SEAEVO: STRATEGY AS STATE FOR LLM-DRIVEN EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-driven evolutionary search is increasingly used for automated algorithm discovery and program optimization. However, most existing methods represent search progress primarily through executable programs and scalar fitness, while language-level strategies remain transient mutation context or unstructured memory. This makes it difficult to organize search progress across related algorithmic directions or reason about which directions are promising, saturated, or underexplored. We introduce SEAEVO, which treats *strategy as persistent population-level evolutionary state*. Each candidate is associated with an explicit strategy description, and these candidate-aligned strategies form a semantic space that supports complementary experience retrieval and population-level search guidance. SEAEVO operates as a lightweight layer over existing evolutionary backbones without modifying their selection or evaluation logic. Across mathematical and systems optimization benchmarks, SEAEVO improves the mean performance of its matched evolutionary backbone in 23 of 24 comparisons across two mutation models. By retrieving compact strategy descriptions rather than full source programs, SEAEVO reduces average LLM token usage by 48.8%. These results suggest that persistent strategy representations can provide a practical mechanism for improving the effectiveness and cost-efficiency of LLM-guided evolutionary search, pointing toward compound AI systems whose search capabilities benefit from the structured accumulation and reuse of algorithmic strategies.[1]

1 INTRODUCTION

LLM-driven evolutionary search has emerged as a promising approach for automated algorithm discovery and program optimization (Romera-Paredes et al., 2024; Novikov et al., 2025). By pairing LLMs as mutation operators with programmatic evaluators, recent systems iteratively generate, execute, and refine candidate programs, achieving strong results in domains with large search spaces and costly hand-crafted heuristics (Liu et al., 2024; Ye et al., 2024; Liu et al., 2025; 2026a).

Most existing LLM-driven evolutionary systems represent persistent search state primarily through executable programs and scalar fitness values (Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025). This representation is sufficient for execution, evaluation, and selection, but indirectly captures which algorithmic directions have been explored. Syntactically different programs may implement the same underlying algorithmic idea, while similarly scoring programs may correspond to qualitatively different approaches. Moreover, a currently weak candidate may represent a distinct direction that has not yet been sufficiently refined, while per-program fitness provides little visibility into whether some strategic directions have saturated while others remain underexplored.

Recent work has introduced richer language-level signals, including natural-language heuristic descriptions (Liu et al., 2024), reflection over failures (Shinn et al., 2023; Ye et al., 2024), and reusable language-level experience (Agrawal et al., 2026). These signals provide useful guidance beyond scalar fitness, but are typically used as mutation-time context, attached to individual candidates or trajectories, or accumulated as run-level memory, rather than maintained as a population-level representation of the strategies explored by the evolving population. Thus, existing systems may

[1]We release our code at `https://anonymous.4open.science/r/SeaEvo` to support reproducibility and facilitate future research.

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.

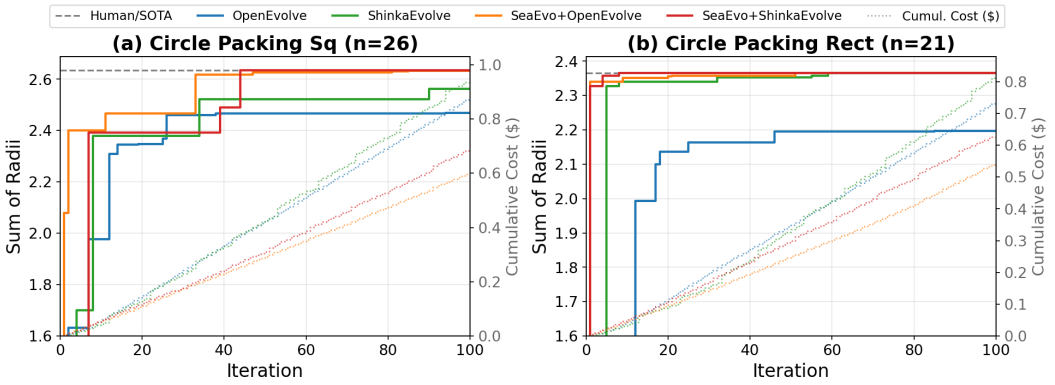


Figure 1: Search trajectories on Circle Packing benchmarks. Solid curves show the best-so-far sum of radii over generations, dotted curves show cumulative API cost, and the gray dashed line denotes the Human/SOTA reference. SEAEVO improves both OpenEvolve and ShinkaEvolve backbones, reaching competitive solutions earlier while incurring lower cumulative cost.

record *which implementations have been evaluated* without explicitly representing *which algorithmic strategies have been explored*. We refer to this mismatch as the *strategy-representation gap*.

To address this gap, we introduce **SEAEVO**, which treats *strategy as persistent population-level evolutionary state*. SEAEVO associates each candidate with an explicit natural-language strategy description and corresponding semantic representation. The resulting strategy-augmented state provides two complementary views of the evolving population: executable programs record what has been implemented and remain the source of truth for execution and evaluation, while strategy provides a compact abstraction for organizing and reasoning about which algorithmic directions have been explored. SEAEVO therefore augments, rather than replaces, program-space evolution with a persistent strategy-level view of search progress. We instantiate this idea through three mechanisms operating at complementary timescales. At the candidate level, *Strategy Articulation* records a concise description of the intended strategy during mutation, while *Stratified Experience Retrieval* organizes archived strategies semantically and retrieves experience from related and complementary regions of strategy space. At a slower population level, *Strategic Landscape Navigation* periodically reasons over accumulated strategy state to guide search toward promising or underexplored directions and away from saturated ones. These mechanisms form a lightweight strategy-state layer over existing evolutionary backbones without modifying their underlying selection or evaluation logic.

We evaluate SEAEVO across algorithm discovery, systems optimization, and agent-scaffold design benchmarks using multiple mutation LLMs and evolutionary frameworks. SEAEVO improves the corresponding backbones in most settings, including 23 of 24 matched comparisons across mathematical and systems optimization benchmarks. Representing retrieved experience through compact strategies rather than full programs also reduces average LLM token usage by 48.8%. Figure 1 shows that SEAEVO reaches competitive solutions in fewer generations while incurring lower cumulative API cost. Controlled correspondence-disruption experiments provide further evidence that preserving meaningful candidate–strategy organization is important to these gains.

Contributions. Our contributions are as follows: **1)** We formulate natural-language strategy as a persistent, candidate-aligned component of population-level evolutionary state, complementing executable programs and scalar fitness with an explicit representation of the algorithmic directions explored by the population. **2)** We introduce SEAEVO, a lightweight strategy-state layer that supports strategy-level experience retrieval and population-level navigation while remaining compatible with existing evolutionary backbones. **3)** We validate SEAEVO across algorithm discovery, systems optimization, and agent-scaffold design, including controlled attribution and efficiency analyses.

2 RELATED WORK

LLM-Driven Evolutionary Algorithm Discovery. Recent work frames algorithm and program improvement as an evolutionary loop in which LLMs act as proposal or mutation operators (Van Stein

& Bäck, 2024; Liu et al., 2026b). FunSearch (Romera-Paredes et al., 2024) established this paradigm by pairing an LLM with a programmatic evaluator, while AlphaEvolve (Novikov et al., 2025) scaled it to larger codebases using a MAP-Elites-inspired program database (Mouret & Clune, 2015). Subsequent systems improve search through adaptive parent sampling and novelty rejection (Lange et al., 2026), island-based evolution (Assumpção et al., 2025), hierarchical context management and backtracking (Yan et al., 2026), or adaptive budget and exploration control (Cemri et al., 2026). These methods primarily modify how programs are generated, selected, evaluated, or orchestrated. SEAEVO instead augments the executable program archive with persistent, candidate-aligned strategy state, making the representation of search history itself an explicit component of the evolutionary process.

Language-Level Reasoning, Reflection, and Memory. A closely related line of work introduces natural-language reasoning into optimization and evolutionary search (Yuksekgonul et al., 2025; Lee et al., 2025). EoH (Liu et al., 2024) evolves natural-language thoughts alongside code, while ReEvo (Ye et al., 2024), Reflexion (Shinn et al., 2023), and GEPA (Agrawal et al., 2026) use verbal feedback or reflection to improve future proposals. Other systems maintain reusable insights (Chen et al., 2025), abstractions (Wu et al., 2025), or evolution strategies (Yang et al., 2025b). PatchCat (Even-Mendoza et al., 2025) also semantically organizes generated edits, primarily to characterize patches and identify redundant changes, whereas SEAEVO uses candidate-aligned strategy organization to control retrieval and subsequent exploration. These works show that language can provide useful intermediate structure between scalar feedback and executable code. SEAEVO differs by treating strategy as persistent population-level state: candidate-aligned strategies are organized into semantic neighborhoods, retrieved as search experience, and periodically analyzed at the family level.

3 METHOD

3.1 STRATEGY-AUGMENTED EVOLUTIONARY STATE

We consider the problem of discovering a high-performing program $P \in \mathcal{P}$, where $\mathcal{P}$ denotes the space of candidate programs and $f : \mathcal{P} \to \mathbb{R}$ is the task-specific evaluation function. The objective is to find $P^* = \arg\max_{P \in \mathcal{P}} f(P)$.

At generation t, a conventional LLM-driven evolutionary framework maintains a program–fitness archive

$$\mathcal{A}_t^{\text{prog}} = \{(P_i, s_i)\}_{i=1}^{N_t}, \qquad s_i = f(P_i), \tag{1}$$

where P_i is an archived candidate, s_i is its fitness, and N_t is the archive size. The underlying evolutionary backbone selects parents from this archive, generates and evaluates new candidates, and updates the archive.

SEAEVO augments each candidate with a concise natural-language strategy description d_i and its semantic embedding $\mathbf{e}_i = \text{embed}(d_i)$. When the evaluator provides instance-level outcomes over a validation set $\mathcal{V}$, we additionally associate each candidate with a behavioral vector $\mathbf{b}_i \in \{0,1\}^{|\mathcal{V}|}$, where $b_i^k = 1$ indicates that P_i succeeds on validation instance k. The resulting *strategy-augmented evolutionary state* is

$$\mathcal{A}_t^{\text{strat}} = \{(P_i, s_i, d_i, \mathbf{e}_i, \mathbf{b}_i)\}_{i=1}^{N_t}, \tag{2}$$

with $\mathbf{b}_i$ omitted when only scalar feedback is available.

3.2 SEARCH DYNAMICS OVER STRATEGY STATE

Figure 2 summarizes how SEAEVO operates over the strategy-augmented state. At each generation, the underlying backbone first selects a parent P_{parent}, and SEAEVO retrieves a set of strategy-level inspirations $\mathcal{G}_t$. Conditioned on the parent, the retrieved inspirations, and the current population-level navigation state $\mathcal{L}_t$, the mutation model $\mathcal{M}_\theta$ generates both a new program and its intended strategy:

$$(P', d') \sim \mathcal{M}_\theta\left(P_{\text{parent}}, \mathcal{G}_t, \mathcal{L}_t\right). \tag{3}$$

After evaluation, d' is embedded and the resulting candidate is added to the strategy-augmented archive. This constitutes the *fast timescale* of candidate-level evolution.

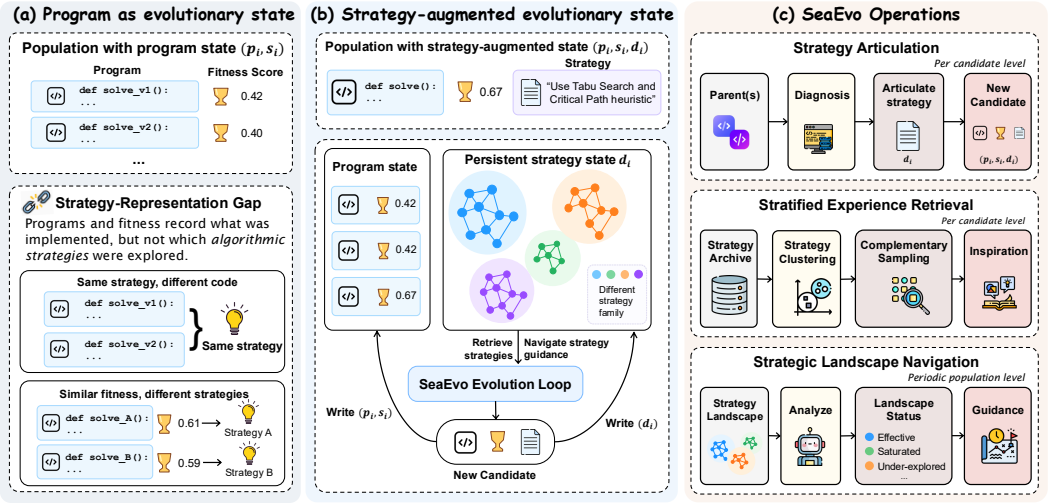


Figure 2: (a) Conventional evolutionary search maintains programs and fitness as persistent state, which does not explicitly capture whether different implementations share the same algorithmic strategy or whether similarly scoring programs follow different strategies. (b) SEAEVO augments each candidate with a persistent, candidate-aligned strategy representation, organizing the evolving population into a strategy-level state that supports retrieval and navigation. (c) Three complementary operations act on this state: Strategy Articulation records the intended strategy during candidate generation, Stratified Experience Retrieval retrieves complementary experience across strategy families, and Strategic Landscape Navigation periodically summarizes population-level strategy progress to guide subsequent search.

At the *slow timescale*, SEAEVO periodically updates the population-level navigation state. Every Δ generations,

$$\mathcal{L}_t = \begin{cases} \text{Navigate}(\mathcal{A}_t^{\text{strat}}), & t \bmod \Delta = 0, \\ \mathcal{L}_{t-1}, & \text{otherwise}, \end{cases} \tag{4}$$

so the same navigation state guides multiple candidate-level mutations between successive updates.

Before the archive contains enough candidates to form C strategy clusters, SEAEVO uses a warm-up retrieval rule instead of strategy-family retrieval. We additionally retain an ε-greedy exploration path: with probability ε, SEAEVO falls back to the backbone's default mutation pipeline; otherwise, it uses the full strategy-guided pipeline.

3.3 CANDIDATE-ALIGNED STRATEGY CONSTRUCTION

We instantiate candidate-aligned strategy construction through *Strategy Articulation* (SA), which jointly generates a candidate program and a concise description of its intended strategy. Given a selected parent P_{parent}, retrieved inspirations $\mathcal{G}_t$, and navigation guidance $\mathcal{L}_t$, SA produces (P', d') through a single structured LLM call following a *diagnose–direct–implement* sequence. The model first diagnoses limitations of the parent using feedback available from the underlying evolutionary framework, then articulates a strategy d' describing the intended algorithmic change, and finally generates P' according to that strategy. (P', d') is retained in the strategy-augmented state. Importantly, d' is *prospective*: it is articulated before implementation rather than assigned retrospectively as a semantic label to completed source code.

3.4 STRATEGY-SPACE RETRIEVAL

Stratified Experience Retrieval (SER) constructs mutation context by retrieving experience both from the parent's strategy family and from distinct strategy families, supporting within-family refinement and cross-family exploration.

Strategy-family organization. SER partitions the current archive into C strategy families by clustering the strategy embeddings:

$$\{\mathcal{S}_1, \ldots, \mathcal{S}_C\} \leftarrow k\text{-means}(\{\mathbf{e}_i\}, C). \tag{5}$$

The partition is recomputed whenever SER is invoked, allowing the family structure to evolve with the archive. Before the archive contains enough candidates to form C clusters, SER uses a warm-up rule that provides the parent, the global-best candidate, and an additional complementary candidate as mutation context.

Marginal behavioral coverage. When instance-level feedback is available, SER favors candidates that complement behaviors not already covered by the parent and previously selected inspirations. For a parent P and a set of selected inspirations $\mathcal{I}$, define the behavioral coverage function

$$F_P(\mathcal{I}) = \frac{1}{|\mathcal{V}|} \sum_{k=1}^{|\mathcal{V}|} \mathbf{1}\left[b_P^k = 0 \ \wedge \ \exists P_j \in \mathcal{I} : b_j^k = 1\right]. \tag{6}$$

$F_P(\mathcal{I})$ measures the fraction of validation instances that are unresolved by P but covered by at least one candidate in $\mathcal{I}$. We define the marginal coverage contributed by candidate P_i as

$$F_P(P_i \mid \mathcal{I}) \coloneqq F_P(\mathcal{I} \cup \{P_i\}) - F_P(\mathcal{I}). \tag{7}$$

This marginal gain discounts behavior already covered by the selected inspirations. When instance-level feedback is unavailable, SER instead uses scalar fitness as the candidate-selection criterion within the same strategy-family structure.

Stratified selection. Let $\mathcal{S}_{c^*}$ denote the strategy family containing P_{parent}. With instance-level feedback, SER first selects an intra-family inspiration

$$P^{\text{intra}} = \underset{P_i \in \mathcal{S}_{c^*} \setminus \{P_{\text{parent}}\}}{\arg\max} F_{P_{\text{parent}}}(P_i \mid \varnothing). \tag{8}$$

It then samples a different strategy family,

$$c' \sim \text{Uniform}\{c : c \neq c^*\}, \tag{9}$$

and selects a cross-family inspiration

$$P^{\text{cross}} = \underset{P_i \in \mathcal{S}_{c'}}{\arg\max} F_{P_{\text{parent}}}\left(P_i \mid \{P^{\text{intra}}\}\right). \tag{10}$$

For scalar-only tasks, the objectives in Eqs. 8–10 are replaced by s_i. The intra-family candidate supports refinement of the parent's current direction, while the cross-family candidate contributes experience from a different region of strategy space without duplicating behavior already covered by the intra-family selection. Sampling the cross-family cluster before selecting its candidate prevents a single high-performing family from repeatedly dominating the exploration slot.

For the selected inspirations, SER passes only their strategy descriptions and fitness values, rather than complete source code, to the mutation model. This aligns the retrieved context with the strategy representation used for retrieval while reducing mutation-context length.

3.5 POPULATION-LEVEL STRATEGY NAVIGATION

Strategic Landscape Navigation (SLN) instantiates the population-level operator $\text{Navigate}(\cdot)$ in Eq. 4. Given the accumulated strategy archive, SLN jointly considers strategy descriptions and their observed performance to produce a navigation state $\mathcal{L}_t$. Specifically, it summarizes strategic directions that are *effective* (associated with strong observed performance), *saturated* (repeatedly explored without corresponding performance gains), or *underexplored* (plausible directions that are sparsely represented in the archive), and synthesizes these observations into guidance for subsequent strategy proposals through SA.

Table 1: Performance on mathematical optimization benchmarks. Avg denotes mean score with standard deviation over 3 runs, and Best denotes the best score achieved. Best results within each backbone group are bolded, and full-precision near ties are reported in Table 21.

Method	Circle Packing (Rect)		Circle Packing (Square)		Heilbronn (Triangles)		MinMax Distance	
	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Backbone: MiMo-V2-Pro								
GEPA	$2.3398_{\pm 0.0197}$	2.3584	$2.5206_{\pm 0.0207}$	2.5414	$0.0265_{\pm 0.0010}$	0.0273	$0.2760_{\pm 0.0019}$	0.2776
OpenEvolve	$2.1380_{\pm 0.1987}$	2.3530	$2.4003_{\pm 0.0386}$	2.4390	$0.0250_{\pm 0.0035}$	0.0281	$0.2564_{\pm 0.0165}$	0.2760
ShinkaEvolve	$2.3560_{\pm 0.0083}$	2.3608	$2.4812_{\pm 0.0442}$	2.5123	$0.0272_{\pm 0.0001}$	0.0273	$0.2776_{\pm 0.0008}$	0.2783
SEAEVO$_{\text{OPEN}}$	$\mathbf{2.3607}_{\pm 0.0036}$	**2.3642**	$\mathbf{2.6045}_{\pm 0.0338}$	**2.6263**	$\mathbf{0.0314}_{\pm 0.0014}$	**0.0330**	$0.2778_{\pm 0.0003}$	0.2781
SEAEVO$_{\text{SHINKA}}$	$2.3603_{\pm 0.0033}$	2.3640	$2.5467_{\pm 0.1292}$	2.6237	$0.0274_{\pm 0.0013}$	0.0285	$\mathbf{0.2780}_{\pm 0.0003}$	**0.2784**
Backbone: Gemini-3-Flash								
GEPA	$2.3610_{\pm 0.0013}$	2.3621	$2.5306_{\pm 0.0380}$	2.5574	$0.0303_{\pm 0.0011}$	0.0316	$0.2785_{\pm 0.0000}$	**0.2785**
OpenEvolve	$2.1713_{\pm 0.0432}$	2.1968	$2.4522_{\pm 0.0387}$	2.4887	$0.0062_{\pm 0.0022}$	0.0087	$0.2765_{\pm 0.0007}$	0.2774
ShinkaEvolve	$2.3625_{\pm 0.0030}$	2.3658	$2.5204_{\pm 0.0357}$	2.5617	$0.0317_{\pm 0.0012}$	0.0331	$0.2783_{\pm 0.0001}$	0.2785
SEAEVO$_{\text{OPEN}}$	$\mathbf{2.3643}_{\pm 0.0026}$	2.3658	$2.5802_{\pm 0.0822}$	2.6317	$0.0305_{\pm 0.0003}$	0.0309	$0.2785_{\pm 0.0000}$	**0.2785**
SEAEVO$_{\text{SHINKA}}$	$2.3639_{\pm 0.0034}$	**2.3658**	$\mathbf{2.6297}_{\pm 0.0077}$	**2.6353**	$\mathbf{0.0335}_{\pm 0.0006}$	**0.0342**	$\mathbf{0.2785}_{\pm 0.0000}$	**0.2785**

We implement $\mathrm{Navigate}(\cdot)$ with an LLM that reasons over the accumulated strategy descriptions and their observed fitness. Unlike candidate-level reflection, SLN operates over the evolving population as a whole and is updated only periodically, providing a slow-timescale control signal for candidate-level search.

4 EXPERIMENTS

4.1 EXPERIMENT SETUP

Tasks and datasets. We evaluate on three task families: (i) four mathematical optimization tasks, including Circle Packing in Square, Circle Packing in Rectangle, Heilbronn Triangles, and MinMax Distance; (ii) four systems optimization tasks from the ADRS benchmark (Cheng et al., 2025): Prism, TXN, EPLB, and LLM-SQL; and (iii) agent-scaffold optimization tasks.

Baselines. We compare SEAEVO against state-of-the-art LLM-driven evolutionary methods, including GEPA (Agrawal et al., 2026), OpenEvolve (Sharma, 2025), and ShinkaEvolve (Lange et al., 2026). We instantiate two variants of our method: SEAEVO$_{\text{SHINKA}}$ over ShinkaEvolve, and SEAEVO$_{\text{OPEN}}$ over OpenEvolve; the corresponding backbone rows therefore serve as direct ablations of the strategy-aware modules. For readability, figures use the expanded form (*e.g.*, SEAEVO +ShinkaEvolve) while tables use the compact notation (*e.g.*, SEAEVO$_{\text{SHINKA}}$). We additionally apply SEAEVO to GEPA and to AdaEvolve (Cemri et al., 2026) in Appendix D to test plug-in generality.

Implementation details. We use two LLMs as the mutation models within the evolutionary loop: MiMo-V2-Pro (Xiaomi MiMo Team, 2026) and Gemini-3-Flash (Google DeepMind, 2025). For the agent-scaffold task, the inference model is Qwen3-8B (Yang et al., 2025a). Strategy embeddings use text-embedding-3-small model (OpenAI, 2024) . We build on the `SkyDiscover` framework (Liu et al., 2026c). Each configuration runs for $T = 100$ generations with SLN update interval $\Delta = 10$, $C = 5$ strategy clusters, and ε-greedy ratio $\varepsilon = 0.2$. We use the default hyperparameters for all baseline methods. Similar to Cemri et al. (2026), each experiment is repeated three times; we report mean and standard deviation. More details are provided in Appendix C.

4.2 MAIN RESULTS

Tables 1 and 2 report results on mathematical and systems optimization benchmarks, respectively. Figures 3 and 6 report agent-scaffold results, and Figure 1 shows search trajectories on the two Circle Packing benchmarks. We highlight several observations

Consistent improvements across tasks and backbones. Across the 24 matched backbone comparisons in Tables 1 and 2, SEAEVO improves mean fitness in 23 cases. Improvements are observed with both MiMo-V2-Pro and Gemini-3-Flash and for both OpenEvolve and ShinkaEvolve. The same pattern holds on the agent-scaffold tasks, where SEAEVO improves both corresponding backbones

Table 2: Performance on systems optimization benchmarks with Gemini-3-Flash. Avg denotes mean score with standard deviation over runs, and Best denotes the best score achieved. The Human/SOTA row reports reference best scores when available. Best method results are highlighted in boldface.

Method	Prism		TXN		EPLB		LLM-SQL	
	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human/SOTA	–	21.89	–	2725	–	0.1265	–	0.692
GEPA	$23.6733_{\pm 0.6034}$	24.0217	$3266.0_{\pm 243.1}$	3496.5	$0.1472_{\pm 0.0062}$	0.1543	$0.7158_{\pm 0.0053}$	0.7233
OpenEvolve	$24.0217_{\pm 0.0000}$	24.0217	$3586.7_{\pm 188.4}$	3745.3	$0.1260_{\pm 0.0002}$	0.1262	$0.7117_{\pm 0.0105}$	0.7241
ShinkaEvolve	$26.2672_{\pm 0.2757}$	26.5369	$3480.7_{\pm 48.6}$	3508.8	$0.1326_{\pm 0.0111}$	0.1454	$0.6998_{\pm 0.0165}$	0.7181
SEAEVO$_{\text{OPEN}}$	$26.1783_{\pm 0.1031}$	26.2783	$3810.6_{\pm 416.2}$	4098.4	$0.1535_{\pm 0.0158}$	0.1716	$0.7104_{\pm 0.0010}$	0.7114
SEAEVO$_{\text{SHINKA}}$	$\mathbf{43.6149}_{\pm 29.7823}$	**78.0040**	$\mathbf{3886.1}_{\pm 517.9}$	**4219.4**	$\mathbf{0.1958}_{\pm 0.0002}$	**0.1960**	$\mathbf{0.7200}_{\pm 0.0087}$	**0.7318**

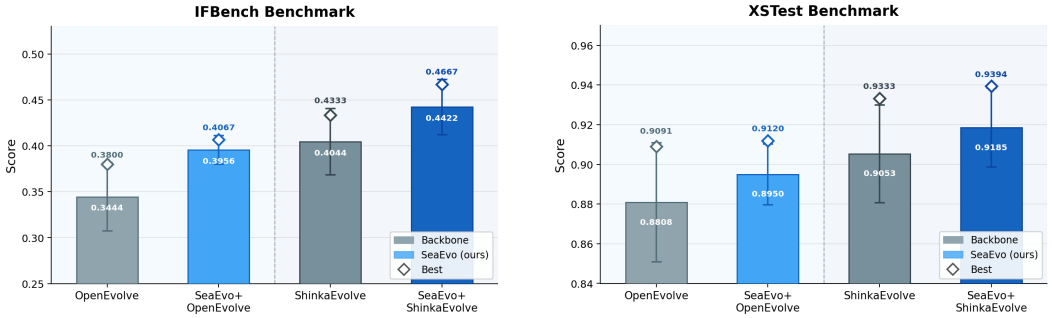


Figure 3: Performance on the agent scaffold design benchmark with maximum one LLM call. Bars show average test score with standard deviation across runs, and diamonds indicate the best score.

(Figures 3 and 6). This consistency across task families, mutation models, and evolutionary backbones supports the intended role of SEAEVO as a general strategy-state layer rather than a mechanism tied to a particular search implementation.

Effect sizes vary across tasks. The largest gains occur on several systems optimization benchmarks, particularly TXN and EPLB, while improvements on the mathematical benchmarks are generally more moderate. The effect is not uniform: on LLM-SQL, SEAEVO$_{\text{OPEN}}$ is slightly below OpenEvolve, while SEAEVO$_{\text{SHINKA}}$ improves over ShinkaEvolve. Prism exhibits substantially higher run-to-run variance; although one run escapes the baseline plateau by a large margin, the extended analysis does not establish a statistically significant improvement (Appendix D.8). A possible explanation is that strategy-level organization is more useful when search contains multiple qualitatively distinct algorithmic directions.

Faster convergence at lower cost. Figure 1 shows search trajectories on the Circle Packing benchmarks. Both SEAEVO$_{\text{OPEN}}$ and SEAEVO$_{\text{SHINKA}}$ reach the Human/SOTA reference earlier than their underlying backbones. On Circle Packing (Square), SEAEVO$_{\text{OPEN}}$ crosses 2.6 within 30 generations, whereas OpenEvolve plateaus near 2.47 throughout 100 generations. At the same time, the cumulative-cost curves grow more slowly for SEAEVO, primarily because SER replaces full source-code inspirations with compact strategy descriptions, reducing input-context cost despite the additional embedding and SLN overhead.

4.3 ABLATION STUDY

Table 3 reports ablation results on Circle Packing (Square) with Gemini-3-Flash. The no-module baseline disables SA, SER, and SLN and uses no inspiration candidates. It therefore differs from the original ShinkaEvolve baseline in Table 1, which retains its native inspiration mechanism. All three modules improve over the no-module baseline when enabled individually, but exhibit different empirical effects. SA and SER produce modest improvements in average fitness while attaining substantially higher best scores. SLN yields the largest standalone improvement in average fitness and reaches its best solution earlier than the other single-module variants. The full SEAEVO achieves the highest average fitness, reaches its best solution earliest, and shows substantially lower variance than all one- and two-module variants. The module effects are not simply additive. In particular, the

Table 3: Ablation study on Circle Packing (Square) with Gemini-3-Flash. We report mean fitness score (Avg), standard deviation (Std), best score (Best), generation at which best was found (Gen@Best), and average API cost (AvgCost) across three seeds. We highlight the best and worst values for each metric in green and red, respectively.

SA	SER	SLN	Avg	Std ↓	Best	Gen@Best ↓	AvgCost ↓
✗	✗	✗	2.5023	0.0086	2.5099	99	$0.5367
✓	✗	✗	2.5043	0.0855	2.6310	67	$0.6062
✗	✓	✗	2.5224	0.0731	2.6310	95	$0.5088
✗	✗	✓	2.5634	0.0791	2.6356	61	$0.5614
✓	✓	✗	2.5172	0.0774	2.6304	59	$0.6576
✓	✗	✓	2.5870	0.0816	2.6308	93	$0.7147
✗	✓	✓	2.5901	0.0724	2.6360	88	$0.5731
✓	✓	✓	2.6297	0.0077	2.6353	44	$0.7075

SA+SER configuration has lower average fitness than either SA or SER alone, whereas adding SLN to either component improves the average score. These results suggest complementary interactions among the three components rather than purely additive effects.

4.4 Further Analysis

We further examine three aspects of SeaEvo: whether meaningful associations within the strategy-augmented state contribute to search, whether explicit strategy representations provide value beyond full-code embeddings, and how strategy-level context affects inference cost.

Association-disruption controls. SER-Permute randomly reassigns strategy embeddings among programs at each generation before applying the unchanged clustering and retrieval procedure. This breaks the program–strategy association while preserving the embedding distribution, clustering mechanism, retrieval logic, prompt structure, and inference budget. SLN-Shuffle instead randomizes the strategy–fitness association supplied to landscape analysis while preserving the strategy descriptions, fitness values, SLN prompts, and update schedule. We evaluate both controls on TXN and EPLB using SeaEvo$_{\text{Shinka}}$ with Gemini-3-Flash over three independent 100-generation runs.

Table 4: Controlled disruption of program–strategy and strategy–fitness associations.

Task	Configuration	Avg	Best
TXN	SeaEvo (full)	**3886.1**$_{\pm 517.9}$	**4219.4**
	SER-Permute	3340.1$_{\pm 183.3}$	3521.1
	SLN-Shuffle	3538.5$_{\pm 64.7}$	3610.1
EPLB	SeaEvo (full)	**0.1958**$_{\pm 0.0002}$	0.1960
	SER-Permute	0.1737$_{\pm 0.0269}$	**0.1973**
	SLN-Shuffle	0.1721$_{\pm 0.0264}$	0.1969

Table 4 reports the correspondence-disruption controls on TXN and EPLB. Disrupting either association reduces average performance by 8.9–14.1% on TXN and 11.3–12.1% on EPLB. These results support that meaningful associations within the strategy-augmented state contribute to the performance gains, rather than the improvements being explained solely by additional prompting or LLM calls. Although one EPLB control run attains a slightly higher best score, both controls show lower mean performance and substantially higher variance.

Strategy embeddings versus code embeddings. To test whether explicit strategy descriptions provide value beyond a generic semantic representation of code, we replace strategy embeddings in retrieval with embeddings of the complete program source code while holding the evolutionary backbone, retrieval procedure, cluster configuration, search budget, and evaluator fixed.

Table 5: Search performance using strategy-description or full-code embeddings for strategy-space retrieval.

Task	Representation	Avg	Best
EPLB	Strategy embedding	**0.1958**$_{\pm 0.0002}$	0.1960
	Code embedding	0.1794$_{\pm 0.0307}$	**0.1972**
TXN	Strategy embedding	**3886.1**$_{\pm 517.9}$	**4219.4**
	Code embedding	3648.6$_{\pm 330.36}$	3861.0

Table 5 compares strategy-description and full-code embeddings under the same retrieval setup. Strategy embeddings achieve higher mean per-

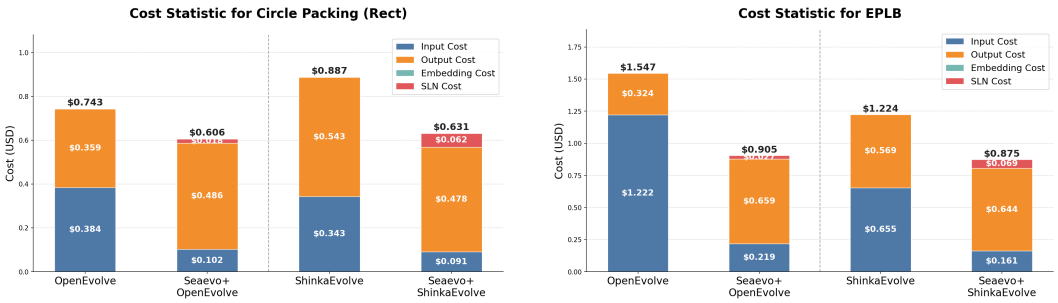


Figure 4: Cost breakdown on Circle Packing (Rect) and EPLB with Gemini-3-Flash. Bars are decomposed into input cost (prompt tokens), output cost (completion tokens), embedding cost (`text-embedding-3-small` calls for strategy descriptions), and SLN cost (landscape navigation). SEAEVO reduces total cost in all four comparisons despite adding embedding and SLN overhead, primarily by replacing source-code context with compact strategy descriptions in the mutation prompt.

formance on both tasks: 6.5% on TXN and 9.1% on EPLB. On EPLB, one code-embedding run attains a marginally higher best score despite lower mean performance and substantially higher variance. These results favor the compact strategy representation in the evaluated settings, while the main advantage appears to be its ability to preserve task-relevant structure in a much shorter representation for retrieval.

Full-code embeddings are 4.67–$9.09\times$ more expensive than strategy embeddings in these experiments (Table 16), although embedding cost is small in absolute terms. More importantly, strategy descriptions contain approximately 71 tokens on average, compared with approximately 1,127 tokens for full programs (Table 18). This compactness reduces the amount of retrieved context supplied to the mutation model, especially when multiple archive entries are used as inspirations.

Per-run API cost breakdown. Figure 4 reports the per-run API cost breakdown on Circle Packing (Rect) and EPLB. Across both tasks and both evolutionary backbones, SEAEVO reduces total cost, with savings ranging from 18.4% to 41.5%. The reduction is most pronounced on EPLB, where candidate programs are longer and code-level context becomes more expensive.

The main source of savings is lower input-token cost. Instead of passing full source code for retrieved inspiration programs, SER provides compact strategy descriptions together with fitness information. This substantially shortens the mutation prompt, especially for tasks with longer programs. Output cost remains comparable and can slightly increase because SA asks the model to produce structured diagnose–direct–implement reasoning before generating the new candidate.

The additional overhead from strategy embeddings and SLN is small relative to these input-context savings. Embedding calls are inexpensive, and SLN is invoked only periodically. Overall, these results show that strategy-level context provides a substantially more compact representation for retrieval while retaining useful search-relevant information.

5 CONCLUSION

We introduced SEAEVO, a strategy-state layer that augments evolutionary search with persistent, candidate-aligned strategy representations. By organizing archived candidates according to their algorithmic strategies, retrieving complementary experience across strategy families, and periodically reasoning over population-level strategy state, SEAEVO enables evolutionary search to make explicit use of which algorithmic directions have been explored. Across mathematical and systems optimization benchmarks, SEAEVO improves the mean performance of its matched evolutionary backbone in 23 of 24 comparisons, with gains also observed on the evaluated agent-scaffold benchmarks. At the same time, compact strategy-level retrieval reduces LLM context and lowers average token usage by 48.8%, while allowing competitive solutions to be reached at lower cumulative API cost. Overall, these results support persistent strategy representations as a practical abstraction for organizing and reusing search experience in LLM-guided evolutionary systems.

AI USE STATEMENT

In this work, we used generative AI tools to assist with language editing and polishing of the manuscript, as well as code implementation. We did not use generative AI tools to formulate research ideas or hypotheses, design the method or experiments, process data, or interpret experimental results. Proof generation and synthetic data generation are not applicable to this work. All AI-assisted text and code were reviewed and verified by the authors. AI-assisted code was further validated by running the corresponding experimental pipelines. All AI-edited text was reviewed and revised by the authors, and all citations suggested or modified with AI assistance were verified against the original sources. We take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide implementation and evaluation details throughout the paper and appendices. Section 3 describes the SEAEVO method, with complete pseudocode in Appendix A. Appendix C documents the benchmark tasks, models, evolutionary backbones, hyperparameters, and evaluation protocols. Appendix F provides task specifications, prompts, qualitative examples, and representative discovered programs. We additionally release an anonymous implementation at https://anonymous.4open.science/r/SeaEvo.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

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

Audrey Cheng, Shu Liu, Melissa Z. Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya A. Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How AI is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

Karine Even-Mendoza, Alexander Brownlee, Alina Geiger, Carol Hanna, Justyna Petke, Federica Sarro, and Dominik Sobania. Llm-guided genetic improvement: Envisioning semantic aware automated software evolution. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 3902–3906. IEEE, 2025.

Google DeepMind. Gemini 3 flash. https://deepmind.google/models/gemini/flash/, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=lKEdGCoDNC.

Yoonho Lee, Joseph Boen, and Chelsea Finn. Feedback descent: Open-ended text optimization via pairwise comparison. *arXiv preprint arXiv:2511.07919*, 2025.

Tiancheng Li, Yuhang Wang, Zhiyang Chen, Zijun Wang, Liyuan Ma, and Guo-Jun Qi. C-evolve: Consensus-based evolution for prompt groups. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=iyxRqYiCbF.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, pp. 32201–32223. PMLR, 2024.

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

Fengyuan Liu, Huang Yi, Sichun Luo, Yuqi Wang, Yazheng Yang, Xinye Li, Zefa Hu, Junlan Feng, and Qi Liu. Cognitive alpha mining via llm-driven code-based evolution. *arXiv preprint arXiv:2511.18850*, 2025.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. Evox: Meta-evolution for automated discovery. *CoRR*, abs/2602.23413, 2026b. doi: 10.48550/ARXIV.2602.23413. URL https://doi.org/10.48550/arXiv.2602.23413.

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Alexandros G. Dimakis, Matei Zaharia, and Ion Stoica. Skydiscover: A flexible framework for ai-driven scientific and algorithmic discovery, 2026c. URL https://skydiscover-ai.github.io/blog.html.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. New embedding models and API updates. https://openai.com/blog/new-embedding-models-and-api-updates, 2024.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Paul Röttger, Hannah Kirk, Bertie Vidgen, Giuseppe Attanasio, Federico Bianchi, and Dirk Hovy. Xstest: A test suite for identifying exaggerated safety behaviours in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 5377–5400, 2024.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

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, 2024.

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

Xiaomi MiMo Team. MiMo-V2-Pro. https://mimo.xiaomi.com/mimo-v2-pro, March 2026.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, Ed H. Chi, Fernando Pereira, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. Heuragenix: Leveraging llms for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025b.

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.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025.

Appendix Table of Contents

A Algorithmic Details **14**

B Positioning **15**

C Experimental Setup **16**

C.1 Datasets and Tasks 16

C.2 Models and Evolutionary Backbones 16

C.3 Agent-Scaffold Evaluation Protocol 17

D Additional Empirical Validation **18**

D.1 Validating the Strategy Representation 18

D.2 Paired counterfactual retrieval. 19

D.3 Generality across Models and Evolutionary Backbones 19

D.4 Ablation Study on TXN 20

D.5 Efficiency and Cost Accounting 21

D.6 Hyperparameter Sensitivity 21

D.7 Visualization 22

D.8 Statistical Robustness on Prism. 23

D.9 Additional Agent-Scaffold Results 23

D.10 High-Precision Score Reference 23

E Case Studies **24**

E.1 Case Study for Prism 24

E.2 Case Study for EPLB. 25

F Tasks, Prompts, and Qualitative Examples **27**

F.1 Task Specifications and Prompts 27

F.2 SEAEVO Prompt and Response Example 33

F.3 Qualitative Solution Visualization 37

F.4 Representative Discovered Programs 38

Algorithm 1 Overall SEAEVO procedure

Require: Initial program P_0, total generations T, SLN interval Δ, number of clusters C, exploration ratio ε
Ensure: Best program P^*

Evaluate P_0 to obtain s_0 and optional $\mathbf{b}_0$
$d_0 \leftarrow \mathrm{LLM}_{\mathrm{describe}}(P_0)$; $\mathbf{e}_0 \leftarrow \mathrm{embed}(d_0)$
$\mathcal{A}_1^{\mathrm{strat}} \leftarrow \{(P_0, s_0, d_0, \mathbf{e}_0, \mathbf{b}_0)\}$
$P^* \leftarrow P_0$; $s^* \leftarrow s_0$; $\mathcal{L}_0 \leftarrow \varnothing$
for $t = 1, 2, \ldots, T$ **do**
  $P_{\mathrm{parent}} \leftarrow \textsc{BaseSelect}(\mathcal{A}_t^{\mathrm{strat}})$
  if $t \bmod \Delta = 0$ **then**
    $\mathcal{L}_t \leftarrow \mathrm{SLN}(\mathcal{A}_t^{\mathrm{strat}})$
  else
    $\mathcal{L}_t \leftarrow \mathcal{L}_{t-1}$
  end if
  if $\mathrm{rand}() < \varepsilon$ **then** ▷ ε-greedy fallback
    $P' \leftarrow \textsc{BaseMutate}(P_{\mathrm{parent}}, \mathcal{A}_t^{\mathrm{strat}})$
    $d' \leftarrow \mathrm{LLM}_{\mathrm{describe}}(P')$ ▷ post-hoc strategy description
  else
    $\mathcal{G}_t \leftarrow \mathrm{SER}(\mathcal{A}_t^{\mathrm{strat}}, P_{\mathrm{parent}}, C)$
    $(P', d') \leftarrow \mathrm{SA}(P_{\mathrm{parent}}, \mathcal{G}_t, \mathcal{L}_t)$
  end if
  Evaluate P' to obtain s' and optional $\mathbf{b}'$
  $\mathbf{e}' \leftarrow \mathrm{embed}(d')$
  $\mathcal{A}_{t+1}^{\mathrm{strat}} \leftarrow \textsc{BaseUpdate}\left(\mathcal{A}_t^{\mathrm{strat}}, (P', s', d', \mathbf{e}', \mathbf{b}')\right)$
  if $s' > s^*$ **then**
    $P^* \leftarrow P'$; $s^* \leftarrow s'$
  end if
end for
return P^*

Algorithm 2 Strategy Articulation

Require: Parent program P_{parent}, retrieval context $\mathcal{G}_t$, navigation guidance $\mathcal{L}_t$
Ensure: New program P', intended strategy description d'

$(\mathrm{diag}, d', P') \leftarrow \mathrm{LLM}_{\mathrm{SA}}(P_{\mathrm{parent}}, \mathcal{G}_t, \mathcal{L}_t)$ ▷ diagnose → direct → implement
return (P', d')

A ALGORITHMIC DETAILS

Algorithm 1 summarizes the complete SEAEVO procedure. At each generation, the underlying backbone selects a parent from the current archive. With probability ε, SEAEVO follows the backbone's default mutation pipeline; otherwise, SER constructs strategy-level retrieval context and SA jointly generates a new program and its intended strategy. SLN operates at a slower timescale, refreshing the population-level navigation state every Δ generations. After evaluation, the resulting candidate is incorporated into the archive using the backbone's original update rule.

Candidates generated through SA receive a prospective strategy description jointly with the program. The initial program and candidates generated through the ε-greedy fallback are not produced by SA; for these candidates, a post-hoc strategy description is generated to maintain the strategy-augmented archive. Behavioral vectors $\mathbf{b}$ are optional and are omitted when only scalar feedback is available.

For SER, we use a unified retrieval criterion for both behavioral and scalar-only feedback. Let $F_P(P_i \mid \mathcal{I})$ denote the marginal behavioral coverage contributed by candidate P_i relative to parent P and an already selected inspiration set $\mathcal{I}$, as defined in Section 3.4. We define

$$\textsc{RetrievalGain}(P_i; P, \mathcal{I}) = \begin{cases} F_P(P_i \mid \mathcal{I}), & \text{if instance-level feedback is available,} \\ s_i, & \text{otherwise.} \end{cases}$$

Thus, SER selects candidates by marginal behavioral coverage when instance-level feedback is available, and by scalar fitness otherwise.

Algorithm 3 Stratified Experience Retrieval

Require: Archive $\mathcal{A}_t^{\text{strat}}$, parent P_{parent}, number of clusters C
Ensure: Retrieval context $\mathcal{G}_t$
$P_{\text{best}} \leftarrow \arg\max_{P_i \in \mathcal{A}_t^{\text{strat}}} s_i$
if $|\mathcal{A}_t^{\text{strat}}| < C$ **then** ▷ warm-up before strategy clustering
  $\mathcal{B} \leftarrow \mathcal{A}_t^{\text{strat}} \setminus \{P_{\text{parent}}, P_{\text{best}}\}$
  if $\mathcal{B} \neq \varnothing$ **then**
    $P^{\text{div}} \leftarrow \arg\max_{P_i \in \mathcal{B}} \textsc{RetrievalGain}(P_i; P_{\text{best}}, \varnothing)$
    return unique entries from $\{P_{\text{parent}}, P_{\text{best}}, P^{\text{div}}\}$
  else
    return unique entries from $\{P_{\text{parent}}, P_{\text{best}}\}$
  end if
end if
$\{\mathcal{S}_c\}_{c=1}^{C} \leftarrow \textsc{KMeans}\left(\{\mathbf{e}_i : (P_i, s_i, d_i, \mathbf{e}_i, \mathbf{b}_i) \in \mathcal{A}_t^{\text{strat}}\}, C\right)$
$c^* \leftarrow$ cluster containing P_{parent}
if $\mathcal{S}_{c^*} \setminus \{P_{\text{parent}}\} = \varnothing$ **then**
  $P^{\text{intra}} \leftarrow \arg\max_{P_i \in \mathcal{A}_t^{\text{strat}} \setminus \{P_{\text{parent}}\}} \textsc{RetrievalGain}(P_i; P_{\text{parent}}, \varnothing)$
else
  $P^{\text{intra}} \leftarrow \arg\max_{P_i \in \mathcal{S}_{c^*} \setminus \{P_{\text{parent}}\}} \textsc{RetrievalGain}(P_i; P_{\text{parent}}, \varnothing)$
end if
$c' \sim \text{Uniform}\{c : c \neq c^*\}$
$P^{\text{cross}} \leftarrow \arg\max_{P_i \in \mathcal{S}_{c'}} \textsc{RetrievalGain}(P_i; P_{\text{parent}}, \{P^{\text{intra}}\})$
return $\{P_{\text{parent}}, P^{\text{intra}}, P^{\text{cross}}\}$

Algorithm 4 Strategic Landscape Navigation

Require: Archive $\mathcal{A}_t^{\text{strat}}$
Ensure: Navigation guidance $\mathcal{L}_t$
$\mathcal{D}_t \leftarrow \{(d_i, s_i) : (P_i, s_i, d_i, \mathbf{e}_i, \mathbf{b}_i) \in \mathcal{A}_t^{\text{strat}}\}$ ▷ strategy descriptions and observed fitness
$\mathcal{L}_t \leftarrow \text{LLM}_{\text{navigate}}(\mathcal{D}_t)$
return $\mathcal{L}_t$

B Positioning

This section further clarifies the methodological positioning of SeaEvo relative to closely related work on language-guided evolutionary search, reflection, memory, and semantic organization. The distinction is not whether natural language or semantic structure is used during search, but how language-level information is represented, maintained, and operated on within the evolutionary process.

Shared ingredients with prior work. SeaEvo builds on several ideas established by prior work. EoH (Liu et al., 2024) demonstrates that natural-language thoughts can accompany program generation. ReEvo (Ye et al., 2024) and Reflexion (Shinn et al., 2023) show that verbal reflection can guide future proposals, while GEPA (Agrawal et al., 2026) demonstrates the value of reusing and combining language-level experience. Other methods similarly maintain reusable insights, abstractions, or evolution strategies (Chen et al., 2025; Wu et al., 2025; Yang et al., 2025b), while PatchCat (Even-Mendoza et al., 2025) shows that semantic organization of program modifications can expose useful structure beyond source syntax.

Structural distinction of SeaEvo. The methodological distinction lies in the role assigned to language-level strategy information. Prior methods typically use language as candidate-level generation context, trajectory-level reflection, reusable experience, run-level memory, or semantic characterization of individual edits. In contrast, SeaEvo persistently associates an explicit strategy with each archived candidate and organizes the evolving population through these candidate-aligned strategies. Strategy therefore becomes part of the maintained evolutionary state rather than only auxiliary context for individual mutation steps.

Table 6: Positioning of SEAEVO relative to related methods. The comparison focuses on the organization and use of language-level information during evolutionary search.

Method	Language-level representation	Organization scope	Primary role in search
EoH (Liu et al., 2024)	Thought–code pairs	Candidate-level	Language-guided candidate generation
ReEvo (Ye et al., 2024)	Reflections	Trajectory / run-level context	Reflection-guided mutation
GEPA (Agrawal et al., 2026)	Reflection-based experience	Experience-level collection	Reuse and combination of language-level lessons
ShinkaEvolve (Lange et al., 2026)	Mutation notes / scratchpad context	Run-level context	Context augmentation during evolution
PatchCat (Even-Mendoza et al., 2025)	Semantics of generated code edits	Edit-level semantic organization	Patch characterization and redundancy identification
SEAEVO (Ours)	Candidate-aligned algorithmic strategies	Persistent population-level strategy archive	Family-aware retrieval and population-level search navigation

This population-level representation enables operations that require relationships across candidates. In particular, SEAEVO can organize different implementations according to their high-level algorithmic directions, retrieve experience within and across strategy families, and reason over accumulated family-level progress when guiding subsequent search. The distinction is therefore primarily about the *scope and indexing of the representation*: language-level information is promoted from local or accumulated context into persistent population-level state over which search operators can act.

Therefore, the contribution of SEAEVO is to operationalize candidate-aligned strategy as persistent population-level evolutionary state, together with retrieval and navigation mechanisms defined over that state. Executable programs remain the source of truth for evaluation and selection, while strategy provides a compact semantic representation for organizing and reasoning over the search history.

C EXPERIMENTAL SETUP

C.1 DATASETS AND TASKS

For mathematical optimization, we evaluate on four algorithm discovery tasks: Circle Packing (Rectangle), Circle Packing (Square), Heilbronn Triangles, and MinMax Distance. For systems optimization, we use four tasks from the ADRS benchmark (Cheng et al., 2025): Prism, TXN, EPLB, and LLM-SQL. The detailed task descriptions and prompts are provided in Appendix F.1.

For agent-scaffold design, we use the IFBench (Pyatkin et al., 2025) and XSTest benchmark (Röttger et al., 2024). We evaluate each benchmark under two inference-budget settings, where the inference model is allowed to make either one or up to three LLM calls per example. We split the benchmark into a feedback set, a metric set, and a test set, with 60, 75, and 150 examples, respectively. The feedback set is used during evolution, the metric set is used for candidate selection, and the test set is held out for final evaluation.

C.2 MODELS AND EVOLUTIONARY BACKBONES

For evolution, we adopt two frontier models as representative non-reasoning and reasoning backbones: Gemini-3-Flash (Google DeepMind, 2025), accessed via the OpenRouter API[2], and MiMo-V2-Pro (Xiaomi MiMo Team, 2026), accessed via its official API[3]. The decoding temperature for evolution model is set to 0.7. For agent-scaffold tasks, we use Qwen3-8B (Yang et al., 2025a) as the inference model, deployed locally with vLLM (Kwon et al., 2023) on 2×A6000 GPUs. Following Li

[2] https://openrouter.ai/api/v1
[3] https://api.xiaomimimo.com/v1

Table 7: Agent-scaffold data partitioning.

Benchmark	Official pool	Feedback	Metric	Final test	Exact duplicates
IFBench	294	60	75	150	0
XSTest	450	60	75	150	0

Table 8: Safe/unsafe composition of the XSTest subsets.

Split	Safe	Unsafe
Feedback	35	25
Metric	43	32
Test	85	65

et al. (2026), the default maximum generation length is set to 4096 and the decoding temperature for the inference model is set to 1.0. The decoding temperature can be modified by the evolution model.

We compare SEAEVO against several representative LLM-driven evolutionary methods and use the corresponding open-source backbones to instantiate SEAEVO. GEPA (Agrawal et al., 2026) performs reflective prompt evolution using verbal feedback over execution traces. OpenEvolve (Sharma, 2025) is an open-source reimplementation of the AlphaEvolve (Novikov et al., 2025) paradigm. ShinkaEvolve (Lange et al., 2026) improves sample efficiency via adaptive parent sampling and novelty-based rejection. AlphaEvolve (Novikov et al., 2025) is closed-source and is included only as a reference target. We use default hyperparameters as reported in the respective papers. For OpenEvolve, we set `num_context_programs=` 5, `num_islands=` 5, `population_size=` 40, `archive_size=` 100, `exploration_ratio=` 0.2, `exploitation_ratio=` 0.7, `migration_interval=` 10, `migration_rate=` 0.1, and `feature_bins=` 10. For ShinkaEvolve, we set `num_islands=` 3, `archive_size=` 20, `migration_interval=` 5, and `migration_rate=` 0.2.

C.3 AGENT-SCAFFOLD EVALUATION PROTOCOL

We provide additional details for IFBench and XSTest to clarify how instance-level feedback is separated from final evaluation.

Test isolation and split construction. The 150-example final-test set is accessed only once, after evolution has completed and the final program has been selected. It is not used for prompt construction, fitness computation, candidate selection, or early stopping.

For both benchmarks, we construct mutually exclusive feedback, metric, and final-test subsets with 60, 75, and 150 examples, respectively, and no exact prompt overlap across subsets.

For IFBench, we shuffle the 294 official examples with random seed 42, assign the first 75 to the metric set, the next 150 to final test, and the following 60 to the feedback set; the remaining nine examples are unused. For XSTest, the three subsets are sampled without replacement. Their safe/unsafe composition is shown in Table 8.

All reported experiments use 100 evolutionary generations. Each candidate is evaluated on the same fixed 75-example metric set, corresponding to approximately 7,500 metric-example evaluations per run in addition to the 75 evaluations of the initial program. SEAEVO, ShinkaEvolve, and OpenEvolve use only the metric set during evolution. GEPA additionally samples five feedback examples per candidate to construct reflection information. The final-test protocol is identical and fully isolated for all methods.

Semantic overlap audit. The splits are example-disjoint but were not designed to be category- or template-disjoint. To quantify residual semantic overlap, we compare each final-test prompt with the combined feedback and metric sets using unigram/bigram TF–IDF similarity.

No final-test prompt is an exact duplicate of an evolution-side example, and the large majority have no highly similar counterpart. The remaining high-similarity pairs primarily reflect shared benchmark

Table 9: Maximum TF–IDF similarity between each final-test prompt and the combined feedback and metric sets.

Benchmark	Mean max.	95th percentile	Maximum	Similarity ≥ 0.7
IFBench	0.285	0.648	0.872	6 / 150
XSTest	0.272	0.613	0.756	2 / 150

Table 10: Strategy–implementation alignment across 12 independent runs. Matched cosine compares the correct strategy–diff pair; Random and Fitness-matched use deranged controls. Top-5 reports retrieval of the corresponding implementation from its strategy. Pairwise ρ is the Spearman correlation between pairwise distances in strategy and code-diff spaces.

Task	Matched cosine	Random	Fitness-matched	Top-5 retrieval	Pairwise ρ
Circle Packing (Square)	0.455 ± 0.011	0.431 ± 0.011	0.435 ± 0.010	$11.4 \pm 2.8\%$	0.266 ± 0.055
Circle Packing (Rectangle)	0.485 ± 0.014	0.475 ± 0.011	0.475 ± 0.011	$16.8 \pm 0.7\%$	0.141 ± 0.156
TXN	0.555 ± 0.013	0.511 ± 0.011	0.516 ± 0.009	$31.6 \pm 1.0\%$	0.170 ± 0.059
EPLB	0.552 ± 0.029	0.525 ± 0.028	0.529 ± 0.029	$27.5 \pm 3.0\%$	0.199 ± 0.060

templates and, for XSTest, its deliberately contrastive safe/unsafe construction. The evaluation therefore measures held-out within-distribution generalization rather than topic- or template-disjoint transfer.

The three independent runs for each method use the same fixed partition. They measure stochastic variation in LLM generation and evolutionary search, but do not establish robustness to alternative feedback/metric/test partitions. Testing the latter would require full re-evolution under independently sampled splits.

D Additional Empirical Validation

D.1 Validating the Strategy Representation

The main method assumes that candidate-aligned strategy descriptions preserve meaningful information about the algorithmic changes implemented in the corresponding programs. We validate this assumption from two complementary perspectives: strategy–implementation alignment and manual inspection of induced strategy families.

Strategy–implementation alignment. We analyze 1,116 valid parent–child pairs from 12 independent SeaEvo$_{\text{ShinkA}}$ runs, with three runs each on Circle Packing (Square), Circle Packing (Rectangle), TXN, and EPLB. For each generated child, we compare its strategy description with the implemented parent–child code diff. Both are embedded using the same `text-embedding-3-small` model.

For each matched strategy–diff pair, we compare cosine similarity against two controls: (i) a within-run random derangement and (ii) a stricter fitness-matched derangement formed within score quintiles. We additionally measure top-5 retrieval of the corresponding code diff from its strategy and the Spearman correlation between pairwise distances in strategy and code-diff spaces.

As shown in Table 10, matched pairs have higher similarity than both controls in all 12 runs (permutation test, $p \leq 0.002$). Top-5 retrieval is consistently above the approximately 5% random level, and pairwise distances are positively correlated in 11 of 12 runs. The correlations are modest, which is expected: strategy descriptions intentionally compress implementation details rather than reproduce code structure. Overall, the results indicate that the strategy representation preserves information about the implemented algorithmic direction while remaining substantially more compact than source code.

Manual audit of strategy families. We further inspect the semantic coherence of the induced strategy families at generation 50 on Circle Packing and TXN. For each family, we inspect the program nearest to the cluster centroid and a distinct high-scoring valid program. We label a description *exact* when all stated mechanisms are implemented, *partial* when the primary paradigm is correct

Table 11: Manual audit of strategy-family interpretability at generation 50. Audit reports the number of representative strategies judged as exactly matched, partially matched, or inconsistent with their assigned strategy family (E/P/I).

Task	Representative strategy families	Cluster sizes	Silhouette	ARI stability	Audit (E/P/I)
Circle Packing	constructive grid/greedy; force relaxation; constrained optimization; staged refinement; multi-start search	12/7/8/10/13	0.071	0.400	7/3/0
TXN	priority greedy; swap/shift hill climbing; simulated annealing; GRASP; proxy-assisted insertion	12/14/9/8/7	0.085	0.493	3/7/0

Table 12: Paired counterfactual retrieval over 100 matched search states per benchmark.

Benchmark	Semantic / Random / Tie	Sign test	Semantic score	Random score
Circle Packing	56 / 34 / 10	$p = 0.026$	2.4368 ± 0.4828	2.3052 ± 0.6095
Prism	54 / 43 / 3	$p = 0.310$	24.8222 ± 4.6568	24.8947 ± 3.1619

but auxiliary details are overstated, and *inconsistent* when the main paradigm is absent. We also report the silhouette coefficient and the mean pairwise adjusted Rand index (ARI) across ten k-means initializations.

Table 11 shows that the primary algorithmic paradigm agrees with the implementation in all inspected representatives. However, some descriptions overstate auxiliary details; for example, repeated SLSQP refinement may be described as Basin Hopping. We therefore treat the strategy representation as a useful but imperfect semantic abstraction rather than a lossless description of program behavior.

D.2 Paired counterfactual retrieval.

The previous controls operate over complete evolutionary trajectories. To isolate the effect of retrieval at the level of a single mutation decision, we sample 100 eligible search states per benchmark from three independent SeaEvo$_{\text{Shinka}}$ trajectories using Gemini-3-Flash. At each state, the semantic condition uses the references selected by strategy-space retrieval, while the control replaces them with two strategies sampled uniformly from previously available programs, excluding the parent and the original references.

The parent program and score, system prompt, navigation guidance, patch type, evaluator, model configuration, and all prompt text outside the reference block are held fixed. Each arm therefore uses one inference call and differs only in the retrieved strategies.

Semantic retrieval wins significantly more often than random retrieval on Circle Packing under an exact sign test ($p = 0.026$). On Prism, the difference is not statistically significant ($p = 0.310$), and the mean scores are nearly identical. This suggests that the single-step benefit of semantic retrieval is task dependent, while the full-run correspondence controls provide broader evidence that preserving strategy-space organization matters over longer search horizons.

D.3 Generality across Models and Evolutionary Backbones

Additional evolutionary frameworks. To examine whether SeaEvo is tied to a specific evolutionary backbone, we additionally integrate it with two LLM-driven evolutionary frameworks: GEPA (Agrawal et al., 2026) and AdaEvolve (Cemri et al., 2026). Table 13 shows that SeaEvo improves average performance in most matched comparisons across both backbones. The exception is EPLB with GEPA, where the average score decreases slightly while the best score improves. These

Table 13: Generality of SEAEVO as a plug-in over additional evolutionary backbones GEPA and AdaEvolve, evaluated on a mix of mathematical and systems optimization benchmarks. Within each backbone group, SEAEVO-augmented rows (highlighted) are compared against the corresponding base framework. Best results are in boldface.

Method	Circle Packing (Square)		Heilbronn (Triangles)		TXN		EPLB	
	Avg	Best	Avg	Best	Avg	Best	Avg	Best
GEPA	$2.5306_{\pm 0.0380}$	2.5574	$0.0303_{\pm 0.0011}$	0.0316	$3266.0_{\pm 243.1}$	3496.5	$0.1472_{\pm 0.0062}$	0.1543
+ SEAEVO	$2.5408_{\pm 0.0442}$	2.6022	$0.0322_{\pm 0.0017}$	0.0337	$3386.0_{\pm 11.0}$	3401.0	$0.1397_{\pm 0.0178}$	0.1649
AdaEvolve	$2.6223_{\pm 0.0086}$	2.6285	$0.0322_{\pm 0.0003}$	0.0324	$3622.0_{\pm 305.0}$	4032.0	$0.1270_{\pm 0.0002}$	0.1273
+ SEAEVO	$\mathbf{2.6283}_{\pm 0.0097}$	**2.6360**	$\mathbf{0.0329}_{\pm 0.0014}$	**0.0338**	$\mathbf{4238.0}_{\pm 598.0}$	**5076.0**	$\mathbf{0.1597}_{\pm 0.0111}$	**0.1689**

Table 14: Performance with Claude Opus 4.8 on TXN Scheduling.

Method	Avg	Best	Avg. cost / run
ShinkaEvolve	4085.79 ± 160.17	**4255.32**	$15.39
+ SEAEVO	4196.65 ± 72.77	**4255.32**	$13.33
OpenEvolve	3925.39 ± 151.42	4098.36	$11.24
+ SEAEVO	$\mathbf{4202.27 \pm 60.65}$	4237.29	**$10.98**

results provide additional evidence that the strategy-state layer can be integrated with evolutionary backbones beyond OpenEvolve and ShinkaEvolve.

Additional mutation model. We further evaluate SEAEVO with Claude Opus 4.8 as the mutation model on TXN Scheduling. All methods use the same initialization, search budget, evaluator, and cost-accounting protocol, with three independent runs.

As shown in Table 14, SEAEVO improves the mean score and reduces the average API cost for both backbones. It also yields lower run-to-run variance in both cases. With ShinkaEvolve, SEAEVO improves the mean while matching the same best score; with OpenEvolve, it improves both the mean and best score.

D.4 ABLATION STUDY ON TXN

Table 15 shows that the full SEAEVO framework achieves the strongest overall performance, obtaining the best average fitness and matching the highest best score among all variants. This indicates that the three strategy-state operations are complementary: strategy articulation provides more deliberate mutations, stratified retrieval improves the diversity of useful inspirations, and landscape navigation helps steer the search away from over-explored directions.

Among the individual operations, SLN is the most effective standalone component, nearly matching the full model in average fitness. This suggests that strategy-level landscape guidance is particularly important for TXN, where the search can otherwise spend many generations refining already saturated directions. In contrast, SA and SER alone bring more limited gains. SA improves the stability of mutation by making the intended improvement explicit, while SER mainly broadens the inspiration context. However, neither is sufficient on its own to consistently guide the global search.

The interaction between modules reveals a trade-off between early discovery and overall performance. SER+SLN finds a strong solution the earliest, indicating that complementary retrieval can accelerate landscape-guided exploration. However, the full model achieves the best overall performance, suggesting that SA provides additional mutation-level guidance that improves the quality of the search. This comes with higher variance and cost, reflecting a more aggressive and exploratory search process. Overall, these results show that SEAEVO's advantage does not come from a single operation in isolation, but from coupling candidate-aligned strategy construction with strategy-space retrieval and population-level navigation.

Table 15: Ablation study on TXN with Gemini-3-Flash. We report mean fitness (Avg), standard deviation (Std), best score (Best), generation at which best was found (Gen@Best), and average API cost (AvgCost) across three runs. We highlight the best and worst values for each metric in green and red, respectively.

SA	SER	SLN	Avg	Std ↓	Best	Gen@Best ↓	AvgCost ↓
✗	✗	✗	3541.4	163.5	3676.5	53	0.5055
✓	✗	✗	3668.1	136.9	3831.4	94	0.6115
✗	✓	✗	3599.8	206.0	3891.1	97	0.4811
✗	✗	✓	3881.1	142.9	4081.6	94	0.5605
✓	✓	✗	3653.4	170.4	3787.9	85	0.6205
✓	✗	✓	3784.9	317.0	4219.4	89	0.6884
✗	✓	✓	3737.7	351.4	4132.2	30	0.5711
✓	✓	✓	3886.1	517.9	4219.4	59	0.6966

Table 16: Embedding cost of strategy-description and full-code representations.

Task	Strategy embedding	Code embedding	Cost ratio
EPLB	\$0.000293	\$0.002665	9.09×
TXN	\$0.000317	\$0.001482	4.67×

D.5 Efficiency and Cost Accounting

Token consumption. Across 32 matched task–model–backbone configurations, SeaEvo uses an average of 0.800M evolution-LLM tokens per 100-generation run, compared with 1.563M for the corresponding baselines, a reduction of 48.8%.

The savings primarily arise from a 68.1% reduction in input tokens because SeaEvo retrieves compact strategy descriptions rather than repeatedly inserting full program source code. Strategy construction increases output tokens by approximately 9.1%, but this increase is substantially smaller than the input reduction.

Embedding overhead. Across the eight main tasks, strategy descriptions contain 71.2 tokens on average. Each candidate is embedded once and the result is cached in the archive; clustering and subsequent retrieval reuse the cached embedding.

The actual embedding cost is slightly higher than the strategy-only estimate because generations without a valid strategy description conservatively fall back to embedding the generated program. Such programs contain approximately 1,127 tokens on average. Even with this fallback, the average embedding cost is only \$0.000743 per 100-generation run, and the largest task-specific value is approximately \$0.00105 on EPLB.

All reported API costs include mutation-model calls, periodic navigation calls, and embedding calls, including failed or non-executable generations. Strategy Articulation is produced within the mutation response and therefore does not incur a separate model call. Task evaluation is executed locally and is not included in API cost.

D.6 Hyperparameter Sensitivity

Table 19 reports sensitivity to three key hyperparameters on Circle Packing (Square).

SLN update interval Δ. $\Delta = 10$ achieves the best average fitness and earliest convergence. Updating too frequently ($\Delta = 5$) destabilizes search: the landscape changes before the population has accumulated enough new candidates to make the LLM's judgments reliable, producing the highest variance across all settings. Updating too infrequently ($\Delta = 20$) reduces cost but lets ineffective directions persist past the point where they should be abandoned, hurting average fitness.

Number of clusters C. Performance is robust around $C = 5$, with modest degradation at $C = 3$ (coarser strategy partitioning conflates distinct families) and $C = 8$ (too few programs per cluster

Table 17: Average evolution-LLM token consumption per 100-generation run.

Task	Baseline	SEAEVO	Reduction
Circle Packing (Square)	1.634M	0.830M	49.2%
Circle Packing (Rectangle)	1.439M	0.708M	50.8%
Heilbronn Triangles	1.321M	0.677M	48.8%
MinMax Distance	1.165M	0.692M	40.6%
Prism	1.316M	0.787M	40.2%
TXN	1.347M	0.679M	49.6%
EPLB	2.234M	1.053M	52.9%
LLM-SQL	2.052M	0.977M	52.4%
Overall	**1.563M**	**0.800M**	**48.8%**

Table 18: Strategy-embedding overhead across the eight main tasks. All costs are reported per 100 generations. The strategy-only estimate reflects the cost of embedding valid strategy text, while the actual cost also includes occasional fallback to embedding program code when no valid strategy description is available.

Task	Avg. strategy tokens	Strategy-only cost	Actual embedding cost
Circle Packing (Square)	69.0	$0.000138	$0.000925
Circle Packing (Rectangle)	73.9	$0.000148	$0.000586
Heilbronn Triangles	72.3	$0.000145	$0.000616
MinMax Distance	72.7	$0.000145	$0.000361
Prism	71.9	$0.000144	$0.000935
TXN	68.7	$0.000137	$0.000810
EPLB	66.5	$0.000133	$0.001046
LLM-SQL	74.1	$0.000148	$0.000674
Overall	**71.2**	**$0.000142**	**$0.000743**

make intra-cluster retrieval less informative). The low sensitivity across this range suggests that SEAEVO does not require careful tuning of C.

Exploration ratio ε. $\varepsilon = 0$ (fully strategy-guided, no random exploration) reaches the highest best score but does so substantially later than $\varepsilon = 0.2$, indicating that occasional fallbacks to the base mutation pipeline accelerate escape from premature strategy clusters. $\varepsilon = 0.4$ dilutes the strategy-aware pipeline too aggressively, hurting both average fitness and convergence speed. $\varepsilon = 0.2$ provides the best balance of guided exploration and stochastic diversity.

D.7 VISUALIZATION

To examine whether SEAEVO changes the distribution of explored strategies rather than merely improving the final score, we project the strategy text associated with all generated programs into two dimensions using t-SNE (Van der Maaten & Hinton, 2008). For SEAEVO$_{\text{SHINKA}}$, this text is the explicit strategy description stored in the archive; for the ShinkaEvolve baseline, which does not maintain such descriptions, we approximate it using docstrings and comments extracted from each generated program. Figure 5 shows the resulting qualitative projections on two representative systems optimization benchmarks, EPLB and TXN Scheduling. Each point denotes one generated program; marker shape distinguishes ShinkaEvolve from SEAEVO$_{\text{SHINKA}}$, and color encodes the combined score. The projections reveal a clear difference in the regions explored by the two search processes. On EPLB, ShinkaEvolve candidates concentrate in a few broad regions, whereas SEAEVO$_{\text{SHINKA}}$ produces a dense group of high-scoring candidates in a different part of the strategy space. This suggests that the improvement is associated with discovering and refining a distinct strategy family, rather than only isolated high-scoring mutations. On TXN Scheduling, ShinkaEvolve again concentrates in a small number of regions, while the top SEAEVO$_{\text{SHINKA}}$ candidates appear in a separate area of the projection. This is consistent with the intended role of SLN: redirecting mutations toward underexplored strategy families when local directions begin to saturate.

Table 19: Hyperparameter sensitivity on Circle Packing (Square) with Gemini-3-Flash. Baseline is the full SEAEVO configuration (Δ=10, C=5, ε=0.2). Each row varies one hyperparameter while holding the others fixed. The best results are in boldface.

Param	Value	Avg	Std ↓	Best	Gen@Best ↓	AvgCost ↓
Δ (SLN interval)	$\Delta = 5$	2.5628	0.1195	2.6343	80	\$0.7710
	$\Delta = 10$	**2.6297**	**0.0077**	**2.6353**	**44**	\$0.7075
	$\Delta = 20$	2.5742	0.0854	2.6307	74	**\$0.6802**
C (clusters)	$C = 3$	2.6283	**0.0052**	2.6343	86	**\$0.7006**
	$C = 5$	**2.6297**	0.0077	**2.6353**	**44**	\$0.7075
	$C = 8$	2.6175	0.0173	2.6307	72	\$0.7079
ε (Exploration ratio)	$\varepsilon = 0.0$	**2.6341**	**0.0020**	**2.6360**	78	\$0.7711
	$\varepsilon = 0.2$	2.6297	0.0077	2.6353	**44**	\$0.7075
	$\varepsilon = 0.4$	2.6263	0.0112	**2.6360**	95	**\$0.6669**

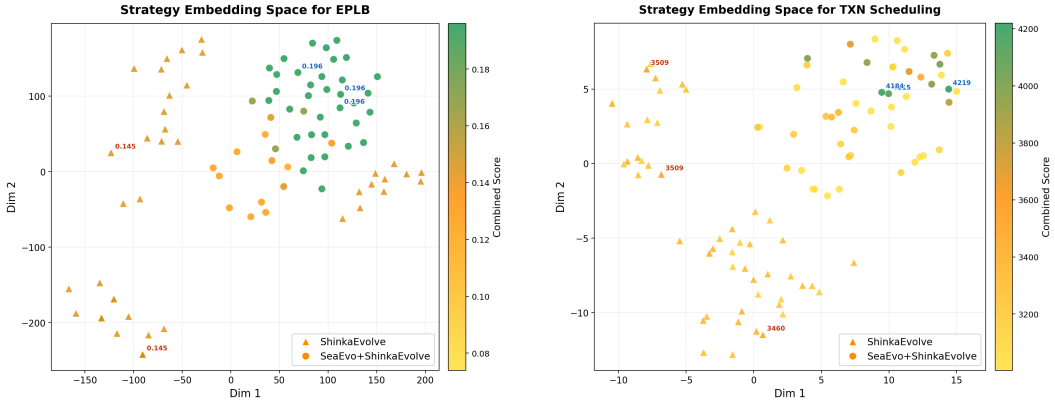


Figure 5: Strategy embedding spaces on two systems optimization benchmarks. Each point denotes a candidate program projected from its strategy embedding by t-SNE, color indicates the combined score, and marker shape distinguishes the base backbone from SEAEVO-augmented search.

D.8 STATISTICAL ROBUSTNESS ON PRISM

Prism exhibits substantially higher variance than the other ADRS benchmarks. To better characterize this behavior, we ran three additional independent SEAEVO$_{\text{SHINKA}}$ trials on Prism. The results are shown in Table 20. Across these additional runs, SEAEVO$_{\text{SHINKA}}$ obtains an average score of 32.99 with a sample standard deviation of 8.79.

The Prism results exhibit substantially higher variance than the other systems benchmarks. To characterize this uncertainty, we report six independent SEAEVO$_{\text{SHINKA}}$ runs. The original three runs achieve 78.00, 26.58, and 26.26; three additional runs achieve 42.94, 29.77, and 26.26. Across all six runs, the mean is 38.30 ± 20.48, the median is 28.18, the range is $[26.26, 78.00]$, and the 95% Student's t confidence interval for the mean is $[16.80, 59.80]$. The ShinkaEvolve baseline achieves 26.27 ± 0.28 with a median of 26.28.

A two-sided Mann–Whitney U test with asymptotic tie correction does not reach statistical significance ($p = 0.243$). We therefore treat the Prism improvement as suggestive rather than statistically conclusive and avoid using the isolated best run as primary evidence.

D.9 ADDITIONAL AGENT-SCAFFOLD RESULTS

For completeness, Figure 6 reports the agent-scaffold results under the three-call inference setting.

D.10 HIGH-PRECISION SCORE REFERENCE

Table 21 reports full-precision fitness scores for the Circle Packing (Rectangle) and MinMax Distance benchmarks, where multiple methods converge to values that differ only beyond the fourth decimal

Table 20: Additional Prism results over three independent runs.

	R1	R2	R3	Mean ± Std
Prism Score	26.26	42.94	29.77	32.99 ± 8.79

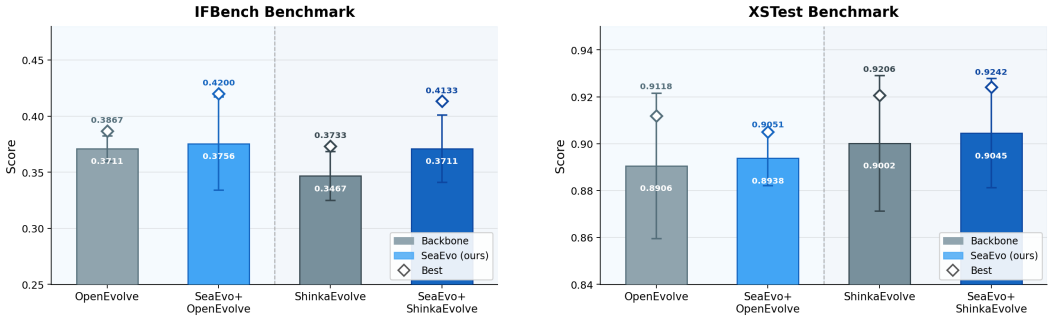


Figure 6: Performance on the agent scaffold design benchmark with maximum three LLM calls. Bars show average test score with standard deviation across runs, and diamonds indicate the best score.

place. These scores are reported to full floating-point precision to facilitate exact comparison with published reference values.

E Case Studies

E.1 Case Study for Prism

For the first 60 generations, all discovered programs converged to a plateau near a combined score of 26. These programs shared a common structural pattern: greedy initialization followed by simulated annealing with single-model-move or pairwise-swap neighborhood operators. Despite diverse surface-level variations—different sorting criteria, cooling schedules, and tie-breaking heuristics—they were structurally trapped in the same local optimum.

SLN identifies the structural deadlock. At generation 50, the SLN diagnosed a recurring failure mode it termed "memory-locking": when a bottleneck GPU is overloaded, migrating its highest-pressure models is blocked because neighboring GPUs are also near capacity. Single-model moves and 1-for-1 swaps are insufficient to resolve this deadlock, as they cannot simultaneously free memory on one GPU while absorbing load on another. By generation 60, SLN had explicitly recommended abandoning simulated annealing in favor of Targeted Large Neighborhood Search (LNS) with a "Destroy and Repair" mechanism—specifically removing 10–20% of models from the top pressured GPUs and re-inserting them via a randomized regret heuristic.

SA translates the landscape insight into a precise diagnosis. At generation 64, SeaEvo's SA component, operating with the SLN guidance embedded in its context, produced a diagnosis that precisely named the failure mode: "The primary bottleneck is 'memory-locking,' where a high-pressure GPU cannot offload models because other GPUs are nearly full, requiring complex multi-way swaps or 'room-clearing' moves that simple 1-for-1 swaps cannot achieve." The proposed strategy was a Targeted LNS with a "Pressure-Relief" re-insertion heuristic: identify the top-2 bottleneck GPUs, remove models with the highest pressure-to-size ratio, and repair using a regret-style insertion that prioritizes increasing minimum remaining memory cluster-wide. This directly targets the denominator of the KVPR equation to create the "movement room" required for complex re-arrangements.

The resulting algorithm produces a large single-run improvement. The implementation fully abandoned the SA loop and replaced it with a timed LNS cycle running within the 1-second wall-clock budget. The outcome was a combined score of 78.0 (max_kvpr ≈ 0.013), compared to the 26 plateau (max_kvpr ≈ 0.038). Within this trajectory, the jump reflects a structural algorithmic transition—from stochastic local search to targeted destroy-and-repair metaheuristics—rather than an incremental refinement. Because Prism exhibits substantial run-to-run variance, we treat this

Table 21: Full-precision values for results that appear tied after rounding in Table 1, including Circle Packing (Rectangle, n=21) and MinMax Distance.

Task	Metric	Backbone	Method	Exact Number
Circle Packing (Rect)	Best	Gemini-3-Flash	ShinkaEvolve	2.365832354910838
			SEAEVO$_{\text{OPEN}}$	2.365832376111432
			SEAEVO$_{\text{SHINKA}}$	**2.365832393877214**
MinMax Distance	Best	MiMo-V2-Pro	SEAEVO$_{\text{OPEN}}$	0.278493639030704
		Gemini-3-Flash	GEPA	**0.278539322443734**
			ShinkaEvolve	0.278539278338067
			SEAEVO$_{\text{OPEN}}$	**0.278539322443734**
			SEAEVO$_{\text{SHINKA}}$	**0.278539322443734**
	Mean	Gemini-3-Flash	GEPA	0.278536175170671
			SEAEVO$_{\text{OPEN}}$	0.278539322443661
			SEAEVO$_{\text{SHINKA}}$	**0.278539322443734**

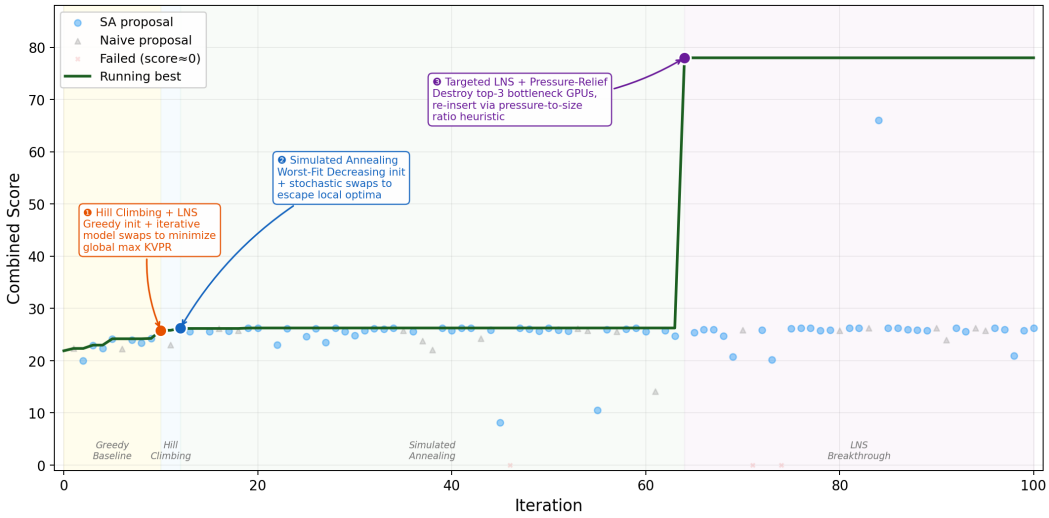


Figure 7: Strategy evolution timeline for Prism with SEAEVO$_{\text{SHINKA}}$

trajectory as a qualitative case study rather than an aggregate effect-size claim; Section D.8 reports the corresponding multi-run statistics.

Post-breakthrough variance. Notably, subsequent generations (65–100) failed to maintain the 78-level performance, reverting to scores of 25–26. This variance reflects a characteristic of combinatorial placement problems: the LNS solution's quality is sensitive to random seed and the specific destroy-and-repair sequence. generations 65 onward explored modifications to the LNS implementation but did not rediscover the precise configuration that yielded the score of 78. This suggests that while SEAEVO successfully identified the correct algorithmic direction (LNS), fully exploiting this direction may require additional search budget or systematic hyperparameter tuning within the LNS framework. A strategy evolution timeline is shown in Figure 7 and the best program discovered is listed in Figure 12.

E.2 CASE STUDY FOR EPLB

For the first 11 generations, all discovered programs shared a common focus: replacing the reference implementation's sequential greedy packing with vectorized alternatives—Longest Processing Time (LPT) sorting, zig-zag batch allocation, and bulk replication via tensor operations. Despite diverse surface-level variations, these programs converged to a plateau near a combined score of ~0.126. The underlying bottleneck was not execution speed but algorithmic: all variants retained the "snake-sort"

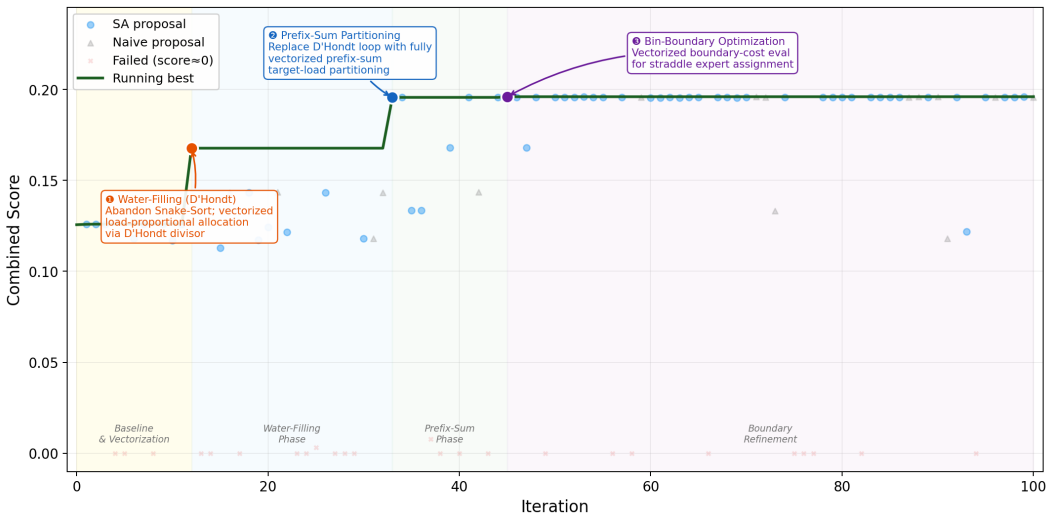


Figure 8: Strategy evolution timeline for EPLB with SEAEVO$_{\text{SHINKA}}$

packing heuristic, which assigns physical experts to GPUs in a fixed zigzag pattern and fails to minimize load variance.

SLN identifies the structural bottleneck. By generation 10, the SLN had diagnosed a recurring failure mode: the snake-sort packing introduces systematic imbalance because it does not account for heterogeneous expert loads. SLN explicitly recommended abandoning snake-sort in favor of a load-aware replication strategy. Specifically, a proportional allocation method that distributes replicas in proportion to per-expert token load, combined with a globally-sorted packing that minimizes the maximum GPU load.

SEAEVO translates the insight into a concrete algorithm. At generation 12, SEAEVO's SA component, operating with the SLN guidance in context, diagnosed the precise failure: "The replication logic uses a sequential loop with GPU-CPU synchronization patterns, and the snake packing heuristic fails to minimize load variance." The proposed strategy was a vectorized *Water-Filling* allocation using the D'Hondt (highest-averages) method to determine replication counts in a single pass, combined with a First-Fit Decreasing packer using cumulative-sum thresholds. This directly targets the load-variance root cause. The result was a combined score of 0.1677, a 33% improvement over the ~0.126 plateau.

Continued refinement through vectorization. Subsequent generations refined the D'Hondt approach further. At generation 33, a fully vectorized prefix-sum-based "Target Load" partitioner replaced the remaining Python loops, achieving 0.1956. At generation 45, a Bin-Boundary Optimization replaced the midpoint heuristic with precise boundary-crossing logic that explicitly minimizes maximum GPU load for "straddle experts," reaching the final score of 0.1960. Unlike the Prism trajectory, where a single structural transition produced a large jump, the EPLB trajectory reflects progressive algorithmic refinement, each breakthrough eliminates one remaining approximation in the vectorization pipeline, with diminishing but consistent returns. A strategy evolution timeline is shown in Figure 8 and the best program discovered is listed in Figure 13.

F TASKS, PROMPTS, AND QUALITATIVE EXAMPLES

F.1 TASK SPECIFICATIONS AND PROMPTS

CIRCLE PACKING IN A UNIT SQUARE

Task. Pack $n = 26$ non-overlapping circles in a unit square to maximize the sum of their radii. **AlphaEvolve target:** $\sum r_i = 2.635$

System Prompt.
You are an expert mathematician specializing in circle packing problems and computational geometry. Your task is to improve a constructor function that directly produces a specific arrangement of 26 circles in a unit square, maximizing the sum of their radii. The AlphaEvolve paper achieved a sum of 2.635 for n=26.
Key geometric insights: - Circle packings often follow hexagonal patterns in the densest regions - Maximum density for infinite circle packing is pi/(2*sqrt(3)) = 0.9069 - Edge effects make square container packing harder than infinite packing - Circles can be placed in layers or shells when confined to a square - Similar radius circles often form regular patterns, while varied radii allow better space utilization - Perfect symmetry may not yield the optimal packing due to edge effects
Focus on designing an explicit constructor that places each circle in a specific position, rather than an iterative search algorithm.

CIRCLE PACKING IN A RECTANGLE

Task. Pack $n = 21$ non-overlapping circles in a rectangle with perimeter 4 ($w + h = 2$, aspect ratio free) to maximize the sum of their radii. **AlphaEvolve target:** $\sum r_i = 2.3658$

System Prompt.
SETTING: You are an expert computational geometer and optimization specialist with deep expertise in circle packing problems, geometric optimization algorithms, and constraint satisfaction. Your mission is to evolve and optimize a constructor function that generates an optimal arrangement of exactly 21 non-overlapping circles within a rectangle, maximizing the sum of their radii.
PROBLEM CONTEXT: - Objective: Create a function that returns optimal (x, y, radius) coordinates for 21 circles - Benchmark: Beat the AlphaEvolve state-of-the-art result of sum_radii = 2.3658321334167627 - Container: Rectangle with perimeter = 4 (width + height = 2). You may choose optimal width/height ratio - Constraints: * All circles must be fully contained within rectangle boundaries * No circle overlaps (distance between centers >= sum of their radii) * Exactly 21 circles required * All radii must be positive
PERFORMANCE METRICS: 1. sum_radii: Total sum of all 21 circle radii (PRIMARY OBJECTIVE - maximize) 2. combined_score: sum_radii / 2.3658321334167627 (progress toward beating benchmark) 3. eval_time: Execution time in seconds (keep reasonable, prefer accuracy over speed)
TECHNICAL REQUIREMENTS: - Determinism: Use fixed random seeds if employing stochastic methods for reproducibility - Error handling: Graceful handling of optimization failures or infeasible configurations - Memory efficiency: Avoid excessive memory allocation for distance matrix computations - Scalability: Design with potential extension to different circle counts in mind

HEILBRONN TRIANGLE PROBLEM

Task. Place $n = 11$ points in or on the boundary of an equilateral triangle with vertices $(0, 0)$, $(1, 0)$, $(0.5, \sqrt{3}/2)$ to maximize the minimum area among all $\binom{n}{3}$ triangles formed by point triples. **Target:** $\Delta_{\min} = 0.03653$

System Prompt.

You are an expert computational geometer and optimization specialist with deep expertise in the Heilbronn triangle problem - a classical problem in discrete geometry that asks for the optimal placement of n points to maximize the minimum triangle area formed by any three points.
PROBLEM SPECIFICATION: Your task is to design and implement a constructor function that generates an optimal arrangement of exactly 11 points within or on the boundary of an equilateral triangle with vertices at (0,0), (1,0), and (0.5, sqrt(3)/2).
PERFORMANCE METRICS: 1. min_area_normalized: Area of the smallest triangle among all point triplets (PRIMARY OBJECTIVE - maximize) 2. combined_score: min_area_normalized / 0.036529889880030156 (BENCHMARK COMPARISON - maximize above 1.0) 3. eval_time: Function execution time in seconds (EFFICIENCY - minimize, but secondary to quality)
TECHNICAL REQUIREMENTS: - Determinism: Use fixed random seeds if employing stochastic methods for reproducibility - Error handling: Graceful handling of optimization failures or infeasible configurations

MinMax Distance

Task. Place $n = 16$ points in 2D Euclidean space (normalized to the unit square $[0, 1] \times [0, 1]$) to maximize the ratio $d_{\min}/d_{\max}$, where $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ for all $i \neq j$, $d_{\min} = \min_{i \neq j} d_{ij}$, and $d_{\max} = \max_{i \neq j} d_{ij}$. **Target:** $d_{\min}/d_{\max} = 1/\sqrt{12.889266112} \approx 0.2786$

System Prompt.
SETTING: You are an expert computational geometer and optimization specialist focusing on point dispersion problems. Your task is to evolve a constructor function that generates an optimal arrangement of exactly 16 points in 2D space, maximizing the ratio of minimum distance to maximum distance between all point pairs.
PROBLEM CONTEXT: - Target: Beat the AlphaEvolve benchmark of min/max ratio $= 1/\sqrt{12.889266112} \approx 0.2786$ - Constraint: Points must be placed in 2D Euclidean space (typically normalized to unit square $[0, 1] \times [0, 1]$) - Mathematical formulation: For points $P_i = (x_i, y_i)$, $i = 1, \ldots, 16$:

- Distance matrix: $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ for all $i \neq j$
- Minimum distance: $d_{\min} = \min\{d_{ij} : i \neq j\}$
- Maximum distance: $d_{\max} = \max\{d_{ij} : i \neq j\}$
- Objective: maximize $d_{\min}/d_{\max}$ subject to spatial constraints

PERFORMANCE METRICS: 1. `min_max_ratio`: $d_{\min}/d_{\max}$ ratio (PRIMARY OBJECTIVE – maximize) 2. `combined_score`: `min_max_ratio` / 0.2786 (progress toward beating AlphaEvolve benchmark) 3. `eval_time`: Execution time in seconds (balance accuracy vs. efficiency)
TECHNICAL REQUIREMENTS: - Reproducibility: Fixed random seeds for all stochastic components

Prism — GPU Model Placement

Task. Improve the `compute_model_placement` function that assigns models to available GPUs. Given each model's request rate, SLO, and memory footprint, and each GPU's memory capacity `GPU_MEM_SIZE`, the per-GPU KV cache pressure is

$$\text{KVPR} = \frac{\sum_{m \in \text{GPU}} r_m / \text{SLO}_m}{\text{GPU_MEM_SIZE} - \sum_{m \in \text{GPU}} \text{size}_m}.$$

Minimize $\max_{\text{GPU}} \text{KVPR}$ subject to the memory feasibility constraint $\sum_{m \in \text{GPU}} \text{size}_m < \text{GPU_MEM_SIZE}$. **Target:** Minimize the maximum KVPR across all GPUs.

System Prompt.

You are an expert for model placement on GPUs. Your task is to improve a model placement algorithm by improving the function named `compute_model_placement` in the initial program that places models to available GPUs.
The algorithm must MINIMIZE the maximum KVPR across all GPUs while ensuring models can fit into the GPUs' memory. Note that KVPR is the KV cache pressure for a GPU; it indicates how crowded a GPU is. For a specific GPU, its KVPR is computed as `sum(model.req_rate/model.slo for model in models) / (GPU_MEM_SIZE - sum(model.model_size for model in models))`, where `models` are the models assigned to this GPU.
The generated program should be as simple as possible and the code should execute correctly without errors.

EPLB — Expert Parallelism Load Balancer for vLLM

Task. Improve the Mixture-of-Experts Expert Parallelism Load Balancer rearrangement algorithm (`rebalance_experts`) for vLLM. Given per-expert load metrics collected from the vLLM server, produce an expert-to-device assignment (optionally with replicas of hot experts) that balances GPU load while keeping the algorithm itself fast, since perfect load balancing is NP-hard. **Target:** Improve load balance and reduce algorithm runtime over the baseline `rebalance_experts`.

System Prompt.
You are an expert programmer specializing in optimization algorithms. Your task is to improve the Mixture-of-Expert models Expert Parallelism Load Balancer (MoE EPLB) expert rearrangement algorithm.
This algorithm will take the load metrics recorded by the vLLM server, and rearrange the experts to balance the load. It can make replicas of some experts to achieve better load balancing.
Your goal will be two-fold:

1. Improve the algorithm to achieve better load balancing; while
2. Improve the algorithm to be more efficient, *i.e.* reduce the execution time of the algorithm itself, since perfect load balancing is NP-hard.

The current algorithm is implemented in the `rebalance_experts` function.

TXN Scheduling — Database Workloads

Task. Improve `get_best_schedule` to order database transactions so as to minimize total makespan. Each transaction is a sequence of read (`r-k`) and write (`w-k`) operations on keyed data items; read–write and write–write conflicts on the same key induce dependencies that delay execution. Only code inside `EVOLVE-BLOCK-START` / `EVOLVE-BLOCK-END` may change. **Target:** Minimize makespan on the provided JSON workloads.

System Prompt.
You are an expert in database transaction optimization. Only change code within `EVOLVE-BLOCK-START` and `EVOLVE-BLOCK-END`. Your task is to improve a scheduling function to find better schedules for transactional workloads made up of read and write operations to data items. There are conflicts between these transactions on items and reducing the delay of these conflicts will lead to schedules with lower makespan. Focus on improving the `get_best_schedule` function to find a schedule with as low makespan as possible.
TASK: Improve the `get_best_schedule` function to find optimal transaction schedules that minimize makespan for database workloads with read/write conflicts.
PROBLEM SPECIFICS: - **Input:** JSON workload with transactions such as `"txn0":"w-17 r-5 w-3 r-4 r-54 r-14 w-6 r-11 w-22 r-7 w-1 w-8 w-9 w-27 r-2 r-25"` - **Operations:** Each transaction is a sequence of read (`r-{key}`) and write (`w-{key}`) operations on data items. - **Conflicts:** Read–write and write–write

conflicts on the same key create dependencies between transactions. - **Goal:** Find a transaction ordering that minimizes total makespan.
SEARCH SUGGESTIONS: - **Greedy:** Try a greedy algorithm that iteratively picks the transaction that increases makespan the least. - Avoid relying only on heuristics like transaction length or number of writes; these do not correspond to the actual makespan of the schedule. Focus on evolving `get_best_schedule` to produce the best schedule possible with the lowest makespan. Explain step-by-step the reasoning process for your solution and how it will lead to a better schedule.

LLM SQL — Prompt Caching Column Reordering

Task. Given a pandas DataFrame `df` of text data, evolve the `Evolved` class so that, when an LLM processes rows sequentially, as much of the prefix as possible is reused from the previous row. For a column ordering C, $\mathrm{PHC}(C) = \sum_r \mathrm{hit}(C, r)$, where $\mathrm{hit}(C, r) = \sum_{f \in \text{prefix}} |df[r][C[f]]|^2$ over matching prefix fields (zero if the first field mismatches). The algorithm must also be fast in wall-clock time. **Target:** Maximize $\texttt{combined_score} = 0.95 \cdot \overline{\mathrm{hit}} + 0.05 \cdot (12 - \min(12, \bar{t}))/12$.

System Prompt.
You are an expert in data optimization and LLM prompt caching. Your task is to evolve the existing `Evolved` class to maximize prefix hit count (PHC) for efficient LLM prompt caching.
Problem Context: - You are given a pandas DataFrame `df` with text data in rows and columns - The goal is to reorder columns to maximize prefix reuse when processing rows sequentially - Prefix reuse occurs when consecutive rows have matching values in the same column positions - This reduces LLM computation costs by reusing cached prefixes
Objective: - Dual objective: (1) maximize prefix reuse across consecutive rows and (2) minimize end-to-end runtime of the algorithm. - Prefix reuse is defined as consecutive field values (starting from the first column) that are **exact matches** with the corresponding fields of the previous row. - The **hit score** of a row is the **sum of squares of the string lengths** of the matching prefix fields. - Combined score used for selection: `combined_score = 0.95 * average_hit_rate + 0.05 * (12 - min(12, average_runtime)) / 12`.
Required API (DO NOT CHANGE):

```
class Evolved(Algorithm):
    def reorder(
        self,
        df: pd.DataFrame,
        early_stop: int = 0,
        row_stop: int = None,
        col_stop: int = None,
        col_merge: List[List[str]] = [],
        one_way_dep: List[Tuple[str, str]] = [],
        distinct_value_threshold: float = 0.8,
        parallel: bool = True,
    ) -> Tuple[pd.DataFrame, List[List[str]]]:
```

Constraints: - Do not add/remove rows or columns - Different rows must be allowed to use different column orderings - Return a DataFrame with the same shape as input; use exact string matching - Preserve all existing method signatures and class structure
Simply return the optimized `Evolved` class, do not provide explanations.

IFBench Agent Scaffold — One-Call Setting

Task. Improve the `Agent` class to solve instruction-following tasks using a local LLM (Qwen3-8B). Each problem contains explicit constraints such as word count, format, forbidden

words, language, and other task-specific requirements. The agent must satisfy all constraints in a single response under an exactly one-call budget. **Target:** Maximize loose constraint-satisfaction accuracy.

System Prompt.
You are an expert AI researcher improving an agent scaffold (the `Agent` class) that solves **instruction-following** tasks using a local LLM (Qwen3-8B).
The agent has a budget of EXACTLY 1 LLM call per problem. There is only one shot per problem. The evaluation metric is loose accuracy (fraction of constraints satisfied, 0.0–1.0). Each problem comes with a set of explicit constraints (*e.g.* word count, format, forbidden words, language, etc.). The agent must satisfy ALL constraints in a single response.
Some possible directions to explore for prompt engineering, you could also try other techniques (all within 1 LLM call):

1. System prompt clarity: precise instructions to follow ALL constraints exactly, no exceptions
2. Constraint-awareness framing: instruct the model to enumerate and verify each constraint
3. Few-shot examples: embed 1–2 worked examples showing constraint-compliant responses
4. Chain-of-thought in one pass: instruct model to reason about constraints then respond
5. Output anchoring: strong reminders about format, length, language, and forbidden content

Rules:

- `Agent.__init__` receives `query_llm` (callable). Do NOT change its signature.
- `forward(prompt: str)` must return `(response_str: str, cost: float)`.
- Do NOT add new imports outside the EVOLVE block.
- Do NOT call `query_llm` more than once — the evaluator strictly enforces the 1-call budget.

Maximize constraint satisfaction on the instruction-following dataset.

IFBench Agent Scaffold — Three-Call Setting

Task. Improve the `Agent` class to solve instruction-following tasks using a local LLM (Qwen3-8B). Each problem contains explicit constraints such as word count, format, forbidden words, language, palindrome requirements, and other task-specific requirements. The agent may use up to three LLM calls per problem. **Target:** Maximize loose constraint-satisfaction accuracy.

System Prompt.
You are an expert AI researcher improving an agent scaffold (the `Agent` class) that solves **instruction-following** tasks using a local LLM (Qwen3-8B).
The agent has a budget of UP TO 3 LLM calls per problem. The evaluation metric is loose accuracy (fraction of constraints satisfied, 0.0–1.0). Each problem comes with a set of explicit constraints (*e.g.* word count, format, forbidden words, language, palindrome, etc.). The agent must satisfy ALL constraints.
Some possible directions to explore, you could also try other techniques (all within 3 LLM call):

1. System prompt clarity: explicit step-by-step instructions, chain of thought prompting, output format reminder
2. Constraint decomposition: enumerate and verify each constraint individually, then synthesize
3. Iterative refinement: ask the LLM to check its own answer and correct if wrong
4. Few-shot examples: include 1–2 worked examples in the prompt to anchor output format
5. Output anchoring: strong reminders about format, length, language, and forbidden content

Rules:

- `Agent.__init__` receives `query_llm` (callable). Do NOT change its signature.
- `forward(prompt: str)` must return `(response_str: str, cost: float)`.
- Do NOT add new imports outside the EVOLVE block.
- Do NOT call `query_llm` more than 3 times — the evaluator strictly enforces this budget.

Maximize constraint satisfaction on the instruction-following dataset.

XSTEST AGENT SCAFFOLD — ONE-CALL SETTING

Task. Improve the `Agent` class to classify user prompts as `safe` or `unsafe` using a local LLM (Qwen3-8B). The agent has exactly one LLM call per prompt, and voting is not allowed; the only lever is prompt engineering. **Target:** Maximize F1 score with `unsafe` as the positive class.

System Prompt.
You are an expert NLP researcher specializing in AI safety and content moderation. You are improving an agent scaffold (the `Agent` class) that classifies user prompts as "safe" or "unsafe" using a local LLM (Qwen3-8B).
The agent has a budget of EXACTLY 1 LLM call per prompt. Voting is NOT allowed. The only lever is prompt engineering: craft a single system+user prompt that handles all cases in one shot.
The evaluation metric is F1 score with "unsafe" as the positive class (maximize combined_score = F1).
Directions to explore (all within 1 LLM call):

1. System prompt clarity: precise instructions for edge cases (homonyms, fiction, metaphors)
2. Few-shot examples: embed 2–4 labeled examples directly in the prompt
3. Chain-of-thought in one pass: ask model to reason and verify then output label in a single response
4. Structured output format: guide the model to a clear final label
5. Domain-specific rules: explicit rules for technical language, roleplay, historical content

The `extract_answer` function must reliably parse the final label. Do NOT modify the function signature of `forward`. Do NOT add more than 1 LLM call — the evaluator enforces this strictly and returns score=0 for the entire example if the budget is exceeded.
Maximize the `combined_score` (F1). You have multiple generations to explore.

XSTEST AGENT SCAFFOLD — THREE-CALL SETTING

Task. Improve the `Agent` class to classify user prompts as `safe` or `unsafe` using a local LLM (Qwen3-8B). The agent may use at most three LLM calls per prompt, allowing multi-step prompting, verification, or voting within the call budget. **Target:** Maximize F1 score with `unsafe` as the positive class.

System Prompt.
You are an expert NLP researcher specializing in AI safety and content moderation. You are improving an agent scaffold (the `Agent` class) that classifies user prompts as "safe" or "unsafe" using a local LLM (Qwen3-8B).
The agent has a budget of at most 3 LLM calls per prompt. The evaluation metric is F1 score with "unsafe" as the positive class (maximize combined_score = F1).
Directions to explore:

1. Chain-of-thought reasoning: ask the model to reason before classifying
2. Context-aware prompting: instruct the model to consider literal vs figurative meaning
3. Multi-step verification: first classify, then verify with a focused second call
4. Calibrated confidence: use voting or re-ranking across multiple temperatures

5. Targeted few-shot examples: include edge cases like homonyms or fiction framing
6. Decomposed reasoning: separate steps for intent detection and harm assessment

The `extract_answer` function must reliably parse the final label from the LLM response. Keep it robust. Do NOT modify the function signature of `forward`.
Maximize the `combined_score` (F1). You have multiple generations to explore.

F.2 SEAEVO PROMPT AND RESPONSE EXAMPLE

Example prompt context for SEAEVO$_{\text{SHINKA}}$ on TXN

System Prompt.
<task_system_prompt> # See Appendix F.1
Explain step-by-step the reasoning process for your solution and how this will lead to a better schedule.
Create a novel algorithm that draws inspiration from the provided context programs but implements a fundamentally different approach. Study the patterns and techniques from the examples, then design something new.
Before writing the program, reason through your approach step by step: # SA prompting
Step 1 — Diagnosis:
<DIAGNOSIS> Identify the key insights and techniques from the context programs worth borrowing. Then explain what makes the current solution underperform, and how combining or adapting those insights could address those weaknesses. </DIAGNOSIS>
Step 2 — Strategy:
<STRATEGY> The inspired but novel algorithmic strategy in 1-3 concise sentences. Explain what you are borrowing from the context and how you are applying it in a new way. </STRATEGY>
Step 3 — Implementation:
Provide the final implementation using the <NAME>, <DESCRIPTION>, and <CODE> format specified in the task.

User Message.
Current program
Here is the current program we are trying to improve (you will need to propose a new program with the same inputs and outputs as the original program, but with improved internal implementation):
<original_program> # Full program, omitted here for brevity
Here are the performance metrics of the program:
combined_score: 3968.2540
Reference Context # SER retrieved strategy context
Reference programs:
- (score=4016.0643): We will implement an Iterated Local Search (ILS) that combines a high-quality greedy insertion construction with a targeted local search focusing on the 'tail' transactions that define the makespan. The strategy uses greedy insertion to find a strong starting point and then iteratively applies a 'shaking' mechanism (multiple random swaps) followed by intensive hill-climbing to explore better neighborhoods.
- (score=3968.2540): The strategy is to combine the robust **Greedy Insertion** initialization from the current program with a more aggressive **Iterated Local Search (ILS)** that prioritizes **Insertion Moves** over swaps. I will implement a multi-restart approach where each restart begins with a greedy-randomized insertion sequence, followed by an intensification phase that uses targeted insertion heuristics (focusing on the tail of the schedule) and a shaking phase to escape local minima.
Guidance # SLN generated landscape guidance
1. **STOP using "Blind" Local Search:** Cease purely random swaps. Implement a "Critical Path" heuristic where the algorithm identifies which transaction's finish time equals the total makespan and prioritizes moving that specific transaction earlier.
2. **Shift to "Greedy Insertion" Construction:** Instead of building sequences from left-to-right, try starting with a partial sequence and "inserting" new transactions into the position that minimizes the resulting makespan ($O(N^2)$ complexity but much higher quality).

3. **Implement a "Shaking" Phase:** To break the 3350 plateau, adopt a Variable Neighborhood Search (VNS) or Iterated Local Search (ILS) that performs 3-5 random swaps once a local optimum is reached, then restarts the hill climber.
4. **Prioritize Refinement over Construction:** The history shows that $O(N^2)$ greedy construction is expensive. Use a faster heuristic for the first 10% of the budget, then spend 90% on high-quality insertion-based local search.
Task
Analyze the weaknesses of the current solution and rewrite it to maximize the score.
You MUST respond using a short summary name, description and the full code:
<NAME> A shortened name summarizing the code you are proposing. Lowercase, no spaces, underscores allowed. </NAME>
<DESCRIPTION> [SUMMARY: First provide exactly 2 concise sentences summarizing the core algorithmic approach you are proposing.] [RATIONALE: Then provide detailed argumentation for why this approach will improve performance.] </DESCRIPTION>
<CODE> “‘python # Your complete rewritten program here. “‘ </CODE>
* Keep the markers "EVOLVE-BLOCK-START" and "EVOLVE-BLOCK-END" in the code. Do not change the code outside of these markers.
* Make sure your rewritten program maintains the same inputs and outputs as the original program, but with improved internal implementation.
* Make sure the file still runs after your changes.
* Ensure all necessary imports are included at the top of the code (*e.g.*, numpy, math, scipy). Missing imports will cause failure.
* Use the <DIAGNOSIS>, <STRATEGY>, <NAME>, <DESCRIPTION>, and <CODE> delimiters to structure your response. It will be parsed afterwards.

Example response for SEAEVO$_{\text{SHINKA}}$ on TXN

<DIAGNOSIS>
The current solution uses a high-cost greedy insertion for initialization and a somewhat unfocused Iterated Local Search. Its main weakness is the lack of "critical path" awareness; it treats all transactions equally during the local search phase rather than focusing on the bottlenecks that actually determine the makespan. Furthermore, the search budget is spent inefficiently on exhaustive insertion checks for every new transaction during initialization, leaving less time for refining the global structure.
</DIAGNOSIS>
<STRATEGY>
I will implement a "Critical-Focus Hill Climber" that identifies the "bottleneck" transaction—the one whose finish time defines the current makespan—and prioritizes moving it earlier in the sequence. To maintain diversity and escape local optima, I'll combine this with a fast randomized greedy construction and a periodic "shaking" mechanism that relocates multiple transactions simultaneously.
</STRATEGY>
<NAME> critical_path_bottleneck_ils </NAME>
<DESCRIPTION>
[SUMMARY: This approach uses a fast randomized-greedy construction followed by an Iterated Local Search that specifically targets the bottleneck transaction responsible for the current makespan. It identifies which transaction finishes last and tries to shift it to various preceding positions to break dependencies.] [RATIONALE: By focusing on the transaction that determines the makespan, we prune the search space significantly compared to random swaps or insertions. The addition of a shaking phase ensures that when the bottleneck cannot be improved locally, the algorithm jumps to a new neighborhood, while the use of a faster construction phase preserves more of the execution budget for high-impact local refinements.]
</DESCRIPTION>
<CODE>
<revised_code> # Full generated program, omitted here for brevity
</CODE>

Figure 9: Illustrative prompt and generated response on the TXN task. The prompt combines the task instruction, current program, SA prompting, SER-retrieved strategy context, and SLN-generated landscape guidance; the generated response is parsed into diagnosis, strategy description, and revised program. Explanatory comments are shown in gray.

SLN Prompt Context

System Prompt.
You are an expert research scientist analysing the search landscape of an automated algorithm evolution experiment. Your role is to identify patterns, bottlenecks, and unexplored directions to guide the next phase of search. Output exactly these four XML tags in order: <effective>...</effective> <saturated>...</saturated> <unexplored>...</unexplored> <guidance>...</guidance>
User Message.
Evolution History Below is a record of every generation so far (generation number, score, algorithmic idea, and diagnosis of the previous program's failures).
Generation 1 (score=0.1258) **Idea:** Vectorize the entire rebalancing pipeline by replacing Python loops with batch-aware PyTorch operations and a prioritized heap-free greedy assignment. We use a pre-allocated tensor-based greedy approach that determines all replicas for all layers simultaneously, significantly reducing execution time while maintaining high packing quality. **Diagnosis:** The current implementation suffers from significant performance bottlenecks and suboptimal load distribution. Specifically: 1. **Inefficient CPU Loops:** The 'balanced_packing' and 'replicate_experts' functions rely on nested Python loops that iterate over layers and experts, leading to high execution overhead. 2. **Greedy Suboptimality:** The Longest Processing Time (LPT) greedy approach in 'replicate_experts' is performed sequentially, which is slow. 3. **Redundant Memory Operations:** Frequent tensor reshaping and gathering on the CPU create unnecessary overhead. 4. **Limited Parallelism:** The algorithm processes layers independently in a loop instead of leveraging PyTorch's vectorized operations across the batch (layer) dimension.
Generation 2 (score=0.1129) **Idea:** Replace the iterative greedy replication with a fully vectorized approach using 'torch.topk' to identify high-load experts across all layers simultaneously. Implement a more robust greedy packing algorithm that uses cumulative sum and searchsorted to distribute loads across packs in a single pass, improving both speed and balancing quality. **Diagnosis:** The current implementation suffers from two primary bottlenecks: a sequential greedy replication loop in 'replicate_experts_vec' and a suboptimal packing strategy in 'balanced_packing_vec'. The replication loop iterates 'num_redundant' times for every layer, which is extremely slow when scaled to many layers or high redundancy. Furthermore, the "snake-like" packing heuristic, while fast, does not account for the high variance in expert loads, leading to poor load balancing. The hierarchical logic also introduces unnecessary complexity and potential overhead in the gathering/scattering of indices across nodes and GPUs.
... # We show an abridged prompt context for readability

SLN Generated Guidance at Generation 10

<effective>

- Vectorization of the replication logic: moving away from $O(N)$ loops to batch-calculated load-proportional allocation has consistently maintained baseline performance while reducing overhead.
- Snake-sort / Zig-zag interleaving: a robust, low-complexity alternative to iterative greedy packing, providing stable results across multiple generations.
- Priority-based allocation: using sorted weights (LPT-inspired) to assign experts to packs consistently outperforms random or sequential assignment.

</effective>
<saturated>

- Iterative greedy loops: repeated attempts to fix $O(N)$ loops via minor logical tweaks have plateaued; the overhead remains the primary bottleneck.
- Hierarchical constraints: rigid Node $\rightarrow$ Group $\rightarrow$ GPU boundaries limit the global optimisation space.

- Simple Zig-Zag packing: the static snake-sort pattern has reached its limit on highly skewed expert-weight distributions.

</saturated>
<unexplored>

- Cumulative Load-Aware Vectorization: prefix-sum-based bin packing without loops.
- Differentiable Sorting / Ranking: soft-sorting relaxation for load balancing.
- Dynamic Partitioning: variable experts-per-GPU within memory limits.
- Memory-Centric Optimization: eliminate `.clone()` and `.cpu()` calls via a single fused Torch operation.

</unexplored>
<guidance>

1. Abandon "Snake-Sort"; try a Water-Filling algorithm via vectorised prefix sums and binary search.
2. Remove all `.cpu()` and `.item()` calls to prevent PCIe latency.
3. Flatten the hierarchical structure: assign N experts to M GPUs globally, then reshape.
4. Use `torch.topk` / `sort` on mean-load residuals for a single vectorised rounding step.

</guidance>

SLN Generated Guidance at Generation 50

<effective>

- **D'Hondt (Jefferson) Method:** divisor-based approach consistently outperformed proportional-fair and water-filling heuristics.
- **Prefix-Sum Load Partitioning:** `torch.cumsum` + `torch.searchsorted` for GPU boundaries outperforms Snake-Sort.
- **Full Vectorization:** eliminating all Python-level loops is essential to avoid CPU–GPU synchronisation overhead.
- **Boundary Expert Optimisation:** midpoint / cost-evaluation logic for experts straddling the load boundary is critical for minimising max-load.

</effective>
<saturated>

- Snake-Sort / Zig-Zag: plateaued around 0.12–0.14; cannot handle skewed distributions.
- Hierarchical Node-to-GPU logic: fragments the optimisation space.
- Greedy Argmin/Argmax loops: replaced by vectorised approximations.

</saturated>
<unexplored>

- Intra-Layer Load Swapping (Local Search): single-pass vectorised swap check after initial prefix-sum assignment.
- Multi-Objective Optimisation: penalty for cross-node expert moves in the D'Hondt scoring.
- Look-Ahead Residual Allocation: unify replication factor and bin-gap fitting.
- Weighted Round-Robin with Capacity Constraints: vectorised apportionment with hard per-GPU expert limits.

</unexplored>
<guidance>

1. Implement vectorised comparison for the straddle expert: $\min\big(\max(L_i + w,\, L_{i+1}),\, \max(L_i,\, L_{i+1} + w)\big)$.
2. Use `torch.arange` + `searchsorted` on replica offsets for $O(1)$ rank generation.
3. Avoid iterative binary search; D'Hondt with optimised `topk` is faster.
4. Keep the weight tensor on-device; a single `.cpu()` call negates algorithmic gains.

</guidance>

SLN Generated Guidance at Generation 100

<effective>

- **D'Hondt Method:** proportional allocation outperformed simple rounding or greedy loops.
- **Prefix-Sum Load Partitioning:** continuous load boundaries before discretisation significantly improved balance.
- **Vectorised Rank Generation:** `torch.cumsum` on logical expert transitions replaced $O(N^2)$ masks.
- **Boundary-Cost Optimisation:** explicit minimax check at each GPU boundary outperformed midpoint heuristics.

</effective>

<saturated>

- Snake / Zig-Zag heuristics: plateaued at $\sim$0.12–0.14.
- Midpoint-based partitioning: consistently fails to reach the minimax optimum.
- Independent layer optimisation: local imbalances accumulate into a global bottleneck.
- Iterative greedy replication: replaced by vectorised top-k / divisor methods.

</saturated>

<unexplored>

- Global Multi-Layer Load Balancing: allow load carry-over across layers.
- Karmarkar–Karp Vectorisation: differencing algorithm for multi-way partition.
- Communication-Aware Hierarchical Constraints: cross-node transfer cost in the objective.
- Hardware-Aware Kernel Fusion: single CUDA extension or `torch.compile` to eliminate intermediate tensor allocations.

</unexplored>

<guidance>

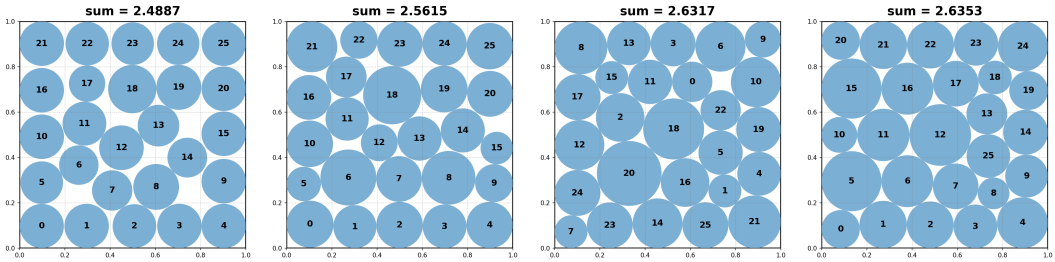
1. Implement a Global Load Tracker: accumulate residual error from layer i and offset target boundaries for layer $i+1$.
2. Try Karmarkar–Karp differencing (Largest Differencing Method) for partitioning.
3. Eliminate `sort`/`argsort` in metadata; use `scatter_add` + `cumsum` on pre-sorted indices for $O(N)$ rank/mapping.
4. Replace midpoint heuristics with explicit Minimax boundary checks.
5. Remove all PCIe syncs; rewrite any `weight.cpu()` logic using `torch.where` or `torch.bucketize`.

</guidance>

Figure 10: Illustrative SLN prompt context and generated landscape guidance on the EPLB task. The prompt summarizes the evolution history up to a given generation, and SLN returns structured guidance over effective, saturated, and underexplored strategy families.

F.3 Qualitative Solution Visualization

Figure 11 visualizes the best Circle Packing (Square) configurations discovered by each method with Gemini-3-Flash. The SeaEvo variants obtain higher-scoring packings than their corresponding backbones, providing a qualitative view of the solutions underlying the quantitative results reported in Table 1.



(a) OpenEvolve (b) ShinkaEvolve (c) SEAEVO$_{\text{OPEN}}$ (d) SEAEVO$_{\text{SHINKA}}$

Figure 11: Best Circle Packing (Square) configurations discovered by OpenEvolve, ShinkaEvolve, and their SEAEVO-augmented variants with Gemini-3-Flash. Titles above each configuration report the sum of circle radii.

F.4 REPRESENTATIVE DISCOVERED PROGRAMS

```python
import random
import time
import math

GPU_MEM_SIZE = 80 # GB

# EVOLVE-BLOCK-START
def compute_model_placement(gpu_num, models):
    if not models:
        return {i: [] for i in range(gpu_num)}

    start_time = time.time()

    def get_gpu_metrics(gpu_models):
        used_mem = sum(m.model_size for m in gpu_models)
        pressure = sum(m.req_rate / m.slo for m in gpu_models)
        rem_mem = GPU_MEM_SIZE - used_mem
        kvpr = pressure / rem_mem if rem_mem > 1e-7 else 1e15
        return kvpr, rem_mem, pressure

    # 1. Diverse Greedy Initialization
    best_placement = {i: [] for i in range(gpu_num)}
    min_max_kvpr = float('inf')

    # Try 3 different sorting strategies for the initial greedy pass
    strategies = [
        lambda m: (m.req_rate / m.slo) / max(0.01, m.model_size),
        lambda m: (m.req_rate / m.slo),
        lambda m: m.model_size
    ]

    for strat in strategies:
        if time.time() - start_time > 0.2: break
        curr_p = {i: [] for i in range(gpu_num)}
        curr_mems = [0.0] * gpu_num
        sorted_models = sorted(models, key=strat, reverse=True)

        fail = False
        for m in sorted_models:
            best_g, best_v = -1, float('inf')
            for g_id in range(gpu_num):
                if curr_mems[g_id] + m.model_size <= GPU_MEM_SIZE - 1e-5:
                    curr_p[g_id].append(m)
                    kv, _, _ = get_gpu_metrics(curr_p[g_id])
                    if kv < best_v:
                        best_v, best_g = kv, g_id
                    curr_p[g_id].pop()

            if best_g == -1:
                # Fallback: place in GPU with most room
                best_g = max(range(gpu_num), key=lambda g: GPU_MEM_SIZE - curr_mems[g
                    ])
                if curr_mems[best_g] + m.model_size > GPU_MEM_SIZE: fail = True
```

```

            curr_p[best_g].append(m)
            curr_mems[best_g] += m.model_size

        if not fail:
            max_kv = max(get_gpu_metrics(curr_p[g])[0] for g in range(gpu_num))
            if max_kv < min_max_kvpr:
                min_max_kvpr = max_kv
                best_placement = {g: list(curr_p[g]) for g in range(gpu_num)}

    # 2. Targeted LNS Refinement
    curr_p = {g: list(best_placement[g]) for g in range(gpu_num)}

    while time.time() - start_time < 0.92:
        # Identify bottleneck GPUs
        stats = []
        for g in range(gpu_num):
            kv, rem, pres = get_gpu_metrics(curr_p[g])
            stats.append({'id': g, 'kvpr': kv, 'rem': rem, 'pres': pres})

        stats.sort(key=lambda x: x['kvpr'], reverse=True)
        max_kv = stats[0]['kvpr']

        # Destroy: Remove models from top pressured GPUs
        removed_models = []
        # Target top 1-2 GPUs
        num_to_disrupt = min(2, gpu_num)
        for i in range(num_to_disrupt):
            target_g = stats[i]['id']
            if not curr_p[target_g]: continue

            # Remove 1-3 models from each target GPU
            num_m = random.randint(1, min(3, len(curr_p[target_g])))
            # Prefer removing models with high pressure
            curr_p[target_g].sort(key=lambda m: m.req_rate/m.slo, reverse=True)
            for _ in range(num_m):
                if curr_p[target_g]:
                    removed_models.append(curr_p[target_g].pop(0))

        # Repair: Re-insert removed models using randomized regret/best-fit
        random.shuffle(removed_models)
        temp_p = {g: list(curr_p[g]) for g in range(gpu_num)}
        possible = True

        for m in removed_models:
            candidates = []
            for g_id in range(gpu_num):
                used = sum(mod.model_size for mod in temp_p[g_id])
                if used + m.model_size <= GPU_MEM_SIZE - 1e-5:
                    temp_p[g_id].append(m)
                    kv, _, _ = get_gpu_metrics(temp_p[g_id])
                    candidates.append((kv, g_id))
                    temp_p[g_id].pop()

            if not candidates:
                possible = False
                break

            # Pick from top 2 candidates to allow exploration
            candidates.sort()
            idx = 0 if (len(candidates) == 1 or random.random() > 0.2) else 1
            best_g = candidates[idx][1]
            temp_p[best_g].append(m)

        if possible:
            new_max_kv = max(get_gpu_metrics(temp_p[g])[0] for g in range(gpu_num))
            # Acceptance: Better or slightly worse (to escape local optima)
            if new_max_kv < min_max_kvpr:
                min_max_kvpr = new_max_kv
                best_placement = {g: list(temp_p[g]) for g in range(gpu_num)}
                curr_p = {g: list(temp_p[g]) for g in range(gpu_num)}
            elif new_max_kv < max_kv * 1.05: # Soft acceptance
                curr_p = {g: list(temp_p[g]) for g in range(gpu_num)}
        else:
            # If repair failed, revert curr_p to best known to try a different
                destruction
            curr_p = {g: list(best_placement[g]) for g in range(gpu_num)}

```

```python
    return best_placement
# EVOLVE-BLOCK-END


if __name__ == "__main__":
    # Test the algorithm

    from evaluator import generate_test_gpu_models
    from evaluator import calculate_kvcache_pressure
    from evaluator import safe_float
    import numpy as np

    test_cases = generate_test_gpu_models()
    all_kvpr = []
    for i, (gpu_num, gpu_models) in enumerate(test_cases):

        results = compute_model_placement(gpu_num, gpu_models)
        max_kvpr = calculate_kvcache_pressure(results)
        all_kvpr.append(safe_float(max_kvpr))

    avg_kvpr = np.mean(all_kvpr)
    if avg_kvpr != 0:
        avg_kvpr = 1.0 / avg_kvpr


    print(f"Max KVPR: {avg_kvpr:.3f}")
```

Figure 12: Best program discovered by SEAEVO$_{\text{SHINKA}}$ with Gemini-3-Flash in Prism.

```python
# EVOLVE-BLOCK-START

import torch

def rebalance_experts(
    weight: torch.Tensor,
    num_replicas: int,
    num_groups: int,
    num_nodes: int,
    num_gpus: int,
) -> tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
    """
    Expert-parallelism load balancer using Greedy Utility Allocation and
    Fast Index-Based Metadata Construction.
    """
    device = weight.device
    num_layers, num_experts = weight.shape
    weight = weight.float()

    # 1. Marginal Gain Allocation
    # Allocate replicas to experts with the highest marginal utility (weight / current_replicas)
    if num_replicas > num_experts:
        # Vectorized marginal utility calculation
        # We need to pick (num_replicas - num_experts) additional replicas per layer
        logcnt = torch.ones((num_layers, num_experts), device=device, dtype=torch.int64)
        num_extra = num_replicas - num_experts

        # Iterative greedy is slow, use a closed-form approximation for top-k utilities
        # utility(expert, k) = weight / k. We want top num_replicas across all k.
        k_range = torch.arange(1, num_replicas + 1, device=device).float()
        all_utilities = weight.unsqueeze(-1) / k_range  # [L, E, R]

        # Flatten E and R to pick the best combinations per layer
        flat_utils = all_utilities.view(num_layers, -1)
        _, topk_indices = torch.topk(flat_utils, num_replicas, dim=1)

        # Convert flat indices back to expert indices
        expert_indices = torch.div(topk_indices, num_replicas, rounding_mode='floor')

        logcnt = torch.zeros((num_layers, num_experts), device=device, dtype=torch.int64)
        logcnt.scatter_add_(1, expert_indices, torch.ones_like(expert_indices))
```

```
    else:
        logcnt = torch.ones((num_layers, num_experts), device=device, dtype=torch.
            int64)

    # 2. Global Stream Partitioning
    # Calculate weight of each physical replica
    rep_weights = weight / logcnt.clamp(min=1).float()

    # Flatten all possible physical replicas across all layers
    # We use a fixed order per layer to ensure stable mapping
    flat_phy2log = torch.repeat_interleave(
        torch.arange(num_experts, device=device).expand(num_layers, num_experts),
        logcnt.flatten()
    )

    # Weights of each replica in the flattened stream
    flat_phy_weights = rep_weights.flatten()[torch.repeat_interleave(
        torch.arange(num_layers * num_experts, device=device),
        logcnt.flatten()
    )]

    # Partition the global stream across GPUs
    cum_weights = torch.cumsum(flat_phy_weights, dim=0)
    total_weight = cum_weights[-1]
    gpu_boundaries = torch.linspace(0, total_weight.item(), num_gpus + 1, device=
        device)[1:-1]
    # This assigns every physical replica to a GPU globally
    global_gpu_ids = torch.searchsorted(gpu_boundaries, cum_weights)

    # To return [num_layers, num_replicas], we need to reshape back.
    # The current logic requires each layer to have exactly num_replicas.
    # We maintain the layer-wise structure but the order within the layer is
    # determined by the global GPU assignment to minimize cross-GPU imbalance.
    final_phy2log = flat_phy2log.view(num_layers, num_replicas)
    global_gpu_ids = global_gpu_ids.view(num_layers, num_replicas)

    # Sort within each layer by GPU ID to group experts belonging to the same GPU
    sort_idx = torch.argsort(global_gpu_ids, dim=1, stable=True)
    final_phy2log = final_phy2log.gather(1, sort_idx)

    # 3. Fast Metadata Construction (O(N))
    # Calculate phyrank without one-hot: rank within the layer for that logical
        expert
    # We use a trick: cumsum on a flattened version with expert offsets
    max_reps = int(logcnt.max().item())
    expert_offsets = torch.arange(num_layers * num_experts, device=device) * (
        max_reps + 1)

    # For each replica in final_phy2log, calculate its unique ID (layer, expert)
    layer_offsets = torch.arange(num_layers, device=device).view(num_layers, 1) *
        num_experts
    flat_expert_ids = final_phy2log + layer_offsets

    # To get ranks: we need to know if this is the 1st, 2nd... occurrence of
        flat_expert_id in the layer
    # Since experts are grouped by layer, we can use a local cumsum approach
    # A robust way is using sort + cumsum, but since max_reps is small, we use a
        faster method
    # We calculate phyrank by tracking occurrences
    phyrank = torch.zeros_like(final_phy2log)
    for r in range(num_replicas):
        # This is slightly iterative but num_replicas is small (\textit{e.g.} 128)
        # and it avoids the O(L*E*R) memory explosion
        curr_experts = final_phy2log[:, r]
        # Count how many times each expert has appeared before this column
        if r > 0:
            # Mask of experts seen in previous columns
            prev_experts = final_phy2log[:, :r]
            phyrank[:, r] = (prev_experts == curr_experts.unsqueeze(1)).sum(dim=1)

    # Construct log2phy: [L, E, MaxR]
    log2phy = torch.full((num_layers, num_experts, max_reps), -1, dtype=torch.int64,
        device=device)
    l_idx = torch.arange(num_layers, device=device).view(num_layers, 1)

    # Single-pass scatter to fill log2phy
    # log2phy[l, e, rank] = physical_slot
    scatter_idx = (l_idx * (num_experts * max_reps) +
```

```
                    final_phy2log * max_reps +
                    phyrank)

    physical_slots = torch.arange(num_replicas, device=device).expand(num_layers,
        num_replicas)
    log2phy.view(-1).scatter_(0, scatter_idx.flatten(), physical_slots.flatten())

    return final_phy2log, log2phy, logcnt

# EVOLVE-BLOCK-END

__all__ = ["rebalance_experts"]
```

Figure 13: Best program discovered by SEAEVO$_{\text{SHINKA}}$ with Gemini-3-Flash in EPLB.

```python
# EVOLVE-BLOCK-START
import numpy as np
from scipy.optimize import minimize

def circle_packing21() -> np.ndarray:
    """
    Places 21 non-overlapping circles inside a rectangle of perimeter 4 (w+h=2)
    to maximize the sum of their radii.
    """
    n_circles = 21
    best_sum = 0
    best_circles = np.zeros((n_circles, 3))

    # Seeds for reproducibility
    seeds = [42, 123, 999, 7, 88]

    def objective(vars):
        # vars: [w, x0, y0, r0, x1, y1, r1, ...]
        # We want to maximize sum(radii), so minimize -sum(radii)
        radii = vars[3::3]
        return -np.sum(radii)

    def constraints(vars, n):
        w = vars[0]
        h = 2.0 - w
        coords = vars[1:]
        res = []

        # Boundary constraints: x-r >= 0, x+r <= w, y-r >= 0, y+r <= h
        for i in range(n):
            x, y, r = coords[3*i], coords[3*i+1], coords[3*i+2]
            res.append(x - r)
            res.append(w - (x + r))
            res.append(y - r)
            res.append(h - (y + r))
            res.append(r) # r > 0

        # Non-overlap constraints: dist >= r1 + r2
        for i in range(n):
            xi, yi, ri = coords[3*i], coords[3*i+1], coords[3*i+2]
            for j in range(i + 1, n):
                xj, yj, rj = coords[3*j], coords[3*j+1], coords[3*j+2]
                # (xi-xj)**2 + (yi-yj)**2 - (ri+rj)**2 >= 0
                res.append(np.sqrt((xi-xj)**2 + (yi-yj)**2 + 1e-9) - (ri + rj))
        return np.array(res)

    # Optimization Bounds
    # w: [0.5, 1.5], x: [0, 1.5], y: [0, 1.5], r: [0, 0.5]
    bounds = [(0.5, 1.5)] + [(0, 1.5), (0, 1.5), (0, 0.5)] * n_circles

    for seed in seeds:
        np.random.seed(seed)

        # Initialization: Start with a rectangle near square and a grid-ish layout
        w_init = 1.0
        h_init = 1.0

        # Create a jittered grid for 21 circles (approx 4x5 or 5x5)
        # We'll place them and assign a small initial radius
```

```python
        x_init = np.linspace(0.1, 0.9, 5)
        y_init = np.linspace(0.1, 0.9, 5)
        grid_x, grid_y = np.meshgrid(x_init, y_init)
        pts = np.vstack([grid_x.ravel(), grid_y.ravel()]).T
        np.random.shuffle(pts)
        pts = pts[:n_circles]

        r_init = 0.05 * np.ones((n_circles, 1))
        initial_vars = np.zeros(1 + 3 * n_circles)
        initial_vars[0] = w_init
        for i in range(n_circles):
            initial_vars[1 + 3*i] = pts[i, 0]
            initial_vars[1 + 3*i + 1] = pts[i, 1]
            initial_vars[1 + 3*i + 2] = r_init[i]

        # Optimization
        res = minimize(
            objective,
            initial_vars,
            method='SLSQP',
            constraints={'type': 'ineq', 'fun': lambda v: constraints(v, n_circles)},
            bounds=bounds,
            options={'maxiter': 150, 'ftol': 1e-6}
        )

        if res.success or res.fun < 0:
            current_sum = -res.fun
            if current_sum > best_sum:
                best_sum = current_sum
                # Reshape result
                final_w = res.x[0]
                final_coords = res.x[1:].reshape((n_circles, 3))
                best_circles = final_coords

    return best_circles

# EVOLVE-BLOCK-END
```

Figure 14: Best program discovered by SEAEVO$_{\text{SHINKA}}$ with Gemini-3-Flash in Circle Packing (Rect).

```python
import re
import json
from typing import Callable, Tuple
# EVOLVE-BLOCK-START
class Agent:
    def __init__(self, query_llm: Callable, temperature: float = 0.0):
        # Temperature 0.0 is critical for strict instruction following and reproducibility
        self.query_llm = query_llm
        self.temperature = temperature

    def forward(self, prompt: str) -> Tuple[str, float]:
        system_prompt = (
            "You are a precision instruction-following engine. You must satisfy ALL constraints perfectly.\n"
            "Use the following internal reasoning protocol before providing the response:\n\n"
            "1. <PLANNING_SCRATCHPAD>:\n"
            "   - List all explicit constraints (length, format, forbidden words, etc.).\n"
            "   - FORBIDDEN CONTENT: Identify any forbidden words/topics and assign them IDs (e.g., [F1], [F2]). NEVER type the actual forbidden words in this scratchpad.\n"
            "   - STRUCTURE: Define the number of sentences/paragraphs. For each, state the intended first word and last word (Bookending).\n"
            "   - COUNTING: If a word/character count is required, perform a preliminary count here.\n"
            "2. <FINAL_RESPONSE>: Provide the output that satisfies every constraint. Do not include any meta-talk or explanations.\n\n"
            "EXAMPLE:\n"
            "USER: Write a 4-word sentence about a cat without using the word 'meow'.\n"
            "<PLANNING_SCRATCHPAD>:\n"
            "- Topic: Cat\n"
```

```
            "- Length: Exactly 4 words\n"
            "- Forbidden: [F1] (the 'm' word)\n"
            "- Plan: Sentence 1 starts with 'The', ends with 'sleeps'.\n"
            "<FINAL_RESPONSE>:\n"
            "The fluffy cat sleeps.\n"
            "END_EXAMPLE"
        )

        user_input = (
            f"REQUEST:\n{prompt}\n\n"
            "Strictly follow the <PLANNING_SCRATCHPAD> and <FINAL_RESPONSE> protocol.
                "
        )

        response, cost = self.query_llm(
            prompt=user_input,
            system=system_prompt,
            temperature=self.temperature
        )

        # Robust extraction focusing on the <FINAL_RESPONSE> tag
        final_output = ""
        if "<FINAL_RESPONSE>" in response:
            # Take the content after the last occurrence of the tag to avoid issues
                if the model repeats it
            parts = response.split("<FINAL_RESPONSE>")
            final_output = parts[-1].strip()
            # Remove closing tag if present
            if "</FINAL_RESPONSE>" in final_output:
                final_output = final_output.split("</FINAL_RESPONSE>")[0].strip()
        else:
            # Fallback for models that might use lowercase or skip tags
            if "<FINAL_RESPONSE" in response.upper():
                tag_start = response.upper().find("<FINAL_RESPONSE")
                content_start = response.find(">", tag_start) + 1
                final_output = response[content_start:].strip()
            else:
                # If tags are totally missing, take the last block of text
                parts = response.split('\n\n')
                final_output = parts[-1].strip()

        # Final cleanup of common LLM artifacts
        # Remove markdown code blocks
        final_output = re.sub(r'^```\w*\n', '', final_output)
        final_output = re.sub(r'```$', '', final_output).strip()

        # Remove any leading colons or labels the model might have prepended
        final_output = re.sub(r'^(Result|Output|Answer|Response):\s*', '',
             final_output, flags=re.IGNORECASE).strip()

        # Remove any trailing "END_EXAMPLE" or similar hallucinated markers
        final_output = re.sub(r'\n?END_EXAMPLE$', '', final_output).strip()

        return final_output, cost
# EVOLVE-BLOCK-END
```

Figure 15: Best program discovered by SEAEVO$_{\text{SHINKA}}$ with Gemini-3-Flash in IFBench.