BEHEVOLVE: BEHAVIOR-GUIDED EVOLUTION OF MULTI-AGENT PROGRAM POLICIES UNDER SPARSE TASK FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) can synthesize executable multi-agent policies, but improving these policies through program evolution requires feedback that distinguishes promising candidates. Under sparse task-completion feedback, unsuccessful programs receive the same fitness, creating zero-fitness plateaus where fitness alone cannot determine which program to expand or how to revise it. We present BehEvolve, an idea–program evolution framework that leverages execution behavior to guide program evolution without requiring hand-designed process rewards. The framework retains parent–child code changes and paired execution traces, from which a feedback model produces behavioral diagnoses for program mutation and relative preferences for parent selection. Behavioral progress further guides Idea-level scheduling when a strategy branch stalls. Across the eight-task VMAS suite and five Overcooked layouts, BehEvolve achieves the strongest average performance, reaching an 81.5% average VMAS success rate and 12.6 soups on average on Overcooked, while outperforming the strongest genera evolution baseline (AdaEvolve) and the strongest Overcooked-specific baseline (KnowPC) by 29.0% and 18.9%, respectively. Further analyses show that the three behavioral feedback pathways work together across program mutation, parent selection, and idea-level scheduling to improve search efficiency and final performance.

1 INTRODUCTION

Large language models (LLMs) can synthesize executable policies from natural-language task descriptions (Liang et al., 2023; Singh et al., 2023). Such policies express decisions through explicit control flow, API calls, and rules. Because these decisions are represented as code, the resulting policies can be automatically executed, evaluated, and revised. However, executability alone does not guarantee task success: a policy may execute without errors yet still fail to complete its task.

LLM-driven program evolution provides an approach to improving such policies. It repeatedly selects a parent program, uses an LLM to generate a child, evaluates the child with a task evaluator to obtain its fitness, and continues the search from promising candidates (Romera-Paredes et al., 2024; Novikov et al., 2025). In this loop, fitness determines which candidates are retained, which programs are expanded, and how the search budget is allocated. The effectiveness of program evolution therefore depends on whether the evaluator can distinguish promising candidates.

This distinction becomes difficult when the task evaluator provides only sparse completion feedback. In multi-agent coordination, the limitation is especially visible because task success depends on agents' joint behavior: an agent may complete a subtask, avoid a collision, or coordinate more effectively without changing the team's final score. Distinct program revisions can receive the same task score, often zero, despite producing different task-relevant behaviors (Andrychowicz et al., 2017; Carroll et al., 2019). A dense process reward could distinguish such candidates, but defining it requires task-specific choices about which intermediate states or events constitute progress and how they should be weighted, adding a manual design burden (Ng et al., 1999).

However, most existing program-evolution methods rely primarily on task evaluator scores, or histories derived from those scores, to retain candidates and allocate the search budget (Novikov et al., 2025; Lange et al., 2025; Cemri et al., 2026). When sparse completion feedback assigns the same

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.

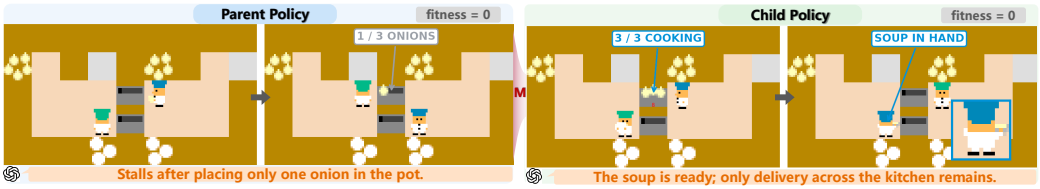


Figure 1: Sparse task scores can conceal behavioral progress. In this Overcooked example, the parent stalls after placing one onion, while its mutated child completes the soup but has not delivered it. Both receive zero task scores, yet their behavior reveals progress missed by completion-only feedback.

task score to multiple candidates, task scores alone provide no basis for deciding which candidate to expand or how to revise it. MLES takes a step beyond score-based feedback by incorporating multimodal behavioral evidence from individual policy executions to diagnose failures and guide local program revision (Hu et al., 2026). However, existing approaches do not systematically use behavioral evidence to guide multiple decisions across the evolutionary search, including candidate selection, program revision, and search allocation.

Our key observation is that *sparse task scores do not imply sparse behavioral information*. Even when a parent and its mutated child receive the same task score, their executions may reveal meaningful differences in task-relevant behavior. Figure 1 illustrates this mismatch: the two policies are indistinguishable under task fitness, yet the child advances farther toward task completion. Such behavioral differences provide relational evidence for comparing candidates and guiding evolutionary search when fitness is tied.

Building on this observation, BehEvolve addresses this gap by using parent–child behavioral comparisons to guide program evolution when fitness is uninformative. We organize the search into high-level strategy *Ideas* and the code-policy lineages that implement them. For each mutation, we retain the code change, matched execution traces, and task outcomes of the parent and child. An LLM analyzes this evidence to produce a behavioral diagnosis and a relative preference: the diagnosis guides *how to modify* the program next, while the preference helps decide *which program to expand* when fitness is tied. At the Idea level, behavioral progress informs whether to continue refining an existing strategy or replace it with a new one. In this way, execution behavior guides evolutionary search without requiring a hand-designed process reward.

We evaluate BehEvolve in two complementary multi-agent settings: continuous-control coordination in VMAS (Bettini et al., 2024; Ji et al., 2025) and discrete, long-horizon coordination in Overcooked-AI (Carroll et al., 2019). We compare it with general-purpose LLM-guided program-evolution methods and domain-specific control-policy methods. Across the eight-task VMAS suite and five Overcooked layouts, BehEvolve achieves the strongest average performance among the compared methods. On VMAS, it surpasses AdaEvolve, the strongest general-purpose program-evolution baseline; on Overcooked, it surpasses KnowPC, the strongest Overcooked-specific baseline. Its largest gains occur on challenging tasks with long zero-fitness plateaus. Further analyses show complementary benefits across program mutation, parent selection, and idea-level scheduling.

Our contributions are summarized as follows:

- A relational behavioral-feedback formulation that uses matched parent–child executions to reveal progress hidden by tied task scores without requiring a hand-designed process reward.
- BehEvolve, an idea–program evolution framework that uses behavioral diagnoses for mutation, behavioral preferences for fitness-tied parent selection, and behavioral progress for Idea-level scheduling.
- An empirical evaluation against general-purpose LLM-guided program-evolution methods and domain-specific control-policy methods, with analyses of plateau escape, parent selection, and the complementary effects of the three feedback pathways.

2 Background and Related Work

LLM-generated executable control policies. LLMs have been used to generate executable policies for both single-agent and multi-agent decision-making settings (Liang et al., 2023; Singh et al., 2023).

Representative applications include task allocation (Kannan et al., 2023), collaborative planning (Mandi et al., 2023), and multi-agent coordination (Ji et al., 2025). These works primarily study how executable policies are synthesized and deployed. BehEvolve focuses on the subsequent evolution of such policies through repeated execution and code revision, particularly when sparse task feedback cannot distinguish behaviorally different candidates.

LLM-guided program evolution. LLM-guided program evolution uses language models as variation operators in a generate–evaluate–select loop (Lehman et al., 2023; Romera-Paredes et al., 2024). AlphaEvolve scales this paradigm through population-based search, while ShinkaEvolve improves candidate exploration through parent sampling and novelty-based rejection (Novikov et al., 2025; Lange et al., 2025). AdaEvolve further adapts exploration and resource allocation using fitness-improvement histories (Cemri et al., 2026). Across these approaches, evaluator scores remain the primary basis for candidate retention and expansion, making it difficult to identify progress when task feedback is sparse. MLES incorporates multimodal behavioral evidence from individual policy executions to diagnose failures and guide local program revision (Hu et al., 2026). BehEvolve extends behavioral evidence to search-level decisions by deriving relational evidence from matched parent–child executions and using it to guide program mutation, selection among fitness-tied candidates, and Idea-level exploration.

3 METHOD

In this section, we first formulate program evolution under sparse task feedback and introduce the idea–program evolutionary state, in which program transitions retain paired behavioral evidence (Sections 3.1–3.2). We then describe how this evidence guides program mutation and selection among fitness-tied candidates (Section 3.3). Finally, we present behavior-aware idea scheduling (Section 3.4). Figure 2 summarizes BehEvolve.

3.1 PROBLEM FORMULATION

Let $\mathcal{P}$ denote the space of executable multi-agent code policies, where each policy controls a team of agents. Each policy $p \in \mathcal{P}$ has parameters $\theta \in \Theta_p$. Given a task x and a search budget B, let $F_x(p, \theta)$ denote the task-level fitness of policy p with parameters θ. The search objective is $\max_{p\in\mathcal{P},\, \theta\in\Theta_p} F_x(p, \theta)$. Program evolution searches this space by repeatedly selecting a parent, mutating its code, and evaluating the resulting child.

Evaluating candidate i returns $(f_i, \tau_i) = \mathrm{Evaluate}(x, p_i, \theta_i)$, where $f_i = F_x(p_i, \theta_i)$ is its fitness and τ_i is its execution trace. We consider sparse task feedback in multi-agent coordination tasks, where F_x is a terminal task fitness that evaluates task completion without introducing intermediate progress rewards. Agents may complete intermediate subtasks or improve their coordination without changing the task fitness. Consequently, policies with the same fitness may still exhibit substantially different behaviors.

3.2 IDEA–PROGRAM SEARCH HIERARCHY AND BEHAVIORAL FEEDBACK

Idea–Program Evolutionary State. We organize the search as an idea–program hierarchy. The task x is associated with a set of active ideas $\{z_k\}_{k=1}^{K}$, where each idea describes a high-level strategy and owns a program lineage of candidate nodes and parent–child transitions:

$$\mathcal{T} = \left(x, \left\{ z_k \mapsto \left(\mathcal{U}^{(k)}, \mathcal{E}^{(k)} \right) \right\}_{k=1}^{K} \right).$$

Here, $\mathcal{U}^{(k)}$ denotes the set of program nodes associated with idea z_k, and $\mathcal{E}^{(k)}$ denotes the set of directed parent–child evolutionary edges between them. Each node $u_i = (p_i, \theta_i^\star) \in \mathcal{U}^{(k)}$ contains an executable policy and its optimized parameter configuration, where $\theta_i^\star = \mathrm{HPO}(p_i, \Theta_{p_i})$ is obtained by hyperparameter optimization to maximize task fitness. After evaluation, it is associated with its task fitness f_i and execution trace τ_i.

The resulting idea–program hierarchy supports two types of expansion: program expansion extends an existing idea with a new child, whereas idea expansion initializes a new program lineage with a new high-level strategy.

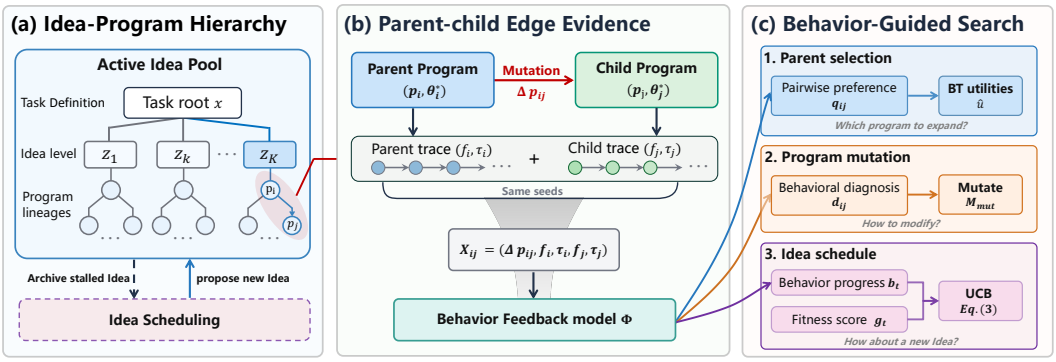


Figure 2: Overview of BehEvolve. (a) The active idea pool organizes candidate programs into separate lineages and exchanges ideas with an external buffer. (b) Each mutation forms a parent–child edge whose code change, matched task scores, and execution traces are processed by the behavior feedback model. (c) The resulting pairwise preference and behavioral diagnosis guide parent selection and program mutation, while fitness gain and behavioral progress jointly inform idea scheduling.

Parent–Child Behavioral Feedback. For each evaluated policy, we represent its execution trace as $\tau_i = (s_{i,t}, a_{i,t}, e_{i,t})_{t=1}^{T_i}$, where $s_{i,t}$ is the multi-agent state, $a_{i,t}$ is the joint action, and $e_{i,t}$ denotes an optional environment event.

When mutation transforms a parent u_i into a child u_j, a behavioral feedback model Φ analyzes their code change and matched evaluation results to identify task-relevant behavioral differences:

$$X_{ij} = (\Delta p_{ij}, f_i, \tau_i, f_j, \tau_j), \qquad (q_{ij}, d_{ij}) = \Phi(x, z_k, X_{ij}). \tag{1}$$

Here, Δp_{ij} denotes the code change from parent to child, q_{ij} represents their relative preference, and d_{ij} captures the behavioral changes from parent to child. We attach this feedback to the evolutionary edge $e_{ij} = (u_i \to u_j, \Delta p_{ij}, q_{ij}, d_{ij})$.

3.3 Behavior-Guided Program Expansion

Building on the parent–child behavioral feedback defined above, we use the relative preference q_{ij} to decide which program to expand and the behavioral diagnosis d_{ij} to guide its modification.

Which program to expand. Each mutation produces a local comparison $u_j \succ u_i$, $u_i \succ u_j$, or an abstention. Because these comparisons are initially confined to parent–child edges, they do not necessarily establish preferences between candidates from different branches. When a child is preferred over its parent, we additionally compare it with the current behavior-best candidate at the same fitness level if they belong to different branches. This selectively connects promising branches without requiring comparisons between every pair of candidates.

Let $\mathcal{W}^{(k)} = \{(u_w, u_l) : u_w \succ u_l\}$ denote the observed preferences under idea z_k. We aggregate them using a regularized Bradley–Terry model:

$$\hat{\mathbf{s}} = \arg\max_{\mathbf{s}} \left[\sum_{(u_w, u_l) \in \mathcal{W}^{(k)}} \log \sigma(s_w - s_l) - \frac{\lambda}{2} \|\mathbf{s}\|_2^2 \right],$$

where s_i is the preference utility of candidate u_i, σ denotes the sigmoid function, and λ controls the regularization strength.

For candidates that are tied under task fitness, we use these utilities to reweight the base parent-selection policy π_0:

$$\pi^\star(u_i \mid z_k) = \frac{\pi_0(u_i \mid z_k) \exp(\beta \hat{s}_i)}{\sum_{u_j \in \mathcal{C}^{(k)}} \pi_0(u_j \mid z_k) \exp(\beta \hat{s}_j)}, \tag{2}$$

where $\mathcal{C}^{(k)}$ is the tied candidate set and β controls the strength of behavioral preference. Candidates without comparison evidence receive zero preference adjustment. When fitness already distinguishes candidates, we use π_0 directly. Behavioral preference therefore guides parent selection when task fitness cannot distinguish candidates.

How to modify the selected program. Once a parent u_i is selected, we retrieve the diagnosis d_i attached to the edge that produced it. The diagnosis summarizes three types of behavioral evidence: behaviors to preserve, regressions to repair, and unresolved failures to address. It serves as mutation guidance rather than as an additional scalar objective. The mutation model takes the task x, the active Idea z_k, the parent code p_i, and the diagnosis d_i as input. It generates an executable revised policy together with its tunable parameter space:

$$(p_j, \Theta_{p_j}) \sim M_{\text{mut}}(x, z_k, p_i, d_i).$$

The generated revision is required to preserve the program interface and action semantics, retain the behaviors identified for preservation, repair the observed regressions, and address the unresolved failures. The resulting code difference is recorded as Δp_{ij}. Invalid or non-executable revisions are rejected and retried. For the initial program of an idea, we set $d_i = \varnothing$. Its first mutation is conditioned only on the task, active idea, and seed program code. After the child is optimized and evaluated, the behavioral feedback model Φ compares the matched parent–child executions and produces the next preference q_{ij} and diagnosis d_{ij} for subsequent search decisions.

3.4 IDEA SCHEDULING AND EXPANSION

At the idea level, the scheduler selects which idea should receive the next program expansion and determines when a stalled idea should be replaced.

Which idea to expand. Building on AdaEvolve's fitness-based bandit allocation across program islands (Cemri et al., 2026), we treat each active idea as a bandit arm and additionally use behavioral progress when fitness cannot distinguish them. After expanding idea z_k with a child u_j, we measure its normalized fitness gain as

$$g_t^{(k)} = \begin{cases} \dfrac{f_j - f^\star_{k,t-1}}{|f^\star_{\text{global},t-1}| + \epsilon}, & f_j > f^\star_{k,t-1}, \\ 0, & \text{otherwise}, \end{cases}$$

where $f^\star_{k,t-1}$ and $f^\star_{\text{global},t-1}$ denote the best fitness within idea z_k and across all active ideas before the expansion. When fitness does not improve, behavioral preference provides an additional notion of progress. After updating the Bradley–Terry model, let $\hat{s}^{(k)}_{\max,t} = \max_{u_i \in \mathcal{U}^{(k)}_{t-1}:\, f_i = f^\star_{k,t-1}} \hat{s}_{i,t}$ denote the highest utility among previous programs at the current best-fitness level. We set $b_t^{(k)} = \mathbb{I}[f_j = f^\star_{k,t-1} \wedge \hat{s}_{j,t} > \hat{s}^{(k)}_{\max,t}]$, which equals 1 when the new child becomes the behavior-best candidate at the idea's current best-fitness level.

For the expanded idea, we update the decayed statistics as $G_t^{(k)} = \rho G_{t-1}^{(k)} + g_t^{(k)}$, $H_t^{(k)} = \rho H_{t-1}^{(k)} + b_t^{(k)}$, and $V_t^{(k)} = \rho V_{t-1}^{(k)} + 1$. We use fitness-based scheduling when ideas have distinct fitness values, and behavioral progress when fitness cannot distinguish them.

The next idea is selected by

$$\bar{r}_k(t) = \begin{cases} G_t^{(k)}/V_t^{(k)}, & \text{fitness-separated}, \\ H_t^{(k)}/V_t^{(k)}, & \text{fitness-tied}, \end{cases} \qquad k^\star = \arg\max_k \left[\bar{r}_k(t) + c\sqrt{\frac{\log(t+1)}{n_k(t)}} \right]. \tag{3}$$

where $n_k(t)$ is the number of times idea z_k has been selected.

When to introduce a new idea. The search begins with K diverse ideas, each initialized with an evaluated seed program. An expansion counts as progress if it either improves the global best fitness or produces a new behavior-best candidate at its idea's current best-fitness level. If no progress occurs for L consecutive expansions, the idea model proposes a new strategy z_{new} using active and archived ideas, representative programs, and unresolved failures as context.

To keep the active pool at size K, we replace the lowest-value eligible idea while preserving the global elite and insufficiently explored ideas. The new idea is initialized with an evaluated seed program and then enters the scheduler. The final program is selected by task fitness over all active and archived lineages. See Appendix A for the full procedure and details.

Table 1: Overall performance on the VMAS and Overcooked multi-agent benchmarks. Panel (a) reports success rates (%) over eight VMAS tasks, and panel (b) reports soups delivered over five Overcooked layouts. Avg. is the arithmetic mean across the reported tasks or layouts. Bold and underlined entries mark the best and second-best means, respectively; ± denotes the sample standard deviation over five independent searches.

(a) VMAS success rate (%)

Method	Aggreg.	Encirc.	Bridging	Coverage	Explor.	Cluster.	Pursuit	Transp.	Avg.
LLM-guided program evolution									
LLaMEA-HPO	72.3±12.7	48.0±11.1	62.0±7.6	0.0±0.0	57.0±15.4	81.0±8.0	6.0±3.2	1.3±1.0	41.0
OpenEvolve	29.7±23.0	23.3±18.1	62.7±14.6	0.0±0.0	16.7±12.9	67.7±24.6	0.0±0.0	0.0±0.0	25.0
GEPA	<u>98.0</u>±0.9	**60.0**±3.9	<u>88.7</u>±0.3	0.0±0.0	32.0±24.8	**100.0**±0.0	0.0±0.0	<u>49.0</u>±19.2	53.5
ShinkaEvolve	0.0±0.0	34.0±13.4	87.7±1.8	0.0±0.0	41.3±19.8	**100.0**±0.0	21.3±16.5	0.0±0.0	35.5
AdaEvolve	94.3±1.4	58.3±4.7	84.3±1.6	35.3±14.3	78.0±6.8	98.0±1.2	30.3±12.7	27.0±11.5	63.2
MLES	96.3±2.1	56.0±11.8	85.0±5.3	22.3±17.4	73.0±9.7	98.0±1.6	36.0±18.5	25.7±16.5	61.5
Domain-specific control-policy methods									
GenSwarm	79.0±13.8	56.0±21.1	74.0±11.8	<u>64.0</u>±24.6	61.0±29.8	69.0±18.5	<u>52.0</u>±26.9	22.0±23.1	59.6
BehEvolve (Ours)	**100.0**±0.0	<u>59.0</u>±2.7	**90.0**±3.6	**67.3**±29.0	**96.3**±2.3	<u>99.3</u>±1.2	**74.3**±20.3	**65.7**±28.0	**81.5**

(b) Overcooked soups delivered

Method	Cramped	Asymmetric	Coord. Ring	Forced Coord.	Counter Circuit	Avg.
LLM-guided program evolution						
LLaMEA-HPO	6.1±2.1	6.4±1.8	4.0±1.6	1.2±0.8	1.5±1.0	3.8
OpenEvolve	10.5±1.1	11.0±1.0	7.7±1.2	4.2±0.9	4.7±1.1	7.6
GEPA	13.1±0.3	14.4±0.4	8.5±1.3	5.1±1.3	<u>8.0</u>±1.2	9.8
ShinkaEvolve	10.2±1.8	11.5±1.4	7.4±1.3	4.0±1.1	3.9±1.6	7.4
AdaEvolve	<u>13.5</u>±0.4	13.7±0.6	<u>9.9</u>±0.7	6.5±1.1	7.2±0.9	10.2
MLES	12.3±0.9	12.8±1.1	9.1±1.2	6.1±1.3	7.2±1.1	9.5
Domain-specific control-policy methods						
KnowPC	9.5±0.2	<u>17.0</u>±0.7	7.3±0.2	**11.2**±0.2	7.6±0.2	<u>10.6</u>
BehEvolve (Ours)	**14.1**±0.5	**17.9**±0.2	**11.8**±0.9	<u>10.7</u>±1.7	**9.4**±1.4	**12.6**

4 EXPERIMENTS

We conduct experiments to address the following research questions:

RQ1: How does BehEvolve compare with existing methods on sparse-feedback multi-agent tasks?

RQ2: Does behavioral feedback accelerate escape from zero-fitness plateaus?

RQ3: Can behavioral preferences guide parent selection toward higher-quality continuations when fitness is tied?

RQ4: How do feedback mechanisms affect performance, search efficiency, and token cost?

4.1 EXPERIMENTAL SETUP

Environments. We evaluate BehEvolve in two complementary sparse-feedback settings. For continuous multi-agent coordination, we use an eight-task VMAS suite (Bettini et al., 2024) based on the multi-robot task settings studied by GenSwarm (Ji et al., 2025), covering formation control, spatial coverage, pursuit, and object manipulation. For discrete, long-horizon coordination, we use the official Overcooked-AI benchmark (Carroll et al., 2019). Complete task definitions, success criteria, and score conversions are provided in Appendix B. Beyond these main benchmarks, we report an exploratory transfer analysis on ARC-AGI-2 in Appendix C.

Backbone LLM and Baselines. For BehEvolve and the LLM-guided program-evolution baselines, we use `gpt-oss-20b` as the backbone LLM. We compare BehEvolve against two groups of baselines. The first comprises six LLM-guided program-evolution methods: LLaMEA-HPO (van Stein et al., 2025), OpenEvolve (Sharma, 2025), GEPA (Agrawal et al., 2025), ShinkaEvolve (Lange et al., 2025), AdaEvolve (Cemri et al., 2026), and MLES (Hu et al., 2026) , which uses multimodal execution feedback to guide the evolution of programmatic control policies. The second comprises domain-specific control-policy methods: GenSwarm (Ji et al., 2025) on our VMAS-based coordination suite, and self-play (SP) (Carroll et al., 2019) and KnowPC (Gu et al., 2024) on Overcooked.

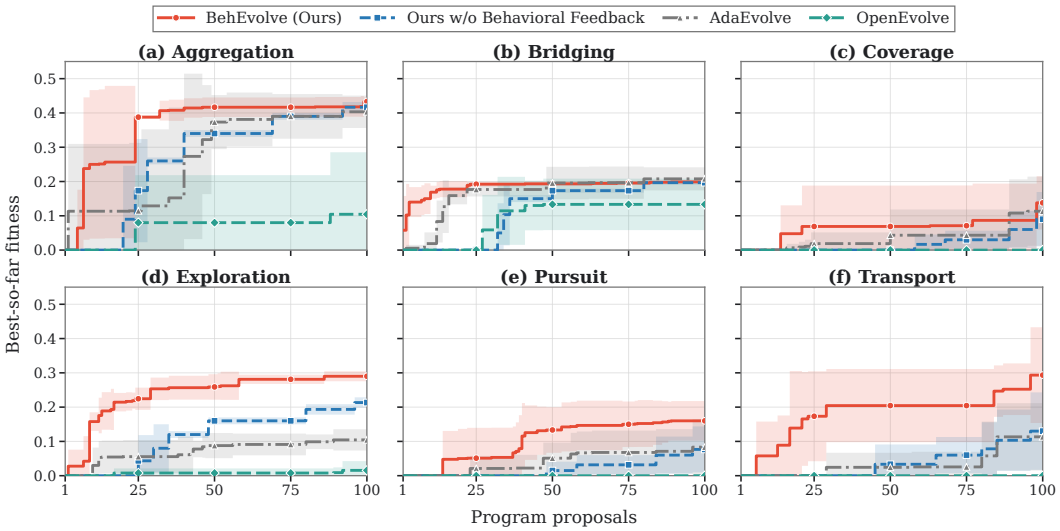


Figure 3: Best-so-far task fitness as a function of program proposals. Curves represent the compared methods, and shaded regions indicate one sample standard deviation across five independent searches.

Evaluation Protocol and Metrics. All methods use the same task evaluator and a matched budget of $B = 100$ program-evolution proposals. For VMAS, an episode receives fitness 0 if the task fails and fitness $1 - 0.01t$ if it succeeds, where t is the completion step and all VMAS tasks use a maximum horizon of 100 steps. The search fitness of each candidate is the average of its episode fitness over the three fixed evaluation seeds. For Overcooked, we retain the official environment and cumulative team reward; for reporting, we divide the reward by 20 to obtain the number of soups delivered. We report the mean and standard deviation over five independent searches. Detailed task definitions and environment configurations are provided in Appendix B.

4.2 Overall Performance (RQ1)

Table 1 compares the best programs discovered by each method on our eight-task VMAS-based suite and five Overcooked layouts.

- **Obs. 1: BehEvolve achieves the strongest overall VMAS performance, with larger gains on challenging coordination tasks.** It attains an 81.5% average success rate, exceeding the strongest general-purpose evolution baseline, AdaEvolve (63.2%), and the domain-specific GenSwarm baseline (59.6%). It obtains the highest mean success rate on six of the eight tasks, with the largest gains on Pursuit, Exploration, and Transport. Near-saturated tasks such as Aggregation and Clustering leave less room for improvement.
- **Obs. 2: BehEvolve also performs strongly on discrete, long-horizon coordination.** On Overcooked, BehEvolve delivers 12.6 soups on average, compared with 10.6 for the strongest competing baseline, KnowPC. It obtains the highest mean score on four of the five layouts, while KnowPC remains stronger on Forced Coordination. These results show that behavior-guided program evolution applies across continuous and discrete action spaces.

4.3 Search Dynamics on Zero-Fitness Plateaus (RQ2)

To examine search behavior when task fitness is uninformative, Figure 3 plots the best fitness found over 100 program proposals on six VMAS tasks. We compare BehEvolve with its behavioral-feedback ablation, AdaEvolve, and OpenEvolve. Curves report the mean over five independent searches, with shaded regions indicating one sample standard deviation.

- **Obs. 3: Behavioral feedback accelerates plateau escape and improves post-escape search.** Compared with its behavioral-feedback ablation, BehEvolve escapes the zero-fitness plateau earlier on all six tasks. It also escapes earlier than the external baselines on five of the six tasks; Aggregation is the exception, where AdaEvolve finds a successful program earlier. After escaping the plateau, however, BehEvolve continues to improve and achieves the highest mean best-so-far

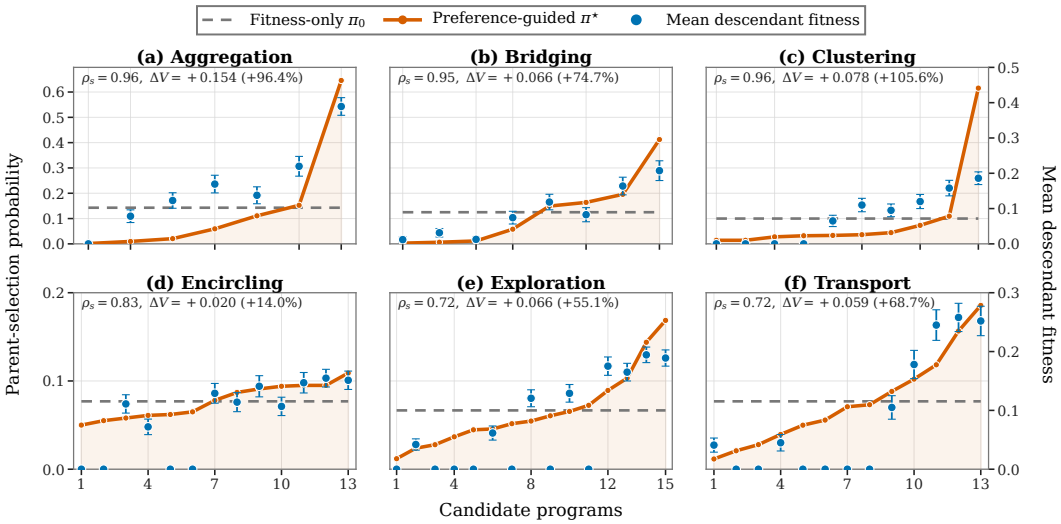


Figure 4: Parent-selection probabilities and descendant fitness for fitness-tied candidates on zero-fitness plateaus across six VMAS tasks. Dashed gray lines denote the uniform fitness-only policy π_0, orange curves denote the preference-guided policy $\pi^\star$, and blue dots show the mean descendant fitness $\bar{Y}_i$. Candidates are ordered by increasing BT utility $\hat{s}_i$. Panel annotations report ΔV and the Spearman correlation ρ_s between $\hat{s}_i$ and $\bar{Y}_i$, with 95% confidence intervals for the reported statistics.

fitness at the end of the budget on five of the six tasks. On Aggregation, AdaEvolve escapes earlier but improves less thereafter, while BehEvolve reaches a higher fitness.

4.4 Preference-Guided Parent Selection (RQ3)

Next, we examine whether behavioral preferences provide a useful parent-selection signal before any candidate obtains nonzero task fitness. We collect candidate pools from this early stage, where every program has zero fitness and the fitness-only policy π_0 is therefore uniform. BehEvolve selects parents using the preference-guided policy $\pi^\star$ defined in Equation 2, which incorporates the BT utilities accumulated during search. To assess whether this policy favors parents that lead to stronger subsequent programs, we continue mutation from each candidate u_i in a separate 20-generation continuation and let $\bar{Y}_i$ denote the mean fitness of the resulting descendant programs. We quantify the expected gain from preference-guided selection over uniform selection as $\Delta V = \sum_i \pi^\star(i)\bar{Y}_i - \sum_i \pi_0(i)\bar{Y}_i$. A positive ΔV means that $\pi^\star$ achieves higher expected descendant fitness than uniform selection. Figure 4 summarizes the parent-selection probabilities, mean descendant fitness $\bar{Y}_i$, and ΔV for six VMAS tasks.

- **Obs. 4: Behavioral preferences correlate with evolutionary potential among fitness-tied candidates.** Across all six tasks, BT utility is positively associated with the mean descendant fitness $\bar{Y}_i$, as measured by the Spearman correlation ρ_s. Because these descendant outcomes are not used to compute the BT utilities, this held-out alignment indicates that behavioral preferences capture differences in future potential that task fitness cannot reveal.
- **Obs. 5: Preference-guided selection improves expected descendant fitness.** Compared with the uniform fitness-only policy π_0, the preference-guided policy $\pi^\star$ yields a positive ΔV across the six tasks. Thus, the preference-guided policy allocates selection probability in a way that increases the expected fitness of subsequent descendants.

4.5 Feedback Mechanism and Cost Analysis (RQ4)

To assess the contribution of behavioral feedback, we compare the full method with an absolute behavioral-feedback variant that produces single-policy diagnoses without parent–child comparison. We also separately ablate diagnosis, preference selection, and idea-level feedback, and evaluate a score-only hierarchical variant that removes all three behavioral signals, with AdaEvolve as an external reference. Table 2 summarizes the comparison.

Table 2: Feedback mechanism comparison and component ablations on the eight-task VMAS suite.

Method	Final success (%) ↑	Escape rate (%) ↑	First success ↓	Mean Tokens (M) ↓
AdaEvolve	63.2	72.2	42.6	1.61
BehEvolve (Ours)	**81.5**	**88.9**	**19.2**	1.74
Feedback mechanism comparison				
Fitness-only	66.1	55.6	52.3	1.49
Absolute behavioral feedback	69.9	63.4	43.3	1.56
Relative behavioral feedback	74.8	68.0	39.5	1.58
Component ablations				
w/o Behavioral Diagnosis	72.8	66.7	40.6	1.63
w/o Preference Selection	76.2	77.8	30.4	1.64
w/o Idea-level Feedback	78.0	83.3	27.8	1.69

- **Obs. 6: Relative behavioral feedback outperforms absolute and fitness-only feedback.** Under the controlled mechanism comparison, relative behavioral feedback achieves the best results across final success, plateau escape, and first-success efficiency. This result suggests that comparing parent and child executions provides a more informative mutation signal than analyzing an individual program in isolation.
- **Obs. 7: Preference selection and idea-level feedback provide further gains at modest cost.** Adding these search-level feedback pathways to relative mutation feedback raises final success from 74.8% to 81.5% and plateau escape rate from 68.0% to 88.9%, while increasing mean token use from 1.58M to 1.74M per search. Thus, the additional feedback pathways improve final performance and plateau escape with a modest increase in average token cost.

To test whether the advantage depends on a larger inference budget, we compare methods with comparable token accounting under matched cumulative token budgets. BehEvolve achieves the highest mean best-so-far VMAS fitness at every checkpoint, from 0.230 at 0.5M to 0.298 at 1.75M tokens, versus 0.229 for AdaEvolve at the final checkpoint (Appendix B.3).

5 Limitations and Future Work

BehEvolve demonstrates that program execution behavior can guide program evolution when task-level feedback is sparse, but it introduces additional trajectory evaluations and LLM feedback calls. The quality of its behavioral diagnoses and preference signals depends on the feedback model and the informativeness of the observed traces. Abstention can reduce the influence of unreliable feedback, but it cannot eliminate the risk of misinterpreting incidental trajectory differences. Our main evaluation focuses on VMAS and Overcooked, with an exploratory ARC-AGI-2 transfer analysis reported in Appendix C; broader validation across application domains and larger-scale program-search tasks remains to be systematically evaluated. Future work could develop adaptive feedback allocation to focus behavioral analysis on informative candidates and reduce unnecessary model queries.

6 Conclusion

Sparse task-completion feedback can leave many candidates tied, especially at zero fitness, even when their executions reveal meaningful behavioral differences. BehEvolve turns these differences into search guidance by retaining parent–child code changes and execution traces, then using them to guide program mutation, parent selection, and Idea-level exploration. Across VMAS and Overcooked, BehEvolve achieves the strongest average performance among the compared methods, improving the average VMAS success rate by 29.0% and the average number of soups delivered in Overcooked by 18.9% over the strongest competing baseline in each suite. Controlled analyses further show that the three feedback pathways complement one another across program mutation, parent selection, and idea-level scheduling. These findings support execution behavior as a useful auxiliary signal for navigating sparse task feedback while leaving the task objective unchanged.

AI USE STATEMENT

In this work, we have not used generative AI tools for tasks with required disclosure, and the remaining required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools to improve the readability of the manuscript and assist with software coding. We have reviewed all AI-assisted work. AI-assisted text and code were reviewed and verified by both authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://arxiv.org/abs/1707.01495.

Matteo Bettini, Ryan Kortvelesy, Jan Blumenkamp, and Amanda Prorok. VMAS: A vectorized multi-agent simulator for collective robot learning. In *Distributed Autonomous Robotic Systems*, Springer Proceedings in Advanced Robotics, pp. 42–56. Springer Nature Switzerland, 2024. doi: 10.1007/978-3-031-51497-5_4.

Micah Carroll, Rohin Shah, Mark K. Ho, Thomas L. Griffiths, Sanjit A. Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human–ai coordination. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

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

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Yin Gu, Qi Liu, Zhi Li, and Kai Zhang. KnowPC: Knowledge-driven programmatic reinforcement learning for zero-shot coordination. *arXiv preprint arXiv:2408.04336*, 2024. URL https://arxiv.org/abs/2408.04336.

Qinglong Hu, Xialiang Tong, Mingxuan Yuan, Fei Liu, Zhichao Lu, and Qingfu Zhang. Multimodal LLM-assisted evolutionary search for programmatic control policies. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=OHFNJoNtjW.

Wenkang Ji, Huaben Chen, Mingyang Chen, Guobin Zhu, Lufeng Xu, Roderich Groß, Rui Zhou, Ming Cao, and Shiyu Zhao. GenSwarm: Scalable multi-robot code-policy generation and deployment via language models. *arXiv preprint arXiv:2503.23875*, 2025.

Shyam Sundar Kannan, Vishnunandan L. N. Venkatesh, and Byung-Cheol Min. SMART-LLM: Smart multi-agent robot task planning using large language models. *arXiv preprint arXiv:2309.10062*, 2023.

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

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pp. 331–366. Springer, 2023.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591.

Zhao Mandi, Shreeya Jain, and Shuran Song. RoCo: Dialectic multi-robot collaboration with large language models. *arXiv preprint arXiv:2307.04738*, 2023.

Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287, 1999.

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

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

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

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530, 2023.

Niki van Stein, Diederick Vermetten, and Thomas Bäck. In-the-loop hyper-parameter optimization for LLM-based automated design of heuristics. *ACM Transactions on Evolutionary Learning and Optimization*, 2025. doi: 10.1145/3731567. URL https://doi.org/10.1145/3731567.

A Search Algorithm and Configuration

A.1 Complete Search Procedure

Algorithm 1 summarizes the complete BehEvolve search procedure.

Here, $\mathcal{Z} = \{z_k\}_{k=1}^{K}$ denotes the active Idea set, $\mathcal{T}$ is the idea–program hierarchy defined in the method section, and $\mathcal{A}$ stores archived Ideas. Each candidate node $u_i = (p_i, \theta_i^\star)$ stores its task fitness f_i and execution trace τ_i; a non-seed node also carries the diagnosis d_i attached to the edge that produced it, with $d_i = \varnothing$ for an initial seed. For each Idea, $\mathcal{W}^{(k)}$ denotes its observed strict preference pairs, and the active Ideas maintain the scheduler statistics used by Eq. 3. Notation follows Section 3. A superscript (k) identifies quantities maintained for Idea z_k; in the schematic, $\hat{s}$, g_t, and b_t abbreviate the corresponding current-Idea quantities $\hat{\mathbf{s}}^{(k)}$, $g_t^{(k)}$, and $b_t^{(k)}$. In the Idea-pool update below, k_t is the Idea expanded at step t, so $b_t = b_t^{(k_t)}$.

Algorithm 1 The BehEvolve search procedure

Require: task x, proposal budget B, active Idea count K, stagnation window L

$\mathcal{Z}, \mathcal{T}, \mathcal{A} \leftarrow$ InitializeIdeas(x, K)
for $t = 1, \ldots, B$ **do**
  // Idea-level exploration
  if Stagnant(L) **then**
    Summarize the current search history from $\mathcal{T}$ and $\mathcal{A}$
    $z_{\text{new}} \leftarrow$ GenerateIdea$(x, \mathcal{Z}, \text{summary})$
    ReplaceIdea(z_{new})
  // Select program to expand
  $z_k \leftarrow$ SelectIdea$(\mathcal{Z})$ ▷ Eq. 3
  Fit Bradley–Terry utilities $\hat{\mathbf{s}}^{(k)}$ from $\mathcal{W}^{(k)}$
  $u_i \leftarrow$ SelectParent$(\mathcal{U}^{(k)}, \hat{\mathbf{s}}^{(k)})$ ▷ Eq. 2
  // Mutate and evaluate
  $(p_j, \Theta_{p_j}) \leftarrow$ Mutate(x, z_k, p_i, d_i)
  $\theta_j^\star \leftarrow$ HPO(p_j, Θ_{p_j})
  $u_j \leftarrow (p_j, \theta_j^\star)$
  $(f_j, \tau_j) \leftarrow$ Evaluate$(x, p_j, \theta_j^\star)$
  Store f_j and τ_j on u_j and record the code change Δp_{ij} from p_i to p_j
  // Behavior feedback and update
  $X_{ij} \leftarrow (\Delta p_{ij}, f_i, \tau_i, f_j, \tau_j)$
  $(q_{ij}, d_{ij}) \leftarrow$ BehaviorFeedback(x, z_k, X_{ij}) ▷ Eq. 1
  Update the hierarchy, preference pairs, and Idea statistics with u_j, q_{ij}, and d_{ij}
return the evaluated candidate with the highest stored task fitness

A.2 Idea-Pool Update

For expansion t, define global progress

$$\Delta_t = \mathbb{I}\left[f^\star_{\text{global},t} > f^\star_{\text{global},t-1} \ \vee \ b_t = 1\right], \tag{4}$$

where k_t is the Idea expanded at step t and $b_t \equiv b_t^{(k_t)}$ indicates that the new program replaced the behavior-best program at that Idea's current fitness level. An Idea update is eligible when $\sum_{\ell=t-L+1}^{t} \Delta_\ell = 0$. The stagnation window restarts after an update.

Let z_{elite} be the Idea with highest historical fitness, breaking fitness ties by the current exploitation value $\bar{r}_k$. The set of replaceable Ideas is

$$\mathcal{R}_t = \{z_k \in \mathcal{Z}_t : n_k(t) \geq n_{\min}, \ z_k \neq z_{\text{elite}}\}. \tag{5}$$

If $\mathcal{R}_t$ is empty, replacement is deferred. Otherwise, the algorithm archives $\arg\min_{z_k \in \mathcal{R}_t} \bar{r}_k(t)$, breaking ties by the oldest last-progress time. The new Idea starts with $G^{(\text{new})} = H^{(\text{new})} = V^{(\text{new})} = n_{\text{new}} = 0$ and receives one forced expansion before ordinary UCB selection. This forced

Table 3: Fixed hyperparameters and runtime configuration.

Hyperparameter	Value
Idea pool and replacement	
Active pool size (K)	5
Stagnation window (L)	5
Minimum expansion count ($n_{\min}$)	3
Behavior aggregation	
Reward-decay factor (ρ)	0.9
UCB exploration coefficient (c)	1.41
Bradley–Terry regularization (λ)	1
Behavior-selection strength (β)	1
Parent sampling temperature	0.5
Search and evaluation	
Maximum HPO trials per program	100
Search evaluation seeds	20–22
Held-out evaluation	100 episodes per selected program
LLM inference	
Decoding temperature	0.7
Maximum generation length	32,000 tokens
Request timeout	600 s
Maximum request retries	3
Retry delay	5 s

expansion is the next proposal and counts toward the same total budget B; it does not create an additional proposal or search budget.

Table 3 summarizes the fixed hyperparameters and runtime configuration.

A.3 Feedback Evidence and Output Schema

The preference and diagnosis calls receive the task completion criterion, current Idea, parent and child identifiers, a structured code diff, matched rollout identifiers, task outcomes, and aligned trace fields as applicable. Their responses are parsed and combined into the edge-level record below. A missing evidence pointer, invalid label, or parse error causes a bounded retry; exhaustion produces `abstain` and the search falls back to its fitness policy.

Table 4: Required fields in an edge-level behavior-feedback record.

Field	Meaning
`preference`	`parent`, `child`, `tie`, or `abstain`.
`evidence`	Trace timestamps/events supporting the preference.
`preserve`	Behavior introduced or retained by the child that should remain.
`regressions`	Observed behavior worse than the matched parent execution.
`unresolved`	Failure modes still visible in the child execution.

The comparison, selective-best, mutation, and Idea-generation prompts instruct the model not to infer unobserved state, not to convert the analysis into a scalar task reward, and to abstain when matched executions do not support a direction. The required fields are listed in Table 4.

A.4 Behavior-Feedback Prompts

The frozen prompts are organized by feedback operation. Each block records the system message, user-message fields, output schema, abstention rule, and retry behavior, following the prompt presentation used in prior program-evolution work.

Parent–child Behavior Preference. We employ a two-stage preference evaluation procedure to compare the task-relevant behaviors of a child program and its direct parent. First, an LLM judge independently evaluates their matched physical-state trajectories for each seed, producing a local preference supported by explicit state references. These local judgments are then contract-validated and aggregated across seeds to obtain an overall expansion preference. By separating local behavioral assessment from cross-seed aggregation, the procedure ensures that the final preference is grounded in verifiable trajectory evidence rather than unsupported global judgments.

Parent–child behavior preference

Stage I: Single-seed behavior comparison.
Input: Task requirements and success conditions, public parent–child scores, and matched raw physical states for a single evaluation seed.
Instruction:
Compare the task-relevant behavior of Parent and Child using only the supplied raw physical states and task requirements. Consider only matched steps within the given seed.
Judge observed behavior rather than code intent, parameter values, or presumed mutation effects. Assign `child` or `parent` only when the observed states demonstrate a clear task-relevant advantage. Assign `tie` when the behaviors are materially indistinguishable, and `abstain` when the evidence is insufficient, mixed, or ambiguous.
Do not assume that greater motion, proximity, dispersion, or symmetry is inherently preferable unless justified by the task requirements. Do not derive or report numerical metrics absent from the supplied physical states.
Each non-abstaining judgment must include one or two evidence references identifying the seed, matched parent–child steps, and observed physical fields. Do not generate mutation advice or global conclusions.
Stage II: Cross-seed preference aggregation.
Input: Task requirements and success conditions, public parent–child scores, and contract-validated single-seed analyses.
Instruction:
Aggregate the local behavior comparisons into a single parent–child expansion preference, treating all seeds equally.
Use only local analyses with valid status. Prefer a direction only when the valid seed-level evidence consistently supports it. Assign `tie` when the aggregate behaviors are materially indistinguishable, and `abstain` when the evidence is insufficient or conflicting.
Support each non-abstaining judgment with one or two evidence references copied directly from the valid local analyses. Do not infer additional physical-state information, generate mutation advice, or repeat the detailed single-seed analyses.
Output format.
Both stages return a JSON object containing `schema`, `preference`, `brief_reason`, and `evidence`.
The preference must be one of `child`, `parent`, `tie`, or `abstain`.
The reason must compare Parent and Child in a single sentence of at most 24 words. Each evidence reference identifies the evaluation seed, matched parent–child steps, and supporting physical fields.
For `abstain`, return an empty evidence list. Do not output additional prose.

Comparative Process Feedback. This prompt compares a child with its direct parent using matched outcomes and trace evidence, then produces structured, evidence-grounded guidance for subsequent evolution.

Parent–child comparative behavior diagnosis

System message.
You are an evaluator responsible for analyzing the current child program C and generating structured feedback $F(C)$ to guide its subsequent evolution.
Analyze the child's execution trajectory, evaluation metrics, and source code. Compare C exclusively with its direct parent P using matched-seed evaluation evidence.
Your feedback should identify the child's successful behaviors and failure patterns, distinguish improvements, regressions, and retained capabilities relative to its parent, and provide evidence-grounded guidance for future evolution.
All conclusions must be supported by observable execution evidence. Do not infer hidden mechanisms, invent evaluation results, or attribute behavioral changes to code modifications without supporting evidence.
User message fields.
The evaluator receives the following information:

- **Task requirements:** The task objectives and success criteria.
- **Child execution trajectory:** Rollout traces and episode outcomes of the current child.
- **Child evaluation metrics:** Per-seed performance measurements and aggregate scores.
- **Direct parent–child comparison:** Matched-seed task outcomes and aligned execution traces for the child and its direct parent.
- **Parent program and code change:** The direct parent's source code or identifier and the structured parent–child code diff.
- **Child program:** The source code of the current child.
- **Cached parent feedback:** Available edge-level diagnosis associated with the direct parent, when present.

Analysis instructions.
First, summarize the current child's program structure and identify its successful and failed evaluation seeds. Group failed seeds by observable failure patterns and assess which task requirements remain unmet.
Second, compare the child exclusively with its direct parent using matched-seed evidence. Identify observable improvements, regressions, and retained capabilities. Each comparison must reference the corresponding matched seeds and be supported by task outcomes, observable trace fields, or both. Equal task scores do not preclude a behaviorally meaningful difference.
Finally, generate guidance for subsequent evolution. Preserve capabilities supported by observed improvements or retained behaviors, and explore unresolved failure patterns and regressions. Each recommendation must reference its supporting analysis or comparison claims.
Prioritize execution trajectories and episode outcomes, followed by matched-seed comparisons, cached parent feedback, and program source code. Avoid unsupported causal claims and comparisons with earlier ancestors or siblings.
Output format.
Return a JSON object containing `program_summary`, `successful_seeds`, `failure_analysis`, `improvements`, `regressions`, `preserve`, `unresolved`, and `next_generation_guidance`.
Each comparison must be grounded in matched-seed task outcomes or trace evidence, and each evolutionary recommendation must reference its supporting evidence. If matched evidence cannot support a comparative field, mark that field as `abstain` and leave its claim list empty. Omit claims that cannot be substantiated by the available evidence.

Controller Mutation. Given the current program, task requirements, active Idea, and behavioral diagnosis, the mutation prompt asks the model to produce an executable structural revision while preserving the supplied interface and action semantics.

Controller mutation

System message.
You are an expert in program synthesis, multi-agent control, coordination, and LLM-guided program search. Improve the current parameterized policy for the supplied task and environment. Use the task objective, success criteria, observation and action interfaces, and environment constraints supplied in the user message. Preserve the required program interface and return valid actions for the specified environment.
Focus on structural improvements to the controller, including control decomposition, feature construction, coordination logic, phase-dependent behavior, interaction terms, and adaptive schedules or mode switches.
Favor diverse controller structures over minor numerical adjustments. The goal is to produce a strong parameterized controller rather than hand-tune a fixed implementation.
User message fields.
The mutation agent receives the following information:

- **Task and environment:** The task objective, success criteria, observation and action interfaces, and relevant environment constraints.
- **Current Idea:** The high-level strategy associated with the selected program lineage.
- **Current program:** The executable source code of the program to be mutated.
- **Behavioral diagnosis:** Behaviors to preserve, observed regressions, and unresolved failure modes from the selected parent's edge feedback.
- **Search context:** Available evaluation feedback, mutation instructions, and parameter search-space requirements, when provided.

Mutation instructions.
Propose structural modifications that improve the controller's task-relevant behavior. Introduce new control features, coordination mechanisms, or adaptive logic when appropriate.
Express benchmark-dependent numerical quantities as tunable parameters through the supplied configuration interface, such as `config.get(...)` when available, with reasonable initial defaults. These defaults serve only as starting points for downstream automatic parameter optimization.
Do not provide manual parameter-tuning advice. Prioritize controller structure and parameterization over numerical fine-tuning.
Preserve the required function interface and action shape. Ensure that the modified controller is deterministic, executable without manual edits, numerically stable, and free of unnecessary dependencies.
Output format.
Return code modifications using the exact SEARCH/REPLACE diff format:

```
<<<<<<< SEARCH
# Original code
=======
# Replacement code
>>>>>>> REPLACE
```

Each SEARCH block must match the original source code exactly, including whitespace and indentation. Prefer minimal, unambiguous replacement blocks.
The modified program must preserve the supplied interface and return valid actions for the task environment. Include concise function docstrings describing the implemented control approach.

Stagnation-triggered Idea Generation. When progress stalls, the idea-generation prompt asks the model to propose one independently implementable algorithmic direction that targets unresolved failure modes without changing the evaluation protocol. The generated Idea is subsequently initialized with an evaluated seed program before entering the Idea scheduler.

Stagnation-triggered idea generation

System message.
You are an expert algorithm researcher. The evolutionary search has stagnated. Generate high-level breakthrough ideas that are fundamentally different from previously attempted approaches. Analyze the current best program, evaluator feedback, and stagnation history to identify unresolved failure modes and structural limitations.
Do not repeat approaches that have already failed or produced no improvement. Examine their potential failure causes before proposing alternative algorithmic strategies.
User message fields.
The idea-generation agent receives the following information:

- **Problem context:** The task description and input/output requirements.
- **Evaluator code:** The evaluation procedure and optimization objectives.
- **Active and archived Ideas:** The current Idea pool, archived Ideas, and their associated representative programs or lineages.
- **Current representative programs and scores:** Representative programs from active lineages and their evaluation results.
- **Stagnation summary:** The recent search history, fitness progress, and behavior-best progress during the stagnation window.
- **Previously tried ideas:** Previously explored algorithmic approaches and their observed outcomes.
- **Evaluator feedback:** Behavioral diagnoses, Idea-level feedback, identified performance limitations, and unresolved failure modes.

Idea-generation instructions.
Generate one independently implementable algorithmic idea that explores a fundamentally different structural direction.
Each idea must identify a concrete mechanism, its target metric or failure mode, an implementation direction, and important constraints or risks.
Prioritize structural diversity and approaches that address unresolved limitations of the current best program. Avoid repeating unsuccessful strategies or proposing minor parameter adjustments. Preserve the task's input/output requirements. Do not modify the evaluator, task definition, random seeds, evaluation budget, or execution environment.
Output format.
Return one JSON object containing the generated Idea.
The Idea must contain the following fields: `idea`, `description`, `what_to_optimize`, `cautions`, and `approach_type`.
The output must contain only the JSON object, without additional explanations or prose.

B Environments and Evaluation

Our custom continuous-control coordination suite is implemented on the VMAS simulator interface and contains aggregation, encircling, bridging, coverage, exploration, clustering, pursuit, and transport. Table 5 summarizes the shared environment and evaluation protocol; task-specific geometry and completion predicates are defined below.

For all VMAS tasks, the episode fitness is

$$f_{\text{ep}} = \begin{cases} 1 - 0.01t, & \text{if the task succeeds at completion step } t, \\ 0, & \text{otherwise,} \end{cases} \qquad T_{\max} = 100.$$

Table 5: Shared environment and evaluation configuration for the VMAS-based coordination suite. Each search returns one best program, which is then reevaluated under the held-out protocol shown here.

Configuration	Setting
Simulator	VMAS vectorized simulator (Bettini et al., 2024)
Domain	Continuous two-dimensional multi-agent coordination
Task suite	Aggregation, encircling, bridging, coverage, exploration, clustering, pursuit, and transport
Search budget	100 program proposals per independent search
Independent searches	5 per method and task
Final evaluation	100 held-out episodes for the best program from each search
Held-out seeds	1000–1099, disjoint from search and HPO seeds
Episode termination	Task completion or the 100-step horizon
Search fitness	Mean episode fitness over three fixed evaluation seeds (20–22)
Reported metrics	Success rate and mean completion steps
Evaluator exceptions	Counted as failed episodes

The search fitness of a candidate is the mean of f_{ep} over the three fixed evaluation seeds. The task-specific criteria below define only the binary success condition and introduce no intermediate progress rewards.

B.1 VMAS Task Definitions and Success Criteria

The first seven tasks use the task definitions and completion metrics of GenSwarm (Ji et al., 2025); we restate them here to make the evaluator self-contained. Transport uses the native VMAS transport scenario (Bettini et al., 2024).

Let N denote the number of agents, $\mathbf{p}_i \in \mathbb{R}^2$ the position of agent i, and $\|\cdot\|_2$ Euclidean distance. Each episode is successful exactly when all criteria listed for its task are satisfied before the horizon. These criteria define the binary success condition used by the VMAS episode fitness above.

Aggregation. Agents must form a single compact group. We measure the largest nearest-neighbor distance,

$$d_{\text{maxmin}} = \max_i \min_{j \neq i} \|\mathbf{p}_i - \mathbf{p}_j\|_2. \tag{6}$$

The episode succeeds when $d_{\text{maxmin}} < 1$.

Encircling. Agents must occupy a ring of desired radius $r_{\text{des}} = 1$ around a moving prey at $\mathbf{p}_{\text{prey}}$. Radial accuracy is measured by

$$d_{\text{ring}} = \frac{1}{N} \sum_{i=1}^{N} \left| \|\mathbf{p}_i - \mathbf{p}_{\text{prey}}\|_2 - r_{\text{des}} \right|. \tag{7}$$

The episode succeeds when $d_{\text{ring}} < 0.1$.

Bridging. Agents must form an evenly spaced vertical bridge along $x = 0$ over $y \in [-2, 2]$. Let $\mathbf{p}_{i,\text{T}}$ be the target bridge positions and $\mathbf{Q}$ a permutation matrix that assigns agents to targets. We use the assignment-invariant Procrustes error

$$d_{\text{proc}} = \min_{\mathbf{Q}} \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{p}_i - \mathbf{Q}\mathbf{p}_{i,\text{T}}\|_2^2. \tag{8}$$

The episode succeeds when $d_{\text{proc}} < 0.1$.

Coverage. Agents must spread over the available region while remaining evenly spaced. The occupied-area ratio is

$$\rho_{\text{area}} = \frac{A_{\text{occupied}}}{A_{\text{total}}}, \qquad A_{\text{occupied}} = (x_{\max} - x_{\min})(y_{\max} - y_{\min}), \tag{9}$$

Table 6: Overcooked-AI environment and evaluation configuration. The environment follows the official benchmark; only the reported episode score is converted from team reward to the number of soups delivered.

Configuration	Setting
Benchmark	Official Overcooked-AI (Carroll et al., 2019)
Players	2 cooperative players with a shared team reward
Layouts	Cramped Room, Asymmetric Advantages, Coordination Ring, Forced Coordination, and Counter Circuit
Official layout IDs	`simple`, `unident_s`, `random1`, `random0`, and `random3`
Action space	Up, down, left, right, no-op, and interact
Episode horizon	400 environment steps
Soup recipe	3 onions, followed by plating and delivery
Cooking duration	20 environment steps
Delivery reward	20 shared reward units per delivered soup
Reported episode score	Cumulative team reward divided by 20, equal to the number of soups delivered
Environment modifications	None

and spacing uniformity is the variance of the nearest-neighbor distances,

$$\mathrm{Var}_{\mathrm{NND}} = \mathrm{Var}\left(\left\{\min_{j\neq i}\|\mathbf{p}_i - \mathbf{p}_j\|_2\right\}_{i=1}^{N}\right). \tag{10}$$

The episode succeeds when $\rho_{\mathrm{area}} > 0.8$ and $\mathrm{Var}_{\mathrm{NND}} < 0.1$.

Exploration. Agents must collectively visit all N_{total} landmarks. A landmark is counted as visited when an agent comes within 0.1 of its center. The visit ratio is

$$\rho_{\mathrm{visit}} = \frac{N_{\mathrm{visit}}}{N_{\mathrm{total}}}. \tag{11}$$

The episode succeeds when $\rho_{\mathrm{visit}} = 1$.

Clustering. Each agent is assigned to a target region according to the quadrant containing its initial position. If $N_{q,\mathrm{achieved}}$ is the number of agents from quadrant q that finish within 0.1 of the corresponding region, then

$$r_{\mathrm{achieve}} = \frac{\sum_{q=1}^{4} N_{q,\mathrm{achieved}}}{N}. \tag{12}$$

The episode succeeds when $r_{\mathrm{achieve}} = 1$.

Pursuit. Agents must follow the moving prey while maintaining a cohesive group. Let $\mathbf{p}_{\mathrm{avg}} = N^{-1}\sum_i \mathbf{p}_i$ be the agent centroid. We measure prey tracking by

$$d_{\mathrm{avg-prey}} = \|\mathbf{p}_{\mathrm{avg}} - \mathbf{p}_{\mathrm{prey}}\|_2 \tag{13}$$

and reuse d_{maxmin} from Eq. 6 for group connectivity. The episode succeeds when $d_{\mathrm{avg-prey}} < 1$ and $d_{\mathrm{maxmin}} < 1$.

Transport. Agents must cooperatively move a package to its circular goal region. Let $\mathcal{P}_t$ denote the package footprint at time t and $\mathcal{G}$ the goal region. Following the native VMAS termination condition, completion is the binary overlap predicate

$$s_{\mathrm{transport}} = \mathbb{I}[\mathcal{P}_t \cap \mathcal{G} \neq \varnothing \text{ for some } t \leq T]. \tag{14}$$

The episode succeeds when $s_{\mathrm{transport}} = 1$; otherwise its episode fitness is zero at the horizon. For successful episodes, the completion-speed fitness is assigned using the VMAS definition above.

B.2 Overcooked-AI Benchmark and Evaluation Score

We use the official two-player Overcooked-AI benchmark (Carroll et al., 2019) without changing its layouts, transition dynamics, action space, recipe, or episode horizon. The two players share a team

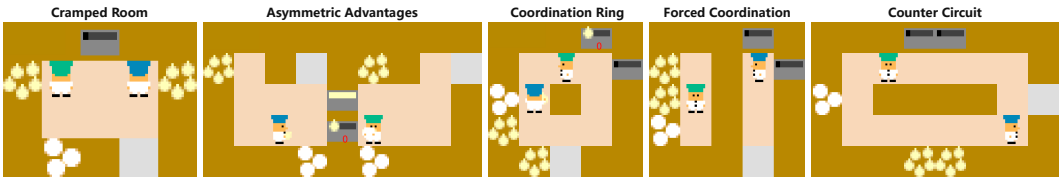


Figure 5: Official Overcooked-AI layouts used in evaluation. From left to right: Cramped Room, Asymmetric Advantages, Coordination Ring, Forced Coordination, and Counter Circuit. The images and layout definitions are unchanged from the official benchmark.

reward and must repeatedly collect onions, cook soup, place the soup in a dish, and deliver it. Table 6 summarizes the environment configuration used for all five layouts.

Let R_T denote the cumulative sparse team reward at the 400-step horizon and N_{serve} the number of soups delivered. Because the official environment assigns 20 reward units to each delivery, we report

$$s_{\text{OC}} = \frac{R_T}{20} = N_{\text{serve}}. \tag{15}$$

This conversion changes only the reporting unit. Overcooked performance is not thresholded into a binary success variable; a larger s_{OC} indicates that the team completed more deliveries within the fixed horizon. The candidate search fitness is obtained by averaging this episode score over the evaluation seeds.

Figure 5 shows the five official layouts used in our evaluation. They expose complementary coordination demands, including movement in a confined shared workspace, asymmetric roles, circulation through narrow paths, mandatory cross-kitchen handoffs, and counter-based item passing.

In both VMAS and Overcooked-AI, trace fields are observations for feedback; none changes the scalar task score.

B.3 Compute-Budget-Matched Evaluation

To separate search efficiency from the number of program proposals, we compare the LLM-guided program-evolution methods under fixed cumulative token budgets per independent search. We include the methods using the shared text-only `gpt-oss-20b` backbone: LLaMEA-HPO, OpenEvolve, GEPA, ShinkaEvolve, AdaEvolve, and BehEvolve. MLES is omitted because it additionally relies on a vision model, which makes its token accounting not directly comparable. At each token cap, Table 7 reports the mean best-so-far VMAS fitness over the eight tasks and five independent repeats. Token accounting includes input and visible output tokens from all recorded LLM calls, including retries.

BehEvolve achieves the highest mean fitness at every token cap, improving from 0.230 at 0.5M tokens to 0.298 at 1.75M tokens. At the final cap, it outperforms AdaEvolve (0.229) and GEPA (0.202), while the other methods remain below 0.18. The consistent lead under matched per-search token budgets indicates that the advantage is not explained solely by a larger language-model inference budget.

Two patterns are notable. First, BehEvolve already achieves the highest mean fitness among the compared methods at the smallest budget of 0.5M tokens, with a score of 0.230. Its performance continues to improve at every checkpoint, reaching 0.298 at 1.75M tokens. In contrast, most baselines largely plateau by 1.5M tokens: LLaMEA-HPO remains unchanged, while OpenEvolve, GEPA, and ShinkaEvolve show no further improvement after that checkpoint; AdaEvolve improves only marginally from 0.227 to 0.229. This pattern indicates that BehEvolve both finds stronger programs early and continues to benefit from additional inference budget rather than saturating early.

C ARC-AGI-2 Transfer Experiment

We evaluate BehEvolve on an additional benchmark family to examine whether behavior-guided program evolution generalizes beyond interactive multi-agent coordination.

Table 7: VMAS best-so-far fitness under matched cumulative token budgets per independent search. Each entry reports the mean best-so-far fitness at the indicated token cap, aggregated over eight VMAS tasks and five independent repeats.

Method	**Fitness @ 0.5M**	**Fitness @ 1.0M**	**Fitness @ 1.5M**	**Fitness @ 1.75M**
LLM-guided program evolution				
LLaMEA-HPO	0.176	0.176	0.176	0.176
OpenEvolve	0.065	0.071	0.078	0.078
GEPA	0.160	0.186	0.202	0.202
ShinkaEvolve	0.098	0.112	0.118	0.118
AdaEvolve	0.140	0.199	0.227	0.229
BehEvolve (Ours)	**0.230**	**0.264**	**0.282**	**0.298**

ARC-AGI-2 Tasks Chollet et al. (2025) evaluate abstract and compositional reasoning across 120 benchmark instances. BehEvolve, OpenEvolve, and AdaEvolve use the same OSS-20B backbone and operate under a matched evolution budget. Each method performs 30 evolution rounds per task on the training examples, followed by evaluation on the corresponding test examples.

ARC-AGI-2 differs substantially from the interactive control environments considered in the main experiments. Instead of multi-agent execution trajectories, each candidate program produces output grids for a set of input examples. For BehEvolve, behavioral evidence is constructed by comparing the outputs of a parent program and its mutated child on the same training examples. The feedback model identifies transformations that the child preserves, corrects, or violates relative to its parent, and uses these differences to guide subsequent mutation and parent selection. The task evaluator remains unchanged and assigns fitness solely according to exact output correctness.

Because ARC-AGI-2 is neither an interactive control benchmark nor explicitly designed for evolutionary search, we treat this experiment as an exploratory transfer analysis rather than a direct extension of the multi-agent evaluation. In particular, it tests whether parent–child behavioral comparisons remain useful when behavior is represented by program input–output relations rather than multi-agent trajectories.

Table 8: Performance on ARC-AGI-2 under a matched budget of 30 evolution rounds per task. All methods use the OSS-20B backbone. Values denote final test accuracy across 120 benchmark instances.

Method	Test accuracy
OpenEvolve	40%
AdaEvolve	45%
BehEvolve (Ours)	52%

Under the matched evolution budget, BehEvolve achieves 52% test accuracy, outperforming OpenEvolve and AdaEvolve by 12 and 7 percentage points, respectively. This result provides preliminary evidence that behavior-guided evolution is not restricted to multi-agent coordination. Parent–child behavioral differences can also guide search in a non-interactive program-reasoning setting, where sparse exact-match feedback leaves behaviorally distinct candidate programs tied.

D Qualitative Case Studies

This section presents successful VMAS rollouts and two logged program lineages. All task definitions and success predicates follow Appendix B.

D.1 Successful VMAS Rollouts

Figure 6 visualizes one successful rollout for each task at five normalized episode times. Agents are shown in red, and their trailing curves record the motion completed up to each snapshot. Task-specific entities are distinguished by color: the prey and its trajectory are blue in encircling and pursuit, obstacles and exploration landmarks are gray, and the package and goal are orange and blue in transport.

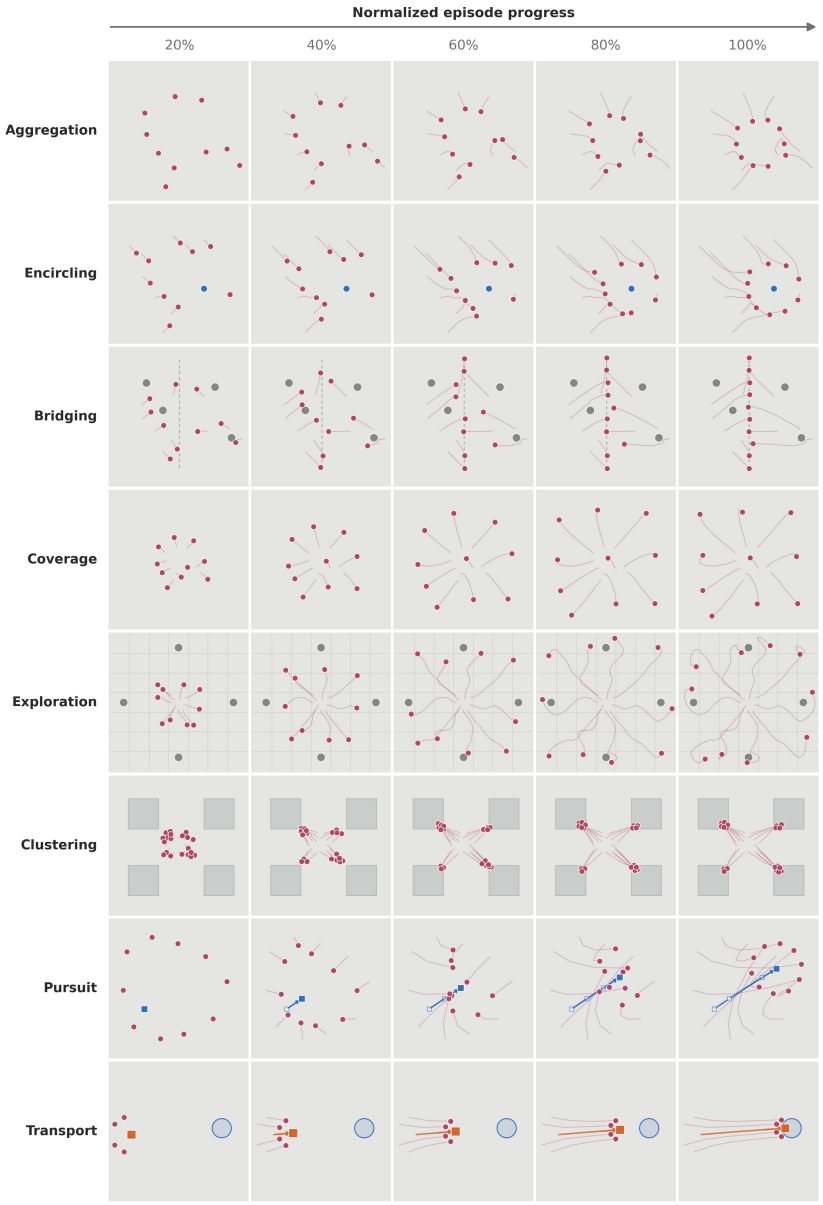


Figure 6: Successful rollouts in the eight-task VMAS-based coordination suite. Columns show states at 20%, 40%, 60%, 80%, and 100% of the episode. Each row uses the same initial condition and trajectory throughout its five snapshots. Task success is determined by the predicates in Section B.1.

D.2 Exploration Lineage

Figure 7 shows a single logged exploration lineage across four generations. The initial program keeps the agents near the center and covers only 22 of the 49 cells. The first child reaches all 49 cells, but finishes slowly; the next child preserves the wide sweep and reduces completion from 90 to 74 steps, and the final child reduces it further to 72 steps. This case illustrates improvement through spatial dispersion and coverage rather than package contact.

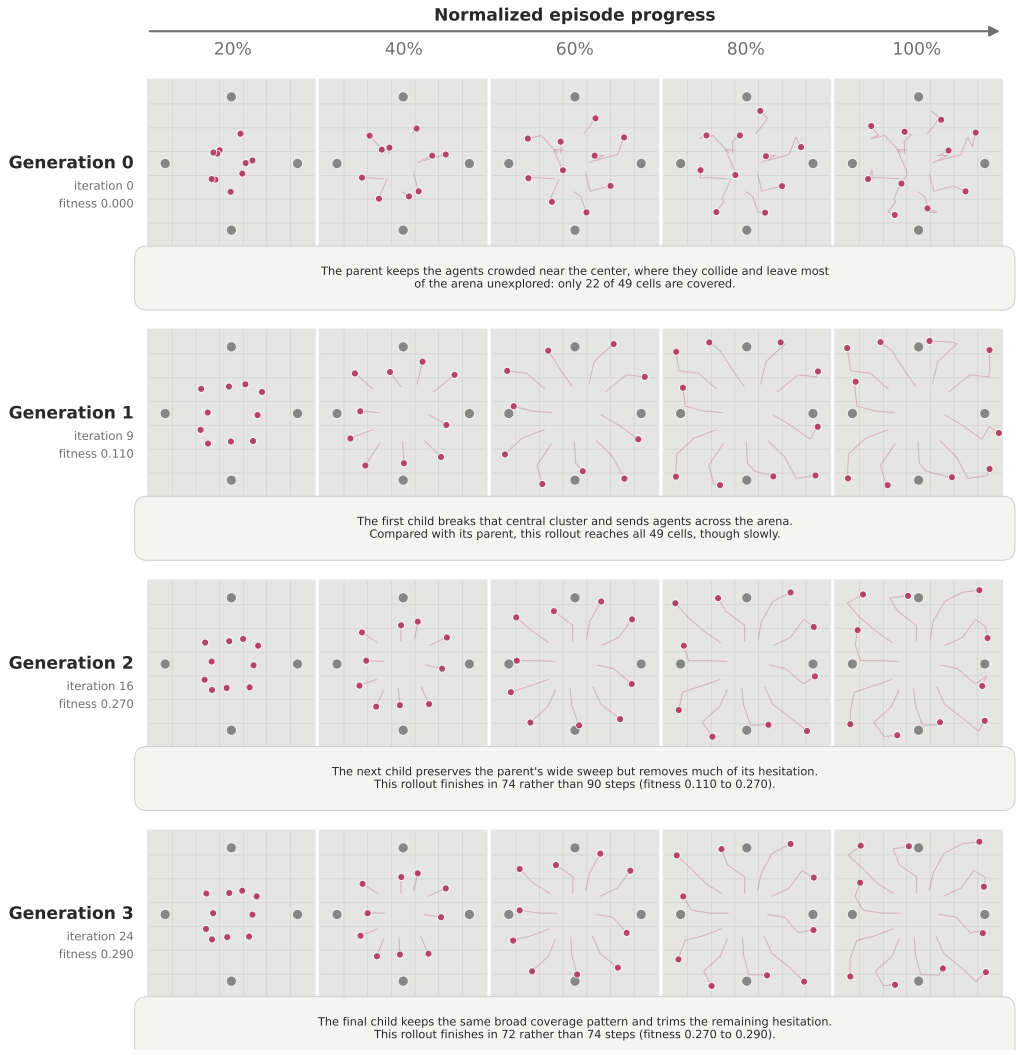


Figure 7: Exploration lineage case study. Each row shows one generation of the logged lineage at 20%, 40%, 60%, 80%, and 100% of the episode. The labels report the episode fitness of the displayed case. The lineage progresses from central collisions and incomplete landmark coverage to a retained wide sweep with faster completion.

D.3 TRANSPORT LINEAGE

Figure 8 presents a single logged transport lineage across four lineage steps. The initial program approaches the package from different sides, loses contact, and leaves it nearly stationary. The first child gives the push a shared direction but still fails to complete the delivery. The next child keeps the agents behind the package and establishes steady contact, while the final child preserves this alignment and reduces completion from 88 to 66 steps.



Figure 8: Transport lineage case study. Each row shows one lineage step of the logged lineage at 20%, 40%, 60%, 80%, and 100% of the episode. The labels report the episode fitness of the displayed case. The lineage progresses from loss of package contact to coordinated pushing, completed delivery, and faster goal completion.