

Accelerating Neural Architecture Search with Code Semantics

Zhen Liu^{1*}, Wanqi Zhou^{1*}, Shuanghao Bai¹,
Wei Song³, He Zhang^{4,5}, Yuhan Liu^{2†}, Jingwen Fu^{4,5†}

¹State Key Laboratory of Human-Machine Hybrid Augmented Intelligence,
Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University

²MiLM Plus, Xiaomi Inc.

³Department of Computer Science and Technology, North China University of Technology

⁴Zhongguancun Academy

⁵Zhongguancun Institute of Artificial Intelligence

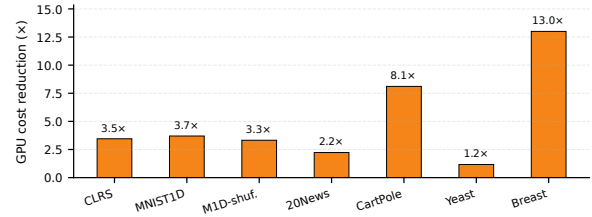
*Equal contribution. †Correspondence: yhliu0808@gmail.com; fujingwen@bza.edu.cn

Abstract

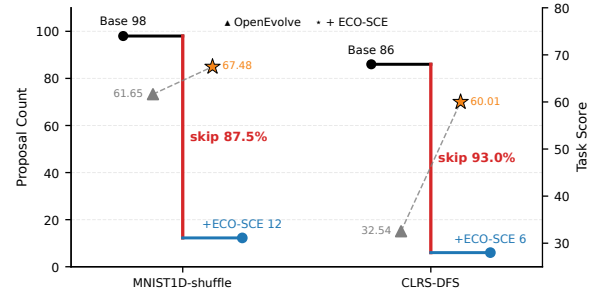
Neural architecture search (NAS) is a core technique for automating AI model design, yet its efficiency is limited by the high cost of reliable architecture evaluation. To address this bottleneck, we introduce *code-semantic transition modeling* for NAS acceleration: before spending evaluator budget, we ask whether a proposal’s semantic change from its parent suggests a worthwhile architecture evolution. We instantiate this idea with ECO-SCE, an *Evolutionary Code Semantic Change Estimator* placed between the architecture generator and evaluator. Given a parent-to-proposal code transition, ECO-SCE filters likely low-gain proposals and reserves evaluator calls for proposals that are more likely to provide useful evolutionary progress. To train ECO-SCE, we construct NB101-SemPair from NAS-Bench-101 by sampling beneficial, harmful, and marginal transitions with diverse semantic and structural changes. ECO-SCE learns risk-aware transition utility from global semantic shifts, sparse semantic overlaps, and explicit structural edits, allowing it to skip likely low-gain proposals while sending promising or uncertain transitions to the evaluator. Across seven downstream benchmarks including CLRS, ECO-SCE reduces average GPU time by 70.88% and improves task performance in 6 out of 7 cases when inserted zero-shot into OpenEvolve.

1 Introduction

Neural architecture search (NAS) is a key step toward automated AI model design, aiming to reduce manual architecture engineering by automatically discovering high-performing neural networks (Elsken et al., 2019). Most NAS methods follow a generate–evaluate–refine loop: candidate architectures are proposed, evaluated, and then used to guide subsequent search. This loop underlies reinforcement-learning-based NAS (Zoph and



(a) GPU cost reduction across downstream benchmarks. Each bar reports the relative reduction ratio of OpenEvolve over ECO-SCE-guided evaluation. GPU hours are measured on a single NVIDIA RTX 3090 GPU.



(b) Valid evaluator-call collapse and task-score improvement on representative benchmarks. ECO-SCE substantially reduces valid evaluator calls while preserving or improving downstream performance.

Figure 1: Efficiency and effectiveness of ECO-SCE-guided architecture search. Detailed experimental results are reported in Section 4.3 and Appendix H.

Le, 2017), evolutionary NAS (Real et al., 2019), differentiable NAS (Liu et al., 2019), and recent LLM-driven architecture search (Chen et al., 2023; Nasir et al., 2024; Ji et al., 2025). Despite their different search mechanisms, these methods share a common bottleneck: reliable feedback remains costly, which usually requires training, partial training, validation, or benchmark querying (White et al., 2023). The bottleneck is amplified in evolution-style and LLM-based code-rewriting NAS. Such methods can flexibly generate executable architecture proposals by modifying existing code, but many proposals provide little gain over their parents (Chen et al., 2023; Nasir et al.,

2024; Ji et al., 2025).

A natural response is to make each evaluation cheaper. Prior work follows this direction through weight sharing and one-shot NAS (Pham et al., 2018; Guo et al., 2020), low- or multi-fidelity evaluation (Trofimov et al., 2023), zero-cost proxies (Abdelfattah et al., 2021), and performance predictors (Baker et al., 2017). These methods reduce the cost or fidelity of assessing a candidate, but they still assume that a generated architecture should receive some form of quality assessment.

However, proposal-based NAS exposes two limitations that are not directly addressed by cheaper architecture scoring. **First**, the decision unit is different. In evolutionary and LLM-based code-rewriting NAS, a candidate is generated by modifying a parent architecture. Therefore, the key question is not only whether the candidate has a good estimated score, but whether the edit provides meaningful gain over its parent. A proposal can look reasonable under a proxy or predictor while still being redundant, marginal, or low-gain relative to the current architecture. **Second**, a practical filter should not require downstream task-specific training. Many predictors, calibrated proxies, or low-fidelity strategies rely on evaluated candidates, target search-space statistics, or task-specific validation signals. If each new task or generator requires additional evaluated architectures to train or tune a screening model, the evaluation cost is shifted rather than removed. Thus, proposal-based NAS needs a model that can learn transition utility offline and be deployed zero-shot in downstream search. This motivates our transition-level question: before invoking the evaluator, can we estimate from the semantic and structural change between a parent and a proposal whether the proposal deserves evaluation budget?

To answer this question, we design ECO-SCE around two principles. **First**, the model should learn *parent-relative transition utility* rather than task-specific architecture scores. Since NAS-Bench-101 (Ying et al., 2019) provides evaluated architectures but not evolution trajectories, we construct NB101-SemPair by pairing architectures into directed architecture-change pairs. Each pair is treated as a parent-to-proposal transition and supervised by the proposal’s performance change over the parent. The pairs are sampled to cover beneficial, harmful, and marginal transitions under diverse semantic and structural changes, enabling ECO-SCE to learn general patterns of use-

ful architecture edits without being trained inside a downstream search loop. **Second**, ECO-SCE should rely only on signals available before evaluation. Since downstream accuracy or validation feedback is unavailable at screening time, the decision must be made from the architecture change itself. We represent each transition using global semantic shift, sparse semantic overlap, and explicit structural-edit features, which describe how the proposal differs from the parent at both code-semantic and code-structural levels. ECO-SCE then combines direction, magnitude, and local-evidence signals in a risk-aware routing policy, skipping clearly low-gain transitions while preserving uncertain or potentially beneficial proposals for evaluation.

Figure 1 illustrates the effect of ECO-SCE after zero-shot insertion into OpenEvolve. Across seven downstream benchmarks, ECO-SCE consistently reduces GPU cost. On representative MNIST1D-shuffle and CLRS-DFS runs, it sharply reduces full evaluator calls while improving task scores.

Our contributions are summarized as follows:

- We formulate NAS acceleration as *code-semantic transition modeling*, shifting the focus from scoring architectures in isolation to estimating whether a parent-to-proposal change deserves costly evaluation.
- We construct *NB101-SemPair*, a transition-level dataset derived from NAS-Bench-101, and propose *ECO-SCE*, a lightweight zero-shot screening model that learns risk-aware proposal utility from semantic and structural code changes.
- We show that ECO-SCE preserves high-gain transitions better than generic filtering baselines and accelerates downstream architecture evolution, reducing average GPU hours by 70.88% across seven benchmarks when inserted zero-shot into OpenEvolve.

2 Related Work

Classical and evolutionary NAS. Neural architecture search (NAS) automates model design by searching over architecture spaces under validation or benchmark feedback. Early reinforcement-learning-based NAS trains a controller to sample architectures and updates it using evaluated performance (Zoph and Le, 2017). Evolutionary NAS

maintains a population of candidate architectures and improves it through mutation, selection, and survival mechanisms (Real et al., 2019). Differentiable NAS relaxes discrete architecture choices into continuous variables, enabling gradient-based optimization of architecture parameters (Liu et al., 2019). Although these paradigms differ in their search mechanisms, they all rely on evaluated candidates to guide subsequent search. This dependence is especially explicit in evolutionary NAS: candidates are generated by modifying existing architectures, and after evaluation, the search algorithm can decide whether they should enter the population or archive. This makes evaluator calls a critical bottleneck in architecture evolution. Our work follows this evolutionary view, but focuses on the pre-evaluation decision of whether a generated transition deserves expensive evaluation.

LLM-driven architecture generation. Recent work uses LLM to extend NAS from predefined operators or fixed search spaces to code-level architecture generation (Zheng et al., 2023; Chen et al., 2023; Hu et al., 2025). LLMatic combines LLM-based code generation with quality-diversity optimization, using LLMs to introduce diverse variations in neural architecture code (Nasir et al., 2024). CoLLM-NAS studies a collaborative multi-agent design, where a Navigator LLM guides the search direction, a Generator LLM synthesizes candidate architectures, and a Coordinator manages their interaction (Li et al., 2026). Other LLM-driven NAS methods further improve proposal generation through reflective zero-cost feedback (Ji et al., 2025), design-principle transfer across search spaces (Zhou et al., 2025), and code-difference-based refinement (Sridhar and Chen, 2025). These approaches make proposal generation more flexible by rewriting executable architecture code rather than selecting from fixed operators. However, a single code rewrite can entangle structural, implementation, and module-level changes, making many valid proposals low-gain relative to their parents. This motivates a complementary question to architecture generation: after a proposal is generated, can we determine whether the transition is worth evaluating?

Efficient architecture evaluation. A major line of NAS research reduces search cost by making architecture evaluation cheaper. Weight-sharing and one-shot NAS amortize training by sharing parameters among many sub-architectures (Pham

et al., 2018; Guo et al., 2020). Low-fidelity and multi-fidelity evaluation approximate final performance using cheaper signals such as shorter training, smaller datasets, or partial training schedules (Trofimov et al., 2023). Zero-cost proxies avoid full training by computing training-free scores from initialization, gradients, or architecture statistics (Abdelfattah et al., 2021). Performance predictors learn to estimate final accuracy or ranking from architecture representations or early learning signals (Baker et al., 2017). These methods are effective for reducing the cost of scoring individual architectures. In architecture evolution, however, the operational decision is parent-relative: a proposal should be evaluated only if it is likely to improve over its parent or provide useful feedback for the future trajectory. A candidate may have plausible standalone quality but still be redundant or low-gain relative to its parent. ECO-SCE therefore differs from standard proxies and predictors by modeling the code-semantic and code-structural change of a directed parent-to-proposal transition.

3 Method

ECO-SCE is designed to answer a parent-relative question before costly evaluation: given a parent architecture and a generated proposal, does this transition represent a worthwhile architecture evolution? Instead of scoring the proposal in isolation, ECO-SCE models the directed code transition from the parent to the proposal and makes a risk-aware routing decision.

3.1 Problem Formulation

As shown in Figure 2, ECO-SCE is inserted between the architecture generator and the costly evaluator. Given a parent architecture a_p , the generator produces a proposal a_c . A standard architecture evolution pipeline sends a_c directly to the evaluator to obtain a score, which is then used to update the archive or population. ECO-SCE instead asks a pre-evaluation question: is the transition from a_p to a_c likely to be useful enough to justify evaluator cost? We define a directed architecture transition as

$$\tau = (a_p, a_c), \quad (1)$$

and define its hindsight evolutionary gain as

$$\Delta(\tau) = s(a_c) - s(a_p), \quad (2)$$

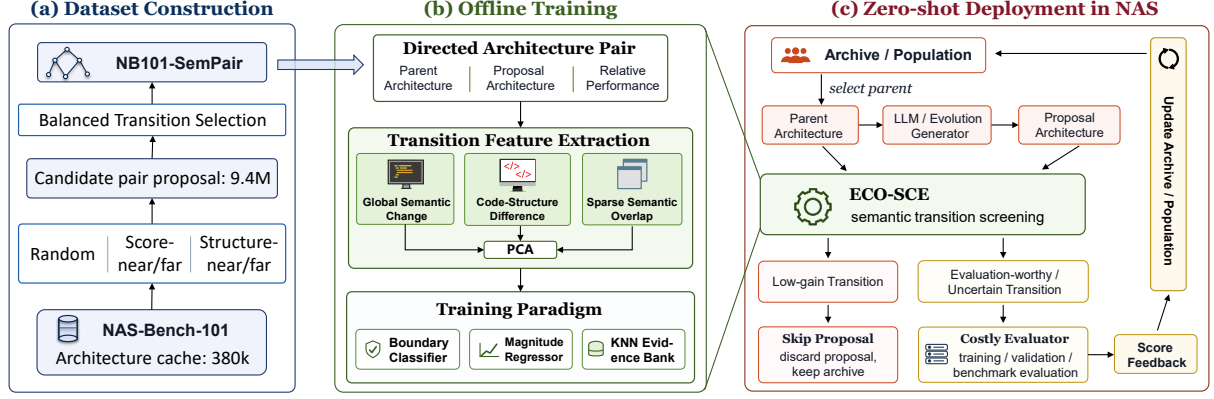


Figure 2: Overview of ECO-SCE. (a) Dataset construction: ECO-SCE builds NB101-SemPair from NAS-Bench-101 by generating candidate architecture pairs and selecting balanced transitions with random, score-near/far, and structure-near/far sampling strategies. (b) Offline training: each parent-proposal architecture pair is encoded into transition features, including global semantic change, code-structure difference, and sparse semantic overlap, which are used to train a boundary classifier, magnitude regressor, and KNN evidence bank. (c) Zero-shot deployment in NAS: during search, ECO-SCE screens generated proposals before costly evaluation, filtering low-gain transitions while forwarding evaluation-worthy or uncertain proposals to the full evaluator. The resulting score feedback updates the archive/population for subsequent proposal generation.

where $s(\cdot)$ denotes the task score. ECO-SCE does not aim to replace the evaluator or predict the exact final score of a_c . Instead, it estimates the utility of the parent-to-proposal transition before evaluation. Likely low-gain transitions are skipped, while uncertain or potentially useful transitions are sent to the evaluator as a safety fallback.

3.2 NB101-SemPair: Transition-Level Supervision

NAS-Bench-101 (Ying et al., 2019) provides evaluated architectures but no explicit evolution trajectories. We construct *NB101-SemPair* by converting its static architecture records into directed parent-to-proposal architecture-change pairs. Each pair is supervised by the relative gain $\Delta(\tau) = s(a_c) - s(a_p)$, enabling ECO-SCE to learn parent-relative transition utility rather than task-specific absolute scores. The dataset is designed to cover beneficial, harmful, and marginal transitions under diverse code-level changes. Algorithm 1 details the construction procedure.

3.3 Code-Semantic Transition Representation

For each transition $\tau = (a_p, a_c)$, ECO-SCE represents the architecture edit from the parent to the proposal rather than the proposal alone. Let $\mathcal{C}_p$ and $\mathcal{C}_c$ denote the architecture-defining EVOLVE code regions of a_p and a_c . We construct a transition

feature

$$x(\tau) = [x_{\text{dense}}; x_{\text{code}}; x_{\text{sparse}}] \in \mathbb{R}^{1316}, \quad (3)$$

which combines three complementary pre-evaluation signals: global semantic shift, explicit structural edit, and sparse semantic overlap.

Global semantic shift. The dense feature captures the overall semantic direction of the code rewrite:

$$\begin{aligned} x_{\text{dense}} &= \text{Embed}_{\text{dense}}(\mathcal{C}_c) - \text{Embed}_{\text{dense}}(\mathcal{C}_p), \\ x_{\text{dense}} &\in \mathbb{R}^{1024}. \end{aligned} \quad (4)$$

We implement $\text{Embed}_{\text{dense}}(\cdot)$ with text-embedding-v4 (Zhang et al., 2025). This difference form describes how the proposal moves away from the parent in dense semantic space.

Explicit structural edit. The code feature $x_{\text{code}} \in \mathbb{R}^{285}$ records concrete program-level changes that may be blurred by dense embeddings. It summarizes token statistics, AST patterns, line edits, module-call types, and def-use relations, using parent/proposal profiles, their differences, and similarity scores.

Sparse semantic overlap. The sparse feature $x_{\text{sparse}} \in \mathbb{R}^7$ measures how similar the parent and proposal remain in sparse embedding space. It compares the nonzero sparse dimensions of the two code embeddings using nonzero counts, intersection and union sizes, Jaccard similarity, sparse

Algorithm 1 Construction of NB101-SemPair

Require: NAS-Bench-101 records $\mathcal{D}$, random seed 42

- 1: Randomly sample 420,000 architecture records from $\mathcal{D}$.
 - 2: Partition sampled records into 150 score-quantile bins.
 - 3: Uniformly sample from bins to obtain a score-balanced cache $\mathcal{A}$ of 380,000 architectures.
 - 4: Build a score index over $\mathcal{A}$ by benchmark score.
 - 5: Build a code index over $\mathcal{A}$ by lightweight code statistics, including line count and token count.
 - 6: Generate 9.4M candidate pairs using the five policies in Table 11.
 - 7: Remove self-pairs and duplicated pairs.
 - 8: **for** each remaining ordered pair (a_p, a_c) **do**
 - 9: Compute $\Delta(\tau) = s(a_c) - s(a_p)$.
 - 10: Assign the pair to beneficial if $\Delta(\tau) > 0$ and harmful if $\Delta(\tau) < 0$.
 - 11: **end for**
 - 12: Stratify beneficial and harmful pairs separately into 16 bins by $|\Delta(\tau)|$.
 - 13: Sample 5,000 beneficial and 5,000 harmful transitions from the 32 cells.
 - 14: Avoid keeping both (a_i, a_j) and (a_j, a_i) ; limit each architecture to at most two selected pairs.
 - 15: Split pairs by parent architecture into 60%/20%/20% train/validation/test sets.
 - 16: **return** NB101-SemPair with 10,000 directed architecture-change pairs.
-

ℓ_1 distance, and sparse cosine similarity. This feature helps distinguish small semantic edits from larger semantic shifts. Together, these features describe both the semantic and structural change induced by a proposal, allowing ECO-SCE to identify low-gain or redundant edits before evaluation. Before estimation, we standardize $x(\tau)$ using training-set statistics and project it to a compact representation:

$$z(\tau) = \text{PCA}_d(\text{Std}(x(\tau))), \quad d = 64. \quad (5)$$

The standardization and PCA projection are fitted only on the training split and reused unchanged for validation, testing, and downstream deployment.

3.4 ECO-SCE Estimator

ECO-SCE uses a lightweight estimator bundle over the shared representation $z(\tau)$. It consists

of three components: a dual-boundary classifier for transition direction, a magnitude regressor for change strength, and a KNN evidence bank for local reliability calibration. All components are trained or constructed on NB101-SemPair and are fused only at inference time by the routing policy.

Dual-boundary classification. Given the transition gain $\Delta(\tau) = s(a_c) - s(a_p)$, we define a margin $m = 0.02$ and construct two boundary labels:

$$\begin{aligned} y_{\text{bad}}(\tau) &= \mathbb{I}[\Delta(\tau) \leq -m], \\ y_{\text{good}}(\tau) &= \mathbb{I}[\Delta(\tau) \geq m]. \end{aligned} \quad (6)$$

We use two linear logistic regressors over $z(\tau)$ to estimate the negative and positive transition boundaries:

$$\begin{aligned} p_{\text{bad}}(z) &= P(\Delta \leq -m \mid z), \\ p_{\text{good}}(z) &= P(\Delta \geq m \mid z). \end{aligned} \quad (7)$$

These two classifiers form a single ordered boundary module. The negative boundary detects clearly harmful transitions, while the positive boundary detects clearly beneficial transitions. Transitions within $-m < \Delta < m$ are treated as small or ambiguous changes and are handled conservatively by the routing policy.

Magnitude and signed benefit estimation. The boundary probabilities indicate transition direction but not change scale. We therefore train a linear ridge regressor over $z(\tau)$ to estimate the absolute gain magnitude:

$$\hat{a}(z) \approx |\Delta(\tau)|. \quad (8)$$

Since $\hat{a}(z)$ is unsigned, we combine it with the boundary probabilities to obtain a signed benefit estimate:

$$\hat{r}(z) = (p_{\text{good}}(z) - p_{\text{bad}}(z)) \hat{a}(z). \quad (9)$$

The sign of $\hat{r}(z)$ predicts whether a transition improves or degrades the parent. For risk control, high-impact transitions with uncertain direction are evaluated rather than skipped.

KNN evidence bank. The KNN evidence bank calibrates global predictions with local transition evidence. For a candidate transition, we retrieve its $K = 80$ nearest training transitions by Euclidean distance in the PCA space, denoted by

$\mathcal{N}_K(z)$. The local bad and good supports are

$$\begin{aligned} s_{\text{bad}}(z) &= \frac{1}{K} \sum_{i \in \mathcal{N}_K(z)} y_{\text{bad}}(\tau_i), \\ s_{\text{good}}(z) &= \frac{1}{K} \sum_{i \in \mathcal{N}_K(z)} y_{\text{good}}(\tau_i). \end{aligned} \quad (10)$$

We estimate local uncertainty using the 0.9 quantile of neighbor residuals:

$$u(z) = \text{Quantile}_{0.9}(\{|\Delta(\tau_i) - \hat{r}(z_i)| \mid i \in \mathcal{N}_K(z)\}) \quad (11)$$

Here, s_{bad} and s_{good} indicate whether similar transitions were mostly harmful or beneficial, while $u(z)$ measures local prediction uncertainty.

3.5 Risk-Aware Routing

As shown in Figure 2, the routing policy combines boundary, benefit, and KNN evidence:

$$d(\tau) = \begin{cases} \text{reject}, & \mathcal{R}_{\text{rej}}(z) = 1, \\ \text{uncertain}, & \mathcal{R}_{\text{unc}}(z) = 1, \\ \text{eval-worthy}, & \text{otherwise.} \end{cases} \quad (12)$$

Here, $d(\tau)$ denotes a screening state rather than a final archive action.

The reject rule is activated when global and local evidence consistently indicate a harmful transition:

$$\begin{aligned} \mathcal{R}_{\text{rej}}(z) = 1 \quad \text{if} \quad & p_{\text{bad}}(z) \geq \tau_{\text{bad}}, \\ & p_{\text{good}}(z) \leq \tau_{\text{protect}}, \\ & s_{\text{bad}}(z) \geq p_{\text{bad}}(z), \\ & \hat{r}(z) + \lambda u(z) < 0. \end{aligned} \quad (13)$$

The uncertain-transition rule is defined symmetrically: it requires high p_{good} , low p_{bad} , sufficient local good support, and a positive uncertainty-adjusted benefit. All remaining non-rejected transitions are regarded as evaluation-worthy. For uncertain transitions, ECO-SCE uses $\hat{r}(z)$ only as an auxiliary screening estimate rather than a surrogate evaluator score.

Both uncertain and evaluation-worthy transitions are sent to the full evaluator. Only rejected proposals bypass evaluation, and only evaluator-verified scores can update the archive or influence subsequent search.

4 Experiments

We organize the experiments around five questions covering offline transition-utility estimation, downstream architecture evolution,

evaluator-budget allocation, comparisons with filtering/proxy baselines, and ablation/sensitivity studies. Unless stated otherwise, OpenEvolve-based experiments use 100 evolution rounds, and EvoPrompting-based (Chen et al., 2023) experiment follows its original 1,600-iteration setting. GPU hours denote the end-to-end evolution runtime, including search, ECO-SCE screening, and evaluation, measured on a single RTX 3090 GPU. A detailed component-wise runtime breakdown, covering both offline preparation and online inference costs, is provided in the Appendix. Hyperparameter settings and selection are provided in Appendix B.

4.1 RQ1: Can ECO-SCE Identify High- and Low-Gain Proposals?

Each sample in NB101-SemPair is an ordered parent-to-proposal pair with a known score difference, together with semantic and code-structure differences between the two architectures. This allows us to directly test whether ECO-SCE can distinguish high-gain proposals from low-gain ones before full evaluation. Since ECO-SCE is used for routing rather than plain classification, we report routing-oriented metrics as shown in Table 1. Table 1 shows that larger transition-level supervision improves routing quality. With 10,000 training pairs, ECO-SCE reaches the highest proxy reduction of 75.02% and the best AUC of 85.76%, while keeping 94.80% of top-10 high-gain proposals. Accept Precision also reaches 90.39%, and Missed Improve remains below 9%, suggesting that ECO-SCE saves more evaluations without blindly filtering valuable transitions.

4.2 RQ2: Can ECO-SCE Improve Downstream Architecture Evolution?

We next study whether ECO-SCE improves an existing architecture evolution pipeline on CLRS (Velickovic et al., 2022), a 30-task algorithmic reasoning benchmark. We compare against SPARK (Liu et al., 2026), EvoPrompting (Chen et al., 2023), OpenEvolve (Sharma, 2025), and Triplet-GMPNN (Ibarz et al., 2022). As shown in Table 2, OpenEvolve+ECO-SCE achieves the best average accuracy over all tasks, 86.10%, and the best cross-task accuracy excluding DFS, 87.00%. It also increases task-level wins from 1/30 for vanilla OpenEvolve to 16/30. These results show that ECO-SCE improves evaluator-budget allocation during archi-

Training Pairs	Proxy Red. $\uparrow$	Missed Imp. $\downarrow$	Accept Bad $\downarrow$	Accept Prec. $\uparrow$	Reject NPV $\uparrow$	Top-10 Kept $\uparrow$	AUC $\uparrow$
500	35.80%	5.20%	6.82%	93.18%	87.62%	96.00%	76.86%
1,000	51.60%	7.20%	11.35%	88.65%	85.71%	94.00%	80.58%
5,000	67.22%	8.12%	11.39%	88.61%	87.87%	96.00%	81.34%
10,000	75.02%	8.98%	9.61%	90.39%	88.19%	94.80%	85.76%

Table 1: Effect of training-pair scale on ECO-SCE. Proxy Red. measures the fraction of candidates that avoid full evaluation; Missed Imp. measures beneficial proposals mistakenly skipped; Accept Bad and Accept Prec. measure the reliability of uncertain transition proposals; Reject NPV measures whether rejected proposals are truly low-gain; Top-10 Kept measures preservation of the highest-gain proposals; and AUC measures overall separability.

Method	Avg. Params $\downarrow$	DFS Acc. $\uparrow$	Avg. Acc. $\uparrow$	Avg. Acc. w/o DFS $\uparrow$	Wins
Triplet-GMPNN (Ibarz et al., 2022)	467.3K	47.79	75.98	76.95	1/30
OpenEvolve (Sharma, 2025)	498.6K	32.54	62.69	63.73	1/30
EvoPrompting (Chen et al., 2023)	478.2K	68.14	80.88	81.32	2/30
Spark (Liu et al., 2026)	470.2K	83.74	83.91	83.92	10/30
OpenEvolve+ECO-SCE (ours)	485.9K	60.01	86.10	87.00	16/30

Table 2: CLRS downstream architecture evolution results. For EvoPrompting and Triplet-GMPNN, we report the per-task CLRS results from their original papers. All other methods are evolved on the CLRS-DFS task under Spark’s experimental setting and evaluated zero-shot on all 30 CLRS algorithmic reasoning tasks. “DFS Acc.” reports performance on the search task. “Avg. Acc.” averages over all tasks, while “Avg. Acc. w/o DFS” excludes the search task and measures cross-task performance. “Wins” counts the number of tasks on which each method achieves the best accuracy among the compared methods. Detailed results on each task can be seen in Table 17.

texture evolution, enabling OpenEvolve to discover architectures with stronger cross-task algorithmic reasoning performance. Broader zero-shot results on seven downstream benchmarks are reported in Appendix H.

4.3 RQ3: Does ECO-SCE Allocate Evaluation to More Promising Proposals?

We examine whether ECO-SCE reduces evaluation cost through meaningful allocation rather than invalid-program filtering. We consider only valid proposals that pass basic execution checks and test whether ECO-SCE can screen runnable but low-gain transitions before full evaluation. Table 3 reports two complementary settings. MNIST1D-shuffle covers four architecture families (LR, MLP, CNN, and GRU), while CLRS-DFS uses the benchmark-defined DFS setting. On MNIST1D-shuffle, ECO-SCE evaluates only 12.25 of 98.00 valid proposals on average, skipping 87.50%. On CLRS-DFS, it evaluates 6 of 86 valid proposals, skipping 93.02%. For MNIST1D-shuffle CNN, among 93 valid proposals, 83 are rejected, 3 are routed as uncertain transitions and subsequently fully evaluated, and 7 are directly evaluated. Thus, the reduction comes from screening valid low-gain transitions rather than invalid code, while allocating full-evaluator calls to a compact subset of po-

Dataset	Setting	Valid	Eval.	Skip	Skip Rate
MNIST1D-shuf.	LR	100	19	81	81.00%
	MLP	99	16	83	83.84%
	CNN	93	10	83	89.25%
	GRU	100	4	96	96.00%
	Avg.	98.00	12.25	85.75	87.50%
CLRS-DFS	CLRS-protocol	86	6	80	93.02%

Table 3: Evaluator-call analysis of OpenEvolve+ECO-SCE over 100 search iterations. “Valid” denotes runnable proposals, “Eval.” denotes valid proposals sent to the full evaluator, and “Skip” denotes valid proposals screened before full evaluation.

tentially useful proposals.

4.4 RQ4: Is ECO-SCE More Than Simple Filtering?

To verify that ECO-SCE does not merely benefit from reducing the number of evaluations, we compare it with generic filtering and proxy-evaluation alternatives. We consider two complementary settings. First, on NB101-SemPair, we perform a controlled offline comparison where random and rule-based filters are matched to ECO-SCE with the same filtering budget. This isolates whether ECO-SCE selects better proposals under the same amount of skipped evaluation. Second, inside OpenEvolve, we compare ECO-SCE with random

Filter	Missed Imp. ↓	Top-10 Kept ↑	Eval. Red. ↑
Random	74.97%	25.53%	75.02%
Rule-based	64.08%	38.80%	75.02%
Acc. Pred.	54.54%	99.40%	75.02%
ECO-SCE	8.98%	94.80%	75.02%

Table 4: Controlled filtering comparison on NB101-SemPair under the same evaluation-reduction budget.

filtering and ZeroProxy (Abdelfattah et al., 2021) under comparable search budgets. This evaluates whether the learned transition utility leads to better downstream architecture evolution.

Controlled filtering on NB101-SemPair. We first compare ECO-SCE with random filtering, rule-based structural filtering, and an accuracy-prediction filter under the same evaluation-reduction budget of 75.02%. This setting controls the number of skipped evaluations, so the comparison reflects which proposals each method preserves. The results in Table 4 suggest that standalone accuracy prediction is insufficient for architecture evolution: a proposal may have a high predicted score but still provide little gain over its parent. In contrast, ECO-SCE uses parent-to-proposal transition information to identify genuinely useful evolutionary changes. Details of the rule-based filter are provided in Appendix E.

Comparison inside OpenEvolve. We further test whether this filtering advantage transfers to an actual OpenEvolve loop on the MNIST1D-shuffle CNN setting. To isolate proposal-selection quality, Random Filter is budget-matched to ECO-SCE: after ECO-SCE sends 10 runnable proposals to the full evaluator, Random Filter randomly selects the same number of runnable proposals for evaluation. As shown in Table 5, ECO-SCE improves evaluator allocation, rather than merely reducing the number of evaluations.

4.5 RQ5: Are ECO-SCE’s Components and Routing Hyperparameters Robust?

We provide component and feature-level ablations and routing-hyperparameter sensitivity studies in Appendices F and G. Appendix Table 15 shows complementary roles of the boundary classifier, evidence bank, and magnitude regressor. Appendix Table 16 further compares semantic-only, structural-only, and combined representations. Structural features are stronger alone, while

Method	Full Eval. ↓	GPU Hours ↓	Best Acc. ↑
OpenEvolve	100	1.27	59.10
+ ZeroProxy	18	1.23	61.10
+ Random Filter	10	0.68	67.40
+ ECO-SCE	10	0.46	72.40

Table 5: Comparison with filtering and proxy-evaluation strategies inside OpenEvolve on MNIST1D-shuffle CNN setting. Random Filter is budget-matched to ECO-SCE: after ECO-SCE sends 10 proposals to the full evaluator, Random Filter randomly selects the same number of runnable proposals for evaluation.

semantic features provide complementary information; combining both improves AUROC and Macro-F1 by 3.75 and 5.88 points. Appendix Figure 3 shows stable performance across broad routing-hyperparameter ranges. Overall, ECO-SCE benefits from complementary representation and estimator components rather than any single feature or narrowly tuned threshold.

5 Conclusion

We introduced ECO-SCE, a code-semantic transition model that accelerates NAS by deciding whether parent-to-proposal transitions merit costly evaluation. Instead of scoring architectures independently, ECO-SCE learns parent-relative proposal utility from NB101-SemPair and routes proposals using semantic-change, sparse-overlap, and code-structural signals. Experiments on NB101-SemPair and downstream OpenEvolve show reduced evaluator usage and GPU cost while preserving or improving search quality. These results establish code-semantic transition modeling as a promising direction for evaluator-aware architecture evolution. Code will be released at [this repository](#) for reproducibility.

Limitations

We summarize the limitations as follows. First, ECO-SCE depends on code-level semantic representations, so noisy, overly long, or poorly structured architecture code may increase token cost and weaken transition features. Second, since NB101-SemPair is derived from NAS-Bench-101, validation on larger architecture spaces remains future work. Third, as a pre-evaluation screening model, ECO-SCE may still miss some beneficial proposals, although uncertain transitions are sent to the evaluator as a fallback. Future work will extend ECO-SCE to broader search spaces.

Acknowledgement

This work is supported by the Zhongguancun Academy (Grant No.XTS0070). He Zhang is supported by the Zhongguancun Academy under Grant No.C20250512 and the State Key Laboratory of Human-Machine Hybrid Augmented Intelligence, Xi'an Jiaotong University, No.HMHAI-KF202504.

References

- Mohamed S. Abdelfattah, Abhinav Mehrotra, Lukasz Dudziak, and Nicholas Donald Lane. 2021. Zero-cost proxies for lightweight NAS. In *ICLR*. OpenReview.net.
- Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*.
- Angelica Chen, David Dohan, and David R. So. 2023. Evoprompting: Language models for code-level neural architecture search. In *NeurIPS*.
- Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20:55:1–55:21.
- Sam Greydanus and Dmitry Kobak. 2024. Scaling down deep learning with MNIST-1D. In *Proceedings of the 41st International Conference on Machine Learning*.
- Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. 2020. Single path one-shot neural architecture search with uniform sampling. In *European conference on computer vision*, pages 544–560. Springer.
- Yuxuan Hu, Jihao Liu, Ke Wang, Jinliang Zheng, Weikang Shi, Manyuan Zhang, Qi Dou, Rui Liu, Aojun Zhou, and Hongsheng Li. 2025. [LM-searcher: Cross-domain neural architecture search with LLMs via unified numerical encoding](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 9408–9421, Suzhou, China. Association for Computational Linguistics.
- Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bannani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bosnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Velickovic. 2022. [A generalist neural algorithmic learner](#). In *Learning on Graphs Conference, LoG 2022, 9-12 December 2022, Virtual Event*, Proceedings of Machine Learning Research, page 2. PMLR.
- Zipeng Ji, Guanghui Zhu, Chunfeng Yuan, and Yihua Huang. 2025. RZ-NAS: enhancing llm-guided neural architecture search via reflective zero-cost strategy. In *ICML, Proceedings of Machine Learning Research*. PMLR / OpenReview.net.
- Zhe Li, Zhiwei Lin, and Yongtao Wang. 2026. Collmnas: Collaborative large language models for efficient knowledge-guided neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3273–3282.
- Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: differentiable architecture search. In *ICLR (Poster)*. OpenReview.net.
- Zhen Liu, Yuhan Liu, Jinjun Wang, Wei Song, Jianyi Liu, and Jingwen Fu. 2026. [Structured progressive knowledge activation for LLM-driven neural architecture search](#). In *Forty-third International Conference on Machine Learning*.
- Tom Mitchell. 1997. Twenty Newsgroups. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5C323>.
- Kenta Nakai. 1991. Yeast. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5KG68>.
- Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher W. Cleghorn. 2024. Llmatic: Neural architecture search via large language models and quality diversity optimization. In *GECCO*. ACM.
- Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. In *ICML, Proceedings of Machine Learning Research*, pages 4092–4101. PMLR.
- Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI*, pages 4780–4789. AAAI Press.
- Asankhaya Sharma. 2025. [Openevolve: an open-source evolutionary coding agent](#).
- Arjun Sridhar and Yiran Chen. 2025. Delta-nas: Difference of architecture encoding for predictor-based evolutionary neural architecture search. In *WACV*, pages 7857–7865. IEEE.
- Mark Towers, Ariel Kwiatkowski, Jordan K. Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. 2024. Gymnasium: A standard interface for reinforcement learning environments. *CoRR*, abs/2407.17032.

Ilya Trofimov, Nikita Klyuchnikov, Mikhail Salnikov, Alexander Filippov, and Evgeny Burnaev. 2023. Multi-fidelity neural architecture search with knowledge distillation. *IEEE Access*, 11:59217–59225.

Petar Velickovic, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. 2022. The CLRS algorithmic reasoning benchmark. In *ICML, Proceedings of Machine Learning Research*, pages 22084–22102. PMLR.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. 2023. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*.

William H. Wolberg, Olvi L. Mangasarian, Nick Street, and W. Nick Street. 1993. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5DW2B>.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. Nasbench-101: Towards reproducible neural architecture search. In *International conference on machine learning*, pages 7105–7114. PMLR.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*.

Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. 2023. Can GPT-4 perform neural architecture search? *CoRR*, abs/2304.10970.

Xun Zhou, Xingyu Wu, Liang Feng, Zhichao Lu, and Kay Chen Tan. 2025. Design principle transfer in neural architecture search via large language models. In *AAAI*, pages 23000–23008. AAAI Press.

Barret Zoph and Quoc V. Le. 2017. Neural architecture search with reinforcement learning. In *ICLR*. OpenReview.net.

6 Appendix

A Additional Details of NB101-SemPair

We generate 9.4M candidate pair proposals with five complementary policies: random, score-near, score-far, code-near, code-far, as summarized in Table 11.

B Hyperparameter Details

Table 13 reports the default hyperparameter values used by ECO-SCE. The margin m , PCA dimension d , and KNN size K are fixed design parameters used to define the transition representation and local evidence bank. The remaining parameters, τ_{bad} , τ_{good} , τ_{protect} , and λ , control the risk-aware routing policy. We select the reported values on the validation split to obtain a balanced trade-off between evaluation reduction and missed-improvement risk. Sensitivity analysis in Section 4.5 further shows that ECO-SCE is not overly sensitive to a narrow hyperparameter setting.

Hyperparameter selection. ECO-SCE has two coupled goals: reducing full-evaluator calls and avoiding unsafe routing decisions. Therefore, we do not select hyperparameters by maximizing Proxy Reduction alone. Instead, we first define a safety-feasible set:

$$\begin{aligned} \mathcal{F} = \{x \mid & \text{MissedImp}(x) \leq \epsilon_m, \\ & \text{AcceptBad}(x) \leq \epsilon_b, \\ & \text{Top10Kept}(x) \geq \epsilon_t\}. \end{aligned} \quad (14)$$

We set $\epsilon_m = 10\%$, $\epsilon_b = 10\%$, and $\epsilon_t = 90\%$. Thus, a setting is considered safe only if it keeps missed improvements and bad accepts below 10% while preserving at least 90% of the top-10 high-gain proposals.

Among safe settings, we select from the high-reduction plateau. Let $R(x)$ denote Proxy Reduction and

$$R_{\max} = \max_{x \in \mathcal{F}} R(x). \quad (15)$$

The plateau is defined as

$$\mathcal{P} = \{x \in \mathcal{F} \mid R(x) \geq R_{\max} - \Delta_R\}, \quad (16)$$

where Δ_R is the median nonzero change between adjacent points on the sensitivity curve. This makes the plateau tolerance adaptive to each hyperparameter sweep.

We then choose a representative point from $\mathcal{P}$ rather than always taking the most aggressive endpoint. This procedure, referred to as the Feasible Plateau Representative Score (FPRS), favors settings that are both safe and close to the best feasible reduction. Using FPRS, we select

$$(\tau_{\text{bad}}, \tau_{\text{good}}, \tau_{\text{protect}}, \lambda) = (0.40, 0.40, 0.40, 0.25). \quad (17)$$

This configuration achieves 75.02% Proxy Reduction with 8.98% Missed Imp., 9.61% Accept Bad, 90.39% Accept Prec., 88.19% Reject NPV, and 94.80% Top-10 Kept.

C Runtime and Cost Accounting

We separate the computational cost of ECO-SCE into **one-time offline preparation** and **online evolution**. Offline preparation includes NB101-SemPair construction, embedding computation, ECO-SCE training, and PCA/KNN-bank construction. These components are built once and reused unchanged across downstream tasks. In contrast, online evolution includes proposal generation, ECO-SCE screening, and full evaluator execution. Accordingly, all GPU hours reported in our downstream experiments measure the end-to-end online evolution runtime on a single NVIDIA RTX 3090 GPU and already include ECO-SCE overhead.

Table 10 summarizes the cost breakdown. For the online components, we profile the MNIST1D-shuffle CNN experiment as a representative case. ECO-SCE requires only 126.17 ms per proposal on average. Including a one-time initialization cost of 2.57 s, its total overhead over 100 evolution rounds is 15.23 s. The complete evolution run takes 0.46 GPU hours, indicating that ECO-SCE contributes only a small fraction of the total online runtime.

D Case study

Qualitative analysis. We further inspect the best programs found on the CLRS-DFS setting. The initial program and final best program proposed by OpenEvolve and OpenEvolve + ECO-SCE can be seen in the following subsections. This example is consistent with the design of ECO-SCE’s transition representation. ECO-SCE does not judge a proposal only by its standalone code, but by the semantic and structural difference between the proposal and its parent. The se-

lected ECO-SCE architecture preserves the original message-passing backbone and introduces a localized graph-conditioned gate in the triplet branch. Such an edit corresponds to a small but semantically meaningful transition: it adds adaptive control to high-order messages without disrupting the main aggregation path. In contrast, the vanilla OpenEvolve solution performs a much larger rewrite of the triplet branch, adding multiple coupled mechanisms such as multi-head projections, edge gates, direction gates, adjacency refinement, and DFS-specific gating. Although expressive, this type of entangled edit is harder to validate under limited evaluator budget and may be less stable.

This behavior is also aligned with the construction of NB101-SemPair. Because the training pairs cover both code-near and code-far transitions with positive and negative score changes, ECO-SCE can learn which semantic/code differences tend to correspond to useful parent-relative gains. Thus, the model is not simply biased toward small edits; rather, it learns to favor transitions whose semantic and structural changes resemble beneficial evolution patterns observed in the training data.

D.1 Initial Program

Listing 1 shows the original triplet branch used by the initial processor. We omit the shared first-order message-passing layers, residual path, and optional gates because they are not the focus of this case study and are largely shared by the evolved programs. The highlighted lines indicate the baseline high-order triplet projection and message construction.

```

1 class PGN(Processor):
2     # ... shared constructor and first-order message-passing
3     # ... backbone ...
4     def _maybe_lazy_init(self, z, edge_fts, graph_fts):
5         if self._lazy:
6             in_z, in_e, in_g = z.shape[-1], edge_fts.shape[-1],
7             graph_fts.shape[-1]
8         # Original high-order triplet projections.
9         if self.use_triplets:
10            self.t_1 = nn.Linear(in_z, self.nb_triplet_fts)
11            self.t_2 = nn.Linear(in_z, self.nb_triplet_fts)
12            self.t_3 = nn.Linear(in_z, self.nb_triplet_fts)
13            self.t_e_1 = nn.Linear(in_e, self.nb_triplet_fts)
14            self.t_e_2 = nn.Linear(in_e, self.nb_triplet_fts)
15            self.t_e_3 = nn.Linear(in_e, self.nb_triplet_fts)
16            self.t_g = nn.Linear(in_g, self.nb_triplet_fts)
17            self.o3 = nn.Linear(self.nb_triplet_fts, self.
18            out_size)
19        # ... unchanged first-order layers, normalization,
20        # ... and gates ...
21        def _compute_tri_msgs(self, z, edge_fts, graph_fts):
22            if not self.use_triplets:
23                return None
24            tri_1, tri_2, tri_3 = self.t_1(z), self.t_2(z), self.
25            t_3(z)
26            tri_e_1 = self.t_e_1(edge_fts)
27            tri_e_2 = self.t_e_2(edge_fts)

```

Value	Proxy Red.	Missed Imp.	Accept Bad	Accept Prec.	Reject NPV	Top-10 Kept	Safe	Plateau	FPRS
0.30	75.39	9.24	9.61	90.39	87.96	94.80	Yes	Yes	0.60
0.35	75.30	9.16	9.61	90.39	88.04	94.80	Yes	Yes	0.75
0.40	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	1.00
0.45	74.05	8.28	9.61	90.39	88.82	95.80	Yes	Yes	0.75
0.50	71.00	6.72	9.61	90.39	90.11	96.00	Yes	No	0.00
0.55	67.68	5.08	9.61	90.39	91.72	97.00	Yes	No	0.00
0.60	66.24	4.50	9.61	90.39	92.30	97.20	Yes	No	0.00
0.65	62.49	3.12	9.61	90.39	93.88	97.80	Yes	No	0.00
0.70	61.92	2.92	9.61	90.39	94.14	97.80	Yes	No	0.00
0.75	59.33	2.20	9.61	90.39	95.07	98.20	Yes	No	0.00
0.80	56.73	1.58	9.61	90.39	95.99	98.40	Yes	No	0.00
0.85	52.30	0.84	9.61	90.39	97.25	98.80	Yes	No	0.00
0.90	47.43	0.42	9.61	90.39	97.98	99.40	Yes	No	0.00

Table 6: Sensitivity results and FPRS values for τ_{bad} under the 10,000-training-pair setting. All percentage metrics are reported in %. Unsafe configurations are assigned an FPRS of -1 , feasible but non-plateau configurations are assigned 0, and the selected plateau representative is assigned 1.

Value	Proxy Red.	Missed Imp.	Accept Bad	Accept Prec.	Reject NPV	Top-10 Kept	Safe	Plateau	FPRS
0.30	75.50	8.98	9.94	90.06	88.19	94.80	Yes	Yes	0.60
0.35	75.29	8.98	9.89	90.11	88.19	94.80	Yes	Yes	0.75
0.40	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	1.00
0.45	73.51	8.98	8.42	91.58	88.19	94.80	Yes	Yes	0.75
0.50	71.18	8.98	6.93	93.07	88.19	94.80	Yes	No	0.00
0.55	69.06	8.98	5.37	94.63	88.19	94.80	Yes	No	0.00
0.60	67.48	8.98	4.64	95.36	88.19	94.80	Yes	No	0.00
0.65	65.37	8.98	3.89	96.11	88.19	94.80	Yes	No	0.00
0.70	62.39	8.98	2.69	97.31	88.19	94.80	Yes	No	0.00
0.75	59.36	8.98	1.69	98.31	88.19	94.80	Yes	No	0.00
0.80	57.18	8.98	1.44	98.56	88.19	94.80	Yes	No	0.00
0.85	54.52	8.98	0.62	99.38	88.19	94.80	Yes	No	0.00
0.90	50.87	8.98	0.23	99.77	88.19	94.80	Yes	No	0.00
0.95	45.96	8.98	0.00	100.00	88.19	94.80	Yes	No	0.00

Table 7: Sensitivity results and FPRS values for τ_{good} under the 10,000-training-pair setting.

```

28 tri_e_3 = self.t_e_3(edge_fts)
29 tri_g = self.t_g(graph_fts)
30
31 # Original triplet message construction.
32 triplets = (
33     tri_1.unsqueeze(2).unsqueeze(3)
34     + tri_2.unsqueeze(1).unsqueeze(3)
35     + tri_3.unsqueeze(1).unsqueeze(2)
36     + tri_e_1.unsqueeze(3)
37     + tri_e_2.unsqueeze(2)
38     + tri_e_3.unsqueeze(1)
39     + tri_g.unsqueeze(1).unsqueeze(2).unsqueeze(3)
40 )
41 tri_msgs = self.o3(torch.amax(triplets, dim=1))
42 return self.activation(tri_msgs) if self.activation
    else tri_msgs

```

Listing 1: Initial program: original triplet branch

D.2 ECO-SCE Best Program

Listing 2 highlights the localized change selected by ECO-SCE. The original triplet construction is retained, and only a graph-conditioned gate is added to modulate high-order triplet messages.

```

1 class PGN(Processor):

```

```

2 # ... same constructor and shared backbone as Listing 1
3 ...
4 def maybe_lazy_init(self, z, edge_fts, graph_fts):
5     if self.lazy:
6         in_z = z.shape[-1]
7         in_e = edge_fts.shape[-1]
8         in_g = graph_fts.shape[-1]
9         # ... unchanged first-order layers ...
10
11     if self.use_triplets:
12         # Original triplet branch is preserved.
13         self.t_1 = nn.Linear(in_z, self.nb_triplet_fts)
14         self.t_2 = nn.Linear(in_z, self.nb_triplet_fts)
15         self.t_3 = nn.Linear(in_z, self.nb_triplet_fts)
16         self.t_e_1 = nn.Linear(in_e, self.nb_triplet_fts)
17         self.t_e_2 = nn.Linear(in_e, self.nb_triplet_fts)
18         self.t_e_3 = nn.Linear(in_e, self.nb_triplet_fts)
19         self.t_g = nn.Linear(in_g, self.nb_triplet_fts)
20         self.o3 = nn.Linear(self.nb_triplet_fts, self.
21                               out_size)
22
23     # Added: graph-conditioned triplet gate.
24     self.triplet_gate_proj = nn.Linear(in_g, self.
25                                         nb_triplet_fts)
26     self.triplet_gate_sigmoid = nn.Sigmoid()
27
28 def compute_triplets(self, z, edge_fts, graph_fts):
29     if not self.use_triplets:
30         return None
31     # ... unchanged computation of tri_1, tri_2, tri_3,
32     # tri_e_1, tri_e_2, tri_e_3, and tri_g ...
33
34 # Added: use graph context to modulate high-order

```

Value	Proxy Red.	Missed Imp.	Accept Bad	Accept Prec.	Reject NPV	Top-10 Kept	Safe	Plateau	FPRS
0.05	59.25	2.20	9.61	90.39	95.05	98.80	Yes	No	0.00
0.10	67.80	5.08	9.61	90.39	91.75	97.40	Yes	No	0.00
0.15	73.00	7.76	9.61	90.39	89.22	96.00	Yes	No	0.00
0.20	74.19	8.42	9.61	90.39	88.68	95.60	Yes	Yes	0.80
0.25	74.89	8.94	9.61	90.39	88.20	94.80	Yes	Yes	0.84
0.30	74.97	8.98	9.61	90.39	88.17	94.80	Yes	Yes	0.89
0.35	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.94
0.40	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	1.00
0.45	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.94
0.50	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.89
0.55	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.84
0.60	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.80
0.65	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.76
0.70	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.73
0.75	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.70
0.80	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.67
0.85	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.64
0.90	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.62
0.95	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.59
1.00	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	0.57

Table 8: Sensitivity results and FPRS values for $\tau_{protect}$ under the 10,000-training-pair setting.

```

33 messages.
34 triplet_gate = self.triplet_gate_sigmoid(
35     self.triplet_gate_proj(graph_fts)
36 )
37 triplet_gate = triplet_gate.unsqueeze(1).unsqueeze(2)
38
39 triplets = (
40     tri_1.unsqueeze(2).unsqueeze(3)
41     + tri_2.unsqueeze(1).unsqueeze(3)
42     + tri_3.unsqueeze(1).unsqueeze(2)
43     + tri_e_1.unsqueeze(3)
44     + tri_e_2.unsqueeze(2)
45     + tri_e_3.unsqueeze(1)
46     + tri_g.unsqueeze(1).unsqueeze(2).unsqueeze(3)
47 )
48 # Added: localized modulation; construction remains
49 # unchanged.
50 triplets = triplets * triplet_gate.unsqueeze(3)
51 tri_msgs = self.o3(torch.amax(triplets, dim=1))
52 return self.activation(tri_msgs) if self.activation
53 else tri_msgs
54
55 def forward(self, node_fts, edge_fts, graph_fts, adj_mat
56 , hidden, **unused):
57     # ... unchanged first-order aggregation ...
58     tri_msgs = self._compute_tri_msgs(z, edge_fts,
59     graph_fts)
60     return ret, tri_msgs

```

Listing 2: ECO-SCE best program: localized graph-conditioned triplet gate

D.3 OpenEvolve Best Program

Listing 3 shows the larger rewrite found by vanilla OpenEvolve. Compared with ECO-SCE, this program changes the triplet branch more aggressively and introduces multiple coupled mechanisms simultaneously.

```

1 class PGN(Processor):
2     # ... same constructor and shared backbone as Listing 1
3     ...
4     def _maybe_lazy_init(self, z, edge_fts, graph_fts):
5         if self._lazy:
6             in_z = z.shape[-1]
7             in_e = edge_fts.shape[-1]
8             in_g = graph_fts.shape[-1]
9             # ... unchanged first-order layers ...

```

```

10 if self.use_triplets:
11     # Rewritten: multi-head triplet branch.
12     nb_heads = 4
13     self.nb_heads = nb_heads
14     self.head_dim = self.nb_triplet_fts // nb_heads
15     self.head_source = nn.Linear(in_z, nb_heads * self
16     .head_dim)
17     self.head_target = nn.Linear(in_z, nb_heads * self
18     .head_dim)
19     self.head_global = nn.Linear(in_g, nb_heads * self
20     .head_dim)
21     self.o3 = nn.Linear(self.nb_triplet_fts, self.
22     out_size)
23
24     # Rewritten: multiple coupled gates and
25     # refinements.
26     self.edge_gate = nn.Linear(2 * in_z + in_e,
27     nb_heads)
28     self.dir_gate = nn.Linear(self.head_dim, self.
29     head_dim)
30     self.adj_refine = nn.Linear(in_e, nb_heads * self.
31     head_dim)
32     self.dfs_gate = nn.Sequential(
33         nn.Linear(in_z, self.out_size), nn.Sigmoid()
34     )
35
36 def _compute_tri_msgs(self, z, edge_fts, graph_fts,
37 adj_mat):
38     if not self.use_triplets:
39         return None
40
41     b, n, _ = z.shape
42     source = self.head_source(z).view(b, n, self.nb_heads,
43     self.head_dim)
44     target = self.head_target(z).view(b, n, self.nb_heads,
45     self.head_dim)
46     z_i = z.unsqueeze(2).expand(-1, -1, n, -1)
47     z_j = z.unsqueeze(1).expand(-1, n, -1, -1)
48
49     # Rewritten: direction-aware pairwise interaction.
50     gated_target = torch.sigmoid(
51         self.dir_gate(target.view(-1, self.head_dim))
52     ).view(b, n, self.nb_heads, self.head_dim)
53     tri_head = source.unsqueeze(2) * gated_target.
54     unsqueeze(1)
55
56     # Rewritten: edge-conditioned gating and global
57     # injection.
58     edge_gate = torch.sigmoid(
59         self.edge_gate(torch.cat([edge_fts, z_i, z_j], dim
60     =-1)))
61     .unsqueeze(-1)
62     global_feat = self.head_global(graph_fts).view(
63     b, 1, 1, self.nb_heads, self.head_dim)
64
65     tri_head = (tri_head + global_feat) * edge_gate
66     # Rewritten: adjacency refinement and DFS-specific

```

Value	Proxy Red.	Missed Imp.	Accept Bad	Accept Prec.	Reject NPV	Top-10 Kept	Safe	Plateau	FPRS
0.10	85.08	11.30	12.26	87.74	86.71	93.80	No	No	-1.00
0.15	82.73	10.80	11.59	88.41	87.02	94.00	No	No	-1.00
0.20	79.20	9.94	10.62	89.38	87.53	94.40	No	No	-1.00
0.25	75.02	8.98	9.61	90.39	88.19	94.80	Yes	Yes	1.00
0.30	70.82	8.08	8.11	91.89	88.74	95.20	Yes	Yes	0.50
0.35	66.72	7.24	7.12	92.88	89.24	95.20	Yes	No	0.00
0.40	62.82	6.46	6.42	93.58	89.75	95.40	Yes	No	0.00
0.45	58.93	5.74	5.73	94.27	90.24	95.80	Yes	No	0.00
0.50	55.20	5.00	5.12	94.88	90.85	96.60	Yes	No	0.00
0.60	48.19	3.74	4.35	95.65	91.98	97.20	Yes	No	0.00
0.70	41.57	2.72	3.84	96.16	93.05	97.60	Yes	No	0.00
0.80	35.72	2.08	3.14	96.86	93.58	98.00	Yes	No	0.00
0.90	30.29	1.76	3.10	96.90	93.23	98.20	Yes	No	0.00
1.00	25.62	1.18	2.60	97.40	94.22	98.40	Yes	No	0.00
1.25	16.41	0.48	1.74	98.26	95.01	99.00	Yes	No	0.00
1.50	11.29	0.18	1.32	98.68	95.50	99.40	Yes	No	0.00

Table 9: Sensitivity results and FPRS values for λ under the 10,000-training-pair setting.

Table 10: Computational cost of ECO-SCE. Offline components are constructed once and reused across downstream tasks. Online costs are profiled on the MNIST1D-shuffle CNN experiment.

Stage	Component	Cost
Offline	NB101-SemPair construction	~10 h
	Embedding computation	19,460 codes, ~0.58 h
	ECO-SCE training	CPU only
	PCA and KNN-bank construction	One-time, reused
Online	ECO-SCE inference	126.17 ms / proposal
	Initialization	2.57 s / run
	Total ECO-SCE overhead	15.23 s / 100 rounds

Source	Attempts	Role
Random	3.0M	Global diversity
Score-near	0.6M	Small score differences
Score-far	3.0M	Clear score changes
Code-near	0.6M	Local code changes
Code-far	2.2M	Larger structural changes

Table 11: Candidate pair proposal policies for constructing NB101-SemPair.

```

53     masking.
54     adj_refine = F.gelu(
55         self.adj_refine(edge_fts).view(b, n, n, self.
56             nb_heads, self.head_dim)
57     )
58     tri_head = tri_head + adj_refine * adj_mat.unsqueeze
59     (-1).unsqueeze(-1)
60     tri_msgs = self.o3(tri_head.view(b, n, n, self.
61         nb_triplet_fts))
62     tri_msgs = tri_msgs * self.dfs_gate(z_i - z_j)
63     tri_msgs = tri_msgs * adj_mat.unsqueeze(-1)
64     return self.activation(tri_msgs) if self.activation
        else tri_msgs
def forward(self, node_fts, edge_fts, graph_fts, adj_mat
    , hidden, **unused):
    # ... unchanged first-order aggregation ...
    tri_msgs = self._compute_tri_msgs(z, edge_fts,
        graph_fts, adj_mat)
    return ret, tri_msgs

```

Listing 3: OpenEvolve best program: entangled multi-head triplet rewrite

E Details of Rule-Based Filtering Baselines

We include rule-based structural filters as non-learning baselines to test whether simple hand-designed heuristics can replace transition-level modeling. Each heuristic assigns a risk score to a parent-to-proposal transition using one structural or semantic-change statistic. Under the controlled filtering setting, the heuristic rejects the same number of proposals as ECO-SCE, so performance differences reflect proposal-selection quality rather than filtering budget.

The intuition behind these rules is as follows. `param_decrease` assumes that a large reduction in parameter count may weaken model capacity, while `param_increase` assumes that a large parameter increase may introduce inefficient or over-complex architectures. `diff_replaced_lines_large` treats extensive code replacement as a sign of unstable architecture modification. `aux_complexity_large` aggregates token-, AST-, line-, and call-level changes and rejects transitions with the largest overall structural shift. `semantic_l2_large` uses the dense semantic embedding distance between parent and proposal as a proxy for large semantic deviation. These heuristics capture intuitive risk factors, but they do not learn how different semantic and structural changes jointly relate to parent-relative evolutionary gain.

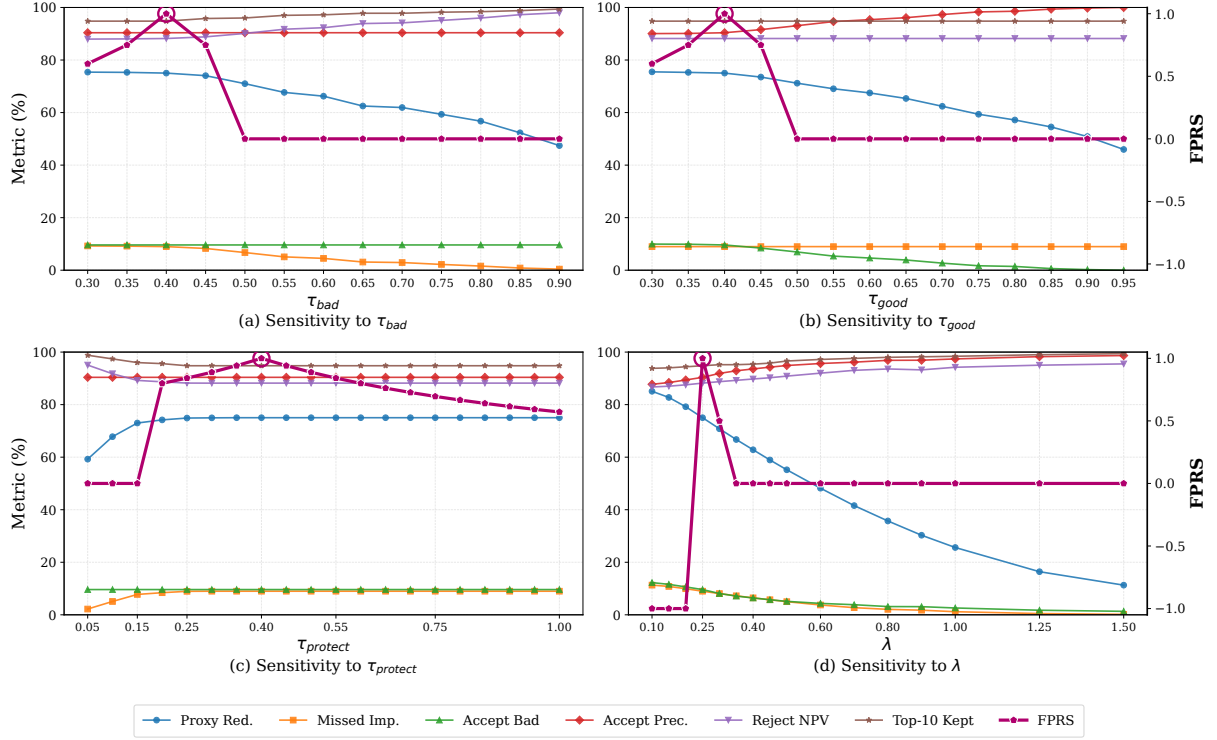


Figure 3: Sensitivity of ECO-SCE to routing hyperparameters on NB101-SemPair. We vary τ_{bad} , τ_{good} , $\tau_{protect}$, and λ , and report Proxy Reduction, Missed Improvement, Accept Bad, Accept Precision, Reject NPV, Top-10 Kept, and FPRS. FPRS selects safety-feasible high-reduction settings by balancing evaluation reduction with routing reliability. The circled points indicate the selected default hyperparameters. Detailed results are shown in Table 6, 7, 8 and 9.

Heuristic	Category	Rejection criterion
param_decrease	Parameter change	Largest parameter decrease
diff_replaced_lines_large	Code diff	Largest number of replaced lines
aux_complexity_large	Structural change	Largest aggregated structural-change score
semantic_l2_large	Dense semantic shift	Largest dense-embedding distance
param_increase	Parameter change	Largest parameter increase

Table 12: Rule-based filtering heuristics. Each rule ranks candidate transitions by a hand-designed risk signal and rejects the highest-risk candidates under the same filtering budget as ECO-SCE.

F Ablation Study: Which Components and Features Matter?

We perform both component-level and feature-level ablations on the MNIST1D-shuffle CNN setting inside OpenEvolve. The component-level study examines how the boundary classifier, magnitude regressor, and evidence bank contribute to evaluator allocation and search quality. The boundary classifier provides directional evidence, the magnitude regressor estimates the strength of performance changes, and the evidence bank calibrates decisions using local nearest-neighbor evidence. All variants use the same 100-iteration evolution budget.

Parameter	Value	Role
m	0.02	Boundary margin
d	64	PCA dimension
K	80	KNN neighbors
τ_{bad}	0.4	Auto-reject threshold
τ_{good}	0.4	Auto-accept threshold
$\tau_{protect}$	0.4	Reject protection
λ	0.25	Uncertainty weight

Table 13: ECO-SCE hyperparameters for the balanced routing policy.

Table 15 shows that all three components are useful but play different roles. Removing the boundary classifier causes the largest drop in search quality, reducing the best accuracy from

Dataset	Protocol	GPU Hours ↓		Task Score ↑	
		OpenEvolve	+ECO-SCE	OpenEvolve	+ECO-SCE
CLRS	CLRS-protocol	6.01	1.74 ^{-71.01%}	62.69	86.10 ^{+23.41}
MNIST1D	Family-controlled	0.74	0.20 ^{-72.97%}	70.25	80.10 ^{+9.85}
MNIST1D-shuffle	Family-controlled	1.03	0.31 ^{-69.90%}	61.65	67.48 ^{+5.83}
20 Newsgroups	Random-start	0.29	0.13 ^{-55.17%}	57.10	57.91 ^{+0.81}
CartPole-v1	Random-start	0.73	0.09 ^{-87.67%}	377.6	500.0 ^{+122.4}
Yeast	Random-start	0.21	0.18 ^{-14.29%}	61.78	62.63 ^{+0.85}
Breast Cancer	Random-start	0.13	0.01 ^{-92.31%}	96.90	96.81 ^{-0.09}
Average	–	1.31	0.38 ^{-70.88%}	6/7 improved	

Table 14: Zero-shot deployment of ECO-SCE with OpenEvolve across downstream architecture evolution benchmarks. Superscripts denote relative GPU-hour reduction and absolute task-score change over OpenEvolve. Scores are accuracies for classification tasks, mean test return for CartPole-v1, and average task score for CLRS. MNIST1D-style tasks use LR/MLP/CNN/GRU controlled searches, CLRS uses the benchmark-defined initialization, and the remaining tasks use random-start evolution. The average row reports mean GPU hours across benchmarks; raw task scores are not averaged because their scales differ.

Method	Bound.	Mag.	Evid.	Passed	Full Eval.	Skip	GPU	Best Acc.
OpenEvolve	–	–	–	100	100	0	1.27	59.10
Full ECO-SCE	✓	✓	✓	93	10	83	0.46	72.40
w/o Evidence	✓	✓	×	85	26	41	1.30	71.40
w/o Magnitude	✓	×	✓	98	19	71	0.88	71.80
w/o Boundary	×	✓	✓	89	24	59	1.11	66.50

Table 15: Component ablation of ECO-SCE inside OpenEvolve on the MNIST1D-shuffle CNN setting. GPU hours are measured for the evolution run.

72.40% to 66.50%. This indicates that directional evidence is critical for distinguishing beneficial transitions from harmful ones. Without this module, the system can still skip some evaluations, but it becomes less reliable in preserving high-value proposals.

The evidence bank mainly contributes to evaluation saving. Without local evidence calibration, the number of full evaluator calls increases from 10 to 26 and GPU time rises from 0.46h to 1.30h, while the best accuracy remains relatively close to the full model. This suggests that local evidence is especially important for identifying regions where ECO-SCE can safely skip evaluation. The magnitude regressor also reduces evaluator usage: removing it increases full evaluations from 10 to 19 and GPU time from 0.46h to 0.88h, with only a small accuracy change. Overall, the full ECO-SCE achieves the best balance between search quality and evaluation cost, showing that direction prediction, magnitude estimation, and local evidence calibration are complementary.

We further isolate the contribution of transition representations by comparing semantic-only,

Table 16: Feature-level ablation of ECO-SCE using the same data split and estimator.

Representation	AUROC	Macro-F1
Semantic only	76.34%	51.05%
Structural only	82.34%	60.25%
Semantic + Structural	86.09%	66.13%

structural-only, and combined features under the same data split and estimator. As shown in Table 16, structural features provide a stronger standalone signal, achieving 82.34% AUROC and 60.25% Macro-F1, while semantic features alone achieve 76.34% and 51.05%, respectively. Combining both yields the best performance, reaching 86.09% AUROC and 66.13% Macro-F1. This indicates that semantic embeddings provide complementary information beyond explicit structural edits, improving AUROC by 3.75 points and Macro-F1 by 5.88 points over structural features alone.

G Hyperparameter Sensitivity

We further study the sensitivity of ECO-SCE to the routing hyperparameters on NB101-SemPair.

Specifically, we vary the reject threshold τ_{bad} , accept threshold τ_{good} , protection threshold τ_{protect} , and uncertainty weight λ , while keeping the other hyperparameters fixed. We report Proxy Reduction, Missed Improvement, Accept Bad, Accept Precision, Reject NPV, Top-10 Kept, and FPRS.

Figure 3 shows that ECO-SCE exhibits stable routing behavior across broad hyperparameter ranges. For τ_{bad} and τ_{good} , increasing the thresholds makes routing more conservative: Proxy Reduction decreases, while Missed Improvement and Accept Bad generally remain within the safety range. For τ_{protect} , larger values increase protection against mistakenly rejecting potentially beneficial transitions, with only moderate changes in the main reliability metrics. The uncertainty weight λ controls the strongest cost-risk trade-off: larger λ makes ECO-SCE more conservative, reducing Proxy Reduction but also lowering missed improvements and bad accepts. Across all sweeps, Top-10 Kept remains high and Accept Precision stays around or above 90% near the selected operating points, indicating that ECO-SCE preserves high-gain proposals while reducing evaluator usage. FPRS selects $(\tau_{\text{bad}}, \tau_{\text{good}}, \tau_{\text{protect}}, \lambda) = (0.40, 0.40, 0.40, 0.25)$ as the default setting, which lies in the safety-feasible high-reduction region for all four sensitivity curves.

H Additional Tests on RQ2

We examine whether adding ECO-SCE to an existing architecture evolution pipeline reduces evaluation cost while preserving search quality across diverse benchmarks. Specifically, we insert ECO-SCE into OpenEvolve and compare the original pipeline with its ECO-SCE-augmented variant. The evaluation covers seven benchmarks from diverse domains: CLRS (Velickovic et al., 2022) for algorithmic reasoning, MNIST1D and MNIST1D-shuffle for 1D sequence classification (Greydanus and Kobak, 2024), 20 Newsgroups for text classification (Mitchell, 1997), CartPole-v1 (Towers et al., 2024) for reinforcement-learning control, and Yeast (Nakai, 1991) and Breast Cancer Wisconsin Diagnostic (Wolberg et al., 1993) for scientific tabular classification. MNIST1D-style tasks use controlled searches over LR, MLP, CNN, and GRU architecture types. CLRS starts from the benchmark-defined initial architecture, while the remaining benchmarks use random-start evolu-

tion. We report GPU hours and task score, where task score denotes classification accuracy, mean test return for CartPole-v1, and average task score for CLRS.

Table 14 shows that ECO-SCE reduces GPU hours on all seven benchmarks. On average, GPU hours decrease from 1.31 to 0.38, corresponding to a 70.88% reduction. Task performance improves in six out of seven benchmarks, with only a negligible 0.09 drop on Breast Cancer. These results indicate that ECO-SCE can reduce downstream architecture evolution cost while maintaining or improving search quality.

In addition, the detailed performance of CLRS-based experiments is shown in Table 17.

Task	Triplet-GMPNN Params			OE Params	EP Params	SPARK Params	Ours Params	Triplet-GMPNN Acc.	OE Acc.	EP Acc.	SPARK Acc.	Ours Acc.
articulation_points	531913	563259	497969	534880	532945	88.32	61.32	93.46	97.90	98.60	98.03	98.60
activity_selector	262204	293550	262204	265171	263236	95.18	95.43	95.05	98.03	99.42	98.03	99.42
bellman_ford	524604	555950	568660	527571	525636	97.39	95.31	97.50	96.04	99.02	96.04	99.02
bfs	523963	555309	522931	526930	524995	99.73	100.00	99.99	70.56	100.00	70.56	100.00
binary_search	262204	293550	262204	265171	263236	77.58	39.31	77.98	87.60	90.67	87.60	90.67
bridges	532556	563902	576612	535523	533588	93.99	79.68	97.57	98.15	99.65	98.15	99.65
bubble_sort	524477	555823	568533	527444	525509	67.68	85.45	88.87	98.12	80.32	98.12	80.32
dag_shortest_paths	793287	824633	793287	796254	794319	98.19	96.97	98.01	97.80	99.76	97.80	99.76
dfs	661190	692536	660158	664157	662222	47.79	32.54	68.14	83.74	60.01	83.74	60.01
dijkstra	525886	557232	524854	528853	526918	96.05	95.85	97.30	99.22	98.73	99.22	98.73
find_maximum_subarray_kadane	264514	295860	261290	267481	265546	76.36	77.05	75.35	85.64	80.42	85.64	80.42
floyd_warshall	625089	656435	669145	628056	626121	48.52	11.24	61.43	32.91	30.17	61.43	32.91
graham_scan	398409	429755	397377	401376	399441	93.62	83.29	93.76	96.36	95.97	96.36	95.97
heapsort	659654	691000	703710	662621	660686	31.04	16.31	69.90	75.63	79.05	87.60	79.05
insertion_sort	524477	555823	523445	527444	525509	78.14	25.78	89.47	97.51	95.21	97.51	95.21
jarvis_march	264898	296244	308954	267865	265930	91.01	36.42	90.36	94.88	91.52	94.88	91.52
kmp_matcher	398021	729367	396989	400988	399053	19.51	10.09	16.29	14.94	64.16	14.94	64.16
les_length	270419	301765	270419	273386	271451	80.51	87.53	85.75	87.60	97.39	87.60	97.39
matrix_chain_order	624448	655794	624448	627415	625480	91.68	80.84	90.77	87.29	87.65	87.29	87.65
minimum	261307	292653	260275	264274	262339	97.78	98.10	98.40	99.07	98.83	99.07	98.83
mst_kruskal	399691	431017	443747	402658	400723	89.80	76.57	91.47	86.23	92.23	86.23	92.23
mst_prim	525886	557232	569942	528853	526918	86.39	83.98	88.74	88.67	92.38	88.67	92.38
naive_string_matcher	262588	293934	259364	265555	263620	78.67	8.93	79.77	77.27	99.95	77.27	99.95
optimal_bst	625987	357333	624955	628954	627019	73.77	71.73	78.66	96.14	84.87	96.14	84.87
quickselect	395714	427060	377130	398681	396746	0.47	3.71	0.79	10.69	13.72	10.69	13.72
quicksort	525759	557105	524727	528726	526791	64.64	16.26	85.23	97.80	91.41	97.80	91.41
segments_intersect	263359	294705	262327	266326	264391	97.64	96.85	98.15	97.07	97.98	97.07	97.98
strongly_connected_components	663243	694589	707299	666210	664275	43.43	45.21	41.86	77.34	87.84	77.34	87.84
task_scheduling	262333	293679	262333	265300	263365	87.25	82.68	88.23	87.44	90.21	87.44	90.21
topological_sort	660164	691510	660164	663131	661196	87.27	86.31	88.12	99.72	100.00	99.72	100.00
Average	467274.80	498620.13	478181.73	470241.80	485946.40	75.98	62.69	80.88	83.91	86.10	83.91	86.10

Table 17: Full per-task CLRS results. For each of the 30 CLRS tasks, we report model size and accuracy for Triplet-GMPNN, OpenEvolve (OE), EvoPrompting (EP), SPARK, and OpenEvolve+ECO-SCE.