WHERE AUTONOMY BELONGS: A SEPARATION OF POWERS FOR AGENTIC PROGRAM EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-driven program evolution rests on three decisions: govern directs the search from accumulated evidence, propose constructs the next candidate, and judge settles validity and fitness. All three appear in every system, and existing work splits into two lines over who makes each. Controller-centric systems strengthen govern and neglect the autonomy of propose, which caps what evolution can reach. Agent-centric systems let a coding agent take over both govern and propose, at a cognitive and computational cost. Neither line protects the independent authority of judge, so benchmark reliability suffers. These failures share one root: autonomy is misallocated across the three decisions. We address this problem with EvoTrias, a separation-of-powers architecture in which each decision has one owner: Structured Governance for what spans evolution rounds, Open Proposal for construction inside a task, and Protected Judgment behind a private evaluator. These owners exchange typed contracts only, and runtime hooks reinforce the boundaries. This separation also lets EvoTrias run as a plugin that turns an existing coding agent into an evolution system. EvoTrias sets a new state of the art on mathematical, systems, and algorithm-design benchmarks. Three further results show what each owner buys: separating Governance from Proposal cuts token cost by 64%, Open Proposal extends evolution to a 680K-line industrial repository, and Protected Judgment keeps benchmark scores faithful to transfer.

1 INTRODUCTION

LLM-driven program evolution repeatedly constructs, verifies, and retains candidate programs under a finite evaluation budget. A public task contract and an evaluator guide the search, which has produced results in mathematics, algorithm design, and systems optimization (Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025). Its ceiling depends on three decisions. *Govern* uses accumulated evidence to direct the remaining search. *Propose* constructs the next candidate. *Judge* settles whether a candidate is valid and how well it scores. All three appear in every system, and what differs is who is allowed to make each (Figure 1). Existing work forms two lines, controller-centric and agent-centric, which differ in where they place govern and propose.

Controller-centric systems give govern to a controller, the evolutionary algorithm that drives the loop. AdaEvolve, EvoX, and their successors (Cemri et al., 2026; Liu et al., 2026a; Du et al., 2026; Zhang et al., 2026a; Xing et al., 2026) keep asking how to build a better algorithm for assembling each iteration's context. Propose stays a single LLM call inside that context and has little autonomy of its own. It sees only what the controller assembles, so harder problems, such as end-to-end optimization of an industrial code repository, stay out of reach. The judge boundary weakens as well. Evaluator sources are not kept private, so govern and propose can both learn how candidates are scored. A search that sees the evaluator's fixed test cases can fit them instead of the task. The official score then overstates how well its programs transfer to new instances.

Agent-centric systems move the other way. As coding agents grew more capable of acting on their own, this line asked how much autonomy they could be given. CORAL represents the direction (Qu et al., 2026). It replaces the single LLM call with a coding agent that is both more autonomous and more capable, and it hands govern to that agent as well. Subsequent work pushes the delegation further (Mishra, 2026; Yan et al., 2026b; Virk et al., 2026). The agent that writes the candidates now also reads the history, writes the shared memory, and plans the next move. Two costs follow.

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.

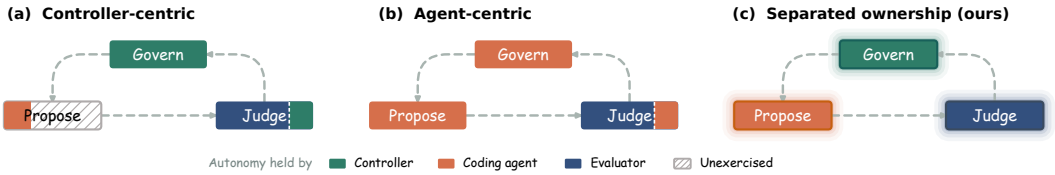


Figure 1: Who holds govern, propose, and judge in an LLM-driven evolutionary loop. **(a)** A controller holds govern, while propose is a single model call with little autonomy of its own. **(b)** The coding agent holds both govern and propose. In **(a)** and **(b)** a dashed seam marks judge: the evaluator still returns the official score, but its sources are not private, so govern and propose erode its autonomy. **(c)** EvoTrias gives each decision one owner, and a runtime hook layer reinforces the split.

The first is cognitive. One agent's lesson from a single attempt goes straight into the memory every other agent reads. No independent owner checks whether it holds, how far it should travel, or when it should expire, so the search narrows before the evidence justifies it. The second cost is computational. Long trajectories spend expensive model compute on re-reading history, rebuilding search state, and planning the next move. Judge fares no better. CORAL grants its agents unrestricted tool access and points them at the evaluator source, which invites the same overfitting to the evaluator.

The two lines fail in mirror-image ways. Controller-centric systems strengthen govern but leave propose too little autonomy for harder tasks. Agent-centric systems give propose that autonomy, yet let it spill into govern, where no owner checks the lessons the agent writes. Neither line keeps judge behind a boundary. These failures share one root: autonomy is misallocated across the three decisions. Each line raises the autonomy of one decision and neglects the other two, along with the boundaries between them. The question is therefore not how much autonomy to grant, but where it belongs. We call this problem *autonomy allocation*: *who should own each of the three decisions, and what holds the boundary once ownership is assigned.*

We answer with **EvoTrias**, a separation of powers that gives each decision its own owner, called a *plane*. **Structured Governance** handles everything that spans evolution rounds. It distills settled outcomes into evidence that drives the selection policy, the durable experience, and the context each new task receives. **Open Proposal** owns open-ended implementation and artifact construction inside an assigned task. A general-purpose coding agent, such as Claude Code or OpenCode, realizes this plane. It pulls the task's context from Governance over MCP, then calls tools and reasons on its own until an artifact exists. The artifact crosses a narrow interface into **Protected Judgment**, which keeps official evaluator evidence private and returns only a normalized outcome: validity, fitness, and metrics. Under this division, each plane decides alone inside its own boundary and reaches the others only through explicit contracts. A runtime hook layer reinforces the split, so no plane can erode the rights of another. These contracts and hooks sit outside the coding agent, so EvoTrias leaves the agent unchanged and runs as a plugin. The user's existing session orchestrates native subagents, and the coding agent itself becomes an evolution system inside the development workflow.

EvoTrias sets a new state of the art across the Math, ADRS, and Frontier-CS benchmarks, ahead of OpenEvolve, EvoX, AdaEvolve, and one- and four-agent CORAL. Three further results show what each owner buys. Open Proposal extends program evolution to industrial repositories: on the CP-SAT local search of OR-Tools, 680K lines of C++, it improves end-to-end solver quality by up to 16.9%. Structured Governance keeps search state out of the coding agent, so EvoTrias reaches stronger solutions under a matched wall-clock budget at about 64% lower token cost than four-agent CORAL. Protected Judgment keeps the benchmark score a more reliable estimate of transfer, as a controlled study of evaluator exposure shows.

2 Related Work

LLM-guided program evolution. LLM-guided program evolution replaces hand-designed mutation operators with semantic transformations selected by executable feedback. ELM introduced learned code variation, and FunSearch, EoH, and ReEvo established the loop around an LLM proposer and an evaluator (Lehman et al., 2022; Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024). AlphaEvolve scales it to multi-component programs, and OpenEvolve provides an open implementation (Novikov et al., 2025; Sharma, 2025). Later work splits into two lines that advance govern and propose.

Controller-centric evolution. One line refines the controller that allocates search effort. AdaEvolve regulates exploration, cross-island resources, and tactic intervention from improvement signals (Cemri et al., 2026). EvoX evolves the executable search strategy during a run, and ShinkaEvolve improves sample efficiency through archive-aware sampling and novelty control (Liu et al., 2026a; Lange et al., 2025). Later systems widen the adaptive surface to context, memory, strategy populations, cost, and the evolution procedure itself (Yan et al., 2026a; Du et al., 2026; Luo et al., 2026; Jiang et al., 2026; Zhang et al., 2026b; Agrawal et al., 2025; Zhang et al., 2025; 2026a; Xing et al., 2026). The effort concentrates on govern, the evolutionary algorithm itself, while the capability of propose goes unaddressed and evaluator sources stay reachable from the controller side.

Agent-centric evolution. Coding agents opened a second line. SWE-agent and OpenHands made repository-level navigation, execution, and repair routine (Yang et al., 2024; Wang et al., 2025). CORAL applies long-running coding agents to parallel evolutionary lineages and coordinates them through scored attempts, Notes, Skills, and heartbeat reflection (Qu et al., 2026). Later systems push the delegation further through evolutionary memory, specialized per-generation agents, and global steering of branch-local trajectories (Mishra, 2026; Yan et al., 2026b; Virk et al., 2026; Li et al., 2026). One agent owns govern and propose together here, and its open tool access erodes the independent authority of judge.

EvoTrias. EvoTrias assigns one owner per decision, and this clean division of responsibilities lets it deploy as a plugin on mainstream coding agents (Appendix A.1).

3 Problem Setup

We study agentic program evolution under a finite proposal budget. A task $\mathcal{Q} = (R_{\mathcal{Q}}, C_{\mathcal{Q}}, J_{\mathcal{Q}}, B_{\mathcal{Q}})$ contains an initial repository, a public contract, an official evaluator, and a hard budget $B_{\mathcal{Q}}$ on proposal slots. Each slot returns at most one valid artifact x_i; evaluating that artifact yields

$$y_i = J_{\mathcal{Q}}(x_i) = (v_i, f_i, \mathbf{m}_i), \qquad x^\star = \underset{\substack{x_i:\, v_i=1 \\ i \in \mathcal{J}_{B_{\mathcal{Q}}}}}{\arg\max} f_i, \tag{1}$$

where v_i, f_i, and $\mathbf{m}_i$ denote validity, scalar fitness (larger is better), and aggregate metrics, and $\mathcal{J}_{B_{\mathcal{Q}}}$ indexes the terminal artifacts returned by the admitted proposal slots.

4 Method

4.1 Three Decision Planes

EvoTrias settles where autonomy belongs by giving each of the three decisions its own plane (Figure 2). **Structured Governance**, a Skill and MCP control plane, owns govern: the persistent cross-trajectory state and the context bound to every new task. **Open Proposal**, realized on a general-purpose coding agent, owns propose, the construction of a candidate inside that context. **Protected Judgment**, an isolated benchmark evaluator, owns judge, the official validity and fitness. Each plane decides alone inside its own boundary and holds no standing outside it. The planes communicate only through typed contracts,

$$\Gamma_t = G(\mathcal{H}_t), \qquad x_t = P(R_{\mathcal{Q}}, C_{\mathcal{Q}}, \Gamma_t), \qquad y_t = J_{\mathcal{Q}}(x_t), \tag{2}$$

where Governance G sends a context capsule Γ_t, Proposal P submits a candidate artifact x_t, and Judgment $J_{\mathcal{Q}}$ returns a normalized outcome y_t. Tasks run asynchronously. Here t marks the logical instant at which a new task binds its context, and $\mathcal{H}_t$ collects the outcomes settled by then, so each settled outcome shapes only later bindings. The hard budget terminates the run. These contracts fix the semantics of G and $J_{\mathcal{Q}}$ independently of how P is implemented, so any host that meets them can supply candidate construction under the same allocation of decision rights. Appendix A.9 discusses what this division costs and what it buys.

4.2 Structured Governance

Structured Governance owns govern. For every new task it selects a construction starting point: one parent candidate and a set of inspiration candidates. When the search experience is worth persist-

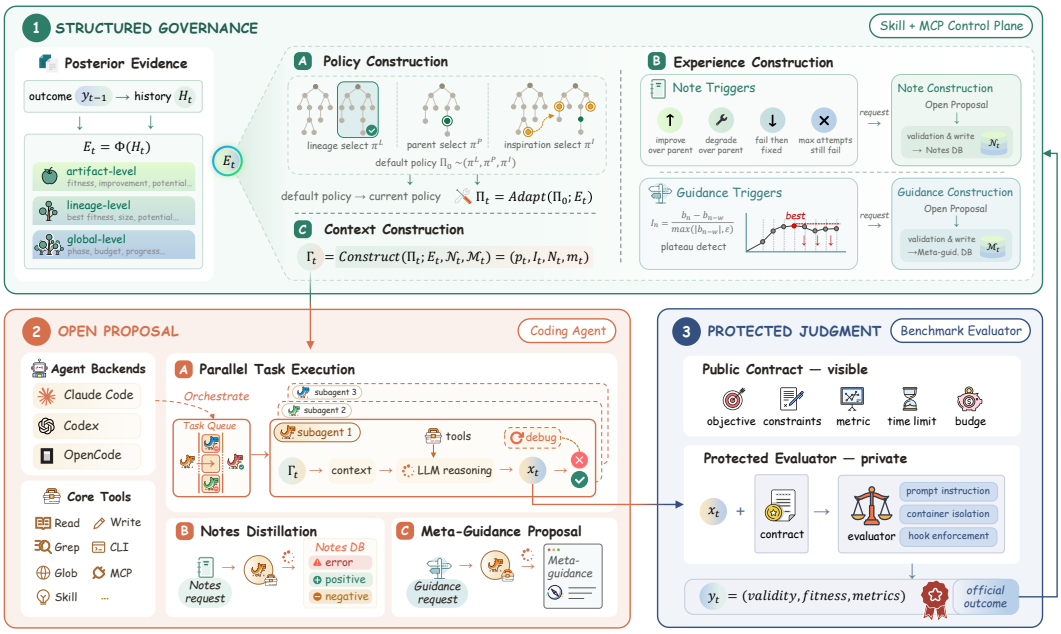


Figure 2: Overview of EvoTrias: Structured Governance (§4.2) binds each task's context capsule, Open Proposal (§4.3) constructs candidate artifacts, and Protected Judgment (§4.4) settles official outcomes that shape later bindings.

ing, it also triggers two kinds of durable experience, Notes and Meta-guidance. These decisions share one source, the posterior evidence distilled from settled outcomes. The evidence feeds three stages: policy construction decides how to select, experience construction decides what to persist, and context construction decides what the new task actually receives.

Posterior evidence distillation. Each official outcome y_{t-1} merges into the settled history $\mathcal{H}_t$, and the evidence operator Φ distills this history into the current evidence

$$E_t = \Phi(\mathcal{H}_t). \tag{3}$$

E_t organizes evidence at three scopes. Artifact-level fields describe a single candidate: verified fitness, improvement over its parent, selection count, and descendant potential. Lineage-level fields aggregate each *lineage*, the descent subtree of a seeded root: best fitness, size, realized share of the successful population, and lineage potential (Appendix A.2.1). Global-level fields track the whole run: search phase, budget progress, and the recent movement of the best fitness.

Policy construction. Governance maintains a default policy Π_0, adapted from population-based program evolution (Sharma, 2025; Cemri et al., 2026): select a lineage, select a parent within it, then sample inspirations around it. These form the three selection rules (π^L, π^P, π^I). Before each new task starts, evidence adjusts the default into the current policy:

$$\Pi_t = \mathrm{Adapt}(\Pi_0; E_t). \tag{4}$$

Adapt makes four kinds of adjustment. *Phase switching*: the root phase assigns each task a distinct initial direction d_i from $\mathcal{D}_0 = \{d_i\}_{i=1}^{n_{\mathrm{dir}}}$, with the parent and inspiration branches closed. Once all n_{dir} roots settle, the run enters the evolution phase and all three distributions take effect. *Progress annealing*: as the budget depletes, selection temperatures and exploration coefficients tighten, moving the search from broad to deep. *Share rebalancing*: when the entropy of lineage shares falls, the source distribution of cross-lineage inspirations tilts toward the higher-quality lineages within a fixed bound. *Guidance anchoring*: a task assigned active Meta-guidance switches its parent branch to the current canonical best, so a new direction starts from the strongest point.

Experience construction. Search experience settles into two kinds of durable content, stored in the notes pool $\mathcal{N}_t$ and the guidance pool $\mathcal{M}_t$. Notes address single attempts. A note request fires

when an attempt's relative fitness delta clears an adaptive threshold, turns a failure into a success, or ends in terminal failure. Governance sends the request to Open Proposal, then validates the returned typed note and commits it to the pool, and later tasks load it by relevance. Meta-guidance addresses the whole search. When the best fitness stalls across a fixed window and patience, Governance gathers the canonical best, the public contract, lineage summaries, admitted notes, and the outcomes of earlier guidance. It then asks Open Proposal for semantically distinct directions. Each committed direction carries a bounded use count, and each use is measured against the canonical best at the moment that use is assigned. Appendices A.4 and A.5 give the full trigger rules and replay both channels on representative runs, and Appendix C.6 audits them run by run.

Context construction. With the policy in place, Governance executes it and constructs the actual context for the new task:

$$\Gamma_t = \mathrm{Construct}(\Pi_t; E_t, \mathcal{N}_t, \mathcal{M}_t) = (p_t, I_t, N_t, m_t). \tag{5}$$

Construction proceeds in order. The parent takes the highest-scoring candidate,

$$p_t = \mathrm{argmax}_{i \in \ell_t} \big(q_i + U_{i,t} + \lambda_P \, g_i\big), \tag{6}$$

where ℓ_t is the lineage drawn from π^L, which favors lineages whose quality-deserved share exceeds their realized share; q_i is rank-normalized quality; $U_{i,t}$ is a UCB bonus that shrinks with selection count; g_i is descendant potential, and λ_P bounds its influence. A task assigned active guidance skips this rule: p_t anchors to the canonical best and m_t takes that guidance; otherwise m_t is empty. Each inspiration entry is drawn independently from a two-level mixture,

$$i \sim (1-\mu)\, \pi_{\mathrm{local}}(\cdot \mid \ell_t, T_t) + \mu \sum_{k \neq \ell_t} \rho_t(k \mid E_t)\, \pi_{\mathrm{elite}}(\cdot \mid k). \tag{7}$$

With probability $1-\mu$ it stays in the task's own lineage ℓ_t, where π_{local} is Boltzmann over quality at temperature T_t. With probability μ it crosses: ρ_t weights each donor lineage k by the elites it can supply and, as lineage shares grow uneven, by its best fitness; π_{elite} then draws uniformly from the top-n_e candidates of k. This tilt in ρ_t is the share rebalancing above. Finally, N_t loads the notes from $\mathcal{N}_t$ most relevant to p_t and I_t. Appendix A.2 gives the full details, and Appendix Table 9 lists the default value of every scheduled constant. The finished capsule crosses to Open Proposal (§4.3).

4.3 Open Proposal

Open Proposal owns propose. Its host is a general-purpose coding agent that supplies file editing, command execution, MCP calls, native subagents, and hook points around tool calls and subagent lifecycle. A workflow skill activates EvoTrias in the user's existing session and makes it the orchestrator, which never touches task content and only spawns native subagents under a fixed scheduling policy. These subagents produce the content Governance requests: candidates, Notes, and Meta-guidance. Every exchange runs over MCP into Structured Governance: the orchestrator asks only whether to continue and how many tasks to mint, and each task executor binds its own capsule.

Candidate proposal. At initialization, a dedicated subagent generates $\mathcal{D}_0$, with one direction guiding the root-artifact construction of each lineage (Appendix A.3). The orchestrator executes the resulting n_{dir} root proposals through a rolling queue with c concurrent slots. Each slot spawns a task-executor subagent that binds its capsule Γ_t and studies the contract and repository. It then conducts the construction trajectory: implement, build, run permitted development checks, read failures, and repair within bounded attempts. Trajectory observations decide tool order, edit locations, and the local plan; no mutation template constrains the candidate. The subagent submits its artifact for official evaluation and terminates. The freed slot is refilled immediately, so settled outcomes from finished tasks reach Governance while sibling tasks still run.

Note and guidance proposal. A note request comes back on the same control-plane call that records the attempt, and the task executor distills that attempt into a typed note before it terminates. An *error* note names the failure and its trigger. A *positive* or *negative* note records which tactic helped or hurt, and under what conditions. A fixed analysis prompt enforces condition-scoped summaries over universal rules, keeping one attempt's experience from hardening into doctrine. A guidance request is answered elsewhere, by a stateless meta actor whose job is to lift the search out of the plateau that triggered the request. It reads the search's own trace and returns a few directions that are semantically distinct from one another and from what the search has already tried.

Runtime hooks. The hook layer spans all three planes, but its rules bear almost entirely on Open Proposal. A coding agent's openness is what the design wants, and the same openness puts two things at risk. The agent may drift from the evolution workflow until the run's state no longer adds up. It may also reach past the information the design granted it. Scheduler hooks address the first risk and permission hooks the second.

Scheduler hooks maintain the run's task pool. The hook is its sole writer, and every agent is a reader. It records each state change from a host event only the runtime can emit: a spawn returning a handle, a subagent starting or stopping. The orchestrator polls the pool to decide what has finished and how many tasks to mint. Scheduling therefore follows what happened, not what an agent reports.

Permission hooks run before every tool call. One rule family denies file reads of evaluator sources and limits the shell to the evaluator command, which keeps judge with Protected Judgment alone. A second denies reads of another task's results. A task therefore works from exactly the capsule Governance bound to it, which keeps govern with Structured Governance alone. A thin adapter maps each host's native events onto this layer, which Appendix A.7 details.

4.4 Protected Judgment

None of the compared systems protects the independent authority of judge. Each keeps a separate evaluator with non-private sources, so govern and propose can erode judge; AdaEvolve, EvoX, and CORAL direct their components to read them (Appendix A.8). Protected Judgment closes this gap through a contract-visible, evidence-private split. The public contract exposes the objective, constraints, metric semantics, time limits, and budget. The evaluator stays private. The artifact x_t enters through a narrow interface, and the response returns only the normalized outcome y_t.

Three measures keep evaluator sources out of reach. The first is instruction: every proposal prompt states the prohibition (Appendix D). The second is isolation: where the benchmark allows, an isolated service holds the evaluator behind a request interface. The third is enforcement: the permission hooks of §4.3 deny evaluator reads at the tool level. The settled outcome then merges into $\mathcal{H}_{t+1}$ and shapes only contexts bound afterwards, closing the search loop.

5 Experiments

5.1 Experimental Setup

We evaluate on three benchmarks: Math (Novikov et al., 2025), ADRS (Cheng et al., 2025), and Frontier-CS (Mang et al., 2025). We compare against OpenEvolve (Sharma, 2025), EvoX (Liu et al., 2026a), AdaEvolve (Cemri et al., 2026), and CORAL (Qu et al., 2026) in one- and four-agent configurations. On Math and ADRS we run two backbones, GLM-5.2 and Opus-5, with three runs per problem, and on Frontier-CS GLM-5.2 once on the final 30 problems (compute constraints preclude repeats). All iteration-based methods receive 50 proposal slots and the same number of evaluator calls (Table 7); CORAL is instead matched by wall clock, with CORAL-4 reporting the best of four complete top-level agents against EvoTrias's one (Appendix B.2). Main EvoTrias experiments use $n_{\mathrm{dir}} = 4$ initial directions and $c = 4$ concurrent task-executor slots; §6.4 analyzes c. EvoTrias and CORAL run on Claude Code with web search disabled. Appendix B gives details.

5.2 Main Results

Table 1 reports the Math and ADRS results. Under each of the two backbones, EvoTrias ranks first on nine of the twelve tasks. The largest margins arise where the search still has headroom: under GLM-5.2, EvoTrias leads the strongest baseline by 6.3% on Heilbronn Convex and by 3.3% on Signal Processing. The five ties under Opus-5 all fall on saturated tasks, where the leading methods return the same score to the last reported digit. Appendix C.1 adds best-run and standard-deviation results and a paired task-level analysis (Table 11): EvoTrias improves on OpenEvolve, EvoX, AdaEvolve, and CORAL-1 on all 12 tasks, with every confidence interval positive and Holm-corrected $p = 0.0024$, and leads CORAL-4 on 9 of 12.

Figure 3 reports the Frontier-CS score distributions. EvoTrias obtains the best mean and, by a wide margin, the best median: 80.67 against 67.14 for CORAL-4. Figure 4 shows the same outcomes as

Table 1: Mean results on Math and ADRS. Best results are highlighted as first and second.

	GLM-5.2						Opus-5					
Task	OpenE.	EvoX	AdaE.	CORAL-1	CORAL-4	EvoTrias	OpenE.	EvoX	AdaE.	CORAL-1	CORAL-4	EvoTrias
Math												
Circle-Pack. (Sq.)	2.54317	2.60956	2.63152	2.62337	2.63418	2.63484	2.63598	2.63598	2.63598	2.63598	2.63598	2.63599
Circle-Pack. (Rect.)	2.35930	2.34876	2.36487	2.32097	2.36459	2.36493	2.36583	2.36583	2.36584	2.36583	2.36583	2.36584
Heilbronn-Tri.	0.0253	0.0302	0.0318	0.0320	0.0332	0.0335	0.0357	0.0361	0.0364	0.0364	0.0365	0.0365
Heilbronn-Conv.	0.0250	0.0269	0.0275	0.0284	0.0283	0.0302	0.0290	0.0307	0.0304	0.0309	0.0309	0.0309
MinMax-Dist.	0.23937	0.23970	0.23987	0.23975	0.24003	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005
Signal-Proc.	0.6373	0.7098	0.6961	0.6638	0.7083	0.7329	0.6930	0.7861	0.7976	0.8251	0.8280	0.8313
Third-Autocorr. ↓	1.4683	1.4601	1.4614	1.4814	1.4608	1.4609	1.4832	1.4587	1.4568	1.4572	1.4559	1.4563
ADRS												
Cloudcast ↓	664.114	646.304	631.130	622.570	618.0869	618.0671	618.069	618.0603	618.071	618.066	618.068	618.0648
EPLB	0.1278	0.1328	0.143327	0.1423	0.1437	0.143332	0.1436	0.1437	0.1439	0.1445	0.1450	0.1448
LLM-SQL	0.6942	0.6794	0.7148	0.6742	0.6890	0.7206	0.7162	0.7229	0.7232	0.7240	0.7318	0.7398
Prism	26.213	26.246	25.359	26.252	26.248	26.256	26.256	26.256	26.256	26.256	26.256	26.256
TXN	3,700	3,843	3,928	4,078	4,280	4,203	3,920	4,281	4,292	4,498	4,532	4,573

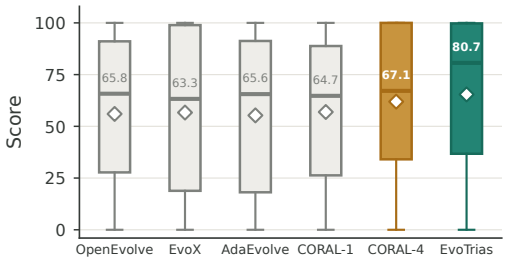


Figure 3: Frontier-CS score distributions.

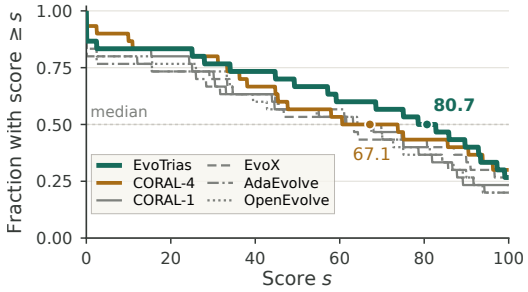


Figure 4: Frontier-CS survival curves.

survival curves, where each curve reports the fraction of problems with score at least s; from the middle of the score range to the mid-90s, EvoTrias's curve lies above every baseline. Appendix C.3 lists per-problem scores and a paired problem-level analysis that confirms this lead: EvoTrias's mean gain is positive against every baseline, with positive confidence intervals against four of the five.

5.3 Ablation Study

Table 2 lists Structured Governance and host/model portability variants; Π_0 is the default policy from Section 4.2, and each cell is mean fitness over three runs.

Table 2: Ablation study results.

Configuration	Circle Packing	Signal Processing
Full EvoTrias	**2.6348** $\pm$0.0019	**0.7329** $\pm$0.0058
Level 1: Governance and initialization		
w/o Notes	2.6122 $\pm$0.0264	0.7193 $\pm$0.0281
w/o Meta-guidance	2.6087 $\pm$0.0102	0.7041 $\pm$0.0094
w/o adaptation (Π_0)	2.6267 $\pm$0.0079	0.6926 $\pm$0.0132
Initial directions $n_{\text{dir}} = 1$	2.5969 $\pm$0.0448	0.6763 $\pm$0.0554
Initial directions $n_{\text{dir}} = 2$	2.6310 $\pm$0.0067	0.7167 $\pm$0.0190
Level 2: Host and model portability		
ClaudeCode + DS-V4-Flash	2.6241 $\pm$0.0126	0.7104 $\pm$0.0117
OpenCode + GLM-5.2	2.6325 $\pm$0.0016	0.7383 $\pm$0.0240
OpenCode + DS-V4-Flash	2.6240 $\pm$0.0126	0.7094 $\pm$0.0208
OpenCode + Opus-5	2.6360 $\pm$0.0000	0.8145 $\pm$0.0153

The Governance ablations stay close to the baseline on Circle Packing but separate more clearly on Signal Processing, where a single initial direction costs 7.7%, the default policy Π_0 5.5%, and removing Meta-guidance 3.9%. This matches their task scope: Signal Processing optimizes a filtering algorithm over a family of signals, whereas Circle Packing targets one fixed 26-circle arrangement, so the broader task gains more from Meta-guidance, adaptation, and initial search breadth. Elevated standard deviations after removing Notes or setting $n_{\text{dir}} = 1$ show that Notes and directional diversity stabilize evolution; a single direction anchors search to the lineage root, amplifying root-dependent variation. Appendix C.5 adds an agentic proposer to the controller-centric baselines; none of the resulting configurations reaches EvoTrias.

The host and model ablations separate two sources of variation. With GLM-5.2 fixed, Claude Code and OpenCode yield mean scores within 0.8% of each other on both tasks; changing the LLM under OpenCode produces a wider spread, and Opus-5 lifts Signal Processing by 10%. EvoTrias is thus less sensitive to the host than to the model: the Proposal host can be replaced without a pronounced loss (Appendix A.1), and construction quality still benefits from stronger LLMs.

6 Analysis

6.1 Repository-Scale Proposal

Beyond the scores of §5.2, Open Proposal holds a second advantage over controller-centric systems: the kind of task it can take on. The benchmarks of §5.1 hand the proposer one file behind a fixed interface, so the only open question is what to write inside it. In contrast, this section evolves the CP-SAT local search of OR-Tools 9.15 (Perron & Furnon, 2026), a C++ repository of about 680K lines, scored by end-to-end solver quality on 300 MiniZinc instances. The repository marks no region to evolve, and solver quality comes from code spread over several functions and files. The proposer must first read the repository, trace how the local search works, and choose the functions to edit. The single LLM call that controller-centric systems use for proposal cannot do this, because a 680K-line repository far exceeds its context window.



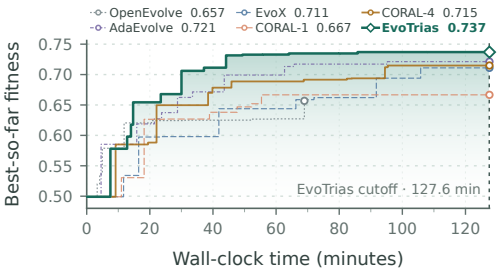
Figure 5: Gain over the unpatched solver; dotted lines mark the one-shot mean.

EvoTrias gives Open Proposal only the objective and a broad edit surface. Open Proposal does the rest: it retrieves and reads the relevant code, selects its change sites, keeps cross-function invariants intact, and exports a Git patch that applies to the untouched source tree. The 300 instances come from the MiniZinc Challenge 2023–2025, and each candidate is scored against the unpatched solver by the Challenge's Borda-count rule; Appendix C.8 details the settings. Figure 5 reports the gain over 50 iterations on all 300 instances and on a 60-instance subset. Both rise steadily, ending at $+13.2\%$ and $+16.9\%$ over the unpatched solver. A single pass of the same agent under the same configuration averages -0.1% and $+1.3\%$ over four runs (dotted lines), so the gain comes from the evolution loop, not from the coding agent alone.

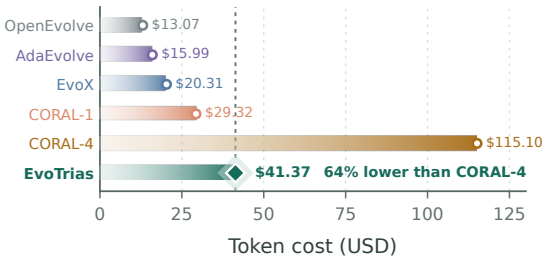
6.2 Runtime and Token Efficiency

Figure 6 measures the computational cost of agent-centric evolution (§1). On Signal Processing, in the best run of each method, EvoTrias holds the best fitness from about 15 minutes onward and ends at 0.737, while CORAL-4 consumes the full matched budget to reach 0.715 (Figure 6a); only OpenEvolve completes its 50 iterations before the cutoff. The token account tells the same story across all seven Math tasks (Figure 6b): a full 50-iteration EvoTrias run costs $41.37 on average, 64% below the $115.10 of CORAL-4.

On time, the advantage of EvoTrias and OpenEvolve comes mainly from parallel task execution; §6.4 isolates this trade-off. On tokens, the three controller-centric systems spend the least, and both agent-centric systems pay for coding-agent trajectories, but EvoTrias keeps the bill bounded. Widening CORAL from one agent to four multiplies its bill by 3.9×, whereas EvoTrias's cost varies by less than 5% from $c = 1$ to $c = 8$. Leading six of seven Math tasks (Table 1) at about a third of CORAL-4's token cost supports the allocation claim: Structured Governance moves history interpretation and search routing out of the expensive agent trajectory, so agent compute concentrates on candidate construction.



(a) Best-so-far fitness over wall-clock time.



(b) Mean token cost of a full run.

Figure 6: Runtime and token efficiency with GLM-5.2. (a) Signal Processing, best of three runs, cut off at EvoTrias's 50-iteration runtime. (b) Average over the seven Math tasks, budgets as in §5.1.

6.3 Evaluator Exposure and Overfitting

This section measures what the judge boundary of §4.4 buys. For task-family benchmarks such as Signal Processing, evaluator source reveals both problem-level guidance (the task definition and scoring rule) and instance-level information (fixed test cases). The former can improve algorithms; we test whether the latter encourages overfitting and distorts performance on unseen instances. We compare exposed and protected settings for CORAL-1, CORAL-4, and EvoTrias, with three runs per configuration. The exposed setting removes Protected Judgment, so this is an ablation of the third plane. Programs are evaluated on Original, IID held-out (new instances of the same families), and OOD held-out (new signal and noise distributions). Appendix C.10 details the protocol.

Table 3: Evaluator exposure on Signal Processing (Gap: OOD − Original).

Method	Evaluator	Score ↑			Gap ↑
		Original	IID	OOD	
CORAL-1	Exposed	0.793	0.750	0.705	−0.089
	Protected	0.664	0.646	0.639	−0.025
CORAL-4	Exposed	0.789	0.756	0.711	−0.078
	Protected	0.708	0.716	0.731	0.023
EvoTrias	Exposed	0.803	0.745	0.721	−0.083
	Protected	0.733	0.732	0.741	0.007

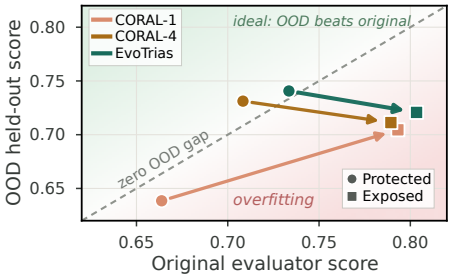


Figure 7: Generalization shift from protected to exposed evaluators.

Table 3 shows that exposure raises the Original score by 0.093 on average, but this advantage contracts to 0.009 OOD. CORAL-1 retains a 0.066 OOD gain, whereas CORAL-4 and EvoTrias each incur a 0.020 loss; their protected programs also score higher OOD than on Original. Figure 7 summarizes how each method's Original and OOD performance changes as evaluator access shifts from protected to exposed. CORAL-4 and EvoTrias cross the zero-gap diagonal. Thus, exposure can aid task and score understanding, but visible cases can inflate benchmark gains and reduce OOD performance. Protecting evaluator code preserves the Original score as a more reliable estimate of transfer. On Circle Packing, whose evaluator scores the one specified arrangement, exposure shifts the score by less than 1%. Appendix C.10.3 provides code-level examples.

6.4 Concurrency Trade-off

Our main experiments use $c = 4$ concurrent task-executor slots (§5.1). Here we examine how c affects wall-clock time and final fitness. Parallel execution reduces wall-clock time, but tasks bind their contexts before sibling tasks finish (§4.3), so recent outcomes remain unavailable as c grows.

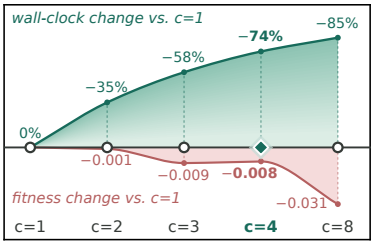


Figure 8: Concurrency trade-off on Signal Processing.

To characterize this trade-off, we vary $c \in \{1, 2, 3, 4, 8\}$ on Signal Processing. Each setting is run three times, and Figure 8 reports the mean wall-clock and fitness changes from $c = 1$. Wall-clock savings show diminishing returns: $c = 4$ already achieves a 74% reduction, while doubling to $c = 8$ adds only 11 percentage points. Fitness loss remains below 0.009 through $c = 4$ but rises to 0.031 at $c = 8$. Together, these results place the knee at $c = 4$. Appendix A.6 models the evidence lag behind this loss.

7 Conclusion

We introduced EvoTrias, a separation-of-powers architecture for LLM-driven program evolution. It answers autonomy allocation by giving govern, propose, and judge one owner each. Typed contracts connect the three planes, and a runtime hook layer reinforces the split. Across the Math, ADRS, and Frontier-CS benchmarks, EvoTrias sets a new state of the art. It also extends program evolution to a 680K-line industrial repository, cuts token cost by 64% against four-agent CORAL, and keeps the evaluator private so that benchmark gains carry over to unseen instances. The same decision boundaries support deployment as a coding-agent plugin, equipping existing hosts with multi-round optimization under dedicated governance and evaluation.

AI USE STATEMENT

We used AI tools in limited, supporting roles for writing assistance (including grammar correction and language polishing), retrieval and discovery of related work, and implementation of parts of the code. For these uses, the authors retained full responsibility for the research decisions and carefully reviewed and verified all resulting text, code, analyses, and results. The coding agents in EvoTrias and in every baseline use LLMs as their behavioral backbone as part of the experimental systems studied here. The authors take responsibility for the final content and conclusions.

ETHICS STATEMENT

This work does not involve human subjects or personal data. All experiments run on public benchmarks (Math, ADRS, and Frontier-CS) and on the open-source OR-Tools repository, each used under its license, and every LLM backbone is accessed under its provider's terms of use for research.

REPRODUCIBILITY STATEMENT

We provide an anonymized EvoTrias code package as a single ZIP file in the supplementary material. It contains the implementation, setup and run instructions, and the task definitions, initial programs, and evaluator scripts for the Math, ADRS, and Frontier-CS benchmarks used in our experiments. The released workflow uses the default operating settings in Table 9. Appendix B specifies the evaluation protocol, run budgets, and baseline configurations; Appendix A describes the governance rules and runtime hooks; and Appendix D provides the role prompts.

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, Daniel Klein, Matei Zaharia, and Omar Khattab. GEPA: reflective prompt evolution can outperform reinforcement learning. *CoRR*, abs/2507.19457, 2025. doi: 10.48550/ARXIV.2507.19457. URL https://doi.org/10.48550/arXiv.2507.19457.

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. *CoRR*, abs/2602.20133, 2026. doi: 10.48550/ARXIV.2602.20133. URL https://doi.org/10.48550/arXiv.2602.20133.

Audrey Cheng, Shu Liu, Melissa Z. Pan, Zhifei Li, Bowen Wang, Alexander Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya A. Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How AI is upending systems research. *CoRR*, abs/2510.06189, 2025. doi: 10.48550/ARXIV.2510.06189. URL https://doi.org/10.48550/arXiv.2510.06189.

Shangheng Du, Xiangchao Yan, Jinxin Shi, Zongsheng Cao, Shiyang Feng, Zichen Liang, Boyuan Sun, Tianshuo Peng, Yifan Zhou, Xin Li, Jie Zhou, Liang He, Bo Zhang, and Lei Bai. Mlevolve: A self-evolving framework for automated machine learning algorithm discovery. *CoRR*, abs/2606.06473, 2026. doi: 10.48550/ARXIV.2606.06473. URL https://doi.org/10.48550/arXiv.2606.06473.

Jiachen Jiang, Huminhao Zhu, and Zhihui Zhu. Smcevolve: Principled scientific discovery via sequential monte carlo evolution. *CoRR*, abs/2605.15308, 2026. doi: 10.48550/ARXIV.2605.15308. URL https://doi.org/10.48550/arXiv.2605.15308.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *CoRR*, abs/2509.19349, 2025. doi: 10.48550/ARXIV.2509.19349. URL https://doi.org/10.48550/arXiv.2509.19349.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. *CoRR*, abs/2206.08896, 2022. doi: 10.48550/ARXIV.2206.08896. URL https://doi.org/10.48550/arXiv.2206.08896.

Junbo Li, Boyi Liu, Canwen Xu, Yite Wang, Yuxiong He, Zhangyang Wang, Qiang Liu, and Zhewei Yao. The optimizer is the agent: Reasoning-driven search across prompts, programs, and ML workflows. *CoRR*, abs/2608.06714, 2026. doi: 10.48550/ARXIV.2608.06714. URL https://doi.org/10.48550/arXiv.2608.06714.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

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

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Z. Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Matei Zaharia, Alexandros G. Dimakis, and Ion Stoica. Skydiscover: A flexible, adaptive framework for ai-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems, CAIS 2026, San Jose, CA, USA, May 26-29, 2026*, pp. 1223–1227. ACM, 2026b. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

Sichun Luo, Yi Huang, Haochen Luo, Fengyuan Liu, Guanzhi Deng, Lei Li, Qinghua Yao, Zefa Hu, Junlan Feng, and Qi Liu. Seaevo: Advancing algorithm discovery with strategy space evolution. *CoRR*, abs/2604.24372, 2026. doi: 10.48550/ARXIV.2604.24372. URL https://doi.org/10.48550/arXiv.2604.24372.

Qiuyang Mang, Wenhao Chai, Zhifei Li, Huanzhi Mao, Shang Zhou, Alexander Du, Hanchen Li, Shu Liu, Edwin Chen, Yichuan Wang, Xieting Chu, Zerui Cheng, Yuan Xu, Tian Xia, Zirui Wang, Tianneng Shi, Jianzhu Yao, Yilong Zhao, Qizheng Zhang, Charlie Ruan, Zeyu Shen, Kaiyuan Liu, Runyuan He, Dong Xing, Zerui Li, Zirong Zeng, Yige Jiang, Lufeng Cheng, Ziyi Zhao, Youran Sun, Wesley Zheng, Meiyuwang Zhang, Ruyi Ji, Xuechang Tu, Zihan Zheng, Zexing Chen, Kangyang Zhou, Zhaozi Wang, Jingbang Chen, Aleksandra Korolova, Peter Henderson, Pramod Viswanath, Vijay Ganesh, Saining Xie, Zhuang Liu, Dawn Song, Sewon Min, Ion Stoica, Joseph E. Gonzalez, Jingbo Shang, and Alvin Cheung. Frontiercs: Evolving challenges for evolving intelligence. *CoRR*, abs/2512.15699, 2025. doi: 10.48550/ARXIV.2512.15699. URL https://doi.org/10.48550/arXiv.2512.15699.

Suyash Mishra. Prism: An evolutionary memory substrate for multi-agent open-ended discovery. *CoRR*, abs/2604.19795, 2026. doi: 10.48550/ARXIV.2604.19795. URL https://doi.org/10.48550/arXiv.2604.19795.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131. URL https://doi.org/10.48550/arXiv.2506.13131.

Laurent Perron and Vincent Furnon. OR-Tools v9.15. Google, 2026. URL https://developers.google.com/optimization/.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: towards autonomous multi-agent evolution for open-ended discovery. *CoRR*, abs/2604.01658, 2026. doi: 10.48550/ARXIV.2604.01658. URL https://doi.org/10.48550/arXiv.2604.01658.

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. URL https://doi.org/10.1038/s41586-023-06924-6.

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

Yuvraj Virk, Zack Edds, Chunqiu Steven Xia, and Lingming Zhang. Swarmresearch: Orchestrating coding agents for open-ended discovery. *CoRR*, abs/2607.02807, 2026. doi: 10.48550/ARXIV.2607.02807. URL https://doi.org/10.48550/arXiv.2607.02807.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Sixue Xing, Haoyu He, Kerui Wu, Zhuo Yang, Haozheng Luo, Tianfan Fu, and Aarthy Nagarajan. Compute allocation for self-evolving llms: From depth-breadth to multi-armed bandits. *CoRR*, abs/2605.29268, 2026. doi: 10.48550/ARXIV.2605.29268. URL https://doi.org/10.48550/arXiv.2605.29268.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, Ed H. Chi, Fernando Pereira, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. Pacevolve: Enabling progress-aware consistent evolution. *CoRR*, abs/2601.10657, 2026a. doi: 10.48550/ARXIV.2601.10657. URL https://doi.org/10.48550/arXiv.2601.10657.

Yuping Yan, Jirui Han, Fei Ming, Yuanshuai Li, and Yaochu Jin. HMACE: heterogeneous multi-agent collaborative evolution for combinatorial optimization. *CoRR*, abs/2605.07214, 2026b. doi: 10.48550/ARXIV.2605.07214. URL https://doi.org/10.48550/arXiv.2605.07214.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, Bang Liu, Chenglin Wu, and Yuyu Luo. Harnessing agentic evolution. *CoRR*, abs/2605.13821, 2026a. doi: 10.48550/ARXIV.2605.13821. URL https://doi.org/10.48550/arXiv.2605.13821.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *CoRR*, abs/2510.04618, 2025. doi: 10.48550/ARXIV.2510.04618. URL https://doi.org/10.48550/arXiv.2510.04618.

Yansen Zhang, Yilu Liu, Tianyu Liu, Jiamin Chen, Xiaokun Zhang, Kai Xie, Xue Liu, Yiyan Qi, and Chen Ma. Budget-aware llm discovery via cost-calibrated frontier utility. *CoRR*, abs/2607.26828, 2026b. doi: 10.48550/ARXIV.2607.26828. URL https://doi.org/10.48550/arXiv.2607.26828.

APPENDIX CONTENTS

A Method Details **15**
A.1 Integration with Coding-Agent Hosts 15
A.2 Design Details of Structured Governance 16
A.3 Initial Directions 17
A.4 Notes 18
A.5 Meta-guidance 22
A.6 Context Staleness under Concurrency 24
A.7 Runtime Hooks 25
A.8 Judge in Existing Systems 28
A.9 Discussion: What the Boundaries Cost and Buy 29
A.10 Limitations 30
B Experimental Setup Details **30**
B.1 Evaluation Protocol 30
B.2 Budget Alignment Across Baselines 30
B.3 Hyperparameter Configuration 32
B.4 Baseline Configurations 33
C Additional Results and Analysis **33**
C.1 Math and ADRS: Statistical Results 33
C.2 Where Backbone Strength Shows Up 35
C.3 Frontier-CS: Statistical Results 36
C.4 Best-so-Far Optimization Trajectories 37
C.5 Controller-Centric Baselines with an Agentic Proposer 38
C.6 Cross-Run Prevalence of the Two Experience Channels 39
C.7 Mechanistic Content of Evolution Notes 41
C.8 OR-Tools Case: Experimental Settings 44
C.9 Evolutionary Lineage Case Studies 46
C.10 Evaluator Exposure: Protocol, Held-out Suites, and Code Cases 48
D Prompts **52**
D.1 Orchestrator 52
D.2 Initial Directions 54
D.3 Task Executor 55
D.4 Notes 57
D.5 Meta-guidance 59

A METHOD DETAILS

A.1 INTEGRATION WITH CODING-AGENT HOSTS

EvoTrias's allocation of decision rights defines how an existing coding agent can participate in evolutionary search. Governance owns cross-round selection, experience admission, and budget accounting; Judgment owns official evaluation. Proposal consumes the assigned context and returns artifacts through the contracts of Equation 2. Separating these responsibilities allows the implementation of Proposal to change across hosts while retaining the governance rules and evaluation semantics. EvoTrias realizes this separation as a plugin that adds an evolution workflow to the host's native execution capabilities; every EvoTrias result in this paper was produced through this plugin, on Claude Code or OpenCode.

Workflow in the host session. For a prepared task and evaluator, a skill activates the workflow in the user's current coding-agent session. That session becomes the orchestrator: it initializes the run, follows the scheduling decisions returned over MCP, and dispatches native subagents for construction and requested experience generation. Executors retrieve their assigned context from Governance and submit artifacts to Judgment; settled outcomes inform subsequent tasks. The plugin supplies the workflow skills, agent roles, MCP configuration, and runtime hooks. The host supplies the conversational entry point, tools, and native subagent execution; dedicated components maintain search state and official evaluation. Appendix D.1 gives the orchestrator's bounded dispatch protocol.

What a host must supply. A compatible host must support the context, artifact, and outcome contracts of Equation 2. At runtime it must provide native task dispatch, stable run and task identities, and lifecycle events that can maintain the shared ledger. Its integration must also enforce the task-access and evaluator-access boundaries of §4.3. A thin adapter translates the host's native tool calls, identifiers, and events into these shared interfaces: the four event roles of Table 5 for scheduling, the pre-tool event for permission interception, and the host's own role and MCP configuration. Appendix A.7.5 shows this mapping for Claude Code and OpenCode. Under these requirements, host replacement changes the Proposal implementation while selection, Notes admission, budget accounting, and evaluation retain their defined semantics.

Two paths into a task. Project knowledge and search evidence reach a task by different paths. Project files, build instructions, and skills enter through the public task contract and the task's access rules, so knowledge the project already holds is available to construction. Search evidence enters only through Governance, which decides which parent, inspirations, Notes, and guidance each task receives. Subagents run in separate conversation contexts, so this second path is the only route by which one attempt's experience reaches another. Artifacts leave a task the way project changes normally do: in the OR-Tools case (Appendix C.8), construction exports a Git patch against the frozen tree, and the project's review workflow applies it directly.

Operating an evolution system versus becoming one. Figure 9 places the two systems side by side. CORAL (Qu et al., 2026) deploys as a standalone runtime: a Python AgentManager launches and supervises the coding agents that carry out the search, and the user operates that runtime through a CLI that launches, inspects, resumes, and stops runs, with a dashboard for observation. EvoTrias launches no agents of its own: it deploys as a plugin, and the search runs inside the host coding agent. CORAL also ships a plugin, but as a supplementary entry point: it lets the user's coding agent issue the same CLI operations, so the user's agent operates the evolution system from outside it.[1] EvoTrias's plugin makes the user's coding agent the evolution system: the session the user already works in orchestrates the search, its native subagents construct candidates, and the plugin adds the governance and judgment components those subagents talk to. The same session that develops a project can therefore evolve it, with cross-round decisions and official evaluation held outside the agent's discretion.

[1] CORAL public revision `0123dfb` (2026-09-06): the operator skill under `plugin/skills/`, the CLI launch in `coral/cli/start.py`, and the AgentManager in `coral/agent/manager.py`.

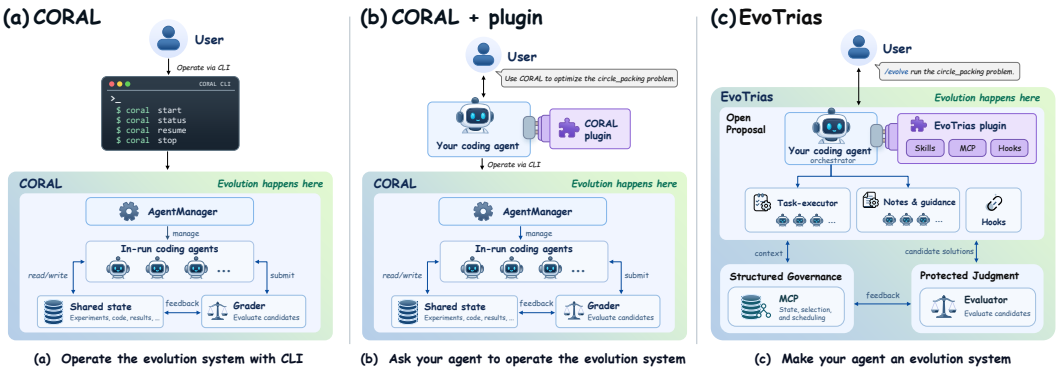


Figure 9: From operating an evolution system to becoming one. (a) The user operates CORAL through its CLI. (b) CORAL's plugin lets the user's coding agent issue the same operations; the search still runs in a separately managed runtime. (c) EvoTrias's plugin makes the user's coding agent the evolution system: the existing session orchestrates native subagents, while dedicated components govern the search and judge its artifacts. All three assume a prepared task and evaluator.

A.2 Design Details of Structured Governance

Appendices A.2 to A.5 complete the design of §4.2. This section covers the parts owned by Governance alone: the descendant potential and the full selection rules behind (π^L, π^P, π^I). The initial directions and the two experience channels, whose content is produced on the Open Proposal side, follow in Appendices A.3, A.4, and A.5. Default values for all constants appear in Table 9.

A.2.1 Descendant Potential

For a parent or lineage u whose settled descendants number n_u, with s_u of them improving on their parent,

$$g_u = \frac{\alpha + s_u}{\alpha + \beta + n_u} - \frac{\alpha}{\alpha + \beta} \tag{8}$$

shrinks this improvement rate toward a neutral prior: credit accrues only as improving descendants accumulate, and $g_u = 0$ without descendant evidence. The same form applies at both granularities. Parent-level g_i enters the parent score of Equation 6 with weight λ_P; lineage-level g_ℓ enters the routing score below with weight λ_L. Both weights are fixed and small, so potential breaks ties among comparable candidates without overriding verified quality.

A.2.2 Selection Rules

Annealing. Every scheduled constant follows one exponential form. For endpoints (x_0, x_1) and budget progress $\beta_t \in [0, 1]$,

$$x_t = x_0 \, (x_1/x_0)^{\beta_t}. \tag{9}$$

The routing temperature τ_t, the exploration coefficient $c_{\text{local},t}$, and the inspiration temperature T_t all anneal this way, which moves the search from broad to deep as the budget depletes.

Lineage routing. For lineage ℓ, q_ℓ is the min-max normalized best verified fitness and a_ℓ its share of the successful population:

$$r_\ell = q_\ell + \lambda_L g_\ell, \qquad d_\ell = \frac{\exp(r_\ell/\tau_t)}{\sum_k \exp(r_k/\tau_t)},$$

$$D_\ell = \frac{d_\ell}{a_\ell + 1/N}, \qquad \ell_t^\star = \arg\max_\ell D_\ell(1 + \eta r_\ell), \tag{10}$$

$$\pi_t^L(\ell) = p_{\text{exploit}} \mathbf{1}[\ell = \ell_t^\star] + (1 - p_{\text{exploit}}) \frac{D_\ell}{\sum_k D_k}.$$

Quality and potential set the deserved share d_ℓ; the deficit D_ℓ compares it with the realized share a_ℓ, smoothed by $1/N$ for a population of N candidates, favoring strong but under-covered lineages.

With probability p_{exploit} the routing takes the deficit leader with a quality bonus of strength η; the remaining mass is deficit-weighted roulette.

Parent rule. In Equation 6, quality and exploration take

$$q_i = \frac{N - \text{rank}(i)}{\max(N-1, 1)}, \qquad U_{i,t} = c_{\text{local},t}\sqrt{\frac{\log(S_{\ell_t}+1)}{e_i^{\text{eff}}+1}}, \tag{11}$$

where N is the population size, $\text{rank}(i)$ orders all candidates by fitness, and S_{ℓ_t} counts parent selections from the lineage. The effective exposure count applies an elite shield,

$$e_i^{\text{eff}} = \begin{cases} (1-s)\, e_i, & \text{rank}(i) \le \lceil \kappa N \rceil, \\ e_i, & \text{otherwise}, \end{cases} \tag{12}$$

where e_i is the raw selection count, κ the elite rank fraction, and s the shield strength. Repeated selection therefore suppresses ordinary candidates faster than top-ranked ones, which keeps the best candidates reachable late in the run. Each selection increments e_i immediately, so concurrent tasks see the updated count.

Inspiration rule. In Equation 7, local sampling is Boltzmann over min-max normalized fitness within the parent lineage, $\pi_{\text{local}}(i \mid \ell_t, T_t) \propto \exp(\widetilde{f}_i/T_t)$. The cross-lineage pool collects the top n_e records of every other lineage, and the donor distribution ρ_t weights these pools. Drawing uniformly over the pooled records is the baseline while the population stays balanced: it gives every lineage donor mass in proportion to the elites it can actually supply, which is equal once every lineage holds n_e of them. Let H_t be the normalized Shannon entropy of the lineage shares. As it falls, the donor mass tilts toward quality,

$$\rho_t(k \mid E_t) \propto \min(|k|, n_e)\,\big(1 + \gamma_t\, \widetilde{q}_k\big), \qquad \gamma_t = \gamma_{\max}\,\text{clip}\Big(\frac{H^\star - H_t}{H^\star - H_{\min}}, 0, 1\Big), \tag{13}$$

where $\widetilde{q}_k$ is the min-max normalized best fitness of lineage k and $|k|$ is the number of records lineage k holds. The ramp spans $[H_{\min}, H^\star]$ rather than $[0, H^\star]$ because normalized entropy is insensitive to mild skew: four lineages holding $(20, 5, 5, 5)$ records sit at $H_t = 0.83$, and $(14, 6, 6, 6)$ at 0.94. The tilt is bounded by construction: the quality factor never exceeds $1 + \gamma_{\max}$, so quality can move a lineage's donor mass by at most that factor, and at $H_t \ge H^\star$ the weights fall back to the pool-size baseline. The remaining variation is the pool-size factor $\min(|k|, n_e)$, which is bounded by n_e and equal across lineages once each holds n_e records. The migration rate stays fixed, so the volume of cross-lineage material is unchanged and only its composition moves. Each inspiration entry draws independently with migration probability μ, skips records already selected, and falls back to the whole population when the local pool is exhausted. The prompt presents the parent first, then local inspirations, then migrated ones, each group sorted by fitness.

Boundary cases. The rules above run on the population of settled candidates that the evaluator marked valid with a numeric fitness; invalid attempts never enter N, the rank, or any lineage share (Appendix A.7.2). $\text{rank}(i)$ is global over this population and strict: two candidates with equal fitness are ordered by settlement time, the earlier one ranking higher, so q_i never assigns the same quality to two candidates. Every min-max normalization guards a zero range. When all lineages share the same best fitness, q_ℓ and $\widetilde{q}_k$ are set to zero, so routing falls back to potential and deficit alone and the donor weights fall back to the pool-size baseline; when all candidates in a lineage share one fitness, π_{local} is uniform. A lineage without a settled success has $a_\ell = 0$ and so the largest deficit for its deserved share; this is the intended behavior, since such a lineage has received the least coverage. With a single lineage, routing is skipped and the parent rule runs directly; before any lineage exists, the parent is the global best.

A.3 Initial Directions

Root establishment opens every run. The orchestrator dispatches a dedicated init-explorer agent that reads the public contract and the initial reference program, writes a short structural analysis of the problem (its type, the available feedback, and the decision axes where solvers can diverge), and

generates the n_{dir} directions $\mathcal{D}_0 = \{d_i\}_{i=1}^{n_{\text{dir}}}$, one per seeded root. Each d_i is a single line of 8–25 words naming a concrete approach. A pairwise cross-check then replaces any two directions that would converge on the same core strategy, so the roots start from distinct approach families instead of variations of the reference program. Each d_i binds only its root task, and the descent subtrees of the roots become the lineages of §4.2. The generation prompt appears in Appendix D.2.

Figure 10 shows the four directions generated for the MinMax-Dist-3 run of Figure 14. The four lineages settled 7, 24, 8, and 7 candidates. The polyhedron-construction root grew into the dominant lineage: it carried the surrogate optimum that round-1 guidance later diagnosed, and all three guided new bests descend from it. The other three lineages topped out between 0.9948 and 0.9975.

Initial directions MinMax-Dist-3 run of Figure 14; one line per seeded root, quoted verbatim

Root 1 - Spherical Tammes code. "Spherical Tammes code: place all 14 points on the unit sphere using the known optimal 14-point spherical configuration to fix max distance at diameter 2." **Fate:** 7 settled descendants; lineage best 0.9975.
Root 2 - Polyhedron vertex construction. "Catalan/Archimedean polyhedron vertex construction: use vertices of a 14-vertex solid like the rhombic dodecahedron or triakis octahedron as a closed-form symmetric arrangement." **Fate:** 24 settled descendants; carries the canonical best throughout and all three guided new bests of Figure 14; lineage best 1.00002.
Root 3 - Hexagonal antiprism plus poles. "Hexagonal antiprism plus two poles: arrange 12 vertices on two staggered hexagonal rings plus north and south poles along the symmetry axis." **Fate:** 8 settled descendants; lineage best 0.99627.
Root 4 - Numerical optimization. "Numerical optimization approach: use scipy.optimize with multiple deterministic seeds to maximize the minimum pairwise distance, then normalize max distance to 1." **Fate:** 7 settled descendants; lineage best 0.99482.

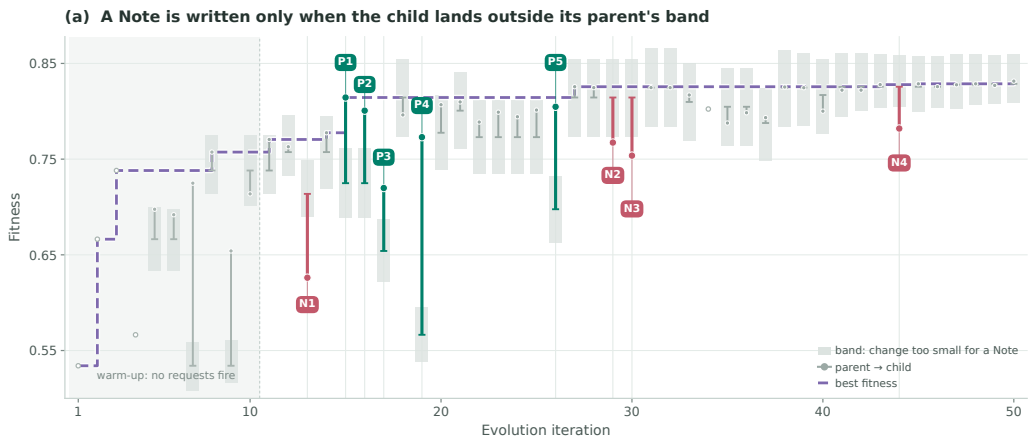
Figure 10: The four initial directions of the MinMax-Dist-3 run, quoted from the stored direction files, with the settled size and best fitness of the lineage each root seeded.

A.4 Notes

Triggers. After each settled attempt with fitness f and parent fitness f_p, the relative delta is $\delta = (f - f_p)/\max(|f_p|, \varepsilon)$, with ε a small numerical floor. A note request fires in three cases: δ clears the positive threshold; the magnitude of a negative δ clears the negative threshold; or the attempt history shows a repair (failure followed by success within the task) or a terminal failure. Each threshold is the q-quantile of the past positive or negative delta magnitudes, with q annealed from 0.80 to 0.60 over iterations 11–50 and the result clamped to the $[0.02, 0.05]$ band; an empty history takes the band ceiling. Early noise therefore cannot flood the pool, and late thresholds stay meaningful. A positive request must also set a new best or satisfy $f/b^- \geq r_t$, where b^- is the best settled fitness before the attempt and r_t anneals from 0.80 to 1.00 over iterations 11–50. Negative requests use only the corresponding delta threshold. No requests fire during the first ten iterations. Near-duplicate positive requests at the same fitness level are skipped, and every request carries a deterministic id, so retries commit at most once.

Loading. Note loading runs after the parent and inspirations are fixed. Together they form the task's *anchor set*, and a stored note is *anchor-matched* when its own provenance, the task it was distilled from together with that task's parent and inspirations, intersects this set. The most recent error notes load first, up to ten; they load by recency and are never anchored. Positive and negative notes then fill a target of five: anchor-matched notes take priority, and a seeded random draw fills the remainder, which keeps repeated loads stable for the same context. The share of loaded positive and negative notes that are anchor-matched is the *anchor hit rate*, which separates retrieval tied to the consuming context from a draw over the whole pool; Appendix C.6 measures it across the audited runs. The analysis prompt sent with each note request appears in Appendix D.4.

Figure 11 replays these rules on one 50-iteration Signal Processing run with Opus-5. Both delta thresholds are relative to the parent, so they can be drawn as a band $[\,p\,(1-\tau_t^-),\ p\,(1+\tau_t^+)\,]$ around each parent's fitness p. This reduces the magnitude test to one visual comparison: a request fires exactly when the child lands outside that band. The band is asymmetric because the positive and negative thresholds adapt independently, and the first four attempts have no parent, so they carry no band. The panel omits the gate; the ledger instead lists f/b^-, which for the five positive Notes can



(b) Ledger of the committed Notes

N1	P1	P2	P3	P4	P5	N2	N3	N4
iteration 13	iteration 15	iteration 16	iteration 17	iteration 19	iteration 26	iteration 29	iteration 30	iteration 44
−12.3%	+12.4%	+10.5%	+10.1%	+36.5%	+15.3%	−5.8%	−7.5%	−5.3%
threshold ±3.3%	threshold ±5.0%	threshold ±5.0%	threshold ±5.0%	threshold ±5.0%	threshold ±5.0%	threshold ±5.0%	threshold ±5.0%	threshold ±2.6%
vs. best 81.3%	vs. best 104.7%	vs. best 98.3%	vs. best 88.4%	vs. best 94.9%	vs. best 98.8%	vs. best 93.0%	vs. best 91.4%	vs. best 94.5%

Figure 11: Note generation in a 50-iteration Signal Processing run with Opus-5. **(a)** Each settled attempt is a segment from its parent's fitness to its own; the grey band is the range around the parent within which the change is too small to trigger a request. Segments that leave the band are colored by the pool they fed. The dashed line is the best fitness settled so far, and the shaded opening interval is the ten-iteration warm-up. **(b)** The nine positive and negative Notes committed in this run, with the iteration, the child–parent delta, the threshold in force, and the ratio to the best fitness settled before the attempt; the run's single error Note is omitted, as in panel (a). P1–P5 and N1–N4 match Figures 12 and 13.

be read against the r_t schedule above. Governance committed five positive and four negative Notes; its failed-to-success event produced a separate error Note and is not counted in the two plotted pools.

P1 Suppress reversals by post-hoc persistence pruning, not by narrowing bandwidth

Lesson. Smoothness and accuracy trade off only when bandwidth is the sole knob. Adding persistence pruning of sub-noise extrema plus monotone PAVA projection after the zero-phase order-3 smoother cut reversals further than narrowing bandwidth, without the lag cost.
Actionable guidance. Keep the smoother bandwidth wide (corner coefficients ~1.0-5.0 over W) and get smoothness from a separate persistence-pruning + PAVA projection stage; select bandwidth, adaptivity and pruning threshold jointly against a surrogate whose accuracy term is a Cp risk on the final pruned output.
Evidence. task_015 fitness 0.8144 vs parent task_007 0.7249 (+12.35%), best inspiration task_011 0.7705. slope_changes 18.0 -> 5.4, false_reversals 13.4 -> 3.2, lag_error 0.208 -> 0.226, correlation 0.872 -> 0.817. Bandwidth-only task_011 got slope_changes 7.8 but lag_error 0.294. The winning internal candidate (corner, gamma, tau) was not isolated.
Confidence. medium.

P2 Cut slope changes with a shape-domain post-stage, not a narrower band

Lesson. Because smoothness_score = 1/(1 + slope_changes/20), reversals dominate fitness. After Whittaker smoothing, 4-sigma persistence pruning and PAVA cut reversals at near-zero correlation cost, unlike spectral narrowing.
Actionable guidance. Keep the adaptive Whittaker core and its _C_NOMINAL=2.2 corner unchanged; tune smoothness via the stage-2 persistence threshold (_PERSIST_K=4.0 times post-filter sigma) instead of lowering the cutoff.
Evidence. task_016 attempt_001 fitness 0.8007 vs parent task_007 0.7249 (+10.46%). slope_changes 18.0->7.8, false_reversals 13.4->5.0, smoothness 0.526->0.719, correlation 0.872->0.869, eval_time 0.13s->0.27s. Stage 2 runs unconditionally, no fallback branch.
Confidence. medium.

P3 Select L1 trend-filter lambda by surrogate sweep

Lesson. On the same L1/ADMM + knot-refit estimator, replacing the parent's fixed discrepancy-principle lambda with a 12-rung geometric lambda ladder scored by an internal surrogate (0.58*smoothness + 0.34*agreement + 0.08*(1-lag)) raised fitness 0.654 -> 0.720. The idea was adapted from inspiration task_014. Tightened knot extraction shipped in the same attempt, so the sweep is not isolated.
Actionable guidance. For a single-knob estimator, sweep a wide geometric ladder and keep the rung maximising a saturating-smoothness surrogate scored on the emitted slice; use the noise-matched rung only as the accuracy reference.
Evidence. task_017/attempt_001 fitness 0.7198 vs parent task_009 0.6540 (+10.1%). slope_changes 20.0->10.6, smoothness 0.500->0.654, correlation 0.746->0.756. Still below task_014 (0.778).
Confidence. medium.

P4 Select reversal-pruning level by surrogate objective

Lesson. Replacing the fixed kappa=3 pruning threshold with an explicit selector likely drove the gain: enumerate the full smallest-swing-first pruning path of a Kalman/RTS track over wide bandwidths (lam_ml x {1,4,16}) and pick the pair maximizing 0.40*corr_proxy + 0.32/(1+0.05*slope_changes), with corr_proxy a Mallows Cp estimate and a floor forbidding corr drops >0.06 below the unpruned track.
Actionable guidance. When the objective mixes observable slope changes with unobservable clean-signal correlation, enumerate pruning once and select with exact slope counts, Cp risk, and an accuracy floor; do not hand-tune a significance constant.
Evidence. task_019/attempt_001 fitness 0.7730 vs parent task_004 0.5664 (+36.5%) and best inspiration task_008 0.7574. slope_changes 24.4 -> 13.8, smoothness_score 0.450 -> 0.592, correlation 0.976 -> 0.959. Costs: lag_error 0.128 -> 0.436, noise_reduction 0.869 -> 0.585, eval_time 1.9s -> 3.8s.
Confidence. medium.

P5 Jointly sweep bandwidth and pruning with a Cp surrogate

Lesson. Freeing the smoothing corner as a second knob alongside pruning level, rather than fixing it at C=3.55, improved every axis at once: slope changes 25.6 -> 13.0, false reversals 19 -> 7.4, correlation 0.911 -> 0.957, lag error 0.223 -> 0.155. The Cp accuracy term likely prevented the over-pruning seen in the pruning-only inspiration.
Actionable guidance. With a cheap O(n) core, jointly sweep bandwidth (1.60-7.50 over W) and the pruning path; score the emitted z[W-1:] slice by Cp accuracy, not against the candidate's stage-1 track.
Evidence. task_026/attempt_001 fitness 0.8047 vs parent task_005 0.6976 (+15.3%) and best inspiration task_014 0.7776. The grid has 5 bandwidths and at most 26 pruning rungs; output is the argmax.
Confidence. medium.

Figure 12: The five positive Notes committed during the Signal Processing run in Figure 11; each card lightly condenses the stored fields, and P1–P5 match the trigger labels in the replay.

N1 Do not swap the L2 roughness post-filter for this unguarded L1 dead-zone + knot refit

Lesson. Swapping the parent's L2 roughness post-filter for an L1 curvature dead-zone on the Kalman/RTS track (0.25-sigma displacement budget, knot refit at rel tol 2e-3 and 20% knot cap) lost accuracy and smoothness simultaneously instead of trading one for the other; the permissive knot set added slope changes rather than removing them.
Actionable guidance. Avoid this unguarded variant; if trying L1 curvature post-filtering, use a much stricter knot tolerance and smaller knot cap, size the budget from the smoothed track's own residual scale, and keep stage 2 only if smoothness and correlation both improve.
Evidence. task_013 attempt_001 fitness 0.6261 vs parent task_010 0.7137 (-12.3%). Correlation 0.854 -> 0.688, smoothness 0.575 -> 0.372, slope_changes 14.8 -> 33.8, false_reversals 11.6 -> 26.2. Best reference is still pure Kalman/RTS with no post-filter (task_003, 0.7381).
Confidence. medium.

N2 Gate L0 changepoint branches on reversal count, not just a 1.30x Cp-risk bar

Lesson. Admitting L0/PELT segmented branches whenever their Cp risk stayed within 1.30x branch A's risk let a many-segment fit win on accuracy while tripling reversals; smoothness_score = 1/(1+slope_changes/20) dominates this objective.
Actionable guidance. If pairing a changepoint branch with the smoother, also require it to match branch A on slope_changes/false_reversals, or route it through the same persistence pruning; do not admit it on a Cp-risk ratio alone.
Evidence. task_029/attempt_001 fitness 0.7673 vs parent task_015 0.8144 (-5.8%). correlation 0.817->0.943, avg_error 0.414->0.340, but slope_changes 5.4->15.0, false_reversals 3.2->12.8, smoothness_score 0.787->0.571, eval_time 2.6s->6.7s. No per-record branch tag in eval output, so the executed branch is unverified.
Confidence. medium.

N3 Guard cross-branch surrogate against reversal inflation when adding a changepoint branch

Lesson. Adding an L0 changepoint branch beside the penalised smoother raised correlation (0.817 -> 0.943) but quadrupled slope_changes (5.4 -> 21.4) and false_reversals (3.2 -> 19.2), so smoothness_score fell 0.787 -> 0.483 and fitness regressed 7.5%. The cross-branch surrogate under-weighted the reversal terms.
Actionable guidance. If adding a changepoint branch, raise the reversal weights in the cross-branch surrogate (_W_SMOOTH >= 0.45, _W_FALSE_REV >= 0.20) or veto branch B when its slope_changes exceed the smoother branch's; do not rely on the correlation term alone to arbitrate.
Evidence. Child 0.754 vs parent 0.814. slope_changes 21.4 vs 5.4, false_reversals 19.2 vs 3.2, correlation 0.943 vs 0.817. Executed branch not recorded in eval_result, so branch attribution is unverified.
Confidence. medium.

N4 Validate replacement selection criteria against realised correlation and noise_reduction

Lesson. A five-term plug-in criterion (reversals 0.36, purity 0.18, noise-corrected correlation 0.20, linear 1-risk/sigma^2 0.14, noise-floor-saturated tracking 0.12) replacing the lineage surrogate picked over-smoothed shapes: reversals halved while correlation and noise_reduction both fell, fitness -5.29%. Smoother and DP were unchanged, so the criterion configuration is implicated.
Actionable guidance. Before adopting a replacement selection criterion, check its realised slope_changes, correlation and noise_reduction against the parent; reuse an existing smoothed rung and cost table for any reference track rather than adding a separate DP pass.
Evidence. task_044/attempt_001 vs parent task_027: fitness 0.7820 vs 0.8257 (-5.29%); slope_changes 2.5 vs 5.0; correlation 0.744 vs 0.821; noise_reduction 0.464 vs 0.641; success_rate 0.8 vs 1.0. Estimator code identical to parent.
Confidence. medium.

Figure 13: The four negative Notes committed during the Signal Processing run in Figure 11. N1–N4 match the trigger labels in the replay.

Across the five positive Notes, the recurring pattern is to preserve a reasonably wide smoother and control reversals through a separate shape-domain stage, while selecting bandwidth and pruning strength with an accuracy-aware surrogate. The four negative Notes mark the complementary failure modes: permissive knot refits, changepoint branches admitted without reversal checks, and replacement criteria that reward smoothness while losing correlation or noise reduction. These observations are specific to the recorded comparisons. Several Notes explicitly state that the winning branch or individual change was not isolated, so they support conditional guidance rather than a general claim about an estimator family.

A.5 META-GUIDANCE

Stall detection. Let $b_1, \ldots, b_n$ be the running best fitness after each success. With window w, the windowed improvement is

$$I_n = \frac{b_n - b_{n-w}}{\max(|b_{n-w}|, \varepsilon)}, \tag{14}$$

and a window stalls when $I_n \leq 0$. The detector requires at least $\max(w, 5)$ successes before judging. A guidance request fires when P_{stall} consecutive windows stall and the newest outcome is not a new best; a new best resets the count.

Request and commit. At most one request is pending at a time. The request packages the canonical best with an artifact excerpt, stall diagnostics and budget progress, per-lineage summaries, a summary of admitted notes, and the full history of earlier guidance with their recorded outcomes. The proposal-side meta actor returns a small set of semantically distinct directions. Each direction is a typed record: an approach label with a one-line idea, a description that argues from the lineage record, a step-level implementation plan, execution cautions, a novelty claim against the tried history, and an avoid-repeating list that names the tactics the lineages have already spent. Governance validates this schema and commits the directions. The request payload and the generation instructions appear in Appendix D.5.

Active use. Committed directions activate one at a time. When a task is assigned the active direction, its parent anchors to the canonical best at assignment time, and the direction's use count increments, up to two uses. After each guided task settles, the fitness change against the anchor is recorded on that direction. Exhausted directions retire into the history that the next request will carry, so the meta actor sees which directions helped and which failed before proposing new ones.

A lifecycle case. Figure 14 replays this lifecycle on one 50-iteration MinMax-Dist-3 run, which asks for 14 points that maximize the squared ratio of minimal to maximal pairwise distance. The fitness f in the figure is the evaluator's normalized score, so $f = 1$ matches the AlphaEvolve reference construction (Novikov et al., 2025); on the raw scale of Table 1 this run's final best $f = 1.00002$ is 0.240051, which rounds to the 0.24005 that table averages over three runs. Two stalls trigger three guidance requests and nine committed directions (Figure 15).

The three rounds do different work. The first request diagnoses the stall from the lineage record: eleven outcomes repeat one smooth surrogate whose annealed gradients share a single basin, so all three directions replace the machinery instead of tuning it. D1's hard constrained program lands the new best, and its two uses also show the protocol's tolerance, one failed implementation and one success. The second round asks whether that optimum can be polished further and spreads its directions over three method families. Two of them return the anchor value to the last digit and the third lands below it; two independent families arriving at the same point is consistent with an exhausted basin. The third round answers that record. D7's closed form returns the identical value a third time, while D8 keeps the program and changes the solver family and D9 keeps the solution and perturbs it. Both escapes cross the reference level, and D9 sets the final best.

The case also shows the lifecycle's bookkeeping at work. Failure history is visible in content: the round-3 avoid-repeating lists name round-2 tactics, and D9 cites D6's seed portfolio verbatim. Anchoring is visible in structure: D1's artifact is the anchor parent of rounds 2 and 3, later directions load it by task id in their plans, and each retirement records its effect against that anchor. The contrast is visible in the unguided lane: between the rounds, unguided descendants keep sampling the surrogate family and settle between 0.9894 and 0.9975, all short of the anchor, and all three new bests after the ramp-up come from guided tasks.

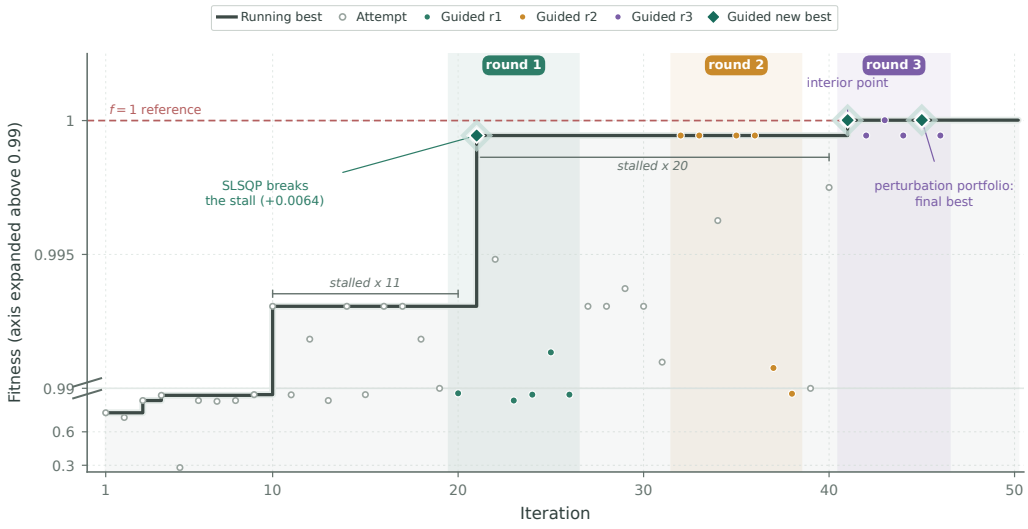


Figure 14: Meta-guidance lifecycle on a 50-iteration MinMax-Dist-3 run. The y-axis expands fitness above 0.99; the double slash marks the compressed range. Shaded bands cover the iterations bound to each guidance round, colored dots mark guided outcomes, diamonds mark guided outcomes that set a new best, and brackets measure the two stalls. Figure 15 lists the committed directions.

Round 1 requested after 11 stalled outcomes; canonical best 0.9931 when the round opened

D1 - Constrained dual program. The request names the bottleneck: every lineage refines one smooth surrogate whose annealed gradients share a single basin at 0.9931. This direction drops the surrogate and attacks the hard program itself, maximizing t subject to $d_{ij}^2 \geq t$ and $d_{ij}^2 \leq 1$ over all pairs with an active-set SQP solver. **Plan:** encode the 43-variable program with explicit pairwise constraints, run SLSQP from 8-12 normalized random starts, keep the best feasible point, then polish on the raw ratio. **Caution:** verify feasibility after SLSQP and re-normalize, since the solver can stop at a slightly infeasible point. **Outcome:** 0.9452 and 0.99944; new best ($+0.0064$), and its artifact anchors both later rounds.
D2 - Exact algebraic construction. Bypass numerical search: build the 14-point configuration from coordinates reported for the 14-point Tammes problem, constructed algebraically so the ratio equals the claimed optimum. **Plan:** hard-code the exact coordinates with cited sources, normalize, and clean up floating-point error with one surrogate polish. **Caution:** the request forbids fabricated coordinates and demands a numerical fallback if the exact optimum is uncertain. **Outcome:** 0.8803 and 0.9331; retired.
D3 - Spherical-code portfolio. Recast the task as a spherical code: place 14 points on the unit sphere for maximal angular separation, seeding from published code tables that the request judges more diverse than the five ring-based seeds the lineages reuse. **Plan:** hard-code three to four candidate codes, run each through the existing polish pipeline, keep the best. **Caution:** skip seeds that are rotations of the rhombic dodecahedron, which would collapse into the known basin. **Outcome:** 0.9913 and 0.9332; retired.

Round 2 requested after 20 stalled outcomes; canonical best 0.99944 when the round opened

D4 - Hybrid global-local polish. Feed the round-1 SLSQP artifact into the smooth-surrogate pipeline of the earlier lineages, on the reasoning that SLSQP fixes the feasibility-boundary structure while the surrogate refines interior spacing. **Plan:** load the anchored best by task id, run the surrogate refine, a Nelder-Mead polish, and two basin hops; return the best of the three stages. **Caution:** if the polish regresses, return the anchor unchanged. **Outcome:** both uses return exactly 0.99944.
D5 - CMA-ES black box. Treat the 42-dimensional configuration as a black box and run derivative-free CMA-ES seeded at the anchored best, since a population method can cross basins that gradient steps cannot. **Plan:** 2,000 evaluations at $\sigma_0 = 0.05$, an optional restart with doubled population, then a Nelder-Mead polish. **Caution:** guard degenerate configurations and fix the random seed for reproducibility. **Outcome:** both uses return exactly 0.99944.
D6 - Structured multi-start. Expand SLSQP from 11 seeds to more than 40, drawn from named symmetry families (hexagonal antiprism with poles, heptagonal bipyramid, snub cube, icosahedron with poles, and others), so each start explores a different KKT neighborhood. **Plan:** normalize every seed, run SLSQP per seed, keep the best feasible result, polish. **Caution:** drop seeds that are rotations of one another. **Outcome:** 0.9908 and 0.9412; retired.

Round 3 requested with the round-2 record in the failure history; canonical best 0.99944 when the round opened

D7 - Dual-anchor ring construction. Solve a closed-form construction with two antipodal poles and two staggered rings of six, where equal near-neighbor distances reduce the whole configuration to a single co-latitude angle found by one-dimensional root finding. **Caution:** if the construction lands below the anchor's raw ratio, fall back to the anchor. **Outcome:** both uses return exactly 0.99944, rediscovering the same optimum from a third method family.
D8 - Interior-point solver. Keep the dual program of D1 and swap the active-set solver for a primal-dual interior-point method (trust-constr), which reaches KKT points through a barrier path that active sets can miss. **Plan:** reuse the parent's analytic constraint Jacobians, run from the 11 seeds plus the anchor, keep the best feasible point, polish. **Caution:** bound iterations, and return the anchor if the method fails to improve. **Outcome:** 1.000013 in both uses; crosses $f = 1$.
D9 - Perturbation portfolio. Re-run SLSQP from twelve structured perturbations of the anchored best: six small Gaussian kicks, three rigid rotations against axis-alignment bias, and three point-pair swaps against permutation symmetry. Its avoid-repeating list cites D6's seed portfolio by name. **Caution:** every perturbation starts from the anchor itself, never from another perturbation, to avoid drift. **Outcome:** 1.0000159, the final best, and 0.99944.

Figure 15: The nine directions committed during the MinMax-Dist-3 run of Figure 14, condensed from the stored direction records and grouped by round. D1–D9 follow commit order; plans and cautions are excerpts of the stored fields, and every outcome is the official fitness of a guided task.

A.6 Context Staleness under Concurrency

This appendix models how concurrency delays the settled evidence available when a task binds its context (§6.4).

Setup. Index tasks by the logical instant $t = 1, \dots, T$ at which each binds its context (§4.1), with a fixed task budget T. Under concurrency c, up to c tasks are in flight simultaneously. Assume equal task durations and FIFO settlement. After the initial queue-filling phase, $c - 1$ previously bound tasks remain in flight at each new binding. With y_j denoting the settled record of task j, task t therefore observes

$$\mathcal{H}_t^{(c)} = \{\, y_j : j \le t - c \,\}, \tag{15}$$

empty when $t \le c$. The capsule binds as $\Gamma_t = G(\mathcal{H}_t^{(c)})$ with evidence state $E_t = \Phi(\mathcal{H}_t^{(c)})$. Serial execution recovers $\mathcal{H}_t^{(1)} = \{y_j : j \le t - 1\}$, the largest history this model permits. Each additional concurrent slot withholds one recent outcome from every post-warm-up binding.

What concurrency withholds. Equation 15 gives the cost of concurrency a precise form. For $c' > c$ and every binding instant $t > c'$, $\mathcal{H}_t^{(c')} \subseteq \mathcal{H}_t^{(c)}$: raising concurrency from c to c' removes the $c' - c$ most recent settled outcomes from every post-warm-up binding, so a task under concurrency c binds with an evidence lag of $c - 1$ outcomes. This subset relation follows from the execution model alone. Whether it costs fitness depends on how much the evidence state E_t (Eq. 3) changes over a window of $c - 1$ outcomes, and that is an empirical question rather than a property of the model.

Two features of the withheld entries matter for reading that question. First, they are official outcomes settled by Protected Judgment and distilled by Governance into ranks and potential estimates (§4.2). They are not the unchecked single-attempt lessons that §1 identifies as the source of premature narrowing in agent-centric memory. The concern there is that a lesson enters shared memory with no owner to test it, so more of it is not better. Here every withheld entry has already passed the check, and Governance rather than the proposing agent decides how far it travels. The loss under concurrency is therefore one of delayed evidence, not of unvetted evidence. Second, the relation is local: it compares two concurrencies on a common settled history. Full adaptive runs follow concurrency-dependent trajectories, since later outcomes were themselves produced under earlier stale contexts, so the end-to-end effect has to be measured.

Measured effect. Figure 8 reports that measurement. Degradation stays small through $c = 4$ and grows at $c = 8$, consistent with the evidence lag $c - 1$ crossing the horizon over which the evidence state changes materially. Serial execution maximizes the evidence available at each binding, while wall-clock time scales approximately as $T\bar{\tau}/c$ for mean task duration $\bar{\tau}$. Figure 8 therefore identifies the practical operating point between these objectives.

A.7 Runtime Hooks

This appendix details the enforcement layer of §4.3. Open Proposal grants broad authorship inside a task, and two things cannot be left to that grant. First, the run's bookkeeping cannot rest on the orchestrator's compliance, since the orchestrator is an autonomous agent too. Second, the prohibition on reading evaluator sources guards the interface to Protected Judgment, and a single file read would breach it. The hook layer enforces both outside the agent, at the events the host runtime emits before and after each tool call and at each subagent lifecycle transition. Scheduler hooks maintain the task ledger. Permission hooks decide individual tool calls.

A.7.1 Interception and Dispatch

The dispatch chain. Each hook event runs as a separate short-lived process. The host writes the event payload to standard input as JSON and reads any response from standard output. Inside that process a broker looks up the handlers registered for the pair (host, event) and calls them in descending priority, chaining each return value into the next call. A handler that raises is logged and skipped while the rest still run, so one faulty rule cannot disable the layer.

Serialization. Handlers share run state across process boundaries, so in-process locks give no protection. The broker takes a cross-process file lock around the whole dispatch, which makes an event's read-modify-write of the ledger atomic against concurrently spawning executors. On timeout the dispatch is abandoned and logged; no handler runs and no partial state is written.

A.7.2 The Task Ledger

A run's tasks occupy one pool with the six states of Table 4. Every transition is driven by a host event, never by an agent's own report. A transition whose precondition fails is rejected and logged, so duplicate or out-of-order events cannot corrupt the pool.

State	Entered when	Event source
`created`	Governance mints identifiers at a convergence check	control-plane call
`pending`	the hook allocates one to an observed executor spawn	before the tool call
`queued`	the spawn returns an agent handle	after the tool call
`running`	the subagent starts	subagent lifecycle
`completed`	the subagent stops with a successful attempt on record	subagent lifecycle
`failed`	the subagent stops with no successful attempt on record	subagent lifecycle

Table 4: States of the task ledger and the events that advance them. Only the first row originates in Governance; the host runtime reports the rest.

Terminal states. The two terminal states distinguish a task that settled a valid candidate from one that did not. When the executor stops, the hook consults the attempt log rather than the executor: a task with a successful attempt on record settles as `completed`. The `failed` state covers every other stop, whether the attempts all failed verification, the executor abandoned the trajectory, or it terminated early; a short failure reason, taken from the executor's final message where the host supplies it, is stored for reporting alone. Neither outcome drops a task from the ledger, and both settle it for scheduling, so the loop replenishes on either.

Scope of the ledger. The ledger counts tasks; it does not score them. Its fitness field copies the evaluator's fitness from the successful attempt and serves reporting alone. No selection, plateau, or budget decision reads it. The fitness that drives evolution arrives on a separate path. The control-plane call that records an attempt accepts no fitness argument: it reads the evaluator's own result file and takes validity and fitness from there (§4.4), and only a result the evaluator marked successful with a numeric fitness enters the population that selection ranks. A task none of whose attempts the evaluator marked successful settles as `failed`, so it is counted but never selected from. No number an agent reports therefore enters the ledger or the search.

A.7.3 IDENTITY AND BUDGET

Identity. An agent never states which task it is running. The hook resolves identity in two steps. A host session identifier resolves to a run, recorded when the orchestrator binds that run through the control plane. A host agent identifier resolves to an entry in the run's agent index, written when the subagent starts.

Identity is assigned in the opposite direction, from the hook to the executor. On observing a spawn whose subagent type is the task executor, the hook takes the next identifier from the pool and rewrites the spawn payload: it sets the spawn description to that identifier and appends the run and task identifiers to the prompt. The executor receives an identity it did not choose, and its ledger entry predates it. The host echoes the rewritten description in the post-call event, which lets the hook match the returned agent handle to its task.

Budget. Governance mints identifiers at each convergence check, up to the requested concurrency and within the run's total task budget. The count it returns tells the orchestrator how many executors to spawn. This bound needs no cooperation from the orchestrator. An orchestrator that spawns extra executors finds an empty pool, so allocation yields nothing, those spawns carry no task identity, and the budget stays intact. Minting also counts tasks already created but not yet spawned, so unclaimed identifiers suppress later minting instead of piling up.

The root barrier. The root phase uses the same minting rule as a barrier, with no separate code path. Its mint ceiling equals the number of lineage roots. Once every root has been minted the pool stays empty until roots settle, so the orchestrator receives zero new tasks and keeps looping without needing to know why. Settled roots lift the barrier, which puts a failed root on the same footing as a successful one.

A.7.4 PERMISSION CONTROL

The decision pipeline. A permission hook runs before the tool executes. It normalizes the call into a tool type with a small typed parameter set, resolves the caller through the identity path above, and runs the pipeline registered for that caller's role. Each rule declares a filter and a check, so inapplicable rules drop out before any argument inspection.

The pipeline resolves by deny-overrides: a denial short-circuits and returns its reason, and an endorsement stands only when no rule denies. A call that no rule claims falls through to the host's own permission flow. That third outcome keeps the layer additive: it only narrows what the host would otherwise permit, never widens it, so one rule set works on any host regardless of its native permission model. Denials are logged with the rule that produced them.

The rule families. The pipeline carries two rule families. The first denies file reads, globs, and greps whose target path resolves to the evaluator source. It also confines the executor's shell to an allowlist: the evaluator command that the benchmark contract names, run for the executor's own task, and the creation of its own attempt directories. Every other shell command is denied, so no shell path reaches the sources either. The second governs another task's result directory, and it works from a whitelist: it admits exactly the artifact paths Governance bound into the capsule and denies every other path under another task's directory, including its evaluation records and metrics. An executor can therefore read the parent and inspirations it was granted, and nothing else of the history. The first rule keeps judge with Protected Judgment; the second keeps govern with Structured Governance.

Two denials in practice. Figures 16 and 17 reproduce one interception by each rule family, both from Circle Packing runs. In the first, a task executor issued a `Read` whose path resolved to the benchmark's `evaluator.py`. The pre-tool hook normalized the call, resolved the caller to the task-executor role, and ran that role's pipeline; the first rule family matched the path and returned DENY with its reason, and the hook answered the host with that decision. The host dropped the call before it executed and handed the reason back to the executor as the tool result, so the executor learns that the read is prohibited and why. The task-executor prompt of Appendix D.3 states the same prohibition; this path holds when the prompt does not.

In the second, Governance had bound the executor of `task_009` a capsule with `task_007` as parent and `task_001`, `task_002`, and `task_006` as inspirations, so the executor could read those four artifact directories and its own. It then tried to read the evaluation record of `task_008`, a sibling that Governance had not selected for it. That path lies under another task's directory and is not among the artifact paths bound into the capsule, so the second rule family denied it. The reason names the whitelist in force: the task's own directory and the `artifact_ref` directories returned by `select_parent` and `select_inspirations`. What reaches a task about the rest of the history therefore remains what Governance chose to bind, not what the executor decides to look up.

The two denials guard the two boundaries that Open Proposal could otherwise erode. The first keeps judge with Protected Judgment: evaluator evidence stays private, so the official score is never shaped by a proposer that has read how it is computed. The second keeps govern with Structured Governance: which parts of the history a task builds on is a selection decision, and the executor cannot widen it on its own. In both cases the executor retains full autonomy over construction inside its capsule and none over the decisions the other two planes own.

Permission hook one denial from a Circle Packing run; paths abbreviated

PreToolUse payload (host → hook)

```
tool_name: Read
tool_input.file_path: .../benchmarks/skills/circle_packing/scripts/evaluator.py
caller: resolved from session_id and agent_id to role task-executor
```

Pipeline log

```
16:03:16 DEBUG pipeline.py[evaluate_permission] permission: no_evaluator_access -> DENY "Access to
evaluator script '.../circle_packing/scripts/evaluator.py' denied: task-executors must NOT read
the evaluator script."
16:03:16 INFO Final decision: DENY (handler: no_evaluator_access)
```

Hook response (hook → host)

```
{"hookSpecificOutput": {"hookEventName": "PreToolUse", "permissionDecision": "deny",
"permissionDecisionReason": "Access to evaluator script ... denied: task-executors must NOT read
the evaluator script."}}
```

Figure 16: One interception by the first rule family. The host drops the call before it executes and returns the stated reason to the executor as the tool result.

Permission hook one denial from a Circle Packing run; paths abbreviated

PreToolUse payload (host → hook)

```
tool_name: Read
tool_input.file_path:
.../runs/circle_packing_20260922190350/tasks/task_008/attempts/attempt_001/eval_result.json
caller: resolved from session_id and agent_id to role task-executor, bound to task_009
capsule: parent task_007; inspirations task_001, task_002, task_006
```

Pipeline log

```
19:31:52 INFO pipeline.py[evaluate_permission] Final decision: DENY, Read of
'.../tasks/task_008/attempts/attempt_001/eval_result.json' denied: under runs/ task-executor may
only read runs/circle_packing_20260922190350/tasks/task_009/ and the artifact_ref directories
returned by select_parent/select_inspirations. (handler: task_executor_read_allowlist_under_runs)
```

Hook response (hook → host)

```
{"hookSpecificOutput": {"hookEventName": "PreToolUse", "permissionDecision": "deny",
"permissionDecisionReason": "Read of '.../task_008/.../eval_result.json' denied: under runs/
task-executor may only read .../tasks/task_009/ and the artifact_ref directories returned by
select_parent/select_inspirations."}}
```

Figure 17: One interception by the second rule family. The executor of `task_009`, granted `task_007` as parent and `task_001`, `task_002`, and `task_006` as inspirations, attempted to read the evaluation record of `task_008`; the whitelist admits only its own directory and the artifact paths Governance bound into its capsule, so the host drops the call and returns the stated reason.

A.7.5 Host Portability

The ledger's state machine is shared across hosts. The scheduling adapter maps native events onto the four lifecycle roles of Table 5, and permission interception runs through the same pre-tool event; Appendix A.1 states the full set a host must supply.

Role	Claude Code	OpenCode
spawn observed	`PreToolUse` on the agent tool	`tool.execute.before` on the task tool
handle returned	`PostToolUse`	`tool.execute.after`
subagent starts	`SubagentStart`	`session.created` carrying a parent
subagent ends	`SubagentStop`	`session.idle`

Table 5: Native events mapped onto the four lifecycle roles the ledger requires, shown for the two hosts used in this paper.

Hosts differ in how directly they expose these roles. OpenCode, for example, supplies the last two only indirectly, and the adapter reconstructs them. Its subagents are ordinary sessions, so a child announces itself by carrying a parent session, and the parent identifier resolves the run. Its post-call payload also reports the arguments as they stood before the hook rewrote them, so the injected identifier never comes back through the host. The adapter therefore keeps a registry keyed by call: allocation records the run and task, and the completion of that same call retrieves them. A second registry keyed by session lets the idle event recover which task a stopping subagent held, and retiring the entry on use fires the terminal transition once. The state machine itself never changes: whatever events a host emits, the adapter's only job is to translate them into these four roles.

A.8 Judge in Existing Systems

Protected Judgment (§4.4) asks three things of a system: evaluator sources stay unreadable from govern and propose, the normalized outcome is the only channel back, and something outside the agent holds that boundary. This appendix checks the five baselines of §5.1 against these three conditions, reading the public revisions our experiments start from: SkyDiscover revision `0d932b6` (2026-09-14) for OpenEvolve, EvoX, and AdaEvolve (Liu et al., 2026b), and CORAL revision `0123dfb` (2026-09-06) (Qu et al., 2026). None of the five meets the first condition, and three of them route the sources into their own prompts. The closing paragraph states how the main results close these paths.

Controller-centric systems. In SkyDiscover the evaluator is a path passed beside the initial program, and the controller reads that path freely: no prompt forbids it, no permission rule covers it, and a plain-Python evaluator is loaded into the controller's own process. A configuration option (`inject_evaluator_context`) prepends the evaluator source to the proposer's system message; it resolves to on when no seed program is given and to off otherwise, so the seeded Math, ADRS, and Frontier-CS runs of this paper leave it off. Two of the three controllers read the source on another path that no option guards. AdaEvolve's paradigm generator, the meta level that issues breakthrough directions when the search stagnates, receives the full evaluator source in a prompt section headed "Evaluator Code (shows how solutions are scored)" and is instructed to analyze how solutions are scored before proposing; the resulting directions are inserted into the proposer's prompt. EvoX's operator generation receives the same source to derive its diverge and refine guidance, and its strategy-evolution prompt asks the model to trace how the combined score is computed; the derived guidance is rendered into every proposer prompt. In both systems govern reads judge's sources as a matter of design and forwards what it learns to propose. OpenEvolve reads nothing on its own, and nothing stops it.

Agent-centric systems. CORAL's paper lists evaluator isolation among its execution safeguards, with grader code held in a private directory that agents cannot inspect or modify (Qu et al., 2026). In the released code the protection is narrower than that description. The private directory holds the grader environment and any answer keys, while the grader source itself is surfaced into every agent worktree with read permission, and the instruction file rendered for each agent points to it as the exact code that scores the agent and invites the agent to read it. In the default host session the private directory is guarded by a read rule on one tool while shell access stays unrestricted; the optional container and sandbox sessions mount the grader source read-only. CORAL's isolation therefore protects the integrity of the grader and the secrecy of answer keys, and leaves the scoring logic readable in every session type.

What the baselines share. In each of the five systems the boundary around judge is a component boundary. A separate evaluator exists, nothing stops govern or propose from reading it, and three systems route it into their own prompts. Protected Judgment differs in what it protects and in who holds the line. It protects the evidence, so the scoring logic itself stays out of reach, and it enforces the boundary outside the agent through the permission hooks of Appendix A.7. §6.3 measures what the difference is worth: with the sources readable, the Original score rises by 0.093 on average, while the out-of-distribution gain shrinks to 0.009 and turns into a loss for CORAL-4 and EvoTrias.

Protocol for the main results. The scores of §5.2 are produced with every one of these paths closed, so the comparison turns on search design and not on evaluator access. For OpenEvolve, EvoX, and AdaEvolve we turn evaluator injection off for the proposer and remove the evaluator source from the meta-level prompts of AdaEvolve and EvoX. For CORAL we add a strict prompt-level prohibition on reading evaluator sources to the agent instruction file and extend the workspace guard described in its paper into a hook that denies agent operations on the evaluator, including read, write, and grep. The exposed setting of §6.3 is the only configuration in which any system can reach the sources (Appendix C.10.1).

A.9 Discussion: What the Boundaries Cost and Buy

The main text follows one line: which plane owns each decision. This section follows three consequences of drawing those boundaries.

Bounded context. An agent-centric proposer reads the whole shared memory: every scored attempt, every lesson, the full history of the run. This access helps in one measurable way. On Frontier-CS, CORAL-4 leaves fewer near-zero scores than EvoTrias (§5.2), which the richer cross-trajectory context plausibly helps to repair. The same access carries the cognitive cost of §1: one attempt's conclusion reaches every agent unchecked, and the search narrows before the evidence justifies it. It also carries a computational cost that grows with the run, since each agent spends more of its budget re-reading a longer history (§6.2). Open Proposal in EvoTrias never sees the raw history. Structured Governance distills settled outcomes into posterior evidence and constructs the capsule from it: a parent and inspirations selected by the current policy, Notes that were validated on admission and loaded by relevance, and Meta-guidance only when the search stalls (§4.2). What reaches a task has been checked and scoped before it arrives. The results bear out this trade. EvoTrias trails CORAL-4 only in repairing zero-scoring attempts, while it ranks first on nine of the twelve Math and ADRS tasks under each backbone, leads Frontier-CS in both mean and median (§5.2), and does so at 64% lower token cost (§6.2). A task receives a bounded context, and the separation of govern from propose that bounds it returns far more than it costs. The Level 1 ablations of Table 2 then measure how much each component of the capsule matters: removing any one of them lowers the score on both tasks, so the information Governance selects is the information Proposal needs. The converse holds as well: handing the controller-centric baselines an agentic proposer without changing their govern leaves every one of them below EvoTrias (Appendix C.5).

Parallelism. Concurrency is where much of EvoTrias's efficiency comes from. The wall-clock advantage in §6.2 comes mainly from parallel task execution, and §6.4 shows that $c = 4$ cuts wall-clock time by 74% at a fitness loss under 0.009, a near-linear gain in end-to-end efficiency. How a design adds concurrency therefore matters. In an agent-centric system, govern lives inside each agent's trajectory, so adding a parallel proposer means replicating the whole agent, and every replica re-reads history and plans its own moves. In EvoTrias, govern and propose are separate: cross-round state lives in Governance, scheduling in the hook-written ledger (Appendix A.7), and a task executor needs only its capsule. Adding a slot therefore means spawning one more executor subagent that constructs one candidate and terminates (§4.3); nothing else in the run changes. The two ways of scaling differ sharply in cost. CORAL-4 spends 3.9× the tokens of CORAL-1 under the same wall-clock budget (Figure 6b), whereas EvoTrias's token cost varies by less than 5% from $c = 1$ to $c = 8$. At the same wall-clock budget and the same four-way width, EvoTrias spends 36% of CORAL-4's tokens (§6.2). What parallelism does cost is evidence lag: a task binds its context before its siblings settle, and Appendix A.6 bounds that lag at $c - 1$ outcomes. Concurrency therefore follows from the allocation itself.

Complementarity. The division brings one more benefit: each plane can improve along its own line without touching the others. Proposal improves with stronger hosts and backbones (Table 2, Level 2) and deploys as a plugin on any host that meets the contracts (Appendix A.1). Governance improves with better search algorithms: the adaptive controllers of §2 are candidate replacements for G, and the default policy Π_0 already adapts population-based rules from OpenEvolve and AdaEvolve (§4.2). Judgment improves with better evaluators behind the same request interface.

A.10 Limitations

Two limitations bound the present results. (1) Open Proposal runs on a general-purpose coding agent, so construction quality tracks the backbone: EvoTrias scores 0.7329 with GLM-5.2 and 0.8313 with Opus-5 on Signal Processing, and every result is reported per backbone (Appendix C.2). (2) Coding-agent trajectories cost tokens. A full 50-iteration run averages $41.37, which is 64% below four-agent CORAL and still above the three controller-centric baselines (§6.2); construction autonomy is what that margin buys.

B Experimental Setup Details

B.1 Evaluation Protocol

Every method scores its candidates through the same evaluator for each task, so a reported fitness means the same thing across systems. The evaluator accepts an artifact and returns the normalized outcome of Equation 1. The protected setting of §4.4 is the default for the main results, so no method may read evaluator sources; the exposed setting of §6.3 is the deliberate exception, and it is the only configuration in which any method can read them.

The search itself optimizes one scalar fitness per task, while Tables 1 and 10 report each task in the unit its own benchmark states. The two coincide on most tasks and differ on those whose evaluator normalizes against a published reference value or inverts a cost. Table 6 gives the conversion, so the internal fitness values quoted in the case studies of Appendix C can be read against the main tables.

Table 6: Reported quantity and internal fitness for each task. The arrow marks the direction of the reported quantity R; the internal fitness f is always larger-is-better. The constants are the published reference values the evaluator normalizes against.

Task	Reported quantity R	Internal fitness f
Math		
Circle Packing (square)	sum of 26 radii ↑	$R/2.635$
Circle Packing (rectangle)	sum of 21 radii ↑	$R/2.365832$
Heilbronn Triangles	normalized minimum triangle area ↑	$R/0.0365299$
Heilbronn Convex	normalized minimum triangle area ↑	$R/0.0309369$
MinMax Distance	squared min-to-max distance ratio ↑	$R/0.2400471$
Signal Processing	composite filtering score ↑	R
Third Autocorrelation	third autocorrelation constant ↓	$1.4556428/R$
ADRS		
Cloudcast	total transfer cost ↓	$1/(1+R)$
EPLB	expert-balance and speed score ↑	R
LLM-SQL	cache-hit and runtime score ↑	R
Prism	inverse KV-cache pressure score ↑	R
TXN	inverse-makespan schedule score ↑	R
Frontier-CS		
All 30 problems	judge score on a 0–100 scale ↑	R

B.2 Budget Alignment Across Baselines

For the controller-centric baselines, we match the outer search budget exactly: EvoTrias, OpenEvolve, EvoX, and AdaEvolve each receive 50 proposal slots. A slot can add at most one

valid candidate. After the first attempt, generation or evaluation failure can trigger up to two repair-and-re-evaluation retries, but every method stops at the first valid result; retries are never used to improve an already valid score. These rules match the 50 opportunities that can add evolved candidates to the search state; retries only recover invalid executions.

Table 7 verifies that this rule holds in the runs behind the main results by counting the evaluator calls each method actually made. Every method lands within a narrow band above the 50-slot floor, the excess being the retries that recovered invalid executions, and EvoTrias sits at or below the baselines' band on every suite. A proposal slot therefore buys the same number of official evaluations for every iteration-based method; what differs is only how the candidate inside that slot is constructed, which is the design decision under test.

Table 7: Mean evaluator calls per run under the shared 50-slot budget (GLM-5.2; three runs per Math and ADRS task, one run per Frontier-CS problem). Each slot admits at most one valid candidate and at most two repair retries, so a run makes between 50 and 150 calls; the realized means show that retries are rare and comparable across methods.

Suite	OpenEvolve	EvoX	AdaEvolve	EvoTrias
Math ($n = 21$ runs)	55.47	57.81	58.76	55.81
ADRS ($n = 15$ runs)	57.60	57.40	54.53	52.87
Frontier-CS ($n = 30$ runs)	61.13	56.37	55.13	56.80

CORAL has no outer proposal loop. For each task and backbone, we therefore match CORAL-1 and CORAL-4 to EvoTrias's mean wall-clock runtime under the 50-slot budget. Table 8 reports these shared limits.

Table 8: Per-task wall-clock budgets shared by EvoTrias, CORAL-1, and CORAL-4.

Problem	**GLM-5.2 (h)**	**Opus-5 (h)**
Math		
Circle Packing Square	1.87	2.67
Circle Packing Rectangle	1.84	2.11
Heilbronn Triangles	1.81	1.90
Heilbronn Convex	2.54	3.16
MinMax Distance	3.04	3.09
Signal Processing	2.18	3.01
Third Autocorrelation	2.70	2.59
ADRS		
Cloudcast	1.94	2.88
EPLB	2.37	2.39
LLM-SQL	3.18	3.75
Prism	3.39	3.90
TXN	2.03	2.48
Frontier-CS		
Average ($n = 30$)	4.81	–

The two CORAL configurations bracket the concurrency comparison. CORAL-1 uses one top-level trajectory, whereas EvoTrias can keep $c = 4$ bounded task executors in flight, so proposal width favors EvoTrias in that comparison. CORAL-4 removes this asymmetry and makes the resource allocation conservative for EvoTrias: it runs four complete top-level agents in parallel and returns their best final score, while EvoTrias retains one top-level orchestrator and implements four-way concurrency only through its task-executor subagents. Under the same wall-clock limit and nominal four-way width, CORAL-4 consumed about $2.8\times$ as many tokens (Figure 6b). This ratio is the realized cost of the matched protocol, not an intrinsic constant of CORAL; it shows that CORAL-4 received the larger aggregate agent-compute allocation.

What CORAL-4 is a comparison against. The asymmetry deserves to be stated plainly, because it decides how the paired statistics of Appendix C.1 and C.3 should be read. CORAL-4 is not one evolution system with four workers. It is four complete evolution systems, each a full top-level CORAL agent with its own trajectory, its own read and write access to the shared memory, and its own freedom to spawn subagents, run for the entire wall-clock budget, after which the best of the four final scores is reported. That is a best-of-four ensemble over independent runs, and it enjoys two advantages that no single system in the comparison has: four times the top-level agent compute, and a final max over four outcomes that suppresses the run-to-run variance every other entry carries in full. EvoTrias, in contrast, is one system: one orchestrator, one Governance state, one budget of 50 proposal slots, and a single final score with no selection over repeated runs. Its subagents are not four agents; they are bounded task executors that construct one candidate each inside a capsule and terminate.

Under this protocol, any lead over CORAL-4 is a lead over a four-way ensemble at $2.8\times$ the token cost, and any tie is a tie against it. EvoTrias leads CORAL-4 on 9 of 12 Math and ADRS tasks with no ties, and achieves a geometric-mean gain of 0.640% with a positive confidence interval (Table 11); on Frontier-CS it raises the mean by 3.49 points and the median by 13.53 (Tables 12 and 13). The exact tests against CORAL-4 are the only ones in the paper that do not reach $p < 0.05$, and that is what one should expect when a single system is measured against a best-of-four ensemble of a strong competitor: the protocol was chosen to stack the comparison against EvoTrias, not to produce a significant margin. The result to take from these tables is that one EvoTrias run matches or beats the best of four CORAL runs while spending 36% of their combined tokens. A comparison at equal top-level agent count is CORAL-1, and there EvoTrias wins all 12 Math and ADRS tasks and 15 of 30 Frontier-CS problems with 2 losses, with every confidence interval positive.

B.3 Hyperparameter Configuration

Table 9 lists EvoTrias's default operating settings, grouped by module; controlled ablations override only the setting named in their description. Every value was fixed before the benchmark runs and is shared by all runs in this paper; no constant was tuned per task or per benchmark. A dash indicates that the setting has no separate mathematical symbol. Annealed parameters are written as start $\rightarrow$ end over the task budget.

Table 9: Default EvoTrias operating settings, grouped by module.

Parameter	Symbol	Value
Scheduling and budget		
Initial exploration directions (roots)	n_{dir}	4
Concurrent task slots	c	4
Task budget (iterations)	T	50
Lineage and parent selection		
Lineage temperature (annealed)	τ_t	$1.5 \to 0.5$
Exploit probability	p_{exploit}	0.7
Quality bonus strength	η	0.3
Local exploration (annealed)	$c_{\mathrm{local},t}$	$1.5 \to 0.5$
Elite rank fraction	κ	0.15
Elite selection shield	s	0.6
Potential prior	α, β	$(1, 3)$
Lineage potential weight	λ_L	0.10
Parent potential weight	λ_P	0.03
Inspiration selection		
Maximum inspiration slots	$\lvert I_t \rvert$	3
Cross-lineage rate	μ	0.2
Foreign elite pool per lineage	n_e	2
Sampling temperature (annealed)	T_t	$1.5 \to 0.5$
Rebalance entropy threshold	$H^\star$	0.9
Rebalance entropy floor	$H_{\min}$	0.5
Rebalance max strength	$\gamma_{\max}$	2.0
Notes		
Delta quantile (annealed)	q	$0.80 \to 0.60$
Relative-delta threshold range	τ_t^+, τ_t^-	$[2\%, 5\%]$
Near-best gate (annealed)	r_t	$0.80 \to 1.00$
Meta-guidance		
Plateau window	w	5
Plateau patience	P_{stall}	4
Maximum directions per request	–	3

B.4 Baseline Configurations

OpenEvolve, EvoX, and AdaEvolve run from the reference implementations and default hyperparameters of the SkyDiscover framework (Liu et al., 2026b), with one change: we set the island count to four, so that every controller-centric baseline opens with the same number of independent search branches as EvoTrias's $n_{\mathrm{dir}} = 4$ initial directions (Appendix A.3). CORAL-1 and CORAL-4 keep their published agent design and run on the same coding-agent host as EvoTrias, Claude Code v2.1.201, so the comparison between the two agent-centric configurations and EvoTrias turns on the architecture rather than on the agent; Appendix B.2 details their budget and concurrency controls. The host ablations of §5.3 replace that host with OpenCode v1.18.25 and change nothing else. Every method receives the same task statement, the same initial reference program, and the same evaluator (Appendix B.1), and none of them may read evaluator sources outside the exposed setting of §6.3.

C Additional Results and Analysis

C.1 Math and ADRS: Statistical Results

Table 10 adds the best run and the sample standard deviation to Table 1 for the five strongest methods; OpenEvolve is omitted, since it places last on almost every task of Table 1 and its spread adds little to the comparison. Figure 18 rescales its Std rows into the coefficient of variation, so that tasks orders of magnitude apart share one axis; the 20 of the 120 task–method pairs whose three runs returned an identical score have zero Std and sit in the lane at the left. Under GLM-5.2, EvoTrias is the most stable of the five methods, with median 1.8×10^{-3} against 3.8×10^{-3} for the next. Under

Opus-5 the comparison loses its grip: all three runs return the same score in 19 of that backbone's 60 pairs, leaving too little spread to rank the methods by.

Table 10: Mean, best-run, and sample standard deviation (Std) on Math and ADRS; OpenEvolve omitted, and lower is better for Third Autocorrelation and Cloudcast (↓). First and second Mean and Best results within each backbone are highlighted as first and second.

		GLM-5.2					Opus-5				
Task	Stat	EvoX	AdaE.	CORAL-1	CORAL-4	EvoTrias	EvoX	AdaE.	CORAL-1	CORAL-4	EvoTrias
Math											
Circle-Pack. (Sq.)	Mean	2.60956	2.63152	2.62337	2.63418	2.63484	2.63598	2.63598	2.63598	2.63598	2.63599
	Best	2.61094	2.63429	2.63135	2.63598	2.63598	2.63598	2.63598	2.63598	2.63598	2.63599
	Std	0.00123	0.00240	0.00794	0.00309	0.00193	1.72e-9	1.52e-9	0	4.57e-9	6.75e-6
Circle-Pack. (Rect.)	Mean	2.34876	2.36487	2.32097	2.36459	2.36493	2.36583	2.36584	2.36583	2.36583	2.36584
	Best	2.35825	2.36575	2.36425	2.36583	2.36583	2.36583	2.36584	2.36583	2.36583	2.36584
	Std	0.0102	0.00152	0.0613	0.00214	0.00157	0	5.24e-6	0	0	5.11e-6
Heilbronn-Tri.	Mean	0.0302	0.0318	0.0320	0.0332	0.0335	0.0361	0.0364	0.0364	0.0365	0.0365
	Best	0.0343	0.0331	0.0361	0.0343	0.0360	0.0365	0.0365	0.0365	0.0365	0.0365
	Std	0.00395	0.00129	0.00427	0.00105	0.00223	0.000438	0.000287	0.000171	0	0
Heilbronn-Conv.	Mean	0.0269	0.0275	0.0284	0.0283	0.0302	0.0307	0.0304	0.0309	0.0309	0.0309
	Best	0.03043	0.02777	0.02870	0.02864	0.03085	0.03094	0.03086	0.03094	0.03094	0.03094
	Std	0.00246	0.000451	0.000439	0.000447	0.000541	0.000445	0.000401	0	0	0
MinMax-Dist.	Mean	0.23970	0.23987	0.23975	0.24003	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005
	Best	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005	0.24005
	Std	0.000448	0.000211	0.000514	4.49e-5	8.92e-8	0	0	0	0	0
Signal-Proc.	Mean	0.7098	0.6961	0.6638	0.7083	0.7329	0.7861	0.7976	0.8251	0.8280	0.8313
	Best	0.7173	0.7214	0.6669	0.7149	0.7370	0.8079	0.8140	0.8264	0.8294	0.8315
	Std	0.00951	0.0226	0.00515	0.00573	0.00583	0.0319	0.0183	0.00118	0.00117	0.000224
Third-Autocorr. ↓	Mean	1.4601	1.4614	1.4814	1.4608	1.4609	1.4587	1.4568	1.4572	1.4559	1.4563
	Best	1.4594	1.4590	1.4593	1.4574	1.4563	1.4571	1.4560	1.4556	1.4548	1.4560
	Std	0.000681	0.00247	0.0310	0.00318	0.00404	0.00145	0.000655	0.00160	0.00127	0.000549
ADRS											
Cloudcast ↓	Mean	646.3036	631.1295	622.5701	618.0869	618.0671	618.0603	618.0707	618.0657	618.0684	618.0648
	Best	627.1164	627.1193	618.0928	618.0648	618.0615	618.0596	618.0699	618.0616	618.0596	618.0596
	Std	17.30	6.939	4.454	0.0198	0.00485	0.000820	0.00126	0.00651	0.00791	0.00890
EPLB	Mean	0.1328	0.143327	0.1423	0.1437	0.143332	0.1437	0.1439	0.1445	0.1450	0.1448
	Best	0.1426	0.143403	0.1426	0.1442	0.143415	0.1439	0.1443	0.1446	0.1452	0.1449
	Std	0.00851	6.60e-5	0.000359	0.000777	0.000119	0.000203	0.000509	0.000185	0.000153	0.000216
LLM-SQL	Mean	0.6794	0.7148	0.6742	0.6890	0.7206	0.7229	0.7232	0.7240	0.7318	0.7398
	Best	0.7110	0.7228	0.7094	0.6934	0.7276	0.7245	0.7329	0.7245	0.7365	0.7437
	Std	0.0279	0.0111	0.0372	0.00617	0.00814	0.00145	0.00920	0.000624	0.00636	0.00386
Prism	Mean	26.246	25.359	26.252	26.248	26.256	26.256	26.256	26.256	26.256	26.256
	Best	26.256	26.233	26.256	26.256	26.256	26.256	26.256	26.256	26.256	26.256
	Std	0.0118	1.469	0.00352	0.0131	0	0	0	0	0	0
TXN	Mean	3,843	3,928	4,078	4,280	4,203	4,281	4,292	4,498	4,532	4,573
	Best	3,906	4,016	4,386	4,329	4,292	4,405	4,310	4,566	4,587	4,587
	Std	96.89	76.90	373.0	68.76	73.54	215.4	31.77	58.85	62.48	24.08

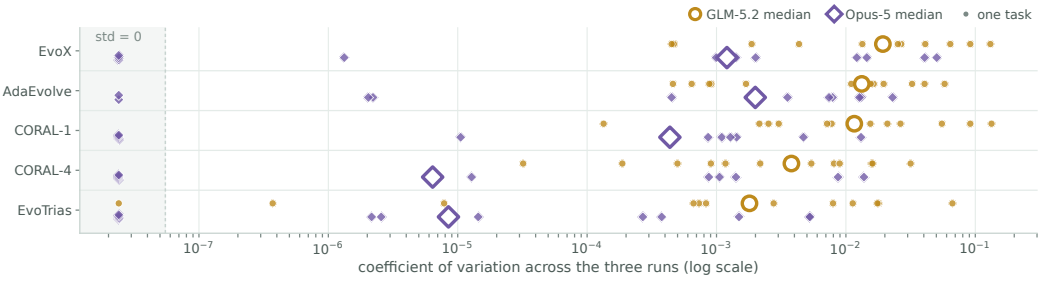


Figure 18: Run-to-run spread behind the Std rows of Table 10, as the coefficient of variation. Small marks are the 12 tasks, large open marks their median.

Paired task-level analysis. We complement the raw-score table with a scale-invariant comparison across all 12 tasks. For task t, backbone b, and baseline m, let $e_{tbm} = d_t \log(\bar{s}_{tb,\mathrm{EvoTrias}}/\bar{s}_{tbm})$, where $d_t = 1$ for higher-is-better tasks and $d_t = -1$ for Third Autocorrelation and Cloudcast. We average e_{tbm} over the two backbones within each task and report $100[\exp(\mathrm{mean}_t\, e_{tm}) - 1]$ as the

geometric-mean relative gain. The 95% confidence intervals resample the 12 paired tasks 200,000 times. We also apply an exact two-sided sign-flip test to the task effects and use Holm correction across the five baselines; the confidence intervals are per-comparison intervals.

Table 11: Paired task-level comparison across Math and ADRS. Positive gains favor EvoTrias. W/T/L counts the signs of the 12 backbone-averaged task effects.

Baseline	Rel. gain	95% CI	W/T/L	Exact p	Holm p
OpenEvolve	+6.403%	$[+2.926, +10.270]\%$	12/0/0	0.000488	0.00244
EvoX	+3.010%	$[+1.502, +4.596]\%$	12/0/0	0.000488	0.00244
AdaEvolve	+2.047%	$[+0.845, +3.422]\%$	12/0/0	0.000488	0.00244
CORAL-1	+1.704%	$[+0.784, +2.756]\%$	12/0/0	0.000488	0.00244
CORAL-4	+0.640%	$[+0.010, +1.382]\%$	9/0/3	0.1646	0.1646

EvoTrias improves on OpenEvolve, EvoX, AdaEvolve, and CORAL-1 on all 12 tasks. The gains range from 1.704% to 6.403%, every confidence interval is positive, and all four comparisons remain significant after Holm correction ($p = 0.00244$). EvoTrias also leads CORAL-4 in aggregate by 0.640% and on 9 of 12 tasks. This is a strong efficiency result: CORAL-4 selects the best outcome from four parallel top-level agents, while EvoTrias uses one top-level agent and coordinates native subagents within it.

C.2 Where Backbone Strength Shows Up

Table 10 shows that the two backbones separate on some tasks and not others. Comparing final scores alone cannot say why. This subsection follows one GLM-5.2 run and one Opus-5 run through the budget on two tasks of differing scope, using the split that §5.3 draws: Circle Packing targets one fixed 26-circle arrangement, while Signal Processing optimizes a filtering algorithm over a family of signals. Fitness in this subsection is the internal score of Table 6; on Circle Packing the internal 1.000373 is 2.63598 on the raw scale of Table 1.

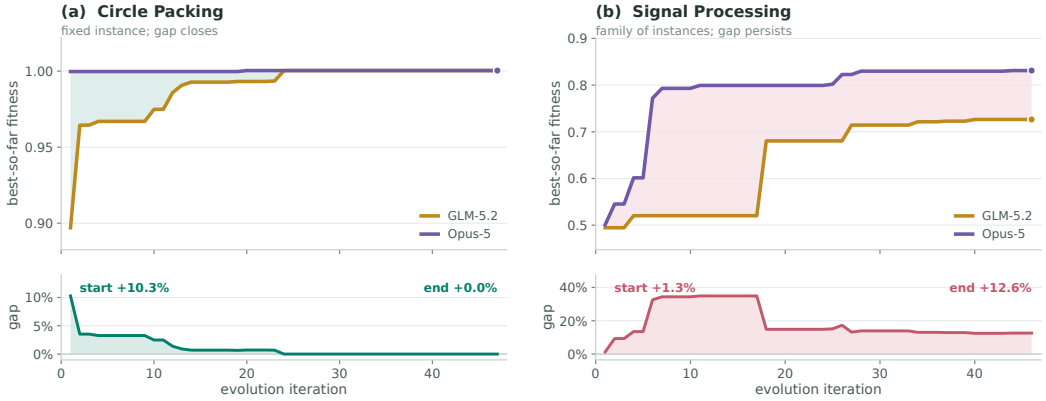
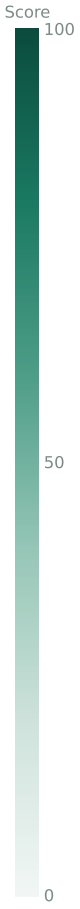


Figure 19: Best-so-far fitness for one GLM-5.2 run and one Opus-5 run on each task, with the gap between them shaded and repeated below each panel as a share of the leading run's score at that iteration.

Two opposite regimes. Throughout this subsection a gap is reported as the difference between the two runs divided by the leading run's score at that iteration. On Circle Packing the stronger backbone opens with a large lead: its first settled candidate already scores 0.9997 against 0.897, a gap of 10.3%. Evolution then removes that lead. The gap falls below 1% by iteration 18 and reaches zero by iteration 24, and both runs finish on the same value of 1.000373. On Signal Processing the two runs instead start level, 0.4945 against 0.5010, a gap of 1.3%. Here the gap is opened by the search rather than inherited from it: it widens to 34.9% by iteration 12, and although GLM-5.2 recovers part of it later, 12.6% still remains at the end.

Table 12: Per-problem Frontier-CS scores (GLM-5.2). Cell shading follows the shared 0–100 scale shown at right.

Problem ID	OpenEvolve	EvoX	AdaEvolve	CORAL-1	CORAL-4	EvoTrias
207	67.42	0.00	0.00	67.90	45.32	57.13
209	0.00	0.00	0.00	0.00	10.99	0.00
210	34.67	46.72	28.35	24.30	38.04	27.98
211	86.95	87.10	85.68	61.58	85.60	85.89
212	0.00	0.00	0.00	25.00	25.00	25.00
213	84.25	33.33	84.25	55.09	96.23	97.26
214	33.94	64.35	61.01	78.66	73.74	82.74
217	64.11	63.14	31.70	33.08	33.33	68.54
220	70.00	100.00	70.00	100.00	100.00	100.00
222	15.52	15.41	15.52	15.37	57.91	78.59
225	0.00	0.00	0.00	0.00	0.00	0.00
227	2.45	2.45	2.45	2.45	2.46	2.45
228	100.00	100.00	100.00	100.00	100.00	100.00
229	100.00	99.67	100.00	100.00	100.00	99.00
231	0.00	0.00	0.00	0.00	0.00	0.00
233	33.20	29.11	61.25	30.20	31.13	59.11
239	54.56	63.39	69.89	74.67	60.54	100.00
241	93.81	96.67	94.28	86.67	93.81	93.33
243	75.00	100.00	75.00	75.00	75.00	75.00
245	89.17	89.96	89.96	89.17	100.00	89.64
247	2.10	12.07	0.00	0.00	9.91	0.00
248	100.00	100.00	100.00	100.00	100.00	100.00
249	100.00	100.00	100.00	100.00	100.00	100.00
252	72.67	80.64	81.37	81.82	100.00	100.00
253	100.00	100.00	100.00	100.00	100.00	100.00
254	25.91	34.68	25.91	30.49	36.23	34.06
255	42.70	45.40	45.56	43.92	47.64	49.26
256	39.44	44.80	44.80	44.59	44.80	44.80
257	100.00	100.00	100.00	100.00	100.00	100.00
258	91.72	90.91	91.69	87.75	90.42	92.87
Mean	55.99	56.66	55.29	56.92	61.94	65.42 +3.49
Median	65.77	63.27	65.57	64.74	67.14	80.67 +13.53



Why scope decides this. Circle Packing has a reachable optimum, and both backbones arrive within the budget, so backbone strength buys a better starting point and nothing more; the search absorbs the difference. Signal Processing offers no such ceiling within 50 iterations, so a run keeps finding headroom, and how much of it the run captures tracks the backbone. Backbone strength therefore acts on where a run begins in the first case, and on how far it climbs in the second.

Corroboration. One run per cell cannot separate these trajectories from run-to-run noise, so we read them against Table 10. Over three runs each, EvoTrias scores 0.7329 with GLM-5.2 and 0.8313 with Opus-5 on Signal Processing, an 11.8% gap, against 2.63484 and 2.63599 on Circle Packing, a gap of 0.043%. Both figures agree with the single-run trajectories that the backbone gap is large on Signal Processing and negligible on Circle Packing.

C.3 Frontier-CS: Statistical Results

Table 12 lists the score of every method on each of the 30 Frontier-CS problems; cell shading follows the score, from pale teal (low) to deep teal (high). The corresponding distribution-level comparison appears in Figures 3 and 4.

Paired problem-level analysis. Unlike Math and ADRS, all Frontier-CS outcomes share the same 0–100 scale and each method has one run per problem. We therefore use the raw paired difference on each of the 30 problems. Mean-difference confidence intervals resample complete problem rows 200,000 times, preserving all method scores in each pair. The reported p-values come from two-sided paired label swaps with 1,000,000 samples and are Holm-corrected across the five baselines.

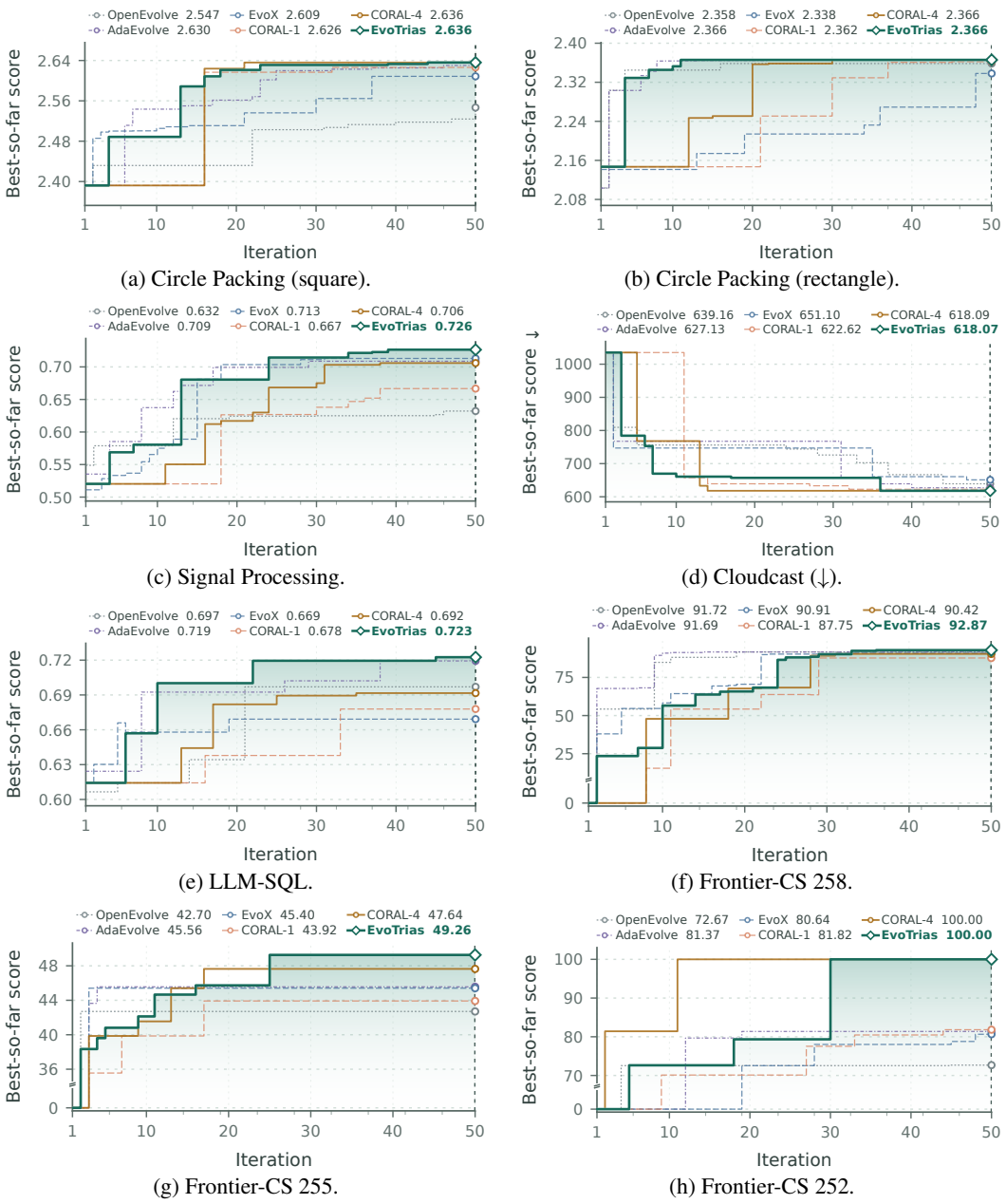
Table 13: Paired Frontier-CS comparisons. Positive score differences favor EvoTrias; W/T/L counts the 30 problem-level outcomes.

Baseline	Mean Δ	95% CI	W/T/L	Paired p	Holm p
OpenEvolve	+9.44	$[+3.65, +16.10]$	14/10/6	0.00348	0.01045
EvoX	+8.76	$[+1.58, +16.86]$	11/11/8	0.03377	0.06755
AdaEvolve	+10.13	$[+4.44, +16.79]$	13/12/5	0.000866	0.00433
CORAL-1	+8.50	$[+3.34, +14.57]$	15/13/2	0.00247	0.00989
CORAL-4	+3.49	$[-0.44, +8.06]$	10/12/8	0.1310	0.1310

The paired analysis confirms EvoTrias's distribution-level lead. Its mean gain is 8.50–10.13 points over OpenEvolve, AdaEvolve, and CORAL-1, with positive confidence intervals and Holm-corrected $p \leq 0.01045$. The gain over EvoX is 8.76 points with a positive confidence interval. Against CORAL-4, EvoTrias raises the mean by 3.49 points and the median by 13.53 points, despite using one top-level agent rather than CORAL-4's best-of-four top-level-agent selection. Together with the 65.42 mean and 80.67 median in Figure 3, these results show that governed subagent parallelism converts the available agent compute into stronger high-scoring solutions.

C.4 Best-so-Far Optimization Trajectories

Figure 20 plots best-so-far fitness over 50 evolution iterations for eight tasks. For the Math and ADRS panels we select the run with the median final score among the three; the Frontier-CS panels have a single run per problem (§5.1). Because CORAL has no outer iteration counter, its observations are aligned to the EvoTrias iteration with matching elapsed time.



(a) Circle Packing (square).
(b) Circle Packing (rectangle).
(c) Signal Processing.
(d) Cloudcast (↓).
(e) LLM-SQL.
(f) Frontier-CS 258.
(g) Frontier-CS 255.
(h) Frontier-CS 252.

Figure 20: Best-so-far fitness over 50 evolution iterations. The Math and ADRS panels show the median-final-score run among three; the Frontier-CS panels show the single run of that problem. Higher is better except for Cloudcast. CORAL observations are aligned by elapsed time to EvoTrias iterations. The Frontier-CS panels preserve zero-score prefixes while compressing the unused score interval.

C.5 Controller-Centric Baselines with an Agentic Proposer

The controller-centric baselines of §5.2 propose with a single model call. This subsection asks how much of their gap to EvoTrias closes once they, too, receive an agentic proposer. SkyDiscover ships such a mode: its `--agentic` flag replaces the single call with a multi-turn tool-calling loop that reads and searches files before it outputs the candidate program. We enable this flag for OpenEvolve, AdaEvolve, and EvoX and change nothing else. The controller, the task statement, the seed program, the evaluation service, and the 50-iteration budget are those of §5.1, so the two configurations of each

Table 14: Controller-centric baselines with SkyDiscover's built-in agentic proposer (GLM-5.2, mean ± sample std over three runs). Each group lists the published single-call configuration, the same controller with `--agentic` enabled, the relative change this switch produces, and the gap that remains to EvoTrias. Green/red cells mark gains/losses relative to the single-call configuration; saturation encodes magnitude.

Method	Circle Packing (Square)	Signal Processing	Heilbronn Triangles
EvoTrias (reference)	**2.6348**$\pm$**0.0019**	**0.7329**$\pm$**0.0058**	**0.0335**$\pm$**0.0022**
OpenEvolve	2.5432$\pm$0.0030	0.6373$\pm$0.0131	0.0253$\pm$0.0012
+ agentic proposer	2.6178$\pm$0.0267	0.6804$\pm$0.0140	0.0304$\pm$0.0019
Δ vs. single-call	+2.93%	+6.76%	+20.12%
Δ vs. EvoTrias	*−0.65%*	*−7.17%*	*−9.06%*
AdaEvolve	2.6315$\pm$0.0024	0.6961$\pm$0.0226	0.0318$\pm$0.0013
+ agentic proposer	2.6150$\pm$0.0144	0.7096$\pm$0.0353	0.0322$\pm$0.0010
Δ vs. single-call	−0.63%	+1.93%	+1.10%
Δ vs. EvoTrias	*−0.75%*	*−3.19%*	*−3.82%*
EvoX	2.6096$\pm$0.0012	0.7098$\pm$0.0095	0.0302$\pm$0.0040
+ agentic proposer	2.6118$\pm$0.0148	0.7012$\pm$0.0112	0.0302$\pm$0.0016
Δ vs. single-call	+0.08%	−1.22%	+0.01%
Δ vs. EvoTrias	*−0.88%*	*−4.33%*	*−9.67%*

system differ only in how propose is realized. Table 14 reports three runs per cell with GLM-5.2 on the two ablation tasks of Table 2 and one further Math task, Heilbronn Triangles.

Two patterns follow. The first is that the gain from the agentic proposer depends on the controller behind it. OpenEvolve improves on every task, by 2.9% on Circle Packing, 6.8% on Signal Processing, and 20.1% on Heilbronn Triangles. For AdaEvolve and EvoX the same switch moves the score by between -1.2% and $+1.9\%$, with no consistent sign. OpenEvolve leaves headroom that a proposer able to inspect the seed program and its context can take on its own. The two adaptive controllers already extract what a single call can offer, and the proposer's added autonomy has nowhere to go: the controller receives one fitness value per iteration, so what the agent learned while constructing a candidate never reaches the selection policy, and a stalled search can only be answered with a different context, not with a direction. Autonomy granted to propose without a govern designed to receive it is spent at the boundary.

The second pattern is that no agentic configuration reaches EvoTrias. The remaining gap ranges from 0.65% to 9.67%. On each task the three agentic variants also end close to one another: 2.612–2.618 on Circle Packing, 0.680–0.710 on Signal Processing, and 0.030–0.032 on Heilbronn Triangles. This ceiling matches the Level 1 ablations of Table 2 from the other direction. Take Signal Processing. Adding an agentic proposer lifts the controller-centric baselines to at most 0.710. Removing Governance components from EvoTrias lowers it into the same range: 0.6926 with the default policy Π_0 in place of evidence-conditioned adaptation, and 0.7041 without Meta-guidance. The full system scores 0.7329. A stronger proposer alone stops near 0.71, and only Structured Governance carries the score from there to 0.7329.

The agentic mode we enable is the one SkyDiscover provides, and it leaves the controller's role unchanged: the controller decides what the proposer sees and takes back a single score. The proposer's autonomy is limited on both sides of that exchange. EvoTrias runs a general-purpose coding agent on the same three tasks because its contracts and hooks (§4.3, Appendix A.7) were designed for a proposer of that kind.

C.6 Cross-Run Prevalence of the Two Experience Channels

Appendix Figures 11 and 14 each replay a single run. A case study shows that a channel *can* fire, but it leaves open whether the channel is a standing part of the search or an artifact of one favorable trajectory. This subsection audits the GLM-5.2 runs behind the main results instead: every Math

and ADRS benchmark with three runs each, together with all 30 Frontier-CS problems with one run each, for 66 runs under the 50-iteration budget of §5.1. For each run we report how often each channel is used and how the loaded notes relate to the task that consumes them.

What the panels show. Figure 21 reports the audit in three panels. Panel (a) plots the per-run count of each note type and of guidance requests, one dot per run, with the interquartile band and the mean. A run emits 3.2 positive, 5.5 negative, and 1.9 error notes on average, and issues 2.3 guidance requests. Appendix A.4 defines the anchor hit rate, the share of loaded positive and negative notes whose provenance intersects the anchor set of the consuming task rather than arriving through the seeded random draw. Panel (b) sorts by that rate the 56 runs that loaded positive or negative notes, and marks the value pooled over all loading events. Panel (c) gives the per-run ledger: one row per run, grouped by benchmark suite, with the note composition, the guidance requests, and the anchor hit rate of that run. The ledger is the panel that separates a prevalent channel from a well-chosen example, since a channel that only worked once would leave most rows empty.

Findings. Of the 66 runs, 65 emit at least one note and 60 issue at least one guidance request. Across the audit, the runs record 697 notes and 149 guidance requests. Note retrieval stays tied to the consuming context rather than degenerating into a random draw over the pool: 70.7% of loaded positive and negative notes are anchor-matched. Ten runs have no defined anchor hit rate because no positive or negative note-loading denominator is available, and are marked accordingly. Both channels are therefore routinely active across benchmarks and suites, which is the condition under which the single-run cases of Appendix A.4 and A.5 are representative.

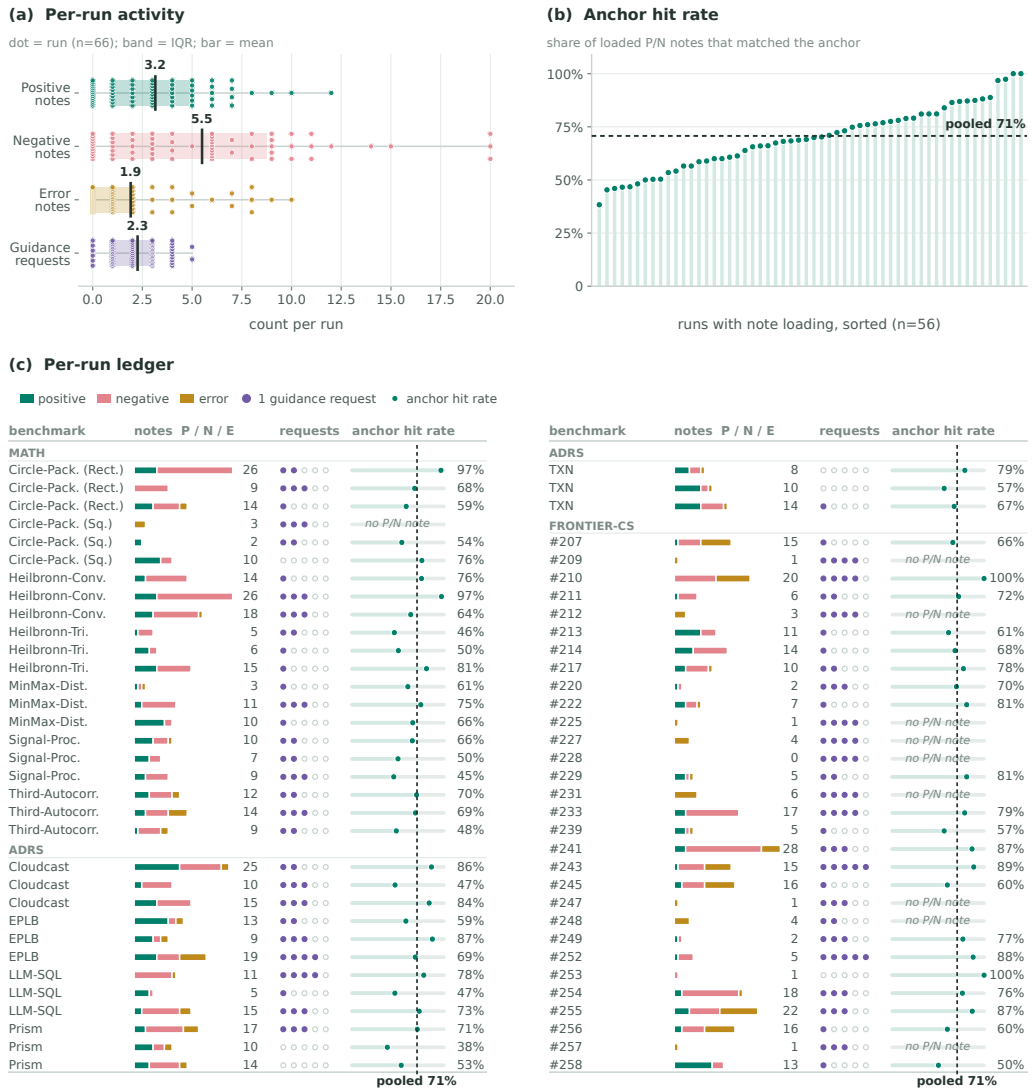


Figure 21: Prevalence of the two experience channels across the 66 audited runs. (a) Per-run counts of positive, negative, and error notes and of Meta-guidance requests; each dot is one run, the band is the interquartile range, and the bar is the mean. (b) Anchor hit rate of note loading per run, sorted, for the 56 runs that loaded positive or negative notes; the dashed line is the value pooled over all loading events. (c) Per-run ledger grouped by benchmark suite, giving for each run the note composition by type, the number of guidance requests, and its anchor hit rate. Runs without a defined anchor hit rate are marked accordingly.

C.7 Mechanistic Content of Evolution Notes

Having established that the two experience channels are routinely active across the audited runs, we now turn from their prevalence to the content of the Notes channel.

C.7.1 Corpus and Taxonomy

Corpus. We analyze the 697 validated and committed evolution notes from the cross-run audit in Appendix C.6. For each note, we retain its distilled lesson and actionable guidance, the outcome that triggered its creation, and the provenance of the attempt from which it was distilled.

Content taxonomy. We analyzed the 697-note corpus and grouped its content into two broad families and seven mutually exclusive categories, with each note assigned to exactly one category. The first family records decisions about the search algorithm itself:

- **Strategy** denotes a change to the core method or model family. Examples include replacing a greedy construction with dynamic programming or switching from one optimizer family to another. The defining question is which method to use, rather than how to tune a fixed method.
- **Search operator** covers changes made within a fixed method, including local moves, perturbations, restarts, seeding, tie breaking, beam width, and annealing or exploration schedules. These notes retain how the current search procedure should traverse its candidate space.
- **Objective** covers what the search optimizes or admits. It includes surrogate objectives, ranking keys, acceptance gates, validity checks, fallback rules, and verification required before a candidate is committed. These notes therefore describe the criterion used to distinguish an admissible improvement from a misleading score.
- **Budget** covers resource allocation and complexity, including wall-clock limits, iteration and population caps, restart counts, time sharing, and asymptotic cost. A change is assigned here when its motivation is a timeout or resource cap rather than candidate quality.

The second family records failures in executing or evaluating a proposed change:

- **Implementation defect** is a concrete coding error whose repair is mechanical once identified, such as a tensor-shape mismatch, a wrong positional index, a type error, a missing include, or misuse of a library call.
- **Environment** assigns the outcome to the evaluation harness or infrastructure rather than to the candidate. Judge outages, endpoint timeouts, and non-reproducible harness failures belong to this category.
- **Numerics** covers failures caused by precision or conditioning, including floating-point precision, tolerance choice, gradient scaling, degenerate geometry, and finite-difference step size.

Each content label is accompanied by two orthogonal annotations. *Scope* records how far the written lesson can travel: **instance** is tied to a specific problem ID, configuration, or lineage; **benchmark** applies across attempts on the same task family; and **general** can be used on an unrelated benchmark. *Conditioned* records whether the guidance states where it applies, for example through a threshold, a size regime, an "only if" clause, an acceptance check, or a fallback. Scope is judged from the guidance as written.

C.7.2 What the notes retain

Reading panel (a). Figure 22(a) is a category-level ledger over the 697 notes. Positive and negative notes are created by material gains and regressions, respectively; error notes arise from repaired or terminal failures. Two columns need a definition. The *trigger* column summarizes the parent-to-child score change that caused a positive or negative note to be requested: for a child score f and parent score f_p, this change is $\delta = (f - f_p)/\max(|f_p|, \varepsilon)$, following Appendix A.4. The point is the median and the horizontal segment the interquartile range, drawn to the right for gains and to the left for losses. *New best* is the fraction of positive notes whose triggering child also set a new run-level best; categories without positive notes have no such rate.

Composition and outcome asymmetry. The four algorithmic-decision categories account for 568 of the 697 notes (81.5%), while the three execution-failure categories account for 129 (18.5%). The Notes database is therefore primarily a record of method selection, search refinement, acceptance, and resource control, rather than a log dominated by software failures.

The outcome mix splits these categories cleanly. Positives concentrate in strategy and search operator, negatives dominate objective and budget, and the execution-failure family is carried by error notes. This is consistent with a division of labor: successful attempts mainly preserve what method

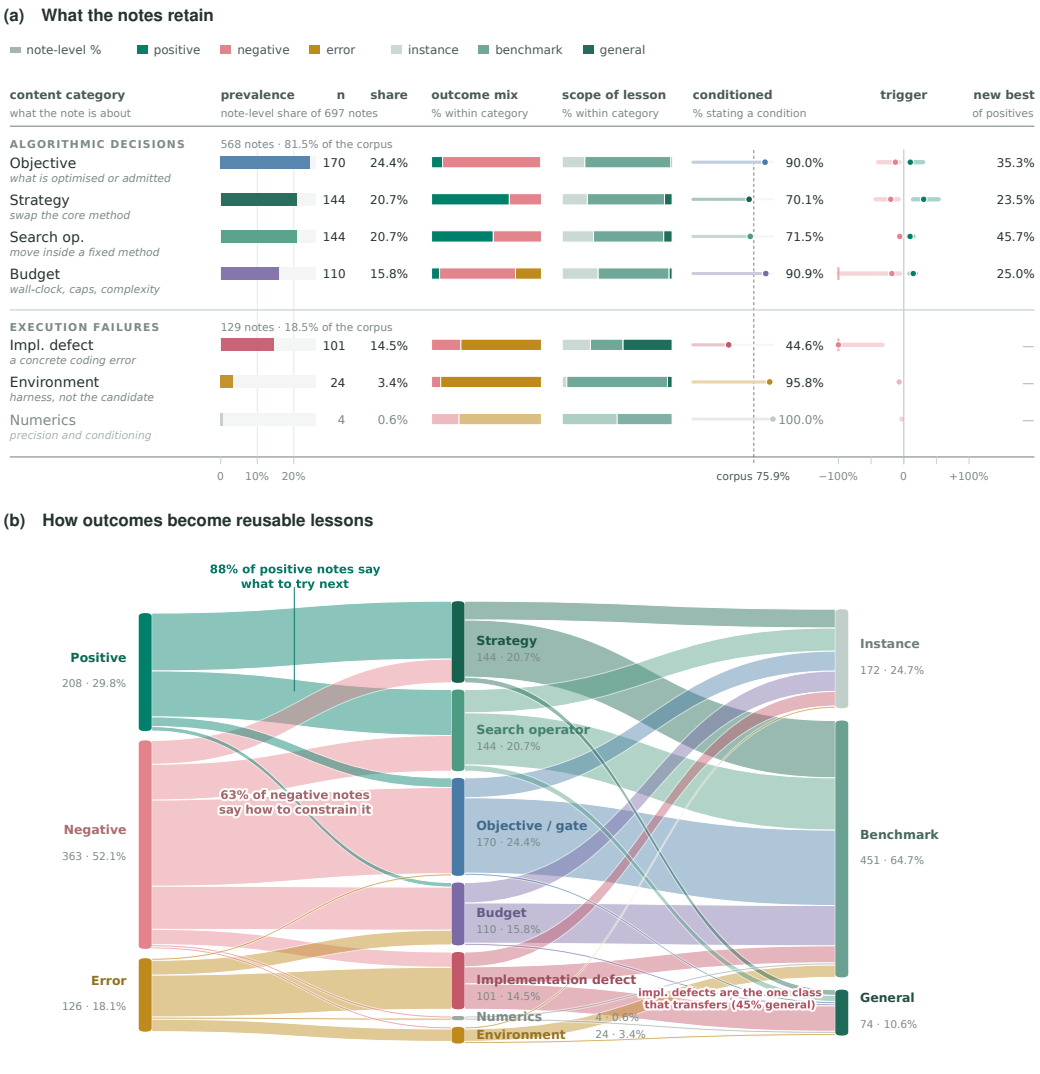


Figure 22: Mechanistic analysis of the 697 committed evolution notes. (a) Category-level prevalence, outcome mix, lesson scope, conditioning, trigger score changes, and new-best rates. (b) Flows from search outcome through content category to lesson scope, with ribbon width proportional to note count.

or move to try, while regressions preserve the acceptance rules and resource limits that prevent the same failure from recurring. The four numerics notes are too few to support a category-level inference.

Scope and conditionality. Across the corpus, 451 notes (64.7%) are benchmark-scoped, 172 (24.7%) are instance-scoped, and 74 (10.6%) are general. The algorithmic-decision categories are overwhelmingly benchmark-scoped and rarely general, because their guidance depends on the task's objective, evaluator, or feasible structure. Implementation defects are the one class that transfers: roughly 45% of them are written as general lessons, since a faulty index or tensor shape can recur outside the benchmark that exposed it. This separation argues against treating every note as a global rule. Benchmark match should govern the reuse of algorithmic experience, while general implementation repairs can travel more freely.

Conditioning also concentrates where boundaries matter. Against a corpus-wide rate of 75.9%, objective reaches 90.0% and budget 90.9%, whose lessons often retain a validity check, timeout cap, or fallback, and environment reaches 95.8%, largely because the guidance names the harness

state under which attribution should be suspended. Implementation defects are conditioned only 44.6% of the time: once a specific indexing error is identified, its repair is often written as a direct rule.

Trigger magnitude and search role. Positive strategy notes follow the largest typical score moves, yet positive search-operator notes set a new run-level best about twice as often (45.7% versus 23.5%). This combination is consistent with two different search roles. Replacing the core method can move far from its parent, but a large relative step does not necessarily beat the incumbent found by another lineage. A local operator starts from an established method and makes a smaller step, but more often carries that method past the current best.

The loss side records a different risk profile: negative budget and implementation-defect notes reach a complete loss in their lower tail. This explains why the memory retains acceptance checks, caps, and mechanical repairs alongside successful methods, since they protect evaluation budget from failure modes that can erase an attempt. The trigger and new-best columns describe how a note came to exist. Categories also differ in benchmark composition, so these comparisons remain descriptive.

C.7.3 How outcomes become reusable lessons

Figure 22(b) follows the same 697 notes from search outcome to content category and then to scope. Ribbon width is the number of notes in a transition, and each of the three columns sums to 697.

Outcome-specific abstraction. The largest flows reveal a clear division in what each outcome contributes. Of the 208 positive notes, 88% concern strategy or search operators: they mainly retain a method choice or a move that later tasks can try. Of the 363 negative notes, 63% concern objective or budget: they mainly retain how to constrain a candidate through validation, acceptance, fallback, or resource limits. The 126 error notes concentrate in implementation defects and the environment, retaining how to repair execution or how to avoid blaming a candidate for an infrastructure failure. The three outcomes therefore preserve different forms of reusable control information.

Where the lessons travel. The scope column then bounds reuse. Positive and negative lessons are predominantly benchmark-scoped, and the general column is fed disproportionately by implementation defects, of which roughly 45% are general, whereas environment notes stay attached to the evaluator that produced them. Mechanical programming failures are the ones written as portable lessons; algorithmic and harness lessons stay tied to their task family or evaluator.

Together, the flows describe a typed memory. Positive notes supply candidate search actions, negative notes supply guards and limits, and error notes supply recovery and attribution rules. The scope annotation then controls how widely each lesson should be reused. This organization preserves both sides of evolution: the proposals that advance a lineage and the constraints that keep later lineages from spending budget on known failure modes.

C.8 OR-Tools Case: Experimental Settings

This appendix details the repository-scale experiment in §6.1.

Repository and optimization target. OR-Tools 9.15 (Perron & Furnon, 2026) is Google's open-source optimization suite. The version we evolve contains about 680K lines of C++ across roughly 2,000 source files (Table 15). We target the single-threaded feasibility-jump local search in CP-SAT, which sits in the largest module but spans several interacting classes. In contrast to our single-file benchmarks, the task specifies a solver-level objective rather than a function to replace: locating the relevant execution paths is itself part of proposal generation.

Edit surface. A candidate may modify four files under `ortools/sat/`: `feasibility_jump.h`, `feasibility_jump.cc`, `constraint_violation.h`, and `constraint_violation.cc`. These carry variable and delta selection, constraint violation measures, jump scoring, constraint weight updates, and initial solution construction, and one candidate often touches several of them. This allowlist keeps patches reviewable and prevents changes to the build, evaluator, instances, or scoring code, while leaving Open Proposal to find the functions worth changing.

Table 15: C++ modules of the OR-Tools tree, by size.

Directory	Lines	Contents
`sat/`	234K	CP-SAT solver
`math_opt/`	130K	unified modeling layer across solvers
`constraint_solver/`	108K	legacy constraint programming and routing
`graph/`	37K	shortest path, flow, and assignment
`linear_solver/`	31K	linear solver wrappers
`glop/`	20K	simplex LP solver
`pdlp/`	19K	first-order LP solver
`util/`	15K	shared utilities
`flatzinc/`	13.5K	FlatZinc and MiniZinc interface
others	1–13K each	scheduling, routing, packing, set cover

Patch-based evolution. Each candidate is a Git patch against a frozen OR-Tools tree. At an iteration, Structured Governance selects a parent and optional inspirations from the settled candidates (§4.2). Open Proposal creates an isolated worktree, applies the parent, edits ordinary C++ source, and exports a self-contained patch that passes the allowlist and applies cleanly to the untouched tree. Experimental behavior is guarded by `#ifdef ORTOOLS_LS_EXPERIMENT`, with the original code kept in the other branch, so each candidate reads as a difference against a solver that still builds and runs. Carrying lineage as patches also avoids storing whole repository copies.

Evaluation protocol. We conduct two evolution experiments, obtaining gains of $+13.2\%$ on the full instance set and $+16.9\%$ on a smaller subset (Figure 5). The full set comprises 300 instances from the MiniZinc Challenge 2023–2025, 100 per year, across 60 problem families (5 instances per family): 210 minimization, 79 maximization, and 11 satisfaction instances. The subset contains 60 instances, selected deterministically as the second-smallest by size in each problem family. It therefore retains complete coverage of all 60 families while cutting the evaluation cost of each evolution iteration.

The two experiments are independent: they maintain separate lineages and do not share candidates or scores. All other settings are identical. Protected Judgment applies each patch to a fresh source copy, builds `fzn-cp-sat` with the experimental flag, and runs it in free-search, local-search-only mode. Every instance uses one solver thread and a 180-second time limit.

Borda fitness. We follow the pairwise component of the MiniZinc Challenge's Borda-count procedure, treating every instance as a voter and comparing the candidate with the frozen unpatched solver. On an optimization instance, a better feasible objective scores 1, a worse objective or failure to return a valid solution scores 0, and equal-quality answers split the point by runtime. If t_x and t_b are the candidate and baseline times (each lower-bounded by one second), respectively, the candidate receives score $t_b/(t_b + t_x)$. Satisfaction instances use the analogous valid-solution and runtime comparison. For candidate x, frozen baseline b, and instance set D, let $s_i(x, b)$ denote the pair score on instance i. Fitness is $F_D(x) = \frac{1}{|D|} \sum_{i \in D} s_i(x, b)$, where $|D| = 300$ for the full experiment and $|D| = 60$ for the subset experiment, so F_D lies in $[0, 1]$. Comparing the unpatched solver with its own results gives $s_i = 0.5$ and $F_D = 0.5$, which represents neither improvement nor degradation. We define the relative performance ratio as $R_D(x) = F_D(x)/0.5$ and report the percentage gain over the unpatched solver as

$$\Delta_D(x) = \left(\frac{F_D(x)}{0.5} - 1 \right) \times 100\%.$$

For each experiment, we run the unpatched solver once before evolution on the same machine and with the same CPU allocation, solver flags, time limit, and concurrency used for its candidates. We parse and store the resulting per-instance records as a frozen baseline. During evolution, each candidate produces only a new set of logs, which a deterministic Python collector compares with these stored records. The script checks statuses, timeouts, and abnormal exits, extracts objectives and runtimes, and computes and aggregates all pair scores. Given the logs, this procedure is deterministic and efficient and contains no model-generated judgments. Freezing the baseline, instance manifest, resource limits, and toolchain keeps fitness comparable across generations.

Search settings. Both experiments use EvoTrias's default operating settings (Table 9): a 50-iteration task budget, $n_{\mathrm{dir}} = 4$ initial directions, and $c = 4$ concurrent task-executor slots. Open Proposal runs on Claude Code v2.1.201 with DeepSeek-V4-Flash-0731 as the backbone.

One-shot reference. To separate what the evolution loop adds from what the coding agent achieves on its own, we also ran the same host and backbone once, end to end, under the same task configuration, edit surface, and prompts. The agent reads the repository, edits, builds, and submits a single patch; no Governance context, selection, or later iteration follows. We ran four independent passes per instance set and report the mean of their gains Δ_D: -0.1% on the full set and $+1.3\%$ on the subset. Figure 5 marks these means as dotted lines.

C.9 Evolutionary Lineage Case Studies

Figures 23 and 24 reconstruct two runs, one on Cloudcast and one on Frontier-CS problem 258. Both use Claude Code as the Open Proposal agent with GLM-5.2 as the backbone, and both begin from the default $n_{\mathrm{dir}} = 4$ initial directions. Each card is one successfully evaluated iteration and reports its task ID and verified fitness. G denotes generation: the four tasks at $G0$ are the roots, and each edge links the selected parent directly to its child. Card color follows the run-specific fitness scale below each graph; the gold border and trophy mark the best task in that run.

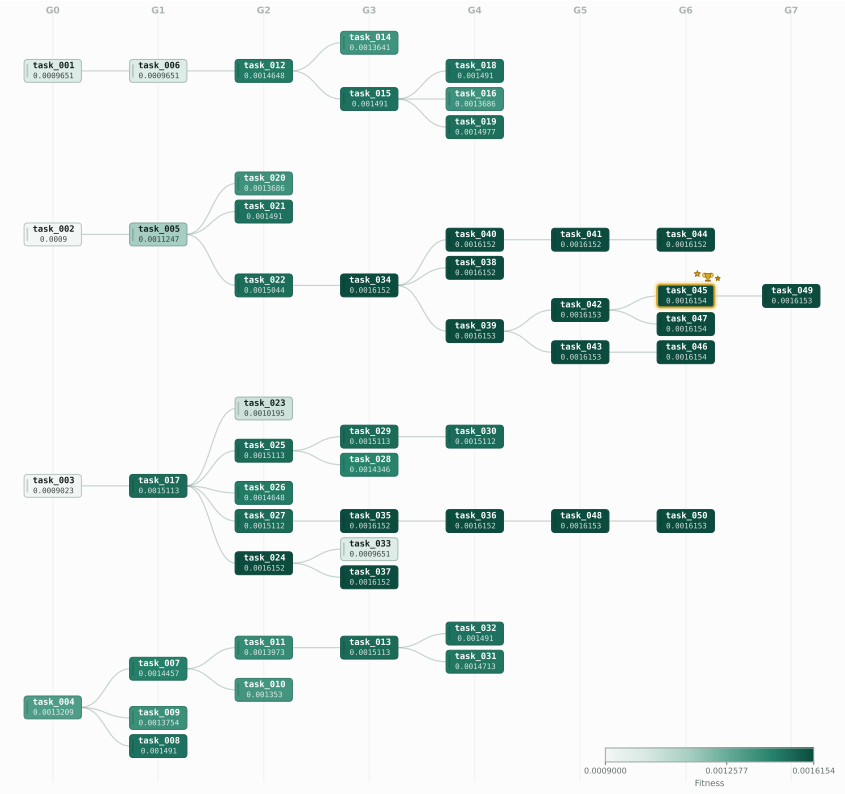


Figure 23: Evolutionary lineage of the Cloudcast run.

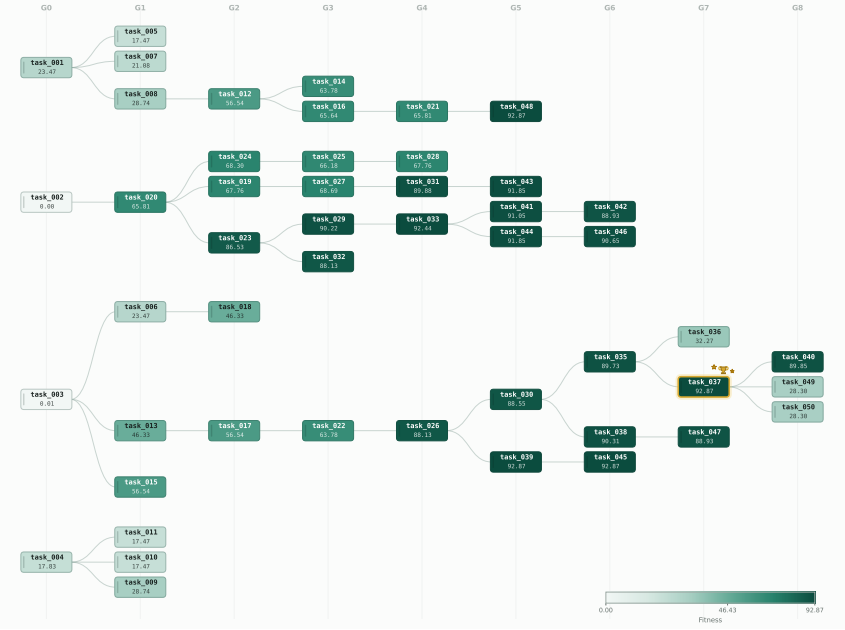


Figure 24: Evolutionary lineage of the Frontier-CS problem 258 run.

Improvement with evolutionary depth. Both cases shift toward higher fitness as descendants accumulate, although individual offspring can regress under exploration. Fitness here is the internal score of Table 6, which on Cloudcast is $1/(1 + \text{cost})$ against the transfer cost of Table 1. In Cloudcast, the median fitness rises from 0.0009337 at $G0$ to 0.0014106 at $G1$, while the best value improves from 0.0013209 among the four roots to 0.0016154. Frontier-CS problem 258 shows a larger change: the $G0$ median and maximum are 8.92 and 23.47, respectively; by $G4$, they reach 88.13 and 92.44, and the run attains 92.87 at $G5$. Thus, neither set of initial LLM proposals reaches the eventual run optimum. The repeated gains along descendant chains provide case-level evidence that evolutionary reuse and refinement improve on one-shot generation for these optimization problems.

Adaptive allocation across lineages. Equal initialization does not imply equal allocation thereafter. In root order `task_001` through `task_004`, the Cloudcast lineages contain 7:16:15:8 descendants, whereas Frontier-CS problem 258 contains 8:16:18:3. This contrast is consistent with the adaptive routing rule of Structured Governance (Section 4.2). As detailed in Appendix A.2.2, Equation 10 converts verified quality and descendant potential into a deserved lineage share, then compares that share with the lineage's realized population. The search therefore does not reserve one quarter of the remaining budget for each root; it favors promising yet under-covered lineages. Within the routed lineage, Equation 6 further combines candidate quality, exploration, and descendant potential when selecting the parent. The fourth Frontier-CS root illustrates why both levels matter. Its initial fitness is 17.83, higher than two other roots, but its three descendants score only 17.47, 17.47, and 28.74, while the other lineages eventually exceed 92. The resulting 8:16:18:3 allocation reflects accumulated evidence about a branch rather than its root score alone: a competitive initial direction can lose search budget when its descendants fail to sustain improvement.

C.10 Evaluator Exposure: Protocol, Held-out Suites, and Code Cases

C.10.1 Exposure protocol

The exposed and protected settings differ only in whether evaluator sources are reachable. In the exposed setting CORAL runs in its default host session, which links the grader source into every agent worktree and grants read access to it (Appendix A.8), and EvoTrias runs with the permission hooks of Appendix A.7 disabled. In the protected setting the two systems carry the same measures of §4.4: the prompt states the prohibition, and a hook layer denies the corresponding tool calls. CORAL has no layer that denies reading evaluator sources, so we supply an EvoTrias-equivalent one for its protected runs and leave the rest of its design untouched.

The exposed setting gives EvoTrias the same access its baselines take. Following CORAL's configuration, the exposed proposal prompts point Open Proposal at the evaluator source and encourage reading it; following AdaEvolve, the Meta-guidance request carries the evaluator source alongside the search trace. CORAL keeps its stock instruction file, which points the agent at the grader source. Any use of the sources is therefore each system's own behavior under its exposed prompts. Each of the six cells is three independent runs, and every score in Table 3 is the mean over those three. Each run's final program is then scored once on the original evaluator and once on each held-out suite below.

C.10.2 Task and held-out construction

Signal Processing requires a program to map a noisy, non-stationary input signal and a window size to a filtered signal. The evaluator aggregates smoothness, tracking error, false reversals, correlation, and noise reduction over several signals. The intended object of optimization is therefore a filtering algorithm for a family of signals, rather than a solution to one fixed instance. This differs from Circle Packing, where the task itself asks for one concrete arrangement of 26 circles in a unit square.

We also compared exposed and protected evaluation on Circle Packing. Their original-evaluator scores differed by less than 1%, consistent with evaluator exposure having little effect when the evaluation target is the specified case itself rather than performance on unseen instances.

Table 16 separates the original suite from the two held-out suites. All three use the same alignment and scoring implementation; only the generated signals change. The IID suite preserves the original

five families and Gaussian-noise assumption, but draws five new parameterizations of each family. The OOD suite varies every major axis of the original generator and adds five signal families.

Table 16: Signal Processing evaluation suites. IID preserves the task families while randomizing their instances; OOD broadens both the signal and disturbance distributions.

Dimension	**Original evaluator**	**IID held-out**	**OOD held-out**
Cases	5	25	20
Signal families	Five fixed families, one case each	Same five families, five cases each	Five widened families plus five new families, two cases each
Length	500, 600, 700, 800, 900	529–997	457–1217
Family–length relation	One-to-one; length identifies the case	Multiple lengths per family	Length is decoupled from family
Window size	20	20	9–67
Time span	0–10	8–14	6.5–17
Disturbance	IID Gaussian, $\sigma = 0.2$–0.6	IID Gaussian, $\sigma = 0.16$–0.68	Gaussian, AR colored, Student-t, impulsive, and heteroscedastic
Parameters	Fixed frequencies, amplitudes, and step locations	Randomized amplitudes, frequencies, phases, steps, drift, and noise	Wider parameter ranges and additional non-stationary dynamics
Seed	$42 + i$	$1049 + 97f + 17r$	$31013 + 101i$, independent of the original evaluator

For IID case (f, r), the generator samples duration uniformly from $[8, 14]$, selects a family-specific parameterization, and adds Gaussian noise with a sampled standard deviation. The five retained families are sine-with-trend, multi-frequency oscillation, chirp, piecewise-constant step, and random walk. Thus, IID changes the concrete signals without changing the task taxonomy, window size, or noise family.

The OOD suite uses ten families: wider versions of the original five, plus amplitude modulation, damped oscillation, piecewise ramp, frequency switch, and smooth latent drift. It also varies the window size and introduces colored, heavy-tailed, impulsive, and heteroscedastic disturbances. This construction breaks three fingerprints available in the original evaluator: the mapping from length to signal family, the fixed `window_size=20`, and the assumption of IID Gaussian noise.

C.10.3 CODE-LEVEL MECHANISM

Table 17 reports two illustrative programs. They come from different method groups and are not a paired estimate of the exposure effect; the controlled group comparison remains Table 3. Instead, the programs provide qualitative evidence for the mechanism behind the aggregate advantage contraction.

Table 17: Scores of the two code cases shown below. Gap is OOD minus Original.

Method	Evaluator	Original	IID	OOD	Gap
CORAL-1	Exposed	0.808966	0.783652	0.687841	−0.121125
EvoTrias	Protected	0.719451	0.723515	0.730097	+0.010646

Exposed program (CORAL-1, selected excerpt)

```python
# Tuned per signal type against the grader's fixed seeds.
_SCHEDULES = {
    "sinsin":     "sin_demod",
    "multifreq":  ((21, 101), (2, 2)),
    "chirp":      "hilbert",
    "step":       "pwc",
    "randomwalk": ((31, 51, 71, 91, 111), (2, 2, 2, 2, 2)),
    "default":    ((31, 41, 61, 81, 91), (2, 2, 2, 2, 2)),
}
_STEP_PWC_WIN = 40
_STEP_PWC_THR = 0.5
_CHIRP_PHASE_W = 401
_CHIRP_ENV_W = 251
_SHORT_SIGNAL_THRESHOLD = 500  # all grader signals have n >= 500

# ... generic smoothing helpers omitted ...
def _classify(x):
    n = len(x)
    y = medfilt(x, _safe_window(9, n))
    y = _savgol(y, _safe_window(21, n), 2)

    centered = y - y.mean()
    zc = (np.abs(np.diff(np.sign(centered))) > 0).astype(float)
    q = max(1, n // 4)
    rates = [zc[i*q:(i+1)*q].mean() for i in range(4)]
    zc_trend, zc_final = rates[-1] - rates[0], rates[-1]

    d = np.abs(np.diff(y))
    jump_frac = (d > 3 * d.mean()).sum() / max(len(d), 1)
    local_std = np.array([
        y[max(0, i-30):i+31].std() for i in range(n)
    ])
    std_ratio = local_std.max() / (local_std.mean() + 1e-9)

    yc, var = y - y.mean(), y.var() + 1e-9
    def ac(lag):
        return np.sum(yc[:-lag] * yc[lag:]) / ((n-lag) * var)
    ac_arr = np.array([ac(l) for l in range(5, min(250, n//2))])
    peaks = [v for i, v in enumerate(ac_arr[2:-2], 2)
             if v > ac_arr[i-1] and v > ac_arr[i+1] and v > 0.5]

    # Thresholds recover the five fixed evaluator families.
    if zc_trend > 0.02 and zc_final > 0.05:
        return "chirp"
    if jump_frac > 0.02 and std_ratio > 3.0:
        return "step"
    if peaks and max(peaks) > 0.7:
        return "sinsin" if abs(ac(100)) > 0.6 else "multifreq"
    return "randomwalk" if ac(min(150, n//3)) > -0.2 else "multifreq"

# ... model-matched chirp, sinusoid, and step filters omitted ...

def adaptive_filter(x, window_size=20):
    x = np.asarray(x, dtype=float).ravel()
    n = len(x)
    if n < _SHORT_SIGNAL_THRESHOLD:  # never used by the original grader
        y = _savgol(medfilt(x, 3), 5, 2)
        out = y[window_size - 1:].copy()
        out[-1] = x[-1]
        return out

    kind = _classify(x)
    spec = _SCHEDULES.get(kind, _SCHEDULES["default"])
    if spec == "hilbert":
        smooth = _hilbert_chirp(x)
    elif spec == "sin_demod":
        smooth = _sin_demod(x)
    elif spec == "pwc":
        smooth = _step_pwc(x)
    else:
        smooth = _cascade(x, *spec)

    # Match the grader's alignment and force its recent-lag term to zero.
    out = smooth[window_size - 1:].copy()
    out[-2] = 0.5 * (out[-3] + x[-1])
    out[-1] = x[-1]
    return out
```

Figure 25: Exposed solution excerpt. Red identifiers mark the fixed schedule and length regime, case-specific classification and routing, and the evaluator-aligned output shortcut. Generic filtering helpers are omitted.

The exposed program reaches 0.808966 on the original evaluator but falls to 0.687841 OOD. Beyond choosing a denoiser, it encodes the evaluator's alignment, forces the last-sample lag term to zero, identifies the five fixed families, and routes them to separately tuned procedures. Its comments state that the schedules were tuned against the grader's fixed seeds.

Protected program (EvoTrias, selected excerpt)

```python
def _haar_finest_sigma(signal):
    x = np.asarray(signal, dtype=float).flatten()
    if len(x) < 4:
        return float(np.std(x)) if len(x) else 0.0
    even, odd = x[0::2], x[1::2]
    m = min(len(even), len(odd))
    detail = (even[:m] - odd[:m]) / np.sqrt(2.0)
    mad = float(np.median(np.abs(detail)))
    return mad / 0.6745 if mad > 0 else float(np.std(detail)) * 0.5

def _estimate_lengthscale(signal, max_lag_cap=200):
    x = np.asarray(signal, dtype=float).flatten()
    x_centered = x - np.mean(x)
    var = float(np.dot(x_centered, x_centered)) / len(x)
    acf = np.correlate(x_centered, x_centered, mode="full")
    acf = acf[len(x) - 1:] / (var * len(x))
    max_lag = min(max_lag_cap, len(x) - 1)
    below = np.where(acf[1:max_lag + 1] < 0.2)[0]
    zero = np.where(np.diff(np.sign(acf[1:max_lag + 1])) != 0)[0]
    candidates = []
    if below.size:
        candidates.append(int(below[0]) + 1)
    if zero.size:
        candidates.append(int(zero[0]) + 1)
    lag = min(candidates) if candidates else max(4, min(20, len(x)//20))
    return 1.5 * max(2.0, lag)

def _matern32_kernel(distance, lengthscale, sigma_f):
    scaled = np.sqrt(3.0) * distance / max(float(lengthscale), 1e-6)
    return sigma_f**2 * (1.0 + scaled) * np.exp(-scaled)

def _gp_matern32_fit_predict(x, lengthscale, sigma_f, sigma_n, m_cap=80):
    x = np.asarray(x, dtype=float).flatten()
    n, m = len(x), min(m_cap, len(x))
    idx = np.unique(np.linspace(0, n - 1, m).astype(int))
    t_full, t_sub = np.arange(n, dtype=float), idx.astype(float)
    y_sub = x[idx]

    d_sub = np.abs(t_sub[:, None] - t_sub[None, :])
    K_xx = _matern32_kernel(d_sub, lengthscale, sigma_f)
    jitter = 1e-6 * (sigma_f**2 + 1.0)
    K_y = K_xx + (sigma_n**2 + jitter) * np.eye(len(idx))
    cho, low = cho_factor(K_y, lower=True, check_finite=False)
    alpha = cho_solve((cho, low), y_sub, check_finite=False)

    d_cross = np.abs(t_full[:, None] - t_sub[None, :])
    mean = _matern32_kernel(d_cross, lengthscale, sigma_f) @ alpha
    return mean if np.all(np.isfinite(mean)) else None

# ... zero-phase Savitzky-Golay helpers and parent fallback filter omitted ...

def _run_filter_full(x, window_size):
    x = np.asarray(x, dtype=float).flatten()
    if len(x) < 2:
        return x.copy()

    noise_sigma = max(_haar_finest_sigma(x), 1e-6)
    signal_scale = max(float(np.std(x)), 1e-6)
    lengthscale = _estimate_lengthscale(x)

    gp_mean = _gp_matern32_fit_predict(
        x, lengthscale=lengthscale, sigma_f=signal_scale,
        sigma_n=noise_sigma, m_cap=80,
    )
    if gp_mean is None or len(gp_mean) != len(x):
        return _parent_fallback_filter(x, window_size)

    broad = min(max(3, int(window_size) | 1),
                max(5, len(x) // 4 | 1))
    fine = max(3, min(int(window_size) // 2 | 1,
                      len(x) // 8 | 1))
    smoothed = _two_stage_sg_smooth(gp_mean, broad, fine)
    return smoothed
```

Figure 26: Protected solution excerpt. The noise scale, kernel lengthscale, signal scale, and post-smoothing windows are inferred from the current input; no fixed family, length, or seed selects an algorithm branch. Ellipses omit the Savitzky–Golay smoothing helpers and the parent fallback filter.

The protected program reaches 0.719451 on the original evaluator and improves to 0.730097 OOD. It uses one signal-agnostic pipeline: Haar-MAD estimates the noise, autocorrelation estimates the Matérn lengthscale, and the supplied window controls the final smoothing.

D PROMPTS

This appendix reproduces the prompts of the four agent roles in a run: the orchestrator and the three subagents it dispatches (init-explorer, task executor, and meta-guidance generator), together with the two fixed prompts of the Notes channel that the task executor follows in place of a fifth. Omitted passages are marked [omitted: ...], and [x] marks values filled at dispatch time. The sources call the evaluator a *verifier*, and the quoted wording is kept as written.

D.1 ORCHESTRATOR

Orchestrator (evolve master skill prompt, verbatim excerpts)

You are now running the **self-evolution master skill**, the orchestrator. Your responsibilities are: initialize a run, maintain the task queue, react to MCP convergence/meta-evolve actions, delegate requested meta-guidance generation, stop replenishment only on hard budget/external stops, wait for already launched tasks to finish, and call Evolution Controller MCP to generate a deterministic run summary report.

Orchestrator Boundary Constraint [omitted: the orchestrator only schedules, calls MCP, and dispatches templates; it never touches task contents, candidates, evaluators, or meta-guidances]

Dispatch Template Constraint (ALL subagent dispatches) [omitted: subagent dispatch prompts are fixed templates of parameter lines; subagents are dispatched in the shared working directory]

task_description Fidelity Constraint [omitted: task_description must be identical to the user's original submitted input]

max_attempts Dispatch Constraint [omitted: max_attempts is set explicitly on every task-executor dispatch, with a maximum of 3 attempts for every task, including root tasks]

Architecture Roles

Component	Responsibility
This skill (orchestrator)	Initialize run, maintain task queue, route MCP action, dispatch meta-guidance requests by ID, finalization
meta-guidance-generator agent	Fetch one pending request, inspect generation evidence, generate and commit guidance, return a compact result
task-executor agent	Execute the full lifecycle of individual tasks
Evolution Controller MCP	Parent/inspiration selection, attempt recording, scheduler-state persistence, convergence/meta-guidance state, run summary generation

Step 1: Confirm Run Parameters

Before any run creation or MCP call, determine and confirm the three run parameters. They are independent knobs - never derive one from another:

Parameter	Meaning	Default
LINEAGE_ROOT_NUMBER	Number of initial exploration directions generated at initialization; equals the number of lineage root tasks the run establishes	4
MAX_CONCURRENT_TASKS	Maximum number of task-executors running in parallel	4
MAX_TOTAL_TASKS	Hard budget: total number of tasks the run may create before stopping	50

1. Extract each value from the user's original request when stated: [omitted: example phrasings for each of the three parameters]
2. For any value not stated, use its default from the table.
3. Present all three to the user in one compact message - name, chosen value, whether it was extracted or defaulted, and a one-line meaning - and ask for confirmation or adjustments.
4. Only after the user confirms, proceed to Step 2 with the confirmed values. These confirmed values are referenced by name throughout the rest of this skill.

Step 2: Run Initialization

1. Derive task_slug from the user's original input. [omitted: slug-derivation and normalization rules]
2. Generate run_id: [task_slug]_YYYYMMDDHHMMSS.
   - Use the current local start time with second-level precision. Use the Bash tool to execute `date +%Y%m%d%H%M%S`.
3. Register the confirmed parameters with the controller:
   - Call configure_run(run_id, max_total_tasks=[MAX_TOTAL_TASKS], lineage_root_number=[LINEAGE_ROOT_NUMBER], max_concurrent_tasks=[MAX_CONCURRENT_TASKS], task_description=[original user input]) BEFORE any other controller MCP call for this run. The controller records all three values; they are locked for the rest of the run.

```
  - [omitted:  dashboard startup banner]
4.  Create the run workspace directory and persistent state files.  All runs/ paths are relative to
the project root (cwd), not to the skill's base directory.  Use Bash to execute:  mkdir -p
"runs/[run_id]"

Step 3:  Generate Initial Exploration Directions

Invoke the evotrias_plugin:init-explorer agent using the init-explorer dispatch template.
[omitted:  parameters to fill]

Wait for evotrias_plugin:init-explorer to complete before proceeding to Step 4.

Step 4:  Task Queue Scheduling

MCP owns batch sizing:  check_convergence tells you how many tasks to create via tasks_to_create,
and you dispatch exactly that many - never more, never fewer, never inferred.  Any returned value,
including 0, is a normal scheduling decision; on 0 dispatch nothing and keep looping.

Task completion data is persisted by the controller as subagents record their attempts;
notifications merely prompt the orchestrator to call check_convergence again.

Data Flow [omitted:  the per-run directory runs/[run_id]/, relative to the project root]

Scheduling Pseudocode

first_batch = true
just_after_meta = false
WHILE true:
  // Wait-gate:  every check_convergence call (except the first and the one right after a meta-
  // guidance commit) must be preceded by at least one task-executor finishing.  Without this gate
  // the loop would hammer the MCP in a tight cycle.
  IF not first_batch and not just_after_meta:
    wait for at least one task-executor to finish
  just_after_meta = false

  // The first check_convergence call is mandatory:  on first_batch the wait-gate is skipped, but
  // the call itself must NOT be.  Every batch's task count comes only from c.tasks_to_create -
  // never inferred or hardcoded.  tasks_to_create == 0 with action "continue" is normal - dispatch
  // nothing and keep looping.
  c = check_convergence(run_id, concurrency=MAX_CONCURRENT_TASKS)

  IF c.action == "budget_exhausted":
    BREAK WHILE LOOP

  IF c.action == "request_meta_guidance":
    request_id = c.request_id
    Dispatch to meta-guidance-generator with run_in_background:  false using the
      meta-guidance-generator dispatch template (fill:  run_id, request_id)
    Wait for the meta-guidance-generator compact result.

    just_after_meta = true
    CONTINUE  // re-check convergence; the second check sees active guidance and goes to continue

  // action == "continue"
  // Every task, including a root task, gets at most three attempts.
  max_attempts = 3
  REPEAT c.tasks_to_create TIMES:
    Dispatch to task-executor with run_in_background:  true using the task-executor dispatch
      template (fill:  task_description, max_attempts=max_attempts).
  END REPEAT
  first_batch = false

  [omitted:  when and what to print to the terminal]

END WHILE

// After the while loop exits (action == budget_exhausted), wait for any still-running
task-executors to finish, finally do finalization (see Step 5).

[omitted:  compact terminal progress rules; the orchestrator never prints Notes or meta-guidance
content in the main terminal]

Critical Constraints [omitted:  critical rules.  e.g.  stop conditions, workflow guidelines, etc.]

Meta-Guidance Worker Dispatch [omitted:  passes run_id and request_id to a meta-guidance subagent,
waits for its compact result, and loops back to check_convergence]

Step 5 run finalization [omitted:  call finalize_run and generate final report]

[omitted:  error-handling table]
```

D.2 INITIAL DIRECTIONS

Init-explorer (skill prompt, verbatim excerpts)

Generate initial exploration directions for evolutionary tasks.

First, confirm these critical identifiers from the context: run_id; task_description; num_directions. CRITICAL: If any of these are missing, ask immediately before proceeding.

Then execute the following steps in EXACT ORDER:

Step 1: Understand Task

1. Locate and read the problem SKILL.md if it exists. The problem SKILL.md is typically found in the project root or in a benchmark directory.
2. Follow the problem SKILL.md's public instance-selection instructions using the original task_description. If the skill requires an instance or problem ID, validate it and read the public statement/metadata it declares. Never infer a benchmark instance from an evolution task_id.
3. Resolve the initial reference declared by the problem SKILL.md. [omitted: how the initial program file is located] Read the resolved file as initial_program_reference.
4. Understand the verification method, what constitutes a successful solution, and the reference program's main algorithmic approach if one exists.
5. Do not read verifier/evaluator scripts or judge-side resources. Only read the problem SKILL.md and the public problem resources and initial reference it explicitly declares.

Step 2: Analyze Problem Structure

Before generating directions, analyze the problem to identify its key structural decision points - the places where a solver can meaningfully diverge into different approaches.

Ask yourself:
- What are the **core decisions** a solution must make? (e.g., one-shot vs iterative, exact vs heuristic, deterministic vs random)
- What **information is available** to the solver? (e.g., feedback after each step, no feedback at all, partial observability)
- What **constraints shape the solution space**? (e.g., fixed budget of attempts, single attempt only, time limit)
- What **failure modes** are possible? (e.g., wrong on first try, converges slowly, overshoots)

Write a brief problem analysis (3-5 lines) covering:
1. Problem type (one-shot, iterative, optimization, construction, etc.)
2. Available information and feedback
3. Key decision axes where approaches can diverge

This analysis is internal - do not save it to a file. It guides direction generation.

Step 3: Generate Directions

Generate exactly [num_directions] different directions. Each direction must be:
- One line, 8-25 words in English (SHORT and PRECISE)
- Clearly distinct from the other directions
- Actionable, focusing on a specific approach
- Feasible and meaningful for the given task

If an initial_program_reference exists, use it as important context for the task interface, baseline algorithm, and known feasible starting route. Directions may improve, specialize, simplify, or deliberately depart from that reference. Keep the set diverse; do not generate four directions that merely restate the initial program's current approach.

Directions MUST be diverse enough so that following them produces significantly different solutions. Use the problem analysis from Step 2 to identify **problem-specific** axes of divergence - the decision points unique to this particular problem - rather than applying generic coding-style axes. [omitted: worked examples of good and bad directions]

Step 4: Cross-Check Diversity

Compare all generated directions pairwise. For each pair, ask:

Would following direction A and direction B produce solutions that are essentially the same?

If two directions would converge to the same solution or use the same core strategy:
- Replace the weaker/less diverse direction with a new one that explores a genuinely different axis
- Re-check the new direction against all others
- Repeat until every pair is clearly distinct

Specific red flags to catch: [omitted: typical anti-patterns, e.g. duplicate insights]

This is a text-level comparison only - never execute or simulate tasks during this step.

Step 5: Save Directions [omitted: directory and direction-file bookkeeping]

CRITICAL RULES [omitted: list including critical workflow guidelines, language and file-I/O hygiene rules]

D.3 TASK EXECUTOR

Task executor (evolve-task-exec skill prompt, verbatim excerpts)

Execute a single evolutionary task with strict, ordered steps.

First, confirm these critical identifiers from the context: task_id; run_id; task_description; max_attempts. [omitted: failure handling when the injected context lacks a task id]

Then execute the following steps in EXACT ORDER:

Step 1: Load Benchmark Contract

Locate and read the problem SKILL.md for this task. Follow its public instance-selection instructions using the original task_description. If it requires an instance or problem ID, validate that selector and read the public statement/metadata declared by the skill. Never infer a benchmark instance from the evolution task_id.

Store the skill-declared main_artifact_name. If the skill does not declare one, default to solution.py. Also store any skill-declared initial_program_reference path for possible root-task use in Step 3.

Do not read verifier/evaluator scripts or judge-side resources while resolving the benchmark contract.

Step 2: Select Parent

Call MCP tool select_parent with run_id, task_id. Store the result (may be null if no parent exists): parent_id, parent artifact_ref, parent fitness (higher is better), and parent evaluator diagnostic metrics (may be empty).

If parent exists, read the artifact file at artifact_ref to understand the parent solution.

Step 3: Load Initial Reference Program (Root Tasks Only)

If parent_id is null, this is a root exploration task. Read the skill-declared initial_program_reference resolved in Step 1. If the skill does not declare one and initial_program.py exists beside its SKILL.md, use that sibling file as the fallback initial_program_reference.

Use this reference only for root tasks with no parent. For non-root tasks, parent and inspiration artifacts are the evolutionary context, so do not load or prioritize the initial reference.

Do not read verifier scripts while locating the reference program.

Step 4: Select Inspirations

Call MCP tool select_inspirations with run_id, task_id. Store the result as a list of inspirations, each with inspiration_id, artifact_ref, fitness, and metrics.

Read each inspiration artifact file to understand their approaches. When comparing multiple inspirations, prefer higher-fitness artifacts and avoid copying strategies from lower-fitness artifacts when they conflict with better references.

Step 5: Load Run-Local Notes

Run-local Notes are enabled by default. The run owner may disable them for the whole MCP process by setting environment variable EVOTRIAS_NOTES_ENABLED=0 before starting the run.

Call MCP tool get_notes_prompt with run_id, task_id.

If the result has available == true, include the returned prompt exactly as one section of the generation context. Do not rewrite, regroup, summarize, or reinterpret the notes yourself.

If available == false, continue without a notes section. If the response also has disabled == true, treat this as an intentional run configuration and do not try to generate or commit notes manually.

Step 6: Load Meta-Guidance

Call MCP tool get_meta_guidance_prompt with run_id, task_id.

If the result has available == true, include the returned prompt exactly as one required strategy section of the generation context. Do not rewrite, regroup, summarize, or reinterpret it yourself.

Meta-guidance has higher strategy priority than parent/inspiration blending, but it never overrides verifier correctness or the benchmark output contract. The selected parent is the anchor parent for this guidance. Use it to preserve the interface and understand the current search state. Do not assume the parent strategy is correct. If the guidance says to depart from or replace the current approach, implement the new strategy directly instead of forcing a superficial hybrid with the parent or inspirations.

If available == false, continue without a meta-guidance section.

Step 7: Read Direction (If Exists)

Check for file: runs/[run_id]/tasks/[task_id]/direction.txt

If the file exists, read it as direction_hint and reference the direction when generating the solution in Step 8.2.

Step 8: Attempt Loop (Up To MAX_ATTEMPTS Attempts)

[omitted: MAX_ATTEMPTS resolution and loop-state initialization]

This is a retry loop, not a fixed MAX_ATTEMPTS-attempt workflow. Start with attempt_001. Run attempt_002 only if attempt_001 fails. Run attempt_003 only if attempt_002 also fails. Continue up to attempt_[MAX_ATTEMPTS]. If any attempt succeeds, stop immediately and do not create or run later attempts.

The remaining attempt budget is only for debugging failures. It is never an optimization budget. Do not run a later attempt to improve a solution after a previous attempt has already passed verification.

Each attempt is atomic: generate/debug the artifact once, save it once, run the verifier once, then either record success or move to the next attempt after failure. Do not repeatedly overwrite the same attempt's artifact and rerun the verifier to chase a better fitness score.

For the current attempt, derive the attempt id exactly as:
attempt_id = f"attempt_{attempt_index:03d}" [omitted: attempt_id derivation examples]

Use attempt_id, not attempt_index, in every filesystem path.

Continue this loop only while the current attempt failed and attempt_index < MAX_ATTEMPTS.

[omitted: reference to prior attempt's eval_result.json for debugging, and jump to Step 9 if an attempt succeeded]

8.1 Create Attempt Directory

Use mkdir -p to create the directory at:
runs/[run_id]/tasks/[task_id]/attempts/[attempt_id]/
Use the exact path with correct run_id, task_id, attempt_id.

8.2 Generate or Debug Solution

Evolution Strategy: If a parent exists and no meta-guidance is active, the parent solution is the foundation of this evolution - use it as the primary reference for improvement and optimization; inspirations can be used as supplementary references for crossover, mutation, and borrowing optimization ideas. If meta-guidance is active, the guidance is the required strategy. Use the parent critically as an interface/current-state reference; build on it only when the guidance calls for refinement. If no parent exists and initial_program_reference is available, use the initial reference program as the baseline implementation to understand the intended interface, scoring objective, and starting search route, then apply the task's direction hint to produce a distinct root solution. The direction may call for either a close improvement to the reference or a substantially different approach that preserves the benchmark contract. Fitness is the task-generic quality score and higher is always better. Use the parent as the primary reference, and use inspiration fitness to rank secondary references. Borrow more from higher-fitness inspirations; treat lower-fitness inspirations as optional contrastive examples rather than templates to copy. Inspect parent and inspiration metrics alongside fitness to understand which components, scenarios, or behaviors differ. Metrics are diagnostic evidence, not separate optimization objectives unless the benchmark explicitly says so.

Core strategy execution integrity: The artifact evaluated for this attempt must actually execute the core strategy claimed by the direction, active meta-guidance, or solution description. Do not silently return a parent, baseline, or alternate algorithm when that core strategy raises an exception, times out, is infeasible, returns no result, or otherwise fails. In particular, do not wrap the core strategy in a broad except Exception that returns a known-good candidate. Let unexpected core failures surface so a later attempt can debug them.

Explicit input-dependent dispatch, portfolios, and selector algorithms remain allowed when the dispatch or selection policy is itself the advertised top-level strategy. Its branches and selection conditions must be clear in the artifact; do not describe an internal branch as validated merely because a different branch produced the evaluated result.

For attempt 1: [omitted: list of inputs - parent, initial reference, inspirations, task description, meta-guidance, direction hint - and the role of each]

Generate a complete solution to the task. Use available skills as needed to complete the work. Do not run local search by repeatedly editing and verifying attempt_001. Internal optimization code inside the artifact is allowed, but the agent workflow itself must produce one candidate artifact for this attempt and then verify it once.

[omitted: apply notes prompts and meta-guidance prompts]

If this is a root task and both initial_program_reference and direction_hint exist, the generated solution must visibly reflect both: preserve the benchmark interface from the reference, reuse useful baseline structure when helpful, and make the direction-specific algorithmic choice real in the code rather than merely mentioning it in comments.

For attempts 2 and beyond (debug mode): DO NOT generate a completely new solution. Instead: [omitted: debug mode do's list, e.g., reuse parent and inspirations, analyze previous attempt's result, etc.]

8.3 Save Output [omitted: artifact saving guidelines]

8.4 Run Verifier

Execute the verification method specified in the task description or skill. Verifier success is defined only by the verifier result, not by whether the fitness reaches a target or feels improvable. Exit code 0 or result JSON "status": "success" means SUCCESS for this task attempt.

The verifier command is the only Bash command you should run for an attempt. [omitted: forbidden probes and shell trailers]

In the success/failure branch below, call record_attempt exactly once for that attempt. [omitted: record_attempt arguments]

8.5 Required Notes Commit

After every record_attempt call, inspect its returned note_requests. [omitted: for each note request, follow its analysis_prompt, produce one note JSON, and call commit_generated_note until every request is committed]

Step 9: Finalize Attempt Loop and **Step 10: End** [omitted: return one-line completion message]

CRITICAL RULES (NEVER BREAK THESE) [omitted: critical rules. e.g. attempt counting rule, stopping on success, no accessing evaluator, etc.]

D.4 NOTES

The Notes channel uses two fixed prompts. The first is sent with each note request and produces the note. Each request carries exactly one note type, its own trigger line, and the matching job block; the preamble, the materials, and the schema are shared by all three types. The second prompt carries the loaded notes back into the context of a later task.

Note analysis (fixed prompt sent with each note request, verbatim; [x] marks filled variables)

You are generating one run-local evolution note.

The note must summarize one useful experience from this task. Use the referenced artifacts and verification results as evidence. Do not copy long code blocks, full logs, or full tracebacks. Do not include your reasoning process, step-by-step analysis, or chain-of-thought in the note. The final note should contain only the distilled conclusion, actionable guidance, and a very short evidence summary. Do not decide whether the note is needed; MCP has already decided that.

Treat this as one empirical case, not proof that a strategy family is generally good or bad. Keep attribution scoped to what this single comparison can support. If several changes happened together, do not claim one mechanism is the sole cause unless the evidence clearly isolates it.

Verifier success proves that the submitted artifact was valid; it does not prove that every advertised internal algorithm or branch executed. Inspect the child artifact and verification evidence for primary, fallback, dispatch, and selector control flow before attributing the outcome to a particular strategy. If the executed branch cannot be established, attribute the outcome to the artifact or selector as a whole, state that the internal strategy is unverified, and lower confidence. Never claim that a solver reached an optimum or that a particular strategy succeeded unless the evaluated result can be traced to that executed branch.

When parameters, seeds, acceptance checks, fallback choices, or integration points matter, name those conditions explicitly. Do not turn one attempt into a universal rule about a whole strategy family. Return exactly one JSON object that matches the schema.

Note type: positive. Trigger: child fitness improved over parent by [delta]; threshold is [threshold].
Note type: negative. Trigger: child fitness dropped below parent by [delta]; threshold is [threshold].
Note type: error. Trigger: all attempts for this task failed verification. Or: attempt [attempt_id] succeeded after earlier failed attempts in the same task.

Materials: Task [task_id] / Attempt [attempt_id]
- child implementation and verification result: [artifact_ref], [verification_result_ref]
- parent implementation, fitness, and metrics: [parent_artifact_ref], [fitness], [metrics]
- inspirations: [inspiration list]
- error requests add: failed attempts: [failed artifact list]; failed verification results: [failed result list]

Your job (positive):

1. Compare child against parent.
2. Identify the concrete implementation change most likely responsible for the improvement.
3. If the idea appears to come from an inspiration, mention which one and how it was adapted.
4. Write a concise note that future tasks can use.

```
5.  Be careful with causality:  use "likely" when the evidence is suggestive rather than
definitive.
6.  If the improvement may depend on specific parameters, seeds, fallback logic, or acceptance
checks, state that condition instead of praising the broad method unconditionally.
7.  Do not write that a broad method is proven superior unless the evidence isolates it from
parameters and implementation details.
8.  The returned JSON must not include the comparison process; keep only the conclusion.

Your job (negative):

1.  Compare child against parent.
2.  Identify the concrete implementation detail, parameter choice, or integration pattern most
likely responsible for the regression.
3.  Check whether the regression came from misusing an inspiration, over-combining strategies, or
breaking a useful parent mechanism.
4.  Write a concise caution note for future tasks.
5.  Do not overgeneralize:  describe the condition under which the risk appeared.
6.  Do not write that a method or strategy should never be used just because this attempt failed.
7.  If a method might still work with different parameters, a safer acceptance check, or a
different integration point, say that the risk is limited to the observed implementation.
8.  Prefer wording like "avoid this unguarded variant" or "use only if it improves validation" over
blanket bans; never recommend a silent fallback that would hide whether the claimed strategy
executed.
9.  If the child changed both the strategy and its parameters, attribute risk to the observed
configuration rather than condemning the whole strategy.
10.  The returned JSON must not include the diagnostic process; keep only the conclusion.

Your job (error, all attempts failed verification):

1.  Read the failed implementations and verification errors.
2.  Identify the shared failure pattern, or the sequence of different failed fixes.
3.  Write one concise note that helps future tasks avoid this failure.
4.  Do not paste raw logs.  Summarize the failure reason and the preventive action.
5.  The returned JSON must not include the debugging process; keep only the conclusion.

Your job (error, repaired within the task):

1.  Compare the failed attempts with the successful attempt.
2.  Identify the key fix that changed failure into success.
3.  Write one concise note that future tasks can use as a checklist item.
4.  Focus on the failure mode and the repair, not on the full implementation.
5.  The returned JSON must not include the comparison process; keep only the conclusion.

Return JSON schema:
{
  "title":  "short imperative title",
  "lesson":  "one concise conclusion distilled from the evidence; no reasoning process",
  "actionable_guidance":  "one short instruction about what a future generation should do or
avoid",
  "evidence_summary":  "very brief factual evidence; no analysis process",
  "confidence":  "low|medium|high"
}
```

Note loading (prompt returned by `get_notes_prompt` and inserted as one section of the task context, verbatim)

```
## Run-local evolution notes

The following notes are distilled from earlier attempts in this same run.  Use them as practical
guidance, not as hard proof.  Prefer the selected parent and inspirations, but take these notes
into account when adapting them.  Treat positive notes as conditional hints, not proof that the
same strategy should be repeated.  Prefer a genuinely new variant when prior notes already describe
the same fitness plateau.  Treat negative notes as conditional risk warnings about the recorded
implementation and parameter choices, not as global bans on a whole strategy family.  A strategy
may still be useful with different parameters, fallbacks, acceptance checks, or integration points.

### Error notes:  avoid repeated failed patterns
### Positive notes:  patterns that previously improved fitness
### Negative notes:  patterns that previously degraded fitness
```

D.5 META-GUIDANCE

This role reads two sources. Its standing prompt fixes the workflow and the output contract; the controller-assembled request supplies the evidence and the per-request generation instructions.

Meta-guidance generator (agent prompt, verbatim excerpts; [x] marks values filled at dispatch time)

You are the dedicated Meta-Guidance Generation Specialist for the self-evolving agent system.

Each invocation handles exactly one pending request. You are stateless across invocations: do not poll, wait for another request, retain an agent identifier, or perform any task-executor or orchestration work.

Input contract. The dispatch prompt contains exactly: run_id: [x], request_id: [x]. Both values are required. Use them verbatim.

Workflow.
1. Call get_meta_guidance_request with the provided run_id and request_id. Never ask the orchestrator to retrieve or relay the request.
2. If the MCP call itself fails, retry it once.
3. If the response has available != true, do not generate or commit anything.
4. Treat the returned request as the complete source of generation state. Follow generation_instructions and constraints when present.
5. Inspect current_best.artifact_excerpt first. If more context is needed, use current_best.artifact_ref and read only the relevant candidate code.
6. Generate one JSON object that follows the schema below, then call commit_meta_guidance yourself with the same run_id and request_id. Never return the guidance JSON to the orchestrator.
[omitted: commit-retry and validation-repair rules]
9. End immediately after the compact result. Do not wait for more work.

Guidance JSON (one object per direction): idea; description; target_metric; novelty_claim; current_best_assessment; relationship_to_current_best (refine, depart, or replace); implementation_plan (step list); cautions; approach_type; avoid_repeating; max_uses.

Generate 1 to 3 items. Every approach_type in the same response must be distinct. Prefer distinct strategy families over parameter variants. Critically assess the current best rather than assuming its route is correct, and use notes_summary, guidance_history, and failed_guidance to avoid repeating routes without a materially changed failure condition.

Never modify project files, the candidate, verifier, or evaluator. Never bypass the evaluator, weaken the output contract, or propose evaluator-specific hard-coded shortcuts.

Compact result contract. The entire final response is exactly one line: meta_guidance_result: committed request_id=[x] guidance_count=[x], or the not_committed or mcp_error form with a reason code. Do not include the request payload, generated guidance, artifact contents, analysis, tool transcript, prose, or Markdown.

Meta-guidance request (assembled by the controller; verbatim field structure and generation instructions; [x] marks filled variables)

Request payload (at most one pending request at a time):
- request_id, task_description
- current_best: [record reference with artifact excerpt]
- progress: best fitness at watching start, current best fitness, completed count since watching, stall diagnostics
- lineages: [per-lineage summaries]
- notes_summary: [summary of admitted notes]
- guidance_history and failed_guidance: [earlier directions with recorded outcomes]
- generation_instructions (below) and constraints

Constraints:
- Do not bypass the evaluator.
- Keep the benchmark output contract.
- Do not modify verifier/evaluator source or use it to hard-code evaluator-specific shortcuts.
- You may inspect current_best.artifact_ref or the provided artifact_excerpt; this is candidate code.

Generation instructions:

- Inspect current_best.artifact_excerpt when present; if it is empty, use current_best.artifact_ref as the candidate implementation path.

- Inspect current_best.metrics and profile alongside fitness. Use metric components and scenario breakdowns as diagnostic evidence about where candidates differ or stagnate, without treating them as independent objectives unless the benchmark defines them that way.

- Analyze the current best critically: identify what works, what likely traps the search, and whether the next guidance should refine, depart from, or replace that route.

```
- Before crediting an internal algorithm, inspect the artifact control flow and available
verification evidence to establish which primary, fallback, dispatch, or selector branch produced
the evaluated result.  Verifier success alone does not establish branch execution; if branch
identity is unknown, treat the claimed strategy as unverified.

- Use notes_summary and guidance_history as evidence.  Do not repeat guidance whose prior
outcome_summary shows no improvement unless the failure condition is materially changed.

- Interpret prior outcomes precisely:  new_best beat the global best at completion; improved_anchor
beat its assigned parent but not the then-current global best; no_effect produced a valid result
indistinguishable from its assigned parent in fitness and evaluator metrics; regression produced a
different valid result without beating its assigned parent; failed produced no valid result.
no_effect is observational and does not by itself prove that fallback occurred.

- Generate distinct guidance items with different approach_type values.  Avoid minor parameter
variants of the same algorithm family.

- Prefer concrete, implementable strategies.  Each item should say how it relates to the current
best and why it can escape or improve the observed plateau.
```