaochu Jin, and Kay Chen Tan. EvoX: A distributed gpu-accelerated framework for scalable evolutionary computation. *IEEE Transactions on Evolutionary Computation*, 29(5):1649–1662, 2025.

S. Huband, P. Hingston, L. Barone, and L. While. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation*, 10(5):477–506, 2006.

Simon Huband, Luigi Barone, Lyndon While, and Phil Hingston. A scalable multi-objective test problem toolkit. In *International conference on evolutionary multi-criterion optimization*, pp. 280–295, 2005.

Howook Hwang, Thom Vreven, Joël Janin, and Zhiping Weng. Protein–protein docking benchmark version 4.0. *Proteins: Structure, Function, and Bioinformatics*, 78(15):3111–3114, 2010.

Keming Jiao, Jie Chen, Bin Xin, and Li Li. A reference vector-based multiobjective evolutionary algorithm with Q-learning for operator adaptation. *Swarm and Evolutionary Computation*, 76: 101225, 2023.

Giorgos Karafotias, Mark Hoogendoorn, and A. E. Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187, 2015.

Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

Abhishek Kumar, Guohua Wu, Mostafa Z Ali, Rammohan Mallipeddi, Ponnuthurai Nagaratnam Suganthan, and Swagatam Das. A test-suite of non-convex constrained optimization problems from the real-world and some baseline results. *Swarm and evolutionary computation*, 56:100693, 2020.

Abhishek Kumar, Guohua Wu, Mostafa Z Ali, Qizhang Luo, Rammohan Mallipeddi, Ponnuthurai Nagaratnam Suganthan, and Swagatam Das. A benchmark-suite of real-world constrained multi-objective optimization problems and some baseline results. *Swarm and Evolutionary Computation*, 67:100961, 2021.

Robert Lange, Tom Schaul, Yutian Chen, Chris Lu, Tom Zahavy, Valentin Dalibard, and Sebastian Flennerhag. Discovering attention-based genetic algorithms via meta-black-box optimization. In *Proceedings of the genetic and evolutionary computation conference*, pp. 929–937, 2023a.

Robert Tjarko Lange. evosax: Jax-based evolution strategies. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, pp. 659–662, 2023.

Robert Tjarko Lange, Tom Schaul, Yutian Chen, Tom Zahavy, Valentin Dalibard, Chris Lu, Satinder Singh, and Sebastian Flennerhag. Discovering evolution strategies via meta-black-box optimization. In *International Conference on Learning Representations*, 2023b.

Pengyi Li, Jianye Hao, Hongyao Tang, Xian Fu, Yan Zheng, and Ke Tang. Bridging evolutionary algorithms and reinforcement learning: A comprehensive survey on hybrid algorithms. *IEEE Transactions on Evolutionary Computation*, 29(5):1707–1728, 2025a.

Wei Li, Peng Liang, Bo Sun, Yafeng Sun, and Ying Huang. Reinforcement learning-based particle swarm optimization with neighborhood differential mutation strategy. *Swarm and Evolutionary Computation*, 78:101274, 2023.

Xiaobin Li, Kai Wu, Yujian B Li, Xiaoyu Zhang, Handing Wang, and Jing Liu. Pretrained optimization model for zero-shot black box optimization. In *Advances in Neural Information Processing Systems*, volume 37, pp. 14283–14324, 2024.

Xiaobin Li, Kai Wu, Xiaoyu Zhang, and Handing Wang. B2opt: Learning to optimize black-box optimization with little budget. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39 (17):18502–18510, 2025b.

Xiaodong Li, Andries Engelbrecht, and Michael G Epitropakis. Benchmark functions for cec'2013 special session and competition on niching methods for multimodal function optimization. *RMIT University, evolutionary computation and machine learning Group, Australia, Tech. Rep*, 2013.

Zhenyu Liang, Hao Li, Naiwei Yu, Kebin Sun, and Ran Cheng. Bridging evolutionary multiobjective optimization and GPU acceleration via tensorization. *IEEE Transactions on Evolutionary Computation*, 30(1):420–434, 2026.

Zuowen Liao, Qishuo Pang, and Qiong Gu. Differential evolution based on strategy adaptation and deep reinforcement learning for multimodal optimization problems. *Swarm and Evolutionary Computation*, 87:101568, 2024.

Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*, 2023.

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. In *Proceedings of the International Conference on Machine Learning*, volume 235, pp. 32201–32223, 2024a.

Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Qinglong Hu, Ping Guo, Xi Lin, Xialiang Tong, Mingxuan Yuan, et al. LLM4AD: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024b.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, et al. A systematic survey on large language models for algorithm design. *ACM Computing Surveys*, 58(8):1–32, 2026.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large language models as evolutionary optimizers. In *IEEE Congress on Evolutionary Computation*, pp. 1–8, 2024c.

Yaxian Liu, Hui Lu, Shi Cheng, and Yuhui Shi. An adaptive online parameter control algorithm for particle swarm optimization based on reinforcement learning. In *IEEE Congress on Evolutionary Computation*, pp. 815–822, 2019.

Sijie Ma, Zeyuan Ma, Weijia Cao, Yue-Jiao Gong, Lingling Ma, Zhiyang Huang, and Jun Zhang. Learning to optimize uav path planning for data sensing in wireless sensor networks. *arXiv preprint arXiv:2609.16629*, 2026a.

Sijie Ma, Zeyuan Ma, Yue-Jiao Gong, and Ran Cheng. Meta-learning-assisted constraint relaxation for constrained black-box optimization. *arXiv preprint arXiv:2602.00532*, 2026b.

Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Zhenrui Li, Guojun Peng, Yue-Jiao Gong, Yining Ma, and Zhiguang Cao. Metabox: A benchmark platform for meta-black-box optimization with reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 10775–10795, 2023.

Zeyuan Ma, Jiacheng Chen, Hongshu Guo, Yining Ma, and Yue-Jiao Gong. Auto-configuring exploration-exploitation tradeoff in evolutionary computation via deep reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1497–1505, 2024.

Zeyuan Ma, Jiacheng Chen, Hongshu Guo, and Yue-Jiao Gong. Neural exploratory landscape analysis for Meta-Black-Box-Optimization. In *International Conference on Learning Representations*, pp. 15373–15393, 2025a.

Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo, Wenjie Qiu, Sijie Ma, Hongqiao Lian, Jiajun Zhan, Kaixu Chen, Chen Wang, Zhiyang Huang, Zechuan Huang, Guojun Peng, Ran Cheng, and Yining Ma. Metabox-v2: A unified benchmark platform for meta-black-box optimization. In *Advances in Neural Information Processing Systems*, volume 38, 2025b.

Zeyuan Ma, Hongshu Guo, Yue-Jiao Gong, Jun Zhang, and Kay Chen Tan. Toward automated algorithm design: A survey and practical guide to meta-black-box-optimization. *IEEE Transactions on Evolutionary Computation*, 30(2):667–687, 2026c.

Konstantinos Metaxiotis and Konstantinos Liagkouras. Multiobjective evolutionary algorithms for portfolio management: A comprehensive literature review. *Expert systems with applications*, 39 (14):11685–11698, 2012.

Weikang Ning, Baolong Guo, Xinxing Guo, Cheng Li, and Yunyi Yan. Reinforcement learning-aided parameter control in multi-objective evolutionary algorithm based on decomposition. *Progress in Artificial Intelligence*, 7(4):385–398, 2018.

Mohammad Rasoul Omidvar. Benchmark functions for the cec'2008 special session and competition on large scale global optimization. 2007.

Jiyuan Pei, Yi Mei, Jialin Liu, Mengjie Zhang, and Xin Yao. Adaptive operator selection for metaheuristics: A survey. *IEEE Transactions on Artificial Intelligence*, 6(8):1991–2012, 2025.

Sebastian Pineda Arango, Hadi S. Jomaa, Martin Wistuba, and Josif Grabocka. HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML. In *Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

Wenjie Qiu, Hao Guo, Zeyuan Ma, and Yue-Jiao Gong. Advancing CMA-ES with learning-based cooperative coevolution for scalable optimization. In *Genetic and Evolutionary Computation Conference*, pp. 928–937, 2026.

Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4): 376–384, 1991.

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.

Yoshitaka Sakurai, Kouhei Takada, Takashi Kawabe, and Setsuo Tsuruta. A method to control parameters of evolutionary algorithms by using reinforcement learning. In *International Conference on Signal-Image Technology and Internet-Based Systems*, pp. 74–79, 2010.

Hussein Samma, Chee Peng Lim, and Junita Mohamad Saleh. A new reinforcement learning-based memetic particle swarm optimizer. *Applied Soft Computing*, 43:276–297, 2016.

Mudita Sharma, Alexandros Komninos, Manuel López-Ibáñez, and Dimitar Kazakov. Deep reinforcement learning based parameter control in differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 709–717, 2019.

Mhd Ali Shehadeh and Jakub Kŭdela. Benchmarking global optimization techniques for unmanned aerial vehicle path planning. *Expert Systems with Applications*, 293:128645, 2025.

Nidul Sinha, R Chakrabarti, and Pranab Kumar Chattopadhyay. Evolutionary programming techniques for economic load dispatch. *IEEE Transactions on evolutionary computation*, 7(1):83–94, 2003.

Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

Niki van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29 (2):331–345, 2025.

Jianyong Sun, Xin Liu, Thomas Bäck, and Zongben Xu. Learning adaptive differential evolution algorithm from optimization experiences by policy gradient. *IEEE Transactions on Evolutionary Computation*, 25(4):666–680, 2021.

Zhiping Tan and Kangshun Li. Differential evolution with mixed mutation strategy based on deep reinforcement learning. *Applied Soft Computing*, 111:107678, 2021.

Yujin Tang, Yingtao Tian, and David Ha. EvoJax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 308–311, 2022.

Ye Tian, Shichen Peng, Xingyi Zhang, Tobias Rodemann, Kay Chen Tan, and Yaochu Jin. A recommender system for metaheuristic algorithms for continuous optimization based on deep recurrent neural networks. *IEEE Transactions on Artificial Intelligence*, 1(1):5–18, 2020.

Ye Tian, Xiaopeng Li, Haiping Ma, Xingyi Zhang, Kay Chen Tan, and Yaochu Jin. Deep reinforcement learning based adaptive operator selection for evolutionary multi-objective optimization. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 7(4):1051–1064, 2023.

Vishnu TV, Pankaj Malhotra, Jyoti Narwariya, Lovekesh Vig, and Gautam Shroff. Meta-learning for black-box optimization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 366–381, 2019.

Diederick Vermetten, Furong Ye, Thomas Bäck, and Carola Doerr. MA-BBOB: Many-affine combinations of BBOB functions for evaluating AutoML approaches in noiseless numerical black-box optimization contexts. In *Proceedings of the International Conference on Automated Machine Learning*, volume 224, pp. 7/1–7/14, 2023.

Chao Wang, Licheng Jiao, Lingling Li, Jiaxuan Zhao, Guanchun Wang, Fang Liu, and Shuyuan Yang. Task-free adaptive meta black-box optimization. In *International Conference on Learning Representations*, volume 2026, pp. 85355–85385, 2026a.

Chao Wang, Lingling Li, Licheng Jiao, Jiaxuan Zhao, Fang Liu, and Shuyuan Yang. Learning evolution via optimization knowledge adaptation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 48(9):10442–10459, 2026b.

Chen Wang, Sijie Ma, Zeyuan Ma, and Yue-Jiao Gong. Evolution of benchmark: Black-box optimization benchmark design through large language model. In *Proceedings of the International Conference on Machine Learning*, 2026c.

David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

Xingyu Wu, Yan Zhong, Jibin Wu, Bingbing Jiang, and Kay Chen Tan. Large language model-enhanced algorithm selection: towards comprehensive algorithm representation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024.

Xingyu Wu, Yinglan Feng, and Jibin Wu. Llm-driven evolutionary algorithm selection: Bidirectional framework and sampling paradigm. In *International Conference on Machine Intelligence and Nature-Inspired Computing*, pp. 1–7, 2025.

Zhuoliang Xie, Fei Liu, Zhenkun Wang, and Qingfu Zhang. Llm-driven neighborhood search for efficient heuristic design. In *IEEE Congress on Evolutionary Computation*, pp. 1–8, 2025.

Zhenxing Xu, Yizhe Zhang, Weidong Bao, Hao Wang, Ming Chen, Haoran Ye, Wenzheng Jiang, Hui Yan, and Ji Wang. AutoEP: Llms-driven automation of hyperparameter evolution for metaheuristic algorithms. In *International Conference on Learning Representations*, volume 2026, pp. 139927–139945, 2026.

Ke Xue, Jiacheng Xu, Lei Yuan, Miqing Li, Chao Qian, Zongzhang Zhang, and Yang Yu. Multi-agent dynamic algorithm configuration. In *Advances in Neural Information Processing Systems*, pp. 20147–20161, 2022.

Ke Xue, Ruo-Tong Chen, Rong-Xi Tan, Xi Lin, Yunqi Shi, Siyuan Xu, Mingxuan Yuan, and Chao Qian. Bboplace-bench: benchmarking black-box optimization for chip placement. *IEEE Transactions on Evolutionary Computation*, 2026.

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, 2024a.

Qingyong Yang, Shu-Chuan Chu, Jeng-Shyang Pan, Jyh-Horng Chou, and Junzo Watada. Dynamic multi-strategy integrated differential evolution algorithm based on reinforcement learning for optimization problems. *Complex & Intelligent Systems*, 10(2):1845–1877, 2024b.

Xu Yang, Rui Wang, Kaiwen Li, and Hisao Ishibuchi. Meta-black-box optimization for evolutionary algorithms: Review and perspective. *Swarm and Evolutionary Computation*, 93:101838, 2025a.

Xu Yang, Rui Wang, Kaiwen Li, Wenhua Li, Tao Zhang, and Fujun He. Platmetax: An integrated matlab platform for meta-black-box optimization. *arXiv preprint arXiv:2503.22722*, 2025b.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 27144–27152, 2025.

Danial Yazdani, Mohammad Nabi Omidvar, Ran Cheng, Jürgen Branke, Trung Thanh Nguyen, and Xin Yao. Benchmarking continuous dynamic optimization: Survey and generalized test suite. *IEEE Transactions on Cybernetics*, 52(5):3380–3393, 2022.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in neural information processing systems*, volume 37, pp. 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, 2023.

Shihong Yin and Zhengrong Xiang. Adaptive operator selection with dueling deep Q-network for evolutionary multi-objective optimization. *Neurocomputing*, 581:127491, 2024.

Shiyuan Yin, Yi Liu, GuoLiang Gong, Huaxiang Lu, and Wenchang Li. RLEPSO: Reinforcement learning based ensemble particle swarm optimizer. In *Proceedings of the International Conference on Algorithms, Computing and Artificial Intelligence*, pp. 1–6, 2021.

Shiyuan Yin, Min Jin, Huaxiang Lu, Guoliang Gong, Wenyu Mao, Gang Chen, and Wenchang Li. Reinforcement learning-based parameter adaptation method for particle swarm optimization. *Complex & Intelligent Systems*, 9(5):5585–5609, 2023.

Haotian Zhang, Jianyong Sun, Zongben Xu, and Jialong Shi. Learning unified mutation operator for differential evolution by natural evolution strategies. *Information Sciences*, 632:594–616, 2023.

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, 2024a.

Yisong Zhang and Guoxing Yi. Laos: Large language model-driven adaptive operator selection for evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 517–526, 2025.

Yisong Zhang, Guoxing Yi, Hao Wang, Yu Cheng, Yiran Chen, and Zhennan Wei. An improved abc algorithm based on deep reinforcement learning for multi-uav target assignment. In *China Automation Congress*, pp. 4921–4926, 2024b.

Yisong Zhang, Guoxing Yi, Huapin Geng, and Hao Wang. Leveraging large language models for dynamic multi-objective optimization in uav sensor-target assignment. *IFAC-PapersOnLine*, 59 (20):2314–2319, 2025.

Yisong Zhang, Ran Cheng, Guoxing Yi, and Kay Chen Tan. A systematic survey on large language models for evolutionary optimization: From modeling to solving. *IEEE Computational Intelligence Magazine*, 2026.

Bowen Zheng, Ran Cheng, and Kay Chen Tan. EvoRL: A GPU-accelerated framework for evolutionary reinforcement learning. *ACM Transactions on Evolutionary Learning*, 2025a.

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

Ningning Zhu, Fuqing Zhao, and Jie Cao. A hyperheuristic and reinforcement learning guided meta-heuristic algorithm recommendation. In *International Conference on Computer Supported Cooperative Work in Design*, pp. 1061–1066, 2024.

CONTENTS OF THE APPENDIX

A Use of Large Language Models **19**

B Modular Composition of MetaBBO Methods **19**

B.1 Composing Algorithm Selection Methods 19

B.2 Composing Algorithm Configuration Methods 21

B.3 Composing Solution Manipulation Methods 23

B.4 Composing Algorithm Generation Methods 26

C Composable MetaBBO Baselines **28**

D Problem Collection and Task Instantiation **30**

D.1 Collection Overview 30

D.2 Task Instantiation and Benchmark Splits 31

E Details of RQ2 **33**

E.1 Experimental Setup 33

E.2 Detailed Results on BBOB 33

F Details of RQ3 **35**

F.1 Motivation 35

F.2 Experimental Setup 35

F.3 Detailed Results and Analysis 36

G Details of RQ4 **38**

G.1 Motivation 38

G.2 Experimental Setup 38

G.3 Detailed Results and Analysis 39

H Details of RQ5 **41**

H.1 Motivation 41

H.2 Experimental Setup 41

H.3 Detailed Results and Analysis 41

H.4 Prompt Templates and Example Input 42

A USE OF LARGE LANGUAGE MODELS

Large Language Models (LLMs) were used exclusively as a general-purpose writing assistance tool. Their use was limited to language polishing, grammatical correction, and improving the clarity and fluency of the manuscript. LLMs were not involved in the development of research ideas, experimental design, data analysis, or the interpretation of experimental results. All intellectual contributions, including problem formulation, methodological development, and the conclusions drawn from the study, were made entirely by the human authors.

B MODULAR COMPOSITION OF METABBO METHODS

This section details how **`EvoX MetaBench`** composes methods for algorithm selection (AS), algorithm configuration (AC), solution manipulation (SM), and algorithm generation (AG). Each subsection defines the task and its execution step, describes component assembly and the corresponding learning or design-search process, and concludes with a concrete baseline instantiation.

B.1 COMPOSING ALGORITHM SELECTION METHODS

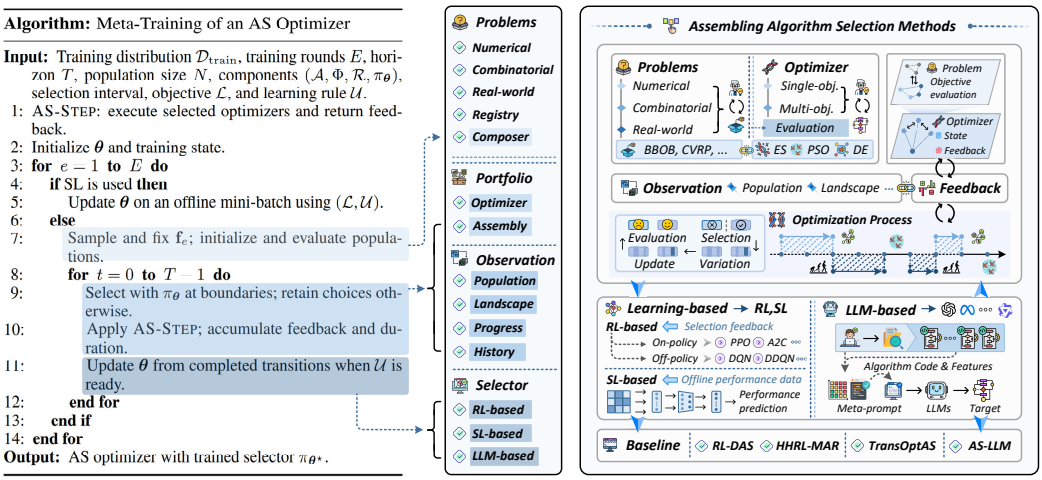

Figure 10: **Unified AS meta-training and framework organization. Left:** Generic AS meta-training lifecycle and strategy-dependent selector-update boundaries. **Right:** Core components, the framework path from problem and portfolio setup through observation and selector assembly to learning-rule selection, and the pre-assembled baselines provided by the framework.

Task formulation and scope. Algorithm Selection chooses a complete optimizer from a portfolio $\mathcal{A} = \{\mathcal{O}_1, \ldots, \mathcal{O}_K\}$ to solve a problem. We consider population-based AS, where an optimizer can be selected at the start of a run or reselected during the search. Given a batch $\mathbf{f}_e$ of B problem instances, let t index execution windows, each comprising a prescribed interval of optimizer execution. Let $\mathcal{Z}_t$ collect the current populations $\mathbf{X}_t$, their fitness values $\mathbf{F}_t = \mathbf{f}_e(\mathbf{X}_t)$, the internal states of the portfolio optimizers, and any required portfolio statistics. At a selection boundary, the feature constructor produces one observation per problem instance, $\mathbf{s}_t = \Phi(\mathcal{Z}_t)$, and the selector returns the algorithm identities $\mathbf{a}_t = \pi_{\boldsymbol{\theta}}(\mathbf{s}_t) \in \{1, \ldots, K\}^B$. These choices are retained until the next selection boundary. One execution window is expressed as

$$(\mathcal{Z}_{t+1}, \mathbf{r}_t) = \text{AS-Step}(\mathcal{Z}_t, \mathbf{a}_t; \mathcal{A}, \mathbf{f}_e). \tag{3}$$

During this window, the selected optimizer advances the search for each problem instance, thereby producing the updated state $\mathcal{Z}_{t+1}$ and a feedback vector $\mathbf{r}_t$ with one value per problem. The next observation is constructed from the updated state as $\mathbf{s}_{t+1} = \Phi(\mathcal{Z}_{t+1})$.

Unified meta-training lifecycle. The left panel of Figure 10 summarizes the AS meta-training lifecycle. Within each rollout of up to T execution windows, the sampled batch of training problems $\mathbf{f}_e$ remains fixed. The selector chooses one optimizer per problem at selection boundaries and retains each choice across the intervening AS-STEP executions. RL records optimization outcomes and selection durations, while SL uses offline problem–portfolio performance data. The learning rule $\mathcal{U}$ updates selector parameters $\boldsymbol{\theta}$ from the corresponding records at method-specific update boundaries.

Composing the AS-STEP. The right panel of Figure 10 summarizes how researchers compose AS-STEP from a portfolio of complete optimizers and compatible selection components. Researchers first choose the portfolio $\mathcal{A}$, then a feature constructor Φ that determines which population, landscape, progress, or portfolio-history information is available to the selector. A feedback rule $\mathcal{R}$ specifies how optimization outcomes are assessed. Researchers further select a controller $\pi_{\boldsymbol{\theta}}$ with one categorical action for each of the K portfolio optimizers and set the selection interval.

An `AlgorithmSelectionSpec` registers these components under a shared `SelectionContract`, which checks that observations and feedback are defined per problem and that the controller's action space matches the portfolio. Binding this specification to a problem suite, population size, and search dimension yields an executable `AlgorithmSelectionTask` implementing Eq. 3. During each execution window, the task runs the selected optimizer for each problem, evaluates its candidate solutions, and constructs the subsequent observation and feedback through Φ and $\mathcal{R}$.

Binding the meta-training rule. A trainable AS method pairs the assembled selector $\pi_{\boldsymbol{\theta}}$ with a compatible learning objective $\mathcal{L}$ and update rule $\mathcal{U}$. RL trains the selector through value-based or policy-gradient updates using selection transitions that record optimization feedback and the selected optimizer's execution duration. SL uses offline problem–portfolio performance data to train a predictor of optimizer performance or a direct selector. The learning rule $\mathcal{U}$ specifies the corresponding update schedule. The same AS-STEP also supports ICL, where an LLM selects an optimizer using problem and portfolio information supplied in its context, with the LLM parameters fixed.

Concrete baseline instantiation. Finally, Listing 1 instantiates RL-DAS with the NL-SHADE-RSP, MadDE, and JDE21 portfolio. It combines fitness-landscape and algorithm-movement features, defines feedback from improvement in the best fitness, and pairs the resulting AS-STEP with a PPO selector.

```python
import evox.alg_select as aset


def make_rldas_trainer(
    problem_factory, problem_specs, *,
    agent_kwargs, pop_size, dim, batch_size,
    horizon, period_fes, device,
):
    # Optimizer portfolio (A).
    library = aset.AlgorithmLibrary()
    for kernel in (
        aset.NLSHADERSPKernel,
        aset.MadDEKernel,
        aset.JDE21Kernel,
    ):
        library.register(kernel)
    pool = library.build_shared_pool(
        state_factory=aset.DESharedStateFactory(
            period_fes=period_fes
        ),
        budget=aset.EvaluationPeriodBudget(
            period_fes=period_fes
        ),
        name="rldas",
    )

    # Feature construction (Phi).
```

```
    phi = aset.population_observation([
        aset.fitness_landscape_features(),
        aset.algorithm_movement_features(
            len(pool), dim=dim, algorithm_names=pool.names,
        ),
    ])

    # AS-Step composition and PPO training rule.
    method = aset.compose(
        name="rldas",
        paradigm="rl",
        algorithm_pool=pool,
        observation=phi,
        reward="population_best",
        controller=aset.build_ppo,
        controller_kwargs={"agent_kwargs": agent_kwargs},
        recipe=aset.RLTrainerConfig(
            horizon=horizon,
            env_batch_size=batch_size,
        ),
        selection_steps=1,
        default_horizon=horizon,
    )
    task = method.spec.bind(
        problem_factory, problem_specs,
        pop_size=pop_size, dim=dim,
    )
    controller = method.build_controller(device=device)
    return aset.RLTrainer(
        task, controller, method.training, device=device,
    )
```

Listing 1: Explicit RL-DAS assembly and rollout-wise PPO training.

With `selection_steps=1`, the selector chooses an optimizer at every execution window, whose evaluation interval is governed by `period_fes`. The returned trainer collects selection trajectories over rollouts of up to `horizon` windows and updates the selector with PPO after each rollout.

B.2 Composing Algorithm Configuration Methods

Task formulation and scope. Algorithm Configuration uses a meta-level controller to adapt the operator choices or parameter values of a given optimizer $\mathcal{O}$ during optimization. Given a batch $\mathbf{f}_e$ of B problem instances, let $\mathbf{X}_t$ denote the populations at generation t, each containing N individuals, and $\mathbf{F}_t = \mathbf{f}_e(\mathbf{X}_t)$ their fitness values. The feature constructor Φ builds an observation $\mathbf{s}_t$ from the available search information, and the controller produces a configuration action $\mathbf{a}_t = \pi_{\boldsymbol{\theta}}(\mathbf{s}_t)$. One generation is expressed as

$$(\mathbf{X}_{t+1}, \mathbf{F}_{t+1}, \mathbf{r}_t, \mathbf{s}_{t+1}) = \text{AC-Step}(\mathbf{X}_t, \mathbf{F}_t, \mathbf{a}_t; \mathbf{f}_e). \tag{4}$$

Within AC-Step, the configuration adapter Ψ maps $\mathbf{a}_t$ to operator choices, parameter values, or both, and applies them to $\mathcal{O}$. These configurations can be shared across a population or assigned to individual solutions. The configured optimizer then advances one generation under $\mathbf{f}_e$, after which the feedback rule $\mathcal{R}$ computes $\mathbf{r}_t$ and Φ constructs the next observation $\mathbf{s}_{t+1}$.

Unified meta-training lifecycle. The left panel of Figure 11 summarizes the AC meta-training lifecycle. Within each rollout of up to T generations, the sampled batch of training problems $\mathbf{f}_e$ remains fixed. At each generation, the controller produces a configuration action, and AC-Step advances the optimizer $\mathcal{O}$ under this configuration. RL collects the resulting transitions or trajectories, while evolution strategies (ES) evaluate candidate controllers using rollout-level performance. The learning rule $\mathcal{U}$ updates controller parameters $\boldsymbol{\theta}$ from the corresponding records at method-specific update boundaries.

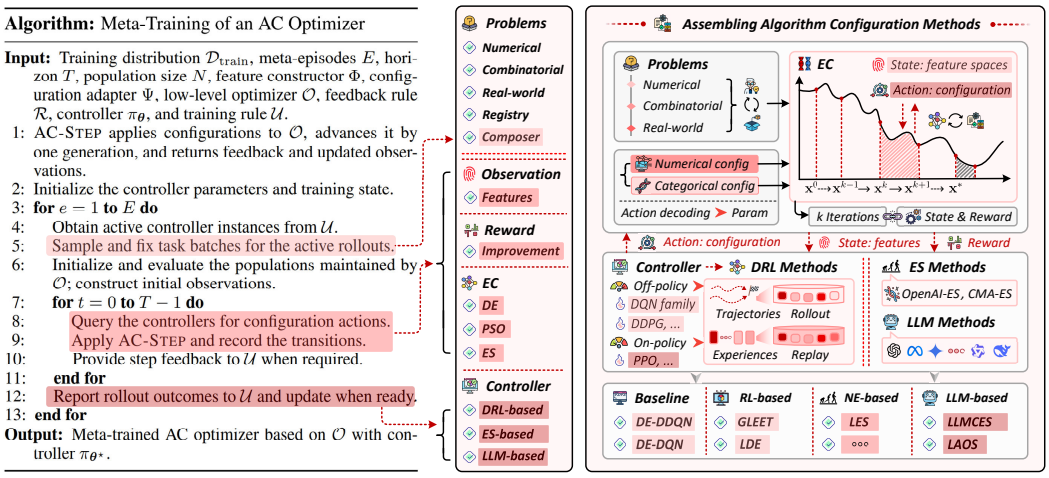

Figure 11: **Unified AC meta-training and framework organization. Left:** Generic AC meta-training lifecycle and strategy-dependent controller-update boundaries. **Right:** Core components, the framework path from problem setup and state construction to low-level optimizer and controller selection, and the pre-assembled baselines provided by the framework.

Composing the AC-STEP. The right panel of Figure 11 summarizes the composition of AC-STEP around a chosen optimizer $\mathcal{O}$. Researchers select the operator choices or parameter slots to control, then choose an adapter Ψ that maps categorical or continuous actions to these settings. They also specify whether configurations are shared across a population or assigned to individual solutions. Researchers then choose Φ to construct the controller's observations and $\mathcal{R}$ to assess optimization outcomes, and select a controller π_θ with compatible observation and action interfaces. The observation and feedback scopes are chosen separately, with each defined at the population or individual level.

A `ConfigurationSpec` registers the components within the framework under a shared `ConfigurationContract`, which checks their interface dimensions and scopes. Binding the specification to a problem suite, population size, and search dimension yields an executable `ConfigurationTask` implementing Eq. 4. At runtime, the task applies each controller action through Ψ, advances $\mathcal{O}$ by one generation under $\mathbf{f}_e$, and constructs the resulting feedback and observation through $\mathcal{R}$ and Φ. Researchers can replace the feature constructor, feedback rule, controller, or adapter to examine alternative configuration methods around the same optimizer.

Binding the meta-training rule. A trainable AC method pairs the assembled controller π_θ with a compatible learning objective $\mathcal{L}$ and update rule $\mathcal{U}$. **`EvoX MetaBench`** supports off-policy RL, on-policy RL, and ES for parameter learning. Off-policy RL stores optimization transitions in replay memory and uses sampled mini-batches for generation-wise controller updates. On-policy RL collects trajectories using the current controller and updates its parameters at rollout boundaries. ES evaluates candidate controller parameterizations over complete rollouts and updates θ according to their relative optimization performance. The same AC-STEP also supports ICL, where an LLM produces configuration actions from search information and available choices in its context, with the LLM parameters fixed.

Concrete baseline instantiation. Finally, Listing 2 instantiates DE-DDQN by composing individual-level operator-performance observations, a categorical DE strategy adapter, the `r2` reward, and a DDQN controller.

```
import evox.alg_config as ac
from evox.alg_config.ec_op.de import de_operator_pool


def make_deddqn_trainer(
    problem_factory, problem_specs, *,
```

```
    pop_size, batch_size, horizon, device,
):
    # DE strategy pool and feature construction (Phi).
    pool = de_operator_pool(
        ("rand/1", "rand/2", "rand-to-best/2", "current-to-rand/1")
    )
    phi = ac.individual_observation(
        [ac.operator_performance_features(len(pool))]
    )

    # Configuration adapter (Psi), feedback (R), and controller.
    spec = ac.compose_configuration(
        name="deddqn",
        observation=phi,
        reward=ac.make_reward("r2"),
        search_operator_factory=lambda:
    ac.DiscreteOperatorSearch(pool),
        controller_builder=ac.build_ddqn,
        action_scope="individual",
        training_paradigm="rl",
        horizon=horizon,
    )
    task = spec.bind(
        problem_factory, problem_specs,
        pop_size=pop_size,
        scale_factor=0.5,
        crossover_rate=1.0,
    )
    controller = task.build_controller(device=device)

    # Generation-wise DDQN training.
    return ac.RLTrainer(
        task, controller,
        ac.RLTrainerConfig(
            horizon=horizon,
            env_batch_size=batch_size,
        ),
        device=device,
    )
```

Listing 2: Explicit DE-DDQN assembly and generation-wise DDQN training.

Each action selects one of four mutation strategies for an individual, with the scale factor fixed at 0.5 and the crossover rate at 1.0. The `r2` reward credits improvements over the parent and assigns higher credit to trials that improve the best fitness. The returned trainer stores individual transitions in replay memory and uses sampled mini-batches for generation-wise DDQN updates.

B.3 Composing Solution Manipulation Methods

Task formulation and scope. Solution Manipulation directly generates or transforms candidate solutions through a parameterized search policy. Given a batch $\mathbf{f}_e$ of B problem instances, let $\mathbf{X}_t$ denote the populations at generation t, each containing N individuals, and $\mathbf{F}_t = \mathbf{f}_e(\mathbf{X}_t)$ their fitness values. One generation is expressed as

$$(\mathbf{X}_{t+1}, \mathbf{F}_{t+1}) = \text{SM-Step}_{\boldsymbol{\theta}}(\mathbf{X}_t, \mathbf{F}_t; \Phi, \mathcal{M}, \mathcal{S}, \mathbf{f}_e), \tag{5}$$

where $\boldsymbol{\theta}$ parameterizes the learned components. The feature constructor Φ extracts search information, $\mathcal{M}$ contains modules for candidate manipulation, such as mutation, crossover, or direct candidate generation, and $\mathcal{S}$ performs survivor selection. SM-Step coordinates these components with objective evaluation under $\mathbf{f}_e$ to construct the next generation. The returned populations and their fitness values satisfy $\mathbf{F}_{t+1} = \mathbf{f}_e(\mathbf{X}_{t+1})$.

Unified meta-training lifecycle. The left panel of Figure 12 summarizes the SM meta-training lifecycle. Within each rollout of up to T generations, the sampled batch of training problems $\mathbf{f}_e$

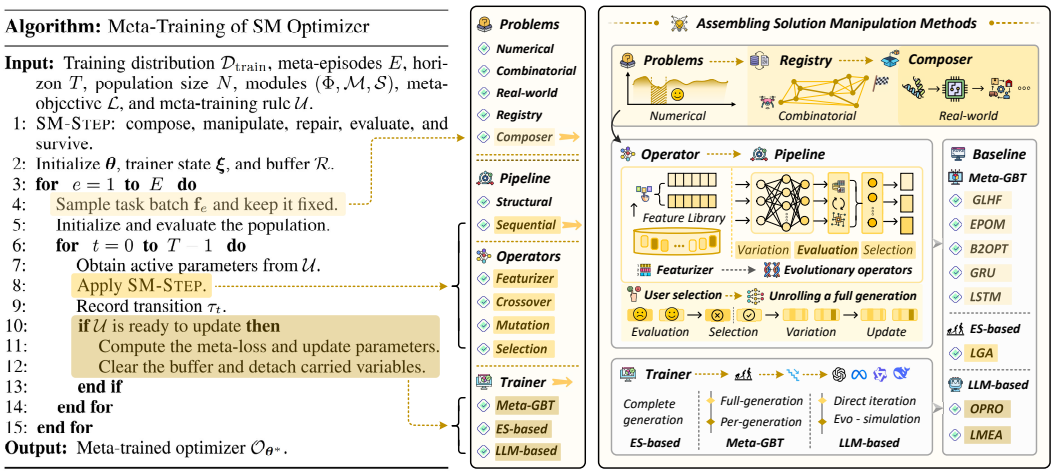

Figure 12: **Unified SM meta-training and framework organization. Left:** Generic SM meta-training lifecycle and strategy-dependent update boundaries. **Right:** Core components, the framework path from problem setup through operator assembly to training-rule selection, and the pre-assembled baselines provided by the framework.

remains fixed. At each generation, SM-STEP uses the current parameters $\boldsymbol{\theta}$ to produce the next population and its corresponding fitness values. SL uses gradients of generation-level or trajectory-level meta-objectives, while ES compares candidate parameterizations using their rollout performance. The learning rule $\mathcal{U}$ updates $\boldsymbol{\theta}$ from these learning signals at method-specific update boundaries.

Composing the SM-STEP. The right panel of Figure 12 summarizes the composition of SM-STEP. Researchers select modules for feature construction Φ, candidate manipulation $\mathcal{M}$, and survivor selection $\mathcal{S}$, connect their inputs and outputs, and specify where candidate populations are evaluated. Let $\mathcal{B}_t^{(k)}$ collect the named intermediate values available after step k. A generation comprising K steps is expressed as

$$\begin{aligned} \mathcal{B}_t^{(0)} &= \{\mathtt{X} : \mathbf{X}_t, \mathtt{F} : \mathbf{F}_t\}, \\ \mathcal{B}_t^{(k)} &= g_k(\mathcal{B}_t^{(k-1)}), \qquad g_k \in \{\mathrm{Block}(m_k), \mathrm{Eval}_{\mathbf{f}_e}\}. \end{aligned} \tag{6}$$

A $\mathrm{Block}(m_k)$ applies an SM module using its declared inputs and outputs, while $\mathrm{Eval}_{\mathbf{f}_e}$ repairs and evaluates a candidate population. The final $\mathtt{X}$ and $\mathtt{F}$ entries define $(\mathbf{X}_{t+1}, \mathbf{F}_{t+1})$.

Researchers assemble these steps using `SequentialOperator`. `StructuredOperator` provides a preset sequence of feature construction, mutation, crossover, and selection, with evaluations after mutation and crossover. Researchers can also implement `BaseOperator` to define other generation procedures. These alternatives expose the same one-generation interface in Eq. 5, so researchers can combine different modules and evaluation schedules into executable SM methods.

Binding the meta-training rule. A trainable SM method pairs the assembled SM-STEP with a compatible meta-objective $\mathcal{L}$ and update rule $\mathcal{U}$. **`EvoX MetaBench`** supports two SL update rules. Trajectory-level SL differentiates $\mathcal{L}$ through a complete rollout and updates $\boldsymbol{\theta}$ once at its end. Generation-wise MetaGBT computes gradients through a single generation and updates $\boldsymbol{\theta}$ after that generation. ES evaluates candidate operator parameterizations over complete rollouts and updates $\boldsymbol{\theta}$ according to their relative optimization performance. The same SM-STEP also supports ICL, where an LLM directly generates or transforms candidate solutions using evaluated solutions and their fitness values supplied in context, with the LLM parameters fixed.

Concrete baseline instantiation. Finally, Listing 3 instantiates GLHF by composing fitness and rank features, learned mutation and crossover modules, and one-to-one selection. The crossover features additionally include normalized cosine similarity between parent and mutant vectors.

```python
import evox.sol_man as sm


def make_glhf_trainer(evaluator, *, pop_size, dim, batch_size, horizon):
    # Parent and crossover features (Phi).
    parent_phi = sm.FeatureComposer(
        sm.FitnessZScore(source="F"),
        sm.PopulationIndexZScore(like="F"),
    )
    crossover_phi = sm.FeatureComposer(
        sm.IdentityFeature(
            source="H",
            feature_dim=parent_phi.feature_dim,
        ),
        sm.JointNormalizedCosine(left="X", right="V"),
    )

    # Learned candidate manipulation (M).
    mutation = sm.LearnedMutationMatrix(
        feature_dim=parent_phi.feature_dim
    )
    crossover = sm.GLHFCrossover(
        feature_dim=crossover_phi.feature_dim
    )

    # Ordered SM-Step with explicit evaluation.
    steps = [
        sm.Block(
            sm.SortByFitness(),
            reads=("X", "F"),
            writes=("X", "F"),
        ),
        sm.Block(parent_phi, reads=parent_phi.input_keys, writes="H"),
        sm.Block(mutation, reads=("X", "H"), writes="V"),
        sm.Block(crossover_phi, reads=crossover_phi.input_keys, writes="Z"),
        sm.Block(crossover, reads=("X", "V", "Z"), writes="U"),
        sm.Eval(reads="U", writes="F_U"),
        sm.Block(
            sm.OneToOneSelection(),
            reads=("X", "F", "U", "F_U"),
            writes=("X", "F"),
        ),
    ]
    operator = sm.SequentialOperator(steps)

    # Meta-objective and training rule.
    binding = sm.GLHF_BASELINE.training_binding("robust")
    return sm.make_trainer(
        operator,
        evaluator,
        strategy=binding.strategy,
        loss_fn=binding.loss.function,
        pop_size=pop_size,
        dim=dim,
        batch_size=batch_size,
        horizon=horizon,
    )
```

Listing 3: Explicit GLHF assembly and generation-wise MetaGBT training.

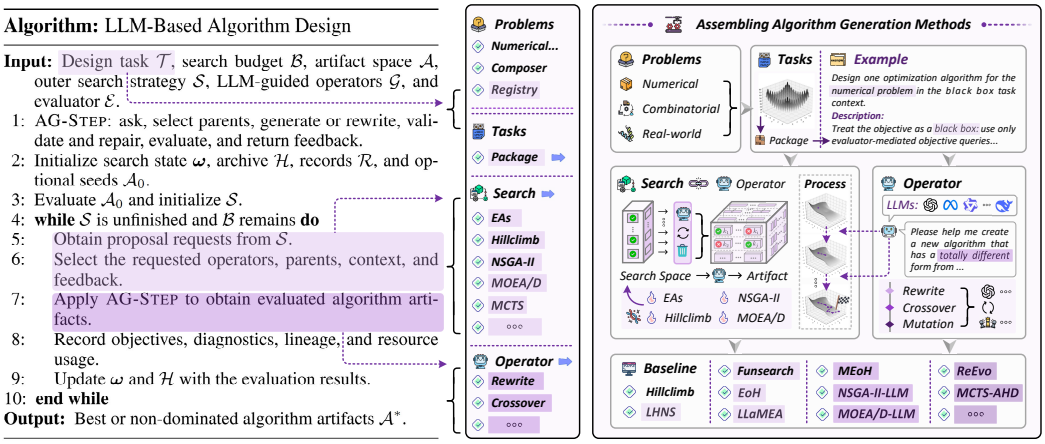

Figure 13: **Unified AG framework organization. Left:** Generic AG lifecycle and strategy-dependent artifact generation, evaluation, and update boundaries. **Right:** Core components, the framework path from task specification through generation-operator assembly to search-strategy selection, and the pre-assembled baselines provided by the framework.

The mutant population V feeds crossover, and the resulting trial population U is repaired and evaluated before one-to-one selection. The robust training binding couples the framework's stabilized GLHF meta-objective with generation-wise MetaGBT updates.

B.4 Composing Algorithm Generation Methods

Task formulation and scope. Algorithm Generation casts algorithm design itself as the outer optimization problem: the search variable is an algorithm artifact. Given a design task $\mathcal{T}$, an artifact space $\mathcal{A}$, and a search budget $\mathcal{B}$, AG iteratively generates and refines artifacts according to their task-level performance. One design iteration is written as:

$$(\boldsymbol{\omega}_{t+1}, \mathbf{h}_t, \mathbf{z}_t) = \text{AG-Step}\left(\boldsymbol{\omega}_t; \mathcal{T}, \mathcal{B}, \mathcal{A}, \mathcal{S}, \mathcal{G}, \mathcal{E}\right), \tag{7}$$

where $\boldsymbol{\omega}_t$ denotes the outer-search state, $\mathcal{S}$ determines how the artifact space is explored, and $\mathcal{G}$ denotes the LLM-guided operators that generate or rewrite algorithm proposals. The evaluator $\mathcal{E}$ maps the resulting artifact batch $\mathbf{h}_t$ to algorithm-level records $\mathbf{z}_t$ containing objectives, feedback, diagnostics, and evaluation status. In contrast to a solution-level query $\mathbf{f}(\mathbf{x})$, which evaluates a candidate solution $\mathbf{x}$, $\mathcal{E}(h)$ executes a complete artifact h over the evaluation suite specified by $\mathcal{T}$ and summarizes its overall algorithmic performance.

Unified design-search lifecycle. The left panel of Figure 13 summarizes the AG design-search lifecycle. At iteration t, the search strategy $\mathcal{S}$ selects a generation operator and the parent artifacts and feedback supplied as context. The LLM generates algorithm proposals through ICL, and the resulting artifacts are evaluated by $\mathcal{E}$. The returned records update the search state to $\boldsymbol{\omega}_{t+1}$ and guide subsequent proposals. The LLM parameters remain fixed throughout this process. Search continues until its stopping criterion is met or the budget $\mathcal{B}$ is exhausted, and returns the algorithm artifacts selected by $\mathcal{S}$.

Composing the AG-STEP. The right panel of Figure 13 summarizes the composition of AG-STEP. Researchers specify the design task $\mathcal{T}$, artifact representation, and evaluator $\mathcal{E}$ through a `TaskPackage`. They select a search strategy $\mathcal{S}$, generation operators $\mathcal{G}$, and an artifact builder $\mathcal{C}$ that converts generated drafts into algorithm artifacts. These components are assembled in a

`DesignWorkflow`, which executes Eq. 7 under the search budget $\mathcal{B}$:

$$\begin{aligned} \mathcal{Q}_t &= \mathrm{ask}_{\mathcal{S}}(\boldsymbol{\omega}_t), \\ d_{t,i} &= \mathcal{G}_{\kappa(q_{t,i})}(q_{t,i}), \\ h_{t,i} &= \mathcal{C}(d_{t,i}, q_{t,i}), \\ z_{t,i} &= \mathcal{E}(h_{t,i}), \\ \boldsymbol{\omega}_{t+1} &= \mathrm{tell}_{\mathcal{S}}(\boldsymbol{\omega}_t, \mathbf{z}_t), \qquad q_{t,i} \in \mathcal{Q}_t. \end{aligned} \tag{8}$$

Here, `ask` produces a batch of proposal requests $\mathcal{Q}_t$. Each request $q_{t,i}$ specifies a generation operator through $\kappa(q_{t,i})$ and supplies its context, including parent artifacts and available evaluation feedback. The selected operator produces a draft $d_{t,i}$ using the LLM, and $\mathcal{C}$ extracts its algorithm content to construct $h_{t,i}$. The evaluator executes the resulting algorithm and returns $z_{t,i}$; `tell` incorporates these records into the search state. Researchers can vary the search strategy and generation operators while retaining the same design task and evaluator.

Concrete baseline instantiation. Finally, Listing 4 instantiates EoH for a given LLM client and task package. The baseline combines rank-based parent selection with LLM-driven initialization, recombination, and mutation.

```python
import evox.alg_gen as ag


def make_eoh_workflow(
    llm, task_package, *, population_size, selection_num,
    max_generations, max_evaluations,
):
    baseline = ag.make_baseline(
        "eoh",
        llm=llm,
        population_size=population_size,
        selection_num=selection_num,
        max_generations=max_generations,
        rng_seed=0,
    )
    return ag.assemble_design_workflow(
        baseline=baseline,
        task_package=task_package,
        budget=ag.DesignBudget(
            max_steps=None,
            max_evaluations=max_evaluations,
        ),
        monitor=ag.WorkflowMonitor(),
    )


workflow = make_eoh_workflow(
    llm,
    task_package,
    population_size=4,
    selection_num=2,
    max_generations=50,
    max_evaluations=300,
)
result = workflow.run()
```

Listing 4: EoH baseline instantiation and task-package binding.

The task package supplies the design objective, Python function interface, and evaluator. EoH uses this specification to construct its prompts and artifact builder. Calling `run` initializes the artifact population and evolves it according to evaluated performance until the generation limit or the budget on algorithm evaluations is reached. The workflow returns the selected artifacts together with the evaluation history.

C COMPOSABLE METABBO BASELINES

We select baselines to capture complementary design choices within each MetaBBO task. The collection includes alternative realizations of shared design principles, as well as methods that adopt different learning or design-search approaches. Their component-based instantiations in **`EvoX MetaBench`** provide reference configurations for comparing complete methods and investigating the design choices that distinguish them. Table 3 summarizes the selected methods, their publication information, and their core mechanisms.

Table 3: MetaBBO baselines instantiated through the framework's modular components, grouped by meta-task. The final column summarizes the core design of each method.

Task	Method	Venue	Technical Summary
AS	RL-DAS (Guo et al., 2024)	TSMC-S	Use PPO to switch among complementary DE algorithms during optimization.
	HHRL-MAR (Zhu et al., 2024)	CSCWD	Use Q-learning to select metaheuristics from optimization feedback.
	TransOptAS (Cenikj et al., 2024)	GECCO	Predict portfolio performance from sampled solutions and fitness values using a Transformer.
	AS-LLM (Wu et al., 2024)	IJCAI	Match problem features with LLM-derived code representations for algorithm selection.
AC	DE-DQN (Tan & Li, 2021)	ASOC	Use DQN to select DE mutation strategies from landscape features.
	DE-DDQN (Sharma et al., 2019)	GECCO	Use DDQN to select mutation strategies for individual DE parents.
	LDE (Sun et al., 2021)	TEVC	Learn recurrent DE parameter control through policy gradients.
	RLEPSO (Yin et al., 2021)	ACAI	Use RL to configure particle subgroups within an ensemble PSO.
	GLEET (Ma et al., 2024)	GECCO	Learn individual exploration–exploitation control with attention and PPO.
	Ada-UCDE (Zhang et al., 2023)	Inf. Sci.	Evolve neural control of a unified DE mutation operator.
	LES (Lange et al., 2023b)	ICLR	Learn attention-based recombination and adaptive distribution updates through neuroevolution.
	LLMCES (Custode et al., 2024)	GECCO	Prompt an LLM to adapt the step size of $(1+1)$-ES.
	LAOS (Zhang & Yi, 2025)	GECCO	Select evolutionary operators through LLM reasoning and retrieved optimization experiences.
SM	RNN-OI (Chen et al., 2017)	ICML	Train recurrent query generation with an observed-improvement objective.
	RNN-Opt (TV et al., 2019)	PKDD	Learn recurrent query generation with discounted regret and constraint feedback.
	LGA (Lange et al., 2023a)	GECCO	Meta-evolve attention-based selection and mutation-rate adaptation.
	GLHF (Li et al., 2024)	NeurIPS	Learn fitness-conditioned mutation and differentiable crossover through generation-wise meta-training.
	B2Opt (Li et al., 2025b)	AAAI	Stack learnable crossover, mutation, and residual selection into optimization blocks.
	EPOM (Han et al., 2025)	NeurIPS	Combine population tokenization, deformable attention, and differentiable weighted-sum crossover.
	OPRO (Yang et al., 2024a)	ICLR	Generate new solutions by prompting an LLM with evaluated candidates.
	LMEA (Liu et al., 2024c)	CEC	Use an LLM for parent selection, crossover, and mutation.
AG	FunSearch (Romera-Paredes et al., 2024)	Nature	Combine LLM program generation with island-based evolutionary search.
	AEL (Liu et al., 2023)	arXiv	Evolve algorithm code through LLM-driven crossover and mutation.
	EoH (Liu et al., 2024a)	ICML	Co-evolve heuristic descriptions and executable code through LLM prompting.
	ReEvo (Ye et al., 2024)	NeurIPS	Guide heuristic evolution with short- and long-term LLM reflection.
	LLaMEA (Stein & Bäck, 2025)	TEVC	Generate and refine metaheuristics using execution and performance feedback.
	LLMEPS (Zhang et al., 2024a)	PPSN	Refine the incumbent heuristic through LLM proposals and $(1+1)$ selection.
	MEoH (Yao et al., 2025)	AAAI	Combine dominance and code dissimilarity for multi-objective heuristic evolution.
	MCTS-AHD (Zheng et al., 2025b)	ICML	Guide LLM heuristic generation with Monte Carlo tree search.
	LHNS (Xie et al., 2025)	CEC	Explore heuristic neighborhoods using LLM-driven ruin and recreation.
	PartEvo (Hu & Zhang, 2025)	NeurIPS	Evolve algorithm niches through reflection, recombination, and adaptive resource allocation.

Algorithm selection. AS baselines capture differences in selection timing and the representation of problems and algorithms. RL-DAS (Guo et al., 2024) and HHRL-MAR (Zhu et al., 2024) use ongoing search feedback to revise the active optimizer. TransOptAS (Cenikj et al., 2024) and AS-LLM (Wu et al., 2024) offer complementary approaches to problem–algorithm matching: the former predicts portfolio performance from sampled solutions and objective values, while the latter matches problem features with LLM-derived code representations. Together, they provide reference configurations for studying adaptive selection and the roles of problem and algorithm information in guiding algorithm choice.

Algorithm configuration. AC baselines cover two complementary forms of control: selecting operators and adjusting parameters. DE-DQN (Tan & Li, 2021) and DE-DDQN (Sharma et al., 2019) learn mutation-strategy selection through RL, whereas LDE (Sun et al., 2021), RLEPSO (Yin et al., 2021), and GLEET (Ma et al., 2024) use RL to adapt optimizer parameters. Ada-UCDE (Zhang et al., 2023) and LES (Lange et al., 2023b) complement these methods with ES-based learning of control mechanisms for DE and evolution strategies. LLMCES (Custode et al., 2024) and LAOS (Zhang & Yi, 2025) revisit parameter adaptation and operator selection through ICL, respectively. Together, these methods provide reference configurations for studying how the configuration decisions made within an optimizer interact with the mechanisms used to learn or infer them.

Solution manipulation. SM baselines connect solution generation with different learning mechanisms and update timescales. RNN-OI (Chen et al., 2017), RNN-Opt (TV et al., 2019), and

B2Opt (Li et al., 2025b) apply gradient updates after complete rollouts, while GLHF (Li et al., 2024) and EPOM (Han et al., 2025) use generation-wise MetaGBT. LGA (Lange et al., 2023a) provides an ES-based alternative for learning evolutionary operators. OPRO (Yang et al., 2024a) and LMEA (Liu et al., 2024c) represent complementary ICL designs: the former directly proposes solutions from evaluation histories, while the latter organizes LLM generation through parent selection, crossover, and mutation.

Algorithm generation. AG baselines cover complementary ways of organizing and guiding design search. LLMEPS (Zhang et al., 2024a) and LLaMEA (Stein & Bäck, 2025) emphasize iterative refinement, while AEL (Liu et al., 2023) and EoH (Liu et al., 2024a) evolve populations of algorithm designs. FunSearch (Romera-Paredes et al., 2024) and MCTS-AHD (Zheng et al., 2025b) organize exploration through islands and search trees, respectively. ReEvo (Ye et al., 2024) and LHNS (Xie et al., 2025) guide new proposals through reflection and code-neighborhood search. PartEvo (Hu & Zhang, 2025) and MEoH (Yao et al., 2025) complement these approaches with niching and multiple design objectives to combine the search for high-performing algorithms with the preservation of diverse designs.

D Problem Collection and Task Instantiation

D.1 Collection Overview

Table 4 summarizes the suites included in **EvoX MetaBench**. The three categories provide complementary settings for developing MetaBBO methods and examining their generalization.

Table 4: Problem suites in **EvoX MetaBench**. *Type* indicates the primary optimization setting. *Size* reports finite-profile task counts or external-instance recipe counts for suites; *Param.* denotes configurable task families.

Category	Suite	Type	Size	Technical Summary
Numerical	Classic	SOO	9	Dimension-scalable analytic functions for controlled diagnostics and ablations.
	BBOB	SOO	2,160	24 functions across six dimensions and 15 official instance identifiers.
	CEC2013	SOO	84	Shifted and rotated basic functions with composition landscapes.
	CEC2014	SOO	120	Basic, hybrid, and composition functions with heterogeneous variable interactions.
	CEC2017	SOO	116	Competition-profile functions spanning basic, hybrid, and composition landscapes.
	CEC2020	SOO	38	Ten selected functions spanning basic, hybrid, and composition landscapes.
	CEC2022	SOO	24	Basic, hybrid, and composition landscapes at two competition dimensions.
	DTLZ	MOO	7	Scalable objectives with linear, spherical, degenerate, and disconnected fronts.
	UF	MOO	10	Nonlinear variable linkages with diverse Pareto-front structures.
	ZDT	MOO	5	Continuous bi-objective functions with varied front shapes and multimodality.
	WFG	MOO	9	Transformation-based landscapes varying bias, deception, and nonseparability.
	SMOP	MOO	24	Eight sparse bi-objective functions at 100, 500, and 1,000 dimensions.
	MaF	MaOO	45	15 functions with diverse Pareto geometries at 5, 10, and 15 objectives.
	CF	C-MOO	10	Nonlinear constraints shaping feasible regions and attainable Pareto fronts.
	MW	C-MOO	26	Diverse interactions between constraints, objective landscapes, and Pareto fronts.
	CEC2013-MMO	MMO	20	Niching landscapes with prescribed global optima and niche radii.
	CEC2010-LSO	LSGO	20	Thousand-dimensional landscapes with separable, grouped, and nonseparable variables.
	CEC2013-LSO	LSGO	15	Large-scale interactions with unequal groups and conforming or conflicting overlaps.
	CEC2017-MTO	MTO	18	Nine task pairs varying optimum intersection and inter-task landscape similarity.
	WCCI2020	MTO	500	Ten groups of fifty shifted and rotated scalar tasks.
	AWCCI	MTO	6,350	Transformed scalar tasks sampled from function subsets at five shift levels.
	MPB	DO	Param.	Moving peaks with configurable positions, heights, widths, and change dynamics.
	GDBG	DO	Param.	Evolving composition landscapes constructed from rotated analytic base functions.
	GMPB	DO	Param.	Evolving peaks with configurable rotations, nonlinear distortions, and widths.
	FPs	DO	Param.	Dynamic peaks with configurable geometry within fixed search-space partitions.
Combinatorial	TSP	SOO	111	TSPLIB95 tours evaluated using native edge-weight conventions.
	CVRP	SOO	100	CVRPLIB X routes decoded under vehicle-capacity limits.
	VRPTW	SOO	56	Solomon routes ranked by vehicle count, then travel distance.
	1D Bin Packing	SOO	160	Falkenauer instances minimizing bin count under capacity limits.
	0–1 MDKP	C-SOO	270	Chu–Beasley binary selection under multiple resource constraints.
	JSSP	SOO	80	Taillard schedules minimizing makespan under job and machine restrictions.
	QAP	SOO	136	QAPLIB permutations minimizing flow-weighted location distances.
	Set Cover	C-SOO	65	OR-Library subsets minimizing cost subject to full coverage.
	CFLP	C-SOO	49	OR-Library facility location and allocation with explicit capacity constraints.
	Max-Cut	SOO	71	G-set graph partitions maximizing the total weight of crossing edges.
Real-world	Engineering-SO	C-SOO	8	Mechanical and structural design, predominantly with explicit constraints.
	Engineering-MO	C-MOO	2	Constrained trade-offs in truss and welded-beam design.
	RE	MOO	16	Engineering trade-offs, including constraint violations represented as objectives.
	CRE	C-MOO	8	Engineering design with explicit inequality constraints.
	BBOPlace	SOO	42	14 chip-placement circuits under three formulations with external evaluators.
	ELD	SOO	5	Single-period thermal dispatch with prescribed internal penalties.
	DED	SOO	2	24-period thermal dispatch with prescribed internal penalties.
	Portfolio	MOO	10	Risk–return trade-offs for MVP and CCP formulations on five market datasets.
	UAV-SO	SOO	56	Scalar path costs over 56 terrain scenarios.
	UAV-MO	MaOO	56	Five path-cost objectives over the same 56 terrain scenarios.
	Protein Docking	SOO	280	Fixed 12-dimensional interaction-energy surrogates for protein docking.
	LJ Clusters	SOO	5	Lennard–Jones energy minimization for clusters of 13, 38, 55, 75, and 98 atoms.
	HPO	SOO	Param.	Configurable model and algorithm hyperparameter searches through workflow adapters.
	Neuroevolution	SOO	Param.	Neural-parameter optimization through supervised-learning and control-environment adapters.

Numerical benchmarks. BBOB provides an established testbed for MetaBBO and supports meta-training in MetaBox (Ma et al., 2023; 2025b) and the evaluation of recent methods such as ABOM (Wang et al., 2026a). Recent studies have further extended MetaBBO to constrained (Ma et al., 2026b) and dynamic optimization (Gao et al., 2026), which motivates coverage of a broader range of optimization settings. Our numerical collection complements BBOB (Hansen et al., 2021) with Classic and CEC landscapes (Ahrari et al., 2022), alongside multi- (Huband et al., 2005; 2006) and many-objective (Cheng et al., 2017) suites that vary Pareto-front geometry, variable linkage, and decision sparsity. Constrained suites (Kumar et al., 2021) introduce diverse feasible-region struc-

tures, while large-scale (Omidvar, 2007), multimodal (Li et al., 2013), multitask (Da et al., 2017), and dynamic suites (Yazdani et al., 2022) expose variable interactions, multiple-optimum structure, inter-task relationships, and changing landscapes. These settings support controlled studies of how MetaBBO components respond to different optimization challenges.

Combinatorial problems. Combinatorial optimization (Beasley, 1990) is a prominent application area of LLM-based AG. Construction rules and local-search operations provide concrete program components that LLMs can generate and refine. Our collection supports this line of research through routing (Reinelt, 1991), scheduling (Solomon, 1987), packing (Falkenauer, 1996), selection (Chu & Beasley, 1998), assignment (Burkard et al., 1997), and graph-partitioning (Goemans & Williamson, 1995) problems drawn from established benchmark libraries. Their decision structures include permutations, discrete selections, and allocations, while capacity limits, coverage requirements, and precedence relations impose different feasibility conditions. This diversity provides a basis for studying generated heuristics and reusable search components across structured decision spaces.

Real-world problems. Hyperparameter optimization (Pineda Arango et al., 2021), protein docking (Hwang et al., 2010), and other application problems (Kumar et al., 2020; Zhang et al., 2025) have been used to evaluate how MetaBBO methods generalize from synthetic training landscapes to application tasks. We broaden this evaluation scope through problems in engineering, energy, finance, science, and machine learning. Engineering design (Hansen et al., 2021) and chip placement (Xue et al., 2026) introduce application-specific design objectives; dispatch (Sinha et al., 2003) and portfolio (Metaxiotis & Liagkouras, 2012) problems capture operating costs and risk–return trade-offs. UAV path planning (Shehadeh & Kůdela, 2025) and molecular problems (Doye et al., 1999) add geometric and physical objectives, while HPO (Pineda Arango et al., 2021) and neuroevolution (Brockman et al., 2016) provide configurable model optimization tasks. Their evaluations draw on analytic domain models, empirical data, simulators, and learning workflows. These resources support generalization studies across diverse application objectives and evaluation settings.

D.2 Task Instantiation and Benchmark Splits

We use the COCO-BBOB test suite (Hansen et al., 2021) to illustrate how task specifications are organized into benchmark splits and instantiated for optimization. A BBOB task is denoted by $\tau_{f,d,i} = (p_f, \alpha_d, \iota_i)$, where f identifies the function template, d specifies the search dimension, and i is an official instance identifier. Together, these fields determine a concrete BBOB landscape.

This example split targets generalization to unseen function templates, with training and test tasks sharing the same dimension and instance configurations. Following the MetaBox-v2 (Ma et al., 2025b) difficult function partition, we define $\mathcal{F} = \{1, \ldots, 24\}$, $\mathcal{F}_{\text{train}} = \{1, 2, 3, 5, 15, 16, 17, 21\}$, and $\mathcal{F}_{\text{test}} = \mathcal{F} \setminus \mathcal{F}_{\text{train}}$. We expand both function sets over $\mathcal{D} = \{2, 3, 5, 10, 20, 40\}$ and $\mathcal{I} = \{1, \ldots, 5, 71, \ldots, 80\}$, and obtain the manifests

$$\mathcal{M}_s^{\text{BBOB}} = \{\tau_{f,d,i} \mid f \in \mathcal{F}_s,\, d \in \mathcal{D},\, i \in \mathcal{I}\}, \qquad s \in \{\text{train}, \text{test}\}. \tag{9}$$

The resulting manifests contain $8 \times 6 \times 15 = 720$ training tasks and $16 \times 6 \times 15 = 1{,}440$ test tasks. Once a task is selected from a manifest, the BBOB construction procedure generates its translation, objective bias, and function-specific transformations from the selected specification. Listing 5 implements this process: `BBOBSuite` enumerates the tasks in each split, and `ProblemComposer` instantiates a selected entry as an executable problem.

```python
import torch

from evox.problems import ProblemComposer
from evox.problems.numerical import BBOBSuite


FUNCTIONS = tuple(range(1, 25))
TRAIN_FUNCTIONS = (1, 2, 3, 5, 15, 16, 17, 21)
TEST_FUNCTIONS = tuple(
    f for f in FUNCTIONS if f not in TRAIN_FUNCTIONS
)
```

```
DIMENSIONS = (2, 3, 5, 10, 20, 40)
INSTANCES = tuple(range(1, 6)) + tuple(range(71, 81))

# Partition functions while sharing dimensions and instance IDs.
train_manifest = BBOBSuite(
    function_ids=TRAIN_FUNCTIONS,
    dimensions=DIMENSIONS,
    instance_ids=INSTANCES,
)
test_manifest = BBOBSuite(
    function_ids=TEST_FUNCTIONS,
    dimensions=DIMENSIONS,
    instance_ids=INSTANCES,
)

# Select a manifest entry and construct the corresponding problem.
task_entry = next(iter(train_manifest))
problem = ProblemComposer().instantiate(
    f"bbob_f{task_entry.function_id:02d}",
    dim=task_entry.dimension,
    instance_id=task_entry.instance_id,
    dtype=torch.float64,
)
```

Listing 5: BBOB manifest construction with disjoint function templates and instantiation of a selected task.

E DETAILS OF RQ2

E.1 EXPERIMENTAL SETUP

RQ2 evaluates optimization performance on 40-dimensional BBOB problems and three application suites. Methods requiring parameter learning are trained on eight BBOB functions, $\{f_1, f_2, f_3, f_5, f_{15}, f_{16}, f_{17}, f_{21}\}$, and tested on the remaining sixteen functions. Subsequent BBOB experiments use the same function partition. Selected trained methods are further evaluated on 86 HPO-B instances, 56 UAV instances, and 5 LJ instances to assess cross-domain generalization. Classical optimizers are evaluated directly on the corresponding test problems.

In contrast, AG methods follow a distinct design-then-test procedure for each target BBOB function, which aligns with the standard evaluation paradigm for algorithm generation. Each method conducts one design run per function with a budget of at most 136 API calls. Candidate algorithms are evaluated and selected on separate design instances of that function. The selected algorithm is then frozen and evaluated on an independent test instance, thereby assessing its performance on new instances of the target function. This protocol differs from the train-test pipeline of parameter-learning methods, which are tested on function templates excluded from training, yet both approaches evaluate generalization under their respective meta-optimization frameworks. All final BBOB evaluations use a unified experimental configuration. Specifically, we use 30 test seeds and a budget of 10,000 function evaluations per optimization run. Under these settings, we report mean normalized improvement S and the mean and standard deviation of final best-so-far fitness.

E.2 DETAILED RESULTS ON BBOB

Tables 5 and 6 report the detailed RQ2 results on the 16 BBOB test functions. The comparison includes classical optimizers and MetaBBO baselines across AS, AC, SM, and AG. Each entry reports the mean $\pm$ standard deviation of final best-so-far fitness over independent runs, which characterizes solution quality and variability on each function. Lower fitness indicates better performance.

The four AG methods achieve the highest mean S after function-specific design (Figure 6). This advantage may partly arise from the interaction between solution search and algorithm design. Inner optimization runs provide performance feedback, which the outer search uses to select and refine candidate algorithms. This interaction supports evolutionary compounding: effective search strategies are retained in reusable programs, so subsequent design iterations can build on earlier improvements. Progress in algorithm design can thus carry forward to solution search on new instances of the target function.

Table 5: Final best-so-far fitness on 16 BBOB test functions (1/2). Each cell reports the mean $\pm$ standard deviation over independent runs. Column bests are shaded and bold; runners-up are underlined.

	Method	f_4	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
Classic	DE	9.352e2±3.066e1	1.127e2±6.347e1	-1.930e2±1.010e1	3.329e2±1.003e2	2.262e2±4.831e1	1.363e5±4.834e4	2.917e2±3.363e1	1.010e6±9.155e5
	PSO	9.177e2±8.987e1	2.112e3±3.179e3	-5.654e1±6.656e1	3.932e3±3.346e3	7.773e2±3.815e2	1.830e5±8.101e4	4.814e2±4.008e2	4.119e7±1.820e7
	ES	1.721e3±6.787e1	2.358e4±8.150e3	1.172e2±3.105e1	2.685e4±4.928e3	2.486e4±4.026e3	1.356e6±2.499e5	2.207e2±2.202e1	1.566e8±1.892e7
AS	RL-DAS	8.278e2±3.406e1	9.436e1±2.184e1	-1.434e2±1.519e1	7.360e2±2.557e2	3.250e2±3.153e0	1.570e5±3.095e4	1.989e2±3.269e1	2.171e6±1.052e6
	HHRL-MAR	7.954e2±6.404e1	8.616e1±7.131e1	-1.373e2±3.688e1	2.846e2±9.616e1	3.603e2±1.420e2	1.183e5±4.108e4	2.379e2±4.016e1	1.460e5±1.104e5
	AS-LLM	8.683e2±7.390e1	1.003e2±5.350e0	-1.859e2±1.151e1	5.476e2±1.440e2	2.751e2±1.187e2	4.951e4±2.959e4	2.431e2±4.718e1	2.564e6±1.891e6
AC	DE-DQN	1.337e3±1.412e2	2.397e5±1.351e5	4.051e1±6.743e1	2.217e4±7.623e3	1.045e3±4.419e2	3.572e5±2.624e5	1.350e2±3.200e1	1.481e8±3.666e7
	DE-DDQN	1.480e3±1.309e2	1.932e5±7.939e4	-1.304e1±6.535e1	2.669e4±1.057e4	1.960e3±1.123e3	5.396e5±2.396e5	1.321e2±3.116e1	1.698e8±3.601e7
	LDE	8.933e2±2.286e1	5.827e1±4.112e1	-1.908e2±8.347e0	2.630e2±3.998e1	1.777e2±3.140e1	9.742e4±2.782e4	1.640e2±2.295e1	4.027e5±1.716e5
	GLEET	1.812e3±3.163e2	4.989e4±3.979e4	2.848e2±1.386e2	7.417e4±3.115e4	1.620e4±1.164e4	1.408e6±7.701e5	5.746e2±8.781e1	1.978e8±7.153e7
	LAOS	1.526e3±1.274e2	2.053e5±9.980e4	-2.591e1±1.005e2	2.240e4±1.335e4	1.232e3±4.262e2	3.163e5±1.474e5	9.849e1±1.948e1	1.599e8±3.613e7
SM	RNN-OI	1.882e3±2.313e-13	1.045e6±1.219e5	8.869e2±5.782e-13	1.242e5±5.920e-11	1.257e3±0.000e0	1.120e7±2.005e6	2.422e2±0.000e0	6.361e8±1.644e7
	RNN-Opt	1.949e3±1.156e-12	1.055e6±1.273e5	9.780e2±3.469e-13	1.038e5±0.000e0	3.620e2±1.156e-13	1.177e7±2.533e6	1.156e4±1.385e4	5.710e8±1.864e6
	LGA	1.150e3±2.145e2	2.001e2±9.443e1	-1.191e2±8.597e1	3.719e2±6.985e2	3.049e3±1.296e4	1.118e5±5.203e4	4.155e2±6.202e1	5.209e5±3.406e5
	GLHF	1.109e3±7.343e1	9.505e2±8.813e2	-1.186e2±2.779e1	1.364e3±6.972e2	2.676e2±3.638e1	2.959e5±1.227e5	2.185e2±4.528e1	2.893e7±1.127e7
	OPRO	1.810e3±8.267e1	1.086e6±1.000e5	1.011e3±5.240e1	9.295e4±8.345e3	3.505e2±5.841e-1	1.250e7±2.062e6	2.298e4±3.897e4	5.612e8±1.792e7
AG	EoH	6.370e2±1.527e1	-1.100e2±8.559e-2	**-2.244e2**±1.330e0	**3.419e1**±1.007e0	**9.760e1**±5.659e-2	**8.953e2**±1.121e3	8.183e1±1.464e1	-9.976e2±2.347e0
	ReEvo	6.295e2±8.705e0	-1.090e2±2.396e0	-2.214e2±2.416e0	6.635e1±2.964e-1	1.315e2±1.094e1	6.613e3±1.735e3	1.074e2±1.608e1	-9.993e2±6.128e-1
	MCTS-AHD	6.297e2±5.059e1	-9.938e1±8.131e0	-2.212e2±2.220e0	6.626e1±1.833e0	1.392e2±1.601e1	5.526e4±3.878e4	1.371e2±2.557e1	2.135e6±8.535e6
	LLMEPS	**5.704e2**±0.000e0	**-1.101e2**±2.579e-2	-2.231e2±2.164e0	6.509e1±9.550e0	1.336e2±1.423e1	1.070e3±5.174e2	**5.586e1**±1.865e1	**-9.995e2**±7.111e-1

Table 6: Final best-so-far fitness on 16 BBOB test functions (2/2).

	Method	f_{13}	f_{14}	f_{18}	f_{19}	f_{20}	f_{22}	f_{23}	f_{24}
Classic	DE	5.472e2±5.904e1	2.986e2±5.666e-1	2.044e2±1.834e0	8.171e2±3.808e-1	7.820e0±1.478e-1	-6.089e1±8.082e0	9.491e1±5.467e-1	3.669e2±1.993e1
	PSO	1.193e3±2.017e2	3.078e2±3.523e0	2.128e2±4.722e0	8.176e2±1.004e0	5.961e2±1.590e3	-3.936e1±1.768e1	9.456e1±1.168e0	4.269e2±5.774e1
	ES	1.774e3±7.218e1	3.181e2±1.692e0	2.254e2±2.317e0	8.199e2±6.546e-1	9.481e3±2.267e3	-7.649e0±2.118e0	9.478e1±5.097e-1	6.732e2±2.923e1
AS	RL-DAS	6.429e2±6.424e1	2.987e2±9.463e-1	2.166e2±2.595e0	8.154e2±4.796e-1	7.188e0±1.010e-1	-4.730e1±5.762e0	9.467e1±3.904e-1	4.932e2±5.023e1
	HHRL-MAR	4.766e2±4.341e1	2.979e2±1.987e-1	2.093e2±3.460e0	8.185e2±7.248e-1	6.642e0±2.048e-1	-6.178e1±2.104e0	9.475e1±4.101e-1	3.927e2±6.460e1
	AS-LLM	6.841e2±8.447e1	3.004e2±2.238e0	2.022e2±1.337e0	8.166e2±2.712e-1	6.827e0±3.672e-1	-5.780e1±2.198e0	9.481e1±3.899e-1	3.669e2±1.934e1
AC	DE-DQN	1.903e3±2.041e2	3.204e2±4.706e0	2.211e2±4.307e0	8.166e2±4.141e-1	5.309e2±1.012e3	-6.321e0±4.687e0	9.481e1±3.907e-1	4.246e2±2.787e1
	DE-DDQN	1.888e3±1.928e2	3.174e2±3.347e0	2.171e2±2.957e0	8.169e2±3.529e-1	2.073e3±3.169e3	-1.163e1±6.576e0	9.485e1±4.372e-1	4.193e2±3.825e1
	LDE	4.952e2±3.720e1	2.982e2±2.768e-1	2.111e2±2.004e0	8.168e2±3.053e-1	7.649e0±8.313e-2	**-6.290e1**±1.490e0	9.469e1±2.968e-1	3.832e2±2.009e1
	GLEET	1.994e3±2.356e2	3.325e2±9.205e0	2.280e2±6.025e0	8.212e2±1.281e0	2.171e4±9.731e3	-1.224e1±1.180e1	9.428e1±5.765e-1	6.632e2±7.247e1
	LAOS	1.994e3±2.051e2	3.187e2±3.022e0	2.188e2±3.828e0	8.166e2±2.912e-1	2.633e3±3.289e3	-8.880e0±6.391e0	9.509e1±4.212e-1	4.299e2±2.994e1
SM	RNN-OI	3.047e3±4.625e-13	3.710e2±1.006e0	2.639e2±2.511e0	8.167e2±1.156e-13	2.311e4±7.400e-12	1.163e1±5.420e-15	9.766e1±7.328e-1	5.691e2±2.313e-13
	RNN-Opt	2.917e3±0.000e0	3.622e2±1.156e-13	2.633e2±2.244e0	8.182e2±5.782e-13	3.332e4±2.220e-11	1.025e1±0.000e0	9.799e1±1.070e0	6.473e2±0.000e0
	LGA	4.929e2±1.058e2	2.986e2±1.809e0	2.124e2±5.269e0	8.144e2±8.935e-1	9.556e0±1.206e1	-5.720e1±1.249e1	9.158e1±4.746e-1	3.032e2±1.089e2
	GLHF	1.004e3±1.259e2	3.059e2±2.314e0	2.117e2±2.611e0	8.157e2±3.805e-1	8.137e0±2.040e-1	-4.375e1±1.185e1	9.467e1±4.742e-1	3.789e2±2.433e1
	OPRO	2.881e3±8.017e1	3.547e2±9.370e0	2.524e2±7.375e0	8.148e2±3.831e0	3.065e4±3.525e3	9.284e0±1.442e0	9.723e1±1.283e0	5.716e2±3.816e1
AG	EoH	3.304e2±2.234e0	2.976e2±2.170e-4	**1.971e2**±3.578e-2	8.104e2±1.612e-3	5.980e0±1.249e-1	-6.275e1±4.477e0	9.129e1±1.163e0	8.686e1±2.687e1
	ReEvo	3.315e2±2.936e0	2.976e2±1.323e-2	1.971e2±2.851e-2	**8.103e2**±0.000e0	6.035e0±1.561e-1	-5.847e1±5.852e0	9.105e1±1.421e-1	9.267e1±2.121e1
	MCTS-AHD	**3.301e2**±9.052e-1	2.976e2±2.053e-3	2.031e2±2.530e0	8.165e2±3.174e-1	**5.956e0**±1.222e-1	-5.648e1±1.123e1	**9.097e1**±1.506e-1	6.150e1±2.308e1
	LLMEPS	3.425e2±1.505e1	**2.976e2**±1.970e-5	1.972e2±5.039e-2	8.136e2±5.403e-1	7.060e0±3.390e-1	-5.569e1±1.748e1	9.179e1±4.278e-1	**4.924e1**±2.032e1

F Details of RQ3

F.1 Motivation

Within the AC-Step in Eq. 4, the configuration adapter Ψ maps controller actions to the native controls of the low-level optimizer $\mathcal{O}$, while its assignment scope determines the *configuration granularity*. Population-shared assignment applies one configuration across the population, whereas per-individual assignment allows each individual to receive a separate configuration. Group-shared assignment provides an intermediate granularity by partitioning the population into configuration-sharing subsets, as illustrated in Fig. 14. Finer-grained assignment permits configuration heterogeneity within the population, whose benefits need not depend on learning. For example, iStratDE shows that randomly assigned, fixed individual-level strategies and parameters can support strong DE performance without feedback-driven adaptation (Feng et al., 2026). Consequently, gains from learned per-individual control may reflect both the heterogeneity enabled by its assignment scope and the configuration rule obtained through meta-training. Comparisons between complete methods can combine these effects with other design differences.

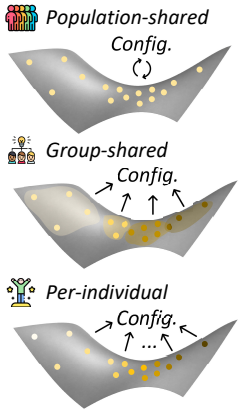

Figure 14: Three levels of configuration assignment granularity.

`EvoX MetaBench` exposes the assignment scope of Ψ through interchangeable components so that configuration granularity and the configuration rule can be examined under shared experimental conditions. This supports an investigation of how granularity affects performance without meta-training, how much meta-training adds at each granularity, and whether these two factors interact. Individual-level control further requires examining whether performance depends on matching configurations to individual states. A controller determines both which configurations to use and how to associate them with individual states, so the benefit of the learned rule need not arise entirely from individual-specific matching. Examining this correspondence alongside granularity and meta-training helps clarify how learned configuration rules exploit the heterogeneity permitted by per-individual assignment.

F.2 Experimental Setup

We compare learned and random configuration selection at population and individual granularities using DE/rand/1/bin. All four variants share a pool of 100 configurations formed by $F \in \{0.2, 0.4, \ldots, 2.0\}$ and $CR \in \{0.1, 0.2, \ldots, 1.0\}$. Meta-Pop and Meta-Ind select configurations using DDQN with individual-level state inputs, whereas Classic-Pop and Classic-Ind sample uniformly at random from the same pool. Both selection mechanisms operate every generation: population-level variants select one (F, CR) pair for the entire population, while individual-level variants select a pair for each individual. The controller architecture and main DDQN hyperparameters are summarized in Table 7. Experiments follow the common BBOB function split at 40 dimensions, with population sizes $N \in \{20, 50, 100, 200\}$. For each population size, we train separate Meta-Pop and Meta-Ind controllers for 100 episodes on the eight training functions. In RQ3, each low-level optimization run lasts 200 generations during both training and testing. All four variants are evaluated on the 16 held-out functions using the same 30 test seeds. For each test function, we report the mean and standard deviation of the final best-so-far fitness over these 30 runs.

Table 7: RQ3 config-control setup.

Component	*Setting*
Base optimizer	DE/rand/1/bin
Parameters	F and CR
Action space	10×10 pairs
Meta controller	DDQN
Hidden layers	$(64, 64)$
Learning rate	10^{-4}
Discount factor	$\gamma = 0.99$
Exploration	$\epsilon = 0.1$
Replay batch	32

For the assignment diagnostic, we select BBOB instances on which meta-training yields clear performance gains. Meta-Random reuses the trained Meta-Ind checkpoint for the corresponding population size and randomly permutes its complete (F, CR) tuples across individuals at each test-time AC step. This preserves the proposed configuration multiset at that step while disrupting its correspondence with individual states. We compare Meta-Ind and Meta-Random on the se-

lected instances across the tested population sizes, using convergence curves and final best-so-far fitness under the same evaluation protocol.

F.3 Detailed Results and Analysis

1) Individual-level configuration benefits search without meta-training. The experimental results are summarized in Tables 8 and 9. Across all four population sizes, Classic-Ind achieves lower mean final fitness than Classic-Pop on at least 14 of the 16 test functions. Since both variants resample uniformly from the same configuration pool at every generation, this improvement demonstrates the value of allowing different configurations to coexist within a population. Configuration heterogeneity therefore provides a source of performance gains even before a learned selection rule is introduced.

Table 8: Final best-so-far fitness on the 16 held-out 40D BBOB functions at $N = 20$ and $N = 50$ (mean $\pm$ standard deviation over 30 test runs; lower is better). Within each function and population size, the best mean is bold with gray shading, and the second best is underlined. Ties at the reported precision share the same marking.

	N = 20				N = 50			
ID	Classic-Pop	Classic-Ind	Meta-Pop	Meta-Ind	Classic-Pop	Classic-Ind	Meta-Pop	Meta-Ind
f_4	1.123e3±5.986e1	1.105e3±5.618e1	1.069e3±6.101e1	**9.137e2**±6.872e1	1.132e3±7.200e1	1.097e3±3.151e1	1.345e3±1.252e2	**8.301e2**±1.669e1
f_6	1.008e3±1.256e3	6.529e2±2.541e2	3.586e2±9.435e1	**2.577e2**±1.033e2	1.549e3±2.873e3	8.762e2±5.836e2	7.787e3±7.535e3	**3.034e2**±5.072e1
f_7	9.284e0±8.365e1	-1.139e1±5.227e1	-9.806e1±4.787e1	**-1.533e2**±2.957e1	9.194e1±9.717e1	4.224e1±6.575e1	3.175e2±1.241e2	**3.467e1**±4.543e1
f_8	6.123e3±3.452e3	4.396e3±1.860e3	2.605e3±1.901e3	**1.698e3**±1.132e3	6.581e3±3.087e3	4.865e3±1.201e3	3.965e4±1.023e4	**3.393e2**±7.414e1
f_9	5.944e3±6.218e3	3.410e3±1.407e3	1.470e3±1.115e3	**4.906e2**±1.605e2	8.731e3±5.124e3	5.227e3±1.721e3	7.160e4±3.908e4	**3.965e2**±9.220e1
f_{10}	9.164e5±3.244e5	8.708e5±2.401e5	1.033e6±3.796e5	**8.700e5**±3.890e5	1.073e6±3.684e5	**9.950e5**±2.433e5	1.993e6±4.577e5	1.325e6±2.404e5
f_{11}	4.704e2±8.654e1	4.902e2±8.915e1	**4.571e2**±5.863e1	5.323e2±9.246e1	4.600e2±5.952e1	4.595e2±5.354e1	**3.470e2**±4.128e1	3.707e2±4.109e1
f_{12}	2.588e7±7.880e6	1.962e7±7.275e6	1.176e7±8.362e6	**1.032e7**±7.118e6	2.390e7±8.754e6	2.123e7±5.314e6	5.722e7±1.686e7	**4.080e5**±1.174e5
f_{13}	1.194e3±1.373e2	1.135e3±1.112e2	9.783e2±1.213e2	**7.568e2**±1.198e2	1.300e3±1.769e2	1.214e3±1.067e2	1.186e3±2.194e2	**6.082e2**±3.561e1
f_{14}	3.107e2±3.749e0	3.097e2±3.029e0	3.034e2±2.563e0	**3.019e2**±2.095e0	3.129e2±4.273e0	3.123e2±3.140e0	3.249e2±5.408e0	**3.016e2**±1.117e0
f_{18}	2.234e2±4.338e0	2.231e2±3.997e0	2.151e2±3.646e0	**2.098e2**±3.133e0	2.221e2±4.794e0	2.218e2±2.212e0	2.253e2±4.949e0	**2.152e2**±2.307e0
f_{19}	8.190e2±7.020e-1	8.189e2±5.370e-1	8.182e2±5.910e-1	**8.175e2**±4.870e-1	8.192e2±9.760e-1	8.190e2±5.780e-1	8.250e2±2.618e0	**8.178e2**±4.030e-1
f_{20}	9.130e2±1.010e3	2.154e2±2.752e2	**1.748e1**±2.780e1	1.000e2±1.719e2	9.394e2±1.187e3	4.414e2±5.242e2	1.455e4±9.156e3	**8.010e0**±1.300e-1
f_{22}	-4.751e1±1.630e1	-4.293e1±1.896e1	-3.817e1±1.407e1	**-5.178e1**±1.414e1	-5.130e1±1.886e1	**-5.300e1**±1.910e1	-4.761e1±1.367e1	-4.557e1±1.256e1
f_{23}	9.523e1±5.450e-1	9.522e1±5.770e-1	9.529e1±4.280e-1	**9.515e1**±5.180e-1	9.473e1±5.060e-1	9.500e1±3.980e-1	9.480e1±4.260e-1	**9.470e1**±4.180e-1
f_{24}	5.046e2±4.936e1	5.036e2±3.824e1	4.663e2±3.173e1	**4.118e2**±2.201e1	5.092e2±4.320e1	5.113e2±4.264e1	5.753e2±1.258e2	**4.338e2**±2.630e1

Table 9: Final best-so-far fitness on the 16 held-out 40D BBOB functions at $N = 100$ and $N = 200$ (mean $\pm$ standard deviation over 30 test runs; lower is better). Within each function and population size, the best mean is bold with gray shading, and the second best is underlined. Ties at the reported precision share the same marking.

	N = 100				N = 200			
ID	Classic-Pop	Classic-Ind	Meta-Pop	Meta-Ind	Classic-Pop	Classic-Ind	Meta-Pop	Meta-Ind
f_4	1.120e3±6.097e1	1.100e3±3.378e1	**7.110e2**±2.678e1	7.163e2±1.300e1	1.122e3±5.835e1	1.099e3±2.386e1	1.269e3±1.164e2	**8.120e2**±5.219e1
f_6	1.473e3±1.411e3	9.906e2±6.374e2	**9.809e1**±6.270e1	2.041e2±4.890e1	1.135e3±9.247e2	8.877e2±6.117e2	**1.452e2**±6.607e1	7.013e3±5.082e3
f_7	1.148e2±9.068e1	9.704e1±6.756e1	**-1.835e2**±2.842e1	-8.301e1±2.405e1	1.541e2±9.842e1	1.086e2±5.920e1	**1.025e2**±1.602e2	3.283e2±1.276e2
f_8	7.803e3±3.812e3	6.412e3±1.277e3	**1.592e2**±3.618e1	2.684e2±3.876e1	8.691e3±4.080e3	6.872e3±1.270e3	**4.973e3**±1.534e4	3.065e4±1.042e4
f_9	9.614e3±4.720e3	8.578e3±2.355e3	**1.646e2**±4.421e1	2.005e2±6.268e1	1.266e4±5.441e3	9.708e3±2.918e3	**1.766e3**±6.638e3	7.908e3±2.066e4
f_{10}	1.112e6±3.795e5	1.068e6±2.351e5	**7.480e5**±2.040e5	9.148e5±1.200e5	1.183e6±2.890e5	1.077e6±1.842e5	**3.266e5**±1.085e5	8.835e5±3.039e5
f_{11}	4.465e2±6.319e1	4.309e2±6.631e1	3.187e2±3.861e1	**3.182e2**±3.518e1	4.039e2±6.135e1	3.932e2±4.417e1	**2.954e2**±3.404e1	3.037e2±3.096e1
f_{12}	2.734e7±1.090e7	2.323e7±3.312e6	7.781e5±1.010e6	**5.271e4**±1.510e4	2.584e7±7.014e6	2.312e7±3.735e6	3.216e7±2.064e7	**7.037e5**±3.848e6
f_{13}	1.341e3±1.627e2	1.296e3±7.822e1	1.033e3±1.718e2	**4.375e2**±1.352e1	1.392e3±1.454e2	**1.342e3**±7.498e1	2.581e3±2.447e2	1.955e3±1.272e2
f_{14}	3.146e2±4.721e0	3.133e2±2.759e0	2.990e2±8.710e-1	**2.983e2**±3.610e-1	3.155e2±4.125e0	3.141e2±2.427e0	3.081e2±6.142e0	**2.993e2**±5.489e0
f_{18}	2.219e2±3.881e0	2.217e2±2.008e0	**2.020e2**±1.929e0	2.129e2±2.096e0	2.222e2±2.714e0	2.218e2±2.052e0	2.218e2±5.313e0	**2.144e2**±1.078e1
f_{19}	8.195e2±9.260e-1	8.191e2±8.240e-1	**8.164e2**±3.270e-1	8.171e2±3.550e-1	8.194e2±7.630e-1	8.191e2±4.820e-1	8.196e2±2.108e0	**8.176e2**±1.991e0
f_{20}	1.684e3±1.829e3	7.056e2±4.311e2	7.544e0±2.790e-1	**7.480e0**±9.600e-2	1.781e3±1.567e3	**9.743e2**±5.731e2	5.164e3±8.533e3	1.499e4±7.925e3
f_{22}	**-5.741e1**±1.378e1	-5.675e1±1.453e1	-4.350e1±1.574e1	-4.872e1±6.815e0	-5.842e1±1.113e1	**-5.958e1**±1.259e1	-1.201e1±1.667e1	-5.118e1±6.557e0
f_{23}	9.455e1±4.440e-1	9.459e1±4.170e-1	**9.449e1**±4.040e-1	9.469e1±3.280e-1	9.437e1±3.620e-1	9.443e1±3.480e-1	9.445e1±4.440e-1	**9.434e1**±4.180e-1
f_{24}	5.176e2±3.578e1	4.989e2±3.180e1	**3.481e2**±2.591e1	4.054e2±2.001e1	5.161e2±3.596e1	**5.057e2**±2.149e1	9.583e2±1.651e2	7.885e2±6.692e1

2) Meta-training adds further gains, whose extent depends on population size. Meta-Ind outperforms Classic-Ind on 15, 14, and 14 functions at $N = 20, 50, 100$, respectively, which shows gains beyond random heterogeneous configuration. At $N = 20$ and $N = 50$, it also outperforms

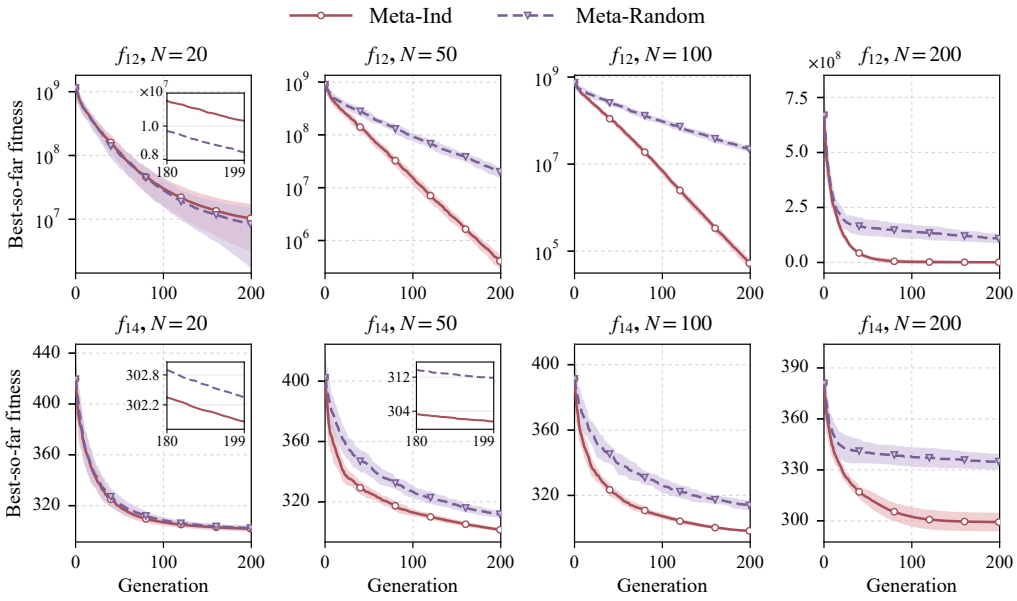

Figure 15: Convergence of META-IND and META-RANDOM on f_{12} and f_{14} across four population sizes. Lines show means over 30 runs, and shaded bands indicate one standard deviation. Insets enlarge the mean curves over the final 20 recorded generations. The first three f_{12} panels use logarithmic vertical axes; the remaining panels use linear axes, with ranges set per panel. Lower fitness is better.

META-POP on 14 functions each. Thus, finer configuration granularity can improve the learned optimizer under this training protocol. This advantage changes as the population grows: at $N = 100$, META-POP outperforms META-IND on ten functions, while increasing N from 100 to 200 worsens META-IND on twelve functions. Larger populations require configuration decisions for more individuals at each generation. This can make the added flexibility harder to exploit within the same 100-episode training budget. These results suggest that population size and training budget should be considered jointly when developing individual-level controllers.

3) Learned configuration assignment contributes to search performance. We further examine the learned policy on f_{12} and f_{14}, where META-IND performs best among the four original variants at every tested population size. Figure 15 shows that META-IND achieves lower mean final fitness in seven of the eight settings, with clear gains at $N = 50, 100, 200$. These results support matching configurations to individual states. Controller design should therefore consider configuration selection and individual-specific assignment jointly.

G Details of RQ4

G.1 Motivation

Within the AC-Step in Eq. 4, the feature constructor Φ maps population and process information to the state $\mathbf{s}_t$ observed by the configuration policy π. Existing AC methods mainly draw on fitness-space information ($\mathbf{F}$), solution-space information ($\mathbf{X}$), and optimization-process information ($\mathbf{P}$), whose contributions can be obscured by other design differences in complete-method comparisons. Our survey of 57 MetaBBO-AC studies reveals a long-tailed reuse pattern: most descriptors remain specific to their originating methods, while a small subset recurs across studies. After consolidating semantically equivalent descriptions, we summarize the ten most frequently reused features and representative studies in Table 10.

Table 10: The most frequently reused state features identified from existing AC methods.

Category	Feature	Description and references
Fitness space ($\mathbf{F}$)	Mean fitness	Population mean of fitness values (Chen et al., 2005; Sakurai et al., 2010; Yi et al., 2023; Yang et al., 2024b; Hu et al., 2023)
	Fitness STD	Population standard deviation of fitness values (Yi et al., 2023; Yang et al., 2024b; Ning et al., 2018; Yin et al., 2023)
	Fitness entropy	Population entropy of fitness values (Chen et al., 2005; Sakurai et al., 2010; Liu et al., 2019)
Solution space ($\mathbf{X}$)	Solution coordinates	Decision variables normalized by search bounds (Tian et al., 2023; Yin & Xiang, 2024; Liao et al., 2024)
	Inter-solution distance	Mean pairwise distance in normalized decision space (Yin & Xiang, 2024; Li et al., 2023; Hu et al., 2023; Ning et al., 2018)
	Distance to the best	Normalized decision-space distance to the incumbent best (Qiu et al., 2026; Li et al., 2023; Samma et al., 2016)
Optimization process ($\mathbf{P}$)	Optimization progress	Used FEs (Chen et al., 2005; Sakurai et al., 2010; Buzdalova et al., 2014; Yi et al., 2023; Liu et al., 2019; Qiu et al., 2026)
	Fitness improvement	Fitness gain over the latest update or recent window (Buzdalova et al., 2014; Yi et al., 2023; Liu et al., 2019; Qiu et al., 2026)
	Stagnation length	Consecutive updates without improvement (Qiu et al., 2026; Ning et al., 2018; Yin et al., 2023; Zhang et al., 2024b)
	Config-success history	Recent success statistics of selected actions (Liao et al., 2024; Fister et al., 2022; Jiao et al., 2023)

Alongside handcrafted descriptors, NeurELA (Ma et al., 2025a) explores learned state construction by mapping population solutions and fitness values to a compact representation. By exposing Φ as an interchangeable component, **`EvoX MetaBench`** enables these state-construction choices to be examined within a common AC pipeline. We investigate which feature groups are useful individually, whether their combinations provide complementary signals, and how handcrafted and learned feature constructors affect optimization performance on held-out problems.

G.2 Experimental Setup

We compare eight state representations for DDQN-controlled DE, PSO, and ES at a fixed population size of $N = 50$. The handcrafted variants, F, X, P, F+X, F+P, X+P, and F+X+P, comprise all non-empty combinations of the three groups containing the ten features in Table 10. NeurELA (Ma et al., 2025a) provides a learned representation of population solutions and fitness values. DDQN uses the same hidden-layer architecture and hyperparameters as in RQ3 (Table 7). Within each optimizer, all variants share the configuration space, reward function, and controller-training settings. The main experimental settings

Table 11: RQ4 state-representation setup.

Component	*Setting*
Base optimizers	DE, PSO, ES
Controller	DDQN
Action space	100 actions
State variants	Seven handcrafted representations and NeurELA

are summarized in Table 11. At each generation, DDQN selects one of 100 discrete configurations for each individual. For DE/rand/1, the configuration pool consists of (F, CR) pairs with $F \in \{0.2, 0.4, \ldots, 2.0\}$ and $CR \in \{0.1, 0.2, \ldots, 1.0\}$. For inertia-weight PSO, the controller selects (w, c_1, c_2) tuples from $w \in \{0.25, 0.5, 0.75, 1.0\}$ and $c_1, c_2 \in \{0.5, 1.0, \ldots, 2.5\}$, with the velocity clipping ratio fixed at 0.2. For ES, each individual undergoes Gaussian mutation followed by one-to-one greedy selection. The controller selects its mutation strength σ from 100 logarithmically spaced values in $[10^{-3}, 1]$. For NeurELA, we reuse the pretrained representation network from the original study and freeze its parameters. Only the DDQN network parameters are updated during training.

Experiments follow the common 40-dimensional BBOB function split. We train a separate controller for each optimizer–representation pair on the eight training functions for 100 episodes, with a training batch size of 16. Each low-level optimization run lasts 200 generations during both training and testing. We evaluate the final checkpoint on the 16 held-out functions using the same 30 test seeds. For each test function, we report the mean and standard deviation of the final best-so-far fitness over these runs.

G.3 Detailed Results and Analysis

State representations should be matched to the optimizer. Results in Tables 12, 13, and 14 show that the value of state information varies across optimizers. Among the three single-group variants, F achieves the lowest mean final fitness on 13 and 11 of the 16 functions for DE and ES, respectively, whereas P does so on 15 functions for PSO. One possible explanation concerns how state information complements the optimizer's search mechanism. DE already exploits solution differences through its variation operators, while fitness statistics can provide complementary feedback on population quality. For PSO, whose updates are guided by personal and social best positions, improvement and stagnation signals may help the controller assess whether the current search remains productive. The effects of combining feature groups further illustrate this dependence. For DE, F+X+P performs worse than F on all 16 functions. For ES, the full combination achieves the lowest mean final fitness among all eight variants on f_9, f_{10}, f_{11}, and f_{20}, although all three two-group combinations perform worse than F on these functions.

Table 12: Final best-so-far fitness of DDQN-controlled DE with different state representations on the 16 held-out 40D BBOB functions ($N = 50$; mean $\pm$ standard deviation over 30 test runs; lower is better). For each function, the best mean is bold with gray shading, and the second best is underlined.

	Handcrafted							Learned
ID	**F**	**X**	**P**	**F + X**	**F + P**	**X + P**	**F + X + P**	NeurELA
f_4	**7.213e2**±1.786e1	2.069e3±1.767e2	1.937e3±2.712e2	1.839e3±1.558e2	9.846e2±2.839e1	1.138e3±4.704e1	1.336e3±7.567e1	1.734e3±1.529e2
f_6	**2.357e2**±5.984e1	7.575e4±2.893e4	6.446e5±1.501e5	2.511e4±1.284e4	2.191e3±1.675e3	1.246e4±6.745e3	3.034e3±4.304e3	2.153e4±1.556e4
f_7	**-8.048e1**±3.568e1	7.854e2±1.182e2	3.512e2±1.636e2	5.728e2±1.083e2	3.071e2±7.968e1	3.989e2±7.737e1	2.958e2±9.224e1	4.634e2±1.650e2
f_8	**2.338e2**±4.262e1	2.948e5±5.432e4	8.449e4±2.676e4	1.144e5±2.977e4	4.960e3±1.259e3	1.340e4±4.700e3	5.910e4±1.969e4	6.271e4±3.378e4
f_9	**1.937e2**±4.731e1	2.096e5±4.404e4	1.827e4±7.980e3	2.036e5±6.545e4	2.352e4±8.329e3	5.459e4±1.304e4	2.595e4±1.010e4	1.364e5±6.466e4
f_{10}	**8.959e5**±2.005e5	2.454e6±4.469e5	3.904e6±1.291e6	2.667e6±4.896e5	1.616e6±4.309e5	1.987e6±3.122e5	1.575e6±3.376e5	2.559e6±5.705e5
f_{11}	3.688e2±5.816e1	5.366e2±7.469e1	**2.089e2**±3.496e1	5.793e2±5.106e1	4.708e2±5.454e1	4.244e2±4.566e1	4.150e2±7.084e1	5.592e2±7.501e1
f_{12}	**2.645e5**±5.993e5	1.999e8±3.427e7	3.344e8±7.303e7	1.465e8±2.855e7	2.402e7±5.071e6	4.468e7±9.729e6	7.442e7±1.666e7	1.385e8±4.924e7
f_{13}	**4.509e2**±2.324e1	2.632e3±9.687e1	2.553e3±1.949e2	2.304e3±1.751e2	1.460e3±1.596e2	1.963e3±7.195e1	2.116e3±1.928e2	2.394e3±2.947e2
f_{14}	**2.987e2**±4.768e-1	3.534e2±6.627e0	3.397e2±8.053e0	3.403e2±5.641e0	3.219e2±4.294e0	3.259e2±3.205e0	3.203e2±2.747e0	3.326e2±5.456e0
f_{18}	**2.139e2**±2.318e0	2.376e2±2.655e0	2.259e2±4.881e0	2.400e2±4.287e0	2.297e2±2.694e0	2.328e2±2.422e0	2.257e2±2.916e0	2.312e2±4.389e0
f_{19}	8.175e2±4.428e-1	8.338e2±2.777e0	**8.171e2**±5.509e-1	8.331e2±4.086e0	8.217e2±8.043e-1	8.234e2±9.479e-1	8.207e2±1.017e0	8.288e2±3.757e0
f_{20}	**8.090e0**±1.386e-1	6.543e4±1.046e4	2.119e4±1.122e4	6.973e4±1.585e4	1.520e3±9.535e2	1.001e4±4.006e3	5.276e3±2.873e3	2.708e4±1.358e4
f_{22}	**-4.985e1**±1.103e1	2.326e0±3.138e0	6.034e0±2.290e0	-2.376e1±1.873e1	-4.800e1±2.025e1	-9.380e0±1.150e1	-8.096e0±1.125e1	4.657e0±3.916e0
f_{23}	9.482e1±3.491e-1	**9.471e1**±5.452e-1	9.476e1±4.407e-1	9.484e1±5.006e-1	9.478e1±4.524e-1	9.477e1±5.274e-1	9.487e1±4.794e-1	9.485e1±4.323e-1
f_{24}	**4.083e2**±2.818e1	1.074e3±6.718e1	5.236e2±6.124e1	1.031e3±1.034e2	6.429e2±4.736e1	7.247e2±3.646e1	6.868e2±7.109e1	8.841e2±1.513e2

Learned representations exhibit a similar dependence on the optimizer: NeurELA achieves the lowest mean final fitness on eight functions for PSO, but performs worse than F on all 16 functions for DE and on 15 functions for ES. Combining features from multiple levels aims to describe the search state more comprehensively, yet these results show that increasing the available information does not

consistently improve configuration decisions. Effective state design therefore requires understanding how information supports the control of a particular optimizer. Identifying optimizer-specific and transferable state information can guide both feature composition and representation learning.

Table 13: Final best-so-far fitness of DDQN-controlled PSO with different state representations on the 16 held-out 40D BBOB functions ($N = 50$; mean $\pm$ standard deviation over 30 test runs; lower is better). For each function, the best mean is bold with gray shading, and the second best is underlined.

	Handcrafted							Learned
ID	**F**	**X**	**P**	**F + X**	**F + P**	**X + P**	**F + X + P**	NeurELA
f_4	1.596e3±1.464e2	1.297e3±7.893e1	1.190e3±1.317e2	1.231e3±4.033e1	1.450e3±8.689e1	1.259e3±4.533e1	1.299e3±6.403e1	**1.055e3**±1.487e2
f_6	5.037e5±9.509e4	4.682e3±9.740e3	9.818e2±1.575e3	1.648e3±3.548e3	7.456e3±1.571e4	1.307e3±1.418e3	1.996e3±3.350e3	**6.673e2**±6.608e2
f_7	1.669e2±8.995e1	1.328e1±8.159e1	-5.142e1±7.480e1	-5.185e1±4.464e1	6.230e1±6.167e1	-5.442e0±3.457e1	4.814e0±5.301e1	**-6.274e1**±6.057e1
f_8	3.569e4±9.743e3	1.126e4±4.653e3	2.733e3±1.955e3	8.554e3±2.003e3	2.648e4±1.473e4	1.028e4±2.657e3	1.131e4±3.408e3	**1.874e3**±3.769e3
f_9	4.415e3±2.265e3	9.516e3±3.117e3	9.143e2±3.498e2	6.971e3±1.561e3	1.776e4±5.196e3	1.104e4±2.279e3	1.138e4±3.447e3	**4.389e2**±1.880e2
f_{10}	2.389e6±8.299e5	5.849e5±1.750e5	**2.249e5**±8.739e4	5.468e5±1.590e5	6.674e5±1.792e5	6.382e5±6.752e4	7.042e5±1.723e5	2.686e5±2.341e5
f_{11}	**1.924e2**±3.084e1	2.659e2±6.532e1	2.264e2±4.558e1	2.267e2±5.223e1	3.439e2±9.029e1	2.674e2±5.695e1	2.680e2±4.493e1	2.158e2±5.857e1
f_{12}	2.223e8±5.851e7	6.202e7±2.227e7	2.872e7±1.136e7	3.443e7±8.338e6	8.516e7±3.558e7	3.787e7±1.545e7	4.145e7±1.387e7	**2.039e7**±1.692e7
f_{13}	2.148e3±1.584e2	1.265e3±1.557e2	1.103e3±1.389e2	1.018e3±1.042e2	1.731e3±1.655e2	1.422e3±9.210e1	1.520e3±1.008e2	**8.640e2**±3.520e2
f_{14}	3.262e2±4.540e0	3.084e2±3.081e0	3.070e2±2.618e0	**3.032e2**±1.209e0	3.155e2±4.395e0	3.076e2±1.691e0	3.093e2±2.171e0	3.045e2±4.025e0
f_{18}	2.260e2±3.035e0	2.231e2±3.434e0	2.196e2±3.944e0	2.171e2±3.038e0	2.240e2±3.145e0	2.168e2±1.967e0	2.193e2±2.649e0	**2.165e2**±3.615e0
f_{19}	8.163e2±4.751e-1	8.188e2±6.733e-1	**8.155e2**±9.040e-1	8.183e2±6.246e-1	8.199e2±7.040e-1	8.189e2±5.878e-1	8.194e2±3.936e-1	8.164e2±1.535e0
f_{20}	4.502e3±3.319e3	3.770e3±2.775e3	**7.151e1**±3.314e2	1.860e3±9.772e2	1.132e4±4.530e3	3.412e3±1.562e3	3.376e3±1.474e3	1.279e2±4.627e2
f_{22}	1.371e0±3.730e0	-2.496e1±1.050e1	**-4.291e1**±8.462e0	-3.574e1±9.326e0	-1.904e1±9.013e0	-3.480e1±1.009e1	-3.493e1±9.219e0	-4.209e1±1.244e1
f_{23}	9.448e1±5.368e-1	9.484e1±5.413e-1	**9.317e1**±5.943e-1	9.479e1±5.624e-1	9.486e1±4.278e-1	9.474e1±5.343e-1	9.483e1±3.886e-1	9.383e1±9.478e-1
f_{24}	4.645e2±4.826e1	5.123e2±4.152e1	**3.184e2**±5.905e1	4.909e2±3.755e1	6.016e2±4.281e1	5.241e2±1.744e1	5.357e2±3.048e1	3.559e2±8.467e1

Table 14: Final best-so-far fitness of DDQN-controlled ES with different state representations on the 16 held-out 40D BBOB functions ($N = 50$; mean $\pm$ standard deviation over 30 test runs; lower is better). For each function, the best mean is bold with gray shading, and the second best is underlined.

	Handcrafted							Learned
ID	**F**	**X**	**P**	**F + X**	**F + P**	**X + P**	**F + X + P**	NeurELA
f_4	1.496e3±6.128e1	**1.480e3**±6.326e1	1.791e3±9.930e1	2.004e3±1.313e2	1.746e3±1.002e2	2.182e3±1.606e2	1.691e3±7.766e1	2.983e3±3.967e2
f_6	**9.850e3**±5.284e3	1.272e4±5.053e3	4.136e4±1.403e4	1.696e5±6.778e4	4.497e4±2.085e4	3.030e5±5.364e4	2.806e4±1.680e4	7.657e5±1.905e5
f_7	**1.502e0**±2.602e1	2.969e1±3.141e1	2.182e2±4.380e1	4.371e2±9.626e1	2.361e2±6.686e1	5.087e2±1.142e2	7.095e1±5.264e1	1.274e3±2.390e2
f_8	**1.222e4**±2.663e3	1.659e4±3.115e3	5.949e4±1.086e4	1.085e5±2.538e4	6.062e4±1.845e4	1.243e5±2.083e4	1.275e4±4.603e3	2.649e5±6.671e4
f_9	9.078e3±2.094e3	1.542e4±2.721e3	6.192e4±1.113e4	1.392e5±2.005e4	5.641e4±1.458e4	6.970e4±1.356e4	**7.669e3**±1.877e3	1.628e5±2.837e4
f_{10}	4.903e5±9.211e4	6.621e5±8.663e4	1.188e6±2.348e5	1.804e6±4.361e5	9.333e5±2.196e5	1.437e6±4.343e5	**4.494e5**±1.058e5	5.132e6±1.343e6
f_{11}	3.014e2±3.696e1	3.196e2±3.407e1	3.670e2±4.004e1	3.759e2±3.911e1	3.567e2±3.529e1	3.301e2±3.618e1	**2.972e2**±4.025e1	3.152e2±3.804e1
f_{12}	**4.819e7**±8.967e6	5.257e7±8.273e6	1.197e8±1.608e7	1.632e8±2.055e7	1.287e8±2.761e7	2.658e8±4.737e7	7.135e7±1.702e7	5.161e8±9.810e7
f_{13}	**1.370e3**±7.360e1	1.524e3±8.435e1	2.048e3±8.469e1	2.544e3±1.682e2	1.944e3±1.475e2	2.458e3±1.595e2	1.472e3±1.161e2	3.165e3±2.487e2
f_{14}	**3.102e2**±1.765e0	3.112e2±1.788e0	3.223e2±2.947e0	3.310e2±3.648e0	3.219e2±3.796e0	3.353e2±5.152e0	3.128e2±1.898e0	3.575e2±8.654e0
f_{18}	2.352e2±4.006e0	2.317e2±3.181e0	2.321e2±3.355e0	2.329e2±3.710e0	**2.267e2**±3.036e0	2.492e2±5.710e0	2.468e2±5.961e0	2.509e2±7.475e0
f_{19}	8.210e2±6.092e-1	**8.209e2**±7.092e-1	8.239e2±1.007e0	8.290e2±1.900e0	8.235e2±1.725e0	8.278e2±1.281e0	8.233e2±9.420e-1	8.390e2±3.466e0
f_{20}	1.890e3±7.966e2	4.021e3±1.320e3	2.393e4±3.715e3	4.648e4±7.367e3	2.019e4±5.944e3	3.079e4±7.051e3	**1.337e3**±5.894e2	7.884e4±1.642e4
f_{22}	**-4.394e1**±7.150e0	-4.318e1±5.855e0	-1.414e1±6.979e0	5.500e0±3.776e0	-1.881e1±9.280e0	7.354e-1±2.692e0	-2.918e1±8.743e0	1.082e1±1.743e0
f_{23}	9.487e1±4.641e-1	9.466e1±5.048e-1	9.481e1±4.024e-1	**9.399e1**±3.348e-1	9.433e1±4.181e-1	9.477e1±3.646e-1	9.485e1±4.523e-1	9.445e1±4.885e-1
f_{24}	7.051e2±4.934e1	**6.472e2**±3.539e1	7.578e2±4.099e1	9.364e2±6.155e1	7.355e2±4.606e1	1.105e3±5.891e1	9.467e2±7.476e1	1.192e3±7.594e1

H Details of RQ5

H.1 Motivation

Within the AG-Step in Eq. 7, an algorithm artifact h serves both as an optimizer assessed by the evaluator $\mathcal{E}$ and as a parent supplied to the generation operator $\mathcal{G}$. Its *optimization performance* measures how well it solves the target problems, whereas its *generation value* concerns the quality of direct offspring obtained from it under a specified generation operator and proposal budget, as illustrated in Fig. 16. These quantities need not induce the same ranking: a parent's usefulness for generation depends on how the operator modifies its structure and decision logic. Whether prioritizing high-performing parents improves the design search may therefore depend on the generation operator.

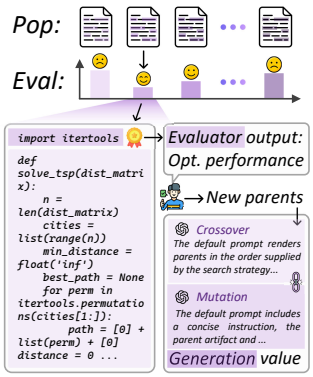

Figure 16: Decoupled evaluation of optimization performance and generation value for algorithm artifacts in AG.

`EvoX MetaBench` separates the search strategy $\mathcal{S}$, including parent selection, from the generation operators $\mathcal{G}$ through compatible artifact and evaluation interfaces. This enables their choices and interactions to be examined under shared experimental conditions. We investigate how closely a parent's optimization performance tracks its generation value, whether this relationship changes across generation operators, and how parent selection and generation jointly affect the complete search. These questions connect the quality of individual artifacts to their role in the process that produces subsequent algorithms.

H.2 Experimental Setup

We study constructive heuristic generation for Euclidean TSP50 under the settings in Table 15. Generated next-city scoring programs share a deterministic route constructor and are evaluated by mean tour length. All variants use the same LLM, decoding settings, program interface, and evaluator. The complete-search comparison crosses rank-weighted and uniform parent selection with local modification and structural rewriting. The four variants use matched initial populations and an identical elitist survivor-selection rule. From prespecified search checkpoints, we sample fixed parents balanced across variants and design-performance tertiles. Each parent is processed by both generation operators in repeated proposal batches; the diagnostic offspring do not enter the original search. Within each batch, the best offspring is selected by design-set performance and evaluated alongside its parent on the independent test instances. The mean test tour length of these selected offspring across batches estimates the parent's generation value under the corresponding operator and proposal budget. We report the association between parent performance and generation value, together with complete-search convergence and final test performance across independent AG runs. Invalid and duplicate proposals consume the proposal budget. Test instances are excluded from generation, parent sampling, and artifact selection.

Table 15: RQ5 algorithm-generation setup.

Component	*Experimental Setting*
Task	Constructive Euclidean TSP50
Design / test	32 / 128 instances
Population	8 artifacts
Search budget	64 proposals per run
Search repeats	5 independent runs per variant
Fixed parents	24 parent records in total
Diagnostic	3 batches of 4 proposals per parent–operator pair

H.3 Detailed Results and Analysis

1) The relationship between parent performance and generation value depends on the generation operator. Figure 9 shows that the Spearman correlation between parent test performance and generation value is 0.947 for local modification and 0.278 for structural rewriting. Under local modification, parents with shorter test tours generally yield offspring with shorter tours. Structural rewriting yields shorter mean offspring test tours than local modification for 16 of the 24 parent records, although its offspring quality is less closely related to parent performance. Pooling repeated

Table 16: Final TSP50 tour lengths over five AG runs (mean ± sample SD). Lower is better. Column bests are shaded and bold; runners-up are underlined.

Selection	Operator	Design	Test
Rank-weighted	Local	6.986±0.375	6.973±0.388
Uniform	Local	6.988±0.158	7.025±0.170
Rank-weighted	Structural	**6.608**±0.142	6.627±0.129
Uniform	Structural	6.620±0.191	**6.624**±0.242

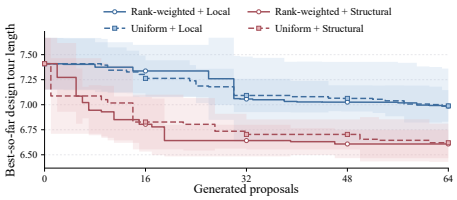

Figure 17: TSP50 search convergence (mean and sample SD over five AG runs).

records for the same parent preserves this difference in association, with correlations of 0.942 and 0.232, respectively. Thus, a parent's current performance provides different information about its generation potential under different operators.

2) Prioritizing high-performing parents does not consistently improve the complete search. Under local modification, rank-weighted selection achieves a slightly lower mean test tour length than uniform selection, but performs better in only two of the five paired runs. Under structural rewriting, their mean test results are nearly identical (Table 16). Structural rewriting achieves lower mean final test tour lengths under both selection strategies, and its mean design trajectories improve earlier in the search (Figure 17). Consequently, the strong association between parent and offspring performance under local modification does not by itself establish an advantage for biasing selection toward currently stronger parents.

3) LLM-based evolution incurs generation costs before evaluation. LLM-based mutation and recombination require inference time and tokens before candidate evaluation. Parent selection therefore controls where this generation budget is spent. The operator-dependent associations observed here motivate inexpensive estimates of generation value to guide parent selection.

H.4 Prompt Templates and Example Input

Finally, we document the prompt structure used in the RQ5 experiments (Figure 18). The complete example prompt texts follow this overview.

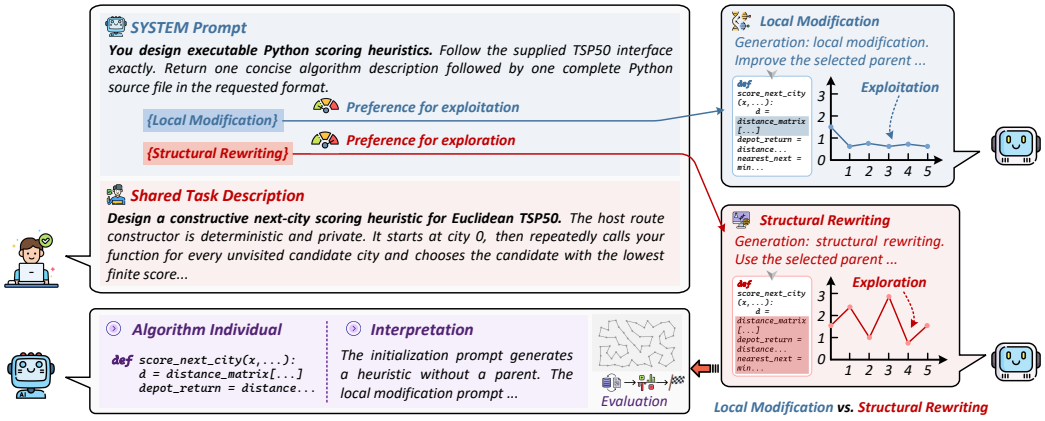

Figure 18: **Relationships among prompt components in RQ5.** The system prompt and shared task description support the operator-specific prompts for local modification and structural rewriting.

System Prompt

```
You design executable Python scoring heuristics. Follow the supplied TSP50
interface exactly. Return one concise algorithm description followed by one
complete Python source file in the requested format.
```

Shared Task Description

```
Design a constructive next-city scoring heuristic for Euclidean TSP50.

The host route constructor is deterministic and private. It starts at city 0,
then repeatedly calls your function for every unvisited candidate city and
chooses the candidate with the lowest finite score, breaking ties by city id.
The completed route is closed by returning to city 0. The outer objective is
the mean closed-tour length on 32 hidden design instances, so lower is better.

Required generated-artifact interface:

def score_next_city(
    current_city: int,
    candidate_city: int,
    unvisited: tuple[int, ...],
    tour: tuple[int, ...],
    coordinates: tuple[tuple[float, float], ...],
    distance_matrix: tuple[tuple[float, ...], ...],
) -> float:
    ...

Return a finite numeric score. Use only public arguments, deterministic logic,
and the Python standard library. Do not assume access to hidden test instances.
Lower scores should mean a more attractive next city.
```

Response Instruction

```
Respond using exactly one description block and one Python code block in this
order:
<algorithm_description>One concise sentence describing the candidate mechanism
.</algorithm_description>
‘‘‘python
# complete source code
‘‘‘
The source must define ‘score_next_city‘ at top level. Do not add text outside
those two blocks.
```

Initialization Prompt

```
Generation mode: independent initialization.
Create a deterministic and robust TSP next-city scoring heuristic from first
principles. Favor concise logic using geometric distance, cheap lookahead,
insertion cost, local density, or other information available through the
required interface.

Task:
{task_description}

{response_instruction}
```

Local Modification Prompt

```
Generation mode: local modification.
```

Local Modification Prompt (continued)

```
Improve the selected parent while preserving its main structure. Make a focused
change to weights, normalization, tie-breaking, lookahead, or a small decision
rule. Avoid a cosmetic rewrite.

Task:
{task_description}

<parent index="1" design_objective="{parent_design_score}">
<diagnostics>{parent_diagnostics}</diagnostics>
<source>
{parent_source_code}
</source>
</parent>

{response_instruction}
```

Structural Rewriting Prompt

```
Generation mode: structural rewriting.
Use the selected parent as evidence, but redesign the scoring mechanism
substantially. Change the feature set, control flow, or decision logic while
keeping the exact required interface and deterministic route-constructor
assumptions.

Task:
{task_description}

Selected parent:
<parent index="1" design_objective="{parent_design_score}">
<diagnostics>{parent_diagnostics}</diagnostics>
<source>
{parent_source_code}
</source>
</parent>

{response_instruction}
```

Example Parent Input

```
Selected parent:
<parent index="1" design_objective="7.84">
<diagnostics>mean_tour_length=7.84000000; instances=32</diagnostics>
<source>
def score_next_city(current_city, candidate_city, unvisited, tour, coordinates,
distance_matrix):
    d = distance_matrix[current_city][candidate_city]
    depot_return = distance_matrix[candidate_city][0]
    nearest_next = min(
        distance_matrix[candidate_city][j] for j in unvisited if j !=
candidate_city
    ) if len(unvisited) > 1 else depot_return
    return d + 0.25 * nearest_next + 0.05 * depot_return
</source>
</parent>
```