FROM EXPERIENCE TO KNOWLEDGE: ALGORITHM DESIGN FOR BLACK-BOX OPTIMIZATION WITH LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have shown promise in automated evolutionary algorithm (EA) design through their ability to generate algorithmic code and iteratively refine it based on performance feedback. However, many existing methods rely on iterative search to design algorithms for target problems, requiring repeated LLM calls and candidate evaluations. Moreover, design experience obtained on a target problem may not provide guidance for algorithm design on subsequent problems. To address these limitations, we propose FETK, a three-stage framework that distills problem algorithm design experience into knowledge, enabling an EA architecture to be generated for each subsequent problem with a single LLM call. First, the Experience Collection module collects problem design experience by searching EA architectures on training problems and recording landscape representations, architectural modifications, and performance outcomes. Second, the Knowledge Learning module learns design knowledge by distilling experience across training problems into natural-language guidance and evaluating candidate knowledge updates on validation problems before retaining them. Finally, the Knowledge Reuse module combines the learned knowledge with the landscape representation of each test problem to generate an EA architecture with a single LLM call. Experiments on BBOB and MA-BBOB problems demonstrate competitive performance compared with manually designed EA methods, learning-based design EA methods, and LLM-based design EA methods. https://anonymous.4open.science/r/fetk-DE22

1 INTRODUCTION

Evolutionary algorithms (EAs) have been extensively studied for challenging black-box optimization (BBO) problems Vikhar (2016), in which neither an explicit analytical form of the objective problem nor auxiliary derivative information is available to the optimizer Hansen et al. (2009). However, designing an effective EA for a target problem often requires human expertise, empirical knowledge, and repeated trial and error Camacho-Villalón et al. (2023). To reduce the reliance on manual algorithm engineering, automated algorithm design methods have been developed to construct effective evolutionary algorithms automatically Bezerra et al. (2016). Recent studies have leveraged large language models (LLMs) to generate heuristic components Liu et al. (2024) or complete optimization algorithms Stein & Bäck (2025), and iteratively refine them using performance feedback to automatically construct effective evolutionary algorithms.

Existing LLM-based algorithm design methods draw on both information about the target problem and experience from previous design attempts. Problem representations, including graph structural information, can provide guidance for generating solver code Li et al. (2025). Candidate implementations Stein & Bäck (2025) and natural-language descriptions offer examples of algorithmic designs Liu et al. (2024), while reflections translate evaluation feedback into suggestions for further refinement Ye et al. (2024). Performance feedback, when analyzed together with code features, can also reveal relationships between algorithm structure and performance, providing guidance for subsequent algorithm refinement van Stein et al. (2026). Search progress can also guide the evolution of strategies for selecting and modifying previous candidates Liu et al. (2026b).

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.

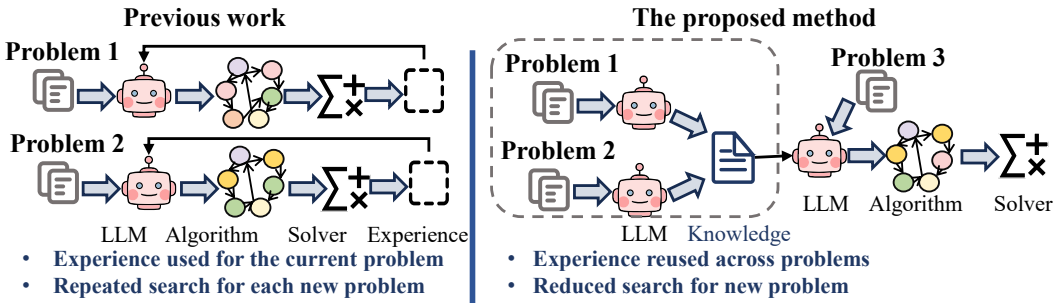


Figure 1: Motivation of the proposed method. Previous work relies on problem-specific experience and repeated search, while our method enables cross-problem experience reuse to reduce search effort on new problems.

These advances demonstrate how problem information and design feedback can guide LLM-based algorithm generation. As illustrated in Fig. 1, algorithm design produces not only a final EA but also experience about design modifications and their effects on optimization performance. Existing iterative methods mainly use this experience to refine candidates for the problem within the current search process Liu et al. (2024); Ye et al. (2024); Wang et al. (2025). When the target problem changes, the search must typically be repeated from scratch, requiring another sequence of LLM calls, candidate generations, and performance evaluations. Consequently, the experience obtained from one search is not effectively reused, and the repeated search increases LLM costs. A modification that improves one problem may be ineffective or even harmful on another, so directly transferring problem-specific experience is unreliable. The central challenge is therefore how to reuse design experience collected from previous problems to guide EA generation on new problems, while reducing the repeated iterative search for problem EA design.

To address this challenge, we propose FETK, a framework that learns knowledge for EA design experience. Rather than transferring the experience from an individual problem directly, FETK jointly considers design experiences from multiple problems to identify knowledge that captures patterns shared across their different landscapes. This knowledge can then help an LLM make informed design for a new problem based on experience accumulated across problems, instead of relying entirely on new rounds of trial and error. In this way, FETK aims to generate competitive EAs for BBO problems with less iterative refinement.

Our main contributions are summarized as follows:

- We propose FETK, a framework for learning design knowledge from algorithm design experience and reusing it to generate EAs for black-box optimization problems.
- We develop a three-stage framework consisting of: collecting design experience through architecture search on training problems; distilling and validating knowledge across problems; and using the learned knowledge to generate EA architectures for problems.
- Experiments on BBOB and MA-BBOB problems demonstrate competitive optimization performance against manually designed EA methods, learning-based design EA methods, and LLM-based design EA methods. Further analyses show that knowledge summarization improves optimization performance.

2 Related Work

2.1 LLM-Based Algorithm Design

LLM-based algorithm design has shown promising results in combinatorial optimization Liu et al. (2024) and continuous black-box optimization Stein & Bäck (2025). Existing methods combine code generation with evolutionary search over executable programs Romera-Paredes et al. (2024) or jointly evolve heuristic implementations and natural-language design descriptions Liu et al. (2024). Further studies investigate diversity management Dat et al. (2025), tree-based exploration Zheng et al. (2025), reflection on candidate performance Ye et al. (2024), and planning over sequences

of heuristic refinements Wang et al. (2025). Recent extensions explore complementary heuristic sets for diverse instances Liu et al. (2026a), evolving strategies for candidate generation Liu et al. (2026b). For continuous black-box optimization, LLMs have been used to generate complete metaheuristics Stein & Bäck (2025) and Bayesian optimization algorithms Li et al. (2026). Code structure and complexity have also been analyzed in relation to algorithm performance to produce guidance for refinement van Stein et al. (2026). Although problem information and design feedback provide guidance for algorithm generation and refinement, design experience gained on one problem may not transfer directly to others. Our work learns design knowledge from experiences across multiple problems to guide EA generation for BBO problems.

2.2 Experience Learning and Reuse in LLM Agents

This knowledge takes different forms. Voyager builds a library of executable skills for exploration, resource gathering, and crafting in Minecraft Wang et al. (2023). ExpeL extracts natural-language insights from training experiences to guide multi-hop question answering, household task execution, and online shopping Zhao et al. (2024). ReasoningBank distills reusable reasoning strategies from successful and failed interactions for web browsing and software engineering tasks Ouyang et al. (2026). SkillOpt learns task execution strategies for tasks such as spreadsheet manipulation, document question answering, and embodied interaction Yang et al. (2026). These studies use accumulated experience to guide LLM agents in subsequent task execution. In black-box optimization, however, the analytical form of the objective problem is unavailable, making it difficult to relate algorithm design choices and their performance outcomes to problem characteristics. Our framework uses exploratory landscape analysis (ELA) to characterize problems from sampled evaluations, providing context for learning when particular design choices are effective.

3 Method

3.1 Problem Formulation

We consider a D-dimensional continuous black-box optimization problem $\min_{\mathbf{x}\in\Omega} f(\mathbf{x})$, where $\Omega \subseteq \mathbb{R}^D$ is the bounded decision space and f can only be accessed through problem evaluations. We adopt the modular EA architecture representation of NeuroEA Tian et al. (2025), in which an algorithm is specified by an operator graph and its associated continuous operator parameters. Let $\mathcal{O}$ denote the set of three *Operator* blocks, including the block Exchange, the block Crossover, and the block Mutation. A valid architecture contains at least one Population block, one operator block, and one Selection block. Let $\mathcal{S}(\mathcal{O})$ denote the corresponding modular EA design space. Each EA architecture $\mathcal{A} \in \mathcal{S}(\mathcal{O})$ is represented as $\mathcal{A} = (\mathbf{o}, \mathbf{E}, \mathbf{R})$, where $\mathbf{o}$ specifies the selected operator nodes from $\mathcal{O}$, $\mathbf{E}$ denotes the directed connections between operators, and $\mathbf{R}$ specifies the associated information-routing ratios. The architecture is associated with continuous operator parameters $\boldsymbol{\theta}(\mathcal{A})$, which control the numerical behaviors of the selected operators and are optimized after architecture generation. Details of the EA architecture are in the **Appendix A**.

Our goal is to learn reusable design knowledge K from $\mathcal{E}_{\text{train}}$ such that an architecture generator LLM_{arc} can construct effective EA architectures for held-out problems:

$$K^* = \arg\min_{K} \operatorname*{Agg}_{j\in D_{val}} \mathcal{J}(\text{LLM}_{\text{arc}}(z_j, K); f_j, B_j), \tag{1}$$

subject to $\text{LLM}_{\text{arc}}(z_j, K) \in \mathcal{S}(\mathcal{O})$, where LLM_{arc} generates an EA architecture, z_j denotes the landscape representation of problem f_j, and $\mathcal{J}$ measures the performance of the generated EA after parameter optimization under budget B_j.

3.2 Overview

To address the knowledge learning problem formulated in Eq. 1, we propose FETK, illustrated in Fig. 2. FETK consists of three stages: experience collection, knowledge learning, and knowledge reuse. During training, experience collection and knowledge learning alternate across batches. After training, the learned knowledge is frozen and applied to unseen test problems in the knowledge reuse stage. In the first stage, FETK performs architecture search on training problems and collects design experiences. In the second stage, these experiences are analyzed by an LLM to extract candidate

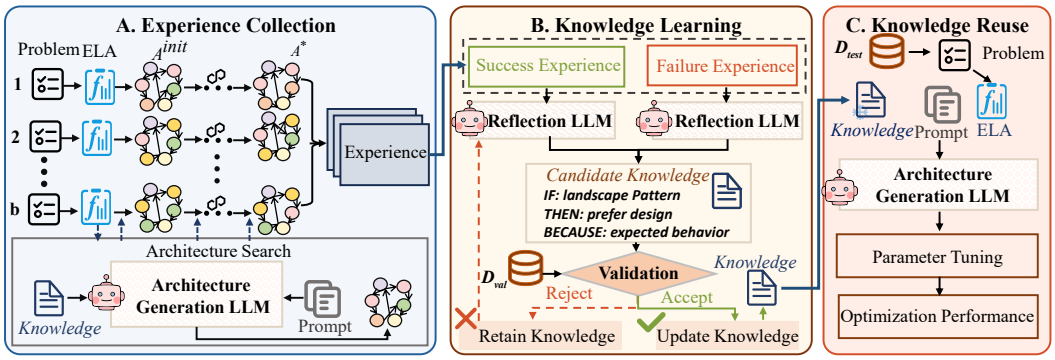


Figure 2: Overview of the FETK framework. **(A)** Experience Collection for generating problem-specific design experience. **(B)** Knowledge Learning and Validation for distilling and validating design knowledge. **(C)** Knowledge Reuse for EA architecture generation on problems.

design knowledge, which is further evaluated on validation problems to retain reliable knowledge updates. After training, the learned knowledge is fixed and reused in the third stage, where it guides the LLM to generate an EA architecture for problems based on their landscape representations.

3.3 Experience Collection

The Experience Collection stage aims to gather problem-specific design experience about effective EA architectures. Rather than merely recording final performance, it seeks to capture which architecture modifications lead to improved architectures under a problem landscape. Training continues for E epochs over a training set containing N_{tr} problems. At each epoch e, the training problems are randomly shuffled and partitioned into $J = \lceil N_{\mathrm{tr}}/b \rceil$ batches of size b. For each problem $f_i \in \mathcal{B}_{e,j}$, we uniformly sample $N_s = 10D$ points from its bounded decision space and evaluate their objective values. We construct a landscape representation $\mathbf{z}_i = [\mathbf{q}_i, \mathbf{e}_i, \mathbf{p}_i]$, where $\mathbf{q}_i$ contains summary statistics of the observed objective values, $\mathbf{e}_i$ contains numerical descriptors computed from the sampled decision–fitness observations, and $\mathbf{p}_i$ encodes a semantic profile derived from these descriptors. The profile summarizes modality, ruggedness, separability, conditioning, interaction-model availability, and the fitness–distance relationship.

Conditioned on $\mathbf{z}_i$ and $K_{e,j-1}$, the architecture generation LLM $\mathrm{LLM}_{\mathrm{arc}}$ generates an architecture retained as $\mathcal{A}_i^{\mathrm{init}} = \mathrm{LLM}_{\mathrm{arc}}(\mathbf{z}_i, K_{e,j-1})$. Starting from $\mathcal{A}_i^{\mathrm{init}}$, the architecture generation LLM performs $T_{\max}$ architecture generations. Each refinement is guided by the current architecture $\mathcal{A}_i^{\mathrm{cur}}$ (updated as $\mathcal{A}_i^{\mathrm{cur}} = \mathcal{A}_i^{t-1}$), the historical-best architecture $\mathcal{A}_i^{\mathrm{best}}$ (updated as $\mathcal{A}_i^{\mathrm{best}} = \mathcal{A}_i^{t-1}$ if $\ell_i(\mathcal{A}_i^{t-1}) - \ell_i(\mathcal{A}_i^{\mathrm{best}}) < 0$, otherwise unchanged), and their fitness values $\ell_i(\mathcal{A}_i^{\mathrm{cur}})$, $\ell_i(\mathcal{A}_i^{\mathrm{best}})$.

$$\mathcal{A}_i^{(t)} = \mathrm{LLM}_{\mathrm{arc}}(\mathbf{z}_i, K_{e,j-1}, \mathcal{A}_i^{\mathrm{cur}}, \mathcal{A}_i^{\mathrm{best}}, \ell_i(\mathcal{A}_i^{\mathrm{cur}}), \ell_i(\mathcal{A}_i^{\mathrm{best}}), q = 0), \tag{2}$$

where $\mathcal{A}_i^{(t)}$ denotes the architecture generated at generation t, and $\ell_i(\mathcal{A})$ denotes the final optimality gap achieved by architecture $\mathcal{A}$ on problem f_i. q is a binary decision variable controlling the architecture update strategy. Specifically, $q = 0$ indicates that the current architecture is refined while preserving its overall structure, whereas $q = 1$ indicates that a new architecture is generated. After T_s consecutive non-improving generations, the search initiates architecture reconstruction by setting $\mathcal{A}_i^{\mathrm{cur}} = \mathcal{A}_i^{\mathrm{best}}$ and resetting the stagnation counter to zero. An architecture is then generated as $\mathcal{A}_i^{(t)} = \mathrm{LLM}_{\mathrm{arc}}\big(\mathbf{z}_i, K_{e,j-1}, \mathcal{A}_i^{\mathrm{cur}}, \mathcal{A}_i^{\mathrm{best}}, \ell_i(\mathcal{A}_i^{\mathrm{cur}}), \ell_i(\mathcal{A}_i^{\mathrm{best}}), q = 1\big)$. The search then explores this direction for T_g generations, following the refinement update rules described above. After $T_{\max}$ generations, we select the best architecture $\mathcal{A}_i^{\star} \in \arg\min_{\mathcal{A} \in \{\mathcal{A}_i^{(1)}, \ldots, \mathcal{A}_i^{(T_{\max})}\}} \ell_i(\mathcal{A})$, and compute its performance improvement: $g_i = \ell_i(\mathcal{A}_i^{\mathrm{init}}) - \ell_i(\mathcal{A}_i^{\star})$. A positive g_i indicates an improvement over the initial architecture. The resulting design experience is summarized as $e_i = (\mathbf{z}_i, \mathcal{A}_i^{\mathrm{init}}, \mathcal{A}_i^{\star}, g_i)$. Experiences with $g_i > 0$ constitute positive examples of effective design architectures, while those with $g_i \leq 0$ provide contrastive evidence. For these non-improving cases, $A_i^{\star}$ in the experience tuple is replaced by the final generated architecture. Recalculate $g_i = \ell_i(\mathcal{A}_i^{\mathrm{init}}) - \ell_i(\mathcal{A}_i^{\star})$. Details of the ELA calculation are provided in the **Appendix B**.

3.4 KNOWLEDGE LEARNING

Design experience derived from an individual problem may capture landscape-specific modifications rather than generalizable design principles. We therefore compare experiences across multiple problems to extract knowledge. For each batch $\mathcal{B}_{e,j}$, we partition the problem-specific experiences into a success set $\mathcal{E}^+_{e,j} = \{e_i \mid f_i \in \mathcal{B}_{e,j},\, g_i > 0\}$ and a failure set $\mathcal{E}^-_{e,j} = \{e_i \mid f_i \in \mathcal{B}_{e,j},\, g_i \le 0\}$. Conditioned on $K_{e,j-1}$ and the rejected-update memory $\mathcal{M}_{e,j-1}$, the reflection LLM LLM_{ref} analyzes the two sets separately and generates success-driven and failure-driven patches $P^+_{e,j} = LLM_{ref}(K_{e,j-1}, \mathcal{E}^+_{e,j}, \mathcal{M}_{e,j-1})$ and $P^-_{e,j} = LLM_{ref}(K_{e,j-1}, \mathcal{E}^-_{e,j}, \mathcal{M}_{e,j-1})$. Each patch $P^\pm_{e,j} = (\Delta^\pm_{e,j}, \rho^\pm_{e,j})$ contains an ordered edit sequence $\Delta^\pm_{e,j}$, where each edit $\delta = (o, \tau, \kappa)$ applies one of four operations: $o \in \{\texttt{append}, \texttt{insert}, \texttt{replace}, \texttt{delete}\}$. Here τ specifies the target content and κ specifies the inserted or replacement content.

For each polarity, the resulting patches are first consolidated independently: $\bar{P}^-_{e,j} = \mathrm{Agg}^-(P^-_{e,j,1}, \ldots, P^-_{e,j,m_-})$, $\bar{P}^+_{e,j} = \mathrm{Agg}^+(P^+_{e,j,1}, \ldots, P^+_{e,j,m_+})$. They are then jointly merged by a failure-prioritized aggregation operator: $P^\star_{e,j} = \mathrm{Agg}(K_{e,j-1}, \bar{P}^-_{e,j}, \bar{P}^+_{e,j})$, where failure-driven edits take precedence over success-driven edits when the two groups conflict. Let $P^\star_{e,j} = (\Delta^\star_{e,j}, \rho^\star_{e,j})$. The merged edit pool is subsequently ranked and clipped: $\widehat{\Delta}_{e,j} = \mathrm{Top}_{L_{e,j}}(\Delta^\star_{e,j})$, with $L_{e,j}$ denoting the step-wise edit budget. The candidate knowledge state is obtained by sequentially applying the retained edits: $\widetilde{K}_{e,j} = \mathrm{Apply}(K_{e,j-1}, \widehat{\Delta}_{e,j})$. Because $\mathrm{Apply}(\cdot)$ operates directly on the Markdown rule document, $\widetilde{K}_{e,j}$ preserves the same rule-based representation as $K_{e,j-1}$. Finally, the candidate is evaluated by the validation gate $a_{e,j} = \mathcal{G}(\widetilde{K}_{e,j}; \mathcal{D}_{\mathrm{val}})$, which determines whether the candidate is accepted as the next knowledge state.

A candidate update may improve training problems yet fail to generalize. We therefore validate $\widetilde{K}_{e,j}$ on the validation set $\mathcal{D}_{\mathrm{val}}$ through paired comparisons. For each validation problem f_v, the current knowledge $K_{e,j-1}$ and the candidate knowledge $\widetilde{K}_{e,j}$ guide the architecture generation LLM to generate an architecture under the same observable probe, random seed, and problem-evaluation budget. This pairing removes irrelevant variation and yields a scale-free improvement

$$\gamma_v = \log_{10} \frac{\max\{\ell_v(K_{e,j-1}), \epsilon\}}{\max\Big\{\ell_v(\widetilde{K}_{e,j}), \epsilon\Big\}}, \qquad v \in \mathcal{D}_{\mathrm{val}}, \tag{3}$$

where $\ell_v(K)$ is the final optimality gap achieved by the architecture generated under knowledge state K. Thus, $\gamma_v > 0$ indicates that the candidate knowledge improves performance on f_v.

To avoid accepting a candidate that helps only a few problems or causes a severe regression, we summarize the paired gains by two statistics:

$$W_{e,j} = \sum_{v \in \mathcal{D}_{\mathrm{val}}} \mathbb{I}[\gamma_v > 0], \qquad \widetilde{\gamma}_{e,j} = \min_{v \in \mathcal{D}_{\mathrm{val}}} \gamma_v. \tag{4}$$

Let τ_{reg} denote the regression thresholds. The candidate is accepted, i.e., $a_{e,j} = 1$, only if all validation problems are successfully evaluated, $W_{e,j} \ge W_{\min}$, and $\widetilde{\gamma}_{e,j} \ge -\tau_{\mathrm{reg}}$; otherwise, $a_{e,j} = 0$ and we keep $K_{e,j} = K_{e,j-1}$.

When a multi-edit candidate is rejected, we validate its component edits sequentially in ranked order and retain those that individually pass the same gate. Rejected edits are recorded in $\mathcal{M}_{e,j}$ to discourage similar updates in subsequent reflection steps. Accepted knowledge persists across batches and epochs. After E epochs, the knowledge state with the best held-out validation performance is retained as $K^\star$. Examples of the knowledge are in the **Appendix C**.

3.5 KNOWLEDGE REUSE

After knowledge learning, the obtained knowledge is frozen as $K^\star$. For an unseen test problem f_t, FETK performs knowledge-conditioned architecture generation instead of iterative architecture search. Given f_t, we first characterize its landscape using the same sampling and feature extraction procedure as in training, obtaining the landscape representation $\mathbf{z}_t$. The sampled solution evaluations are counted toward the total evaluation budget. Conditioned on $\mathbf{z}_t$ and $K^\star$, the architecture

generation LLM constructs a problem-specific EA architecture: $\mathcal{A}_t = \mathrm{LLM}_{\mathrm{arc}}(\mathbf{z}_t, K^\star)$. The generated architecture may contain continuous operator parameters. To adapt these numerical parameters while preserving the generated architecture, we perform bounded GA-based parameter optimization: $\theta_t^\star = \mathrm{HPO}_{GA}(\mathcal{A}_t)$, where HPO_{GA} optimizes only the continuous parameters associated with the fixed architecture under a predefined parameter optimization budget. Thus, FETK generates the architecture once for each test problem and subsequently tunes only its continuous parameters. The generated architecture A_t and optimized parameters $\theta_t^\star$ jointly define the final algorithm, which is evaluated on f_t under the prescribed final-run budget.

4 EXPERIMENT

In this section, we discuss the following research questions : **RQ1:** How effective is FETK compared with representative EA design methods? **RQ2:** Can learned knowledge reduce iterative architecture search while preserving EA design quality? **RQ3:** How does each component contribute to performance? **RQ4:** How sensitive is FETK to the knowledge validation criterion? **RQ5:** How well does FETK generalize to optimization problems with different landscape characteristics?

4.1 EXPERIMENTAL SETTINGS

Test Problems: Two sets of benchmark problems were used: The BBOB benchmark suite consists of 24 noiseless black-box optimization problems (F1–F24); eight mixed-landscape problems generated using MA-BBOB Vermetten et al. (2025).

Baselines: We compared FETK with a manually designed EA and two groups of automated EA design methods. **Manually designed EAs**, we compared with GA Holland (1992) and NeuroEA Tian et al. (2025); **Learning-based design EAs**, we compared with AutoV Tian et al. (2023), RLDE-AFL Guo et al. (2025a), ConfigX Guo et al. (2025b), and Q-Mamba Ma et al. (2025); **LLM-based design EAs** , We compared with LLaMEA-(1+1) and LLaMEA-(1,1) Stein & Bäck (2025).

Parameter Settings: The 24 BBOB problems were split into 12 training, 6 validation, and 6 test problems. For each problem, $N_s = 10D$ samples were used for landscape characterization. Knowledge learning was performed for $E = 2$ epochs with a batch size of $b = 4$. The stagnation threshold and reconstruction budget were set to $T_s = T_g = 5$ generations, and each training problem underwent $T_{\max} = 20$ generations. The edit budget was set to $L_{e,j} = 2$. Candidate knowledge updates were validated on the six held-out problems and accepted when they improved at least $W_{\min} = 3$ problems and satisfied the regression constraint $\tau_{\mathrm{reg}} = 1$. Continuous operator parameters were optimized with a GA, using a population size of 50 and a budget of 5000 parameter evaluations; this stage does not invoke the LLM or modify the architecture structure Tian et al. (2025). Final evaluation was conducted in $D = 10$ dimensions with population size $N = 100$ and a budget of $\mathrm{MaxFEs} = 1000D$. Results were reported over 30 independent optimization runs, with the same final-run budget of $1000D$ objective-function evaluations for all compared methods. Algorithm design and parameter tuning followed the respective methods' protocols, and their costs were excluded from this final-run budget. For LLM-based methods, LLaMEA-(1+1) and LLaMEA-(1,1) were implemented with GLM-5.3 as the LLM backbone. FETK was evaluated with both GLM-5.3 and GPT-5.6-Sol to investigate the impact of different LLM backbones. All knowledge was learned exclusively from the 10-dimensional BBOB training problems and reused without further updates on all subsequent experiments.

4.2 OVERALL PERFORMANCE COMPARISON

To evaluate whether *FETK* could generate effective evolutionary algorithms for black-box optimization problems, we compared two *FETK* variants, namely *FETK-(GLM-5.3)* and *FETK-(GPT-5.6-Sol)*, with manually designed EAs, learning-based design methods, and LLM-based design methods on test problems. *FETK-(GLM-5.3)* achieved the best or tied-best mean performance on F2, F9, and F13, whereas *FETK-(GPT-5.6-Sol)* obtained the best mean performance on F16 and F17. These results demonstrated that *FETK* could generate competitive EA architectures for 10-dimensional black-box optimization. Additional results on 50-dimensional problems were provided in **Appendix E.1**, while results under different test splits were reported in **Appendix E.2**.

Table 1: Optimization results on six test problems. Each entry reports the mean and standard deviation. Lower values are better; the best and second-best means are shown in bold and underlined. The last column reports significant wins/losses/ties ($+/-/=$) of FETK (GLM-5.3 / GPT-5.6-Sol) against each baseline using the Wilcoxon rank-sum test at $\alpha = 0.05$.

Method	F2	F9	F13	F14	F16	F17	Sig.
Manually designed EAs							
GA	5.20E-01 5.58E-01	3.66E+01 2.15E+01	2.69E+01 2.08E+01	6.30E-03 2.94E-03	3.55E+00 1.86E+00	6.12E-01 4.17E-01	6/0/0 6/0/0
NeuroEA	**0.00E+00** 0.00E+00	7.52E+00 4.91E+00	6.66E+00 4.41E-01	1.63E-03 9.62E-04	1.15E+01 1.90E-01	2.20E-02 2.40E-03	5/0/1 5/1/0
Learning-based design EAs							
AutoV	7.80E-08 7.56E-08	2.77E+01 1.12E+01	3.93E+01 2.49E+01	2.80E-03 1.62E-03	5.55E+00 2.73E+00	4.54E-01 2.11E-01	6/0/0 5/1/0
RLDE-AFL	6.45E-05 7.82E-06	2.16E+00 7.98E-01	5.91E+00 5.81E+00	1.02E-03 1.60E-04	4.91E+00 2.22E+00	9.34E-02 6.66E-02	6/0/0 5/1/0
ConfigX	3.18E+02 7.95E+02	3.62E+01 8.52E+01	5.66E+01 8.99E+01	1.43E-01 7.95E-02	7.81E+00 4.00E+00	7.82E-01 9.89E-01	6/0/0 6/0/0
Q-Mamba	8.02E+03 5.60E+03	1.95E+01 2.72E+01	2.40E+01 2.52E+01	1.62E-01 6.85E-01	8.69E+00 2.33E+00	1.43E+00 6.32E-01	6/0/0 6/0/0
LLM-based design EAs							
LLaMEA-(1+1)	6.94E-07 3.09E-08	5.50E-01 1.40E+00	1.45E+01 1.26E+01	5.78E-06 3.21E-06	7.82E+00 3.97E+00	2.44E+00 1.36E+00	4/1/1 3/2/1
LLaMEA-(1,1)	3.25E-10 5.43E-10	5.11E-01 3.11E-01	1.08E+00 1.43E+00	**1.75E-06** 2.21E-06	9.53E+00 8.33E-01	1.74E-01 5.11E-01	4/1/1 3/2/1
FETK-(GLM-5.3)	**0.00E+00** 0.00E+00	**1.57E-01** 2.64E-01	**2.15E-02** 9.54E-03	2.74E-04 1.61E-04	3.12E-01 2.27E-01	9.27E-03 4.56E-03	–
FETK-(GPT-5.6-Sol)	8.72E-02 3.38E-02	4.09E-01 7.63E-01	3.00E-01 1.50E-01	4.45E-04 1.63E-04	**1.42E-01** 1.70E-01	**5.28E-04** 3.38E-04	–

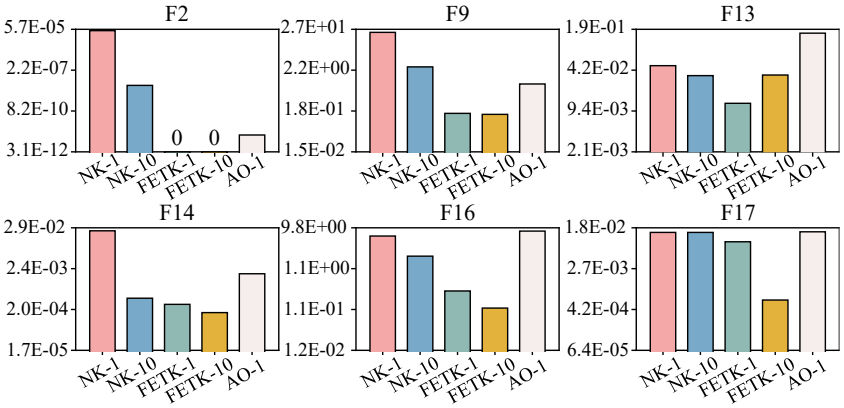


Figure 3: Optimization performance of different knowledge reuse and architecture search settings.

4.3 Reducing Iterative Architecture Search with Learned Knowledge

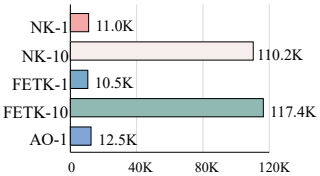


Figure 4: Token consumption of the compared variants.

To investigate whether the learned knowledge could reduce iterative architecture search while preserving design quality, we compared *FETK* with *NK-1*, which generated an architecture without learned knowledge; *NK-10* further evolved this architecture for 10 generations; *FETK-1* generated an architecture with a single LLM call using the learned knowledge; *FETK-10* further evolved the generated architecture for 10 generations; and *AO-1* directly provided the best architectures obtained on the training problems to the LLM without explicit knowledge summarization. As shown in Fig. 3, *FETK-1* achieved performance comparable to NK-10 and outperformed *AO-1*, although both *FETK-1* and *AO-1* generated architectures in a single LLM call. *FETK-10* produced only limited additional improvement over

FETK-1. As illustrated in Fig. 4, *FETK-1* required substantially fewer tokens than *NK-10* because it generated the architecture once instead of performing repeated test-time refinement. These results suggested that knowledge-guided architecture generation with a single LLM call achieved performance comparable to the iterative baselines while requiring fewer LLM tokens. Additional stability analyses, including architecture generation under fixed knowledge and architectures generated using knowledge learned from three separate training runs, are provided in **Appendix E.3,E.4**.

4.4 Ablation Study

To evaluate the contribution of the main components in *FETK*, we conducted ablation studies by removing architecture search, knowledge validation, and landscape information individually. Specifically, *w/o AS* disabled the 20-generation architecture search and treated all initial LLM-generated architectures as successful experiences for knowledge induction. *w/o KV* bypassed knowledge validation and directly accepted all generated knowledge. *w/o ELA* removed the landscape representation $\mathbf{z}_i$. All other settings were unchanged. As shown in Fig. 5, *FETK* generally achieved better convergence and lower final fitness than the three ablated variants across the six test problems, while removing any of the three components led to performance degradation on multiple problems. These results demonstrated that architecture search provided higher-quality design experience, knowledge validation helped retain more reliable and transferable knowledge, and landscape information enabled problem-specific architecture generation, with all three components contributing to the overall performance of *FETK*.

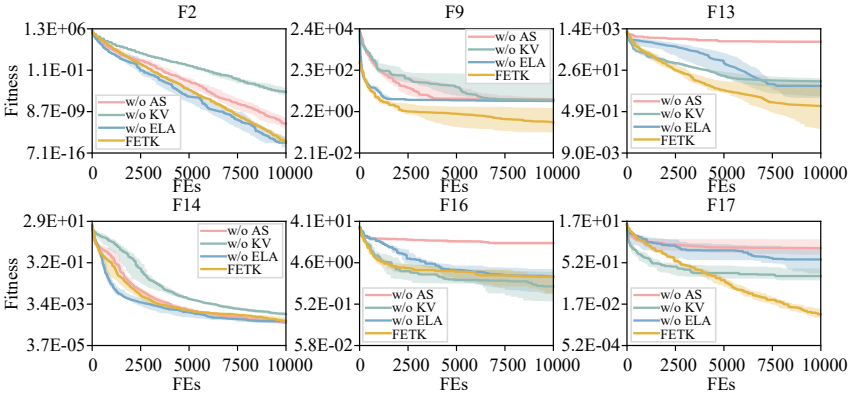


Figure 5: Ablation results of FETK on six test problems by removing architecture search, knowledge validation, and ELA information. Lower fitness is better.

4.5 Sensitivity Analysis of Knowledge Validation

To investigate the sensitivity of *FETK* to the validation criterion, we varied the minimum number of validation problems on which a candidate knowledge update had to improve performance before being accepted. Specifically, we evaluated six non-degenerate settings, denoted by *W0–W5*, corresponding to $W_{\min} \in \{0, 1, \ldots, 5\}$. We also examined the boundary case *W6*, which required improvement on all six validation problems. Since no candidate update satisfied this criterion, W6 accepted no knowledge updates and therefore left the initial knowledge unchanged; we omit this degenerate setting from the figure. As shown in Fig. 6, *W3* achieved the highest average performance across the six test problems among the reported settings. Performance was lower under both less restrictive and more restrictive criteria. This pattern is consistent with a trade-off between admitting potentially useful knowledge updates and filtering insufficiently validated ones. Overall, within the evaluated settings, $W_{\min} = 3$ achieved the best empirical performance. The normalization procedure is detailed in **Appendix E.3**.

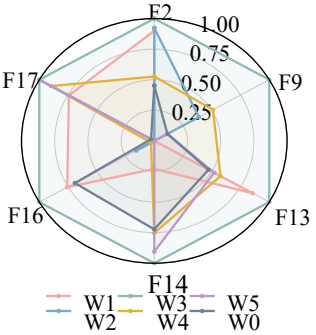


Figure 6: Sensitivity of FETK to the validation threshold W on six test problems. A bigger number is better.

Table 2: Optimization results on eight MA-BBOB test instances. Lower values are better. Each entry reports the mean on the first line and the sample standard deviation in a smaller font on the second line. The best mean value is highlighted in bold, while the second-best mean value is underlined.

Method	MA_1	MA_2	MA_3	MA_4	MA_5	MA_6	MA_7	MA_8
Manually designed EAs								
GA	2.24E-01 1.41E-01	2.60E-02 1.77E-02	<u>9.09E-02</u> 4.01E-03	<u>7.90E-02</u> 7.06E-04	3.42E-01 9.80E-02	1.99E-02 3.53E-03	2.27E-04 5.52E-05	3.02E-01 1.13E-01
NeuroEA	6.45E-03 2.70E-03	4.87E-01 4.38E-01	8.76E-01 4.80E-01	9.20E-02 2.66E-02	2.76E-01 7.72E-02	2.31E+00 1.80E+00	2.99E-01 1.04E-01	1.48E-01 2.31E-01
Learning-based design EAs								
AutoV	2.24E-01 2.82E-01	1.93E-02 1.51E-02	**8.66E-02** 1.41E-03	7.96E-01 8.01E-02	1.65E-01 1.41E-01	7.27E-03 8.83E-04	1.34E-03 5.05E-03	2.84E-01 9.60E-02
RLDE-AFL	<u>6.36E-06</u> 2.93E-06	1.24E-03 2.42E-04	8.02E-01 4.60E-02	4.52E-01 1.37E-01	1.47E-05 1.93E-06	1.85E-04 5.82E-05	8.34E-03 1.15E-02	9.13E-02 4.07E-02
ConfigX	4.06E-01 3.49E-01	5.25E-02 4.27E-02	1.09E+00 2.99E-01	1.08E+00 6.61E-01	1.13E-01 2.34E-01	5.01E-02 1.09E-01	3.29E-01 4.91E-01	4.23E-01 3.79E-01
Q-Mamba	4.95E-01 1.58E-01	6.51E-02 3.64E-02	1.27E+00 1.95E-01	1.44E+00 3.13E-01	3.33E-02 6.24E-02	8.74E-03 2.97E-02	3.95E-01 2.70E-01	7.14E-01 3.26E-01
LLM-based design EAs								
LLaMEA-(1+1)	2.61E-05 1.62E-05	<u>4.51E-04</u> 1.96E-04	9.15E-01 1.70E-01	3.54E-01 1.53E-01	<u>1.04E-06</u> 2.92E-07	<u>8.33E-05</u> 5.28E-05	<u>3.36E-05</u> 4.40E-05	<u>8.05E-03</u> 7.03E-03
LLaMEA-(1,1)	7.88E-06 2.77E-06	1.95E-03 5.40E-04	6.26E-01 6.86E-02	4.24E-01 1.82E-01	1.75E-05 6.67E-06	2.28E-04 1.23E-04	1.06E-02 2.31E-02	7.00E-02 5.41E-02
FETK-(GLM-5.3)	**3.84E-06** 4.26E-06	**7.19E-05** 9.22E-05	2.47E-01 1.59E-01	**4.25E-02** 4.17E-03	**3.32E-07** 8.60E-09	**5.20E-05** 4.19E-06	**2.31E-05** 1.08E-07	**1.77E-03** 2.00E-05

4.6 Generalization to Different Landscape Characteristics

To evaluate whether the learned design knowledge could generalize beyond the problems used for knowledge learning, we tested *FETK* on MA-BBOB problems with mixed landscape characteristics. Specifically, *FETK* and all baseline methods were evaluated on eight MA-BBOB problems under the same experimental protocol. Notably, the knowledge used in this evaluation was learned from the 10-dimensional BBOB training problems and was reused without additional knowledge updates. As shown in Table 2, FETK achieved the lowest mean final fitness on 7 of the 8 problems and remained competitive on the remaining problems, demonstrating strong performance on landscapes that differed from the original BBOB training problems. These results demonstrate that the learned knowledge provides effective and transferable design guidance, enabling *FETK* to maintain competitive performance on problems with different landscape compositions. Additional results in 50 dimensions are reported in **Appendix E.5**.

5 Conclusion

This work investigated how problem-specific algorithm design experience could be transformed into reusable knowledge for black-box optimizer algorithm design. We proposed FETK, which collected design experience from EA architecture search, distilled it into reusable knowledge, and validated its usefulness across problems before reusing it for architecture generation on unseen problems. Experiments on BBOB and MA-BBOB showed that FETK achieved competitive optimization performance. The results further showed that the learned knowledge could be transferred to problems with different landscape characteristics and dimensions, including mixed-landscape MA-BBOB problems and higher-dimensional BBOB problems. Additional analyses showed that the design experience, landscape information, and cross-problem validation each contributed to the final performance. These findings suggested that architecture search could provide more than a final optimizer: it could also produce reusable design knowledge for future optimization problems. The current framework still incurred upfront cost during experience collection, motivating future work on more efficient search, adaptive experience collection, and broader validation across optimization tasks. The search budget analysis is in the **Appendix G**.

6 LLM USAGE STATEMENT

Writing Assistance. Large language models (LLMs) were employed exclusively to improve the quality of written English and for stylistic refinement, with attentive supervision by the authors. No scientific content, experimental design, data analysis, or any intellectual contribution was generated through LLMs, nor did such assistance influence any reported results.

Research Application. LLMs are integral components of the proposed FETK framework. They generate and refine evolutionary algorithm architectures, distill design experience into reusable natural-language knowledge, and consolidate candidate knowledge updates. During testing, an LLM generates an architecture conditioned on the learned knowledge and the landscape representation of the target problem. Candidate knowledge updates and generated architectures are evaluated through optimization experiments under the protocols described in this paper.

7 ETHICS STATEMENT

This work studies LLM-assisted evolutionary algorithm design for continuous black-box optimization. All experiments are conducted on synthetic BBOB and MA-BBOB benchmark problems and do not involve human participants or personal data.

8 REPRODUCIBILITY STATEMENT

The experimental settings are described in Section 4.1, with implementation details and prompt templates provided in Appendix D and F, respectively. The main optimization results are reported over 30 independent runs. The code is available anonymously at https://anonymous.4open.science/r/fetk-DE22.

REFERENCES

Leonardo C. T. Bezerra, Manuel López-Ibáñez, and Thomas Stützle. Automatic component-wise design of multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 20(3):403–417, 2016. doi: 10.1109/TEVC.2015.2474158.

Christian L Camacho-Villalón, Thomas Stützle, and Marco Dorigo. Designing new metaheuristics: manual versus automatic approaches. *Intelligent Computing*, 2:0048, 2023.

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

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, 2025a.

Hongshu Guo, Zeyuan Ma, Jiacheng Chen, Yining Ma, Zhiguang Cao, Xinglin Zhang, and Yue-Jiao Gong. Configx: Modular configuration for evolutionary algorithms via multitask reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26982–26990, 2025b.

Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. Research Report RR-6829, INRIA, 2009. URL https://inria.hal.science/inria-00362633.

John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. The MIT press, 1992.

Wenhu Li, Niki van Stein, Thomas Bäck, and Elena Raponi. Llamea-bo: A large language model evolutionary algorithm for automatically generating bayesian optimization algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 909–918, 2026.

Xijun Li, Jiexiang Yang, Jinghao Wang, Bo Peng, Jianguo Yao, and Haibing Guan. Strcmp: Integrating graph structural priors with language models for combinatorial optimization. *arXiv preprint arXiv:2506.11057*, 2025.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

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

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026b.

Zeyuan Ma, Zhiguang Cao, Zhou Jiang, Hongshu Guo, and Yue-Jiao Gong. Meta-black-box-optimization through offline q-function learning. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=7qCkf5OT3x.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. Reasoningbank: Scaling agent self-evolving with reasoning memory. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 94327–94354, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/980ea04d23d1f6908964eba2a74afe45-Paper-Conference.pdf.

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.

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. doi: 10.1109/TEVC.2024.3497793.

Ye Tian, Xingyi Zhang, Cheng He, Kay Chen Tan, and Yaochu Jin. Principled design of translation, scale, and rotation invariant variation operators for metaheuristics. *Chinese Journal of Electronics*, 32(1):111–129, 2023.

Ye Tian, Xuhong Qi, Shangshang Yang, Cheng He, Kay Chen Tan, Yaochu Jin, and Xingyi Zhang. A universal framework for automatically generating single-and multi-objective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 2025.

Niki van Stein, Anna V Kononova, Lars Kotthoff, and Thomas Bäck. Llamea-sage: Guiding automated algorithm design with structural feedback from explainable ai. *arXiv preprint arXiv:2601.21511*, 2026.

Diederick Vermetten, Furong Ye, Thomas Bäck, and Carola Doerr. MA-BBOB: A Problem Generator for Black-Box Optimization Using Affine Combinations and Shifts. *ACM Transactions on Evolutionary Learning and Optimization*, 5(1):1–19, March 2025. doi: 10.1145/3673908. URL https://hal.sorbonne-universite.fr/hal-04759417.

Pradnya A Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International conference on global trends in signal processing, information computing and communication (ICGTSPICC)*, pp. 261–265. IEEE, 2016.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Hui Wang, Xufeng Zhang, and Chaoxu Mu. Planning of heuristics: Strategic planning on large language models with monte carlo tree search for automating heuristic optimization. *arXiv preprint arXiv:2502.11422*, 2025.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

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

A NeuroEA Architecture Representation

FETK adopts the NeuroEA representation Tian et al. (2025) to encode evolutionary algorithms as executable architectures. Different from traditional evolutionary algorithms that manually define a fixed optimization pipeline, NeuroEA represents an EA as a directed computational graph, where evolutionary operations are organized as interconnected functional blocks.

A.1 Architecture Definition

A NeuroEA architecture is represented as

$$\mathcal{A} = (\mathbf{o}, \mathbf{E}, \mathbf{R}), \tag{5}$$

where $\mathbf{o}$ denotes the selected evolutionary blocks, $\mathbf{E}$ represents the directed connections among blocks, and $\mathbf{R}$ specifies the information-routing ratios associated with the connections. The graph topology determines how candidate solutions flow among evolutionary blocks, while the routing ratios control the proportion of information propagated along different connections.

During optimization, the initial population is propagated through the generated graph. Each block receives candidate solutions from its predecessors, performs a specific transformation, and passes the updated solutions to subsequent blocks. The outputs of terminal blocks are then used to construct the population for the next generation.

The architecture generated by FETK therefore consists of three components: (1) the selection and arrangement of evolutionary blocks, (2) the connections among blocks, and (3) the information-routing ratios between connected blocks. Continuous operator parameters are optimized after architecture generation and are not considered as part of the architecture.

A.2 Evolutionary Building Blocks

NeuroEA contains six types of evolutionary building blocks: Population, Crossover, Mutation, Exchange, Tournament, and Selection. These blocks provide different evolutionary operations and can be flexibly combined to construct diverse optimization workflows.

Population Block. The Population block stores the current candidate solutions and forwards them to subsequent blocks:

$$P_{\text{out}} = P_{\text{in}}. \tag{6}$$

It serves as the input and intermediate representation of solution populations during the evolutionary process.

Crossover Block. The Crossover block generates new candidate solutions by combining multiple parent solutions. For the d-th decision variable, an offspring solution is generated as

$$y_d = \sum_{i=1}^{m} w_i x_{i,d}, \tag{7}$$

where m denotes the number of selected parents, $x_{i,d}$ is the d-th variable of the i-th parent, and w_i is the corresponding recombination coefficient.

Mutation Block. The Mutation block introduces perturbations into candidate solutions. The d-th variable is updated as

$$y_d = x_d + \mathcal{N}(0, \sigma^2)(u_d - l_d), \tag{8}$$

where $\mathcal{N}(0, \sigma^2)$ is a Gaussian perturbation, $[l_d, u_d]$ is the search range, and σ controls the mutation strength.

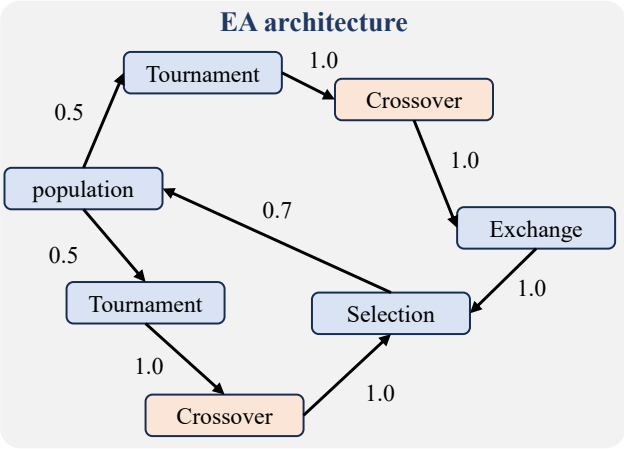


Figure 7: An example of a NeuroEA architecture. Nodes represent evolutionary blocks and edges indicate the information flow between blocks.

Exchange Block. The Exchange block creates new solutions by exchanging decision variables among multiple candidate solutions. For each dimension, the output variable is selected from different parent solutions according to the exchange configuration.

Tournament Block. The Tournament block performs parent selection based on fitness values. Given a tournament size k, candidate solutions are randomly sampled, and the best solution among them is selected as a parent.

Selection Block. The Selection block performs environmental selection to determine the surviving solutions for the next generation. It maintains population quality by retaining solutions according to their objective values.

A.3 Architecture Execution

Given a generated architecture $\mathcal{A}$, the evolutionary process repeatedly executes the computational graph for a predefined number of function evaluations. Different graph architectures lead to different interactions among evolutionary operations. Therefore, the NeuroEA representation enables FETK to generate diverse problem-dependent evolutionary algorithms by modifying the architecture architecture rather than relying on a fixed optimization pipeline.

To better illustrate the NeuroEA representation, we provide an example of a generated evolutionary algorithm architecture in Fig. 7. The architecture is represented as a directed graph, where each node corresponds to an evolutionary block, and each edge indicates the information flow between blocks.

B Construction of the Landscape Representation

For each problem τ_i, we uniformly sample N_s points $\{\mathbf{x}_{ij}\}_{j=1}^{N_s}$ from its bounded decision space and evaluate

$$y_{ij} = f_i(\mathbf{x}_{ij}). \tag{9}$$

Based on the sampled decision–fitness observations, we construct

$$\mathbf{z}_i = [\mathbf{q}_i,\, \mathbf{e}_i,\, \mathbf{p}_i]\,. \tag{10}$$

The three components are defined below.

B.1 Objective-Value Statistics $\mathbf{q}_i$

Let

$$y_i^{\min} = \min_{j=1,\dots,N_s} y_{ij}, \qquad y_i^{\max} = \max_{j=1,\dots,N_s} y_{ij}. \tag{11}$$

We denote the 25th, 50th, and 75th percentiles of the sampled objective values by

$$Q_{i,0.25}, \qquad Q_{i,0.50}, \qquad Q_{i,0.75}, \tag{12}$$

respectively. The interquartile range is

$$s_i^{\mathrm{IQR}} = \max\left(Q_{i,0.75} - Q_{i,0.25}, 10^{-12}\right). \tag{13}$$

The objective-value statistics are therefore represented as

$$\mathbf{q}_i = \left[y_i^{\min}, Q_{i,0.25}, Q_{i,0.50}, Q_{i,0.75}, y_i^{\max}, s_i^{\mathrm{IQR}}\right]. \tag{14}$$

B.2 Numerical Landscape Descriptors $\mathbf{e}_i$

We first normalize every sampled point to $[-1,1]^d$. For the m-th coordinate, the normalized value is

$$\hat{x}_{ijm} = 2\frac{x_{ijm} - \ell_{im}}{u_{im} - \ell_{im}} - 1, \tag{15}$$

where ℓ_{im} and u_{im} are the lower and upper bounds. If $u_{im} = \ell_{im}$, the denominator is replaced by one. Let $\hat{\mathbf{x}}_{ij}$ denote the resulting normalized point.

The numerical landscape descriptor is

$$\begin{aligned}\mathbf{e}_i = \big[&N_i^{\mathrm{peak}}, \bar{\delta}_i, \rho_{i,\mathrm{DR}}, I_i, R^2_{i,\mathrm{lin}}, R^2_{i,\mathrm{add}},\\ &R^2_{i,\mathrm{full}}, \Delta R^2_{i,\mathrm{int}}, \kappa_i, \rho_{i,\mathrm{FDC}}, B_i^{\mathrm{elite}}\big],\end{aligned} \tag{16}$$

.

Neighbourhood descriptors. The pairwise distance between two normalized points is

$$d_{ijk} = \left\|\hat{\mathbf{x}}_{ij} - \hat{\mathbf{x}}_{ik}\right\|_2. \tag{17}$$

The nearest-neighbour distance of sample j is

$$\delta_{ij} = \min_{k \neq j} d_{ijk}, \tag{18}$$

and the mean nearest-neighbour distance is

$$\bar{\delta}_i = \frac{1}{N_s}\sum_{j=1}^{N_s} \delta_{ij}. \tag{19}$$

Let $\mathcal{N}_5(j)$ denote the five nearest sampled neighbours of $\hat{\mathbf{x}}_{ij}$. We define

$$a_{ij} = \mathbb{I}\left[\nexists k \in \mathcal{N}_5(j) : y_{ik} < y_{ij}\right]. \tag{20}$$

The implementation reports

$$N_i^{\mathrm{peak}} = 1 + \sum_{j=1}^{N_s} a_{ij}. \tag{21}$$

Distance-ranked fitness variation. We construct a deterministic ordering by sorting the samples according to their nearest-neighbour distances:

$$\delta_{iw_1} \le \delta_{iw_2} \le \cdots \le \delta_{iw_{N_s}}. \tag{22}$$

The distance-ranked fitness correlation is defined as

$$\rho_{i,\mathrm{DR}} = \mathrm{corr}\left([y_{iw_1}, \dots, y_{iw_{N_s-1}}], [y_{iw_2}, \dots, y_{iw_{N_s}}]\right). \tag{23}$$

This statistic describes the association between consecutive fitness values along the nearest-distance ordering. In the implementation, it is stored under the field name `autocorrelation`.

For the same ordering, define the consecutive fitness differences and their signs as

$$\Delta y_{ik} = y_{iw_{k+1}} - y_{iw_k}, \qquad s_{ik} = \operatorname{sign}(\Delta y_{ik}), \qquad k = 1, \ldots, N_s - 1. \tag{24}$$

The information-content descriptor is

$$I_i = \frac{1}{2(N_s - 2)} \sum_{k=1}^{N_s - 2} |s_{i,k+1} - s_{ik}| \,. \tag{25}$$

It measures how frequently the direction of fitness change switches along the distance-ranked ordering.

Linear and quadratic fits. For a regression model with predictions $\hat{y}_{ij}$, we compute

$$R^2 = 1 - \frac{\sum_{j=1}^{N_s} (y_{ij} - \hat{y}_{ij})^2}{\sum_{j=1}^{N_s} (y_{ij} - \bar{y}_i)^2}, \tag{26}$$

where

$$\bar{y}_i = \frac{1}{N_s} \sum_{j=1}^{N_s} y_{ij}. \tag{27}$$

The resulting value is clipped to $[0, 1]$.

The linear model is

$$\hat{y}_{ij}^{\text{lin}} = \beta_0 + \sum_{m=1}^{d} \beta_m \hat{x}_{ijm}, \tag{28}$$

and its coefficient of determination is denoted by $R^2_{i,\text{lin}}$.

The additive quadratic model is

$$\hat{y}_{ij}^{\text{add}} = \beta_0 + \sum_{m=1}^{d} \beta_m \hat{x}_{ijm} + \sum_{m=1}^{d} \gamma_m \hat{x}_{ijm}^2, \tag{29}$$

with coefficient of determination $R^2_{i,\text{add}}$.

The full quadratic model additionally contains pairwise interaction terms:

$$\begin{aligned} \hat{y}_{ij}^{\text{full}} = \beta_0 + \sum_{m=1}^{d} \beta_m \hat{x}_{ijm} + \sum_{m=1}^{d} \gamma_m \hat{x}_{ijm}^2 \\ + \sum_{1 \le m < n \le d} \eta_{mn} \hat{x}_{ijm} \hat{x}_{ijn}. \end{aligned} \tag{30}$$

Its coefficient of determination is denoted by $R^2_{i,\text{full}}$.

Including the intercept, the full model contains

$$P_{\text{full}} = 1 + 2d + \frac{d(d-1)}{2} \tag{31}$$

parameters. The full model is considered available when

$$N_s > P_{\text{full}}. \tag{32}$$

The interaction gain is

$$\Delta R^2_{i,\text{int}} = \max\left(0,\, R^2_{i,\text{full}} - R^2_{i,\text{add}}\right). \tag{33}$$

Quadratic coefficient ratio. Using the squared-term coefficients from the additive quadratic model, we define

$$c_{im} = |\gamma_m|. \tag{34}$$

Let

$$\mathcal{C}_i = \left\{ c_{im} : c_{im} > 10^{-10} \right\}. \tag{35}$$

The quadratic coefficient ratio is

$$\kappa_i = \frac{\max \mathcal{C}_i}{\min \mathcal{C}_i}. \tag{36}$$

If $\mathcal{C}_i$ is empty or the least-squares solver fails, the implementation sets $\kappa_i = 10^{30}$.

Fitness–distance correlation. Let

$$j_i^\star = \arg\min_j y_{ij} \tag{37}$$

denote the best sampled point. Its distance to sample j is

$$d_{ij}^\star = \left\| \hat{\mathbf{x}}_{ij} - \hat{\mathbf{x}}_{ij_i^\star} \right\|_2. \tag{38}$$

The fitness–distance correlation is

$$\rho_{i,\mathrm{FDC}} = \mathrm{corr}\left([d_{i1}^\star, \ldots, d_{iN_s}^\star], [y_{i1}, \ldots, y_{iN_s}] \right). \tag{39}$$

Elite boundary proximity. For each normalized sample, we define its boundary proximity as

$$b_{ij} = \max_{m=1,\ldots,d} |\hat{x}_{ijm}|. \tag{40}$$

Let $\mathcal{E}_i$ denote the set of sampled points with the smallest objective values, where the number of elite points is

$$N_{\mathrm{elite}} = \max\left(1,\, \min\left(N_s,\, \max\left(3,\, \left\lfloor \frac{N_s}{10} \right\rfloor\right)\right)\right). \tag{41}$$

The mean elite boundary proximity is

$$B_i^{\mathrm{elite}} = \frac{1}{N_{\mathrm{elite}}} \sum_{j \in \mathcal{E}_i} b_{ij}. \tag{42}$$

A larger value indicates that the best sampled points tend to be closer to at least one boundary of the normalized decision space.

B.3 Semantic Landscape Profile $\mathbf{p}_i$

The numerical descriptors are mapped to a semantic profile

$$\mathbf{p}_i = [M_i,\, G_i,\, S_i,\, C_i,\, A_i,\, T_i], \tag{43}$$

where the components describe modality, ruggedness, separability, conditioning, interaction-model availability, and the fitness–distance relationship, respectively.

Modality. Define

$$r_i^{\mathrm{peak}} = \frac{N_i^{\mathrm{peak}}}{N_s}. \tag{44}$$

The modality profile is

$$M_i = \begin{cases} \texttt{low}, & r_i^{\mathrm{peak}} < 0.03, \\ \texttt{medium}, & 0.03 \le r_i^{\mathrm{peak}} < 0.12, \\ \texttt{high}, & r_i^{\mathrm{peak}} \ge 0.12. \end{cases} \tag{45}$$

Ruggedness. The ruggedness profile combines the additive quadratic fit with the distance-ranked fitness descriptors:

$$G_i = \begin{cases} \texttt{smooth}, & R_{i,\mathrm{add}}^2 \ge 0.85, \\ \texttt{highly_rugged}, & R_{i,\mathrm{add}}^2 < 0.85 \wedge \left(\rho_{i,\mathrm{DR}} < 0.25 \vee I_i \ge 0.65\right), \\ \texttt{moderately_rugged}, & \text{otherwise}. \end{cases} \tag{46}$$

Interaction-model availability. The interaction-model status is

$$A_i = \begin{cases} \texttt{available}, & N_s > P_{\text{full}}, \\ \texttt{insufficient_samples}, & N_s \le P_{\text{full}}. \end{cases} \tag{47}$$

Separability. The separability profile is

$$S_i = \begin{cases} \texttt{uncertain}, & A_i = \texttt{insufficient_samples}, \\ \texttt{strongly_separable}, & R^2_{i,\text{add}} \ge 0.85,\ \Delta R^2_{i,\text{int}} < 0.10,\ \kappa_i \le 10, \\ \texttt{weakly_separable}, & R^2_{i,\text{add}} \ge 0.55,\ \Delta R^2_{i,\text{int}} < 0.20, \\ \texttt{non_separable}, & \text{otherwise}. \end{cases} \tag{48}$$

Conditioning. The conditioning profile is

$$C_i = \begin{cases} \texttt{well_conditioned}, & R^2_{i,\text{add}} \ge 0.85 \wedge \kappa_i \le 10, \\ \texttt{severely_ill_conditioned}, & \kappa_i > 100, \\ \texttt{ill_conditioned}, & \text{otherwise}. \end{cases} \tag{49}$$

Fitness–distance relationship. The fitness–distance profile is

$$T_i = \begin{cases} \texttt{strong_global_trend}, & |\rho_{i,\text{FDC}}| \ge 0.65, \\ \texttt{weak_global_trend}, & 0.25 \le |\rho_{i,\text{FDC}}| < 0.65, \\ \texttt{no_global_trend}, & |\rho_{i,\text{FDC}}| < 0.25. \end{cases} \tag{50}$$

C Example of Learned Knowledge

This section presents an example of the learned knowledge obtained by FETK. We compare the initial knowledge $K^{(0)}$ with the final learned knowledge $K^{(T)}$ from the same optimization run to illustrate the form and content of the acquired design rules.

C.1 Presentation Protocol

We use $\mathcal{K}^{(0)}$ to denote the initial knowledge supplied to the agent and $\mathcal{K}^{(T)}$ to denote the knowledge after T optimization iterations. Both versions are collected from the same experimental run and are evaluated under the same task configuration. This notation allows us to separate the effect of knowledge evolution from differences in model, environment, or evaluation protocol.

For readability, we normalize indentation and remove duplicated blank lines, but preserve the original semantic content and the ordering of knowledge items. The comparison table reports semantic changes rather than character-level edits.

C.2 Initial Knowledge

Initial Knowledge $\mathcal{K}^{(0)}$

- NeuroEA architecture Design knowledge for Anonymous Black-Box Objectives
  Use the observable probe evidence to design a problem-adaptive NeuroEA operator composition and topology.
- Probe-Driven Design Strategy
  - Infer algorithm-relevant landscape characteristics and tentative problem-class hypotheses from the observable probe.
  - Treat separability, conditioning, modality, smoothness, ruggedness, and variable interaction as uncertain inferences rather than known facts.

C.3 Final Learned Knowledge

Learned Knowledge $\mathcal{K}^{(T)}$

- NeuroEA architecture Design knowledge for Anonymous Black-Box Objectives

  Use the observable probe evidence to design a problem-adaptive NeuroEA operator composition and topology.

- Probe-Driven Design Strategy

  - Infer algorithm-relevant landscape characteristics and tentative problem-class hypotheses from the observable probe.

  - Treat separability, conditioning, modality, smoothness, ruggedness, and variable interaction as uncertain inferences rather than known facts.

- Parallel Dual-Crossover Branches for Interactive Weak-Trend Landscapes

  If the probe shows an available interaction model with high quadratic_full_r2 $>= 0.75$ but only weak-to-moderate fitness-distance correlation (fdc roughly 0.3-0.55) and elite_mean_boundary_proximity $>= 0.85$, prefer a topology with two Crossover operators using different n_sets (e.g., 5 and 2-3), each fed from a separate Tournament branch and re-merged before the final Selection, because interacting variables without a strong global trend reward recombining building blocks along multiple parallel paths, and differing n_sets sustain recombinant diversity while independent tournament feeds keep both branches productive. Keep each Tournament.n_parents at or above Selection.n_solutions and avoid stacking small fractional ratios in series, so parallel branching does not collapse total offspring flow below the generation budget.

- Dual-Tournament Parallel-Branch Design for Quadratic and Rugged Landscapes

  If observable probe evidence indicates moderate-to-high quadratic_additive_r2 (> 0.6) or highly rugged/ill-conditioned landscapes (high number_of_peaks, low linear_r2), prefer a topology with two or three parallel Tournament nodes drawn from the same Population, each with modestly diversified upper_k values (e.g., 3–10), feeding distinct Crossover and Mutation branches with fractional routing into a shared Selection node before returning to Population. On smooth, well-conditioned landscapes where quadratic_additive_r2 approaches 1.0, favor crossover-heavy branches (Crossover n_sets=3–4) with low tournament pressure (upper_k $=< 5$) to exploit strong additive architecture via recombination. On highly rugged landscapes (peaks_per_sample > 0.1, fdc < 0.3), include an Exchange node chained after one Crossover branch to introduce discrete multi-parent recombination for exploration, and use moderate tournament pressure (upper_k 10–20) on at least one branch to maintain selection diversity. The parallel branching provides both exploitation (crossover paths) and exploration (mutation/exchange paths) simultaneously, which is essential when landscape modality or conditioning makes serial sequential operators insufficient.

- Parallel Tournament with Fractional Exploration-Exploitation Routing for Rugged Ill-Conditioned Landscapes

  If observable probe evidence indicates highly rugged landscapes (peaks_per_sample > 0.1, autocorrelation < 0.15) with low additive architecture (quadratic_additive_r2 < 0.5) and ill-conditioning (quadratic_curvature_ratio > 5), prefer a topology with two parallel Tournament nodes from Population: route one Tournament's output fractionally (0.7) to Crossover (n_sets=3–4) for exploitation and the remainder (0.3) to Exchange (n_parents=3) for exploration; route the other Tournament's output fractionally between Mutation (n_sets=4) and a second Crossover (n_sets=2), all converging into Selection before returning to Population. Use moderate tournament pressure (upper_k 12–18) on the exploitation-focused branch and moderate pressure (upper_k 10–14) on the exploration branch. This design balances exploitation of weak additive signals via Crossover with exploration via multi-parent Exchange and mutation, which is essential when the landscape is too rugged for pure recombination to converge but still has partial architecture to exploit.

Table 3: Training, validation, and test split of the 24 BBOB functions on seed 0.

Subset	**Size**	**Functions**
Training	12	$\{F_1, F_3, F_5, F_7, F_8, F_{11}, F_{12}, F_{15}, F_{18}, F_{21}, F_{22}, F_{23}\}$
Validation	6	$\{F_4, F_6, F_{10}, F_{19}, F_{20}, F_{24}\}$
Test	6	$\{F_2, F_9, F_{13}, F_{14}, F_{16}, F_{17}\}$

Table 4: Composition weights of the eight MA-BBOB instances used in the experiments. Only BBOB component functions with nonzero weights are reported; all omitted component functions have weight zero. The weights of each MA-BBOB instance sum to one, up to floating-point precision.

Instance	Nonzero BBOB component weights
MA_1	$w_{10} = 0.798115$, $w_{14} = 0.201885$
MA_2	$w_{11} = 0.695093$, $w_{21} = 0.304907$
MA_3	$w_6 = 0.043860$, $w_8 = 0.400221$, $w_{16} = 0.051291$, $w_{20} = 0.504628$
MA_4	$w_3 = 0.205380$, $w_4 = 0.241028$, $w_9 = 0.168784$, $w_{12} = 0.109651$, $w_{14} = 0.275158$
MA_5	$w_3 = 0.599402$, $w_{11} = 0.134589$, $w_{13} = 0.095230$, $w_{16} = 0.170780$
MA_6	$w_4 = 0.229098$, $w_7 = 0.189270$, $w_{13} = 0.153170$, $w_{14} = 0.192552$, $w_{18} = 0.131293$, $w_{21} = 0.104618$
MA_7	$w_6 = 0.340327$, $w_9 = 0.098094$, $w_{11} = 0.143624$, $w_{13} = 0.311827$, $w_{16} = 0.106127$
MA_8	$w_2 = 0.179044$, $w_7 = 0.230021$, $w_{10} = 0.084513$, $w_{11} = 0.253514$, $w_{20} = 0.018155$, $w_{21} = 0.234752$

D Implementation and Experimental Details

This section provides additional implementation details for knowledge learning, knowledge validation, and final evaluation.

Dataset Split. The 24 BBOB functions were partitioned into mutually exclusive training, validation, and test sets using a ratio of 2:1:1. Consequently, 12 functions were used for training, 6 functions for validation, and the remaining 6 functions for final testing. The resulting partition is reported in Table 3. The partition was generated by first shuffling the complete function pool with a fixed random seed of $s_{\mathrm{split}} = 0$ and then assigning the shuffled functions to the three subsets according to the 2:1:1 ratio. Fixing the split seed makes the partition deterministic and fully reproducible across experimental runs. The validation set was used for model selection and update acceptance, whereas the test set remained unseen during training and was used only for final performance evaluation.

Composition weights of the MA-BBOB instances. Table 4 reports the composition weights of the eight MA-BBOB instances used in our experiments. Let f_j denote the j-th BBOB component function and $w_{i,j}$ its weight in instance MA_i. Each instance is constructed as a weighted combination of a subset of the 24 BBOB component functions. For clarity, the table reports only components with nonzero weights; all unlisted weights are zero. The displayed values are rounded to six decimal places, whereas the full-precision weights were used during evaluation. For each instance, the weights sum to one up to floating-point precision. These instance definitions were fixed across all compared methods and experimental runs.

Landscape characterization. For each problem, $N_s = 10D$ solutions were sampled and evaluated for landscape characterization. The same procedure was used during training, validation, and testing. An initial population of $N = 100$ solutions was selected uniformly at random without replacement from these evaluated samples, and their objective values were reused without additional evaluations. Thus, all 100 samples were used in 10 dimensions, whereas 100 of the 500 samples were randomly selected in 50 dimensions. All N_s probing evaluations, including those of samples not selected for initialization, were counted toward the total function evaluation budget.

In the 50-dimensional setting, the additive quadratic model contains $p_{\mathrm{add}} = 1 + 2D = 101$ parameters; the full quadratic model contains$p_{\mathrm{full}} = 1 + 2D + \frac{D(D-1)}{2} = 1326$ parameters and remains underdetermined because $N_s = 500 < p_{\mathrm{full}}$. Therefore, we do not draw conclusions about pairwise quadratic interaction or separability from R^2_{full} or $\Delta R^2_{\mathrm{int}}$ in the 50-dimensional experiments. The corresponding landscape is treated as uncertain.

Knowledge learning. Knowledge learning was conducted for $E = 2$ epochs with a batch size of $b = 4$. For each training problem, the architecture-search process was performed for $T_{\max} = 20$

refinement rounds. During design-experience generation, only the EA architecture was optimized; continuous operator parameters were not tuned at this stage. This separation allows the collected experience to primarily reflect architectural modifications and their corresponding performance changes, rather than effects caused by continuous parameter optimization.

Knowledge validation. Each candidate knowledge update was evaluated on the six held-out validation functions using paired comparisons based on the log optimality gap. A candidate update was accepted only if it improved at least $W_{\min} = 3$ validation problems, and satisfied the regression constraint with $\tau_{\text{reg}} = 1$. The numerical stabilizer used in the log-gap calculation was set to $\epsilon = 10^{-300}$. All paired comparisons used the same observable problem information, random seed, and function evaluation budget for the current and candidate knowledge states.

Final evaluation. All main experiments were conducted in $D = 10$ dimensions with a solution population size of $N = 100$ and a function evaluation budget of $\text{MaxFEs} = 1000D$ per optimization run. For additional high-dimensional evaluations, we considered $D = 50$ with the same population size and a budget of $\text{MaxFEs} = 1000D$.

Continuous parameter tuning. After generating an architecture for each test problem, we optimize its continuous operator parameters using a genetic algorithm with a population size of 50 and a budget of 5,000 parameter-configuration evaluations. Each parameter configuration is evaluated by running the generated optimizer on the target problem with a budget of $1000D$ objective-function evaluations. Thus, evaluating 5,000 parameter configurations requires up to $5000 \times 1000D$ objective-function evaluations. This stage adjusts only continuous parameters, without modifying the architecture or invoking the LLM.

The tuning cost is accounted for separately from the $1000D$ objective-function evaluations allocated to each final optimization run. The final-run budget includes the $10D$ landscape-probing evaluations. Accordingly, the reported results compare the resulting optimizers under the same final-run budget after their respective design and tuning procedures.

E Additional Results

E.1 Additional Results on 50-dimensional BBOB

To further evaluate the scalability of FETK, we extended the experiments to 50-dimensional BBOB problems while keeping the same evaluation protocol as the main experiments. Table 5 reported the optimization performance of FETK and the baseline methods in higher dimensions. The results showed that FETK remained competitive on the higher-dimensional problem. Compared with representative baseline methods, FETK achieved superior or comparable performance on most high-dimensional problems, demonstrating the effectiveness of the 50-dimensional setting.

E.2 Additional Results under Different Test Splits

To examine whether the performance of FETK depended on a particular train-test partition, we further evaluated the method using additional test splits. For split seed 1, we relearned the design knowledge from the initial knowledge using only the training and validation problems in the corresponding split. The resulting knowledge was then frozen and evaluated on that split's test problems. Table 6 showed the results of the problem partitioning under seed 1. Figure 8 presented the ranking results of FETK and baseline methods under an alternative split. The results showed that FETK consistently achieved competitive rankings across different test partitions, indicating that the performance improvement was not sensitive to a specific data split.

E.3 Architecture Generation Stability

To evaluate the stability of architecture generation under the same knowledge, we independently invoked the LLM five times on a BBOB 10-dimensional problem using the same knowledge, problem description, landscape information, and generation prompt. The generated architectures were evaluated under the same downstream optimization budget and evaluation protocol. Table 7 reported

Table 5: Optimization results on six 50-dimensional BBOB test functions. Each entry reports the mean on the first line and the sample standard deviation in a smaller font on the second line. Lower values are better; the best and second-best means are shown in bold and underlined, respectively.

Method	F2	F9	F13	F14	F16	F17
Manually designed EAs						
GA	7.36E+00 2.96E+00	8.81E+01 5.92E+01	3.61E+01 1.57E+01	2.47E-02 3.31E-03	2.14E+01 4.08E+00	3.51E+00 5.60E-01
NeuroEA	1.71E+02 1.64E+02	1.30E+02 6.09E+01	1.29E+02 2.55E+01	2.81E-01 1.43E-01	1.50E+02 6.23E+01	3.44E+01 3.04E+00
Learning-based design EAs						
AutoV	5.98E-06 5.94E-06	7.21E+01 3.42E+01	5.37E+01 1.18E+01	9.53E-03 1.21E-03	2.40E+01 6.62E+00	1.83E+00 3.46E-01
RLDE-AFL	9.49E-07 4.73E-08	6.18E+01 2.89E+01	6.99E+00 5.32E+00	7.41E-03 1.93E-04	3.18E+01 5.17E+00	1.51E+00 1.59E-01
ConfigX	9.19E+05 3.46E+05	3.49E+04 7.46E+04	1.30E+03 5.18E+02	1.73E+01 1.03E+01	3.84E+01 6.08E+00	6.83E+00 3.98E+00
Q-Mamba	7.16E+05 1.91E+05	1.92E+03 1.17E+03	7.82E+02 1.30E+02	8.08E+00 2.38E+00	3.09E+01 3.21E+00	8.16E+00 1.32E+00
LLM-based design EAs						
LLaMEA-(1+1)	5.44E+02 6.62E+00	9.67E+01 8.05E-01	6.10E+00 7.72E-01	**5.36E-04** 1.57E-06	5.90E+00 1.01E-01	1.04E-02 1.59E-04
LLaMEA-(1,1)	1.19E-06 5.18E-07	4.79E+01 7.09E-01	1.38E+01 1.17E+00	5.01E-02 1.74E-03	2.06E+01 2.30E-01	2.61E-01 3.69E-02
FETK-(GLM-5.3)	**8.77E-08** 6.47E-08	**1.66E+01** 1.06E+00	**2.56E+00** 1.01E+00	4.94E-03 3.29E-04	**1.22E+00** 1.29E-01	**8.64E-03** 1.63E-04

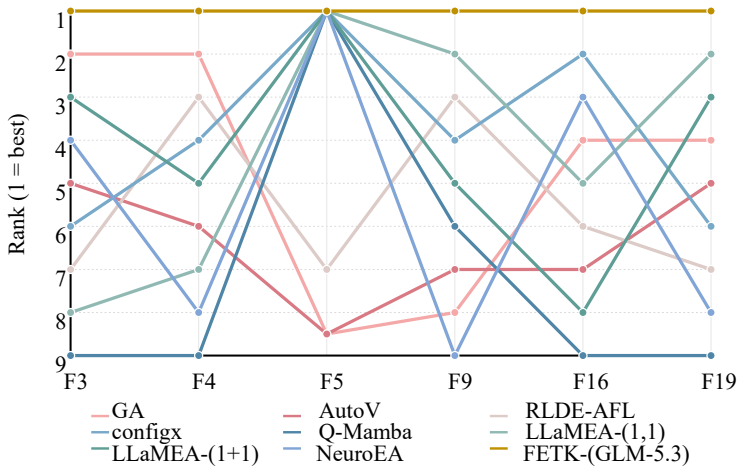


Figure 8: Optimization rankings of FETK and baseline methods under an additional test split. Lower ranks indicate better performance.

the mean and sample standard deviation over the five independent generations. A smaller standard deviation indicated more stable architecture generation under the same knowledge. In particular, all five generations achieved a zero optimality gap on F2. The results showed that the generated architectures exhibited good overall stability across the tested functions.

To improve the comparability of results with substantially different magnitudes, the objective values were normalized independently for each benchmark problem using a reversed log-scale min–max transformation. Specifically, for the result $x_{i,j}$ obtained by method i on problem j, we first computed

$$z_{i,j} = \log_{10}(x_{i,j} + \varepsilon),$$

where $\varepsilon = 10^{-16}$ was introduced to handle zero-valued results. The normalized score was then calculated as

$$s_{i,j} = \frac{\max_k z_{k,j} - z_{i,j}}{\max_k z_{k,j} - \min_k z_{k,j}}.$$

Table 6: Training, validation, and test split of the 24 BBOB functions with split seed 1.

Subset	**Size**	**Functions**
Training	12	$\{F_{21}, F_{12}, F_{24}, F_{18}, F_{22}, F_{14}, F_6, F_{11}, F_{10}, F_7, F_1, F_8\}$
Validation	6	$\{F_2, F_{20}, F_{17}, F_{13}, F_{23}, F_{15}\}$
Test	6	$\{F_{16}, F_4, F_9, F_3, F_{19}, F_5\}$

Table 7: Performance stability of five independent LLM architecture generations under the same knowledge. Values are reported as mean $\pm$ sample standard deviation. Lower values are better.

Problem	Mean $\pm$ STD
F2	$0.00\text{E}{+00} \pm 0.00\text{E}{+00}$
F9	$1.49\text{E}{-01} \pm 9.06\text{E}{-02}$
F13	$1.69\text{E}{-02} \pm 1.65\text{E}{-02}$
F14	$3.33\text{E}{-04} \pm 1.40\text{E}{-04}$
F16	$1.58\text{E}{-01} \pm 1.31\text{E}{-01}$
F17	$1.34\text{E}{-02} \pm 6.64\text{E}{-03}$

The normalization was performed independently for each problem over all compared methods. Consequently, the best result on each axis receives a score of 1, while the worst result receives a score of 0.

E.4 Optimization Stability with Independently Learned Knowledge

To investigate the stability of knowledge learning, we independently train FETK three times and evaluate the resulting knowledge instances on the test set on BBOB 10-dimensional problems. As shown in Fig. 9, different knowledge instances consistently produce effective optimization trajectories. Although the convergence speed varies across independent runs, all runs maintain similar decreasing trends and achieve comparable final fitness values. These results suggest that FETK can reliably learn useful design knowledge without relying on a specific LLM generation trajectory.

E.5 Additional Results on 50-dimensional MA-BBOB

To further evaluate the generalization ability of FETK to higher-dimensional optimization problems, we extend the MA-BBOB experiments from 10 dimensions to 50 dimensions. The same experimental protocol and evaluation budget setting are adopted as in the main experiments. Table 8 reports the results on eight MA-BBOB test instances in 50 dimensions. FETK achieves the best performance on most test instances and remains competitive compared with manually designed, learning-based, and LLM-based EA design methods. These results indicate that the learned design knowledge remains effective when the problem dimensionality increases, demonstrating the scalability of FETK to 50-dimensional optimization problems.

F Prompt Templates

This section provides the prompt templates used in FETK to facilitate reproducibility.

F.1 Prompt Overview

This appendix presents the prompt templates used in our framework. Instead of relying on a single monolithic prompt, we decompose the prompting process according to the major stages of the optimization pipeline. This decomposition makes the responsibility of each prompt explicit and allows the prompts used at different stages.

Specifically, the *system prompt* defines the role of the language model, the overall optimization objective, and the general behavioral and output constraints shared across all stages. The *architecture*

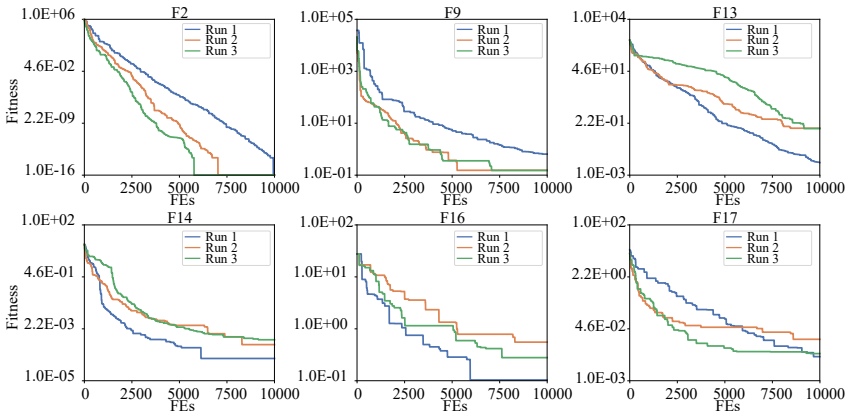


Figure 9: Convergence curves of FETK using architectures generated from three independently learned knowledge instances on six test functions. Each curve represents the optimization performance of an architecture generated with one learned knowledge instance.

Table 8: Optimization results on eight MA-BBOB test instances in 50 dimensions. Lower values are better. Each entry reports the mean on the first line and the sample standard deviation in a smaller font on the second line. The best mean value is highlighted in bold, while the second-best mean value is underlined.

Method	**MA_1**	**MA_2**	**MA_3**	**MA_4**	**MA_5**	**MA_6**	**MA_7**	**MA_8**
Manually designed EAs								
GA	5.64E-01 1.21E-01	2.11E+00 3.01E-01	<u>3.07E-03</u> 4.84E-03	2.70E-02 1.09E-02	8.81E+00 1.57E+00	1.99E-02 9.39E-03	8.10E+00 1.45E+00	5.32E+00 8.29E-01
NeuroEA	3.39E+00 3.62E+00	3.85E+00 5.96E-01	3.12E-01 1.23E-01	1.29E+00 5.51E-01	<u>2.56E+00</u> 4.98E-01	1.02E+01 1.09E+00	2.75E+00 5.47E-01	4.61E+00 6.53E-01
Learning-based design EAs								
AutoV	4.17E-01 7.88E-02	2.30E+00 2.48E-01	6.09E-03 7.90E-03	<u>1.66E-02</u> 1.82E-02	8.18E+00 1.22E+00	1.35E-02 6.98E-03	7.85E+00 1.28E+00	5.67E+00 9.14E-01
RLDE-AFL	<u>2.14E-01</u> 1.98E-02	1.64E+00 1.07E-01	2.95E-02 3.24E-03	8.37E-02 4.21E-02	5.94E+00 5.00E-01	<u>7.48E-03</u> 9.94E-03	4.11E+00 1.58E-01	3.58E+00 1.98E-01
ConfigX	8.85E-01 5.85E-01	1.84E+00 4.16E-01	1.08E-01 2.35E-01	5.97E-01 7.44E-01	8.16E+00 1.72E+00	7.39E-01 1.48E+00	7.24E+00 1.99E+00	5.07E+00 1.45E+00
Q-Mamba	1.78E+00 2.59E-01	2.72E+00 2.50E-01	3.78E-02 6.04E-02	3.00E-01 3.67E-01	1.00E+01 8.28E-01	3.54E-01 7.73E-01	8.89E+00 9.47E-01	7.33E+00 7.41E-01
LLM-based design EAs								
LLaMEA-(1+1)	2.49E-01 5.76E-02	1.03E+00 1.71E-01	2.79E-02 8.24E-03	1.01E-01 9.47E-02	3.51E+00 4.78E-01	1.28E-02 6.73E-03	<u>1.12E+00</u> 7.69E-01	<u>1.46E+00</u> 4.10E-01
LLaMEA-(1,1)	4.13E-01 8.27E-02	<u>8.89E-01</u> 1.62E-01	4.00E-02 1.55E-02	2.49E-01 2.18E-01	6.26E+00 6.23E-01	1.86E-02 2.04E-02	5.18E+00 1.05E+00	2.13E+00 4.59E-01
FETK-(GLM-5.3)	**1.23E-01** 4.26E-02	**3.12E-01** 1.88E-01	**2.45E-04** 9.31E-05	**2.46E-04** 4.43E-04	**1.52E+00** 8.08E-01	**3.14E-06** 5.85E-06	**1.53E-01** 1.52E-01	**1.06E-02** 9.64E-03

generation prompt instructs the model to construct or revise a candidate architecture based on the task specification, the current search context, and the available knowledge. After a candidate has been evaluated, the *knowledge reflection prompt* guides the model to analyze the evaluation results, identify the causes of success or failure, and extract knowledge for subsequent iterations. Finally, the *patch aggregation and knowledge update prompt* consolidates the knowledge patches produced during reflection, resolves redundant or conflicting information, and updates the knowledge used in future architecture-generation steps.

The following subsections provide the complete prompt templates for these stages. Fixed instructions are shown verbatim, whereas task-dependent fields are represented by placeholders enclosed in angle brackets, such as `<TASK_DESCRIPTION>` and `<EVALUATION_RESULTS>`. Unless oth-

erwise specified, the same prompt templates are used throughout all experiments, with only these runtime fields being instantiated for different tasks and optimization iterations.

F.2 System Prompt

The system prompt specifies the persistent instructions shared by all stages of our framework. It defines the role of the language model, states the overall optimization objective, and establishes general constraints on reasoning, tool usage, and output formatting. These instructions remain unchanged across optimization iterations.

System Prompt

Role
You are an expert evolutionary algorithm architect for anonymous continuous black-box minimization.

System-level design knowledge: [CURRENT KNOWLEDGE]

Task: Generate an Algorithm by Composing Built-in Operators
Design one NeuroEA computation graph using the supplied observable landscape context. This is architecture-only search: do not write Python code or create a new operator.

Architecture Generation Constraints

- JSON must contain `nodes` and `edges`.
- Supported kwargs for available standard operators:
  - `Population`: no parameters; must exist and usually serves as starting node 0.
  - `Selection`: optional parameter `n_solutions`, default 100; usually serves as the tail aggregation node.
  - `Crossover`: optional parameters `n_parents` and `n_sets`; use `n_parents`=2 and `n_sets`$\leq$10, preferably `n_sets` ≤ 5. Crossover dimension = $(\texttt{n_parents} - 1) \times \texttt{n_sets} \times 3$ and must be ≤ 30.
  - `Mutation`: optional parameter `n_sets`; it must be ≤ 15, preferably ≤ 5. Mutation dimension = `n_sets` $\times$ 2 and must be ≤ 30.
  - `Exchange`: optional parameter `n_parents`.
  - `Tournament`: optional parameters `n_parents` and `upper_k`.
- HPO dimensionality is not hard-capped. High-dimensional HPO is allowed, but it is slower and less sample-efficient; expose only meaningful scalar controls and avoid D-by-D HPO parameter arrays.
- Except for the generation-loop edge back to node 0, no cycles or self-loops are allowed.
- The data flow must eventually narrow and flow back to node 0.
- Node 0 (`Population`) must have at least one explicit incoming edge from a terminal or aggregation node, for example `Selection` $\rightarrow$ `Population`.

Example JSON Output {

```
"nodes": [
   {"id": 0, "type": "Population"},
   {"id": 1, "type": "Tournament",
    "args": {"n_parents": 200, "upper_k": 10}},
   {"id": 2, "type": "Tournament",
    "args": {"n_parents": 200, "upper_k": 10}},
   {"id": 3, "type": "Mutation",
    "args": {"n_sets": 5}},
   {"id": 4, "type": "Exchange",
    "args": {"n_parents": 3}},
   {"id": 5, "type": "Selection",
    "args": {"n_solutions": 100}}
],
"edges": [
   {"from": 0, "to": 1, "ratio": 1.0},
   {"from": 0, "to": 2, "ratio": 1.0},
   {"from": 1, "to": 4, "ratio": 0.5},
   {"from": 2, "to": 4, "ratio": 0.5},
   {"from": 4, "to": 3, "ratio": 1.0},
   {"from": 3, "to": 5, "ratio": 1.0},
   {"from": 5, "to": 0, "ratio": 1.0}]
```

}

Output Format
Return exactly one complete valid JSON object with keys `nodes` and `edges`, without Markdown fences or surrounding prose.
Never output a Python code block.

F.3 Architecture Generation Prompt

The architecture generation prompt is responsible for constructing the NeuroEA computation graph used in the next optimization iteration. It takes the current search state, the evaluated architecture, the historical-best architecture, the observed performance feedback, and the available landscape information as inputs. To support both local refinement and structural exploration, the prompt operates in two modes controlled by the binary indicator `q`. When `q=0`, the model refines the current architecture while preserving its effective components. When `q=1`, the model performs a broader regeneration based on the observed optimization bottleneck and is allowed to substantially reconstruct the architecture.

Complete Prompt for Architecture Generation

Role
You are an expert evolutionary algorithm architect for anonymous continuous black-box minimization. Design one NeuroEA computation graph by composing the available built-in operators.

Task
Generate one complete NeuroEA computation graph for the current optimization problem. This is architecture-only search. Do not write Python code and do not create new operators.
The current optimization state is described by the following information:

- Current architecture: `[CURRENT JSON ARCHITECTURE]`
- Historical-best architecture: `[HISTORICAL-BEST JSON ARCHITECTURE]`
- Current fitness: `[CURRENT FITNESS]`
- Historical-best fitness: `[HISTORICAL-BEST FITNESS]`
- Landscape context: `[LANDSCAPE CONTEXT]`
- Restart indicator: `q=[RESTART INDICATOR]`

Architecture Update Mode
The restart indicator `q` determines how the architecture should be generated.
When `q=0`: Architecture Refinement
The current architecture is refined based on its evaluation feedback. Identify ineffective operators, redundant branches, or unsuitable parameter settings, and make targeted modifications. Preserve effective components and useful search behavior whenever possible. Do not replace the entire architecture unless the evaluation feedback provides a clear reason for doing so.
When `q=1`: Architecture Regeneration
The current search is considered stagnant or trapped in a local-optimum regime. Analyze the weaknesses of both the current architecture and the historical-best architecture. Perform a substantial reconstruction when necessary by exploring different operator combinations, branch architectures, and parameter settings. The regenerated architecture should address the identified optimization bottleneck rather than merely apply a small variation to the current graph.
In both modes, use the supplied landscape context to determine the required search behavior. The architecture must remain valid under all operator, dimensionality, and topology constraints described below.

Available Operators
The architecture may use the following built-in operators:

- Population: no parameters; must exist and usually serves as the starting node.
- Selection: optional parameter `n_solutions`, with default value 100; usually serves as the terminal aggregation node.
- Crossover: optional parameters `n_parents` and `n_sets`. Use `n_parents=2` and keep `n_sets` no larger than 10, preferably no larger than 5. Its parameter dimension must be no larger than 30.
- Mutation: optional parameter `n_sets`. Keep `n_sets` no larger than 15, preferably no larger than 5. Its parameter dimension must be no larger than 30.

- Exchange: optional parameter `n_parents`.
- Tournament: optional parameters `n_parents` and `upper_k`.

Only meaningful scalar controls should be exposed. Avoid unnecessary high-dimensional or dimension-by-dimension hyperparameter arrays.

Topology Constraints

1. The output must contain exactly two top-level keys: `nodes` and `edges`.
2. Node 0 must be a Population node.
3. Node 0 must have at least one explicit incoming edge from a terminal or aggregation node, such as Selection → Population.
4. Except for the generation-loop edge returning to node 0, the graph must not contain cycles or self-loops.
5. Every node type and keyword argument must be supported by the available-operator list.

Output Format
Return exactly one complete and valid JSON object with the top-level keys `nodes` and `edges`. The JSON object must describe one executable NeuroEA computation graph.
Do not include Markdown fences, explanations, comments, or any surrounding prose. Never output a Python code block.

F.4 KNOWLEDGE REFLECTION PROMPT

The knowledge reflection prompt uses successful and failed trajectories to extract architecture-design knowledge. It identifies effective patterns from improved architectures and unreliable choices from non-improving transitions, and updates the current knowledge.

Complete Prompt for Success- and Failure-Guided Knowledge Reflection

Role
You improve a reusable knowledge base that must generate one strong NeuroEA architecture for an anonymous continuous black-box objective. The final architecture must be generated directly from observable probe evidence.

Input
You are given a reflection minibatch containing evidence from several anonymous optimization tasks. Each task contains:

- an observable probe summary;
- the initial-best architecture and its evaluation results;
- the overall-best architecture and its evaluation results;
- a direct aggregate comparison between the two architectures; and
- for a failure minibatch, at most one non-improving transition from the final generation.

The complete input is provided as follows:
Current knowledge
`[CURRENT knowledge]`
Edit Budget
Produce at most `[EDIT BUDGET]` independent edits.
Previous Steps
`[PREVIOUS HELD-OUT REJECTION CONTEXT, IF AVAILABLE]`
Reflection Minibatch Type
`[SUCCESS OR FAILURE]`
Lineage Reflection Minibatch
`[ANONYMOUS LINEAGE EVIDENCE IN JSON FORMAT]`

NeuroEA Semantics
Use the following operator semantics when interpreting the supplied architectures:

- Population is the persistent population state and must be node 0.
- Tournament emits exactly `n_parents` selected solution objects. The parameter `n_parents` controls output cardinality, not selection pressure. The parameter `upper_k` controls the upper bound on tournament size and selection pressure.

- Crossover performs recombination and emits approximately half as many offspring as its input solutions when `n_parents=2`.
- Mutation performs mutation according to its `n_sets` configuration.
- Exchange performs discrete recombination among multiple parents and reduces the output cardinality according to `n_parents`.
- Selection aggregates and narrows candidate solutions before returning them to the population.
- Each edge specifies the fraction of one node's output routed to another node. Branch count, routing ratio, branch fusion, and the return edge to Population are part of the algorithm design.

Graph legality is not performance evidence. Separate the constraints required for an executable graph from claims that a particular valid topology or parameter setting improves performance.
Small tournament output counts, serial recombination operators, and fractional routing may jointly produce too few offspring. Treat a generation-guard failure as evidence against the complete combination of topology, routing, and cardinality settings. Do not translate such a failure into a recommendation for a nearby small value of `Tournament.n_parents`.

Reflection Policy
First determine whether the supplied minibatch contains successful or failed optimization evidence.
Success Reflection
For a success minibatch, compare the `initial_best` and `overall_best` architectures within each anonymous task. Identify the complete structural difference, including:

- operators added, removed, or reordered;
- changes to operator arguments;
- changes to edges or routing ratios;
- branches that were split, fused, serialized, or removed; and
- differences in performance across paired evaluation seeds.

Use the overall-best architecture as positive evidence only for its corresponding observable probe signature. Extract reusable design principles from structural changes that are consistently associated with improved performance.
Failure Reflection
For a failure minibatch, use the supplied final-generation `failure_transitions` as the primary negative evidence. Compare the `initial_best` architecture with `final_generation_ architecture` and identify the complete combination of structural and parameter changes associated with the non-improving outcome.
Do not treat `overall_best` as a failed offspring. Failure analysis must therefore be grounded in the explicit failed transition.

Evidence Analysis Requirements
For both success and failure reflection:

1. Compare architectures only within the same anonymous task. Objective values from different tasks must not be compared directly.
2. Pair every structural difference with the supplied same-seed evidence, including wins, losses, ties, median fitness, and reported logarithmic fitness gain.
3. Treat single-seed evidence as preliminary rather than robust.
4. When multiple structural components change together, treat them as one bundled change. Do not attribute the performance difference to a single operator unless the evidence isolates it.
5. Infer landscape properties only from the observable probe summary. Relevant properties may include separability, variable interaction, conditioning, modality, smoothness, ruggedness, and the required exploration–exploitation balance.
6. Treat inferred landscape properties as uncertain, evidence-based interpretations rather than oracle labels.
7. Do not infer a benchmark identity or use hidden problem metadata.
8. Identify contradictions across tasks and distinguish consistent evidence from weak, bundled, or conflicting evidence.
9. The absolute target gap is only a reporting objective. Learn from relative paired progress even when the target is not reached.
10. An isolated target hit is not sufficient evidence for a universal design rule.

Patch Construction Policy
Each proposed edit must express one directly usable design rule in the following form:

```
If <observable probe pattern>, prefer/avoid <design>
because <expected search behavior>.
```

Every edit must satisfy the following requirements:

- It must be supported by consistent evidence from at least two anonymous tasks and must not be contradicted by comparable evidence.
- It must describe a one-shot target architecture rather than the evolution of an architecture across generations.
- It must extract the smallest common structural motif supported by the evidence rather than reproduce a complete overall-best architecture.
- It must connect observable probe evidence to a landscape interpretation and then to topology, operator composition, routing, or parameter design.
- It must remain usable when no parent architecture, child architecture, generation number, or fitness feedback is available at inference time.
- It must preserve the distinction between tournament output cardinality, controlled by `n_parents`, and tournament selection pressure, controlled by `upper_k`.
- It must not contain benchmark identities, candidate identifiers, unexplained numeric thresholds, or a hard-coded complete topology.
- It must remain independently testable and fit within one second-level Markdown heading.
- Each `append` edit may introduce at most one second-level Markdown heading and one conditional design rule.
- Previous held-out rejection evidence is stronger than a new single-seed observation. Do not repeat a rejected rule unless the new evidence directly explains its previous losses.
- When the available evidence is weak, redundant, bundled, or contradictory, return an empty `edits` array.

Do not mention parent or child identifiers, generation numbers, lineage progression, or instructions to incrementally add, remove, increase, or decrease an operator in the generated knowledge content. Translate the training evidence into a direct, probe-conditioned architecture-design principle.

Required Output
Respond only with one valid JSON object:

```
{
  "key_evidence": [
    {
      "anonymous_task_index": 0,
      "probe_signature":
        "<exact numeric probe observation relevant to the edit>",
      "comparison":
        "<initial_best_to_overall_best or initial_best_to_final>",
      "structural_delta":
        "<concise complete graph, operator, and routing difference>",
      "paired_outcome":
        "<wins, losses, ties, and log10 fitness gain>",
      "attribution":
        "<isolated, bundled, or contradictory>."
    }
  ],
  "lineage_summary":
    "<cross-lineage conclusions, contradictions, and confidence>",
  "patch": {
    "reasoning":
      "<explain how the cited evidence supports the proposed rules>",
    "edits": [
      {
        "op": "append",
        "content":
          "## <landscape rule>\n\nIf <observable probe pattern>,
```

```
          prefer/avoid <one design> because <expected search behavior>."
      },
      {
        "op": "insert_after",
        "target": "<exact heading or text from Current knowledge>",
        "content": "<markdown>"
      },
      {
        "op": "replace",
        "target": "<exact text from Current knowledge>",
        "content": "<replacement>"
      },
      {
        "op": "delete",
        "target": "<exact text from Current knowledge>"
      }
    ]
  }
}
```

The operations shown above are examples. Include only the operations that are actually needed.
Return only two to four entries in `key_evidence`. Include only evidence that directly supports the final edits. Every edit must be traceable to at least one entry in `key_evidence`; otherwise, omit it. Stay within the supplied edit budget.

F.4.1 Patch Aggregation

The patch aggregation stage consolidates the independently generated patches before they are applied to the current knowledge. Failure-driven and success-driven patches are first merged separately to remove redundant edits and resolve conflicts within each group. The two resulting groups are then combined in a final merge, where failure-driven patches receive higher priority because they are intended to correct unreliable design rules. After aggregation, the selected edits are applied to the current knowledge. The aggregation prompts and the final output format are given below.

Complete Prompt for Patch Aggregation

Role
You are a knowledge-edit coordinator. Consolidate independently generated patches into one coherent, non-redundant patch for updating the current knowledge.

Input
You are given the current knowledge and one of the following patch groups:

- a group of patches generated from failure analysis;
- a group of patches generated from success analysis; or
- two pre-merged groups containing failure-driven and success-driven patches.

The current knowledge and patches are provided as follows:
Current knowledge
`[CURRENT knowledge]`
Patches to Merge
`[PATCHES TO MERGE]`
Aggregation Stage
`[FAILURE GROUP, SUCCESS GROUP, OR FINAL MERGE]`

Aggregation Instructions
For a failure-driven patch group, preserve generalizable corrective edits that explain or address recurring failure patterns. For a success-driven patch group, preserve effective design patterns while discarding isolated or task-specific observations.
For the final merge of both groups, treat failure-driven edits as higher priority. If a failure-driven edit and a success-driven edit address the same issue, retain the failure-driven formulation unless the success-driven edit has substantially stronger evidence. Preserve success-driven edits that provide distinct and non-redundant insights.
In all aggregation stages:

1. Deduplicate overlapping edits and retain the clearest general formulation.

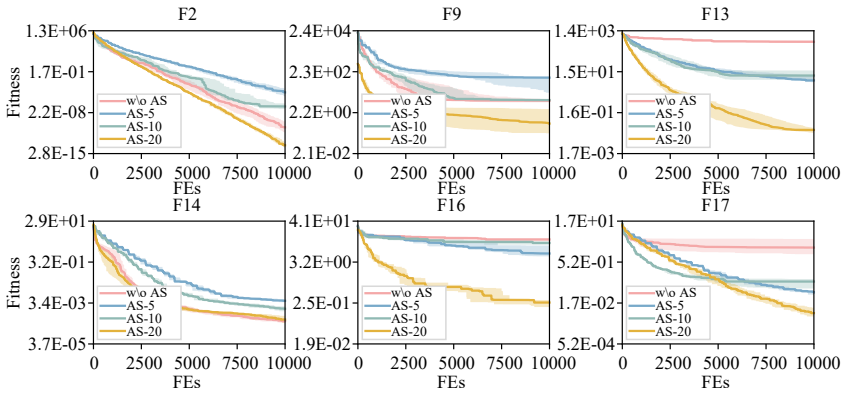


Figure 10: AS-k denotes k architecture-search generations during training, while w/o AS directly uses the initial architecture without further refinement. Lower fitness is better.

Table 9: Token consumption under different experience-collection budgets.

Variant	Token Consumption
w/o AS	322.8k
AS-5	1937.6k
AS-10	3227.7k
AS-20	6545.4k

2. Resolve conflicting edits using the strength, consistency, and generality of their supporting evidence.
3. Preserve unique insights that address distinct design issues.
4. Prefer edits supported by multiple source patches.
5. Ensure that no two edits target the same text region.
6. Preserve the estimated number of supporting source patches as `support_count`.
7. Preserve the source type as `failure`, `success`, or the corresponding source type of the retained edit.

Output Format
Respond only with one valid JSON object:

```
{
  "reasoning": "<summary of the consolidation decisions>",
  "edits": [
    {
      "op": "append|insert_after|replace|delete",
      "target": "<target text when required>",
      "content": "<markdown content when required>",
      "support_count": <integer>,
      "source_type": "failure|success"
    }
  ]
}
```

Use only the operations that are actually needed. Do not include additional keys, explanations, Markdown fences, or surrounding prose.

Knowledge Update
After the final aggregation, the retained edits are applied to the current knowledge document according to their `op`, `target`, and `content` fields.

G LIMITATIONS AND ADDITIONAL ANALYSIS

FETK incurs an upfront cost during the Experience Collection stage, since architecture search is performed on each training problem before the learned knowledge can be reused. To examine the trade-off between experience quality and collection cost, we vary the number of architecture-search generations as $k \in \{5, 10, 20\}$ while keeping all other settings fixed. AS-k denotes k architecture-search generations during training, while w/o AS directly uses the initial architecture without further refinement. As shown in Fig. 10, increasing the experience-collection budget generally improves the downstream optimization performance, indicating that more architecture-search trajectories provide more useful design experience for subsequent knowledge learning. However, Table 9 shows that the corresponding token consumption also increases with the number of search generations because additional architecture refinement requires more LLM calls. Therefore, a larger experience collection budget improves the quality of the learned guidance at the expense of higher token cost. A promising direction for future work is to improve the efficiency of experience collection, such that informative design experience can be obtained with fewer LLM calls and lower token consumption while preserving the quality of the learned knowledge.