HER: HIERARCHICAL EVOLUTIONARY REASONING FOR LLM-DRIVEN ALGORITHMIC OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-driven evolutionary agents hold strong promise for algorithmic optimization, yet most existing paradigms still rely on flat population structures and uniform compute allocation. This fundamentally limits their ability to coordinate broad exploration with sustained refinement. To address this, we present HER (**H**ierarchical **E**volutionary **R**easoning). By organizing candidate programs into interacting layers, the framework couples cross-layer population management with bellwether-guided transitions and layer-wise prompt-context selection. Under limited test-time budgets, such a structure enables adaptive compute allocation and more structured search dynamics, allowing promising candidates to guide higher-level refinement while preserving diversity in lower layers. Extensive evaluation across diverse benchmarks, spanning both mathematical and system tasks, shows that HER consistently outperforms state-of-the-art baselines in solution quality, achieving an average relative improvement of 6.67% on mathematical benchmarks and 12.72% on system benchmarks. These results suggest that hierarchical organization is not merely a structural modification, but a promising strategy for improving LLM-driven algorithmic optimization.

1 INTRODUCTION

Solving complex problems often requires an iterative process of trial and reflection, mirroring the essence of evolution. Recent remarkable advancements in the reasoning capabilities of large language models (LLMs) enable them to integrate this evolutionary mechanism, driving a paradigm shift in the landscape of algorithmic optimization (Romera-Paredes et al., 2024; Cheng et al., 2025; Georgiev et al., 2025). To facilitate the evolutionary procedure, an *evolutionary strategy* identifies the most promising parents, and an LLM *generator model* proposes insightful modifications to yield breakthrough offspring. However, as optimization tasks grow in complexity, it is crucial to design advanced evolutionary strategies to fully harness the model's ability to navigate vast search spaces.

A pioneering LLM-driven evolutionary framework is AlphaEvolve (Novikov et al., 2025) (whose open-source version is OpenEvolve (Sharma, 2025)), which, by integrating the MAP-Elites algorithm (Mouret & Clune, 2015) with an island-based population model (Tanese, 1989), either matches or improves upon best-known bounds across diverse mathematical problems (Georgiev et al., 2025). Subsequently, ShinkaEvolve (Lange et al., 2025) addresses sample inefficiency via a combination of improved parent sampling, code novelty rejection-sampling, and bandit-based LLM ensembles. In addition, concurrent studies have sought to optimize evolutionary strategies from the perspectives of context utilization (Jiang et al., 2026; Su et al., 2026) and meta-guidance adaptive optimization (Cemri et al., 2026; Liu et al., 2026), respectively.

Despite their success, these methods still rely on flat population structures and uniform resource allocation to drive the evolutionary process. Such flat structures limit the flexibility to dynamically balance exploration (i.e., discovering new trajectories with different parental contexts) and exploitation (i.e., performing local refinement within a fixed lineage), causing the process to get stuck in a certain state for prolonged periods. As empirically observed in Figure 1 for OpenEvolve (Sharma, 2025), this flatness disrupts the exploration-exploitation trade-off: the lineage dynamics of the best individual become erratic, either switching parental contexts too frequently or getting trapped within a fixed lineage for too long, ultimately leading to suboptimal results.

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.

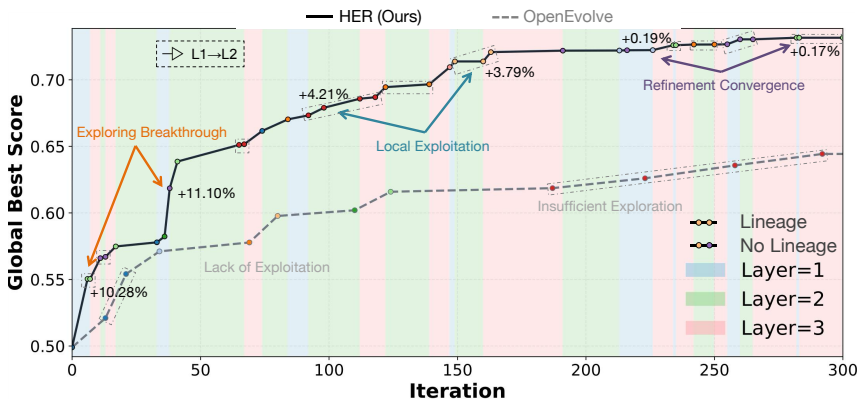


Figure 1: Comparative analysis of evolutionary behaviors under different evolutionary strategies using gpt-oss-120b on the signal processing task. The solid black line represents the best-so-far fitness curve of HER (Ours, a hierarchical database), while the dashed grey line denotes the OpenEvolve baseline (a flat database) (Sharma, 2025). Markers on each curve are colored by their lineage to track shifts in evolutionary branches. The background shading (Layer 1∼3) indicates the current layer where HER's global best is located, where a "layer" represents a hierarchical search space, with lower layers emphasizing broader exploration and higher layers emphasizing more focused refinement.

Unlike the flat structures discussed above, human organizations are typically hierarchical, enabling adaptive resource allocation to maximize collective efficiency (Zafeiris & Vicsek, 2013; Offord et al., 2019). Similarly, hierarchical search strategies, which are typically structured as interacting sub-populations at varying granularities, have proven effective in traditional numerical optimization (Wang et al., 2014; Schwanke et al., 2026). In this context, multi-level population organization can improve search efficiency and solution quality by coordinating broad exploration with focused refinement (Tian et al., 2024). Drawing inspiration from these insights, we hypothesize that embedding a hierarchical paradigm into LLM-driven evolutionary coding agents for algorithmic optimization can better reconcile the exploration-exploitation dilemma. This motivates our research question: *can we design an LLM-driven hierarchical evolutionary framework that dynamically allocates computational resources and coordinates the population, thereby yielding superior optimization performance?*

To this end, we propose an innovative approach, HER (**H**ierarchical **E**volutionary **R**easoning). To our knowledge, this is among the first hierarchical evolutionary coding frameworks that adapt multi-layer population management to LLM-driven algorithmic optimization. As visualized in Figure 2, HER structures candidate programs into multi-level populations featuring explicit upward and downward information flows. By doing so, high-potential individuals can guide the global search without sacrificing diverse exploratory branches. To implement this, we introduce a layer-aware optimization procedure that integrates bellwether-guided transition control with PSO-inspired reference-role scheduling in the velocity prompt. This allows for the dynamic allocation of computational budgets based on each layer's evolutionary state. Consequently, by channeling evolutionary information across layers, the dynamically coupled topology implements interactive cross-layer reasoning rather than classical continuous swarm arithmetic. As illustrated in Figure 1, our proposed HER achieves a better dynamic balance between broad exploration and local refinement, efficiently improving the upper bound of the global optimum under limited test-time budgets. **Overall, the contributions of this paper are summarized as follows.**

- We present HER, a hierarchical evolutionary framework that moves beyond flat population management by introducing a multi-layer structure with explicit cross-layer migration for LLM-driven algorithmic optimization.
- We propose a bellwether-guided evolutionary strategy that integrates layer transition with PSO-inspired, layer-wise inclusion of textual references in the improvement prompt, enabling adaptive compute allocation and a finer balance between exploration and exploitation during test-time search.

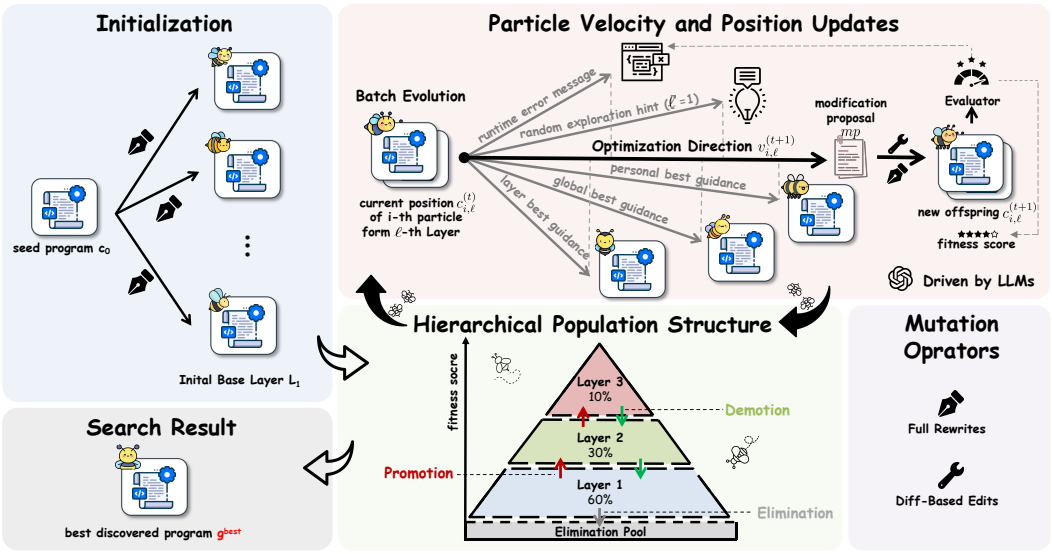


Figure 2: Overview of the HER framework. Starting from an initial pool of candidate programs, HER organizes the population into multiple hierarchical layers and iteratively performs parent selection, LLM-based evolution, and evaluation. Different layers are assigned different search roles and mutation operators, so that lower layers encourage broader exploration while upper layers focus on refinement of promising solutions. During evolution, a batch of candidates is updated via bellwether-guided cross-layer transitions and PSO-inspired prompt-context selection. Under a limited test-time compute budget, this mechanism enables high-quality individuals and improvement directions to propagate across the hierarchy. The updated candidates are then re-evaluated and written back to the hierarchical database, forming a closed-loop optimization process until the final solution is obtained.

- Extensive experiments across mathematical and system benchmarks demonstrate that HER consistently outperforms state-of-the-art baselines on the gpt-oss-120b backbone, achieving average relative improvements of 6.67% and 12.72%, respectively.

Due to space limitations, we defer detailed discussion of related work to Appendix A, which reviews LLM-based evolutionary computation and evolutionary coding agents for algorithmic optimization.

2 PRELIMINARY

Problem Definition Following AlphaEvolve (Novikov et al., 2025), we focus on evolving scientific problems where the solutions can be expressed as executable programs. To drive the evolutionary process, at each iteration t, we employ an LLM, $\mathcal{M}_\theta$, to generate either a feasible offspring program c_t (i.e., the complete code) or a sequence of diff blocks (`SEARCH/REPLACE` format, used for targeted updates of specific parts of the parent code, with an illustrative example provided in Appendix E), based on the meta-prompt and the parent program information. Subsequently, a task-specific evaluator $\mathcal{E}$ is executed to assign a scalar score $s_t = \mathcal{E}(c_t)$, serving as feedback on the fitness of the new solution. Throughout this process, a program database $\mathcal{D}_t$ stores the retained candidate programs together with their fitness scores and provides the parent pool for subsequent iterations. An evolutionary framework is then characterized by its selection mechanism, which samples parents from $\mathcal{D}_{t-1}$, and its survival mechanism, which decides whether an evaluated offspring c_t is added back to $\mathcal{D}_t$. Accordingly, the goal of evolution is to find the program with the highest score in $\mathcal{D}_T$ after T candidate-update steps:

$$c^* = \arg\max_{c \in \mathcal{D}_T} \mathcal{E}(c). \tag{1}$$

where $\mathcal{D}_T$ represents the database state after these T steps. In our setting, the parameters θ of the LLM $\mathcal{M}_\theta$ remain frozen throughout the search, so improvements are obtained entirely by evolving candidate programs rather than updating the model itself.

Particle Swarm Optimization Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995) is a bio-inspired metaheuristic algorithm that mimics the social foraging behavior of bird flocks to efficiently navigate search spaces. The algorithm maintains a population of particles, where each particle iteratively refines its position based on its own experience and the shared knowledge from the swarm, balancing exploration and exploitation. This decentralized and self-organizing mechanism makes PSO particularly suitable for optimization tasks in complex search spaces (Zhang et al., 2025b).

Mathematically, at iteration t, the position $\mathbf{x}_i(t)$ and velocity $\mathbf{v}_i(t)$ of the i-th particle in a d-dimensional space are updated according to the following equations:

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_1 \cdot r_1 \cdot (\mathbf{p}_i - \mathbf{x}_i(t)) + c_2 \cdot r_2 \cdot (\mathbf{g} - \mathbf{x}_i(t)), \tag{2}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1). \tag{3}$$

In this formulation, $\mathbf{p}_i$ denotes the particle's personal best and $\mathbf{g}$ denotes the swarm's global best. The inertia term w preserves search momentum, c_1 and c_2 control the strengths of personal and social attraction, and r_1, r_2 inject randomness for exploration. Together, these terms encourage particles to refine promising directions while still benefiting from shared discoveries.

Our method retains three principles from PSO: maintaining search momentum, leveraging personal experience, and using population guidance. It realizes these principles as semantic prompt context in a discrete program space. The correspondence is conceptual rather than algebraic: HER does not perform numerical vector arithmetic. Table 1 makes this mapping explicit.

Table 1: PSO concepts and their semantic realization in HER.

Concept	Standard PSO	HER realization
Position	Vector $\mathbf{x}_i^{(t)}$	Executable program $c_{i,\ell}^{(t)}$
Velocity	Direction $\mathbf{v}_i^{(t)}$	Textual improvement direction generated by $\mathcal{M}_{\text{vel}}$
Inertia	Weighted history $w\mathbf{v}_i^{(t)}$	Probabilistic reuse of the previous improvement direction
Best guidance	Personal and global best	Adaptive personal-, layer-, and global-best context
Position update	Vector addition $\mathbf{x} + \mathbf{v}$	LLM-generated rewrite or edit (no vector arithmetic)
Topology	Population-wide global guidance	Layered population with controlled migration

3 METHODOLOGY

In this section, we present HER, a **H**ierarchical **E**volutionary **R**easoning framework for algorithmic optimization. An overview of the full framework is depicted in Figure 2. We begin by detailing the LLM-driven mutation operators that serve as the foundational evolution primitives (§3.1). We then describe the hierarchical population design and layer-wise control coefficients for balancing exploration and exploitation (§3.2). Next, we formalize the semantics-aware velocity update mechanism in language space (§3.3) and the code-level position update procedure for generating validated offspring (§3.4). Finally, we detail particle lifecycle operations, including initialization and layer transitions, via population management rules (§3.5).

3.1 LLM-DRIVEN MUTATION OPERATORS

To advance the evolutionary process, we adopt a hybrid strategy comprising two distinct mutation operators to generate new solutions, aiming to balance exploration and exploitation:

- **Full Rewrites.** To reduce the risk of premature convergence to local optima, we employ full program rewrites to maximize offspring diversity and promote broader exploration of the search space (see Figure 4(b)).
- **Diff-Based Edits.** Following Novikov et al. (2025), we implement diff-based edits with `SEARCH/REPLACE` blocks, enabling targeted code revisions that support fine-grained exploitation of promising solutions (see Figure 4(c)).

During the generation of a mutation proposal using the aforementioned operators, we utilize special text markers (# `EVOLVE-BLOCK-START` and # `EVOLVE-BLOCK-END`) to ensure that immutable

code regions remain fixed. In addition, if a candidate program derived from the proposed modifications fails evaluation, we allow for limited error correction based on the execution feedback, thereby minimizing the risk of discarding potentially viable solutions.

3.2 Hierarchical Population Structure

Beyond balancing exploration and exploitation from the perspective of mutation operators, HER incorporates a hierarchical population structure to facilitate convergence while preserving population diversity. In our implementation, each candidate program is treated as a particle within a single population partitioned into L layers (e.g., $L = 3$). Each layer is assigned a distinct capacity and search role.

As illustrated in the lower-center panel of Figure 2, this hierarchical population structure resembles a pyramidal architecture. The base layer (L_1), which constitutes approximately 60% of the population, prioritizes exploration. Particles in this layer are characterized by lower fitness scores and receive random exploration prompts, favoring the *Full Rewrites* operator over *Diff-Based Edits* to encourage global search. Conversely, the top layer (L_3) is dedicated to exploitation. With the smallest capacity ($\sim$10%), under the default population of 10 this yields roughly one particle: a small top refinement niche rather than a multi-particle classical swarm. It comprises high-fitness particles that rely heavily on individual inertia, personal best, and layer-/global-best references, primarily employing the fine-grained *Diff-Based Edits* operator. The intermediate layer (L_2), comprising about 30% of the population with moderate fitness, bridges the gap between exploration and exploitation, with particles favoring *Diff-Based Edits* over *Full Rewrites*. Throughout the evolutionary process, particles dynamically transition between these layers according to promotion and demotion rules.

3.3 Semantic-aware Velocity Update

To provide an improvement direction for the subsequent evolution, HER constructs a velocity update prompt based on the current program and reference solutions derived from various top-performing particles (e.g., pbest, lbest, and gbest) that incorporates multiple background elements (see the upper-center panel of Figure 2). Then, HER utilizes an LLM for semantic-aware refinements. This process involves analyzing the reference solutions and the current program to distill a concise improvement direction, thereby guiding the current program towards the reference solutions.

At layer $\ell \in \{1, 2, 3\}$, the velocity $v_{i,\ell}^{(t+1)}$ of particle $p_{i,\ell}$ at iteration $t+1$ is represented as a natural-language improvement direction generated by the LLM $\mathcal{M}_{\mathrm{vel}}$ from a structured velocity prompt. We keep the formulation in equation form because the update is implemented by concatenating several optional textual fields before passing them to the model:

$$v_{i,\ell}^{(t+1)} = \mathcal{M}_{\mathrm{vel}}\Big(\texttt{Template}_{\mathrm{vel}}(c_{i,\ell}^{(t)} \oplus z_{\mathrm{inertia}}^{(t)} \oplus z_{\mathrm{cog}}^{(t)} \oplus z_{\mathrm{soc}}^{(t)} \oplus z_{\mathrm{explore}}^{(t)} \oplus z_{\mathrm{err}}^{(t)})\Big). \tag{4}$$

Here, each $z_{\star}^{(t)}$ denotes an optional textual component in the prompt: the previous velocity, the personal-best signal, the layer/global-best social signal, an exploration hint, or an error message. Intuitively, HER forms the velocity prompt by combining the current program with a selectively sampled set of historical and cross-layer references, and then asks the LLM to summarize these signals into a concise improvement direction. The detailed sampling rules for these components, together with the probability mapping functions $\phi(\cdot)$, $\psi_\ell(\cdot)$, and $\psi_{\mathrm{glob}}(\cdot)$, are deferred to Appendix D. The resulting output $v_{i,\ell}^{(t+1)}$ is a compact textual suggestion (e.g., a code difference summary or optimization direction) that serves as input for the subsequent position update step.

3.4 Code-level Position Update

Upon obtaining the suggested improvement direction, HER constructs a structured code evolution prompt for the current program $c_{i,\ell}^{(t)}$ (see the upper-center panel of Figure 2). This prompt, which integrates the updated velocity $v_{i,\ell}^{(t+1)}$, the reference particles Ω_{ref} (e.g., pbest, lbest, and gbest), and different mutation operator templates (e.g., Full Rewrites or Diff-Based Edits), is subsequently passed to an LLM $\mathcal{M}_{\mathrm{pos}}$ to generate a modification proposal:

$$mp = \mathcal{M}_{\mathrm{pos}}(\texttt{Template}_{\mathrm{pos}}(c_{i,\ell}^{(t)} \oplus v_{i,\ell}^{(t+1)} \oplus \Omega_{\mathrm{ref}})). \tag{5}$$

Then, a code parsing function $\mathrm{Parser}(\cdot,\cdot)$ is used to extract the new code:

$$c_{i,\ell}^{(t+1)} = \mathrm{Parser}(c_{i,\ell}^{(t)}, mp). \tag{6}$$

During evaluation, if the evaluator $\mathcal{E}(c_{i,\ell}^{(t+1)})$ reports an error, HER performs a fixed number of retries with an appended error-fixing prompt, which summarizes the error message and highlights the faulty code. Finally, the newly evolved offspring program is added to the database $\mathcal{D}$ to facilitate subsequent best particle updates, layer transitions, and potential parent program replacement.

3.5 Population Management

Particle Initialization. In contrast to recent works (Sharma, 2025; Lange et al., 2025; Su et al., 2026) that adhere to the Alpha-style (Novikov et al., 2025) island-based evolutionary framework, we propose a multi-layer hierarchical evolutionary paradigm. As shown in the left-side panel of Figure 2, for a given problem, we first evaluate the seed program c_0 and add it as the first particle to base layer L_1. Subsequently, we invoke an LLM to generate a sequence of diverse heuristic insights based on the problem description, with the aim of broadening the initial search directions. To further initialize a diverse population, we employ the *Full Rewrites* operator to generate variants of c_0, guided by insights sampled from the constructed insight pool. Note that, to improve efficiency, we generate candidate particles in parallel to populate the layer L_1 population until its capacity is met.

Particle Layer Transitions. When a newly evolved particle is validated, it is assigned to a layer if its fitness score exceeds that of its parent, and it is otherwise discarded. To allocate computational budgets efficiently to balance exploration and exploitation, we design the following layer transition rules based on the principle of survival of the fittest to control the population distribution:

- **Promotion:** For each new particle added to layer L_ℓ, we determine whether it should transition to a higher layer $\ell' > \ell$. As depicted in the lower-center panel of Figure 2, if layer $L_{\ell'}$ has residual capacity and the particle's score exceeds the promotion threshold (e.g., a specified proportion of highest score in layer $L_{\ell'}$), the transition is permitted. Alternatively, if layer $L_{\ell'}$ is at capacity but the particle outperforms the lowest-scoring particle within that layer, the new particle swaps with the latter. The process can be applied repeatedly so that a strong particle from layer L_1 may jump directly to a higher layer.
- **Demotion:** If a particle in layer $\ell > 1$ has a stagnation counter that exceeds a layer-specific threshold, it is demoted to layer $L_{\ell-1}$. If the lower layer is full, it swaps with the highest-scoring particle in that layer. Upon demotion, the particle's stagnation counter is reset and its velocity is cleared.
- **Elimination** (L_1 only): Particles in layer L_1 that remain stagnant for an extended period are removed from the active population and placed into an eliminated particle pool, from which the best-performing candidate may be selectively reinserted into layer L_1. Optionally, layer L_1 may undergo periodic *resetting*: the layer is cleared and reinitialized to its startup state to initiate a new round of exploration.

Batch Evolution. At each batch iteration, HER first guarantees that at least one particle is sampled from every layer to populate the sampling pool. If the pool is not yet filled, *layer-weighted sampling* continues selecting particles. During this process, each particle is assigned a sampling weight of $1/(1 + \texttt{selection_count})$, where `selection_count` tracks how many times the particle has been selected in previous rounds of the current run. This inversely proportional weighting prioritizes less frequently selected particles, thereby ensuring computational load balancing. Subsequently, these selected particles are evolved in parallel via an LLM-driven process.

This batch evolution procedure underpins the multi-layer evolutionary dynamics illustrated in Figure 2. The higher layers, characterized by low-density yet high-quality sub-populations, focus on the continuous evolution of high-fitness individuals. Meanwhile, the lower sub-populations leverage their diversity to constantly explore the boundaries of the solution space. By combining bellwether-guided transition control with PSO-inspired prompt assembly, this mechanism effectively mitigates the inefficient sampling problem caused by uniform computational resource allocation under a flattened population structure, ultimately achieving superior performance and excellent convergence. The complete procedure of HER is detailed in Algorithm 1 of the Appendix B.

4 EXPERIMENTS

In this section, we evaluate the performance of HER on a suite of mathematical and system-level algorithmic optimization benchmarks. We focus on the main empirical comparisons, ablation studies, and layer-wise evolutionary dynamics analysis here, while deferring more detailed analyses of convergence behavior, case studies, and additional backbone settings to the Appendix.

4.1 EXPERIMENTAL SETUP

Benchmarks. We consider two benchmark suites. The first suite contains six mathematical optimization tasks from the open-source OpenEvolve repository (Sharma, 2025), including `Circle Packing` (square and rectangle), `Heilbronn` (triangle and convex), `MinMax Distance` (d=3), and `Signal Processing`, covering geometric packing, extremal point placement, distance minimization, and program-based signal design. The second suite contains five system benchmarks from ADRS (Cheng et al., 2025), including `EPLB`, `LLM-SQL`, `Prism`, `TXN`, and `Cloudcast`, covering balancing, retrieval/planning, scheduling, transaction optimization, and resource-allocation scenarios. Detailed benchmark descriptions are in Appendix C.

Baselines. Our primary comparisons under gpt-oss-120b (Agarwal et al., 2025) include strong LLM-based evolutionary frameworks with open-source implementations: OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2025), ContextEvolve (Su et al., 2026), and AdaEvolve (Cemri et al., 2026). In the complementary Gemini-3-Pro (Team, 2025) evaluation, we additionally include EvoX (Liu et al., 2026), for which we report the previously published best scores where applicable. When available, we also list Human/SOTA scores and AlphaEvolve (Novikov et al., 2025) as external reference points.

Implementations. Unless otherwise specified, experiments using gpt-oss-120b run for 300 iterations. The `LLM-SQL`, `Prism`, and `TXN` experiments use 100 iterations. Each method is independently evaluated three times, with mean$\pm$std and the best value reported as the primary statistics. The Gemini-3-Pro experiments evaluate each mathematical task for 100 iterations and report the best result achieved. Depending on the task, better performance is reflected by either a higher ($\uparrow$) or lower ($\downarrow$) score. See Appendix N for further details.

4.2 MAIN RESULTS

Mathematical Benchmarks. Results on the six mathematical tasks for gpt-oss-120b are presented in Table 2. HER achieves the strongest overall performance, demonstrating the best combined competitiveness by ranking first on five of the six metrics and remaining near-optimal on the remaining one. This indicates that hierarchical evolution search improves both exploration quality and exploitation efficiency: the hierarchical decomposition widens the search over diverse program structures while the evolution-level selection sharpens exploitation around the most promising regions. Furthermore, the mean$\pm$std results show that these gains remain consistent across independent runs, confirming that the improvement is not driven by run-level variance. We further conduct performance and convergence analysis on both a frontier large-scale backbone (e.g., Gemini-3-Pro (Team, 2025)) and smaller-scale backbones (e.g., Qwen3-4B (Yang et al., 2025) and Qwen3-8B (Guo et al., 2025)), as detailed in Appendices F.1 and I.

System Benchmarks. To verify the generalization capability of hierarchical evolutionary search, we extend our evaluation to realistic system-level tasks. The gpt-oss-120b results are reported in Table 3. HER consistently outperforms prior LLM-driven evolutionary baselines across all evaluation metrics, and simultaneously achieves the best values on core task metrics (e.g., higher Balance/Hit/Pressure and lower Makespan/Cost where appropriate). These gains indicate that hierarchical migration not only improves final solution quality, but also generalizes effectively from mathematical optimization to realistic system-level decision tasks. This suggests that the hierarchical organization helps allocate search effort more effectively under complex system constraints. It also indicates that the benefits of HER extend beyond synthetic mathematical settings to more realistic optimization scenarios. The results further confirm that HER delivers stronger stability, with superior central tendency and competitive variance over three independent runs.

Table 2: Comparison on six mathematical benchmarks with gpt-oss-120b as the backbone model. We report mean$_{\pm \text{std}}$ over three runs and the best-of-three score. The top score in each column is in **bold**, and the second-highest is underlined.

Method	Circle P. (Sq.)↑		Circle P. (Rect.)↑		Heilbronn (Tri.)↑		Heilbronn (Conv.)↑		MinMax Dist.↓		Signal Proc.↑	
	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best
Human/SOTA	–	2.6340	–	2.3640	–	0.0360	–	0.0306	–	4.1680	–	N/A
AlphaEvolve	–	2.6350	–	2.3658	–	0.0365	–	0.0309	–	4.1658	–	N/A
OpenEvolve	2.6302$_{\pm 0.0057}$	2.6353	2.3583$_{\pm 0.0022}$	2.3608	0.0271$_{\pm 0.0045}$	0.0314	0.0176$_{\pm 0.0000}$	0.0176	4.7006$_{\pm 0.0651}$	4.6257	0.6262$_{\pm 0.0327}$	0.6458
ShinkaEvolve	2.5842$_{\pm 0.0504}$	2.6285	2.3399$_{\pm 0.0171}$	2.3551	0.0308$_{\pm 0.0035}$	0.0332	0.0267$_{\pm 0.0029}$	0.0289	4.6675$_{\pm 0.1118}$	4.5384	0.6864$_{\pm 0.0300}$	0.7093
ContextEvolve	2.5605$_{\pm 0.0786}$	2.6297	2.3586$_{\pm 0.0030}$	2.3621	0.0274$_{\pm 0.0059}$	0.0342	0.0184$_{\pm 0.0013}$	0.0199	4.6742$_{\pm 0.1003}$	4.5584	0.6040$_{\pm 0.1040}$	0.7069
AdaEvolve	2.6300$_{\pm 0.0046}$	2.6343	2.3047$_{\pm 0.0480}$	2.3427	0.0301$_{\pm 0.0005}$	0.0306	0.0230$_{\pm 0.0008}$	0.0239	4.2802$_{\pm 0.1609}$	**4.1658**	0.7221$_{\pm 0.0030}$	0.7254
HER (Ours)	**2.6352**$_{\pm 0.0009}$	**2.6360**	**2.3619**$_{\pm 0.0020}$	**2.3642**	**0.0353**$_{\pm 0.0009}$	**0.0362**	**0.0287**$_{\pm 0.0006}$	**0.0293**	**4.2749**$_{\pm 0.1657}$	4.1747	**0.7232**$_{\pm 0.0086}$	**0.7316**

Table 3: Comparison on five system benchmarks with gpt-oss-120b as the backbone model. We report mean$_{\pm \text{std}}$ over three runs and the best-of-three score. Note that large mean–best gaps on CloudCast (e.g., OpenEvolve) reflect cross-seed variance rather than a direction error. The top score in each column is in **bold**, and the second-highest is underlined.

Method	EPLB↑		LLM-SQL↑		Prism↑		TXN↑		Cloudcast↑	
	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best
Human/SOTA	–	0.1400	–	0.6600	–	22.3400	–	3080.0	–	0.0016
OpenEvolve	0.1450$_{\pm 0.0057}$	0.1516	0.7330$_{\pm 0.0672}$	0.8098	25.2972$_{\pm 0.15}$	25.4687	3817.8$_{\pm 74.9}$	3861.0	0.0048$_{\pm 0.0067}$	0.0126
ShinkaEvolve	0.1769$_{\pm 0.0420}$	0.2238	0.7364$_{\pm 0.0061}$	0.7434	26.2133$_{\pm 0.03}$	26.2332	4040.9$_{\pm 139.1}$	4184.1	0.0013$_{\pm 0.0003}$	0.0017
ContextEvolve	0.1461$_{\pm 0.0024}$	0.1488	0.7140$_{\pm 0.0113}$	0.7244	25.7013$_{\pm 0.10}$	25.8040	3953.3$_{\pm 67.5}$	4000.0	0.0085$_{\pm 0.0002}$	0.0087
AdaEvolve	0.1274$_{\pm 0.0011}$	0.1286	0.7197$_{\pm 0.0159}$	0.7380	26.2560$_{\pm 0.00}$	26.2560	4106.8$_{\pm 132.7}$	4255.3	0.0160$_{\pm 0.0014}$	0.0176
HER (Ours)	**0.2051**$_{\pm 0.0394}$	**0.2359**	**0.8996**$_{\pm 0.0821}$	**0.9780**	**26.5030**$_{\pm 0.34}$	**26.8957**	**4256.7**$_{\pm 349.4}$	**4587.2**	**0.0190**$_{\pm 0.0067}$	**0.0262**

Statistical Significance. We compare HER with each evolutionary baseline using a paired Wilcoxon signed-rank test on the eleven gpt-oss-120b task-level means from Tables 2 and 3. At α=0.05, HER significantly outperforms every baseline (exact one-sided p=4.88×10^{-4} for all comparisons). Details are provided in Appendix G.

4.3 Ablation Study

Impact of Hierarchical Design, Rewrite, and Initialization. Table 4 isolates the contributions of HER's main structural and exploratory components. The complete method attains the best mean score on all three tasks. Removing full-program rewriting causes the largest degradation on Prism and TXN Scheduling, while removing diverse initialization preserves the Circle Packing score but lowers performance on Prism and TXN Scheduling. Both flattening the population (*w/o hierarchy*) and removing the top layer (*w/o* L_3) also consistently underperform the complete method. Moreover, jointly removing the hierarchy and diverse initialization does not recover the full model's performance. These controlled comparisons show that the hierarchy is not interchangeable with either initialization diversity or rewrite-based exploration, and that the three components instead make complementary, task-dependent contributions.

Table 4: Within-HER controlled ablation. Rewrite and diverse init are fixed except where ablated; *w/o hierarchy* uses a flat population. Best and second-best scores are **bolded** and underlined.

Method	Circle P. (Sq.) ↑	Prism ↑	TXN ↑
HER (Ours)	**2.6354**$\pm$0.0010	**26.50**$\pm$0.34	**4256.7**$\pm$349.4
↳ w/o rewrite	2.6300$\pm$0.0103	26.11$\pm$0.26	3707.2$\pm$140.9
↳ w/o diverse init	**2.6354**$\pm$0.0010	26.33$\pm$0.14	3793.0$\pm$41.3
↳ w/o hierarchy	2.6270$\pm$0.0044	26.38$\pm$0.23	3817.0$\pm$172.9
↳ w/o hierarchy, w/o diverse init	2.6324$\pm$0.0032	26.26$\pm$0.00	3957.2$\pm$238.7
↳ w/o L_3	2.6331$\pm$0.0045	26.25$\pm$0.01	3929.2$\pm$121.4

Table 5 evaluates whether transferring HER's diverse initial population to OpenEvolve (OE) improves performance under the same update budget. This transfer improves OE's mean score on Prism from 25.2972 to 26.1628, but yields lower mean scores on Circle Packing (Square) and TXN Scheduling than OE's default initialization. The complete HER framework achieves the highest mean score on all three tasks, suggesting that diverse initialization can help in a task-dependent manner but does not by itself account for HER's overall advantage.

Table 5: Transfer of HER's diverse initialization to OpenEvolve under the same update budget.

Method	Circle P. (Sq.) ↑	Prism ↑	TXN ↑
HER (Ours)	**2.6354**±0.0008	**26.5030**±0.34	**4256.7**±285.3
OE + default init	2.6302±0.0057	25.2972±0.15	3817.8±74.9
OE + HER init	2.6286±0.0013	26.1628±0.05	3765.8±284.7

Table 6: Ablation of search momentum and reference selection under the same update budget.

Mode	Circle P. (Sq.) ↑	Prism ↑	TXN ↑
HER (Ours)	**2.6354**±0.0010	**26.50**±0.34	**4256.7**±349.4
↳ No-inertia	2.6342±0.0018	26.40±0.26	4059.0±172.8
↳ Always best-k	2.6350±0.0017	26.47±0.42	3883.9±128.8

Impact of Search Momentum and Reference Selection. Table 6 isolates search momentum and adaptive reference selection with all other components fixed. The complete strategy leads on all three tasks. Removing inertia lowers every mean, especially on TXN Scheduling, confirming the value of prior improvement directions. *Always best-k* remains competitive on Circle Packing and Prism but drops markedly on TXN, suggesting that fixed elite conditioning can over-concentrate complex searches. Thus, search momentum and adaptive reference guidance are complementary.

4.4 Layer-wise Evolutionary Dynamics Analysis

To illustrate this hierarchical mechanism, Figure 3 depicts the layer-wise evolutionary dynamics for the Signal Processing task. The **left** subplot visualizes the distribution of performance changes (Δ = child − parent) across different layers, revealing a distinct division of labor. Specifically, L_1 exhibits the highest proportion of positive updates ($\Delta > 0$), indicating that it serves as the primary driver for productive exploration. Conversely, L_3 is dominated by zero updates ($\Delta = 0$), suggesting a conservative refinement role that preserves strong candidates with minimal perturbation. The **right** subplot depicts the inter-layer transition matrix, which quantifies the strength of candidate flow between layers. This matrix reveals that information flow is strongest between adjacent layers, especially between L_1 and L_2, while direct transfer between L_1 and L_3 is much weaker. This suggests that HER does not jump abruptly from broad exploration to top-layer exploitation. Instead, candidate programs are progressively filtered, promoted, and stabilized through intermediate layers.

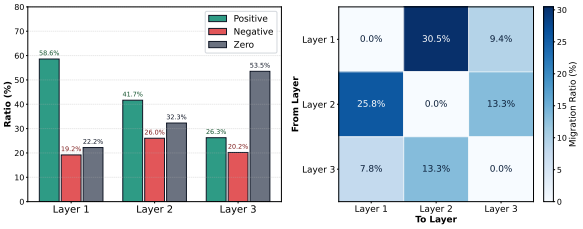


Figure 3: Layer-wise evolutionary dynamics analysis on the Signal Processing task using gpt-oss-120b. **Left**: Layer-wise update sign ratios. **Right**: Inter-layer transition matrix.

In summary, these results demonstrate that the hierarchy is not merely a static structural prior, but an active coordination device that separates exploratory discovery from cautious refinement while maintaining controlled cross-layer communication. For further details on the evolutionary lineage trajectory analysis, see Appendix L.

5 Conclusion

We introduce HER, a hierarchical evolutionary reasoning framework for LLM-driven algorithmic optimization. We observe that flat populations with uniform update rules struggle to balance exploratory diversity and focused refinement under limited test-time compute. To address this, HER turns hierarchy from a static population partition into an active search-control mechanism, coupling bellwether-guided bidirectional migration with layer-specific mutation and prompt-context selection to recycle stalled candidates and progressively refine promising ones.

Experiments demonstrate that HER achieves the best overall performance among representative evolutionary coding baselines across diverse scientific domains, with favorable convergence behavior across backbone scales. Ablations further show that these performance gains stem from hierarchical population management rather than diverse initialization or rewrite-based mutation alone. Moreover, layer-wise update and candidate-trajectory analyses also reveal distinct exploration and refinement roles for lower and upper layers, with cross-layer transitions effectively helping stalled candidates resume improvement. These findings establish hierarchical population organization as a promising design principle for compute-bounded evolutionary optimization with LLMs.

THE USE OF LARGE LANGUAGE MODELS

In accordance with the policy of ICLR 2027, we employed large language models (LLMs) to enhance the linguistic quality and readability of this manuscript, including grammar correction and phrasing refinement.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

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.

Shuvayan Brahmachary, Subodh M Joshi, Aniruddha Panda, Kaushik Koneripalli, Arun Kumar Sagotra, Harshil Patel, Ankush Sharma, Ameya D Jagtap, and Kaushic Kalyanaraman. Large language model-based evolutionary optimizer: Reasoning with elitism. *Neurocomputing*, 622: 129272, 2025.

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

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, et al. Barbarians at the gate: How ai is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.

Shangbin Feng, Zifeng Wang, Yike Wang, Sayna Ebrahimi, Hamid Palangi, Lesly Miculicich, Achin Kulshrestha, Nathalie Rauschmayr, Yejin Choi, Yulia Tsvetkov, et al. Model swarms: Collaborative search to adapt llm experts via swarm intelligence. *arXiv preprint arXiv:2410.11163*, 2024.

David B Fogel. Introduction to evolutionary computation. *Modern heuristic optimization techniques: Theory and applications to power systems*, 1:1–23, 2007.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, pp. 1–3, 2025.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

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

William B Langdon and Riccardo Poli. *Foundations of genetic programming*, volume 90. Springer, 2002.

Robert Lange, Yingtao Tian, and Yujin Tang. Large language models as evolution strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 579–582, 2024.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Shanda Li, Tanya Marwah, Junhong Shen, Weiwei Sun, Andrej Risteski, Yiming Yang, and Ameet Talwalkar. Codepde: An inference framework for llm-driven pde solver generation. *arXiv preprint arXiv:2505.08783*, 2025.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhicheng Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, 2024a.

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

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

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *ACM Transactions on Evolutionary Learning*, 4(4):1–40, 2024.

Risto Miikkulainen and Stephanie Forrest. A biological perspective on evolutionary computation. *Nature Machine Intelligence*, 3(1):9–15, 2021.

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

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Matt Offord, Roger Gill, and Jeremy Kendal. The effects of prestige on collective performance and information flow in a strictly hierarchical institution. *Palgrave Communications*, 5(1):4, 2019.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Andrej Schwanke, Lyubomir Ivanov, David Salinas, Frank Hutter, and Arber Zela. Multi-objective hierarchical optimization with large language models. *arXiv preprint arXiv:2601.13892*, 2026.

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

Yamato Shinohara, Jinglue Xu, Tianshui Li, and Hitoshi Iba. Large language models as particle swarm optimizers. *arXiv preprint arXiv:2504.09247*, 2025.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. Llm-sr: Scientific equation discovery via programming with large language models. *arXiv preprint arXiv:2404.18400*, 2024.

Thilak Shekhar Shriyan, Janavi Srinivasan, Suhail Ahmed, Richa Sharma, and Arti Arya. Swarmprompt: Swarm intelligence-driven prompt optimization using large language models. In *ICAART (3)*, pp. 86–93, 2025.

Hongyuan Su, Yu Zheng, and Yong Li. Contextevolve: Multi-agent context compression for systems code optimization. *arXiv preprint arXiv:2602.02597*, 2026.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Reiko Tanese. *Distributed genetic algorithms for function optimization*. University of Michigan, 1989.

Gemini Team. A new era of intelligence with gemini 3, 2025.

Dongping Tian, Qiu Xu, Xinhui Yao, Guangnan Zhang, Yafeng Li, and Chenghu Xu. Diversity-guided particle swarm optimization with multi-level learning strategy. *Swarm and Evolutionary Computation*, 86:101533, 2024.

Chunhui Wan, Xunan Dai, Zhuo Wang, Minglei Li, Yanpeng Wang, Yinan Mao, Yu Lan, and Zhiwen Xiao. Loongflow: Directed evolutionary search via a cognitive plan-execute-summarize paradigm. *arXiv preprint arXiv:2512.24077*, 2025.

Lin Wang, Bo Yang, and Yuehui Chen. Improving particle swarm optimization using multi-layer searching strategy. *Information Sciences*, 274:70–94, 2014.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

Anna Zafeiris and Tamas Vicsek. Group performance is maximized by hierarchical competence distribution. *Nature communications*, 4(1):2484, 2013.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025a.

Yao Zhang, Chenyang Lin, Shijie Tang, Haokun Chen, Shijie Zhou, Yunpu Ma, and Volker Tresp. Swarmagentic: Towards fully automated agentic system generation via swarm intelligence. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1778–1818, 2025b.

A Related Work

A.1 LLMs for Evolutionary Computation

Evolutionary computation is an optimization paradigm inspired by biological evolution (Fogel, 2007; Miikkulainen & Forrest, 2021). Recent work integrates large language models (LLMs) into evolutionary computation, shifting from *syntax-level* mutation (Langdon & Poli, 2002) toward *semantic-aware* search (Liu et al., 2024b). LLMs serve as mutation and crossover operators, candidate generators, or fitness-aware editors guided by execution feedback (Lange et al., 2024; Meyerson et al., 2024). LMPSO (Shinohara et al., 2025) implements PSO updates by representing particle velocity as language to steer candidate generation. LEO (Brahmachary et al., 2025) employs LLMs as zero-shot optimizers with elitism-aware search that separates stronger from weaker candidates in numerical optimization. Compared with traditional genetic programming, LLM-based operators better preserve functional intent while making larger meaningful edits, expanding the effective search neighborhood for code-centric optimization (Shojaee et al., 2024; Hemberg et al., 2024).

A.2 Evolutionary Coding Agents for Algorithmic Optimization

Building on the above foundation, a second stream investigates autonomous coding agents for algorithmic discovery and optimization (Ouyang et al., 2025; Hubert et al., 2025; Li et al., 2025; Wan et al., 2025). The integration of LLMs and evolutionary algorithms has been shown to enable coding agents to continuously generate unprecedented solutions (Zhang et al., 2025a). A notable example is FunSearch (Romera-Paredes et al., 2024), which combined an LLM with a programmatic evaluator to search novel solutions to open mathematical problems, thereby validating the approach's feasibility for algorithmic optimization. EoH (Liu et al., 2024a) further introduced an LLM-driven evolutionary framework that evolves natural-language heuristic ideas together with their executable implementations through selection, crossover, and mutation. Subsequently, AlphaEvolve (Novikov et al., 2025) establishes a representative paradigm that couples evolutionary search with LLM-driven code revision and evaluator-based selection. Its search structure is effectively built on a flat combination of MAP-Elites and island-based populations, and OpenEvolve (Sharma, 2025) provides an open-source implementation that helped standardize this design in subsequent work.

To optimize sample efficiency, ShinkaEvolve (Lange et al., 2025) retains this overall flat-population setting while improving parent sampling, code novelty rejection-sampling, and bandit-based LLM ensembles. Conversely, CodeEvolve (Assumpção et al., 2025) diversifies search trajectories through context-aware recombination and adaptive meta-prompting (Suzgun & Kalai, 2024). To overcome context inefficiency, ContextEvolve (Su et al., 2026) mimics reinforcement learning mechanisms through multi-agent decomposition, while DeltaEvolve (Jiang et al., 2026) optimizes the evolutionary M-step by replacing full-code snapshots with structured semantic deltas. Additionally, AdaEvolve (Cemri et al., 2026) and EvoX (Liu et al., 2026) leverage meta-level interventions to reformulate LLM-driven evolution as an adaptive optimization problem. Orthogonally, specialized variants such as ThetaEvolve (Wang et al., 2025) and TTT-Discover (Yuksekgonul et al., 2026) improve evolution strategies through the lens of reinforcement learning-based test-time training. In contrast, this work focuses on test-time inference.

Despite this progress, most existing evolutionary coding agents for algorithmic optimization still rely on flat organizational backbones, even when they improve sampling, prompting, or meta-control. In particular, AlphaEvolve (Novikov et al., 2025) and ShinkaEvolve (Lange et al., 2025) serve as strong structural baselines for our setting: both are highly effective, yet neither explicitly organizes the population into interacting hierarchical layers with upward and downward transitions. Our work addresses this gap with a multi-layer hierarchical evolutionary framework that explicitly structures cross-level interaction for more stable and efficient algorithmic optimization.

A.3 Swarm-inspired Coordination in LLMs

Our work is in part inspired by particle swarm optimization (PSO) (Kennedy & Eberhart, 1995), a swarm intelligence algorithm designed for optimization. Recent studies show that swarm intelligence is becoming an important design principle for optimization tasks centered around LLMs. Specifically, SwarmPrompt (Shriyan et al., 2025) uses swarm-based operators to iteratively refine textual instructions. Model Swarms (Feng et al., 2024) applies collaborative swarm-style search to adapt collections

Algorithm 1 Hierarchical Evolutionary Reasoning (HER)

Input: Seed program c_0, evaluator $\mathcal{E}$, velocity model $\mathcal{M}_{\text{vel}}$, position model $\mathcal{M}_{\text{pos}}$, total budget B, layers $\{L_1, \ldots, L_K\}$
Output: Best discovered program g^{best}

Initialize database $\mathcal{D} \leftarrow \{(c_0, \mathcal{E}(c_0))\}$ and set $g^{\text{best}} \leftarrow c_0$
Generate heuristic insights and initialize a diverse base-layer population in L_1 via *Full Rewrites*
Set particle states, velocities, stagnation counters, and per-layer capacities
while budget consumed $< B$ **do**
  Build a batch pool by first sampling at least one particle from each non-empty layer
  Fill remaining batch slots by layer-weighted sampling
  for all selected particles $p_{i,\ell}$ in parallel **do**
    Retrieve reference solutions $p_{i,\ell}^{\text{best}}$, l_{ℓ}^{best}, and g^{best}
    Construct prompt components $z_{\text{inertia}}^{(t)}$, $z_{\text{cog}}^{(t)}$, $z_{\text{soc}}^{(t)}$, $z_{\text{explore}}^{(t)}$, and $z_{\text{err}}^{(t)}$ ▷ Eqs. (8)–(12)
    Generate semantic velocity $v_{i,\ell}^{(t+1)} \leftarrow \mathcal{M}_{\text{vel}}(\texttt{Template}_{\text{vel}}(\cdot))$ ▷ Eq. (4)
    Generate a modification proposal $mp \leftarrow \mathcal{M}_{\text{pos}}(\texttt{Template}_{\text{pos}}(\cdot))$ ▷ Eq. (5)
    Parse offspring code $c_{i,\ell}^{(t+1)} \leftarrow \text{Parser}(c_{i,\ell}^{(t)}, mp)$ ▷ Eq. (6)
    if $c_{i,\ell}^{(t+1)}$ triggers runtime or validation errors **then**
      Retry a fixed number of times with appended error-fixing feedback
    end if
    Evaluate offspring fitness $f_{i,\ell}^{(t+1)} \leftarrow \mathcal{E}(c_{i,\ell}^{(t+1)})$
    Add validated offspring and score to $\mathcal{D}$
    Update personal best, layer best, and global best if improved
    if offspring outperforms parent **then**
      Insert offspring into layer L_ℓ and apply promotion rules to higher layers
    else
      Discard offspring or keep parent unchanged
    end if
  end for
  for all layers L_ℓ **do**
    Demote stagnant particles according to layer-specific thresholds
    Eliminate persistently stagnant particles from L_1 and optionally reset L_1
    Refresh capacities, best-particle statistics, and sampling counts
  end for
end while
return g^{best}

of LLM experts, and SwarmAgentic (Zhang et al., 2025b) uses PSO-inspired population updates to automate agent-system generation. These works indicate that swarm-inspired coordination provides a promising paradigm for navigating the complex solution spaces of LLMs. As for our approach, we borrow PSO only as inspiration for naming reference roles in prompt assembly, thereby performing algorithmic optimization on executable program candidates.

B HER Algorithm Implementation Details

Algorithm 1 summarizes the end-to-end procedure of HER, including hierarchical population initialization, semantic velocity updates, code-level position updates, evaluation, and layer transition control under a fixed computational budget.

C Detailed Benchmark Descriptions

This section provides detailed descriptions of all benchmarks used in our experiments. Specifically, our experimental protocol encompasses two core challenge domains: mathematical tasks and system-level tasks.

C.1 MATHEMATICAL BENCHMARKS

We evaluate on six mathematical tasks: Circle Packing (Square, $N = 26$), Circle Packing (Rectangle, $N = 21$), Heilbronn Triangle ($T = 11$), Heilbronn Convex ($C = 13$), MinMax Distance ($D = 3$), and Signal Processing. These tasks cover geometric packing, extremal point placement, distance minimization, and program-based signal design. For each task, we follow the standard objective definitions and report the task-specific metric shown in the main text tables.

Circle Packing (Square, $N = 26$). Place $N = 26$ pairwise disjoint circles inside a unit square, where each circle must lie entirely within the boundary. The objective is to maximize the sum of radii of all circles. This is the standard "packing circles in a unit square" formulation used in AlphaEvolve-style mathematical benchmarks.

Circle Packing (Rectangle, $N = 21$). Place $N = 21$ pairwise disjoint circles inside a rectangle with fixed perimeter 4, with all circles fully contained in the rectangle. The objective is to maximize the sum of radii. Compared with the square variant, this task additionally requires optimizing under an anisotropic container geometry.

Heilbronn Triangle ($T = 11$). Place $T = 11$ points on or inside a unit-area triangle to maximize the area of the smallest triangle formed by any triplet of selected points. Equivalently, the goal is to avoid near-collinear local configurations that induce extremely small triangle areas.

Heilbronn Convex ($C = 13$). Place $C = 13$ points on or inside a unit-area convex region so that the minimum area among all triangles induced by point triplets is maximized. This extends the Heilbronn objective from a fixed triangular domain to a more general convex domain.

MinMax Distance ($D = 3$). Given n points in D-dimensional Euclidean space (here $D = 3$), optimize the configuration to minimize the ratio between the maximum and minimum pairwise distances. This objective enforces globally compact yet uniformly separated point sets and is often interpreted as a distance-uniformity optimization problem.

Signal Processing. This benchmark targets real-time adaptive signal filtering for non-stationary time series. The evolved algorithm must reduce noise while preserving genuine trend dynamics under causal and low-latency constraints. Evaluation is based on a composite objective that jointly considers reversal suppression, lag/tracking error, and robustness across diverse synthetic signal regimes, following the OpenEvolve signal-processing benchmark design.

C.2 SYSTEM BENCHMARKS (ADRS)

We evaluate on five system-level tasks: EPLB, LLM-SQL, Prism, TXN, and Cloudcast. These tasks correspond to balancing, retrieval/planning, scheduling, transaction optimization, and resource-allocation scenarios, respectively. Following prior work, we report both task-native metrics and normalized score metrics for cross-task comparison.

EPLB (Expert Placement Load Balancing). EPLB is a load-balancing task for Mixture-of-Experts inference systems, focusing on distributing computational workload proportionally across multiple GPUs. Given estimated expert-access frequencies, the algorithm decides expert replication and placement to reduce routing skew while improving rebalancing efficiency when workload patterns change. The key objective is to maximize load imbalance (*Bal.*), ensuring proportionate workload distribution across GPUs, while minimizing rebalancing speed under the same resource budget.

LLM-SQL. LLM-SQL is an SQL optimization task that rearranges query elements to maximize the Key-Value (KV) cache prefix hit rate, thereby accelerating SQL inference. The core challenge lies in balancing cache hit rate (*Hit.*) against the reordering overhead.

Prism (Global Model Scheduling). Prism is a model scheduling task that optimizes how large models are distributed across multiple GPUs to reduce contention and improve resource utilization. The optimization target is to minimize the KV pressure ratio across devices, i.e., to alleviate cache

contention during inference while maintaining effective GPU usage. In this task, better placement corresponds to higher pressure-reduction scores (*Press.*) and stronger overall scheduling quality.

TXN (Transaction Scheduling). TXN is a transaction scheduling task that optimizes the execution order of transactions in a database system to minimize total execution time, i.e., the makespan (*Make.*). The core challenge is to balance the trade-off between transaction ordering decisions and latency.

Cloudcast (Multi-Region Data Transfer). Cloudcast is a multi-region cloud data-transfer optimization task aimed at minimizing cost (*Cost*). The algorithm plans cost-efficient inter-region strategies across heterogeneous bandwidths, latencies, and egress pricing, subject to strict deadline constraints. A lower cost corresponds to a superior policy.

D Definition of Probability Mapping Functions for Velocity Update

This appendix expands the semantic-aware velocity update in Section 3.3. Recall that the velocity in HER is not a numeric displacement, but a natural-language improvement direction produced by the velocity model $\mathcal{M}_{\text{vel}}$. The prompt is assembled by concatenating the current program with several optional textual components:

Layer-wise Prompt Controls. For each layer L_ℓ, the prompt assembly uses three control parameters:

- **Inertia weight** (w_ℓ) directly specifies the probability of retaining the previous velocity $v_{i,\ell}^{(t)}$, allowing the model to continue its current improvement direction.
- **Cognitive coefficient** ($c_{1,\ell}$) parameterizes the adaptive threshold $\phi(c_{1,\ell})$ that controls whether the personal-best program $p_{i,\ell}^{\text{best}}$ is included.
- **Social coefficient** ($c_{2,\ell}$) parameterizes the adaptive thresholds $\psi_\ell(c_{2,\ell})$ and $\psi_{\text{glob}}(c_{2,\ell})$ that control access to the layer-best and, for higher layers, global-best programs.

These controls determine which textual fields enter the prompt and should not be interpreted as continuous PSO coefficients. Their effective inclusion probabilities are specified below.

$$v_{i,\ell}^{(t+1)} = \mathcal{M}_{\text{vel}}\Big(\texttt{Template}_{\text{vel}}(c_{i,\ell}^{(t)} \oplus z_{\text{inertia}}^{(t)} \oplus z_{\text{cog}}^{(t)} \oplus z_{\text{soc}}^{(t)} \oplus z_{\text{explore}}^{(t)} \oplus z_{\text{err}}^{(t)})\Big), \tag{7}$$

where $\oplus$ denotes text concatenation. The role of each optional component is defined below:

$$z_{\text{inertia}}^{(t)} = \begin{cases} v_{i,\ell}^{(t)}, & \text{with prob. } w_\ell, \\ \varnothing, & \text{otherwise,} \end{cases} \tag{8}$$

$$z_{\text{cog}}^{(t)} = \begin{cases} p_{i,\ell}^{\text{best}}, & \text{with prob. } \phi(c_{1,\ell}), \\ \varnothing, & \text{otherwise,} \end{cases} \tag{9}$$

$$z_{\text{soc}}^{(t)} = \begin{cases} l_\ell^{\text{best}}, & \text{with prob. } \psi_\ell(c_{2,\ell}), \\ \varnothing, & \text{otherwise,} \end{cases} \cup \begin{cases} g^{\text{best}}, & \text{with prob. } \psi_{\text{glob}}(c_{2,\ell}) \text{ if } \ell \geqslant 2, \\ \varnothing, & \text{otherwise,} \end{cases} \tag{10}$$

$$z_{\text{explore}}^{(t)} = \begin{cases} \text{Fixed random exploration hint}, & \text{with prob. } p_{\text{explore}} \text{ if } \ell = 1, \\ \varnothing, & \text{otherwise,} \end{cases} \tag{11}$$

$$z_{\text{err}}^{(t)} = \begin{cases} \text{Runtime/Validation error message}, & \text{if error occurs,} \\ \varnothing, & \text{otherwise,} \end{cases} \tag{12}$$

Here, $p_{i,\ell}^{\text{best}}$, l_ℓ^{best}, and g^{best} denote the personal best, layer best, and global best programs, respectively. In words, HER may reuse the previous improvement direction as an inertia signal, retrieve high-quality references from the same particle or higher-level population as cognitive/social signals, inject

an explicit exploration hint in the lowest layer, and append diagnostic feedback when the current candidate fails execution or validation.

The inclusion probabilities for the cognitive and social signals are controlled by adaptive mapping functions based on relative fitness differences. We first define

$$\Delta(f_a, f_b) = \frac{|f_a - f_b|}{\max(f_a, f_b, \epsilon)}, \tag{13}$$

where ϵ is a small constant (e.g., 10^{-6}) preventing division by zero when fitness values are close to zero.

Cognitive Mapping. The function $\phi(\cdot)$ determines the threshold associated with the personal-best signal:

$$\phi(c_{1,\ell}) = c_{1,\ell} \cdot \min\left(1.0, 0.5 + \Delta\left(f_{i,\ell}^{(t)}, f_{i,\ell}^{\text{best}}\right)\right). \tag{14}$$

Layer-social Mapping. The function $\psi_\ell(\cdot)$ determines the threshold associated with the layer-best reference:

$$\psi_\ell(c_{2,\ell}) = c_{2,\ell} \cdot \min\left(1.0, 0.5 + \Delta\left(f_{i,\ell}^{(t)}, f_{\ell}^{\text{best}}\right)\right). \tag{15}$$

Global-social Mapping. For layers $\ell \geqslant 2$, the function $\psi_{\text{glob}}(\cdot)$ additionally controls access to the global best reference, with an extra factor 0.5 so that layer-level guidance remains dominant:

$$\psi_{\text{glob}}(c_{2,\ell}) = 0.5 \cdot c_{2,\ell} \cdot \min\left(1.0, 0.5 + \Delta\left(f_{i,\ell}^{(t)}, f^{\text{global}}\right)\right). \tag{16}$$

Since the implementation samples inclusion using conditions of the form $r > \phi(\cdot)$ or $r > \psi(\cdot)$ with $r \sim \mathcal{U}(0,1)$, the effective inclusion probability is $1 - \phi(\cdot)$ or $1 - \psi(\cdot)$. Consequently, larger mapping values correspond to lower inclusion probabilities. This design reduces over-reliance on a reference when the current particle is still far away from it, thereby avoiding overly aggressive imitation and helping preserve diversity. Conversely, when the fitness gap is smaller, the corresponding reference is more likely to be included, which encourages focused refinement around already promising solutions.

E Illustrative Offspring Examples

```
# EVOLVE-BLOCK-START
import numpy as np

def construct_packing():
    """Constructor-based circle packing for n=26 circles

    Returns:
        Tuple of (centers, radii, sum_of_radii)
        centers: np.array of shape (26, 2) with (x, y)
        coordinates radii: np.array of shape (26) with
        radius of each circle
        sum_of_radii: Sum of all radii
    """
    centers, radii = [], []
    centers.append([0.5, 0.5])
    radii.append(0.5)

    for i in range(25):
        a = 2 * np.pi * i / 25
        centers.append([0.5 + 0.5 * np.cos(a), 0.5 + 0.5 *
        np.sin(a)])
        radii.append(0.0395)

    centers, radii = np.array(centers), np.array(radii)

    return centers, radii, np.sum(radii)

# EVOLVE-BLOCK-END
```

(a) parent program

```
# EVOLVE-BLOCK-START
import numpy as np

def construct_packing():
    """Constructor-based circle packing for n=26 circles

    Returns:
        Tuple of (centers, radii, sum_of_radii)
        centers: np.array of shape (26, 2) with (x, y)
        coordinates radii: np.array of shape (26) with
        radius of each circle
        sum_of_radii: Sum of all radii
    """
    n = 26
    centers, radii = [], []
    radius = 0.07

    for i in range(5):
        for j in range(6):
            if len(centers) < n:
                x = 0.1 + i * 0.2
                y = 0.1 + j * 0.15
                centers.append([x, y])
                radii.append(radius)

    return np.array(centers), np.array(radii), np.sum(radii)

# EVOLVE-BLOCK-END
```

(b) offspring program based on **Full Rewrites**

```
# EVOLVE-BLOCK-START
import numpy as np

def construct_packing():
    """Constructor-based circle packing for n=26 circles

    Returns:
        Tuple of (centers, radii, sum_of_radii)
        centers: np.array of shape (26, 2) with (x, y)
        coordinates radii: np.array of shape (26) with
        radius of each circle
        sum_of_radii: Sum of all radii
    """
    centers, radii = [], []
    centers.append([0.5, 0.5])
    radii.append(0.5)

    for i in range(25):
        a = 2 * np.pi * i / 25
<<<<<<< SEARCH
-       centers.append([0.5 + 0.5 * np.cos(a), 0.5 + 0.5 *
-       np.sin(a)])
-       radii.append(0.0395)
=======
+       r, c, s = 0.0395, np.cos(a), np.sin(a)
+       centers.append([min(max(0.5 + 0.5 * c, r), 1 - r),
+       min(max(0.5 + 0.5 * s, r), 1 - r)])
+       radii.append(r)
>>>>>>> REPLACE
    centers, radii = np.array(centers), np.array(radii)

    return centers, radii, np.sum(radii)

# EVOLVE-BLOCK-END
```

(c) offspring program based on **Diff-Based Edits**

Figure 4: Illustrative examples of offspring candidates c in our framework. The LLM can either generate a complete offspring program or propose targeted edits in `SEARCH/REPLACE` format.

Table 7: Comparison on six mathematical benchmarks with Gemini-3-Pro as the backbone model. Results for compared methods are obtained from the best scores reported in (Cemri et al., 2026; Liu et al., 2026). The top score in each column is in **bold**, and the second-highest is underlined.

Method	Circle P. (Sq.)↑ N=26	Circle P. (Rect.)↑ N=21	Heilbronn (Tri.)↑ T=11	Heilbronn (Conv.)↑ C=13	MinMax Dist.↓ D=3	Signal Proc.↑ –
Human/SOTA	2.6340	2.3640	0.0360	0.0306	4.1680	N/A
AlphaEvolve	2.6350	2.3658	0.0365	0.0309	4.1658	N/A
EvoX	2.6359	**2.3658**	**0.0365**	0.0296	4.1600	0.7429
OpenEvolve	2.5414	**2.3658**	0.0350	0.0280	4.1600	0.5649
ShinkaEvolve	2.6358	**2.3658**	0.0356	0.0287	4.1600	0.5049
AdaEvolve	2.6360	**2.3658**	0.0360	0.0290	**4.1597**	0.6030
HER (Ours)	**2.6360**	**2.3658**	**0.0365**	**0.0309**	4.1658	**0.7542**

Table 8: Comparison on five system benchmarks with Gemini-3-Pro and gpt-oss-120b as the backbone models. Gemini-3-Pro results for compared methods are taken from the reported scores in (Cemri et al., 2026; Liu et al., 2026), whereas the gpt-oss-120b results are obtained from our own runs. For both backbone models, we report both the mean and the best result over three runs. The top score in each column is in **bold**, and the second-highest is underlined.

Method	EPLB↑		LLM-SQL↑		Prism↑		TXN↑		Cloudcast↑	
	Mean	Best	Mean	Best	Mean	Best	Mean	Best	Mean	Best
Human/SOTA	-	0.1400	-	0.6600	-	22.3400	-	3080.0000	-	0.0016
Backbone: Gemini-3-Pro										
EvoX	0.1392	0.1453	0.7278	0.7300	26.2600	26.2600	4267.7500	4310.3400	0.0016	0.0016
OpenEvolve	0.1272	0.1272	0.7251	0.7258	26.2400	26.2400	4109.1900	4273.5000	0.0014	0.0014
ShinkaEvolve	0.1206	0.1272	0.7210	0.7212	26.2500	26.2600	3931.6700	4255.3200	0.0011	0.0010
AdaEvolve	0.1450	0.1453	0.7430	0.7520	26.2600	26.2600	4221.0000	4310.0000	0.0016	0.0016
Backbone: gpt-oss-120b										
OpenEvolve	0.1450	0.1516	0.7330	0.8098	25.2972	25.4687	3817.7837	3861.0039	0.0048	0.0126
ShinkaEvolve	0.1769	0.2238	0.7364	0.7434	26.2133	26.2332	4040.8695	4184.1004	0.0013	0.0017
ContextEvolve	0.1461	0.1488	0.7140	0.7244	25.7013	25.8040	3953.3442	4000.0000	0.0085	0.0087
AdaEvolve	0.1274	0.1286	0.7197	0.7380	26.2560	26.2560	4106.7866	4255.3191	0.0160	0.0176
HER (Ours)	**0.2051**	**0.2359**	**0.8996**	**0.9780**	**26.5030**	**26.8957**	**4256.6840**	**4587.1560**	**0.0190**	**0.0262**

Figure 4 provides a concrete example of what an offspring candidate c looks like in our framework. In practice, the offspring returned by the LLM can take one of two forms: (1) a complete rewritten program that replaces the parent candidate, or (2) a compact set of `SEARCH/REPLACE` edits that modifies only selected code regions. This distinction is important because the two forms support different exploration behaviors: full rewrites encourage broader exploration and larger semantic jumps, whereas diff-style edits enable more localized refinement around a promising parent.

F Additional Results

F.1 Mathematical Results with Gemini-3-Pro

Table 7 reports results on six mathematical benchmarks with Gemini-3-Pro as the backbone. HER achieves the best or tied-best performance on five benchmarks, covering both Circle Packing tasks, both Heilbronn variants, and Signal Processing. In particular, HER obtains the highest Signal Processing score of 0.7542, surpassing the previous best result of 0.7429. On MinMax Distance, HER achieves 4.1658, which remains close to the best result of 4.1597. These results demonstrate that the benefits of hierarchical evolutionary search extend to a high-capability backbone and are not specific to a single model family.

F.2 System Results across Backbone Models

Table 8 consolidates results on five system benchmarks under two backbone settings. For Gemini-3-Pro, the baselines follow previous studies (Cemri et al., 2026; Liu et al., 2026) with a 100-iteration budget. Their relatively small spread is consistent with a high-capability regime in which the

Table 9: Exact one-sided paired Wilcoxon signed-rank tests comparing HER with each evolutionary baseline on gpt-oss-120b task-level means. Wins count positive direction-aligned differences, and W^+ is their rank sum. Statistically significant p-values ($p<0.05$) are **bolded**.

Suite	Baseline	Wins	W^+	Exact one-sided p
Mathematical (N=6)	OpenEvolve	6/6	21	$\mathbf{0.0156}$
	ShinkaEvolve	6/6	21	$\mathbf{0.0156}$
	ContextEvolve	6/6	21	$\mathbf{0.0156}$
	AdaEvolve	6/6	21	$\mathbf{0.0156}$
System (N=5)	OpenEvolve	5/5	15	$\mathbf{0.0313}$
	ShinkaEvolve	5/5	15	$\mathbf{0.0313}$
	ContextEvolve	5/5	15	$\mathbf{0.0313}$
	AdaEvolve	5/5	15	$\mathbf{0.0313}$
All (N=11)	OpenEvolve	11/11	66	$\mathbf{4.88\times10^{-4}}$
	ShinkaEvolve	11/11	66	$\mathbf{4.88\times10^{-4}}$
	ContextEvolve	11/11	66	$\mathbf{4.88\times10^{-4}}$
	AdaEvolve	11/11	66	$\mathbf{4.88\times10^{-4}}$

backbone leaves limited room for the search strategy to differentiate performance. Under gpt-oss-120b (Section 4.1), the differences between strategies become clearer across both mean and best metrics, and HER attains results comparable to or better than the reported Gemini-3-Pro baselines on several tasks. This observation suggests that structured search can recover substantial performance without relying exclusively on the strongest backbone. Because the two settings differ in model, implementation, and result source, however, we do not interpret this comparison as a controlled demonstration of cross-model cost superiority.

G Statistical Significance Analysis

We treat each benchmark as one paired observation and use the mean over three independent runs reported in Tables 2 and 3. To align metric directions, we retain maximization metrics and negate the minimization metric (MinMax Distance), so that larger values are uniformly better. We exclude Human/SOTA and AlphaEvolve because they are external references rather than methods evaluated under our protocol. For each baseline b, we define

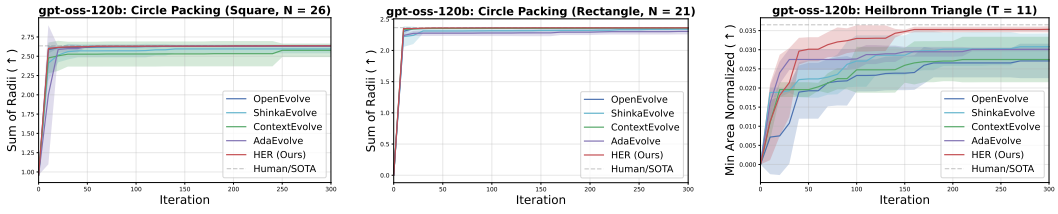
$$d_i^{(b)} = \tilde{s}_i^{\mathrm{HER}} - \tilde{s}_i^{(b)}, \qquad i = 1, \ldots, N,$$

where $\tilde{s}_i$ is the direction-aligned mean for task i. We test H_1: $\mathrm{median}(d^{(b)}) > 0$ using an exact one-sided paired Wilcoxon signed-rank test. No paired difference is zero. Because all N differences are positive in every comparison, $W^+ = N(N+1)/2$ and the exact one-sided p-value is 2^{-N}.

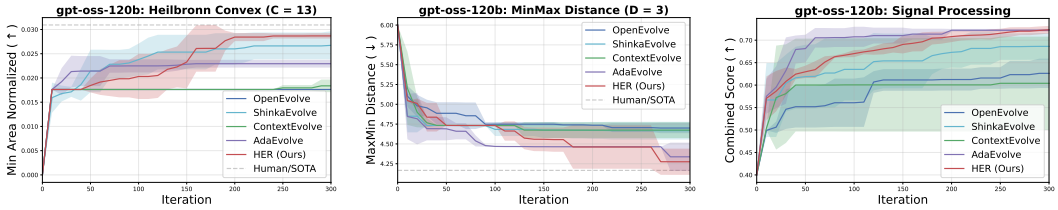
As shown in Table 9, HER achieves a higher direction-aligned mean on every task against every baseline: $6/6$ on the mathematical suite, $5/5$ on the system suite, and $11/11$ overall. The corresponding exact one-sided p-values are 0.0156, 0.0313, and 4.88×10^{-4}, respectively. The four combined-suite comparisons remain significant after Bonferroni correction (α=$0.05/4$=0.0125). These results support a consistent suite-level advantage for HER under gpt-oss-120b; they do not establish significance for individual tasks with only three runs. In particular, they should not be interpreted as evidence of per-task significance for near-ties such as MinMax Distance and Signal Processing, where the mean differences from AdaEvolve are negligible.

H Convergence Behavior Analysis

In this section, we analyze run-time convergence behavior under the gpt-oss-120b backbone across two benchmark groups: mathematical tasks and system tasks. Compared with flat evolutionary baselines, HER generally exhibits faster early-stage improvements and stronger late-stage stabilization, indicating a more effective exploration–exploitation balance throughout optimization.

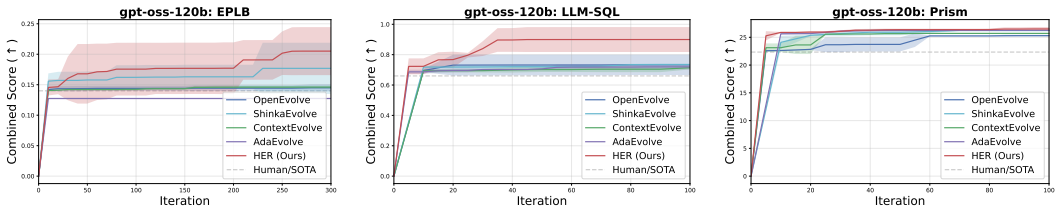


(a) Circle Packing (Square) (b) Circle Packing (Rectangle) (c) Heilbronn Triangle

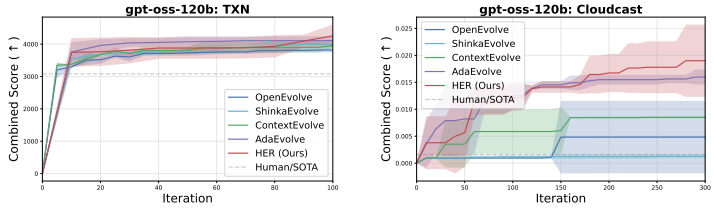


(d) Heilbronn Convex (e) MinMax Distance (f) Signal Processing

Figure 5: Convergence comparison on six mathematical benchmarks using gpt-oss-120b.



(a) EPLB (b) LLM-SQL (c) Prism

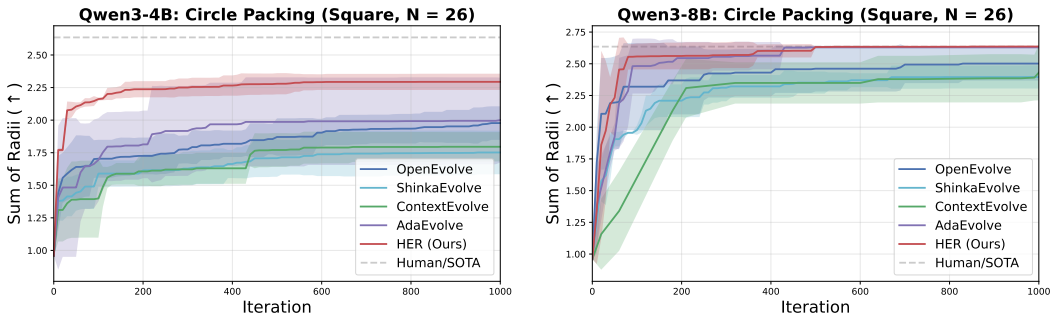


(d) TXN (e) Cloudcast

Figure 6: Convergence comparison on five ADRS benchmarks using gpt-oss-120b.

Mathematical Benchmarks. Figure 5 summarizes the convergence trajectories for Circle Packing (Square), Circle Packing (Rectangle), Heilbronn Triangle, Heilbronn Convex, MinMax Distance, and Signal Processing. A consistent pattern across these tasks is that HER not only enters high-quality solution regions earlier, but also preserves a steadier upward trend after the initial rapid-improvement phase. This behavior is particularly important for mathematical search spaces, where optimization often depends on avoiding early over-commitment to locally attractive but globally suboptimal structures. The observed trajectories therefore suggest that the combination of diverse initialization and hierarchical population coordination improves both early-stage search coverage and late-stage refinement, yielding gains in sample efficiency without sacrificing asymptotic solution quality.

System Benchmarks. Figure 6 reports convergence on EPLB, LLM-SQL, Prism, TXN, and Cloudcast. The same advantage largely transfers to system-oriented optimization, but with an even clearer interpretation: these tasks are shaped by strong constraint coupling, delayed feedback, and high sensitivity to small structural edits. Under such conditions, methods with flat population dynamics are more vulnerable to oscillation or premature stagnation. In contrast, HER consistently remains on or near the strongest trajectories, with particularly visible gains on challenging scheduling



(a) Qwen-4B (b) DeepSeek-8B

Figure 7: Convergence comparison on Circle Packing (Square, $N = 26$) with small backbone models (Qwen3-4B and Qwen3-8B).

and resource-allocation problems such as EPLB and LLM-SQL. This suggests that the hierarchical organization is not merely improving raw search breadth, but is also stabilizing optimization under rugged, non-stationary objective surfaces.

Overall, these convergence results complement the final-score tables by revealing a deeper advantage of HER: the method improves not only the eventual endpoint, but also the quality of the optimization process itself. This process-level evidence strengthens the claim that HER provides a more reliable strategy for long-horizon evolutionary reasoning, rather than merely benefiting from favorable outcomes at termination.

I Convergence Advantage on Small Models

To examine whether the optimization behavior of HER transfers to smaller backbones, we compare convergence trajectories on Circle Packing (Square, $N = 26$) using two compact models: Qwen3-4B-Instruct-2507 (Yang et al., 2025) (Qwen3-4B) and DeepSeek-R1-0528-Qwen3-8B (Guo et al., 2025) (Qwen3-8B). Figure 7 shows that HER reaches strong solutions earlier and maintains a more favorable optimization trend throughout evolution, indicating better sample efficiency under limited model capacity.

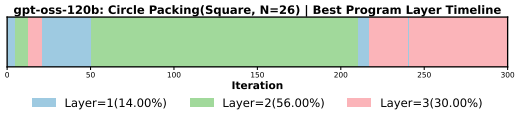
The same pattern appears across both backbones, suggesting that the hierarchical population design and diversity-aware initialization in HER provide robust convergence benefits beyond larger frontier models. These results complement the main-text performance tables and support the claim that HER remains effective in resource-constrained settings.

Concretely, lower layers continue proposing diverse placement candidates, while upper layers selectively absorb and refine high-potential solutions into better-balanced assignments. The resulting timeline reflects persistent cross-layer interaction instead of one-shot convergence, which is consistent with the strong run-level robustness observed on system tasks in the main results.

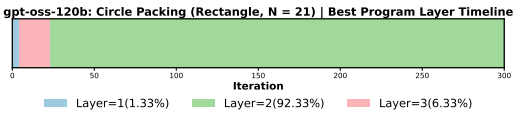
J Layer-wise Evolution of Current Best Programs

We analyze how the *current best program* evolves across hierarchical layers ($L_1 \sim L_3$) as iterations proceed. The timeline plots reveal the point at which the best solution found so far emerges at each layer, reflecting the dynamic interaction between exploration-oriented lower layers and exploitation-oriented upper layers.

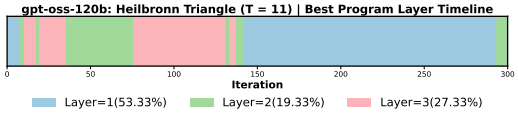
Mathematical Benchmarks. Figure 8 shows that early improvements are often initiated in L_1/L_2, while later-stage best updates increasingly appear in upper layers, especially L_3. This pattern is consistent with the intended behavior of hierarchical evolutionary search: broad search at lower layers followed by progressive refinement at higher layers.



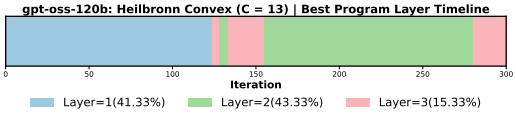
(a) Circle Packing (Square)



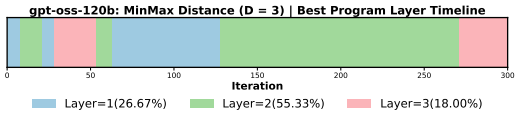
(b) Circle Packing (Rectangle)



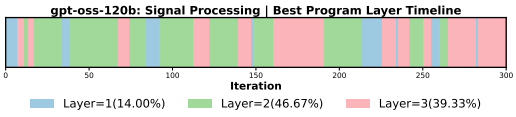
(c) Heilbronn Triangle



(d) Heilbronn Convex

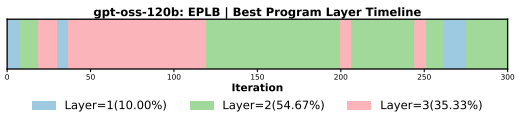


(e) MinMax Distance

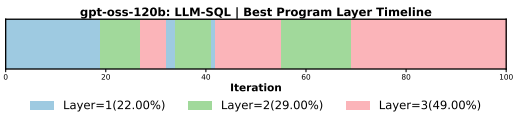


(f) Signal Processing

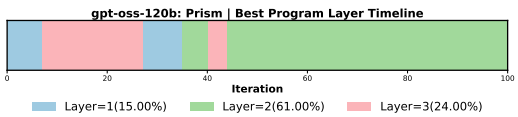
Figure 8: Layer timeline of current best programs ($L_1 \sim L_3$) on mathematical benchmarks.



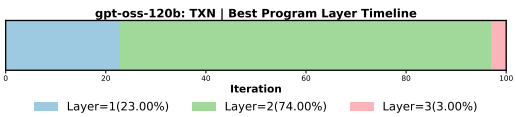
(a) EPLB



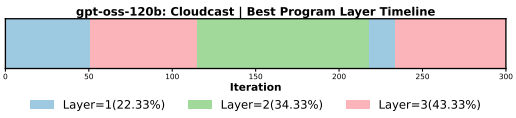
(b) LLM-SQL



(c) Prism



(d) TXN



(e) Cloudcast

Figure 9: Layer timeline of current best programs ($L_1 \sim L_3$) on system benchmarks.

System Benchmarks. Figure 9 presents a similar but task-dependent pattern for system tasks. For relatively smooth tasks, improvements are primarily concentrated in early stages, whereas for scheduling and resource-coupled tasks, updates remain active across layers over extended periods, indicating sustained cross-layer coordination.

Overall, these layer timelines provide process-level evidence that HER does not rely on a single fixed layer, but adaptively shifts the source of best solutions over time.

K Case Study I: Layer-wise Best-Program Trajectory Analysis

K.1 Circle Packing (Square, N = 26)

Circle Packing (Square, $N = 26$) provides a representative view of how hierarchical evolutionary search allocates search effort over time. In the corresponding layer timeline (the Circle Packing (Square) panel in Figure 8 (a)), early best updates are primarily triggered in lower layers (0.9598 $\rightarrow$ 2.5100), indicating that $\mathrm{L}_1/\mathrm{L}_2$ effectively perform broad structural exploration in the initial stage.

As evolution proceeds, best-so-far improvements progressively shift toward upper layers (2.5100 → 2.6360), showing that L_3 increasingly acts as a refinement layer for high-quality candidates.

This transition pattern matches the design intuition of our proposed HER: lower layers diversify candidate structures, while upper layers consolidate and polish promising programs. Compared with flat evolution, the observed multi-stage transfer of "best-program ownership" across layers suggests a more stable exploration-to-exploitation pipeline, which helps avoid premature stagnation while maintaining late-stage progress.

K.2 EPLB (Expert Placement Load Balancing)

EPLB is a system-level optimization task with coupled placement and load-balance constraints, making it a useful case to examine layer coordination under non-trivial scheduling dynamics. In the EPLB panel of Figure 9 (a), current-best updates are distributed across layers for a longer portion of the run, rather than collapsing too early to a single layer. This indicates that different layers contribute complementary search behaviors throughout optimization.

L Case Study II: Evolutionary Lineage Trajectory Analysis

To provide a more concrete view of how hierarchical evolution unfolds in practice, Figure 10 traces the full lifecycle of a single particle on the Signal Processing task, highlighting how diff-based edits, full rewrites, and cross-layer transitions jointly shape its improvement trajectory.

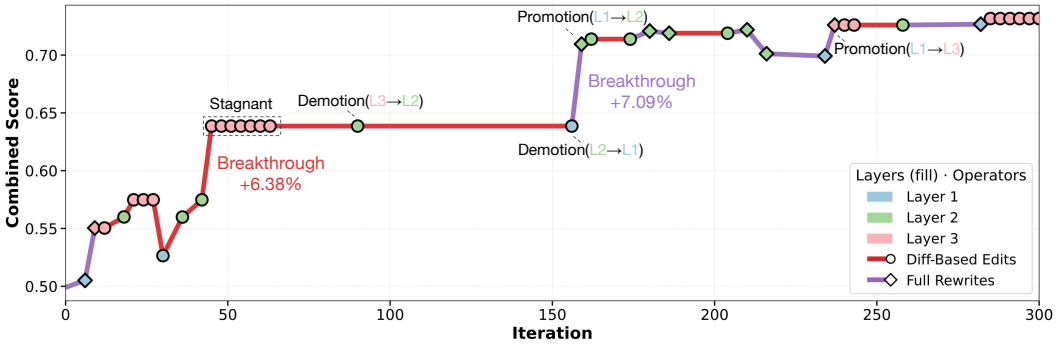


Figure 10: Evolutionary lineage trajectory of the final global-best solution on the Signal Processing task. Marker fill colors denote the destination layer ($L_1 \sim L_3$) after evolution, while line colors and endpoint marker shapes indicate the applied operator types (Diff-Based Edits and Full Rewrites). This trajectory illustrates how HER alternates between localized exploitation and large-scale exploratory jumps through reversible cross-layer transitions.

Edits and rewrites play complementary roles. The first breakthrough at iteration 42 (Appendix P.1) is triggered by diff-based edits, which improve the score by 6.38% without changing the particle's overall search regime. This is consistent with Section 4.4: localized edits are particularly effective once a particle has already entered a promising region, because they preserve useful structure while making targeted corrections. In contrast, the second major jump at iteration 156 (Appendix P.2) is produced by a full rewrite, indicating that when refinement saturates, larger semantic jumps remain necessary to escape local stagnation and discover a stronger solution family.

Layer transitions act as a recovery mechanism rather than a one-way pipeline. This particle does not improve monotonically as it moves upward. Instead, it experiences a long stagnant phase in the upper layer, is demoted back to lower layers, and only then produces another large gain before being promoted again. This behavior complements the migration patterns reported in Figure 3: adjacent-layer transitions allow HER to "recycle" stalled particles by returning them to more exploratory regimes, after which improved candidates can be reabsorbed into higher layers for stabilization. The hierarchy therefore functions not only as a promotion ladder, but as an adaptive feedback loop that couples refinement with recovery.

M Comparison of Computational Cost and Token-Matched Performance

M.1 Computational Cost Analysis

For all main experiments, we impose a common cap on outer-loop candidate updates and scheduled evaluations. Additionally, we record LLM calls, generated tokens, and wall-clock time for all methods on the 11 benchmarks using gpt-oss-120b. Specifically, generated tokens measure model-side inference volume, while wall-clock time also reflects parallelism, server load, restarts, and timeouts. Tables 10 and 11 summarize the results of two representative runs, respectively.

Table 10: Mean per-run compute accounting on Circle Packing (Square). Lowest and second-lowest token totals are **bolded** and underlined.

Method	Calls	Tokens	tok/upd	call/upd	Wall (h)
AdaEvolve	348	**5.69M**	19.0k	1.16	8.3
ContextEvolve	1028	6.94M	23.1k	3.43	1.6
HER	728	7.20M	24.0k	2.43	9.7
OpenEvolve	311	7.69M	25.4k	1.03	32.9*
ShinkaEvolve	718	15.24M	50.8k	2.39	30.1*

Table 11: Mean per-run compute accounting on LLM-SQL. Lowest and second-lowest token totals are **bolded** and underlined.

Method	Calls	Tokens	tok/upd	call/upd	Wall (h)
HER	254	**1.61M**	16.1k	2.54	1.5
AdaEvolve	116	2.42M	24.2k	1.16	2.1
ContextEvolve	307	2.93M	29.3k	3.07	3.8
ShinkaEvolve	231	3.88M	38.8k	2.31	1.9
OpenEvolve	100	4.83M	48.3k	1.00	0.6

*Wall-clock values may include restart or timeout outliers and should not be interpreted as hardware-normalized runtime.

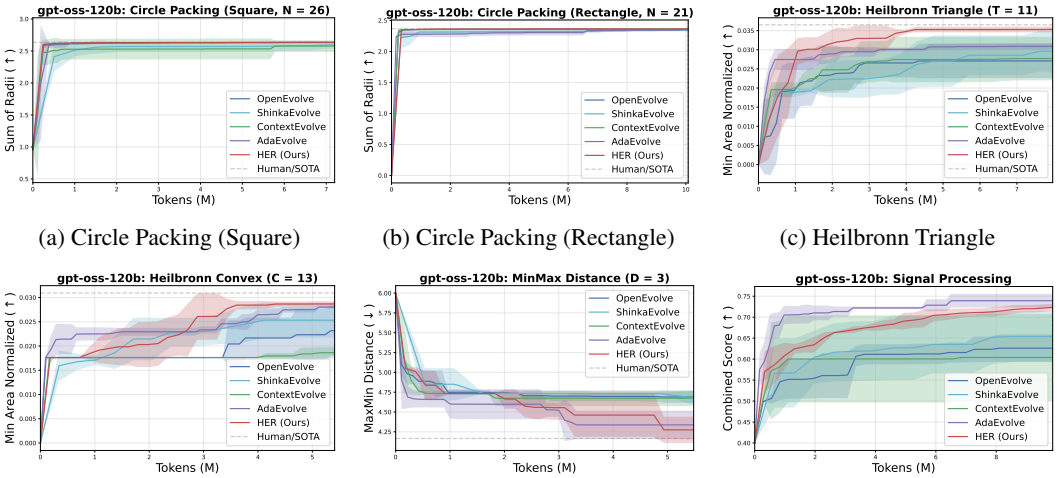
As shown in the tables, for the Circle Packing (Square) task, HER utilizes 728 calls and 7.20M generated tokens, averaging 2.43 calls and 24.0k tokens per update. Its total token usage exceeds that of AdaEvolve and ContextEvolve, but remains lower than that of OpenEvolve and ShinkaEvolve. For the LLM-SQL task, HER consumes 254 calls and 1.61M tokens, achieving the lowest total token count among the five methods. While the number of LLM calls and token usage are not strictly proportional across methods due to differing evolutionary strategies, HER demonstrates a significant advantage in token efficiency and time cost. This is accomplished by optimizing prompts and enforcing concise model outputs, despite requiring an average of two calls per outer loop iteration.

Table 12: Comparison on eleven benchmarks with gpt-oss-120b as the backbone model under a unified token-matched budget. We report mean$_{\pm \text{std}}$ over three runs for each methods. The best and second-best means are **bolded** and underlined.

Task	OpenEvolve	ShinkaEvolve	ContextEvolve	AdaEvolve	HER (Ours)
Mathematical benchmarks					
Circle Packing (Square) ↑	2.6313±0.0062	2.5947±0.0566	2.5768±0.0881	2.6338±0.0025	**2.6354**±0.0010
Circle Packing (Rectangle) ↑	2.3583±0.0022	2.3399±0.0171	2.3610±0.0044	2.3497±0.0061	**2.3619**±0.0020
Heilbronn (Triangle) ↑	0.0271±0.0045	0.0296±0.0053	0.0277±0.0057	0.0309±0.0009	**0.0353**±0.0008
Heilbronn (Convex) ↑	0.0232±0.0052	0.0253±0.0036	0.0186±0.0012	0.0280±0.0003	**0.0287**±0.0005
MinMax Distance ↓	4.6897±0.0733	4.6836±0.0839	4.6742±0.1002	4.3374±0.1793	**4.2749**±0.1657
Signal Processing ↑	0.6261±0.0327	0.6539±0.0609	0.6040±0.1039	**0.7393**±0.0161	0.7232±0.0086
System benchmarks					
EPLB ↑	0.1443±0.0047	0.1630±0.0198	0.1460±0.0024	0.1274±0.0011	**0.2051**±0.0394
LLM-SQL ↑	0.7315±0.0682	0.7207±0.0093	0.7018±0.0159	0.7181±0.0172	**0.8996**±0.0820
Prism ↑	25.48±0.18	25.91±0.56	25.83±0.02	26.26±0.00	**26.50**±0.34
TXN ↑	3827.1±58.7	3853.2±111.6	3953.3±67.5	4106.8±132.7	**4256.7**±349.4
Cloudcast ↑	0.0048±0.0067	0.0012±0.0004	0.0085±0.0002	0.0160±0.0014	**0.0190**±0.0067

M.2 Token-Matched Performance

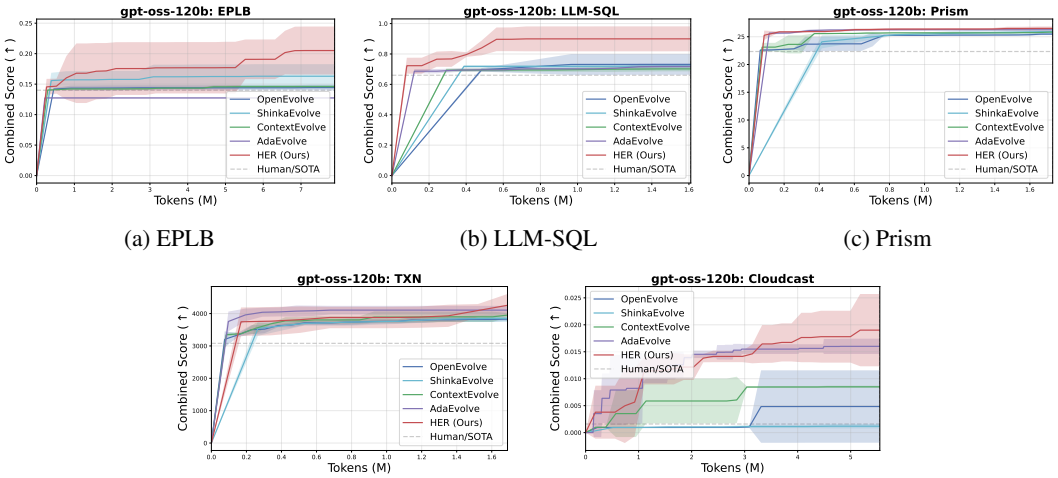
We further compare solution quality under a unified token-matched budget. For a method with U original updates and total tokens T_{method}, we allocate approximately $\lceil T_{\text{HER}}/T_{\text{method}} \rceil U - U$ additional updates plus a five-update buffer, using three seeds. Methods whose initial token usage already meets or exceeds that of HER are not extended.



(a) Circle Packing (Square) (b) Circle Packing (Rectangle) (c) Heilbronn Triangle

(d) Heilbronn Convex (e) MinMax Distance (f) Signal Processing

Figure 11: Comparison on six mathematical benchmarks with gpt-oss-120b as the backbone model under a unified token-matched budget.



(a) EPLB (b) LLM-SQL (c) Prism

(d) TXN (e) CloudCast

Figure 12: Comparison on five system benchmarks with gpt-oss-120b as the backbone model under a unified token-matched budget.

Table 12 reports performance at the estimated HER token budget. HER achieves the best mean result on five of the six mathematical benchmarks and all five system benchmarks. On Signal Processing, AdaEvolve obtains 0.7393, compared with 0.7232 for HER. Figures 11 and 12 plot the corresponding best-so-far trajectories against estimated cumulative tokens. Because cumulative tokens are reconstructed from run-level averages, the reported token alignment is approximate.

N Configuration and Hyperparameters

The computation was performed on a single node equipped with 2 NVIDIA Tesla H100 GPUs, each with 80GB of VRAM. We summarize the key experimental settings used in this paper in Table 13. Unless otherwise stated, the same configuration is applied across all math and ADRS benchmarks. To assess robustness, we vary the two most structural choices, namely the L_2/L_3 promotion thresholds and the $L_1/L_2/L_3$ capacity ratios, while holding L=3, initialization, mutation operators, backbone, and all other parameters fixed.

Table 13: System configuration and hyperparameters used in our experiments.

Parameter	Value	Parameter	Value
LLM settings			
Backbone model	Gemini-3-Pro / gpt-oss-120b	Max retries	3
Temperature	0.7	Top-p	0.95
Max output tokens	32000	API timeout	600s
Database / Layer			
Population size	10	Number of layers	3 ($L_1 \sim L_3$)
L_1: w / c_1 / c_2	0.2 / 0.2 / 0.3	L_1 particle ratio	0.6
L_1 promotion threshold	0.00	L_1 elimination stagnation	3
L_2: w / c_1 / c_2	0.5 / 0.4 / 0.4	L_2 particle ratio	0.3
L_2 promotion threshold	0.90	L_2 demotion stagnation	4
L_3: w / c_1 / c_2	0.8 / 0.5 / 0.2	L_3 particle ratio	0.1
L_3 promotion threshold	0.98	L_3 demotion stagnation	5
Evolution / Evaluation			
Max iterations	100/300	Parallel evaluations	3
Evaluator timeout	360s	Cascade evaluation	False

Sensitivity and Practical Guidance. Table 14 shows that Circle Packing (Square) varies by less than 0.05% from the default mean, while all Prism variants remain within approximately 1%. TXN is more sensitive: the tested changes reduce its mean score by roughly 10–12%. We therefore do not claim invariance to arbitrary retuning. For a new task, we recommend starting from the shared defaults (L=3, ratios $0.6/0.3/0.1$, promotion $0.90/0.98$), loosening promotion only when upper layers remain underfilled, and tightening it when candidates move upward too rapidly. Any change should be selected on held-out development runs before evaluation. This analysis is targeted rather than exhaustive; the stagnation controls and layer-wise context coefficients remain fixed.

Table 14: Sensitivity to promotion thresholds and layer ratios. One factor is varied at a time. Best and second-best means are **bolded** and underlined.

Setting	Promotion (L_2/L_3)	Ratio ($L_1/L_2/L_3$)	Circle P. (Sq.) ↑	Prism ↑	TXN ↑
Default	0.90/0.98	0.6/0.3/0.1	2.6354±0.0010	26.50±0.34	**4256.7**±349.4
↳ Looser promotion	0.85/0.95	0.6/0.3/0.1	2.6354±0.0010	26.37±0.20	3788.6±63.1
↳ Tighter promotion	0.95/0.99	0.6/0.3/0.1	**2.6360**±0.0000	26.26±0.00	3736.8±69.6
↳ More exploration	0.90/0.98	0.7/0.2/0.1	2.6342±0.0030	**26.55**±0.39	3765.9±101.0
↳ More exploitation	0.90/0.98	0.5/0.3/0.2	2.6350±0.0017	26.38±0.22	3837.1±66.5

O Prompt Templates

We provide a detailed description of the prompts utilized by HER.

Initialization Insights Prompt

```
# Task Description
{task_description}

For this task, what are the characteristics of a good initialization and optimization strategy? Present several key insights in a list format.

Output in a json format:
{ "thinking": "...",
"key_insights": ["...", "...", "...", "..."] }
```

Full Rewrites Evolution Prompt

```
# Current Program Information
- Fitness: {fitness_score}

# Program Evolution History
{evolution_history}

# Reference Programs
{reference_programs}

# Current Program
```{language}
{current_program}
```

Task
Rewrite the program to improve its FITNESS SCORE.
Different solutions with similar fitness but different features are valuable.
Provide the complete new program code.

Notes
You may only modify text that lies below a line containing "EVOLVE-BLOCK-START" and above the next "EVOLVE-BLOCK-END". Everything outside those markers is read-only. To ensure safe evaluation, the code outside of "EVOLVE-BLOCK-START" and "EVOLVE-BLOCK-END" must be copied to the new program code unchanged.
* Keep the markers "EVOLVE-BLOCK-START" and "EVOLVE-BLOCK-END" in the code. Do not change the code outside of these markers.
* Do not embed your thought process in the code as comments. Keep the code clean.
* 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.
* The code must be surrounded by the ```{language}...``` block.

Begin!

```{language}
# Your rewritten program here
```
```

Evolution Suggestion (Velocity) Prompt

```
You are a code optimization expert. Analyze and provide improvement directions.

# Task
Analyze the reference programs and current program, then generate a concise improvement direction that guides the current program toward the reference solutions.

# Velocity Context
## Current Program Information
- Fitness: {fitness_score}
- Program:
```{language}
{current_program}
```

Continue Previous Improvement Direction (optional, inertia term)
{current_velocity}

Reference Personal Best (optional, cognitive term)
- Fitness: {pbest.fitness_score}
- Program:
```
```
```

{pbest.program}
```

## Reference Layer Best (optional, social term)
- Fitness: {lbest.fitness_score}
- Program:
```{language}
{lbest.program}
```

Reference Global Best (optional, for layer >= 2)
- Fitness: {gbest.fitness_score}
- Program:
```{language}
{gbest.program}
```

Current Program is Running with Error (optional, if current has error)
{error_msg}
Analyze and provide improvement directions.

Exploration Hint (optional, for layer==1)
- Hint: {random_exploration_hint}

Items to Consider
* Differences: What are the main differences between the current program and the reference programs? What makes the references better?
* Optimization Suggestion: How should the current code be modified to incorporate the strengths from the reference programs? Be specific about which techniques, patterns, or optimizations from the references should be applied.

Focus on learning from the reference programs. Your response should guide the code evolution toward the better solutions shown in the references. The output format should be a markdown table with two columns: "Differences" and "Optimization Suggestion".

Diff-Based Edits Evolution Prompt

Current Program Information
- Fitness: {fitness_score}

Program Evolution History
{evolution_history}

Reference Programs
{reference_programs}

Current Program
```{language}
{current_program}
```

Task
Suggest improvements to the program that will improve its FITNESS SCORE.
Different solutions with similar fitness but different features are valuable.
You MUST use the exact SEARCH/REPLACE diff format shown below to indicate changes:

```
<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=======
# New replacement code
>>>>>>> REPLACE
```

Example of valid diff format:
```

```
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE
```

Notes
* You may only modify text that lies below a line containing "EVOLVE-BLOCK-START" and above the next "EVOLVE-BLOCK-END". Everything outside those markers is read-only.
* Do not repeat the markers "EVOLVE-BLOCK-START" and "EVOLVE-BLOCK-END" in the SEARCH/REPLACE blocks.
* Every block's SEARCH section must be copied **verbatim** from the current file. Including indentation.
* You can suggest multiple changes. Each SEARCH section must exactly match code in the current program.
* Make sure the file still runs after your changes.
* Be thoughtful about your changes and explain your reasoning thoroughly.

Evolution Hints
{evolution_suggestion}

IMPORTANT: Do not rewrite the entire program - focus on targeted improvements.

Error-Containing Candidate Evolution Prompt

Trigger: When the candidate program encounters a runtime error, the objective of the evolution shifts to bug fixing.

You are a code optimization expert. Current program is running with error. Analyze and provide improvement directions.

Current Program Information
- Fitness: {fitness_score}

Current Program
```{language}
{current_program}
```

Error Details
{error_info}

Task
Analyze the provided error and identify the root cause.
Propose minimal changes to fix the issue.
You MUST use the exact SEARCH/REPLACE diff format shown below to indicate changes:

```
<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=======
# New replacement code
>>>>>>> REPLACE

Example of valid diff format:
<<<<<<< SEARCH
for i in range(m):
```

```
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE
```

If multiple changes are needed, provide separate SEARCH/REPLACE blocks.
Maintain the original programming style and conventions.
Be thoughtful about your changes and explain your reasoning thoroughly.

IMPORTANT: Do not rewrite the entire program - focus on program correctness.

Retry-Time Error Fix Prompt

Trigger: When a newly generated program returns an execution error during evaluation, and retry budget remains.

Appended User Suffix:
Previous Error
The previous attempt resulted in an error: {error_msg}

Code That Caused the Error
```{language}
{error_program}
```

Please fix this error in your code.

P Code Examples of Evolutionary Breakthroughs

P.1 Diff-Based Edit at Iteration 42

Breakthrough via Diff-Based Edits at 42 Iteration

```python
# EVOLVE-BLOCK-START
"""
Real-Time Adaptive Signal Processing Algorithm for Non-Stationary Time Series

This version replaces the simple Kalman filter with a lightweight
local-linear-trend (LLT) Kalman filter and adds a zero-phase
forward-backward smoothing pass.  The filter adapts its process and
measurement noise statistics over a sliding window, preserving trends
while reducing phase lag, spurious reversals and lag error.
"""

import numpy as np
from dataclasses import dataclass

# Optional Numba import - if unavailable we fall back to a no-op decorator.
# This enables JIT compilation of hot-spot functions when possible,
# improving runtime performance without breaking compatibility.
try:
    from numba import njit
except Exception:  # pragma: no cover
    def njit(pyfunc=None, **kwargs):
        """Fallback decorator that returns the original function unchanged."""
        if pyfunc is None:
            def wrapper(func):
                return func
            return wrapper
```

```
        return pyfunc

# ---------------------------------------------------------------------
# Helper: robust median-absolute-deviation (MAD)
# ---------------------------------------------------------------------
def _mad(arr: np.ndarray) -> float:
    """Return the median absolute deviation of *arr*."""
    med = np.median(arr)
    return np.median(np.abs(arr - med))


def adaptive_filter(x: np.ndarray, window_size: int = 20) -> np.ndarray:
    """
    Baseline adaptive filter - simple moving average using cumulative sum.

    Args:
        x: Input signal (1-D array)
        window_size: Size of the sliding window

    Returns:
        y: Filtered output (len(x) - window_size + 1)

    Notes:
        If the input length is smaller than ``window_size`` an empty
        array is returned, matching the behaviour of the enhanced filter.
    """
    if len(x) < window_size:
        # Return an empty array for consistency with the enhanced filter
        return np.array([], dtype=float)

    # cumulative sum trick for O(N) moving average
    csum = np.cumsum(np.insert(x, 0, 0.0))
    y = (csum[window_size:] - csum[:-window_size]) / float(window_size)
    return y


@dataclass
class KalmanConfig:
    """
    Configuration container for the Local-Linear-Trend Kalman filter.

    Attributes
    ----------
    window_size : int
        Size of the sliding window used for adaptive noise estimation.
    process_factor : float
        Scaling factor applied to the variance of recent level differences
        when computing the process-noise variance for the level state.
    min_var : float
        Minimum allowed variance to avoid numerical issues.
    init_cov : float
        Initial covariance for both state dimensions.
- adapt_interval : int
-     How often (in samples) the adaptive noise statistics are updated.
-     Updating every sample can be noisy and expensive; a value of
-     ``window_size // 2`` gives a good trade-off.
+ q_level_init : float
+     Initial process-noise variance for the level state.
+ q_slope_init : float
+     Initial process-noise variance for the slope state.
    """
    window_size: int = 20
    process_factor: float = 0.1
    min_var: float = 1e-6
    init_cov: float = 1.0
- adapt_interval: int = 10  # will be overwritten in __post_init__
-
- def __post_init__(self):
-     # sensible default: update twice per window
-     if self.adapt_interval <= 0:
-         self.adapt_interval = max(1, self.window_size // 2)
+ q_level_init: float = 1e-5
+ q_slope_init: float = 1e-6

- def _kalman_llt(x: np.ndarray, config: KalmanConfig) -> np.ndarray:
+ def _kalman_llt_smoother(x: np.ndarray, config: KalmanConfig) -> np.ndarray:
    """
- Local-Linear-Trend Kalman filter with window-based adaptive Q and R.
- Returns a full-length estimate of the level (first state component).
```

```
+ Local-Linear-Trend Kalman filter with adaptive Q/R and RTS smoothing.
+ Returns a zero-phase estimate of the level (first state component).

- This implementation uses a rolling variance for the measurement-noise
- estimate, updating it only every ``config.adapt_interval`` samples.
- This reduces computational cost from O(N*W) to O(N) while providing a
- smoother adaptation.
+ The forward pass stores the filtered state, covariance and the
+ predicted quantities required for the Rauch-Tung-Striebel smoother.
    """
    n = len(x)
    w = config.window_size
- adapt_int = max(1, config.adapt_interval)

    # State: [level, slope]
    s = np.array([x[0], 0.0], dtype=float)          # initial state
    P = np.eye(2, dtype=float) * config.init_cov    # initial covariance

    # Model matrices
- A = np.array([[1.0, 1.0],
-               [0.0, 1.0]])                      # level + slope, slope constant
- H = np.array([[1.0, 0.0]])                      # we observe the level only
+ A = np.array([[1.0, 1.0],
+               [0.0, 1.0]], dtype=float)      # level + slope, slope constant
+ H = np.array([[1.0, 0.0]], dtype=float)         # we observe the level only
    I = np.eye(2, dtype=float)

    # Initial noise estimates
    R = np.var(x[:w]) + config.min_var
- q_level = config.min_var
- q_slope = config.min_var
+ q_level = config.q_level_init
+ q_slope = config.q_slope_init

    # Containers for adaptive statistics
- residuals = np.zeros(w, dtype=float)
- est = np.empty(n, dtype=float)
+ # Buffers for adaptive statistics
+ residuals = np.zeros(w, dtype=float)
+
+ # Storage for forward pass (filtered)
+ s_hist = np.empty((n, 2), dtype=float)      # filtered state
+ P_hist = np.empty((n, 2, 2), dtype=float)   # filtered covariance
+
+ # Storage for predictions (needed for RTS)
+ s_pred_hist = np.empty((n, 2), dtype=float)   # predicted state
+ P_pred_hist = np.empty((n, 2, 2), dtype=float)  # predicted covariance

    for i in range(n):
        # ---------- Predict ----------
        s_pred = A @ s
- P_pred = A @ P @ A.T + np.diag([q_level, q_slope])
+ Q = np.diag([q_level, q_slope])
+ P_pred = A @ P @ A.T + Q
+
+ # Store predictions for smoothing
+ s_pred_hist[i] = s_pred
+ P_pred_hist[i] = P_pred

        # ---------- Update ----------
        z = x[i]
        y_res = z - (H @ s_pred)[0]                # innovation
        S = (H @ P_pred @ H.T)[0, 0] + R           # innovation covariance
- K = (P_pred @ H.T / S).flatten()           # Kalman gain (2,)
+ K = (P_pred @ H.T / S).ravel()             # Kalman gain (2,)

        s = s_pred + K * y_res
- P = (I - K[:, None] @ H) @ P_pred
+ P = (I - np.outer(K, H)) @ P_pred
+
+ # Store filtered results
+ s_hist[i] = s
+ P_hist[i] = P

        # ---------- Adaptive noise update (MAD-based) ----------
        # Store latest residual
        residuals[i % w] = y_res

- if i >= w - 1 and (i - (w - 1)) % adapt_int == 0:
```

```
+ if i >= w - 1:
            # Robust measurement-noise estimate using MAD
            R = (_mad(residuals) ** 2) + config.min_var

            # Process-noise for level: based on recent level differences
- recent_levels = est[i - w + 1 : i + 1]
+ recent_levels = s_hist[i - w + 1:i + 1, 0]
            level_diffs = np.diff(recent_levels)
            q_level = (_mad(level_diffs) ** 2) * config.process_factor + config.min_var

            # Process-noise for slope: based on recent slope changes
            # Use a smoother gradient estimate rather than simple differences
            recent_slopes = np.gradient(recent_levels)
            slope_diffs = np.diff(recent_slopes)
            q_slope = (_mad(slope_diffs) ** 2) * config.process_factor + config.min_var

- return est
+ # -------------------- RTS Smoother --------------------------------
+ s_smooth = np.empty_like(s_hist)
+
+ # Initialise with last filtered estimate
+ s_smooth[-1] = s_hist[-1]
+
+ for i in range(n - 2, -1, -1):
+     # Gain for smoothing
+     G = P_hist[i] @ A.T @ np.linalg.inv(P_pred_hist[i + 1])
+     # Smoothed state
+     s_smooth[i] = s_hist[i] + G @ (s_smooth[i + 1] - s_pred_hist[i + 1])
+
+ # Return the smoothed level component
+ return s_smooth[:, 0]


def _post_smooth(signal: np.ndarray) -> np.ndarray:
    """
    Light zero-phase 3-point FIR (1, 2, 1) / 4 using convolution.
    Edge values are repeated to keep the length unchanged.
    """
    if len(signal) < 3:
        return signal.copy()
    # Pad with edge values to keep output length unchanged
    padded = np.pad(signal, (1, 1), mode='edge')
    kernel = np.array([1.0, 2.0, 1.0]) / 4.0
    return np.convolve(padded, kernel, mode='valid')


def enhanced_filter_with_trend_preservation(
    x: np.ndarray,
    window_size: int = 20,
    zero_phase: bool = True,
) -> np.ndarray:
    """
- Enhanced filter using a forward-backward LLT Kalman filter.
- The zero-phase output reduces lag while preserving genuine trends.
- A light post-smoothing FIR (1,2,1)/4 is applied to curb residual
- spurious reversals.
+ Enhanced filter using a forward-only LLT Kalman filter followed by an
+ RTS smoother to obtain a true zero-phase estimate.  The optional
+ ``zero_phase`` flag is retained for API compatibility - when set to
+ ``False`` the function returns the forward-filtered (causal) estimate.

    Args:
        x: Input 1-D array
        window_size: Sliding window size for adaptive noise estimation

- Returns:
-     y: Filtered output of length ``len(x) - window_size + 1``.
-     The function removes the processing delay of ``window_size - 1`` samples
-     after the forward-backward pass, yielding exactly this many points.
    """
    if len(x) < window_size:
        raise ValueError(
            f"Input signal length ({len(x)}) must be >= window_size ({window_size})"
        )

- # Forward pass
- config = KalmanConfig(window_size=window_size)
- fwd = _kalman_llt(x, config)
-
```

```
- if zero_phase:
-     # Backward pass (run on reversed signal)
-     rev = _kalman_llt(x[::-1], config)[::-1]
-     # Zero-phase estimate: average of forward and backward passes
-     combined = (fwd + rev) / 2.0
- else:
-     combined = fwd
-
- # Light post-smoothing to reduce remaining spurious reversals
- combined = _post_smooth(combined)
+ # Forward pass with RTS smoothing
+ config = KalmanConfig(window_size=window_size)
+ smoothed = _kalman_llt_smoother(x, config)
+
+ if not zero_phase:
+     # If zero_phase is disabled, fall back to the causal forward estimate
+     # (the first element of the smoother output corresponds to the forward-only estimate)
+     # Here we simply return the unsmoothed forward pass for consistency.
+     smoothed = _kalman_llt(x, config)
+
+ # Light post-smoothing to reduce remaining spurious reversals
+ smoothed = _post_smooth(smoothed)

    # Trim to match original API (account for processing delay)
    delay = window_size - 1
- y = combined[delay:]
+ y = smoothed[delay:]

    return y


def process_signal(
    input_signal: np.ndarray,
    window_size: int = 20,
    algorithm_type: str = "enhanced",
    zero_phase: bool = True,
) -> np.ndarray:
    """
    Main entry point - selects the desired filtering algorithm.

    Args:
        input_signal: 1-D array of samples
        window_size: Size of the sliding window
        algorithm_type: "basic" (simple moving average) or "enhanced" (LLT Kalman)

    Returns:
        Filtered signal array
    """
    # Ensure an odd window size for symmetric zero-phase filtering
    if window_size % 2 == 0:
        window_size += 1

    if algorithm_type == "enhanced":
        return enhanced_filter_with_trend_preservation(
            input_signal, window_size, zero_phase=zero_phase
        )
    else:
        return adaptive_filter(input_signal, window_size)


# EVOLVE-BLOCK-END


def generate_test_signal(
    length: int = 1000,
    noise_level: float = 0.3,
    seed: int = 42,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Generate synthetic test signal with known characteristics.

    Args:
        length: Length of the signal
        noise_level: Standard deviation of noise to add
        seed: Random seed for reproducibility

    Returns:
        Tuple of (noisy_signal, clean_signal)
```

```
    """
    np.random.seed(seed)
    t = np.linspace(0, 10, length)

    # Create a complex signal with multiple components
    clean_signal = (
        2 * np.sin(2 * np.pi * 0.5 * t)  # Low frequency component
        + 1.5 * np.sin(2 * np.pi * 2 * t)  # Medium frequency component
        + 0.5 * np.sin(2 * np.pi * 5 * t)  # Higher frequency component
        + 0.8 * np.exp(-t / 5) * np.sin(2 * np.pi * 1.5 * t)  # Decaying oscillation
    )

    # Add non-stationary behavior
    trend = 0.1 * t * np.sin(0.2 * t)  # Slowly varying trend
    clean_signal += trend

    # Add random walk component for non-stationarity
    random_walk = np.cumsum(np.random.randn(length) * 0.05)
    clean_signal += random_walk

    # Add noise
    noise = np.random.normal(0, noise_level, length)
    noisy_signal = clean_signal + noise

    return noisy_signal, clean_signal


def run_signal_processing(
    noisy_signal: np.ndarray | None = None,
    signal_length: int = 1000,
    noise_level: float = 0.3,
    window_size: int = 20,
) -> dict:
    """
    Run the signal processing algorithm on a test signal.

    Args:
        noisy_signal: Input signal to filter (if provided, use this; otherwise generate)
        signal_length: Length if generating signal (for backward compatibility)
        noise_level: Noise level if generating signal (for backward compatibility)
        window_size: Window size for processing

    Returns:
        Dictionary containing results and metrics
    """
    # Use provided signal or generate test signal (for backward compatibility)
    if noisy_signal is not None:
        # Filter the provided signal
        filtered_signal = process_signal(noisy_signal, window_size, "enhanced")
        clean_signal = None  # Not available when using provided signal
    else:
        # Generate test signal (for __main__ and backward compatibility)
        noisy_signal, clean_signal = generate_test_signal(signal_length, noise_level)
        filtered_signal = process_signal(noisy_signal, window_size, "enhanced")

    # Calculate basic metrics (only if we have clean_signal from generation)
    if len(filtered_signal) > 0 and clean_signal is not None:
        # Align signals for comparison (account for processing delay)
        delay = window_size - 1
        aligned_clean = clean_signal[delay:]
        aligned_noisy = noisy_signal[delay:]

        # Ensure same length
        min_length = min(len(filtered_signal), len(aligned_clean))
        filtered_signal = filtered_signal[:min_length]
        aligned_clean = aligned_clean[:min_length]
        aligned_noisy = aligned_noisy[:min_length]

        # Calculate correlation with clean signal
        correlation = np.corrcoef(filtered_signal, aligned_clean)[0, 1] if min_length >
     1 else 0

        # Calculate noise reduction
        noise_before = np.var(aligned_noisy - aligned_clean)
        noise_after = np.var(filtered_signal - aligned_clean)
        noise_reduction = (noise_before - noise_after) / noise_before if noise_before >
     0 else 0

        return {
```

```
            "filtered_signal": filtered_signal,
            "clean_signal": aligned_clean,
            "noisy_signal": aligned_noisy,
            "correlation": correlation,
            "noise_reduction": noise_reduction,
            "signal_length": min_length,
        }
    elif len(filtered_signal) > 0:
        # When using provided signal (no clean_signal available), just return filtered signal
        return {
            "filtered_signal": filtered_signal,
            "clean_signal": None,
            "noisy_signal": None,
            "correlation": 0,
            "noise_reduction": 0,
            "signal_length": len(filtered_signal),
        }
    else:
        return {
            "filtered_signal": [],
            "clean_signal": [],
            "noisy_signal": [],
            "correlation": 0,
            "noise_reduction": 0,
            "signal_length": 0,
        }
```

P.2 Full Rewrite at Iteration 156

Breakthrough via Full Rewrites at 156 Iteration

```
# EVOLVE-BLOCK-START
"""
Enhanced real-time adaptive filter for volatile, non-stationary series.

Key ideas:
* Local-Linear-Trend Kalman filter with per-sample adaptive Q/R.
* Exponential-moving-average (EMA) of residual MAD for robust measurement-noise tracking.
* Optional zero-phase RTS smoothing (true zero-lag) and optional Butterworth low-pass.
* Light FIR post-smoother to suppress spurious reversals.
* Savitzky-Golay polishing when SciPy is present; graceful fallback otherwise.
"""

import numpy as np
from dataclasses import dataclass

# Optional SciPy - degrade gracefully if unavailable.
try:
    from scipy.signal import filtfilt, butter, savgol_filter
except Exception:  # pragma: no cover
    filtfilt = None
    butter = None
    savgol_filter = None


def _mad(arr: np.ndarray) -> float:
    """Median absolute deviation - robust scale estimator."""
    med = np.median(arr)
    return np.median(np.abs(arr - med))


def adaptive_filter(x: np.ndarray, window_size: int = 20) -> np.ndarray:
    """Simple moving-average baseline (O(N))."""
    x = np.asarray(x, dtype=float, order="C")
    if x.ndim != 1:
        raise ValueError("adaptive_filter expects a 1-D input array")
    if window_size < 1:
        raise ValueError("window_size must be >= 1")
    if len(x) < window_size:
        return np.empty(0, dtype=float)
    csum = np.cumsum(np.insert(x, 0, 0.0))
    return (csum[window_size:] - csum[:-window_size]) / float(window_size)


```

```python
@dataclass
class KalmanConfig:
    """
    Configuration for the Local-Linear-Trend (LLT) Kalman filter.
    """
    window_size: int = 20          # adaptation window length
    process_factor: float = 0.2    # scaling for process-noise variance
    min_var: float = 1e-6          # floor for any variance
    init_cov: float = 1.0          # initial state covariance
    q_level_init: float = 1e-5     # initial level process-noise
    q_slope_init: float = 1e-6     # initial slope process-noise
    residual_decay: float = 0.9    # EMA decay for residual variance


def _kalman_forward(x: np.ndarray, cfg: KalmanConfig):
    """
    Forward LLT Kalman filter with adaptive Q and R.
    Returns level estimates and histories required for RTS smoothing.
    """
    n = len(x)
    w = cfg.window_size

    # State: [level, slope]
    s = np.array([x[0], 0.0], dtype=float)
    P = np.eye(2, dtype=float) * cfg.init_cov

    A = np.array([[1.0, 1.0],
                  [0.0, 1.0]], dtype=float)
    H = np.array([[1.0, 0.0]], dtype=float)
    I = np.eye(2, dtype=float)

    # Initial measurement variance from first window
    R = np.var(x[:w]) + cfg.min_var
    R_ema = R

    q_level = cfg.q_level_init
    q_slope = cfg.q_slope_init

    # Buffers for adaptive statistics
    residuals = np.zeros(w, dtype=float)

    # Storage for forward pass
    level_filt = np.empty(n, dtype=float)
    s_pred_hist = np.empty((n, 2), dtype=float)
    P_pred_hist = np.empty((n, 2, 2), dtype=float)
    s_filt_hist = np.empty((n, 2), dtype=float)
    P_filt_hist = np.empty((n, 2, 2), dtype=float)

    for i in range(n):
        # Predict
        s_pred = A @ s
        P_pred = A @ P @ A.T + np.diag([q_level, q_slope])

        s_pred_hist[i] = s_pred
        P_pred_hist[i] = P_pred

        # Update
        z = x[i]
        y_res = z - (H @ s_pred)[0]
        S = (H @ P_pred @ H.T)[0, 0] + R_ema
        K = (P_pred @ H.T / S).ravel()

        s = s_pred + K * y_res
        P = (I - np.outer(K, H)) @ P_pred @ (I - np.outer(K, H)).T + np.outer(K, K) * R_ema

        level_filt[i] = s[0]
        s_filt_hist[i] = s
        P_filt_hist[i] = P

        # Adaptive noise update (per-sample)
        residuals[i % w] = y_res
        if i >= w - 1:
            # Measurement variance via EMA of MAD^2
            mad_res = _mad(residuals)
            R_ema = cfg.residual_decay * R_ema + (1 - cfg.residual_decay) * (mad_res ** 2 + cfg.min_var)

            # Process-noise for level from recent level differences
```

```
            recent_levels = level_filt[i - w + 1 : i + 1]
            level_diffs = np.diff(recent_levels)
            q_level = (_mad(level_diffs) ** 2) * cfg.process_factor + cfg.min_var

            # Process-noise for slope from recent slope changes
            recent_slopes = np.gradient(recent_levels)
            slope_diffs = np.diff(recent_slopes)
            q_slope = (_mad(slope_diffs) ** 2) * cfg.process_factor + cfg.min_var
            q_slope = max(q_slope, cfg.min_var)

    return level_filt, s_pred_hist, P_pred_hist, s_filt_hist, P_filt_hist


def _rts_smoother(level_filt, s_pred_hist, P_pred_hist,
                  s_filt_hist, P_filt_hist, A):
    """Rauch-Tung-Striebel smoother - zero-phase level estimate."""
    n = len(level_filt)
    s_smooth = np.empty_like(s_filt_hist)
    s_smooth[-1] = s_filt_hist[-1]

    for i in range(n - 2, -1, -1):
        G = P_filt_hist[i] @ A.T @ np.linalg.inv(P_pred_hist[i + 1])
        s_smooth[i] = s_filt_hist[i] + G @ (s_smooth[i + 1] - s_pred_hist[i + 1])

    return s_smooth[:, 0]


def _light_fir(signal: np.ndarray) -> np.ndarray:
    """
    Very low-latency 5-point FIR (1,4,6,4,1)/16.
    Falls back to 3-point kernel for very short signals.
    """
    if signal.size < 5:
        padded = np.pad(signal, (1, 1), mode="edge")
        kernel = np.array([1.0, 2.0, 1.0], dtype=float) / 4.0
        return np.convolve(padded, kernel, mode="valid")
    padded = np.pad(signal, (2, 2), mode="edge")
    kernel = np.array([1.0, 4.0, 6.0, 4.0, 1.0], dtype=float) / 16.0
    return np.convolve(padded, kernel, mode="valid")


def _sg_smooth(signal: np.ndarray, window_size: int) -> np.ndarray:
    """
    Savitzky-Golay smoothing (quadratic) when SciPy is present,
    otherwise fall back to the light FIR.
    """
    if savgol_filter is None:
        return _light_fir(signal)
    k = max(5, window_size | 1)          # odd, at least 5
    k = min(k, len(signal))
    if k % 2 == 0:
        k += 1
    return savgol_filter(signal, window_length=k, polyorder=2, mode='interp')


def _zero_phase_lowpass(signal: np.ndarray, cutoff: float, fs: float = 1.0) -> np.
    ndarray:
    """
    Zero-phase Butterworth low-pass via filtfilt (if SciPy available).
    """
    if filtfilt is None or butter is None:
        return signal
    try:
        b, a = butter(N=2, Wn=cutoff, btype='low')
        return filtfilt(b, a, signal, method="gust")
    except Exception:
        return signal


def _trim_delay(arr: np.ndarray, window_size: int) -> np.ndarray:
    """Remove deterministic delay introduced by the adaptation window."""
    delay = max(0, window_size - 1)
    if delay == 0:
        return arr
    if delay >= len(arr):
        return np.empty(0, dtype=arr.dtype)
    return arr[delay:]
```

```python
def enhanced_filter_with_trend_preservation(
    x: np.ndarray,
    window_size: int = 20,
    zero_phase: bool = True,
    preserve_length: bool = False,
) -> np.ndarray:
    """
    Adaptive LLT Kalman filter with optional zero-phase RTS smoothing,
    optional Butterworth low-pass, Savitzky-Golay polishing and a final
    lightweight FIR to damp residual reversals.

    Parameters
    ----------
    x : np.ndarray
        Input 1-D signal.
    window_size : int
        Adaptation window length (must be >= 3).
    zero_phase : bool
        If True, apply RTS smoothing for true zero-lag output.
    preserve_length : bool
        If True, prepend NaNs to keep output length identical to input.

    Returns
    -------
    np.ndarray
        Filtered signal (trimmed or length-preserved according to flags).
    """
    x = np.asarray(x, dtype=float, order="C")
    if x.size < window_size:
        raise ValueError(
            f"Input signal length ({x.size}) must be >= window_size ({window_size})"
        )
    if window_size < 3:
        raise ValueError("window_size must be at least 3 for adaptive processing")

    cfg = KalmanConfig(window_size=window_size)
    A = np.array([[1.0, 1.0],
                  [0.0, 1.0]], dtype=float)

    # Forward pass
    level_fwd, s_pred_hist, P_pred_hist, s_filt_hist, P_filt_hist = _kalman_forward(x,
     cfg)

    # Zero-phase estimate if requested
    if zero_phase:
        est = _rts_smoother(level_fwd, s_pred_hist, P_pred_hist,
                            s_filt_hist, P_filt_hist, A)
    else:
        est = level_fwd

    # Optional Butterworth low-pass (cutoff tuned to window size)
    if filtfilt is not None and butter is not None:
        cutoff = 0.5 / max(1, window_size / 2)    # heuristic
        est = _zero_phase_lowpass(est, cutoff=cutoff)

    # Savitzky-Golay polishing (or fallback FIR)
    est = _sg_smooth(est, window_size)

    # Light FIR post-processing to damp residual reversals
    est = _light_fir(est)

    # Trim or preserve length
    if preserve_length:
        trimmed = _trim_delay(est, window_size)
        pad = np.full(window_size - 1, np.nan, dtype=float)
        return np.concatenate([pad, trimmed])
    else:
        return _trim_delay(est, window_size)


def process_signal(
    input_signal: np.ndarray,
    window_size: int = 20,
    algorithm_type: str = "enhanced",
    zero_phase: bool = True,
    preserve_length: bool = False,
) -> np.ndarray:
    """
    Dispatch to either the baseline moving-average or the enhanced adaptive filter.
```

```
    """
    if window_size <= 0:
        raise ValueError("window_size must be a positive integer")
    if algorithm_type not in {"enhanced", "basic"}:
        raise ValueError('algorithm_type must be "enhanced" or "basic"')

    input_signal = np.asarray(input_signal, dtype=float, order="C")

    if algorithm_type == "enhanced":
        return enhanced_filter_with_trend_preservation(
            input_signal,
            window_size=window_size,
            zero_phase=zero_phase,
            preserve_length=preserve_length,
        )
    else:
        return adaptive_filter(input_signal, window_size)


# Public API
__all__ = [
    "process_signal",
    "adaptive_filter",
    "enhanced_filter_with_trend_preservation",
    "run_signal_processing",
    "generate_test_signal",
]
# EVOLVE-BLOCK-END


def generate_test_signal(
    length: int = 1000,
    noise_level: float = 0.3,
    seed: int = 42,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Generate synthetic test signal with known characteristics.

    Args:
        length: Length of the signal
        noise_level: Standard deviation of noise to add
        seed: Random seed for reproducibility

    Returns:
        Tuple of (noisy_signal, clean_signal)
    """
    np.random.seed(seed)
    t = np.linspace(0, 10, length)

    # Create a complex signal with multiple components
    clean_signal = (
        2 * np.sin(2 * np.pi * 0.5 * t)  # Low frequency component
        + 1.5 * np.sin(2 * np.pi * 2 * t)  # Medium frequency component
        + 0.5 * np.sin(2 * np.pi * 5 * t)  # Higher frequency component
        + 0.8 * np.exp(-t / 5) * np.sin(2 * np.pi * 1.5 * t)  # Decaying oscillation
    )

    # Add non-stationary behavior
    trend = 0.1 * t * np.sin(0.2 * t)  # Slowly varying trend
    clean_signal += trend

    # Add random walk component for non-stationarity
    random_walk = np.cumsum(np.random.randn(length) * 0.05)
    clean_signal += random_walk

    # Add noise
    noise = np.random.normal(0, noise_level, length)
    noisy_signal = clean_signal + noise

    return noisy_signal, clean_signal


def run_signal_processing(
    noisy_signal: np.ndarray | None = None,
    signal_length: int = 1000,
    noise_level: float = 0.3,
    window_size: int = 20,
) -> dict:
    """
```

```
    Run the signal processing algorithm on a test signal.

    Args:
        noisy_signal: Input signal to filter (if provided, use this; otherwise generate)
        signal_length: Length if generating signal (for backward compatibility)
        noise_level: Noise level if generating signal (for backward compatibility)
        window_size: Window size for processing

    Returns:
        Dictionary containing results and metrics
    """
    # Use provided signal or generate test signal (for backward compatibility)
    if noisy_signal is not None:
        # Filter the provided signal
        filtered_signal = process_signal(noisy_signal, window_size, "enhanced")
        clean_signal = None  # Not available when using provided signal
    else:
        # Generate test signal (for __main__ and backward compatibility)
        noisy_signal, clean_signal = generate_test_signal(signal_length, noise_level)
        filtered_signal = process_signal(noisy_signal, window_size, "enhanced")

    # Calculate basic metrics (only if we have clean_signal from generation)
    if len(filtered_signal) > 0 and clean_signal is not None:
        # Align signals for comparison (account for processing delay)
        delay = window_size - 1
        aligned_clean = clean_signal[delay:]
        aligned_noisy = noisy_signal[delay:]

        # Ensure same length
        min_length = min(len(filtered_signal), len(aligned_clean))
        filtered_signal = filtered_signal[:min_length]
        aligned_clean = aligned_clean[:min_length]
        aligned_noisy = aligned_noisy[:min_length]

        # Calculate correlation with clean signal
        correlation = np.corrcoef(filtered_signal, aligned_clean)[0, 1] if min_length >
 1 else 0

        # Calculate noise reduction
        noise_before = np.var(aligned_noisy - aligned_clean)
        noise_after = np.var(filtered_signal - aligned_clean)
        noise_reduction = (noise_before - noise_after) / noise_before if noise_before >
 0 else 0

        return {
            "filtered_signal": filtered_signal,
            "clean_signal": aligned_clean,
            "noisy_signal": aligned_noisy,
            "correlation": correlation,
            "noise_reduction": noise_reduction,
            "signal_length": min_length,
        }
    elif len(filtered_signal) > 0:
        # When using provided signal (no clean_signal available), just return filtered
 signal
        return {
            "filtered_signal": filtered_signal,
            "clean_signal": None,
            "noisy_signal": None,
            "correlation": 0,
            "noise_reduction": 0,
            "signal_length": len(filtered_signal),
        }
    else:
        return {
            "filtered_signal": [],
            "clean_signal": [],
            "noisy_signal": [],
            "correlation": 0,
            "noise_reduction": 0,
            "signal_length": 0,
        }
```