META-REFLECTIVE EVOLUTION EFFECTIVELY IMPROVES PRODUCTION AI SYSTEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM)-guided evolutionary optimization has successfully improved prompts, programs, and even entire AI systems. Most optimization methods select a candidate from the search frontier, then mutate it in one LLM pass based only on its failures. This focus on the frontier can stall on production AI systems, where isolated failures reveal little about complex, hidden interactions. We introduce CoCoEvolve, an evolutionary optimizer built around what we call *meta-reflection*, wherein an agentic mutator selectively retrieves evidence from the entire evolutionary history and probes the live system before verifying and proposing mutations. We apply CoCoEvolve to optimize three classes of production AI systems: data agents, LLM-powered data pipelines, and data engineering workflows. On DABStep, CoCoEvolve tops the public leaderboard, reaching 92.8% held-out hard accuracy compared to 78.4% for Meta-Harness and 30.4% for GEPA under matched conditions. Across six LLM data pipeline workloads, it outperforms every evaluated baseline. On data-eng-bench, CoCoEvolve lifts test-set solve rate from 59.1% to 77.4%, past the best baseline of 73.6%. It also has 11pp higher accuracy than GEPA on HotpotQA while remaining competitive with specialized optimizers on established mathematical and systems benchmarks. Matched ablations confirm that each component of CoCoEvolve contributes to these gains, demonstrating the value of meta-reflection for optimizing production AI systems.

1 INTRODUCTION

Large language models (LLMs) are increasingly used as semantic mutation operators: given a candidate prompt or program and its evaluation feedback, an LLM proposes a revised candidate. This approach, exemplified by evolutionary optimizers like AlphaEvolve and OpenEvolve, has produced breakthrough improvements in prompts, algorithms, and other text-based artifacts because LLMs can make semantically meaningful changes that conventional random mutation cannot (Novikov et al., 2025; Sharma, 2025). However, most LLM-guided evolutionary optimizers select a single parent, use its failures to propose a new candidate in one LLM call, and only then evaluate the result.

This parent-local, one-shot interface is effective when the failure makes both the defect and its repair apparent. It falls short for production AI systems, where isolated failures can reveal little about interacting causes and relevant evidence may be distributed across the search history. First, the selected parent's failure may not reveal the root cause, and a locally beneficial edit may regress capabilities elsewhere in the search history. Second, the mutator cannot interact with the target environment, test hypotheses, or revise its candidate in response to intermediate results. Search can trade one solved task for another, revisit exhausted strategies, or plateau (Zhou et al., 2026).

We introduce CoCoEvolve, an evolutionary optimizer built around *meta-reflection*: (i) selectively retrieving evidence from the accumulated search history beyond the selected parent's failure and (ii) actively probing the target environment while constructing and validating a mutation. To support meta-reflection, CoCoEvolve uses a coding-agent mutator that can inspect the parent, run focused experiments, revise its implementation, and validate the result before committing a child. It accesses evolutionary history through both a compact summary and structured queries, allowing it to retrieve

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.

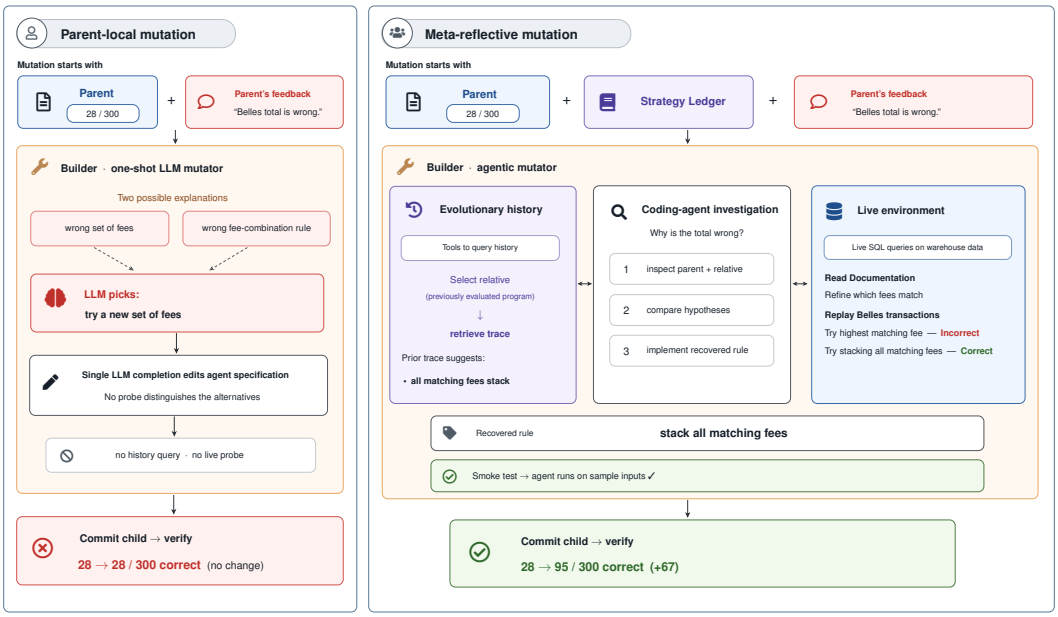

Figure 1: **Left:** given under-determined feedback, the parent-local mutator guesses a repair in one call and makes no progress. **Right:** a meta-reflective mutator queries the evolutionary history and reads the trace of a previously evaluated program to determine that *all matching fees stack*. It also reads documentation to refine which fees match, and synthesizes the two observations. It tests the resulting hypothesis before commit and raises training accuracy from $28/300$ to $95/300$.

prior programs that solved a specific task, inspect the mutation that restored a lost capability, or identify related tasks that have historically regressed together.

Figure 1 illustrates meta-reflection for a DABStep data agent (Egg et al., 2025), where payment-processing rules must be inferred across documentation and live data. Here, a one-shot, parent-local mutation doesn't make progress, whereas CoCoEvolve's meta-reflective mutation retrieves missing evidence, validates the corresponding repair, and succeeds. This behavior repeats with *Lockpicking with Interference*, a synthetic stress test where every leading baseline plateaus below perfect accuracy within 50 iterations, while CoCoEvolve reaches 100% in 6 iterations (Appendix H.1).

We evaluate CoCoEvolve primarily on production AI systems, where correctness depends on interactions with live systems rather than prompt text alone. These include agent specifications coupled to tools and semantic data models, SQL pipelines that invoke LLMs to process textual data (Liskowski et al., 2025), and data-engineering workflows (dbt Labs, 2026). We additionally evaluate standard prompt, mathematical, and systems optimization benchmarks from prior work to test whether the method remains competitive when interaction with the environment is limited or unnecessary.

CoCoEvolve optimizes an agent specification on DABStep (Egg et al., 2025), reaching 92.8% held-out hard accuracy, versus 78.4% for an interactive Meta-Harness baseline and 30.4% for GEPA (Lee et al., 2026; Agrawal et al., 2026b); it also tops the validated public leaderboard.[1] CoCoEvolve also improves over the strongest baseline on six LLM-powered data processing pipeline workloads by up to 10 pp. On held-out HotpotQA, it gains 11 pp over GEPA and 43 pp over the seed (Yang et al., 2018; Agrawal et al., 2026b). On data engineering, it achieves the strongest overall performance among the evaluated evolutionary methods and leads on held-out data-eng-bench (Snowflake AI Research, 2026), solving 77.4% of tasks. Across mathematical and systems optimization objectives, it is competitive with leading evolutionary methods such as AdaEvolve (Cemri et al., 2025), with the largest improvements on objectives exhibiting extended search plateaus. Matched ablations confirm that each component of CoCoEvolve contributes to these gains.

[1] At submission, this score has not been externally validated; we have contacted the DABStep maintainers but have not received a response.

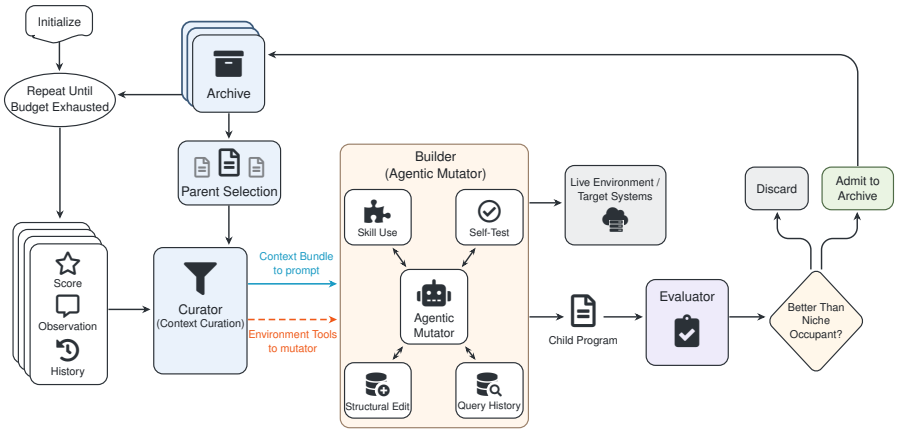

Figure 2: CoCoEvolve proposes a new candidate in each iteration by selecting a parent from the archive, curating evidence from the search history to include in the prompt, and equipping a coding-agent mutator with on-demand tools for querying prior lineages and the live target environment. A coding agent investigates, edits, and self-tests a child; the evaluator scores it, records the resulting observations in the history, and, under the default MAP-Elites policy, either discards it or admits it to the archive if it improves its niche. This loop combines compact prompt-visible evidence with environment feedback while preserving diverse programs for future selection.

2 CoCoEvolve

We introduce CoCoEvolve, an evolutionary optimizer built around *meta-reflection* (Section 2.2). Figure 2 summarizes its evolutionary loop and Algorithm 1 in Appendix D specifies it in full.

2.1 Problem Formulation

We treat the object of optimization as a *program* p — code, a prompt, a configuration, an agent specification, or any other text-representable artifact — drawn from a program space $\mathcal{P}$. Let $\mathcal{T}$ denote the task space and $\mathcal{D}$ a distribution over it. An evaluator

$$f : \mathcal{P} \times \mathcal{T} \to \mathbb{R} \times \mathcal{O}, \qquad f(p, \tau) = \big(s(p, \tau),\ o(p, \tau)\big)$$

returns both a scalar score $s(p, \tau) \in \mathbb{R}$ and an *observation* $o(p, \tau) \in \mathcal{O}$: the full record produced while computing that score, including tool calls, intermediate outputs, executed queries, and error messages. The optimization goal is to find $p^* = \arg\max_{p \in \mathcal{P}} \Phi(p)$ within T iterations, where $\Phi(p) = \mathbb{E}_{\tau \sim \mathcal{D}}[s(p, \tau)]$ represents the aggregate fitness of a program.

CoCoEvolve pursues this goal via an iterative evolutionary loop. After iteration $t-1$, it maintains an archive $\mathcal{A}_{t-1}$ of previously evaluated candidate programs. At iteration t, the harness selects a parent program $p'_t \in \mathcal{A}_{t-1}$ and produces a child p_t. Most prior methods condition mutation only on the selected parent p'_t and its observations; CoCoEvolve instead uses the *full search history* accumulated so far, denoted by

$$\mathcal{H}_{t-1} = \{(p_i, \mathcal{T}_i, S_i, O_i)\}_{i<t},$$

Here, $\mathcal{T}_i \subseteq \mathcal{T}$ is the subset selected to evaluate p_i, with $S_i := \{s(p_i, \tau) \mid \tau \in \mathcal{T}_i\}$ and $O_i := \{o(p_i, \tau) \mid \tau \in \mathcal{T}_i\}$ denoting the resulting sets of scores and observations. This history supports the mutation policy developed next: context curation exposes selected evidence from $\mathcal{H}_{t-1}$, environment augmentation provides tools for retrieving additional evidence on demand, and the coding-agent mutator uses both to construct the child.

2.2 Meta-Reflection

We define *meta-reflection* as reasoning beyond one parent's observations by selectively retrieving historical evidence from the accumulated search history and actively acquiring feedback from the target environment.

Formally, a meta-reflective mutator receives two complementary inputs. First, a compact context bundle c_t, curated from the search history, is injected directly into the mutator's prompt. Second, an environment $\mathcal{E}_t$ provides tools that the mutator invokes on demand. It augments a base environment $\mathcal{E}_0 \subseteq \mathcal{E}$ of execution, validation, and target-system tools with selective access to the search history.

$$c_t = \mathrm{Curate}(\mathcal{H}_{t-1}) \in \mathcal{C}, \qquad \mathcal{E}_t = \mathrm{Augment}(\mathcal{E}_0; p'_t, \mathcal{H}_{t-1}) \subseteq \mathcal{E}.$$

A coding-agent mutator then performs meta-reflection by reasoning over c_t and gathering additional evidence from $\mathcal{E}_t$ as needed while constructing and validating the child p_t before committing it.

$$p_t \sim \mathrm{MetaReflect}(p'_t, c_t; \mathcal{E}_t).$$

Relation to prior reflective mutation. This interface recovers prior reflective mutation as the special case in which $c_t = O'_t$,and $\mathcal{E}_t = \mathcal{E}_0 = \emptyset$: each revision conditions only on that trace or *side information* (Agrawal et al., 2026a). Additionally, these reflective methods implement mutation with a single model call over this preassembled evidence.

In CoCoEvolve, these interface components take the following concrete forms.

Ledger to gauge stagnation. CoCoEvolve's Curate operator constructs a compact ledger of recent strategies and their outcomes (Appendix D.1). The agent uses this ledger to assess stagnation and choose whether to continue the current strategy, adopt a newly proposed one, or revert to undirected mutation. AdaEvolve makes a related search-level decision through a fixed trigger, invoking tactic generation when improvement falls below a threshold (Cemri et al., 2025). CoCoEvolve instead lets the same class of model that performs mutation reason over concurrently tracked strategies and make this decision from the search trajectory, without a hand-tuned stagnation trigger.

Live environment interaction. The base environment $\mathcal{E}_0$ provides domain-specific tools for execution, testing, and target-system queries. For example, in mathematical and systems optimization, $\mathcal{E}_0$ exposes scratch runners that reveal syntax and execution errors. In DABStep agent optimization, it provides tools for SQL execution, scratch-agent deployment, target-question tests, and deployment checks. This feedback lets the agent diagnose and repair failures before committing a child to outer-loop evaluation. Appendix D.2 specifies the interfaces, and Appendix E.2 compares static, edit-only agentic, and self-validating mutation.

Tools to query interlinked history. Beyond these domain capabilities, $\mathrm{Augment}$ forms $\mathcal{E}_t$ by adding on-demand tools that selectively query the interlinked program archive as evidence is needed. These tools expose per-task lineage: which program first solved a task, whether a later mutation lost that capability, and which tasks tend to co-regress. The agent can therefore retrieve the exact diff that last restored a lost capability or vet a candidate child against known co-regressions before committing. Appendix D.2 and Table 6 specify the complete query interface, while Appendix E.3 isolates the contributions of the injected ledger and history-query tools.

After validation, we return the generated child p_t to the outer search loop, where f scores it, $\mathcal{H}_t$ records it, and the search policy governs archive admission and subsequent parent selection.

2.3 Search Policy

By default, CoCoEvolve uses standard MAP-Elites parent selection (Sharma, 2025; Mouret & Clune, 2015). For objectives scored over a pool of questions or test cases, we introduce a novel budgeted ℓ^p roster that prioritizes complementary coverage; Appendix D.5 defines it, and Appendix E.1 compares it with MAP-Elites, best-only selection, and pareto-frontier sampling.

Beyond parent selection, an optional similarity-informed task-selection policy uses proxy scores to reduce evaluation cost and focus mutation on high-information tasks.

Similarity-guided sparse evaluation and observation curation. Prior methods typically expose the mutator to observations from a random task minibatch. When the task pool V is large or individual evaluations are costly, however, exhaustively scoring every candidate is impractical. On DABStep, for example, each agent query takes roughly 3–10 minutes, so scoring the full 300-question search pool requires 15–50 agent-hours before parallelism. At the same time, tasks need not be independent: a repair that equips a candidate to solve one task may transfer to other tasks requiring the same capability. This motivates sparse, targeted evaluation: score a bounded subset, infer likely

performance on the remaining tasks, and direct the next mutation toward an unresolved target with broad expected transfer.

CoCoEvolve realizes this idea with a semantic task graph. Let $\kappa : V \times V \to [0, 1]$ measure task similarity. Using only evaluated tasks, it forms a Nadaraya–Watson kernel estimate of the candidate's score on τ (Nadaraya, 1964; Watson, 1964):

$$\widehat{s}(p, \tau) = \frac{\sum_{\tau' \in \text{tested}(p)} \kappa(\tau, \tau') s(p, \tau')}{\sum_{\tau' \in \text{tested}(p)} \kappa(\tau, \tau')},$$

so evaluated neighbors provide a surrogate expected score for tasks not yet tested. These surrogate scores also determine where mutation effort is likely to be most informative. Each task not solved by the parent receives an information-gain score:

$$\text{gain}(\tau) = \big(1 - \widehat{s}(p, \tau)\big)\delta(\tau) \sum_{\tau' \in V} \kappa(\tau, \tau').$$

Here, $\delta(\tau)$ downweights repeatedly unsuccessful targets; sampling by gain then focuses c_t on likely failures with broad semantic reach. This policy is a task-level counterpart to AdaEvolve's island-level Dynamic Exploration Intensity (Cemri et al., 2025): it directs scarce evaluation and mutation effort toward failures with the greatest estimated transfer. Appendix D.3 details the policy, Table 7 records where it is active, and matched DABStep ablations showcase its value (Appendix E.4).

3 Experiments

Where applicable, we use train/validation/test splits: CoCoEvolve sees training examples and validation scores, but not validation instances, and final results use held-out test data. We evaluate CoCoEvolve on four task categories: agent optimization, AI Functions, data engineering, and mathematical/system/prompt optimization. Appendix D specifies parent selection, Appendix E gives mechanism ablations, and Appendix J details math and systems evaluators.

3.1 Agent Optimization

Managed enterprise agents orchestrate semantic data access, retrieval, code execution, and APIs; their reliability depends on the instructions, tools, and data models that guide them (Snowflake Inc., 2026; Databricks, 2024; Amazon Web Services, 2023).

Setup. We optimize *agent specifications* $p \in \mathcal{P}$ comprising prompts, tool schemas, and planning instructions. On DABStep's 450 payment-processing questions, which require multi-step reasoning over more than nine join dimensions and wildcard joins, CoCoEvolve, GEPA, and Meta-Harness start from the same specification and run for 100 iterations; the benchmark authors report 14.55% state-of-the-art accuracy. CoCoEvolve uses information-gain curation (Section 2.3) because evaluations are costly. Appendix G.1 details the setup.

Results. Table 1 compares CoCoEvolve, Meta-Harness, GEPA, and two unoptimized baselines. CoCoEvolve reaches 95.24% hard accuracy on the official suite, topping the validated public leaderboard; Appendix G gives the full comparison and scoring protocol.

	Train (300)			Test (150)		
Method	Easy	Hard	Total	Easy	Hard	Total
Stock Cortex Agent (baseline)	57.4	13.8	20.7	56.0	8.0	16.0
Snowflake Cortex Code Agent	72.3	26.5	33.7	80.0	21.6	31.3
GEPA	91.5	39.1	47.3	84.0	30.4	39.3
Meta-Harness	87.2	76.7	78.3	80.0	78.4	78.7
CoCoEvolve	**89.4**	**96.4**	**95.3**	**84.0**	**92.8**	**91.3**

Table 1: DABStep accuracy (%) on split (train: 47 easy / 253 hard; test: 25 easy / 125 hard).

Analysis. Under matched splits and budgets, CoCoEvolve outperforms GEPA and Meta-Harness even when the latter is equipped with self-verification and self-testing tools that provide a comparable pre-commit validation surface (Appendix G.1); three mechanisms explain this gap.

Live environment interaction grounds mutation in execution feedback. DABStep intentionally tests domain rules whose downstream implications must be inferred or composed across documentation and data, a failure mode emphasized by the benchmark authors (Egg et al., 2025). The live environment turns this ambiguity from guesswork into testable hypotheses. The CoCoEvolve coding-agent mutator repeatedly formed fee interpretations, tested them through live SQL against a known merchant total, and revised its diagnosis. This live feedback exposed two hidden logic errors and enabled a verified vectorized fee engine, producing the run's largest single gain, from 28 to 95 correct answers (Appendix G.4).

Curated environment tools compress otherwise lengthy investigations. The base environment exposes scripts, managed skills, and developer tools that encode reusable investigation and validation workflows, allowing the mutator to invoke proven procedures instead of reconstructing them across mutation cycles. During the fee-engine revision, the mutator first checked its raw SQL, then used an existing testing skill to deploy the draft as a live scratch agent and exercise multiple question families as self-validation before commit. Each passed in a single tool call, whereas the prior specification failed after roughly 40 tool calls, allowing the mutator to invest its budget in structural reasoning rather than verification plumbing (Appendix D.2).

Structural changes are more robust than prompt changes. Reflective prompt evolution is effective (Agrawal et al., 2026b), but a single model call over preassembled evidence offers no path to explore and validate changes to the underlying tool architecture. In a matched DABStep run, GEPA revised the instructions but left the default single-row fee calculator tool unchanged. Its evolved agent therefore aggregates 26 transaction fees through 26 tool calls and asks the LLM to sum the outputs in context, producing arithmetic drift such as 19.51 instead of 19.36. Such remedies require live environment interaction: inspecting schemas, executing draft SQL, and validating outputs against known totals before commit. By testing structural edits in situ, CoCoEvolve discovered two independently verified remedies: a vectorized fee-engine stored procedure and a dynamic table that materializes per-transaction fees for downstream SQL aggregation. These solutions shift computation from fallible in-context arithmetic into deterministic system components, illustrating how coding-agent mutation can advance where prompt-only search plateaus (Appendix G.4).

3.2 AI FUNCTIONS

Cloud data platforms expose *AI functions*, SQL operators that embed LLM inference for generation, classification, and extraction (Liskowski et al., 2025; Databricks, 2023; Microsoft, 2024). Because workloads compose them with common-table expressions and joins, optimization may revise prompts, operator composition, checks, or intermediate transformations. We therefore search executable SQL programs $p \in \mathcal{P}$, allowing CoCoEvolve to validate both language-level and structural changes against the SQL engine.

Setup. Each task starts from a minimal SQL program p_0 containing one generative AI-function call, which the mutator may rewrite into a multi-stage, multi-operator pipeline. *Lockpicking with Interference* scores exact match on 50 strings governed by hidden rules (Appendix H.1). Beyond this synthetic diagnostic, we evaluate classification on Reddit comments (`go_emotions`) and legal documents (`lex_glue`) (Demszky et al., 2020; Chalkidis et al., 2022). We also study financial reasoning over tables, prose, and dialogue (`finqa`, `convfinqa`) and normalization of noisy social-media tokens (`multilexnorm`) (Chen et al., 2021; 2022; van der Goot et al., 2021). Because evaluation is inexpensive, we omit similarity-informed targeting and let the mutator execute and repair candidate pipelines before committing. **Results.** Table 2 shows that CoCoEvolve leads all six tasks. Its largest gains are 10 pp on Lockpicking, exactly recovering its hidden rules, and 7.7 pp on `multilexnorm`, whose context-dependent patterns echo Lockpicking's latent-rule challenge.

	claude-sonnet-5					
Method	lockpicking_6	go_emotions	finqa	convfinqa	lex_glue	multilexnorm
GEPA	0.820	0.425	0.800	0.920	0.735	0.510
AdaEvolve	0.900	0.366	0.740	0.885	0.695	0.470
OpenEvolve	0.540	0.375	0.770	0.900	0.645	0.470
CoCoEvolve	**1.000**	**0.444**	**0.813**	**0.953**	**0.737**	**0.587**

Table 2: AI-function benchmark performance.

Analysis. *On-demand history queries help escape local minima.* When a highly coupled objective plateaus, reasoning only over the current parent's failures may not expose a route out. AdaEvolve's adaptive scheduler illustrates this limitation on Lockpicking (Cemri et al., 2025): it first reaches 90% at iteration 34; over the remaining 16 iterations, two stagnation-triggered paradigm generations propose targeted worked examples and exhaustive condition matrices for the five residual failures, yet its best score never moves. CoCoEvolve reaches exact recovery at iteration 6, before AdaEvolve's first paradigm generation at iteration 10, by using on-demand history tools to query evidence across lineages rather than repeatedly reframing the same residual errors (Appendix H.1).

Self-validation prevents wasted mutation cycles. Without a cheap validator, mutation can waste substantial search budget on SQL programs that never execute. On `go_emotions`, GEPA repeatedly placed `LATERAL FLATTEN` inside a UDF; because each compiler error collapsed to a zero score without diagnostic feedback, broken variants consumed up to 71% of the budget. Alternatives such as OpenEvolve mitigate evaluation cost with cascading evaluators that stop after compilation failure, but each rejected candidate still consumes an evolutionary iteration (Sharma, 2025). CoCoEvolve instead executes drafts on a cheap sample and repairs syntax failures within the same multi-turn mutation before commit. Self-validation thus converts compiler errors from terminal scores into actionable context, reserving outer-loop evaluation for executable programs.

3.3 Data Engineering

Data engineering transforms raw warehouse data into trusted, queryable models. Agents often fail to construct or repair transformations because the necessary context spans warehouse details, implicit relationships, and domain-specific business rules. We optimize context-curation pipelines $p \in \mathcal{P}$ that produce a grounded knowledge graph (KG) and executable adherence tests, both supplied to a fixed downstream agent before it executes a transformation. Data-eng-bench comprises 103 tasks over 579 tables, 19 business schemas, and more than 8,000 columns (Snowflake AI Research, 2026).

Setup. The seed p_0 is one minimally prompted agent with lean human priors for three KG and adherence-test skills. The mutator may refine the prompt, introduce or extend skills, or reorganize the pipeline into a multi-agent protocol. Only this upstream pipeline evolves: its score is determined after the fixed downstream agent consumes the generated KG and tests, implements the transformation, and is graded by the benchmark verifier (Appendix I).

Results. Table 3 shows that CoCoEvolve raises test solve rate from 59.1% to 77.4%, above GEPA's 73.6% and Meta-Harness and AdaEvolve's 71.7%. It ties the best dev result at 68.0%, exceeding the baseline, AdaEvolve, and GEPA by 14.0, 8.0, and 6.0 pp.

Table 3: Solve rate (%) on the dev and test splits of **data-eng-bench**.

Dataset	`claude-sonnet-5`									
	Baseline		Meta-Harness		GEPA		AdaEvolve		CoCoEvolve	
	Dev	Test	Dev	Test	Dev	Test	Dev	Test	Dev	Test
data-eng-bench	54.0	59.1	68.0	71.7	62.0	73.6	60.0	71.7	68.0	**77.4**

Analysis. *Meta-reflection preserves globally relevant skills.* During evolution, an earlier lineage introduced a skill to resolve a recurring dbt configuration error. A later hillclimb on an unrelated target dropped the skill. When the failure re-appeared, CoCoEvolve's mutator retrieved the archived mutation plan via the ledger that first named the fix and re-implemented the skill, rather than re-deriving it from scratch. We observe that the mutator matches returning failures against past resolutions in the evolution ledger, linking current errors to the earlier mutations that fixed them. Meta-reflection thus allows for capabilities the search has already learned to remain visible to the mutator even when not directly present in the parent's trace.

3.4 Mathematical, Systems, and Prompt Optimization

These experiments ask whether CoCoEvolve can match specialized optimizers on established mathematical and systems benchmarks and whether its benefits persist on HotpotQA, AIME, and IF-

	Mathematical Optimization												ADRS Systems									
	Circle Packing (Square) ($N = 26$)		Circle Packing (Rect) ($N = 21$)		Heilbronn (Triangle) ($N = 11$)		Heilbronn (Convex) ($N = 13$)		MinMax Distance ($d = 3, N = 14$)		Signal Processing		Cloudcast (↓)		EPLB		Prism		LLM-SQL		TXN	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human/SOTA	–	2.636	–	2.366	–	0.037	–	0.031	–	0.240	–	0.718	–	626.200	–	0.127	–	21.890	–	0.692	–	2,725
Backbone: Sonnet-5																						
OpenEvolve	**2.636**	**2.636**	**2.366**	**2.366**	0.032	0.035	0.028	0.029	0.238	0.240	0.650	0.673	660.693	660.689	0.127	0.127	26.236	26.256	0.713	0.726	3,945	4,016
GEPA	2.602	**2.636**	2.365	**2.366**	**0.036**	**0.037**	0.030	**0.031**	**0.240**	**0.240**	0.603	0.631	621.090	618.080	0.133	0.145	26.256	26.256	0.524	**0.752**	3,874	4,000
AdaEvolve	2.633	**2.636**	2.364	**2.366**	0.035	**0.037**	0.029	0.031	**0.240**	**0.240**	0.747	0.766	618.097	618.088	0.145	0.145	**26.391**	**26.660**	0.724	0.733	4,115	4,132
CoCoEvolve	2.635	**2.636**	2.363	**2.366**	0.035	**0.037**	**0.031**	**0.031**	**0.240**	**0.240**	**0.766**	**0.784**	**618.063**	**618.060**	**0.145**	**0.146**	26.256	26.256	**0.728**	0.732	**4,292**	**4,348**
Backbone: GPT-5.6-terra																						
OpenEvolve	**2.636**	**2.636**	2.365	2.366	0.030	0.031	0.025	0.027	0.240	0.240	0.565	0.608	618.096	618.093	0.127	0.127	26.076	26.152	0.724	0.730	4,107	4,184
GEPA	2.634	**2.636**	2.364	2.366	**0.035**	0.035	**0.026**	**0.028**	**0.240**	**0.240**	0.633	0.661	**618.081**	**618.070**	0.127	0.127	26.259	26.264	0.719	0.724	**4,268**	**4,348**
AdaEvolve	2.633	2.634	**2.366**	**2.366**	0.035	0.035	0.024	0.024	0.240	0.240	**0.792**	**0.813**	618.085	**618.070**	0.142	**0.145**	25.674	25.734	0.717	0.724	4,144	4,167
CoCoEvolve	2.634	**2.636**	2.364	2.366	0.034	**0.037**	0.025	0.027	**0.240**	**0.240**	0.756	0.805	618.082	618.071	**0.145**	0.145	**26.338**	**26.501**	**0.728**	**0.730**	4,250	4,329

Table 4: Mathematical and ADRS systems benchmarks. Higher is better except for Cloudcast. Human/SOTA is the maximum published human or automated result reported by AlphaEvolve and AdaEvolve. Full-precision means, standard deviations, and best values appear in Tables 18 and 19.

Bench from GEPA's benchmark suite when live interaction is unavailable. Within each task, all arms share the task definition, initial artifact, backbone, evaluator, and data split where applicable, enabling matched end-to-end comparisons across executable programs and prompt-only systems.

Setup. On math and systems tasks, each evaluator returns one scalar for the complete program. A lightweight reflector detects stagnation and chooses whether to continue, request a new direction, or revert to undirected mutation. When needed, a distinct proposer queries techniques from any lineage while preserving the mutator's turn for implementation (Appendix F). AIME and IFBench use GPT-5.6-luna because Sonnet-5 saturates AIME at the seed, leaving no optimization signal. The main table omits standard deviations for readability; bold and underlining mark the best and second-best results within each backbone. Appendices J and K.1 provide full statistics and prompt protocols.

Results. In Table 4, CoCoEvolve is best or tied best in 25 of 44 Avg/Best cells—13 of 24 math and 12 of 20 ADRS cells—more than any other method. Its largest Sonnet-5 gains occur on plateau-prone Signal Processing, TXN, and Heilbronn Convex; on Signal Processing, it reaches 0.766 versus AdaEvolve's 0.747. Gains largely vanish on saturated objectives, while mixed GPT-5.6-terra results show backbone dependence. Appendix J.3 analyzes plateau escape, structural family switching, and saturation controls task by task. Table 5 shows CoCoEvolve leading GEPA and the seed on every HotpotQA and IFBench split, while AIME remains inconsistent.

	HotpotQA (Sonnet-5)			AIME (luna)			IFBench (luna)		
Method	Train	Val	Test	Train	Val	Test	Train	Val	Test
Baseline (unmodified seed)	0.393	0.313	0.363	$\mathbf{0.919 \pm 0.056}$	$\mathbf{0.882 \pm 0.026}$	$0.833 \pm 0.000^{\dagger}$	0.902 ± 0.003	0.864 ± 0.003	0.663 ± 0.027
GEPA	0.713	0.700	0.683	$\mathbf{0.919 \pm 0.078}$	0.830 ± 0.056	0.833 ± 0.058	0.891 ± 0.018	0.886 ± 0.009	0.683 ± 0.005
CoCoEvolve	**0.800**	**0.783**	**0.793**	$\mathbf{0.919 \pm 0.026}$	$\mathbf{0.882 \pm 0.046}$	$\mathbf{0.878 \pm 0.051}$	$\mathbf{0.927 \pm 0.010}$	$\mathbf{0.921 \pm 0.009}$	$\mathbf{0.705 \pm 0.024}$

Table 5: Prompt optimization on HotpotQA, AIME, and IFBench. HotpotQA reports a single Sonnet-5 run; AIME and IFBench report mean $\pm$ standard deviation over three evaluations of each final candidate using `openai-gpt-5.6-luna`; these repeats measure evaluation noise, not search variance. Bold marks column-best results that are robust to measurement noise; †the observed zero deviation is coincidental.

Analysis. Across the mathematical and systems benchmarks, CoCoEvolve recovers the gains of AdaEvolve's specialized three-level adaptive controller through meta-reflection over the search history, offering a more flexible route to stagnation-aware search.

The strategy ledger turns plateaus into new search directions. On Signal Processing, the global best remained at 0.7534 for 26 iterations even though the current parent and champion represented distinct Savitzky-Golay and Butterworth filter families. Reasoning over the strategy ledger, the lightweight reflector recognized a population-wide plateau and requested a new proposal rather than another local mutation or crossover. Using on-demand history queries to identify what the search had not tried, the separate proposer found an orthogonal family: an L1 trend filter formulated as a linear program. The mutator implemented this paradigm and raised the score to 0.7745 in one iteration. This trace illustrates how the strategy ledger and on-demand history tools separate stagnation detection, paradigm selection, and implementation: the search history reveals not only which ideas succeeded, but which structural alternatives remain unexplored.

On-demand history queries improve prompt optimization even without live environment interaction. On HotpotQA, CoCoEvolve reaches 0.793 held-out accuracy versus GEPA's 0.683 and the seed's 0.363, an 11 pp margin in GEPA's own setting. IFBench preserves the ordering CoCoEvolve $>$ GEPA $>$ seed on every repeat and split, although the held-out margin is narrower. AIME provides no reliable separation: all arms lie in the same 0.83–0.88 test band, while 17–27% of individual answers change correctness across repeated evaluations of an unchanged prompt. Together, these results isolate the value of on-demand history access for text-only artifacts, while AIME illustrates how stochastic evaluation can erase apparent search gains.

4 Related Work

Evolutionary Harnesses. LLM-guided evolutionary optimization shifts improvement from training-time parameter updates toward inference-time search, using LLMs as semantic mutation operators over prompts, algorithms, and programs. Building on quality-diversity search (Mouret & Clune, 2015), systems such as FunSearch, AlphaEvolve, and OpenEvolve apply this paradigm to scientific and algorithmic discovery (Romera-Paredes et al., 2024; Novikov et al., 2025; Sharma, 2025). GEPA enriches mutation with language-valued execution traces that expose reasoning, tool calls, outputs, and errors beyond scalar rewards (Agrawal et al., 2026b). Subsequent work broadens the optimized artifact space through unified declarative APIs (Agrawal et al., 2026a), allocates search adaptively (Cemri et al., 2025; Lange et al., 2025), and standardizes outer harnesses (Ursekar et al., 2026). For prompts and LM pipelines, EvoPrompt and Promptbreeder evolve prompts with LLM-driven operators (Guo et al., 2025; Fernando et al., 2024), MIPROv2 jointly searches instructions and demonstrations for multi-stage programs (Opsahl-Ong et al., 2024), and TextGrad backpropagates natural-language feedback through compound systems (Yuksekgonul et al., 2025). These methods generally retain a parent-local, one-shot mutation interface in which the harness preselects evidence and evaluates the proposed edit only after generation. Promptbreeder also evolves its mutation prompts, whereas CoCoEvolve changes the mutation interface itself through meta-reflection over the search history and target environment. Meta-Harness (Lee et al., 2026) similarly uses a coding-agent proposer with filesystem access to prior traces; CoCoEvolve combines compact curated context with structured, on-demand queries over evolutionary history and a live environment.

Context Curation Approaches. The efficacy of LLM agents relies heavily on rich, domain-specific context, such as organizational schemas (Döhmen et al., 2026) or dynamically constructed failure taxonomies (Cojocaru et al., 2026). Recent frameworks automate context engineering by treating contexts as evolving playbooks (Zhang et al., 2026) or co-evolving curation skills alongside artifacts (Ye et al., 2026). CoCoEvolve applies this principle to the optimization trajectory itself, distilling accumulated search history into a compact ledger.

Limitations of Evolutionary Approaches. AI-driven evolutionary frameworks exhibit systematic vulnerabilities, including hidden regressions (Zhou et al., 2026), overfitting to the initial task set under continual learning (Wang et al., 2026), and context costs that grow with the accumulated archive (Lee et al., 2026). CoCoEvolve addresses these failure modes with bounded, queryable access to historical co-regressions, helping the optimizer avoid cycling through brittle solutions that break previously solved cases.

5 Conclusion

We present CoCoEvolve, an evolutionary optimizer for production AI systems. By both probing the environment and querying an evolutionary ledger and prior lineages, CoCoEvolve's mutator extends parent-local reflection into *meta-reflection* over the search process, enabling it to diagnose failures, reconstruct hidden mechanisms, preserve complementary discoveries, and verify structural revisions before committing. Across domains, our results suggest that the combination of curating selected historical evidence, providing on-demand tools for probing the live environment and evolutionary history, and using a coding-agent mutator to reason over both offers an effective path to improving the prompts, code, tools, data models, and pipelines of real-world AI systems.

REFERENCES

Lakshya A. Agrawal, Donghyun Lee, Shangyin Tan, Wenjie Ma, Karim Elmaaroufi, Rohit Sandadi, Sanjit A. Seshia, Koushik Sen, Dan Klein, Ion Stoica, Joseph E. Gonzalez, Omar Khattab, Alexandros G. Dimakis, and Matei Zaharia. optimize_anything: A universal API for optimizing any text parameter. In *ACM Conference on AI and Agentic Systems (CAIS '26)*, 2026a. URL https://arxiv.org/abs/2605.19633.

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

Amazon Web Services. Agents for Amazon Bedrock. https://aws.amazon.com/bedrock/agents/, 2023. Accessed: 2026-08-09.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. AdaEvolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2025. URL https://arxiv.org/abs/2602.20133.

Ilias Chalkidis, Abhik Jana, Dirk Hartung, Michael Bommarito, Ion Androutsopoulos, Daniel Martin Katz, and Nikolaos Aletras. LexGLUE: A benchmark dataset for legal language understanding in English. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4310–4330, 2022. URL https://aclanthology.org/2022.acl-long.297/.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, and William Yang Wang. FinQA: A dataset of numerical reasoning over financial data. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3697–3711, 2021. URL https://aclanthology.org/2021.emnlp-main.300/.

Zhiyu Chen, Shiyang Li, Charese Smiley, Zhiqiang Ma, Sameena Shah, and William Yang Wang. ConvFinQA: Exploring the chain of numerical reasoning in conversational finance question answering. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 6279–6292, 2022. URL https://aclanthology.org/2022.emnlp-main.421/.

Andrei Cojocaru, Mert Cemri, Melissa Pan, Shu Liu, Shubham Agarwal, Alex Krentsel, Jay Tang, Kannan Ramchandran, Joseph Gonzalez, Matei Zaharia, Alex Dimakis, and Ion Stoica. Fantastic adaptive taxonomies and how to use them. In *FAGEN Workshop at ICML 2026*, 2026. URL https://openreview.net/pdf?id=krDmXYWzmJ. Best Paper Award.

Databricks. Introducing AI functions: Integrating large language models with Databricks SQL. https://www.databricks.com/blog/2023/04/18/introducing-ai-functions-integrating-large-language-models-databricks-sql.html, 2023. Accessed: 2026-08-09.

Databricks. Mosaic AI agent framework. https://www.databricks.com/product/machine-learning/mosaic-ai-agent-framework, 2024. Accessed: 2026-08-09.

dbt Labs. What is dbt? dbt Developer Hub, 2026. URL https://docs.getdbt.com/docs/introduction.

Dorottya Demszky, Dana Movshovitz-Attias, Jeongwoo Ko, Alan Cowen, Gaurav Nemade, and Sujith Ravi. GoEmotions: A dataset of fine-grained emotions. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 4040–4054, 2020. URL https://aclanthology.org/2020.acl-main.372/.

Till Döhmen, Jacob Matson, and Jordan Tigani. Towards a context layer for self-improving data agents. In *SAO Workshop at ACM Conference on AI and Agentic Systems (CAIS '26)*, 2026. URL https://bauplanlabs.github.io/SAO-workshop/papers/73.pdf.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. DABstep: Data agent benchmark for multi-step reasoning. *arXiv preprint arXiv:2506.23719*, 2025. URL https://arxiv.org/abs/2506.23719.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. In *Proceedings of the 31st International Conference on Computational Linguistics*, 2025. URL https://aclanthology.org/2025.coling-main.345/.

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

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *Conference on Language Modeling (COLM)*, 2026. URL https://arxiv.org/abs/2603.28052.

Paweł Liskowski, Benjamin Han, Paritosh Aggarwal, Bowei Chen, Boxin Jiang, Nitish Jindal, Zihan Li, Aaron Lin, Kyle Schmaus, Jay Tayade, Weicheng Zhao, Anupam Datta, Nathan Wiegand, and Dimitris Tsirogiannis. Cortex AISQL: A production SQL engine for unstructured data, 2025. URL https://arxiv.org/abs/2511.07663.

Microsoft. Invoke AI capabilities using Azure SQL database and Azure OpenAI. https://learn.microsoft.com/en-us/azure/azure-sql/database/invoke-ai-with-azure-openai, 2024. Accessed: 2026-08-09.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015. URL https://arxiv.org/abs/1504.04909.

E. A. Nadaraya. On estimating regression. *Theory of Probability and Its Applications*, 9(1):141–142, 1964. doi: 10.1137/1109020. URL https://doi.org/10.1137/1109020.

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. White paper, Google DeepMind, 2025. URL https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/AlphaEvolve.pdf.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

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*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Asankhaya Sharma. OpenEvolve: Open-source implementation of AlphaEvolve. GitHub, 2025. URL https://github.com/codelion/openevolve.

Snowflake AI Research. Data-eng-bench: A data engineering benchmark for AI agents. Snowflake Engineering Blog, 2026. URL https://www.snowflake.com/en/blog/engineering/data-eng-bench-data-engineering-agent-benchmark/.

Snowflake Inc. Cortex agents. Snowflake Documentation, 2026. URL https://docs.snowflake.com/en/user-guide/snowflake-cortex/cortex-agents.

Varun Ursekar, Apaar Shanker, Veronica Chatrath, Yuan Xue, and Samuel Marc Denton. VeRO: A harness for agents to optimize agents. In *ICML 2026*, 2026. URL https://arxiv.org/abs/2602.22480.

Rob van der Goot, Alan Ramponi, Arkaitz Zubiaga, Barbara Plank, Benjamin Muller, Iñaki San Vicente Roncal, Nikola Ljubešić, Özlem Çetinoğlu, Rahmad Mahendra, Talha Çolakoğlu, Timothy Baldwin, Tommaso Caselli, and Wladimir Sidorenko. MultiLexNorm: A shared task on multilingual lexical normalization. In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT)*, 2021. URL https://aclanthology.org/2021.wnut-1.55/.

Wenxiao Wang, Priyatham Kattakinda, and Soheil Feizi. Do agent optimizers compound? a continual-learning evaluation on Terminal-Bench 2.0. *arXiv preprint arXiv:2607.14004*, 2026. URL https://arxiv.org/abs/2607.14004.

Geoffrey S. Watson. Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, 26(4):359–372, 1964. URL https://www.jstor.org/stable/25049340.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380, 2018. URL https://aclanthology.org/D18-1259/.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026. URL https://arxiv.org/abs/2601.21557.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639:609–616, 2025. doi: 10.1038/s41586-025-08661-4.

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. ACE: Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2026. URL https://arxiv.org/abs/2510.04618.

Yajie Zhou, Ao Li, Ashwin Silla, Zaoxing Liu, and Vyas Sekar. AIChilles: Automatically uncovering hidden weaknesses in AI-evolved systems. *arXiv preprint arXiv:2606.15834*, 2026. URL https://arxiv.org/abs/2606.15834.

A Table of Contents

This appendix is organized as follows. Appendix D specifies the evolutionary loop (Appendix D.1), the history-query and workspace tools (Appendix D.2), the similarity-graph InfoGain branch used by agent evolution (Appendix D.3), the experimental hyperparameters (Appendix D.4), and the budgeted ℓ^p roster (Appendix D.5). Appendix E reports parent selection, mutation and self-validation, history access, and similarity-manifold targeting ablations. The history-access study includes Signal Processing and DABStep. Each ablation subsection reports its trial count and score provenance. Appendix F records the mutator, reflector, and proposer prompts. Appendix G gives the DABStep

protocol, analyzes Meta-Harness's post-improvement churn, and presents an evolved agent instruction. Appendix H presents the Lockpicking synthetic diagnostic (Appendix H.1) and describes the AI-function experiments and their evolved artifacts. Appendix I describes the data-engineering pipeline, its evaluation, and an evolved artifact. Appendix J records math and ADRS evaluator settings and the detailed result tables (Appendix J.2). Appendix J.3 gives the per-task analysis: which methods lead each cell, then Heilbronn Convex, Circle Packing, Cloudcast, and TXN traces (Figures 4–7). Appendix K gives the methodology for all three prompt-optimization tasks (Appendix K.1) and a sim-graph-targeted, cascaded-evaluation efficiency variant of HotpotQA (Appendix K.2). Appendix L, placed last, includes the complete base and final programs for Heilbronn Convex, Cloudcast, and TXN.

B USE OF LARGE LANGUAGE MODELS (LLMS) AND OTHER AI TOOLS

The authors used large language models (LLMs) to assist in literature discovery and polishing the manuscript's prose. The authors reviewed and verified all suggested references, claims, and revisions, and assume full responsibility for the final content. Coding assistants helped implement the authors' ideas. The methodology, scientific contributions, and core results are the authors' original work.

C ANONYMIZATION

To preserve double-blind review, some reproduced prompts, tool schemas, code excerpts, and environment outputs have been minimally redacted or renamed to remove affiliation-revealing infrastructure identifiers. These edits do not alter the experimental procedures, system behavior, or reported results.

D ALGORITHM AND METHOD

This appendix instantiates the operators of Section 2. Section D.1 gives the evolutionary loop; Section D.2 lists the history tools added through $\mathrm{Augment}$ and the domain tools supplied by $\mathcal{E}_0$; Section D.3 records the kernel, decay, and softmax used for curated observations; Section D.4 lists the hyperparameters used in the experiments of Section 3; and Section D.5 specifies the budgeted ℓ^p roster used as a population objective and as a parent-selection policy.

D.1 EVOLUTIONARY LOOP

Algorithm 1 CoCoEvolve evolutionary loop.

Require: seed p_0, budget T, evaluator f, parent policy π, base environment $\mathcal{E}_0 \subseteq \mathcal{E}$

$\mathcal{A}_0 \leftarrow \{p_0\}$, $\mathcal{H}_0 \leftarrow \emptyset$
for $t = 1, \ldots, T$ **do**
  $p'_t \sim \pi(\mathcal{A}_{t-1})$ ▷ Parent selection
  $c_t \leftarrow \mathrm{Curate}(\mathcal{H}_{t-1})$ ▷ Context curation
  $\mathcal{E}_t \leftarrow \mathrm{Augment}(\mathcal{E}_0; p'_t, \mathcal{H}_{t-1})$ ▷ Environment augmentation
  $p_t \sim \mathrm{MetaReflect}(p'_t, c_t; \mathcal{E}_t)$ ▷ Agentic investigation
  select $\mathcal{T}_t \subseteq \mathcal{T}$; evaluate p_t with f to obtain S_t and O_t
  $\mathcal{H}_t \leftarrow \mathcal{H}_{t-1} \cup \{(p_t, \mathcal{T}_t, S_t, O_t)\}$
  update $\mathcal{A}_{t-1}$ with p_t to form $\mathcal{A}_t$
end for
return $\arg\max_{p \in \mathcal{A}_T} \Phi(p)$

The loop selects parent p'_t from $\mathcal{A}_{t-1}$ using π; the parent policy uses either MAP-Elites or the ℓ^p roster of Section D.5. Context curation prepares c_t, while environment augmentation adds on-demand history-query tools to $\mathcal{E}_0$ to form $\mathcal{E}_t$. Depending on the task, Curate reflects on the strategy ledger and selects a target instance using the similarity graph of Section D.3. Because operational requirements vary by domain, these preparation mechanisms are layered according to the search objective. We detail the exact workflow applied for each experiment in its task subsection.

In general, the injected bundle c_t contains the approach ledger (recent strategies and their score deltas), the active paradigm text if one is in force, and island elites. When an instance pool is defined, c_t also includes the curated target observation and any donor program that has already solved it. The augmented environment $\mathcal{E}_t$ combines the domain tools in $\mathcal{E}_0$ with the opt-in history tools of Section D.2. The coding-agent mutator then performs $\mathrm{MetaReflect}$ by investigating, implementing, and validating the child, after which admission is typically a MAP-Elites update on a complexity–diversity grid.

D.2 On-demand history and environment tools

Table 6 lists the on-demand tools provisioned in $\mathcal{E}_t$. The mutator can call these tools from its task workspace while editing and validating a candidate. The ledger is injected every iteration; tool results enter the context only when requested, so the full archive need not appear in the prompt. For archive access, the mutator calls `python3 evo.py` to query prior programs, lineages, scores, and traces. For local execution, the constrained `evo_sweep.py` runner executes the candidate without granting unrestricted command execution. The math, ADRS, Signal Processing, and DABStep experiments all expose lineage and trajectory queries. Per-task queries are enabled for DABStep and Signal Processing, whose evaluators expose instance-level or per-signal outcomes, and disabled for the remaining math and ADRS tasks, which return only a scalar score. DABStep additionally overlays warehouse tools that issue SQL, deploy a scratch agent, and pre-screen a draft against a target question before committing the child.

Channel	Tool	Returns
Lineage	`get_program`	code body of a program
	`get_parent` / `get_lineage`	parent id; children with score deltas
	`get_best_programs`	top-N programs by primary score
	`get_diverse_programs`	best program per occupied MAP-Elites cell
	`get_architecture_history`	per-cell landing history
Trajectory	`get_recent_scores`	recent primary-score trace
	`get_thinking_trace`	tool/thinking events of iteration I
Per-task	`get_question_solvers`	top programs on one instance, with parent
	`get_question_memory`	child<parent regressions on one instance
	`get_suggested_questions`	instances ranked worst-first for a program
	`get_hard_questions`	instances by population-best score, ascending
Warehouse (DABStep)	`query`	read-only SQL against live tables
	`execute_sql`	allowed write SQL against the warehouse
	`test_agent` / `test_draft`	score a scratch agent on question Q
	`check_deploy`	compile-check one `CREATE` DDL
	`analyst_query`	text-to-SQL against the semantic view

Table 6: On-demand tools available in $\mathcal{E}_t$. Lineage, trajectory, and per-task tools are added through $\mathrm{Augment}$ to query the evolutionary archive; the DABStep base environment $\mathcal{E}_0$ additionally provides live-warehouse validation tools.

D.3 Curated observations: kernel, decay, and softmax

Section 2.3 writes a generic similarity kernel κ. This section applies to the DABStep agent-evolution setting, which evaluates a program on many instances and maintains a similarity graph over that pool. Math, ADRS, and Signal Processing have no such pool and skip this channel, even though Signal Processing still exposes per-signal query tools (Table 7). We precompute an embedding-similarity matrix κ on V, with values in $[0, 1]$. DABStep specializes the estimator in Section 2.3 by replacing $\kappa(\tau, \tau')$ with $\kappa(\tau, \tau')^4$ in both the numerator and denominator, giving more weight to nearby tested instances. After each mutation, a local success or regression update refines the estimate. Estimates for untested coordinates are capped at 0.7, and coordinates with observed failures are floored at 0.1.

The next target τ^* is drawn by softmax over gain with temperature 0.1. Candidates exclude instances the parent has already solved (score ≥ 0.9). The decay in Section 2.3 is instantiated as

$$\delta(\tau) = \begin{cases} 1 & \text{if any program in } \mathcal{A}_t \text{ has solved } \tau, \\ (1/2)^{c(\tau)} & \text{otherwise,} \end{cases}$$

where $c(\tau)$ is the number of times τ has been selected as the mutation target without success. A solve anywhere in the archive therefore resets the penalty, so a newly acquired donor can be reused; repeated failures on an unsolved instance geometrically suppress it.

D.4 Hyperparameters

Table 7 records the settings used in Section 3. The default math, ADRS, and DABStep runs sample parents from MAP-Elites islands using exploration, elite selection, and fitness-weighted draws. Math and ADRS use exploration weight 0.2 and elite-selection weight 0.65. The AI-function objective and the parent-selection ablation instead use the ℓ^p roster of Section D.5. InfoGain targeting (Section D.3) is on for DABStep only.

	Math / ADRS	Signal Processing	DABStep
Iteration budget T	100	100	100
Islands / population	2 / 50	2 / 80	1 / 40
InfoGain targeting	off	off	on
Per-task query tools	off	on	on
Task workspace + history queries	on	on	on
Warehouse / scratch-agent tools	off	off	on
Cascade thresholds	task-dependent*	$[0.3, 0.6]$	$[0.005]$
LLM temperature	0.7 (framework default)	0.6	0.7

Table 7: Domain hyperparameters and tool availability. All listed domains edit candidates in a task workspace and expose on-demand history queries; DABStep additionally enables InfoGain targeting and warehouse tools. *Cascade evaluation is enabled for Circle Packing (Square) with thresholds $[0.5, 0.8]$ and disabled for the remaining math and ADRS tasks.

Signal Processing exposes per-task query tools because its evaluator returns per-signal scores, although it has no V and does not use InfoGain. AI functions use the ℓ^p roster and full validation evaluation rather than InfoGain targeting; they are omitted from the table because their remaining settings follow Section 3.2.

D.5 Budgeted Population ℓ^p Objective and Parent Selection

Let $\mathcal{A}_t$ denote the archive after iteration t. Suppose each program $p \in \mathcal{A}_t$ is evaluated along N dimensions, with per-dimension score $s_i(p)$. A dimension may represent a task instance, objective, or behavioral criterion. We associate each program with the set

$$\mathcal{I}(p) = \{i \in [N] : s_i(p) \geq \theta_i\}$$

of dimensions on which it is considered successful. The thresholds θ_i may be absolute or determined relative to the current population. For binary per-dimension scores, $\mathcal{I}(p)$ is exactly the set of dimensions solved by the program. For continuous scores, we use either a calibrated absolute threshold or a per-dimension population quantile, allowing each dimension to be compared against its own achievable score range.

Budgeted diversity roster. Although $\mathcal{H}_t$ retains the complete evaluation history, only a budgeted active roster $\mathcal{R}_t \subseteq \mathcal{A}_t$, $|\mathcal{R}_t| \leq K$, is used to represent the population and select the next parent. Starting with $\mathcal{R}_t^{(0)} = \emptyset$ and $C^{(0)} = \emptyset$, we greedily choose

$$p_j = \arg \max_{p \in \mathcal{A}_t \setminus \mathcal{R}_t^{(j-1)}} |\mathcal{I}(p) \setminus C^{(j-1)}|,$$

define its marginal contribution as

$$m_j = |\mathcal{I}(p_j) \setminus C^{(j-1)}|,$$

and update

$$\mathcal{R}_t^{(j)} = \mathcal{R}_t^{(j-1)} \cup \{p_j\}, \qquad C^{(j)} = C^{(j-1)} \cup \mathcal{I}(p_j).$$

Construction stops after K programs or when no remaining program contributes new coverage. This is the standard greedy maximum-coverage construction: it retains programs because of capabilities not already represented by higher-ranked roster members, rather than solely because of aggregate fitness.

Population ℓ^p score. Given the ordered roster marginals $\mathbf{m} = (m_1, \ldots, m_{|\mathcal{R}_t|})$, we define

$$L_p(\mathcal{R}_t) = \frac{\|\mathbf{m}\|_p}{N} = \frac{1}{N}\left(\sum_j m_j^p\right)^{1/p}.$$

Equivalently, the evaluator may report its p-th power

$$J_p(\mathcal{R}_t) = L_p(\mathcal{R}_t)^p = \frac{\sum_j m_j^p}{N^p},$$

which induces the same ordering for fixed p. At $p = 1$, the score is the fraction of dimensions covered by the roster. For $p > 1$, it also rewards consolidation: for the same total coverage, the score is higher when fewer programs carry that coverage. Greedy roster construction retains complementary capabilities, while the objective rewards programs that combine them.

ℓ^p as an optimization objective. For tasks in which population quality is itself the target, a newly evaluated child p_t is assessed by inserting it into the candidate pool, rebuilding the roster, and measuring

$$\Delta J_p = J_p(\mathcal{R}_t \mid \mathcal{A}_t \cup \{p_t\}) - J_p(\mathcal{R}_t \mid \mathcal{A}_t).$$

The evolutionary loop then directly optimizes J_p. This is the formulation used in our AI-function setting with $p = 2$, where $J_2 = \sum_j m_j^2/N^2$ rewards both broad population coverage and the consolidation of specialist capabilities.

ℓ^p as a parent-selection policy. When the task retains its original scalar objective, the same roster can instead be used only to choose the next parent. We assign each roster member probability

$$\Pr(p_j \mid \mathcal{R}_t) = \frac{m_j^p}{\sum_\ell m_\ell^p}.$$

This samples programs according to their contribution to the population ℓ^p score without replacing the task's evaluator. At $p = 0$, the policy is uniform over roster members; at $p = 1$, it is proportional to exclusive marginal coverage; our default $p = 2$ emphasizes large complementary contributions quadratically; and as $p \to \infty$, it concentrates on a maximum-marginal roster member. An agentic variant exposes the same roster, marginals, and population score to an LLM and allows it to choose the parent directly, using the mechanical distribution as a fallback. Table 9 reports a HotpotQA sweep of $p \in \{0.5, 1, 1.5, 2\}$ for this parent-selection policy. The `lp` arm uses $p = 2$.

D.5.1 Rare-capability carrier selection

Marginal coverage alone can under-allocate mutations to valuable specialists. Suppose most programs solve the same large set of easy dimensions, while different programs solve only one or two distinct hard dimensions. The greedy roster retains these specialists, but for $p > 1$ its mechanical distribution can place nearly all probability on the first member, whose marginal includes the common easy dimensions. To remove this dependence on raw instance frequency, let

$$\rho_i = \frac{1}{|\mathcal{A}_t|}\sum_{p \in \mathcal{A}_t} \mathbb{I}[i \in \mathcal{I}(p)], \qquad h_i = (1 - \rho_i)^\gamma,$$

where ρ_i is the population solve rate and $\gamma > 0$ controls how strongly rare dimensions are emphasized. We define the hard-capability mass carried by a program as

$$H(p) = \sum_{i=1}^{N} h_i \, \mathbb{I}[i \in \mathcal{I}(p)].$$

We incorporate rare-capability mass into parent selection by sampling according to

$$q_{\text{rare}}(p) \propto \exp\Big(\beta \Big[\lambda \, \widetilde{\Phi}(p) + (1-\lambda) H(p)^p\Big]\Big),$$

where $\widetilde{\Phi}$ is normalized aggregate fitness. The fitness term favors the strongest carrier of a given rare capability, rather than weak programs in general, while $H(p)^p$ gives increasing preference to lineages that accumulate multiple hard capabilities. If all specialists carry one equally rare capability and have equal fitness, they remain indistinguishable and are sampled symmetrically; once one lineage combines multiple rare capabilities, its selection probability increases. Because no selector can exploit a dimension that no current program solves, this policy can be mixed with the baseline selector,

$$q(p) = (1-\epsilon) q_{\text{rare}}(p) + \epsilon q_{\text{base}}(p).$$

The rare-capability term is part of the parent-selection mechanism. It changes which existing program is selected for mutation without adding a donor or changing the evaluation pool. All candidates are evaluated on the same instances, so their capabilities remain directly comparable.

E Ablation Details

This section reports four ablations against the mechanisms of Appendix D and Section 3. Parent selection (Appendix E.1) compares four parent-selection setups and examines the ℓ^p sampling exponent. Three claim-driven ablations each isolate one further axis of the evolutionary loop: mutation execution and self-validation (Appendix E.2), the history-access interface (Appendix E.3), and similarity-manifold targeting (Appendix E.4).

Common controls. Each claim-driven ablation below estimates the effect of one named treatment while holding the rest of the loop fixed; a comparison that changes two axes at once is not reported as a causal ablation. Unless stated as the named treatment, every arm of a given ablation shares the same backbone model, temperature, and model-facing mutation budget; the same seed program and train/test split; the same parent-selection policy and population configuration; the same editable artifact surface; the same number of outer-loop iterations; and the same evaluator thresholds and per-child evaluation allowance. Persistent per-program and archive-level solved-question state is retained across arms, so a no-graph arm does not remember only the current minibatch. The tables below report the final metrics available for each ablation. Each subsection states the number of trials. These controls are already in force below unless a subsection states an exception.

E.1 Parent selection

We test whether useful parent selection requires both preserving complementary capabilities and directing mutations toward programs that can consolidate them. The four-way comparison examines which programs remain eligible as parents; the HotpotQA exponent sweep probes how mutation effort is allocated among them.

Setup. We implement four parent-selection policies in the same OpenEvolve codebase. Within each task, all arms use the same seed, split, backbone, mutation process, island configuration, and 100-iteration budget. Only the parent-selection policy changes. `select-best` always chooses the current best program. `map-elites` uses OpenEvolve's exploration/exploitation/fitness draw from the MAP-Elites grid. `pareto-frontier` samples the per-instance Pareto frontier by win count, as in GEPA. `lp` builds a greedy K-roster from per-instance solved sets and samples member j with probability proportional to m_j^p, where m_j is its marginal coverage (Appendix D.5); here $p = 2$. The frontier and roster draw from all islands. Table 8 compares the complete setups on each task.

To probe whether stronger preference for large marginal contributions helps the roster consolidate specialists, we compare HotpotQA configurations with $p \in \{0.5, 1, 1.5, 2\}$ while keeping the roster

construction fixed. All four use `openai-gpt-5.6-luna`, a 3-example mutation minibatch from train-150, val-300 exact match for search and export, and one 100-iteration search per setting. The $p = 2$ run reuses the HotpotQA `lp` result above; p is the only treatment varied in this comparison.

Results. Table 8 shows the results for the four parent-selection setups and Table 9 shows the results for the HotpotQA ℓ^p sweep.

	HotpotQA		IFBench	
Parent selection	Test EM	Δ	Test score	Δ
`select-best`	0.6400	-0.0267	0.6592	-0.0360
`map-elites`	0.6667	0	0.6952	0
`pareto-frontier`	0.6433	-0.0234	0.6918	-0.0034
`lp` ($p = 2$)	**0.6933**	$\mathbf{+0.0266}$	**0.7003**	$\mathbf{+0.0051}$

Table 8: Parent-selection setups on HotpotQA and IFBench. One 100-iteration search per arm; entries are single held-out test evaluations. HotpotQA uses exact match; IFBench uses its per-constraint score. Each Δ is the score difference from OpenEvolve's default `map-elites` arm on the same task. The `lp` row uses $p = 2$. Bold marks the highest observed score in each task.

Roster exponent	Validation EM	Test EM	Δ test
$p = 0.5$	0.6033	0.5867	-0.0733
$p = 1$	0.7100	0.6600	0
$p = 1.5$	0.5967	0.5700	-0.0900
$p = 2$	**0.7300**	**0.6933**	$\mathbf{+0.0333}$

Table 9: HotpotQA validation and held-out test exact match across ℓ^p roster exponents. Δ is the test-EM difference from $p = 1$; bold marks the highest observed score.

Analysis. These selectors use different evidence to identify useful parents. `select-best` spends every mutation on the current incumbent, leaving complementary capabilities in other programs unused. `map-elites` preserves programs in distinct grid cells, but those cells need not correspond to distinct solved instances. `pareto-frontier` uses per-instance wins, yet win-count weighting does not discount wins already represented by another parent. The greedy ℓ^p roster instead retains a program only while it adds uncovered instances, then favors roster members with larger exclusive contributions. The ℓ^p roster has the highest observed held-out score on both tasks in Table 8, consistent with allocating mutations by complementary coverage rather than grid diversity alone.

The exponent sweep probes how to allocate mutations after complementary parents have entered the roster. Flatter weighting spends more turns on small-marginal specialists; quadratic weighting concentrates ordinary mutations on parents that already contribute many distinct solves, giving later edits more opportunities to consolidate those capabilities. The $p = 2$ row leads on both validation and held-out test, consistent with a balance between consolidation pressure and specialist access. The dip at $p = 1.5$ shows that the effect is not simply monotonic in p: changing the weighting changes which lineages receive repeated edits, and those search paths can produce different final programs.

E.2 Mutation execution and self-validation

This ablation separates two benefits of an interactive mutator: editing the current candidate with access to its environment, and testing that candidate before committing it. The first can support broad instruction-level fixes; the second is especially useful for changes to executable code or dependent objects.

Setup. We run this ablation on Heilbronn Convex (N=13) and DABStep. Within each task, the three arms share the parent policy, population and island settings, feature grid, history-access in-

terface, iteration budget, targeting configuration, and reflector/proposer configuration. They differ in the mutator's action space. Heilbronn Convex has no instance similarity graph; DABStep keeps its similarity graph enabled across all three arms. The reflector/proposer remains on in every arm, matching the published configurations. A1 (*static mutator*) is one LLM completion over the parent, ledger, and active paradigm; it cannot inspect the workspace or execute anything. A2 (*agentic, no candidate testing*) can edit the candidate and query history through `evo.py`, but cannot execute the edited candidate before committing it. A3 (*full mutator*, published) adds pre-commit execution: a local Python sweep on Heilbronn Convex, or deployment and end-to-end testing of the edited agent on DABStep. On DABStep, A1 can only diagnose from injected artifacts, A2 can additionally query the database but cannot deploy the edited agent, and A3 has the complete query, deployment, and testing interface.

Results. We report three Sonnet-5 trials per Heilbronn Convex arm. One trial stopped after 83 iterations; the others ran for 100. Its score is included in the reported summary, so that arm's mean is not based on three equal-budget trials. One 100-iteration trial is reported per arm on DABStep. All DABStep Train/Test values are fresh full-pool re-evaluations of each arm's champion rather than in-run estimates based on limited evaluations; A1's initial estimate used a 25-question bootstrap sample. A1 never advanced past the first evaluation gate, so its champion is the unmodified seed also reported in Table 1 and as A1 in Table 13. A3 on both tasks reuses the published champion rather than a new run; the DABStep champion is also the C3 arm in Table 13. Each Δ is the raw-count difference from A1.

	Heilbronn Convex (min triangle area / hull area)		DABStep			
Arm	Mean $\pm$ Std	Best	Train (of 300)	Δ	Test (of 150)	Δ
A1 static mutator	0.0260 ± 0.0003	0.0266	101	–	47	–
A2 agentic, no candidate test	0.0270 ± 0.0002	0.0276	202	+101	98	+51
A3 full mutator (CoCoEvolve)	0.03066 ± 0.00036	0.03094	286	+185	137	+90

Table 10: Mutation-execution ablation on Heilbronn Convex and DABStep. Values summarize three trials per arm for Heilbronn Convex and one full-pool champion evaluation per arm for DABStep; Δ is relative to A1.

Analysis. Performance improves monotonically from the static mutator (A1), to agentic editing without pre-commit candidate testing (A2), to the full mutator (A3) on both tasks. The mean Heilbronn Convex score rises from 0.0260 to 0.0270 to 0.03066, while DABStep test accuracy rises from $47/150$ to $98/150$ to $137/150$. The two transitions add different capabilities: A1 to A2 adds direct editing and on-demand inspection, while A2 to A3 adds execution of the edited candidate before it is committed.

Heilbronn Convex shows that the A1-to-A2 improvement is not explained by broader exploration: both arms attempt a similar range of strategies. The difference is how those ideas are incorporated. A1 frequently replaces large portions of the solver; in one matched trial, roughly one quarter of its scored edits regress and another third have no effect. A2 generally preserves the existing solver and adds components around it, reducing regressions to about one in eight, although it learns whether an edit worked only after evolutionary evaluation. A3 can test proposed changes through scratch execution and regresses on roughly 4% of scored edits across the trials. Agentic editing therefore helps preserve prior functionality, while self-validation turns more of the budget into measured refinement.

DABStep reveals why the relative gains differ across tasks. A1 sometimes identifies real instruction problems from injected context, but cannot inspect the database or run the revised agent. A2 can query the database, allowing it to correct fee-rule enumeration; the same accepted edit also tightens the response contract for exact-match answers. Because both changes affect many questions, they raise train/test accuracy from $33.7\%/31.3\%$ to $67.3\%/65.3\%$. This closes roughly 55% of the A1-to-A3 gap, compared with about 22% on Heilbronn Convex. A3 additionally deploys and tests the edited agent, allowing it to validate a structural replacement for the fee-computation procedure and reach $95.3\%/91.3\%$. Thus, database-grounded editing can capture broad instruction-level improvements, whereas executable-code and dependent-object rewrites benefit most from end-to-end validation.

E.3 HISTORY-ACCESS INTERFACE

We test two ways to give the mutator access to earlier search: a short ledger injected into each turn and structured tools for retrieving prior programs and traces. The four arms compare each mechanism alone and both together on Signal Processing and DABStep.

Setup (Signal Processing). The selected parent's code and evaluation are available in every arm; this ablation changes access to other programs' history. B1 (*parent-only*) receives no additional history. B2 (*ledger-only*) receives recent approaches, score changes, stagnation information, and the active paradigm in its prompt, but cannot fetch prior programs or traces. B3 (*tools-only*) can request earlier programs, lineage, scores, and thinking traces through `evo.py`, but receives no ledger. B4 (*full*) has both mechanisms. All four arms use Sonnet-5 for 100 iterations with the same population, islands, evaluator, and model-facing mutation budget. The evolution graph and reflector/proposer are off throughout. B4 therefore uses the published Signal Processing history-access interface without the reflector/proposer.

Results (Signal Processing). Table 11 reports the best `overall_score` reached in one trial per arm.

Arm	Injected ledger	Structured history tools	`overall_score`
B1 parent-only	off	off	0.6614
B2 ledger-only	on	off	0.7074
B3 tools-only	off	on	0.7487
B4 full	on	on	**0.7681**

Table 11: History-access interface ablation, Signal Processing. One 100-iteration Sonnet-5 trial per arm, reflector/proposer off throughout. Best `overall_score` reached within the 100-iteration budget (iterations 73, 97, 99, and 75 respectively).

Analysis (Signal Processing). Scores rise from 0.6614 for B1 to 0.7074 with the ledger alone (B2) and 0.7487 with tools alone (B3). B4, with both mechanisms, reaches 0.7681. Thus each mechanism helps relative to parent-only in these runs, and the combination scores highest. The B4 score is near the published Sonnet-5 mean of 0.7656 (best 0.7841), although the reflector/proposer is disabled in this ablation.

The four final Signal Processing programs use different filtering designs. B1 combines a quadratic ridge-trend filter with a noise-adaptive moving average. B2's champion describes an adaptive Kalman filter. B3 adds Rauch–Tung–Striebel backward smoothing to a Kalman filter. B4 combines Kalman and Savitzky–Golay smoothing with a median prefilter and slope-hysteresis logic.

The inspected winning turns illustrate how the interfaces are used. B1, with no access to other attempts, proposes a larger-window Savitzky–Golay filter and Holt smoothing, tests both, rejects them, and keeps a small parameter change. B3 first queries the best programs, recent scores, and lineage, then verifies a refinement of its Kalman/smoothing filter with local score sweeps. B4 recognizes a plateau of more than thirty iterations, queries high-scoring and diverse programs, and uses a local parameter sweep to find an improvement at a narrow threshold boundary. The B4 turn shows the ledger's broad progress summary and targeted history lookup being used in the same edit.

Setup (DABStep). DABStep uses the same four ledger/history-tool settings as Signal Processing. Its evolution graph and reflector/proposer remain enabled in every arm. The graph selects target questions on the controller side, while this ablation changes the history visible to the mutator; turning the graph off would change a second part of the loop. The reflector/proposer also stays fixed across arms. For B1 and B2, the core `evo.py` history commands and the task-specific `get_performance`, `get_suggested_questions`, and `get_artifacts` history commands are removed. B3 and B4 retain them. All four arms keep the six commands used to inspect and test the current candidate: `query`, `test_agent`, `analyst_query`, `check_deploy`, `execute_sql`, and `test_draft`.

Results (DABStep). We report one primary 100-iteration Sonnet-5 trial per arm. Each champion is freshly scored on all 300 training and 150 held-out test questions. B4 reuses the published flagship champion. A second B4 run is discussed below.

Arm	Injected ledger	Structured history tools	Train (of 300)	Test (of 150)
B1 parent-only	off	off	234 (0.780)	116 (0.773)
B2 ledger-only	on	off	210 (0.700)	99 (0.660)
B3 tools-only	off	on	242 (0.807)	119 (0.793)
B4 full	on	on	**286 (0.953)**	**137 (0.913)**

Table 12: History-access interface ablation on DABStep. One primary 100-iteration Sonnet-5 trial per arm; each champion is freshly scored on the full 300-question training and 150-question test pools. Rows follow B1–B4 arm order.

Analysis (DABStep). The four primary runs show a different pattern from Signal Processing. Ledger-only B2 scores below parent-only B1 on both train (210 versus 234) and test (99 versus 116). Tools-only B3 is only slightly above B1 (242 train and 119 test). Full B4 reaches 286 on train and 137 on test, the highest result in this comparison. On DABStep, the large observed gain appears when both history-access mechanisms are enabled; neither mechanism alone shows a comparable gain. This is the central result of the DABStep four-arm table.

A useful fix lost at the evaluation gate. The B1 and B2 traces show that discovering a useful fix does not guarantee it survives search. Both arms repeatedly identify a seed instruction that lists only five merchants from an older dataset version. For an unlisted merchant, the seed can answer "Not Applicable" without calling a tool, even though the current data contains thirty valid merchants. B1 repairs the instruction, and its assigned target question also improves, so the edit enters the surviving lineage. B2 makes and tests the same kind of repair in its first iteration, but its assigned target is unrelated to the merchant bug; the evaluation gate rejects the edit, and later candidates still inherit the stale instruction.

The ledger helps when the mutator can inspect prior programs. The ledger gives the mutator a live-updated record of earlier approaches and projected scores. Its score column is revised as the similarity mechanism estimates full-pool performance, so a ledger value can differ from the program's immediate measured result. In one traced case, a program scored 0.1767 on its immediate evaluation but later appeared in the ledger at 0.4011. B1 sees no such history table. B2 receives the changing projections on every turn but cannot retrieve the earlier programs and traces needed to judge what a high-scoring entry actually does. The ledger-only arm is therefore exposed to uncertain historical scores without a way to check the underlying work, which helps explain its weaker result. B3 and B4 can inspect that work through their history tools.

How B3 and B4 use history. B3 trails B1 through much of the search and gains most in its final thirty iterations. Before two late improvements, the mutator queries prior programs, lineage, scores, and thinking traces. It then builds a counterfactual simulator for questions about alternate fee schemes and lookup cubes for card-scheme and ACI questions, testing both before acceptance. B1 also attempts substantial late changes, including a case bank for difficult questions; the distinction in these traces is that B3 consults the population's history before deciding where to work, not that B1 is incapable of complex edits.

B4 reaches B1's final score by roughly iteration 40 and improves further. Around iteration 36, it queries earlier high-scoring programs and suggested questions, then builds and checks a vectorized aggregation engine for group-by questions. The checks expose an unsupported array/object parameter type, which the mutator works around before the edit is kept. Around iteration 84, it adds a computation mode for ACI-selection questions and checks it against known cases. These episodes show the full arm using its history tools while carrying out substantial, verified edits.

Variation within the full configuration. A second B4 run uses the same configuration and model as the flagship but scores 248/300 on train and 114/150 on test, rather than 286/300 and 137/150. It

ends near B3's primary $242/300$ and $119/150$ result despite having both history-access mechanisms available. The two B4 runs have similar Bash-call volume per iteration (32.3 versus 33.7), but allocate those calls differently. In the flagship, 2092 of 3229 Bash calls invoke `evo.py` and 974 invoke `evo_sweep.py`, about 95% combined. In the second run, 1671 of 3367 invoke `evo.py` and 127 invoke `evo_sweep.py`, about 53% combined. The remaining calls, many of which inspect the edited blueprint directly, are far more common in the second run (about 47% versus 5%). The tool-use pattern and outcome vary together in these two runs; tool use was observed rather than assigned as another ablation treatment.

E.4 SIMILARITY-MANIFOLD TARGETING

We test whether the true question-similarity graph improves DABStep target selection and expansion. The full semantic configuration (C3) scores well above the uniform and shuffled controls in these runs.

Setup. All three arms use DABStep's 300-question training pool and keep the mutator, history-access interface, parent policy, first-evaluation gate, regression-sample size, expansion-question count, scout count, and acceleration budget fixed. The evolution graph remains enabled in every arm. The arms differ in how they select a target and expand evaluation to previously untested questions, as described below.

C1 (*uniform matched control*) samples the target uniformly from the same unsolved candidate set, keeps the regression step identical, and evaluates the same number of random previously untested questions, including the same scout count; persistent solved-state, donor registry, and target-attempt counts are retained. C2 (*shuffled similarity graph*) runs C3's identical information-gain formula, but reads the similarity term off a symmetric index permutation of the real matrix rather than the real values — same value distribution, wrong question pairing, isolating whether the specific semantic geometry matters rather than merely having *some* structured (non-uniform) sampling bias. C3 (*full semantic manifold*) selects the target by information gain over questions the parent has not solved, runs the usual regression sample from the parent's cumulative solved set, and expands the search to semantically similar previously untested questions; archive-wide solvers may be retrieved as donors. All three arms share the same regression-risk sampler: whatever target a given arm's rule selects, that target's most at-risk previously-solved questions are drawn from the true, unpermuted similarity graph. Only target selection and untested-question expansion differ across arms.

C3's champion is the flagship run already reported in Table 1; we report a fresh full-pool re-evaluation of that same champion here, rather than the in-run bookkeeping trajectory used elsewhere, to match C1/C2's methodology exactly.

Results. One 100-iteration Sonnet-5 trial per fresh arm (C1, C2); C3 and A1 reuse the flagship and seed champions already reported elsewhere, each freshly re-evaluated against the same full pools as C1/C2.

Arm	Train (of 300)	Δ	Test (of 150)	Δ
A1 seed (no evolution)	101	–	47	–
C1 uniform (matched)	95	-6	42	-5
C2 shuffled similarity	113	$+12$	51	$+4$
C3 information-gain (flagship)	**286**	$\mathbf{+185}$	**137**	$\mathbf{+90}$

Table 13: Similarity-manifold targeting on DABStep. C1 and C2 each use one 100-iteration Sonnet-5 search; C3 reuses the published flagship and A1 the seed. All four champions are freshly scored on the full 300-question training and 150-question test pools. Δ is relative to A1. The flagship's in-run trace reached 293 on train, while its independent full-pool re-score here is 286.

Analysis. C3 reaches $286/300$ on train and $137/150$ on test, well above C1 ($95/300$, $42/150$) and C2 ($113/300$, $51/150$). C1 scores below the seed ($101/300$, $47/150$), while C2 is only slightly above it. A repeat held-out evaluation of C2's champion scored $49/150$ rather than $51/150$, underscoring the small size of that margin.

The per-question comparison in Table 14 shows both more gains and fewer regressions for C3: it gains 189 questions and regresses 4, versus 29 gained/35 regressed for C1 and 28 gained/16 regressed for C2. Most gains in every arm transfer beyond its directly targeted questions (C3: 143/189; C1: 25/29; C2: 22/28). All arms use the true graph for regression-risk sampling, so aligning target selection and expansion with that graph may help protect previously solved neighbors. These runs evaluate those two semantic operations together; they do not separate their contributions.

Arm	Gained	Regressed	Net	Gain via target	Gain via transfer
C1 uniform (matched)	29	35	-6	4	25
C2 shuffled similarity	28	16	$+12$	6	22
C3 information-gain (flagship)	**189**	**4**	$+\mathbf{185}$	46	143

Table 14: Per-question diff against the A1 seed baseline on the training pool (300 questions), by arm. Net is Gained minus Regressed. Gain via target/transfer partitions each arm's Gained count by whether that question was ever one of the arm's own Phase-0 targets across its 100 iterations.

F PROMPTS USED

CoCoEvolve uses a mutator, reflector, and proposer. The mutator edits the candidate in its workspace. The reflector chooses `keep`, `new`, or `none` from the approach ledger; `new` invokes the proposer to name a strategy. The excerpts below show the instructions relevant to these roles.

The math, ADRS, and Signal Processing mutators share the same generic interface: an injected ledger, on-demand `evo.py` history queries, and a task workspace for editing and execution. DABStep replaces the generic system message with a production-agent builder prompt that adds data-query and deployment tools. The data engineering prompt defines the scaffolding of the context-curation pipeline that CoCoEvolve improves.

Mutator user (math / ADRS)

```
Richer historical evidence is available on demand through a CLI:
  python3 evo.py get_best_programs -n 5
  python3 evo.py get_lineage --id <pid>
  python3 evo.py get_program --id <pid>
  python3 evo.py get_recent_scores -n 20

The current program is already written to a file in this directory.
Edit that file directly. Do not paste code into the reply.

Before finishing, write the declared approach to APPROACH.txt:
the single word hillclimb or new.
```

Mutator system (DABStep workspace)

```
You are an expert production-agent builder. You evolve agent
specifications to improve accuracy on payment processing questions.

Diagnosis priority: test before you diagnose. Run the existing tool
the failing question would use (query, test_agent, or analyst_query).
If it returns the right result, the fix is routing; if not, the fix
is the tool itself.

Live investigation tools (python3 evo.py <subcommand>):
  query        -- read-only SQL against live tables
  test_agent   -- run a scratch clone of the current agent
  check_deploy -- compile-check a candidate DDL
  execute_sql  -- CREATE/CALL against the warehouse
```

```
  test_draft   -- deploy your edited program.json on question Q
```

Mutator system (Data Engineering)

```
You evolve the **KG construction pipeline** for data-eng-bench tasks.
    ## What a Knowledge Graph is (and why it matters)

    Each data-eng-bench task is a small analytics engineering
    challenge: given a
    database and an `instruction.md`, an agent must write dbt models
    that pass
    pytest. Before that coding step runs, **step 1** produces a **
    Knowledge Graph**
    | a structured markdown file (`KG.md`, ≤300 lines) that acts as a
    briefing
    document for the coder. A KG that only restates `instruction.md`
    is
    useless. A good KG is warehouse-grounded and captures:

    - What the task asks for (outputs, metrics, grain)
    - Which schemas/tables/columns exist and how they join
    - Model layering (staging → intermediate → mart), macros, and
    business rules
    - Edge cases, validation checks, and pitfalls to avoid

    A separate **Harbor coding agent** (fixed, not under your control)
     reads
    `KG.md` and writes dbt SQL. Before scoring, it may iterate against
    **KG-derived self-tests** (`test_kg.py` compiled from that KG
    after step 1).
    Those self-tests are NOT the grader | the hidden verifier is
    unchanged.
    **Your** job is to design **one** KG factory reused on every data-
    eng-bench
    task | the pipeline that *creates* that KG (and the lean
    `kg-adherence-test-author` protocol that shapes how tests are
    derived) via skills and agent orchestration descriptions.
    Better architecture → richer KGs + granular self-tests → higher
    pytest pass rate.

    ## The pipeline explained

    Your pipeline **explores the task environment and
    produces `KG.md`, which is handed off to a different, fixed agent
    that solves
    the task.** That is the only fixed contract. Everything else | how
     many
    agents do the exploring/writing, and **what reusable skills the KG
     factory
    follows | is yours to design.** Harbor step 2 receives **only** `
    KG.md`, not
    catalog or runtime skill files. The skill catalog **SHIPS with
    three
    section-skills** (`kg-agent-usage-guide`, `kg-metrics-and-business
    -rules`,
    `kg-structure`). A fourth catalog key, **kg-adherence-test-author
    **, is
    compile-time only (lean protocol for deriving granular test_kg.py
    from each
    KG | not a KG.md section). Your job
    is to **EDIT or GROW the existing pipeline**: author NEW skills
    and DIVIDE THE
```

```
LABOR across agents or REFINE/IMPROVE the existing skills. Refine
kg-adherence-test-author when traces show tests are too generic or
 Harbor
ignores the self-test loop. Drop any skill that stops earning its
place.

## Two axes of architecture improvement

**Axis 1 | Agent topology** (who runs where, doing what)
- Edit `agents[]`: an ordered list of sequential agent sessions.
- The seed is ONE `architect` that explores and writes `KG.md` in
the same
  session (query results stay in context). This is a STARTING
point.
  Do not strip the warehouse look | a factory that writes from the
  instruction alone produces a useless briefing.
- Add, split, merge, or reorder agents freely. Split into
  explorer → architect only if `explore_coverage` shows the look
was skipped
  and a separate look step would help | a handoff file loses raw
tool output.
- Write each agent's `prompt` (explore the DB, synthesize findings
, verify
  facts, render the final KG, etc.).
- Attach proven skills to specific pipeline steps via `agents[].
skills`.
  Skills guide KG exploration and authoring inside the factory;
Harbor step 2
  receives **only** the generated KG (not catalog or runtime skill
 files).
Agents share one workspace. Prefer look+write in the same session
so
`sql_execute` results stay in context | a handoff file loses raw
tool output.

**Axis 2 | Skills** (what reusable protocols the team needs)
- Add, edit, or remove keys in `skills{}`: `{description}`. Attach
 each skill to a pipeline step via `agents[].skills`.
- The assigned agent writes the full skill this eval →
`runtime_skills/<name>.md`. You never paste markdown into the spec
.

## Evidence (how to improve either axis)

Do not guess topology or skills. Justify an Axis 1 or Axis 2 edit
with a
recurring gap in the parent eval: Harbor traces
(`agent_trace.txt`), verifier stdout (`verifier_output_raw.txt`),
and/or `explore_coverage`.

## Full system (three platform steps)

Step 1 | KG pipeline (you design) → `KG.md`
Step 1b | Platform compiler (fixed) → `test_kg.py` from KG + `kg-
adherence-test-author`
Step 2 | Harbor agent (fixed) → dbt models + self-test iteration
→ pytest score

## Your artifact: `_PIPELINE_SPEC`

Edit between `# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END`:

  agents | ordered steps: {name, prompt, skills[]}
```

```
  skills | catalog of {description} per key (justification
optional; assigned agent writes .md)

## Constraints

- Valid Python dict; generic templates only (no task-specific
table/column names).
- Agent prompts keep `{task_name}`, `{db_name}`, `{timeout_sec}`
placeholders.
- Final KG ≤300 lines; no solution SQL in the KG.
- Never put markdown skill bodies in `skills{}` | metadata only.
```

Reflector system

```
You are a strategy-pool controller for an evolutionary
code-optimization run. One worker slot has just freed up and you
decide ONE thing for it: feed it into an existing active strategy,
spawn a brand-new strategy (a proposer agent will design it), or
run plain hillclimbing with no named strategy. You do NOT design
the new strategy yourself and you do NOT edit code.
```

Proposer system

```
You are an expert algorithm researcher and strategic advisor.
Suggest breakthrough algorithmic IDEAS in a JSON object with an
'ideas' array. Each idea describes a high-level strategy (not code).
Focus on ideas that are fundamentally different from what has been
tried. You MUST always return at least one idea.
```

G AGENT OPTIMIZATION

G.1 DETAILED METHODOLOGY

DABStep optimization uses the InfoGain branch of Algorithm 1 over a 300-question training pool, with a disjoint held-out test split reported in Table 1. The run behind Table 15 uses a 100-iteration budget, one island, population 40, cascade threshold $[0.005]$, a task workspace, `evo.py` lineage tools, and the warehouse overlay of Appendix D.2. The seed p_0 is a production agent specification with instructions, `payment_analytics` text-to-SQL, `docs_search`, and a single-transaction `fee_calculator`. The mutator may rewrite the instructions, tool schemas, stored procedures, and dynamic tables. GEPA receives the same specification but optimizes only its instruction text. A post-hoc full-pool evaluation of its champion scores $142/300$ on train and $59/150$ on test (Table 1).

Meta-Harness (Lee et al., 2026) receives the same seed and 100-iteration budget through a faithful port of the Terminal-Bench-2 loop. The proposer prompt contains only a file pointer and receives no curated history or pre-injected per-question signal, matching the original design. We make two changes to the port. A coding-agent backend replaces the original API, and the pooled-parallel DABStep evaluator replaces Harbor sandboxes. Each child is a production agent specification scored on all 300 training questions.

The main-paper Meta-Harness row uses the self-test variant. Its proposer receives `test_draft`, `test_agent`, and `check_deploy`, which provide the same self-validation surface as CoCoEvolve's A3 arm (Appendix E.2). A held-out re-evaluation of the champion produces 80.0% easy accuracy $(20/25)$, 78.4% hard accuracy $(98/125)$, and 78.7% total accuracy $(118/150)$, with no evaluator errors. These are the values in Table 1. Figure 3 plots best-so-far train accuracy against iteration for all three methods.

The remaining two rows are no-search baselines, each scored once on all 450 questions with the same evaluator and scorer as the search methods. The Snowflake Cortex Code Agent row scores the

unmodified seed; the Stock Cortex Agent row scores a general-purpose coding agent without the seed's specification.

Snowflake Cortex Code Agent redeploys the unmodified seed p_0. Its instructions, `payment_analytics` text-to-SQL tool, `docs_search`, and `fee_calculator` remain unchanged. This row measures what the seed already solves before optimization.

Stock Cortex Agent removes the Cortex Agent product entirely. A single Cortex Code CLI call using `claude-sonnet-5` answers each question in a fresh session without memory or an evolutionary loop. The model receives the DABStep documentation and read-only SQL access to the live tables. It does not receive a semantic view, text-to-SQL tool, fee-calculation procedure, or the seed's domain-specific instructions. Together, these two rows bound performance without search: the first retains the seed specification, while the second retains only a general-purpose coding agent with raw data access.

G.2 Combined-suite leaderboard

Table 15 places our combined-pool evaluations beside public DABStep leaderboard entries (Egg et al., 2025) and the context-engineered configuration of Döhmen et al. (2026). Our CoCoEvolve, Meta-Harness (Lee et al., 2026), GEPA (Agrawal et al., 2026b), and no-search rows are evaluated on the 300 training and 150 test questions together; the Meta-Harness and GEPA rows use the champions from Table 1. Public leaderboard submissions report combined-suite accuracy without a separate train/test breakdown. The table is ranked by hard-question accuracy.

Method	Easy (%)	Hard (%)
CoCoEvolve	87.50	**95.24**
MotherDuck, full-context + data modeling (Döhmen et al., 2026)	93.10	93.20
NVIDIA KGMON (NeMo Agent Toolkit) Data Explorer	87.50	89.95
DataPilot	86.11	87.57
Meta-Harness	84.72	77.25
gg-agent-gpt5-1104-1	88.89	62.96
CambioML energent.ai DS Agent	**94.44**	57.67
GEPA	88.89	36.24
Snowflake Cortex Code Agent	75.00	24.87
Stock Cortex Agent (baseline)	56.94	11.90

Table 15: Accuracy on the combined 450-question DABStep suite, ranked by hard-split accuracy.

G.3 Why self-test tools did not prevent Meta-Harness's churn

Figure 3 shows Meta-Harness's self-test variant landing its one real gain at iteration 53 and then failing to improve for the remaining 47 iterations despite holding the same `test_draft`/`test_agent`/`check_deploy` self-validation surface throughout. We inspected the archived per-iteration tool-call transcripts to understand the stall. They show two recurring problems: repeated searches for nonexistent question tables and semantic-view edits that pass the preliminary check but fail at deployment. These traces explain failures in this run; they do not establish that either problem would persist under every use of the self-test tools.

The query tool works; what recurs is a dead end nobody flagged, relearned every time. Meta-Harness can query the real warehouse schema successfully. At iteration 61, its proposer runs `DESCRIBE TABLE DABSTEP_PAYMENTS` and receives a valid column list. In the same iteration, it reads `DABSTEP_FEES` and `DABSTEP_TXN_FEE_FACT` correctly.

The recurring failure is different. On each occurrence, the proposer spends nine calls looking for `DABSTEP_QUESTIONS`, `DABSTEP_EVAL_QUESTIONS`, and `DABSTEP_EVAL`. These tables do not exist because the question and answer bank is stored only in `frontier_val.json` and `evolution_summary.jsonl`. The prompt tells the proposer to read these files for per-question pass rates, but it never states that they are the only source of question-level information.

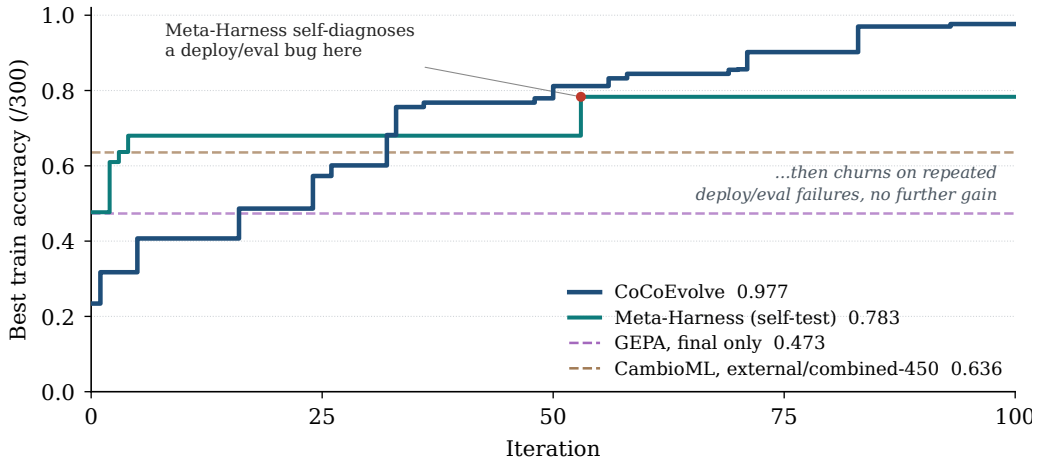

Figure 3: DABStep best-so-far train accuracy (/300) versus iteration. CoCoEvolve and Meta-Harness (self-test) are running-best curves from their evolution logs. GEPA has no per-iteration full-pool trace because its search evaluates small subsamples, sometimes only one question per step, so its final post-hoc score appears as a dashed reference line. The second dashed line shows CambioML energent.ai DS Agent, an external combined-pool leaderboard submission without a matched budget or train-300 trajectory. Its easy and hard accuracies are 94.44% and 57.67%, respectively (Table 15). Appendix G.3 analyzes Meta-Harness's post-iteration-53 stall.

Both harnesses start from the same `initial_program.json`, `agent_spec`, and `referenced_tables`. The difference is therefore not the tool or seed. It is what each prompt surfaces on every turn and whether a discovered fact persists.

The self-test tool cannot see the specific bug class that recurs. The DDL failure is structural. The deployed semantic view contains two representations that must remain synchronized: an outer SQL `tables()/relationships()` clause and an embedded platform-extension JSON object. The JSON separately lists tables, facts, and dimensions.

At iteration 61, the proposer edits both representations with ad hoc Python string operations. It inserts a relationship into the outer clause and separately patches the JSON table list instead of generating both from one source of truth. The two representations can therefore drift apart. In this case, a table appears in `tables()` but not in the JSON.

The resulting text is syntactically well formed, so `check_deploy` cannot reject it in advance. That tool checks whether the DDL compiles but does not perform a data-flow check (`mh_tools.py`). It can catch syntax failures, as it did for an unescaped quote in an earlier iteration, but it cannot validate cross-references between the two representations. Only the live SQL compiler exposes that error, and it returns only a line and position rather than a diff.

The observed failures point to distinct remedies. A prompt that identifies where question-level data lives and carries learned dead ends forward could reduce repeated fruitless queries. Structural validation of the semantic view's two representations could catch the DDL mismatch before deployment. In this run, self-test access alone did not supply either safeguard.

G.4 Examples of Best Programs

DABStep's artifact is a production agent specification (Egg et al., 2025). The champion reached an in-run train estimate of $293/300$ (`combined_score` 0.977). A fresh full-pool re-evaluation of the same artifact scored $286/300$, as reported in Table 13. While the mutator discovered a suite of optimizations across the specification (including refined verified queries, precise orchestration instructions for disambiguation, and injected domain context), we highlight the architectural shift we believe contributed most to its performance: demoting the text-to-SQL surface in favor of a deterministic compute kernel. By routing fee, ranking, and glossary questions around the LLM

entirely, it avoids freehand SQL errors. The champion agent achieved this by rewriting the custom tool from a 5KB SQL procedure to a 53KB Python engine with 14 modes, backed by NumPy, SciPy, and NetworkX. The agent can therefore answer complex questions with a single tool call instead of summing many single-row fees in context, directly addressing the failure mode discussed in Section 3.

The deployed agent reads the following abridged specification, which heavily guides it to trust its new modes over its text-to-SQL capabilities. Key routing instructions and tool descriptions added by the mutator are highlighted in blue:

Evolved specification (champion, abridged)

```
{
  "agent_spec": {
    "instructions": {
      "response": "Provide precise, data-driven answers. Always show
    the exact numbers from query results. <abridged...>",
      "orchestration": "You are a Payment Operations Analyst agent
    specialized in the Dabstep payment processing dataset. <abridged
    ...>
      ## Tool Selection Rules
### New fee_calculator modes for residual question families
Route these DIRECTLY to fee_calculator instead of freehand SQL via
    payment_analytics:
- "For [account type X and/or MCC description: Y...] what would be the
     average fee..." -> mode='avg_fee_criteria'
- "Which fee ID(s) apply to <criteria>" with no merchant scope -> mode
    ='fee_ids_criteria'
- "How many unique/distinct <column> are there" -> mode='
    distinct_count'
<abridged...>

## Fee Rule Engine (fee_calculator tool) -- generalized modes
The fee_calculator tool is a generalized vectorized fee-matching
    engine over a merchant's real transactions (it sums ALL matching
    fee rules per transaction, then aggregates). Route these question
    categories to it directly instead of hand-writing SQL: <abridged
    ...>
Trust this tool's numbers over manually derived SQL for these
    categories -- it already handles monthly fraud/volume bracket
    computation, capture_delay bucketing, and the correct intracountry
     definition. <abridged...>"
    },
    "tools": [
      {
        "tool_spec": {
          "type": "text_to_sql",
          "name": "payment_analytics",
          "description": "Use this tool to query payment transaction
    data, merchant information, fee rules, and acquirer details. <
    abridged...>"
        }
      },
      {
        "tool_spec": {
          "type": "semantic_search",
          "name": "docs_search",
          "description": "Use this tool to search payment processing
    domain documentation including the merchant guide, fee
    optimization strategies, fraud prevention best practices. <
    abridged...>"
        }
      },
```

```
      {
        "tool_spec": {
          "type": "generic",
          "name": "fee_calculator",
          "description": "Generalized vectorized fee-rule engine for a
     merchant's transactions. Use this tool (NOT freehand SQL) for ANY
     question about: (1) which card scheme a merchant should steer
    traffic to for minimum/maximum fees -- mode='steer'; (2) total
    fees a merchant paid/should pay over a year/month/day -- mode='
    total'; <abridged...>",
          "input_schema": {
            "type": "object",
            "properties": {
              "p_mode": {
                "description": "One of: 'steer' (best/worst card
    scheme for min/max fees), 'steer_aci' (best/worst ACI to move a
    merchant's fraudulent transactions towards for min/max fees... <
    abridged...>",
                "type": "string"
              }
              <abridged...>
            }
          }
        }
      }
    ]
  }
}
```

A warehouse stored procedure backs the `fee_calculator` tool. While we omit the full 53KB Python body, the evolution trajectory reveals several striking mechanistic and semantic discoveries that the agent authored and verified via workspace execution:

- **Vectorized tensor contraction.** The initial tool used SQL to evaluate 9-field conditional fee rules. The champion replaces this with a sparse boolean tensor contraction in Python. It precomputes a boolean "allowed" matrix for each field where wildcards become an all-True row, combines them via bitwise `&`, and computes the final fees using sparse matrix-vector products (`scipy.sparse.csr_matrix`).
- **Semantic bug fixes.** The mutator fixed two silent bugs in the human-authored seed. First, it corrected a whitespace bug in a string label (`'> 5'` to `'>5'`) that had silently disabled 92 fee rules for 6 merchants. Second, it redefined "intracountry" transactions from `IP_COUNTRY = ISSUING_COUNTRY` to `ISSUING_COUNTRY = ACQUIRER_COUNTRY`, a definitional shift that alters the evaluation of 70% of the 138,236 rows in the dataset.
- **Emergent domain semantics.** The agent discovered through trial and error that fees *stack*. Its code comments explicitly warn its future self: "because fees STACK... Do NOT take the MAX of a single matching rule... that produces a different (and wrong) ranking."
- **Graph-based join resolution.** Rather than letting the orchestrator guess join keys for text-to-SQL, the tool hardcodes the schema's foreign-key graph and uses `networkx.shortest_path` to emit deterministic `JOIN ... ON` fragments, returning an explicit refusal if no path exists.
- **Regex over semantic search.** For exact-definition questions, the agent deliberately replaces vector search with a regex and `difflib` parser over the documentation, reasoning that embeddings might return a topically-related chunk that omits the exact definition.

Beyond the champion tool shown above, other evolutionary lineages explored alternative structural solutions. The most notable alternatives materialized dynamic tables to pre-compute frequently-joined columns and eliminate flaky on-the-fly joins and produced detailed sets of verified queries encoding domain-specific reasoning paths. While these alternative approaches were interesting, they

ultimately lost to the champion's centralized Python compute kernel. Across all lineages, the agent also continually refined its orchestration guidelines, encoding lessons from observed failures into tight situational guardrails.

H AI FUNCTIONS

H.1 SYNTHETIC DIAGNOSTIC: LOCKPICKING WITH INTERFERENCE

Construction and protocol. This diagnostic isolates recovery of interacting rules from the semantic uncertainty of the natural-data tasks. Each example is a six-digit string $P_1 \dots P_6$ with an integer label in $\{0, \dots, 6\}$. A position first passes a position-specific candidate-set check; twelve ordered trigger rules (24 trigger–target edges) can then invalidate otherwise-valid positions. The label is the number of positions that remain valid. For example,

$$P_1 \in \{1, 4, 7\}.$$

Two rules that can subsequently invalidate P_1 are

$$P_2 = 8 \Longrightarrow P_1, P_3 \text{ invalid}, \qquad P_3 = 3 \Longrightarrow P_1, P_2 \text{ invalid}.$$

The seed provides all six candidate sets but only one of the twelve interference rules. Every method receives the same 50 labeled strings and 50 mutation iterations. There is no held-out partition: train and test are the same fixed set, because the measured quantity is exact reconstruction of the latent rule system rather than distributional generalization.

Why aggregate accuracy is underdetermined. The evaluator returns only the final count, not which positions were invalidated. In the example above, if both triggers fire, omitting either edge into P_1 need not change that row's label. A trigger can likewise be masked when its target was already rejected by the candidate-set check or an earlier rule. Consequently, many structurally different rule tables agree on most of the 50 examples. This property makes the diagnostic useful: a locally successful patch can raise accuracy without identifying the rule responsible for the improvement.

Method	GEPA	OpenEvolve	AdaEvolve	CoCoEvolve
Correct / 50	41	27	45	**50**
Accuracy	0.82	0.54	0.90	**1.00**

Table 16: Accuracy on *Lockpicking with Interference* after 50 iterations.

Recovered search behavior. CoCoEvolve reaches $50/50$ after six mutations by contrasting multiple failures and compiling all twelve recovered rules into deterministic SQL. OpenEvolve ends at $27/50$ with unresolved contradictions in its own worked examples. AdaEvolve reaches $45/50$ at iteration 34 and then plateaus: only two of its twelve learned trigger rows have the correct target pair, five reuse the same `[P3,P5]` shortcut, and none can invalidate P_1. The gap is therefore structural rather than evaluation noise: the baselines fit surrogate rules that score well on the fixed set, whereas CoCoEvolve recovers the latent rule system.

H.2 DETAILED METHODOLOGY

Program representation. Each candidate is an executable SQL expression that forms the body of a scalar AI function. The seed for every task contains one `AI_COMPLETE` call, a task instruction, a row-dependent user message, and a structured-output schema. Evolution may rewrite the instruction and schema descriptions, introduce CTEs or deterministic SQL post-processing, and compose multiple model calls, but it must preserve the function signature, output field and type, and final accessor. Thus the search space includes both prompt optimization and pipeline optimization while every candidate remains directly executable by the target query engine. The seed is deliberately minimal: for example, the `go_emotions` seed asks for every emotion that "clearly applies," and the `multilexnorm` seed gives only four normalization examples.

Tasks and metrics. The five live-data tasks use the following row-level objectives: `go_emotions` uses example-level multi-label F1; `finqa` and `convfinqa` use execution accuracy after canonicalizing the returned numeric answer; `lex_glue` uses exact match over thirteen Supreme Court issue areas; and `multilexnorm` uses exact string match, including case, punctuation, and spacing. The synthetic `lockpicking_6` diagnostic uses exact match on the integer score. A malformed SQL body, invalid structured output, or warehouse execution failure receives score zero together with the compilation or execution diagnostic.

Data partitions. For each live-data task, the search pool is partitioned once with seed 42. Two thirds form a validation split used only for fitness and acceptance, while the remaining third supplies row contents and scorer feedback to the mutation agent. The disjoint test table is never exposed during evolution; after the 50-iteration run, the exported validation champion is re-executed on that test table to obtain Table 2. This separation is important because the mutator can inspect concrete training failures. `lockpicking_6` is intentionally different: all examples are visible and scored because its purpose is to test whether search can reconstruct a hidden rule system rather than generalize to an unseen natural-data split.

Mutation protocol. The AI-function runs instantiate the general loop in Appendix D.1, the history interface in Appendix D.2, and the ℓ^p roster in Appendix D.5. Their task-specific specialization is a fresh-session curator–mutator split: the curator inspects the roster, ledger, and training-only failure history to select a parent, an optional merge partner, and up to ten representative rows; the mutator receives only that bounded packet and may verify its SQL on those rows up to three times. The subsequent full-validation evaluation is the sole admission gate. Runs use 50 mutation attempts, one island, population 40, archive capacity 20, and eight parallel workers; similarity-informed targeting is disabled because every child is evaluated on the complete validation split.

Population objective. AI functions use the continuous $p = 2$ instantiation of Appendix D.5, with a roster budget of ten. Marginals are sums of row-level score improvements rather than binary solved-row counts; all other roster construction, consolidation, and parent-selection details follow that section.

Matched baselines and score provenance. OpenEvolve and AdaEvolve start from the same task-specific SQL body and receive 50 mutation attempts. Their validation champions are scored post hoc on the same held-out test partition. GEPA likewise optimizes the SQL body against the validation partition; Table 2 uses the best completed budget for each task. For the five live-data tasks, the paper values are held-out re-evaluations rather than the optimizer's internal archive fitness. For lockpicking, all methods are scored on the same fixed visible set, consistent with the diagnostic protocol above.

H.3 Examples of Best Programs

The evolved programs remain ordinary SQL bodies, but they are substantially more operational than their seeds. Across tasks, evolution turns a generic instruction into an explicit decision procedure: ordered rules for classification, reference and sign conventions for numerical reasoning, or token-preservation contracts for normalization. We give representative examples below, reproducing the two lockpicking champions in full and abridging the longer live-task programs. For the live tasks, the base boxes show the task-bearing seed instruction; we omit repeated label enumerations, the `AI_COMPLETE` wrapper, and the unchanged structured-output schema.

`lockpicking_6`: exact rules versus a surrogate ruleset. The two champion artifacts make the structural difference directly inspectable. The CoCoEvolve listing is the complete deterministic scoring expression produced by its winning child. In the deployed AI-function body, the resulting integer is passed to `AI_COMPLETE` only for schema-conforming serialization.

`lockpicking_6` base program (instruction)

```
Score a 6-digit program string.
```

```
If the input is not exactly six digits, return 0.
Valid digits:
  P1: {1,4,7}  P2: {2,6,8}  P3: {0,3,9}
  P4: {1,4,7}  P5: {0,5,8}  P6: {2,5,9}

Apply interference rules in order:
  1. IF P1 = 4 -> invalidate P2, P6

Return the number of positions still valid after both checks.
```

CoCoEvolve champion: deterministic scoring expression

```
CASE WHEN NOT REGEXP_LIKE(PROGRAM, '^[0-9]{6}$') THEN '0'
ELSE CAST(
    IFF(SUBSTR(PROGRAM,1,1) IN ('1','4','7') AND NOT (SUBSTR(PROGRAM,2,1)='8' OR
     SUBSTR(PROGRAM,3,1)='3' OR SUBSTR(PROGRAM,5,1)='5' OR SUBSTR(PROGRAM,6,1)='2'), 1, 0)
    + IFF(SUBSTR(PROGRAM,2,1) IN ('2','6','8') AND NOT (SUBSTR(PROGRAM,1,1)='4' OR
     SUBSTR(PROGRAM,3,1)='3' OR SUBSTR(PROGRAM,4,1)='7' OR SUBSTR(PROGRAM,5,1)='8'), 1, 0)
    + IFF(SUBSTR(PROGRAM,3,1) IN ('0','3','9') AND NOT (SUBSTR(PROGRAM,1,1)='1' OR
     SUBSTR(PROGRAM,2,1)='8' OR SUBSTR(PROGRAM,4,1)='7' OR SUBSTR(PROGRAM,6,1)='2'), 1, 0)
    + IFF(SUBSTR(PROGRAM,4,1) IN ('1','4','7') AND NOT (SUBSTR(PROGRAM,2,1)='6' OR
     SUBSTR(PROGRAM,3,1)='0' OR SUBSTR(PROGRAM,5,1)='5' OR SUBSTR(PROGRAM,6,1)='9'), 1, 0)
    + IFF(SUBSTR(PROGRAM,5,1) IN ('0','5','8') AND NOT (SUBSTR(PROGRAM,1,1)='1' OR
     SUBSTR(PROGRAM,2,1)='6' OR SUBSTR(PROGRAM,4,1)='1' OR SUBSTR(PROGRAM,6,1)='9'), 1, 0)
    + IFF(SUBSTR(PROGRAM,6,1) IN ('2','5','9') AND NOT (SUBSTR(PROGRAM,1,1)='4' OR
     SUBSTR(PROGRAM,3,1)='0' OR SUBSTR(PROGRAM,4,1)='1' OR SUBSTR(PROGRAM,5,1)='8'), 1, 0)
AS VARCHAR) END
```

The AdaEvolve listing is its complete $45/50$ champion prompt, including the ten worked examples accumulated during search. We normalize typographic dashes and operators to ASCII for typesetting; the wording and rules are otherwise unchanged.

AdaEvolve champion: evolved executor prompt

```
You are a scoring evaluator for a 6-digit program string. Output ONLY a single integer
    (0-6).

## CRITICAL RULE: Always use ORIGINAL digit values when checking triggers, even if that
    position is base-invalid.

---

## PHASE 1 - Input Validation
If the input is not exactly 6 digit characters (0-9), output 0 and stop.

Label positions left-to-right: P1, P2, P3, P4, P5, P6.

---

## PHASE 2 - Base Validity Lookup

Mark a position BASE-INVALID if its digit is NOT in the listed set.

| Position | Valid digits |
|----------|--------------|
| P1       | 1, 4, 7      |
| P2       | 2, 6, 8      |
| P3       | 0, 3, 9      |
| P4       | 1, 4, 7      |
| P5       | 0, 5, 8      |
| P6       | 2, 5, 9      |

---

## PHASE 3 - Trigger Table

Check EVERY row below independently. If the condition matches the ORIGINAL digit value at
    that position, mark ALL listed positions as TRIGGER-INVALID. Check all 12 rows every
    time - do not skip any.

| Row | Condition | Mark TRIGGER-INVALID |
```

```
|-----|-----------|----------------------|
|  1  | P1 = 4    | P2, P6               |
|  2  | P1 = 1    | P3, P4               |
|  3  | P4 = 1    | P3, P5               |
|  4  | P4 = 7    | P3, P6               |
|  5  | P2 = 6    | P3, P5               |
|  6  | P2 = 8    | P3, P5               |
|  7  | P3 = 0    | P5, P6               |
|  8  | P3 = 3    | P5, P6               |
|  9  | P5 = 5    | P2, P6               |
| 10  | P5 = 8    | P2, P6               |
| 11  | P6 = 2    | P3, P5               |
| 12  | P6 = 9    | P3, P5               |

---

## PHASE 4 - Final Count

A position is VALID if and only if:
- It is NOT base-invalid, AND
- It is NOT trigger-invalid

Count VALID positions and output ONLY that integer (0-6).

---

## WORKED EXAMPLES

**Example 1: '729405'**
P1=7, P2=2, P3=9, P4=4, P5=0, P6=5

Base: all valid (7 in {1,4,7}, 2 in {2,6,8}, 9 in {0,3,9}, 4 in {1,4,7}, 0 in {0,5,8}, 5
     in {2,5,9})

Trigger scan (all 12 rows):
- Row 1: P1=7 != 4 -> skip
- Row 2: P1=7 != 1 -> skip
- Row 3: P4=4 != 1 -> skip
- Row 4: P4=4 != 7 -> skip
- Row 5: P2=2 != 6 -> skip
- Row 6: P2=2 != 8 -> skip
- Row 7: P3=9 != 0 -> skip
- Row 8: P3=9 != 3 -> skip
- Row 9: P5=0 != 5 -> skip
- Row 10: P5=0 != 8 -> skip
- Row 11: P6=5 != 2 -> skip
- Row 12: P6=5 != 9 -> skip

No triggers fired. All 6 positions valid.
**Output: 6**

---

**Example 2: '429405'**
P1=4, P2=2, P3=9, P4=4, P5=0, P6=5

Base: all valid.

Trigger scan:
- Row 1: P1=4 -> FIRES -> mark P2, P6 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P2, P6

P1=VALID, P2=INVALID, P3=VALID, P4=VALID, P5=VALID, P6=INVALID
**Output: 4**

---

**Example 3: '720105'**
P1=7, P2=2, P3=0, P4=1, P5=0, P6=5

Base: all valid.

Trigger scan:
- Row 3: P4=1 -> FIRES -> mark P3, P5 TRIGGER-INVALID
- Row 7: P3=0 -> FIRES -> mark P5 (already), P6 TRIGGER-INVALID
- All other rows: no match
```

```
Trigger-invalid: P3, P5, P6

P1=VALID, P2=VALID, P3=INVALID, P4=VALID, P5=INVALID, P6=INVALID
**Output: 3**

---

**Example 4: '729155'**
P1=7, P2=2, P3=9, P4=1, P5=5, P6=5

Base: all valid.

Trigger scan:
- Row 3: P4=1 -> FIRES -> mark P3, P5 TRIGGER-INVALID
- Row 9: P5=5 -> FIRES -> mark P2, P6 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P2, P3, P5, P6

P1=VALID, P2=INVALID, P3=INVALID, P4=VALID, P5=INVALID, P6=INVALID
**Output: 2**

---

**Example 5: '729489'**
P1=7, P2=2, P3=9, P4=4, P5=8, P6=9

Base: all valid.

Trigger scan:
- Row 10: P5=8 -> FIRES -> mark P2, P6 TRIGGER-INVALID
- Row 12: P6=9 -> FIRES -> mark P3, P5 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P2, P3, P5, P6

P1=VALID, P2=INVALID, P3=INVALID, P4=VALID, P5=INVALID, P6=INVALID
**Output: 2**

---

**Example 6: '129402'**
P1=1, P2=2, P3=9, P4=4, P5=0, P6=2

Base: all valid.

Trigger scan:
- Row 2: P1=1 -> FIRES -> mark P3, P4 TRIGGER-INVALID
- Row 11: P6=2 -> FIRES -> mark P3 (already), P5 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P3, P4, P5

P1=VALID, P2=VALID, P3=INVALID, P4=INVALID, P5=INVALID, P6=VALID
**Output: 3**

---

**Example 7: '789405'**
P1=7, P2=8, P3=9, P4=4, P5=0, P6=5

Base: all valid.

Trigger scan:
- Row 6: P2=8 -> FIRES -> mark P3, P5 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P3, P5

P1=VALID, P2=VALID, P3=INVALID, P4=VALID, P5=INVALID, P6=VALID
**Output: 4**

---

**Example 8: '729705'**
P1=7, P2=2, P3=9, P4=7, P5=0, P6=5

Base: all valid.
```

```
Trigger scan:
- Row 4: P4=7 -> FIRES -> mark P3, P6 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P3, P6

P1=VALID, P2=VALID, P3=INVALID, P4=VALID, P5=VALID, P6=INVALID
**Output: 4**

---

**Example 9: '723705'**
P1=7, P2=2, P3=3, P4=7, P5=0, P6=5

Base: all valid.

Trigger scan:
- Row 4: P4=7 -> FIRES -> mark P3, P6 TRIGGER-INVALID
- Row 8: P3=3 -> FIRES -> mark P5, P6 (already) TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P3, P5, P6

P1=VALID, P2=VALID, P3=INVALID, P4=VALID, P5=INVALID, P6=INVALID
**Output: 3**

---

**Example 10: '769405'**
P1=7, P2=6, P3=9, P4=4, P5=0, P6=5

Base: all valid.

Trigger scan:
- Row 5: P2=6 -> FIRES -> mark P3, P5 TRIGGER-INVALID
- All other rows: no match

Trigger-invalid: P3, P5

P1=VALID, P2=VALID, P3=INVALID, P4=VALID, P5=INVALID, P6=VALID
**Output: 4**
```

`go_emotions`: conservative labeling and ordered rules.

`go_emotions` base program (instruction)

```
Identify every emotion expressed in the text. A comment may express
more than one emotion. Return an array drawn from the declared
28-label vocabulary. Include every category that clearly applies;
use ["neutral"] only when no other emotion applies.
```

The seed's instruction to include every plausible emotion caused systematic over-labeling. At iteration 7, the curator queried the seed's full failure set, inspected representative rows, and then used question memory to contextualize partial scores. For "Happy Cake day!" the gold label was `joy` but the parent received 0.5; for "This is my favorite wild memory" it received 0.4. These scores exposed a common mechanism: the parent was returning the correct emotion together with one or more unnecessary labels. The curator prescribed a conservative single-label default. The mutator's first draft passed eight of ten selected rows; after adding rules for sarcastic positive language and social check-ins, the same live verifier reported ten of ten. The committed child increased the search objective from 0.0022 to 0.1741 in one iteration.

The final champion generalizes this idea into a strict-priority rule system:

`go_emotions` champion (abridged)

```
Classify emotions using the GoEmotions annotation scheme.

ALLOWED LABELS --- use only the 28 labels in the declared vocabulary.
Apply the following rules in strict priority order. Record which rule
```

```
matched, then output only that rule's prescribed emotion list.

RULE A --- GRATITUDE EXPRESSION:
If the text starts or ends with "thanks" or "thank you",
output ["gratitude"] only.

RULE B --- EXPLICIT FIRST-PERSON EMOTION:
Check explicit worry, fear, love, sadness, joy, curiosity, pride,
and relief patterns in order; stop at the first match.

RULE C --- INTERPERSONAL STATE QUESTION:
For a direct question about another person's wellbeing,
output ["caring", "curiosity"].

...

DEFAULT:
Return the single most prominent emotion unless two emotions are
independently and explicitly expressed.
```

The champion was discovered at iteration 47 and obtains the held-out score 0.444 reported in Table 2. Although the artifact is prompt-heavy, its structure is algorithmic: ordered guards, explicit exceptions, a closed vocabulary, and a forced structured output replace the seed's unconstrained semantic judgment.

`finqa` and `convfinqa`: executable arithmetic contracts.

`finqa`/`convfinqa` base programs (instructions)

```
FINQA:
Answer a numerical question over a financial report (text + table).
Work out the answer and return the final numeric value (or yes/no).
Use no units, thousands separators, or percent sign.

CONVFINQA:
Answer the current turn of a multi-turn numerical conversation over
a financial report. Use dialogue history to resolve references such
as "the difference" or "this". Return the final plain numeric value
(or yes/no), with no units, commas, or percent sign.
```

The financial-QA seeds merely ask the model to work out a numeric answer. Their champions instead define an operational contract for reference resolution, formula selection, sign direction, units, and answer serialization. The `finqa` program distinguishes percent change, signed absolute change, relative increase, same-period differences, ratios, averages, CAGR, and yes/no questions. The conversational champion adds rules for resolving "that," "the difference," and "the last year," and for reusing prior answers without silently substituting a nearby year.

`finqa` champion (abridged)

```
Work through each question step by step:
1. Identify the relevant numbers from the text or table.
2. Match the question's exact wording to the correct formula.
3. Compute carefully and show intermediate steps.

FORMULA SELECTION:
"percent change"       -> (new - old) / old; preserve the sign
"change in X"          -> new - old in the table's stated unit
"increase/decrease"    -> (new - old) / old as a signed ratio
"how much bigger"      -> X - Y; do not divide
"average"              -> sum / count
"ratio of X to Y"      -> X / Y
```

```
"CAGR"                -> (end / start)^(1 / years) - 1, then x100

Return plain values only: no currency signs, commas, units, or percent
signs. Do not convert away from the table's stated unit. Sign matters.
```

convfinqa champion (abridged)

```
STEP 1 --- RESOLVE REFERENCES:
Resolve pronouns and shorthand from the dialogue history. If the
    current
question requests a value already stated for the same quantity and
    year,
reuse that exact value.

STEP 2 --- IDENTIFY OPERATION AND SIGNS:
"decline from A to B"      -> A - B
"change from A to B"       -> B - A
"combined total"           -> sum the referenced values
"as a percentage of"       -> component / whole

STEP 3 --- EXTRACT VALUES:
List every required value and its source. Match the requested year
    exactly.

STEP 4 --- COMPUTE:
Show the arithmetic explicitly, then emit a plain numeric answer
with no unit, comma, or percent sign.
```

These contracts address errors that are difficult to repair with isolated answer demonstrations: the same numbers can require a ratio, signed difference, or magnitude depending on a short phrase. The resulting held-out scores are 0.813 for `finqa` and 0.953 for `convfinqa`.

`multilexnorm`: conservative token transformation.

multilexnorm base program (instruction)

```
Normalize noisy social-media text. Rewrite each non-standard token
to its canonical form (e.g., u->you, tmrw->tomorrow,
gonna->going to, dont->don't). Leave standard words, @mentions,
#hashtags, and URLs unchanged. Return the full normalized text.
```

Exact-match normalization is unusually sensitive to helpful but unauthorized rewrites. The evolved champion therefore begins with preservation rules, then applies a bounded lexicon of high-confidence transformations while preserving capitalization, spacing, punctuation, mentions, hashtags, URLs, proper nouns, and uncertain or non-English tokens.

multilexnorm champion (abridged)

```
Rewrite each non-standard English token to its canonical form,
leaving everything else exactly as written.

ALWAYS LEAVE UNCHANGED:
- @mentions, #hashtags, URLs, proper nouns, and non-English words
- already-standard informal forms such as wanna, kinda, lol, and btw
- uncertain tokens; when in doubt, leave the token unchanged
- emotional sound effects whose repeated letters encode emphasis

CAPITALIZATION:
Preserve the exact capitalization pattern of the original token.
```

```
SPACING AND PUNCTUATION:
Preserve all whitespace and punctuation spacing exactly.

HIGH-CONFIDENCE TRANSFORMS:
dont -> don't;  u -> you;  ur -> your;  pls/plz -> please;
tryn/tryna -> trying to;  thx/thnx -> thanks;  nvm -> never mind.
```

The final program also records exceptions for contractions, elongated forms, and truncated `-ing` words. This "transform only when confident" policy is the opposite of the seed's broad invitation to normalize and achieves 0.587 on the held-out set, a 7.7 pp gain over GEPA.

`lex_glue`: hierarchy of legal exceptions.

`lex_glue` base program (instruction)

```
Classify a US Supreme Court opinion by its primary SCDB issue area.
Choose exactly one of:
Criminal Procedure, Civil Rights, First Amendment, Due Process,
Privacy, Attorneys, Unions, Economic Activity, Judicial Power,
Federalism, Interstate Relations, Federal Taxation, Miscellaneous.

Use the label glossary: rights of the accused -> Criminal Procedure;
discrimination and voting -> Civil Rights; speech, press, and religion
-> First Amendment; abortion and personal autonomy -> Privacy;
fees and professional discipline -> Attorneys; labor relations ->
Unions; antitrust, contracts, torts, and business regulation ->
Economic Activity; jurisdiction, standing, mootness, and remedies ->
Judicial Power. Choose the issue area that dominates the holding.
```

The legal classifier evolves from a short glossary into a hierarchy of issue-area distinctions. Its champion begins with a priority scan for attorney-fee holdings, then separates the constitutional substance of a case from its procedural posture. For example, it distinguishes a speaker denied access to a government forum (`First Amendment`) from a captive listener resisting unwanted content (`Privacy`); federal-court jurisdiction over an agency order (`Judicial Power`) from the substantive scope of an agency's economic authority (`Economic Activity`); and federal labor preemption (`Federalism`) from the underlying labor right (`Unions`). The complete prompt is long, but these repeated "default–exception–counterexample" clauses are precisely the structure that the archive accumulated from observed confusions. The champion scores 0.737 on held-out opinions.

`lex_glue` champion (abridged)

```
Choose exactly one SCDB issue area from the declared 13-label
    vocabulary.

FIRST --- ATTORNEY FEES PRIORITY SCAN:
If the primary holding concerns attorney-fee entitlement, computation,
recovery, or denial, classify as Attorneys immediately; the underlying
substantive dispute does not change the classification.

SPEAKER / LISTENER DISTINCTION:
A speaker denied access to a government forum is First Amendment.
A captive listener resisting compelled content is Privacy.

AGENCY CASES:
Whether a federal court may review an agency order is Judicial Power.
The substantive scope of the agency's regulatory authority is
Economic Activity (or Unions for a substantive labor right).

LABOR PREEMPTION:
Federal labor law displacing state or local regulation is Federalism;
```

```
the underlying organizing or bargaining right is Unions.
```

Together, these artifacts show the same pattern at different scales: evolution replaces broad task descriptions with explicit precedence, exception, and serialization rules.

I Data Engineering

I.1 Detailed Methodology

Data engineering optimization applies Algorithm 1 to data-eng-bench. We define 50 dev tasks and 53 held-out test tasks, randomly sampled from the full set of 103 tasks.

Knowledge graphs. A primary output of an evolved pipeline is a *knowledge graph* (KG), produced before the downstream agent attempts solving a task. The KG is a context-grounded markdown briefing, `KG.md`, limited to 300 lines. Evolved KGs describe schemas, tables, columns, dbt models, outputs, and metrics. They also record joins, cardinalities, `source()/ref()` lineage, model dependencies, derivations, model grain, business rules, boundary behavior, null handling, fanout risks, validation checks, and task-specific pitfalls.

The KG is not an RDF store and contains no solution SQL. The authoring pipeline verifies physical table locations, column types, categorical values, null encodings, and row counts using `SHOW`, `DESCRIBE`, `INFORMATION_SCHEMA`, and `sql_execute`. This avoids relying on a potentially stale `_sources.yml` file.

The evolved program. The evolved artifact is the Python dictionary `_PIPELINE_SPEC` in `initial_program.py`. It jointly controls the KG authoring protocols, agent orchestration, and adherence test authoring protocol. Executing this specification produces the task-specific `KG.md` and `test_kg.py` supplied to the downstream coding agent. The mutator can improve this pipeline through the following three components.

1. **KG authoring protocols.** The `skills{}` catalog describes reusable procedures for constructing different parts of the KG, including its usage guide, metrics and business rules, and warehouse and model structure. Catalog entries contain metadata rather than complete skill bodies. During execution, assigned agents expand them into `runtime_skills/<name>.md`. These skills guide KG production and may be shared among agents in the authoring pipeline.
2. **Agent orchestration.** The `agents[]` list defines an ordered sequence of agent sessions that produce the KG. Each session has an editable prompt and a set of assigned skills. The mutator may rewrite prompts, add or remove agents, split or merge roles, reorder the pipeline, and redistribute skills. The agents share a sanitized workspace. This permits either a single agent or a pipeline with separate exploration, verification, and synthesis roles.
3. **Adherence test protocol.** A second form of context for the solver agent is executable tests derived from the KG. It is a protocol used during compilation rather than a section of `KG.md`. A fixed test authoring model consumes the KG together with this evolved protocol and produces `test_kg.py`. Evolution can therefore improve both the declarative guidance given to the solver and the way that guidance is translated into executable feedback.

Inference and evaluation. At inference time, the best evolved program is fixed and its executor materializes the evolved protocols and agent configuration. All evaluated agent roles use the same out-of-the-box `claude-sonnet-5` backbone and harness. These roles include the ordered agent sessions that construct the KG and the downstream agent that consumes it before executing a task. Only `KG.md` is registered as a skill for the Harbor agent. The internal `runtime_skills/*.md` files are not forwarded.

The sole model exception is the fixed adherence test author, which uses `claude-opus-4-8` with temperature 0.2 and a completion limit of 50,000 tokens. It translates `KG.md` and the evolved adherence test authoring protocol into `test_kg.py`. We hard pin this model because

`claude-sonnet-5` frequently produced truncated or syntactically invalid long test suites, particularly unterminated strings. The generated suite is supplied to the Harbor agent as self-correction feedback. It is not the hidden grader, which remains fixed throughout evaluation.

We note that adherence tests were generated for these evaluation runs but were not used for all runs. We therefore treat adherence testing as optional additional context rather than a uniform component of the reported results.

Search methods. The reported CoCoEvolve result uses the best program available after 200 iterations. We configure one island, an active population of 15 candidates, an archive capacity of 8, 8 parallel evaluations, and the P0/P1/P2 evolution graph over the dev pool. We compare against faithful ports of GEPA, Meta-Harness, and AdaEvolve using the same `claude-sonnet-5` backbone, downstream agent, equivalent iteration budget, and seed program `_PIPELINE_SPEC`.

I.2 Examples of Best Programs

I.2.1 Evolved Programs

The evolved specification p^* retains the initial three seed skills and authors an additional four. Two are pre-exploration environment checks: `kg-environment-preflight` confirms a usable, authorized warehouse, and `kg-dbt-profiles-default-path` pins the `~/.dbt/` default so fixes are persisted rather than session-local. The other two are a column-verification protocol requiring `DESCRIBE` evidence for every named column, and a worked-example protocol for weighted or ratio-based metrics. The `kg-adherence-test-author` description also tightens, requiring each stated business rule to be checked by live-SQL recomputation against source and banning negative-only assertions. In this example, the initial `architect` agent is the sole agent; no other agents were spawned.

Initial pipeline specification p_0

```
_PIPELINE_SPEC = {
    "skills": {
        "kg-agent-usage-guide": {
            "description": (
                "Author the Agent Usage Guide: what the downstream
    solver must "
                "produce (object names, grain) and how to know it is
    done. "
                "Generic guidance only --- no task-specific table or
    column names."
            ),
        },
        "kg-metrics-and-business-rules": {
            "description": (
                "Author Metrics & Business Rules: record numeric
    thresholds, "
                "windows, and formulas from the instruction. Do not
    invent values."
            ),
        },
        "kg-structure": {
            "description": (
                "Author Structure: confirmed sources (live warehouse
    location), "
                "required models/outputs (name, grain, columns,
    dependencies), "
                "and joins (keys, cardinality). Verified warehouse
    facts only --- "
                "do not restate the instruction. Generic guidance only
     --- no "
                "task-specific table or column names."
            ),
```

```
        },
        # Compile-time only (platform emits test_kg.py after KG.md ---
     not a KG.md section).
        "kg-adherence-test-author": {
            "description": (
                "One precise [KG-ADHERENCE] test per testable KG fact;
     failure "
                "messages quote the KG. Ban generic smoke tests."
            ),
        },
    },
    # Seed topology: one architect (look + write in the same session
    so
    # query results stay in context). Split only if explore_coverage
    shows
    # the look was skipped and a handoff would help.
    "agents": [
        {
            "name": "architect",
            "prompt": (
                "You architect the Knowledge Graph for task {task_name
    } (database "
                "{db_name}). Budget: {timeout_sec:.0f}s.\n\n"
                "The instruction is a brief, not the warehouse. Do not
     paraphrase "
                "it into KG.md --- that briefing is useless to the
    solver. Looking "
                "is not optional. Use `sql_execute` in THIS session
    before you "
                "write; you are not done until every table you will
    name has been "
                "DESCRIBE (or SHOW COLUMNS). Do not stop at SHOW
    DATABASES. "
                "Confirm each source's actual database.schema.table
    via SHOW / "
                "INFORMATION_SCHEMA. Do not stop because the
    instruction looks "
                "complete. For categorical / boolean / dirty-typed
    columns that "
                "feed a filter, join, or metric, GROUP BY / COUNT
    stored tokens, "
                "null encoding, and row counts. Name only what those
    queries "
                "confirmed. Verified "
                "facts only; no solution SQL. Then write `KG.md`
    (<=300 lines, "
                "first line `# Knowledge Graph: {task_name}`).\n\n"
                "Structure KG.md by following EVERY skill listed under
     \"Skills to "
                "author\" below. Author each as `runtime_skills/<name
    >.md` before "
                "you finish."
            ),
            "skills": [
                "kg-agent-usage-guide",
                "kg-metrics-and-business-rules",
                "kg-structure",
            ],
        },
    ],
}
```

Best evolved pipeline specification p^*

```
_PIPELINE_SPEC = {
    "skills": {
        "kg-agent-usage-guide": {
            "description": (
                "Author the Agent Usage Guide: what the downstream
    solver must "
                "produce (object names, grain) and how to know it is
    done. "
                "Generic guidance only | no task-specific table or
    column names."
            ),
        },
        "kg-metrics-and-business-rules": {
            "description": (
                "Author Metrics & Business Rules: record numeric
    thresholds, "
                "windows, and formulas from the instruction. Do not
    invent values."
            ),
        },
        "kg-structure": {
            "description": (
                "Author Structure: confirmed sources (live warehouse
    location), "
                "required models/outputs (name, grain, columns,
    dependencies), "
                "and joins (keys, cardinality). Verified warehouse
    facts only | "
                "do not restate the instruction. Generic guidance only
     | no "
                "task-specific table or column names."
            ),
        },
        # Compile-time only (platform emits test_kg.py after KG.md |
    not a KG.md section).
        "kg-adherence-test-author": {
            "description": (
                "One precise [KG-ADHERENCE] test per testable KG fact;
     failure "
                "messages quote the KG. Ban generic smoke tests. For
    any stated "
                "filter/dedup/exclusion business rule | especially a
    KG value "
                "explicitly labeled evidence rather than a prescribed
    target | "
                "the assertion must be a live SQL comparison:
    recompute the "
                "correct aggregate from source inside the test by
    embedding the "
                "exact stated predicate/dedup key, then assert the
    model output "
                "matches that live-recomputed value within a tight
    numeric "
                "tolerance. Do not assert only 'output != one known-
    wrong value' "
                "for a business rule | a negative-only check can pass
    while the "
                "rule is still violated a different way; it must be
    paired with "
                "a positive, rule-derived comparison."
            ),
```

```
        },
        "kg-column-verification-protocol": {
            "description": (
                "Before naming any specific column in Structure or
    Metrics & "
                "Business Rules, that column must have an associated "
                "DESCRIBE/SHOW COLUMNS (or information_schema.columns)
     result "
                "from THIS session. If a column was not schema-probed,
     omit it "
                "or mark it explicitly unverified rather than stating
    it as "
                "fact. Generic guidance only | no task-specific table
    or "
                "column names."
            ),
        },
        "kg-environment-preflight": {
            "description": (
                "Before other exploration, verify the actual database
    session "
                "context the downstream solver will inherit: confirm
    which "
                "warehouse is active/authorized for the working role (
    a session "
                "may start with no active warehouse, or a default
    warehouse name "
                "that does not exist or is not authorized). Record the
     "
                "confirmed-working warehouse and the exact persistent
    fix (e.g. "
                "profile/project-level config or an explicit USE
    WAREHOUSE in the "
                "model-build path) as an Agent Usage Guide fact | not
    a "
                "session-local env-file edit a later step could
    silently revert. "
                "Generic guidance only; no task-specific table or
    column names, "
                "and no assumption about which warehouse name is
    broken vs "
                "working in a given account."
            ),
        },
        "kg-dbt-profiles-default-path": {
            "description": (
                "Author a profiles-directory fact for the Agent Usage
    Guide when "
                "the task requires creating a new dbt project (rather
    than "
                "editing a pre-existing one): dbt's CLI defaults to "
                "`~/.dbt/profiles.yml` whenever `--profiles-dir`/"
                "`DBT_PROFILES_DIR` is not explicitly set on a given
    invocation. "
                "If the solver only writes `profiles.yml` to a custom
    path and "
                "relies on exporting `DBT_PROFILES_DIR` for its own
    manual `dbt "
                "run` calls, any OTHER process that later calls `dbt
    run`/`dbt "
                "build` in a fresh shell without that env var will
    fail at the "
```

```
                "default location instead. State explicitly: also
    place (or "
                "symlink) a copy of `profiles.yml` at the OS-default
    dbt "
                "profiles directory, not only the custom one. Verify
    by checking "
                "whether `~/.dbt/` already exists and what it contains
     before "
                "assuming a custom path is sufficient. Generic dbt-CLI
     behavior "
                "| no task-specific table or column names."
            ),
        },
        "kg-composite-formula-worked-example": {
            "description": (
                "For any metric in Metrics & Business Rules that
    involves a "
                "weighted composite score, a percentile/rank direction
     "
                "(ascending vs. inverted), or a ratio-to-median/cap-
    and-invert "
                "normalization, do not describe the formula in prose
    alone. "
                "Pick 1-2 real rows from the live warehouse, compute
    every "
                "intermediate factor by hand following the exact
    stated formula "
                "(including cap/invert/boundary behavior and NULL-
    fallback "
                "substitution), and record the final numeric result as
     a worked "
                "example in the KG. If the instruction's wording is
    genuinely "
                "ambiguous even after this exercise, state the
    ambiguity "
                "explicitly as an open question rather than silently
    picking one "
                "interpretation and presenting it as fact. Never
    invent "
                "thresholds not present in the instruction. Generic
    guidance "
                "only | no task-specific table/column names or copied
    assertion "
                "values."
            ),
        },
    },
    # Seed topology: one architect (look + write in the same session
    so
    # query results stay in context). Split only if explore_coverage
    shows
    # the look was skipped and a handoff would help.
    "agents": [
        {
            "name": "architect",
            "prompt": (
                "You architect the Knowledge Graph for task {task_name
    } (database "
                "{db_name}). Budget: {timeout_sec:.0f}s.\n\n"
                "The instruction is a brief, not the warehouse. Do not
     paraphrase "
                "it into KG.md | that briefing is useless to the
    solver. Looking "
```

```
            "is not optional. Use `sql_execute` in THIS session
    before you "
            "write; you are not done until every table you will
    name has been "
            "DESCRIBE (or SHOW COLUMNS). Do not stop at SHOW
    DATABASES. "
            "Confirm each source's actual database.schema.table
    via SHOW / "
            "INFORMATION_SCHEMA. Do not stop because the
    instruction looks "
            "complete. For categorical / boolean / dirty-typed
    columns that "
            "feed a filter, join, or metric, GROUP BY / COUNT
    stored tokens, "
            "null encoding, and row counts. Name only what those
    queries "
            "confirmed. Verified "
            "facts only; no solution SQL. Then write `KG.md` (≤300
     lines, "
            "first line `# Knowledge Graph: {task_name}`).\n\n"
            "Structure KG.md by following EVERY skill listed under
     \"Skills to "
            "author\" below. Author each as `runtime_skills/<name
    >.md` before "
            "you finish.\n\n"
            "Before any schema exploration, run the environment-
    preflight "
            "check: confirm the session actually has a usable,
    authorized "
            "warehouse (do not assume the default env/session
    warehouse "
            "works | it may not exist or may be unauthorized).
    Also confirm "
            "the dbt profiles-directory resolution fact from "
            "kg-dbt-profiles-default-path when the task builds a
    new dbt "
            "project. Fold both the confirmed-working warehouse/
    persistent "
            "fix and the profiles-directory fact into the Agent
    Usage Guide "
            "section of KG.md; do not add new top-level sections
    for them "
            "and do not exceed the 300-line budget.\n\n"
            "For any weighted composite score or percentile/ratio-
    based "
            "metric in Metrics & Business Rules, follow "
            "kg-composite-formula-worked-example: compute at least
     one "
            "worked numeric example from live rows before writing
    the "
            "formula down, rather than describing it in prose
    alone."
        ),
        "skills": [
            "kg-agent-usage-guide",
            "kg-metrics-and-business-rules",
            "kg-structure",
            "kg-environment-preflight",
            "kg-column-verification-protocol",
            "kg-dbt-profiles-default-path",
            "kg-composite-formula-worked-example",
        ],
    },
```

```
    ],
}
```

I.2.2 Final Pipeline Specification Artifacts

Below is `KG.md` for task `dbt-fix-marketing-attribution`, produced by p^*. It records verified tables, columns, data values, and environment settings for the downstream solver agent. It also flags differences between the task instruction and the live warehouse instead of making unsupported assumptions.

Final `KG.md` for `dbt-fix-marketing-attribution` produced by pipeline specification p^*

```
# Knowledge Graph: dbt-fix-marketing-attribution

## Structure

**Confirmed sources** (database `DBT_BENCH_RETAIL`, schema `MAIN`; all
     confirmed via `SHOW TABLES`/`SHOW VIEWS`/`INFORMATION_SCHEMA.
    TABLES` + `DESCRIBE VIEW`):

- `DBT_BENCH_RETAIL.MAIN.STG_GA__SESSIONS` (VIEW, 5053 rows, 5053
    distinct `SESSION_ID`, 0 NULL `SESSION_ID`/`SESSION_START`)
  Columns: `SESSION_ID`, `VISITOR_ID`, `CUSTOMER_ID`, `CHANNEL_ID`, `
    SESSION_START` (VARCHAR | timestamp-like string, NOT native date/
    timestamp), `SESSION_END` (VARCHAR), `DURATION_SECONDS` (NUMBER),
    `PAGE_VIEWS` (NUMBER), `LANDING_PAGE`, `EXIT_PAGE`, `REFERRER`, `
    UTM_SOURCE`, `UTM_MEDIUM`, `UTM_CAMPAIGN`, `DEVICE_TYPE`, `BROWSER
    `, `OS`, `IP_ADDRESS`, `COUNTRY`, `IS_CONVERTED` (VARCHAR), `
    ORDER_ID` (VARCHAR, always NULL in this env | 0/5053 populated), `
    CREATED_AT`, plus metadata cols `_LOADED_AT`, `_SOURCE_SYSTEM`, `
    _BATCH_ID`, `_ROW_NUMBER`, `_ROW_HASH`.

- `DBT_BENCH_RETAIL.MAIN.STG_GA__EVENTS` (VIEW, 15047 rows)
  Columns: `EVENT_ID`, `SESSION_ID` (1131 rows NULL), `EVENT_TYPE`, `
    EVENT_NAME`, `EVENT_TIMESTAMP` (VARCHAR; 0 literal NULLs but
    3801/15047 rows fail `TRY_CAST(... AS TIMESTAMP)` | dirty/
    malformed timestamp strings), `PAGE_URL`, `ELEMENT_ID`, `
    ELEMENT_CLASS`, `PRODUCT_ID`, `EVENT_VALUE` (VARCHAR storing
    numeric-looking decimal strings, e.g. `86.52592478731349`), `
    EVENT_DATA`, `CREATED_AT`, plus same metadata cols as sessions.

- `DBT_BENCH_RETAIL.MAIN.INT_SESSIONS_EVENTS_JOINED` (VIEW, LEFT JOIN
    of the two above; 15775 rows vs 5053 distinct `SESSION_ID` →
    confirmed fanout ~3.1x rows per session)
  Columns (15 total, confirmed via `DESCRIBE VIEW`): `SESSION_ID`, `
    VISITOR_ID`, `CUSTOMER_ID`, `CHANNEL_ID`, `SESSION_START`, `
    SESSION_END`, `DURATION_SECONDS`, `PAGE_VIEWS`, `T2_EVENT_ID`, `
    T2_EVENT_TYPE`, `T2_EVENT_NAME`, `T2_EVENT_TIMESTAMP`, `
    T2_PAGE_URL`, `T2_ELEMENT_ID`, `T2_ELEMENT_CLASS`.
  **`T2_EVENT_VALUE` DOES NOT EXIST in this view in this environment**
     | confirmed absent from the 15-column `DESCRIBE VIEW` output.
    This matches the instruction's warned "may omit `t2_event_value`
    entirely" branch, and in this live warehouse that branch is the
    actual case: revenue cannot be sourced from `
    int_sessions_events_joined` here.

Rejected/decoy candidates seen in schema search (do not use | wrong
    grain or duplicative of the above): `STG_RAW_GA__SESSIONS*`, `
    STG_SESSIONS*`, `RAW_GA.SESSIONS*`, `STG_RAW_GA__EVENTS*`, `
    FCT_MULTI_TOUCH_ATTRIBUTION`, `FCT_RECONSTRUCTED_SESSIONS`, `
    RPT_SESSION_HOURLY_DISTRIBUTION` | these are separate staging-
    history/rebuild variants and pre-existing marts unrelated to the
```

```
    required `int_sessions_events_joined` / `stg_ga__sessions` / `
    stg_ga__events` lineage named by the instruction.

**Required output** (per instruction, not re-verified in warehouse
    since it does not yet exist): dbt model `marts/marketing/
    rpt_attribution_fixed.sql`, materializing to schema `main` (cloud-
    warehouse backend) or `analytics` (DuckDB backend). Grain: one row
     per (`attribution_date`, `channel`, `medium`, `campaign`).
    Depends on `stg_ga__sessions`, `stg_ga__events`, `
    int_sessions_events_joined` (all confirmed above).

**Joins:**
- `stg_ga__sessions.SESSION_ID` = `int_sessions_events_joined.
    SESSION_ID` | this is the join already baked into the `
    int_sessions_events_joined` view (LEFT JOIN sessions→events);
    cardinality 1:many, fanout confirmed 5053 sessions → 15775 joined
     rows.
- `stg_ga__events.SESSION_ID` = `stg_ga__sessions.SESSION_ID` |
    1131/15047 event rows have NULL `SESSION_ID` (orphan events,
    cannot join).

## Metrics & Business Rules
See `runtime_skills/kg-metrics-and-business-rules.md` for the generic
    capture protocol. Task-specific verified facts:

**Windows:** 90-day trailing window anchored to `MAX(CAST(
    session_start AS DATE))` computed from `stg_ga__sessions` itself (
    per instruction) | NOT anchored to current wall-clock date.
    Confirmed `SESSION_START` values in `stg_ga__sessions` range from
    `1765-02-04` to `2026-01-03`; 46/5053 rows are pre-2020 outlier/
    garbage dates, 5007/5053 fall in a normal recent range. The MAX
    used for the 90-day anchor will be pulled from this same
    distribution (max observed = `2026-01-03 19:00:08`).

**Formula:** `conversion_rate = conversions / sessions`, divide-by-
    zero must be handled (per instruction, `NULLIF`-style guard),
    rounded to 4 decimals.

**Default/mapping rules (verified against actual NULL counts):**
- `UTM_SOURCE` NULL → ''direct'': 387/5053 sessions have NULL `
    UTM_SOURCE`.
- `UTM_MEDIUM` NULL → ''none'': 402/5053 sessions have NULL `
    UTM_MEDIUM`.
- `UTM_CAMPAIGN` NULL → ''none'': 3663/5053 sessions have NULL `
    UTM_CAMPAIGN` (majority of sessions are uncampaigned).

**Fallback/precedence rule (verified, not assumed):** Instruction says
     prefer `t2_event_value` from `int_sessions_events_joined`, else
    use `event_value` from `stg_ga__events`. In this environment, `
    t2_event_value` is confirmed absent (see Structure section) | the
    fallback branch to `stg_ga__events.EVENT_VALUE` is the one that
    applies here.

**Dedup/grain rule:** Instruction requires summing revenue per session
     first, then aggregating to the final (date, channel, medium,
    campaign) grain, to avoid inflation from the confirmed 1:˜3.1
    session→event-row fanout in `int_sessions_events_joined`.

**Verified discrepancies (facts only, not resolved here):**
- Instruction's conversion rule states `is_converted = 1`. Actual
    stored tokens in `STG_GA__SESSIONS.IS_CONVERTED` (VARCHAR column,
    confirmed via `GROUP BY`): ''0''=1532, ''1''=47, ''N''=997, ''Y
    ''=35, ''false''=2364, ''true''=78 (total 5053). Truthy-token
```

```
    candidates (''1'',''Y'',''true'') sum to 160 sessions (~3.2%);
    falsy-token candidates (''0'',''N'',''false'') sum to 4893. A
    literal `= 1` comparison against this VARCHAR column will not
    match the ''Y''/''true'' rows.
- Instruction's revenue/conversion-event rule states `event_name = '
    purchase'` (and `t2_event_name = 'purchase'`). A case-insensitive
    scan (`ILIKE '%purchas%'`) across all 15047 rows of `
    STG_GA__EVENTS.EVENT_NAME` returned **0 matching rows**. Distinct
    `EVENT_NAME` values actually present (51 distinct, 1216 rows
    literal NULL): ''View Product''(1420), ''Search''(1418), ''Filter
    ''(1410), ''Video Play''(1406), ''Click''(1391), ''Form Submit
    ''(1384), ''Add To Cart''(1357), ''Sort''(1344), ''Scroll''(1333),
     ''Remove From Cart''(1321), plus ~40 additional low-frequency
    dirty tokens of the form ''event_N.0'' (each count 1{2, ~46 rows
    total). The same absence of ''purchase'' holds in `
    INT_SESSIONS_EVENTS_JOINED.T2_EVENT_NAME` (11 distinct groups
    observed, same 10 named categories plus NULL, no ''purchase'').
- `STG_GA__EVENTS.EVENT_VALUE` is stored as VARCHAR containing decimal
     numeric strings (e.g. ''86.52592478731349'') | requires explicit
    cast to a numeric type before summing.
- `SESSION_START`, `SESSION_END`, `EVENT_TIMESTAMP` are all stored as
    VARCHAR, not native DATE/TIMESTAMP, in this environment |
    confirmed via `DESCRIBE VIEW`. `TRY_CAST(SESSION_START AS
    TIMESTAMP)` succeeded on sampled rows; `TRY_CAST(EVENT_TIMESTAMP
    AS TIMESTAMP)` failed on 3801/15047 rows (dirty timestamp strings
    in the events table specifically, not the sessions table).
- `STG_GA__SESSIONS.CUSTOMER_ID` is NULL on 3632/5053 rows (guest/
    anonymous sessions) and `ORDER_ID` is NULL on all 5053 rows in
    this environment.

## Environment & Execution Notes

**Fresh-session, zero-override check:** A new `sql_execute` session
    with no prior `USE ...` statements resolved an authorized compute
    warehouse, the benchmark database, and the expected schema without
     manual overrides.

**Compute verification:** `SELECT COUNT(*) FROM DBT_BENCH_RETAIL.MAIN.
    STG_GA__SESSIONS` returned `5053` rows, confirming that the
    default warehouse provides usable compute in this environment.

**Schema alignment (checked, not assumed):** The task instruction
    requires the warehouse output model to land in schema `main` via a
     `generate_schema_name` macro override. The live session already
    resolves to `MAIN` with zero overrides. These agree in this
    environment, so there is no discrepancy between the instruction's
    schema literal and the session default.

**Instruction to downstream solver:** No environment fix was needed to
     produce this KG | the default warehouse/role/database/schema all
    resolved and worked with zero manual overrides. If, in your own
    session, you find you must override any of these (e.g. to get `dbt
     run` or your own self-tests to pass), leave that override in the
    **persisted** profile/environment used for final scoring (e.g. `
    profiles.yml`, environment variables, or whatever config file
    supplies these values at grading time) | do NOT revert it back
    afterward under the belief that reverting is "cleanup." The
    independent grading connection opens its own fresh session and
    inherits only the persisted defaults, not anything you fixed
    transiently in your live session. Immediately before finishing,
    verify with a second, genuinely fresh connection/session (not the
    one you just patched) that the build still succeeds under the
    exact environment state that will be graded. Do not blanket-
```

```
    override the schema/database literal in `profiles.yml`/macros
    based solely on this session's env var reading | the alignment
    check above already confirms they agree in this environment, so no
     override is warranted here without new evidence.

## Known Ambiguities & Risk Flags

- **`is_converted` truthy definition** (real COUNT queries against `
    DBT_BENCH_RETAIL.MAIN.STG_GA__SESSIONS`, 5053 total rows):
    Instruction states a session is converted if `is_converted = 1`.
    Interpretation A | literal `IS_CONVERTED = '1'` → **47** sessions
    . Interpretation B | truthy-token set `IS_CONVERTED IN ('1','Y','
    true')` → **160** sessions (`'1'`=47, `'Y'`=35, `'true'`=78).
    Complementary falsy-token set `IS_CONVERTED IN ('0','N','false')`
    → **4893** sessions (`'0'`=1532, `'N'`=997, `'false'`=2364).
    Disagreement: 47 vs 160 | a ~3.4x difference in how many sessions
    count as converted. Affects: `conversions` numerator, `
    conversion_rate`, and which sessions' revenue is eligible to be
    summed. Not resolved here | solver must choose an interpretation
    and be prepared to justify it; both counts are real and verified,
    not modeled.

- **`event_name = 'purchase'` literal match** (real COUNT query
    against `DBT_BENCH_RETAIL.MAIN.STG_GA__EVENTS`, 15047 total rows,
    1216 NULL `EVENT_NAME`): Instruction states conversion/revenue
    events are identified by `event_name = 'purchase'` (or `
    t2_event_name = 'purchase'` in the joined view). Literal match `
    EVENT_NAME = 'purchase'` → **0** rows. Case-insensitive fuzzy
    match `EVENT_NAME ILIKE '%purchas%'` → **0** rows. Distinct-value
     scan confirms 51 distinct non-NULL `EVENT_NAME` values actually
    present, none resembling purchase (top values: `'View Product
    '`=1420, `'Search'`=1418, `'Filter'`=1410, `'Video Play'`=1406, `'
    Click'`=1391, `'Form Submit'`=1384, `'Add To Cart'`=1357, `'Sort
    '`=1344, `'Scroll'`=1333, `'Remove From Cart'`=1321, plus ~40 low-
    frequency dirty `'event_N.0'` tokens). Risk: under the literal
    instruction rule, `attributed_revenue` will be **0 for every row**
     of the final output, because the named conversion-event literal
    does not exist anywhere in this environment's data under any
    casing tried. This is a structural risk, not a two-way count
    disagreement | flagging it rather than substituting a different
    event-name literal on the solver's behalf.

Runtime skill files authored per assignment (full protocols, generic/
    reusable, no task-specific names inside them):
- `runtime_skills/kg-agent-usage-guide.md`
- `runtime_skills/kg-metrics-and-business-rules.md`
- `runtime_skills/kg-structure.md`
- `runtime_skills/kg-ambiguity-guardrails.md`
- `runtime_skills/kg-environment-handoff.md`
```

The suite below turns KG statements into executable checks. Some tests verify recorded facts against the live data warehouse, while others recompute business rules from source data and compare them with the solver's output. Their failure messages point the solver to the relevant KG statement, helping it locate and fix mismatches.

Adherence test suite `test_kg.py` *authored by* `claude-opus-4-8`

```
"""Self-correction test suite for task: dbt-fix-marketing-attribution.

[KG-ADHERENCE] tests encode specific facts stated in the Knowledge
    Graph (KG):
```

```
required output model, schema, grain, columns, metric formulas, filter predicates,
UTM default mappings, dedup-before-aggregate rules, and the confirmed numeric
anchors (160 truthy is_converted sessions, 1216 NULL event_name rows, 5053 sessions).

[UNIVERSAL] tests encode output-quality invariants the KG's Output requirements
imply (no NULL required columns, conversion_rate in [0,1], conversions <= sessions)
and cross-source reconciliation against the confirmed sources.

These tests are derived EXCLUSIVELY from the KG | not from instruction.md, not from
any solution SQL, and not from the hidden grader. Passing them is strong evidence the
agent followed the KG. A failing test points to a specific KG sentence to re-read
(quoted in each failure message).
"""

import os
import sys

_here = os.path.dirname(os.path.abspath(__file__))
for _p in [_here, os.path.normpath(os.path.join(_here, "..", ".."))]:
    if os.path.exists(os.path.join(_p, "test_backbone.py")):
        sys.path.insert(0, _p)
        break
from test_backbone import execute_query, execute_scalar, make_conn_fixture, require

# -- Task constants ---------------------------------------------------------
AGENT_PROJECT_DIR = "/app/dbt_project"

# Output schema: main (cloud warehouse) or analytics (DuckDB). We resolve per db_type.
SCHEMA_DUCKDB = "analytics"
SCHEMA_WAREHOUSE = "main"

MODEL = "rpt_attribution_fixed"
MODEL_FILE = f"{AGENT_PROJECT_DIR}/models/marts/marketing/rpt_attribution_fixed.sql"

# Confirmed sources (database DBT_BENCH_RETAIL, schema MAIN).
SRC_SCHEMA = "main"
STG_SESSIONS = "stg_ga__sessions"
STG_EVENTS = "stg_ga__events"
INT_JOINED = "int_sessions_events_joined"

# Expected grain columns per KG ("one row per (attribution_date, channel, medium, campaign)").
GRAIN_COLS = ["attribution_date", "channel", "medium", "campaign"]

conn = make_conn_fixture(check_paths=[MODEL_FILE])


def _out_schema(db_type):
    return SCHEMA_DUCKDB if db_type == "duckdb" else SCHEMA_WAREHOUSE
```

```
def _fqn(schema, table):
    return schema + "." + table


def _columns(c, db_type, schema, table):
    if db_type == "duckdb":
        rows = execute_query(
            c, db_type,
            "SELECT name FROM pragma_table_info('" + schema + "." +
    table + "')",
        )
        return set(str(r[0]).lower() for r in rows)
    rows = execute_query(
        c, db_type,
        "SELECT column_name FROM information_schema.columns "
        "WHERE lower(table_schema) = lower('" + schema + "') "
        "AND lower(table_name) = lower('" + table + "')",
    )
    return set(str(r[0]).lower() for r in rows)


# -- Prerequisite ------------------------------------------------------
    --------
def test_dbt_run_prerequisite(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(schema, MODEL),
    )
    require(
        cnt is not None and cnt > 0,
        "[KG-ADHERENCE] Required output dbt model 'marts/marketing/"
        "rpt_attribution_fixed.sql' must be built and populated | run
    dbt first.",
    )


# -- Source sanity anchors (confirmed in KG) ---------------------------
    --------
def test_source_sessions_row_count(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS),
    )
    require(
        cnt == 5053,
        "[UNIVERSAL] KG: 'STG_GA__SESSIONS (VIEW, 5053 rows)'.",
    )


def test_source_sessions_distinct_session_id(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(DISTINCT session_id) FROM " + _fqn(SRC_SCHEMA,
    STG_SESSIONS),
    )
    require(
        cnt == 5053,
        "[UNIVERSAL] KG: '5053 distinct SESSION_ID, 0 NULL SESSION_ID
    '.",
```

```
    )


def test_source_events_row_count(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_EVENTS),
    )
    require(
        cnt == 15047,
        "[UNIVERSAL] KG: 'STG_GA__EVENTS (VIEW, 15047 rows)'.",
    )


def test_source_int_joined_fanout(conn):
    c, db_type = conn
    total = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, INT_JOINED),
    )
    distinct = execute_scalar(
        c, db_type,
        "SELECT COUNT(DISTINCT session_id) FROM " + _fqn(SRC_SCHEMA,
    INT_JOINED),
    )
    require(
        total == 15775 and distinct == 5053,
        "[KG-ADHERENCE] KG: 'INT_SESSIONS_EVENTS_JOINED ... 15775 rows
     vs 5053 "
        "distinct SESSION_ID -> fanout ~3.1x rows per session'. This
    fanout is "
        "why revenue must be summed per session BEFORE final
    aggregation.",
    )


def test_int_joined_lacks_t2_event_value(conn):
    c, db_type = conn
    cols = _columns(c, db_type, SRC_SCHEMA, INT_JOINED)
    require(
        "t2_event_value" not in cols,
        "[KG-ADHERENCE] KG: 'T2_EVENT_VALUE DOES NOT EXIST in this
    view in this "
        "environment' | the fallback to stg_ga__events.EVENT_VALUE
    applies here.",
    )


# -- is_converted truthy anchor (160) ---------------------------------
    ---------
def test_is_converted_truthy_anchor_160(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE is_converted IN ('1','Y','true')",
    )
    require(
        cnt == 160,
        "[KG-ADHERENCE] KG: truthy-token set IS_CONVERTED IN ('1','Y
    ','true') "
        "-> 160 sessions ('1'=47,'Y'=35,'true'=78).",
```

```
    )


def test_is_converted_literal_one_anchor_47(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE is_converted = '1'",
    )
    require(
        cnt == 47,
        "[KG-ADHERENCE] KG: literal IS_CONVERTED = '1' -> 47 sessions
    "
        "(Interpretation A of the is_converted ambiguity).",
    )


def test_is_converted_falsy_anchor_4893(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE is_converted IN ('0','N','false')",
    )
    require(
        cnt == 4893,
        "[KG-ADHERENCE] KG: falsy-token set IS_CONVERTED IN ('0','N','
    false') "
        "-> 4893 sessions.",
    )


# -- event_name = 'purchase' structural risk -------------------------
    ---------
def test_event_name_null_anchor_1216(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_EVENTS) +
        " WHERE event_name IS NULL",
    )
    require(
        cnt == 1216,
        "[KG-ADHERENCE] KG: 'STG_GA__EVENTS ... 1216 NULL EVENT_NAME
    '.",
    )


def test_event_name_purchase_absent(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_EVENTS) +
        " WHERE event_name = 'purchase'",
    )
    require(
        cnt == 0,
        "[KG-ADHERENCE] KG: literal EVENT_NAME = 'purchase' -> 0 rows.
     "
        "Structural risk: attributed_revenue may be 0 for every output
     row.",
    )
```

```
# -- UTM default-mapping NULL anchors ---------------------------------
    ---------
def test_utm_source_null_anchor_387(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE utm_source IS NULL",
    )
    require(
        cnt == 387,
        "[KG-ADHERENCE] KG: 'UTM_SOURCE NULL -> direct: 387/5053
    sessions'.",
    )


def test_utm_medium_null_anchor_402(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE utm_medium IS NULL",
    )
    require(
        cnt == 402,
        "[KG-ADHERENCE] KG: 'UTM_MEDIUM NULL -> none: 402/5053
    sessions'.",
    )


def test_utm_campaign_null_anchor_3663(conn):
    c, db_type = conn
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        " WHERE utm_campaign IS NULL",
    )
    require(
        cnt == 3663,
        "[KG-ADHERENCE] KG: 'UTM_CAMPAIGN NULL -> none: 3663/5053
    sessions'.",
    )


# -- Output model: existence, columns, grain, uniqueness -------------
    --------
def test_model_exists_as_table(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    if db_type == "duckdb":
        rows = execute_query(
            c, db_type,
            "SELECT table_name FROM information_schema.tables "
            "WHERE lower(table_schema)=lower('" + schema + "') "
            "AND lower(table_name)=lower('" + MODEL + "')",
        )
    else:
        rows = execute_query(
            c, db_type,
            "SELECT table_name FROM information_schema.tables "
            "WHERE lower(table_schema)=lower('" + schema + "') "
```

```
            "AND lower(table_name)=lower('" + MODEL + "')",
        )
    require(
        len(rows) >= 1,
        "[KG-ADHERENCE] KG: Required output dbt model 'marts/marketing
    /"
        "rpt_attribution_fixed.sql' materializing to schema '" +
    schema + "'.",
    )


def test_model_has_grain_columns(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    cols = _columns(c, db_type, schema, MODEL)
    missing = [g for g in GRAIN_COLS if g not in cols]
    require(
        len(missing) == 0,
        "[KG-ADHERENCE] KG grain: 'one row per (attribution_date,
    channel, medium, "
        "campaign)'. Missing grain columns: " + str(missing),
    )


def test_model_has_metric_columns(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    cols = _columns(c, db_type, schema, MODEL)
    needed = ["sessions", "conversions", "conversion_rate"]
    missing = [n for n in needed if n not in cols]
    require(
        len(missing) == 0,
        "[KG-ADHERENCE] KG metrics: sessions, conversions,
    conversion_rate "
        "(= conversions / sessions). Missing: " + str(missing),
    )


def test_model_has_revenue_column(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    cols = _columns(c, db_type, schema, MODEL)
    require(
        "attributed_revenue" in cols,
        "[KG-ADHERENCE] KG: attributed_revenue derived from
    t2_event_value else "
        "stg_ga__events.EVENT_VALUE, summed per session first.",
    )


def test_model_grain_is_unique(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    dup = execute_query(
        c, db_type,
        "SELECT attribution_date, channel, medium, campaign, COUNT(*)
    AS n "
        "FROM " + _fqn(schema, MODEL) +
        " GROUP BY attribution_date, channel, medium, campaign HAVING
    COUNT(*) > 1",
    )
    require(
        len(dup) == 0,
```

```
        "[KG-ADHERENCE] KG grain: 'one row per (attribution_date,
    channel, medium, "
        "campaign)' | the grain key must be unique (no duplicate
    combinations).",
    )


def test_model_row_count_positive(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    cnt = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(schema, MODEL),
    )
    require(
        cnt is not None and cnt > 0,
        "[UNIVERSAL] Output model must produce at least one grain row
    .",
    )


# -- Data quality invariants (Output requirements) ------------------
    --------
def test_no_null_required_columns(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT * FROM " + _fqn(schema, MODEL) +
        " WHERE attribution_date IS NULL OR channel IS NULL OR medium
    IS NULL "
        "OR campaign IS NULL OR sessions IS NULL OR conversions IS
    NULL "
        "OR conversion_rate IS NULL",
    )
    require(
        len(rows) == 0,
        "[UNIVERSAL] Output requirements forbid NULL in required
    columns "
        "(grain keys, sessions, conversions, conversion_rate).",
    )


def test_conversion_rate_in_range(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT conversion_rate FROM " + _fqn(schema, MODEL) +
        " WHERE conversion_rate < 0.0 OR conversion_rate > 1.0",
    )
    require(
        len(rows) == 0,
        "[UNIVERSAL] KG: 'conversion_rate = conversions / sessions'
    must be in "
        "[0.0, 1.0].",
    )


def test_conversions_le_sessions(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
```

```
        c, db_type,
        "SELECT sessions, conversions FROM " + _fqn(schema, MODEL) +
        " WHERE conversions > sessions",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: converted sessions are a subset of all
    sessions | "
        "conversions must never exceed sessions.",
    )


def test_sessions_nonnegative(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT sessions FROM " + _fqn(schema, MODEL) +
        " WHERE sessions IS NULL OR sessions < 0",
    )
    require(
        len(rows) == 0,
        "[UNIVERSAL] sessions must be a non-negative count of distinct
     sessions.",
    )


def test_conversions_nonnegative(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT conversions FROM " + _fqn(schema, MODEL) +
        " WHERE conversions IS NULL OR conversions < 0",
    )
    require(
        len(rows) == 0,
        "[UNIVERSAL] conversions must be a non-negative count.",
    )


# -- conversion_rate formula recomputation ----------------------------
    ---------
def test_conversion_rate_formula(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # divide-by-zero guard: rate must be 0 (or NULL-guarded) when
    sessions=0.
    rows = execute_query(
        c, db_type,
        "SELECT sessions, conversions, conversion_rate FROM " + _fqn(
    schema, MODEL) +
        " WHERE sessions > 0 AND "
        "ABS(conversion_rate - ROUND(CAST(conversions AS DOUBLE)/
    sessions, 4)) > 0.0001",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 'conversion_rate = conversions / sessions
    ... rounded "
        "to 4 decimals' with divide-by-zero handled.",
    )
```

```
def test_conversion_rate_zero_sessions_guard(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT * FROM " + _fqn(schema, MODEL) +
        " WHERE sessions = 0 AND conversion_rate IS NULL",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 'divide-by-zero must be handled (NULLIF-
    style guard)' | "
        "conversion_rate should not be NULL/undefined when sessions =
    0.",
    )


# -- Cross-model reconciliation against confirmed sources ------------
    ---------
def test_total_sessions_reconciles_to_90day_window(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # KG: 90-day trailing window anchored to MAX(CAST(session_start AS
     DATE))
    # from stg_ga__sessions itself, counting distinct session_id.
    src_total = execute_scalar(
        c, db_type,
        "WITH anchored AS ("
        " SELECT session_id, CAST(session_start AS DATE) AS sd,"
        " (SELECT MAX(CAST(session_start AS DATE)) FROM " +
        _fqn(SRC_SCHEMA, STG_SESSIONS) + ") AS max_sd"
        " FROM " + _fqn(SRC_SCHEMA, STG_SESSIONS) +
        ") SELECT COUNT(DISTINCT session_id) FROM anchored"
        " WHERE sd > max_sd - 90",
    )
    model_total = execute_scalar(
        c, db_type,
        "SELECT SUM(sessions) FROM " + _fqn(schema, MODEL),
    )
    require(
        src_total is not None and model_total is not None
        and abs(int(model_total) - int(src_total)) <= 0,
        "[UNIVERSAL] KG: '90-day trailing window anchored to "
        "MAX(CAST(session_start AS DATE))' with distinct session
    counting | "
        "SUM(sessions) in the mart must equal distinct sessions in
    that window "
        "(model=" + str(model_total) + ", source=" + str(src_total) +
    ").",
    )


def test_attribution_date_within_90day_window(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    max_sd = execute_scalar(
        c, db_type,
        "SELECT MAX(CAST(session_start AS DATE)) FROM " +
        _fqn(SRC_SCHEMA, STG_SESSIONS),
    )
    rows = execute_query(
        c, db_type,
```

```
        "SELECT attribution_date FROM " + _fqn(schema, MODEL) +
        " WHERE CAST(attribution_date AS DATE) <= "
        "(SELECT MAX(CAST(session_start AS DATE)) - 90 FROM " +
        _fqn(SRC_SCHEMA, STG_SESSIONS) + ")",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 90-day trailing window anchored to "
        "MAX(CAST(session_start AS DATE)) (max observed 2026-01-03). "
        "All attribution_date values must fall inside that window (
    max_sd=" +
        str(max_sd) + ").",
    )


def test_channel_defaults_direct_present_or_valid(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # KG: UTM_SOURCE NULL -> 'direct' default. No NULL/empty channel
    should remain.
    rows = execute_query(
        c, db_type,
        "SELECT channel FROM " + _fqn(schema, MODEL) +
        " WHERE channel IS NULL",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 'UTM_SOURCE NULL -> direct' | channel must
     never be "
        "NULL in the output.",
    )


def test_medium_defaults_none_present_or_valid(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT medium FROM " + _fqn(schema, MODEL) +
        " WHERE medium IS NULL",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 'UTM_MEDIUM NULL -> none' | medium must
    never be NULL.",
    )


def test_campaign_defaults_none_present_or_valid(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT campaign FROM " + _fqn(schema, MODEL) +
        " WHERE campaign IS NULL",
    )
    require(
        len(rows) == 0,
        "[KG-ADHERENCE] KG: 'UTM_CAMPAIGN NULL -> none' | campaign
    must never be "
        "NULL.",
    )
```

```
# -- Revenue: dedup-before-aggregate & non-negativity ----------------
    ---------
def test_attributed_revenue_nonnegative(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    rows = execute_query(
        c, db_type,
        "SELECT attributed_revenue FROM " + _fqn(schema, MODEL) +
        " WHERE attributed_revenue < 0",
    )
    require(
        len(rows) == 0,
        "[UNIVERSAL] attributed_revenue must be non-negative.",
    )


def test_revenue_not_inflated_by_fanout(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # KG: sum revenue per session first, then aggregate | to avoid
    1:~3.1 fanout
    # inflation. Total attributed_revenue must not exceed the total
    per-session
    # summed EVENT_VALUE upper bound across the whole events table.
    total_model = execute_scalar(
        c, db_type,
        "SELECT COALESCE(SUM(attributed_revenue),0) FROM " + _fqn(
    schema, MODEL),
    )
    # Per-session summed event value (numeric cast), a safe upper
    bound.
    if db_type == "duckdb":
        cast_expr = "TRY_CAST(event_value AS DOUBLE)"
    else:
        cast_expr = "TRY_CAST(event_value AS DOUBLE)"
    upper = execute_scalar(
        c, db_type,
        "WITH per_session AS ("
        " SELECT session_id, SUM(" + cast_expr + ") AS rev"
        " FROM " + _fqn(SRC_SCHEMA, STG_EVENTS) +
        " WHERE session_id IS NOT NULL GROUP BY session_id"
        ") SELECT COALESCE(SUM(rev),0) FROM per_session",
    )
    require(
        total_model is not None and upper is not None
        and float(total_model) <= float(upper) + 0.01,
        "[KG-ADHERENCE] KG: 'sum revenue per session first, then
    aggregate ... to "
        "avoid inflation from the confirmed 1:~3.1 session->event-row
    fanout'. "
        "Total attributed_revenue (" + str(total_model) + ") must not
    exceed the "
        "per-session summed EVENT_VALUE bound (" + str(upper) + ").",
    )


def test_revenue_zero_when_purchase_absent(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # KG structural risk: EVENT_NAME='purchase' has 0 matches under
    any casing.
```

```
    # A spec-faithful (literal 'purchase') solution yields
    attributed_revenue = 0
    # for every row. If the agent used a substitute event token, this
    test flags
    # it so the choice is deliberate & justified rather than
    accidental.
    purchase_events = execute_scalar(
        c, db_type,
        "SELECT COUNT(*) FROM " + _fqn(SRC_SCHEMA, STG_EVENTS) +
        " WHERE event_name = 'purchase'",
    )
    total_model = execute_scalar(
        c, db_type,
        "SELECT COALESCE(SUM(attributed_revenue),0) FROM " + _fqn(
    schema, MODEL),
    )
    if purchase_events == 0:
        require(
            float(total_model) == 0.0,
            "[KG-ADHERENCE] KG structural risk: literal EVENT_NAME='
    purchase' -> "
            "0 rows, so under the spec's purchase rule
    attributed_revenue is 0 for "
            "every output row. Got total=" + str(total_model) + " | if
     non-zero, "
            "the agent substituted a different event token (must be
    justified).",
        )
    else:
        require(True, "")


# -- Metric consistency: conversions must reflect converted sessions -
    --------
def test_conversions_consistent_with_truthy_bound(conn):
    c, db_type = conn
    schema = _out_schema(db_type)
    # KG: converted numerator depends on is_converted interpretation
    (47 literal
    # or 160 truthy). Total conversions in the mart must not exceed
    the truthy
    # upper bound of 160 (within the 90-day window it is <= 160).
    total_conv = execute_scalar(
        c, db_type,
        "SELECT COALESCE(SUM(conversions),0) FROM " + _fqn(schema,
    MODEL),
    )
    require(
        total_conv is not None and int(total_conv) <= 160,
        "[KG-ADHERENCE] KG is_converted ambiguity: truthy set ('1','Y
    ','true') = "
        "160 sessions is the maximum eligible. SUM(conversions)=" +
        str(total_conv) + " must not exceed 160.",
    )
```

J Math & System Benchmark Optimization

J.1 Detailed Methodology

All math and ADRS runs share a 100-iteration budget and a matched backbone (`claude-sonnet-5` or `openai-gpt-5.6-terra`). AdaEvolve, OpenEvolve, and GEPA mutate through a stateless completion; CoCoEvolve uses the agentic loop of Appendix D.1,

skipping the InfoGain branch because these tasks have no instance pool. The AdaEvolve row in Table 4 reruns AdaEvolve's control loop with the same backbone model as CoCoEvolve. Matching the backbone removes model choice as an explanation for the observed gap between these search setups. We report the best objective reached within the shared budget.

Task instances and metrics. Table 17 records the instance size, the scalar reported in the main tables, and the Human/SOTA source. Higher is better except Cloudcast, which reports total broadcast cost.

Task	Instance	Reported metric	Human/SOTA
Circle Packing (Square)	N=26, unit square	sum of radii	2.63598
Circle Packing (Rect)	N=21, perimeter 4	sum of radii	2.36583
Heilbronn (Triangle)	N=11	min triangle area / host area	0.0365
Heilbronn (Convex)	N=13	min triangle area / hull area	0.0309
MinMax Distance	d=3, N=14	$(\min / \max)^2$	0.24004
Signal Processing	causal window filter	`overall_score` composite	0.718
Cloudcast (↓)	multi-cloud broadcast	total cost	626.2
EPLB	MoE expert placement	balancedness/speed mean	0.1265
Prism	GPU model placement	$1/\text{avg_kvpr}$ + success	21.89
LLM-SQL	column reorder	$0.95\cdot$hit-rate $+0.05\cdot$runtime	0.692
TXN	scheduling	combined throughput-style objective	2,725

Table 17: Math and ADRS evaluator settings. Math Human/SOTA is the better of the published human reference and the best previously published automated result, from AlphaEvolve (Novikov et al., 2025) and AdaEvolve (Cemri et al., 2025). Systems Human/SOTA values are the single AdaEvolve references.

J.2 Higher-precision math and ADRS tables

The main-text table rounds values for readability. Tables 18 and 19 report values at greater precision and retain rankings computed from the underlying values.

	Circle Packing (Square)		Circle Packing (Rect)		Heilbronn (Triangle)		Heilbronn (Convex)		MinMax Distance		Signal Processing	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human/SOTA	–	2.63598	–	2.36583	–	0.0365	–	0.0309	–	0.24004	–	0.718
Backbone: Sonnet-5												
OpenEvolve	2.63598 ± 0.00000	**2.63598**	2.36557 ± 0.00045	**2.36583**	0.03197 ± 0.00317	$\underline{0.0346}$	0.02775 ± 0.00216	0.02900	$\underline{0.23780 \pm 0.00309}$	$\underline{0.23991}$	0.6499 ± 0.0396	0.6727
GEPA	2.60218 ± 0.05856	**2.63598**	$\underline{2.36530 \pm 0.00091}$	**2.36583**	0.03588 ± 0.00080	**0.0365**	$\underline{0.03018 \pm 0.00132}$	**0.03094**	0.24005 ± 0.00000	**0.24005**	0.6031 ± 0.0481	0.6309
AdaEvolve	2.63329 ± 0.00258	**2.63598**	2.36423 ± 0.00250	**2.36583**	$\underline{0.03538 \pm 0.00196}$	**0.0365**	0.02913 ± 0.00148	$\underline{0.03076}$	0.24005 ± 0.00000	**0.24005**	$\underline{0.7470 \pm 0.0247}$	$\underline{0.7656}$
CoCoEvolve	$\underline{2.63542 \pm 0.00098}$	**2.63598**	2.36285 ± 0.00516	**2.36583**	0.03529 ± 0.00185	**0.0365**	0.03066 ± 0.00036	**0.03094**	0.24005 ± 0.00000	**0.24005**	0.7656 ± 0.0243	**0.7841**
Backbone: GPT-5.6-terra												
OpenEvolve	2.63598 ± 0.00000	**2.63598**	$\underline{2.36478 \pm 0.00091}$	$\underline{2.36583}$	0.03038 ± 0.00018	0.0305	0.02483 ± 0.00226	0.02709	0.23974 ± 0.00016	0.23983	0.5653 ± 0.0380	0.6075
GEPA	$\underline{2.63435 \pm 0.00283}$	**2.63598**	2.36407 ± 0.00186	$\underline{2.36583}$	0.03507 ± 0.00024	$\underline{0.03521}$	0.02607 ± 0.00301	**0.02781**	0.24005 ± 0.00000	**0.24005**	0.6327 ± 0.0491	0.6610
AdaEvolve	2.63315 ± 0.00200	$\underline{2.63431}$	2.36584 ± 0.00001	**2.36584**	$\underline{0.03457 \pm 0.00059}$	0.0352	0.02353 ± 0.00104	0.02413	$\underline{0.23982 \pm 0.00019}$	$\underline{0.24002}$	0.7921 ± 0.0197	**0.8133**
CoCoEvolve	2.63375 ± 0.00255	**2.63598**	2.36378 ± 0.00354	$\underline{2.36583}$	0.03408 ± 0.00214	**0.0365**	$\underline{0.02544 \pm 0.00205}$	$\underline{0.02712}$	0.24005 ± 0.00000	**0.24005**	$\underline{0.7563 \pm 0.0608}$	$\underline{0.80478}$

Table 18: Mathematical optimization benchmarks. We report **Mean** $\pm$ **Std** and **Best** objective values. Higher is better for all metrics. (N values: Square=26, Rect=21, Triangle=11, Convex=13; MinMax d=3, N=14). See Table 17. Human/SOTA is the better of the published Human reference and the best previously published automated result on each problem, drawn from AlphaEvolve Novikov et al. (2025) and AdaEvolve Cemri et al. (2025). Values at greater precision corresponding to the main-paper Table 4. Bold/underline follow the same within-backbone ranking on the underlying values.

Signal Processing's `overall_score` is a weighted blend of a composite $J(\theta)$ (slope changes, lag, error, false reversals), smoothness, correlation, noise reduction, and success rate.

Signal Processing strategy convergence. Surprisingly, AdaEvolve and CoCoEvolve independently converge to essentially the same champion strategy: both fit the complete signal with an L1 data term and an L1 second-difference penalty, expressed as the same sparse linear program, and return the aligned tail of the resulting piecewise-linear trend. Their material difference is only how the regularization weight is chosen. CoCoEvolve scales a fixed coefficient by a robust noise estimate, whereas AdaEvolve uses a clipped noise-to-motion rule. In the reported exchange experiment, swapping the regularization rules also swaps the measured performance. The two champions

	Cloudcast (↓)		EPLB		Prism		LLM-SQL		TXN	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human/SOTA	–	626.2	–	0.1265	–	21.89	–	0.692	–	2,725
Backbone: Sonnet-5										
OpenEvolve	660.693089 ± 0.007070	660.689007	0.1273 ± 0.0001	0.1273	26.236 ± 0.032	26.256	0.7132 ± 0.0115	0.7263	$3{,}945 \pm 123$	4,016
GEPA	621.090169 ± 5.213452	618.080181	0.1332 ± 0.0103	0.1451	26.256 ± 0.000	26.256	0.5235 ± 0.3733	**0.7522**	$3{,}874 \pm 218$	4,000
AdaEvolve	618.096950 ± 0.008222	618.087600	0.1446 ± 0.0006	0.1450	26.391 ± 0.233	**26.660**	0.7235 ± 0.0161	0.7328	$4{,}115 \pm 17$	4,132
CoCoEvolve	618.063457 ± 0.006616	**618.059637**	0.1454 ± 0.0002	**0.1455**	26.256 ± 0.000	26.256	0.7280 ± 0.0058	0.7316	$4{,}292 \pm 55$	**4,348**
Backbone: GPT-5.6-terra										
OpenEvolve	618.096205 ± 0.005931	618.092781	0.1272 ± 0.0000	0.1272	26.076 ± 0.136	26.152	0.7242 ± 0.0103	0.7301	$4{,}107 \pm 134$	4,184
GEPA	618.080973 ± 0.019122	**618.069933**	0.1272 ± 0.0000	0.1272	26.259 ± 0.005	26.264	0.7193 ± 0.0076	0.7237	$4{,}268 \pm 74$	**4,348**
AdaEvolve	618.085178 ± 0.013202	**618.069933**	0.1415 ± 0.0060	**0.1454**	25.674 ± 0.080	25.734	0.7168 ± 0.0065	0.7236	$4{,}144 \pm 20$	4,167
CoCoEvolve	618.081749 ± 0.010282	618.071097	0.14472 ± 0.00035	0.1451	26.338 ± 0.141	**26.501**	0.7283 ± 0.0020	**0.7302**	$4{,}250 \pm 81$	4,329

Table 19: ADRS systems benchmarks. We report **Mean** $\pm$ **Std** and **Best** objective values. Higher is better, except for **Cloudcast** where lower (↓) is better. Instance sizes and metrics are in Table 17. Human/SOTA values are from Cemri et al. (2025), which reports only a single reference value. Values at greater precision corresponding to the main-paper Table 4. Bold/underline follow the same within-backbone ranking on the underlying values.

therefore share the same broad filtering approach, and the regularization rule appears to explain their score difference in this comparison.

J.3 Per-task analysis

Across both backbones, CoCoEvolve is best or tied best in 13 of 24 math cells and 12 of 20 ADRS cells when Avg and Best are counted separately. It leads 25 of 44 cells overall, compared with 18 for GEPA, 13 for AdaEvolve, and 6 for OpenEvolve.

On Sonnet-5, CoCoEvolve has the highest Heilbronn Convex mean and ties GEPA on Best. In ADRS, it leads both Avg and Best on Cloudcast, EPLB, and TXN, and it leads LLM-SQL on Avg. AdaEvolve leads Prism, while GEPA leads LLM-SQL on Best.

On GPT-5.6-terra, the leaders are distributed across all four methods. CoCoEvolve leads Prism and LLM-SQL and has the highest Heilbronn Triangle Best. AdaEvolve leads Signal Processing and Circle Packing (Rect). GEPA leads Heilbronn Convex and TXN and has the best Cloudcast Avg. OpenEvolve has the best Circle Packing (Square) mean. Circle Packing at both instances and MinMax Distance produce several ties at or near the reference values, so their means rather than their Bests are the informative column.

The traces below examine two patterns in the table. Heilbronn Convex tests whether stalled trials recover near a known reference score; TXN shows a structural rewrite with room for a larger gain. Circle Packing provides a saturated case where final best scores are tied, and Cloudcast shows several methods finishing close together. These traces explain selected runs, not every aggregate result. InfoGain is inactive here: these tasks have no instance pool.

Figures. Heilbronn Convex layouts and best-so-far: Figures 4–5 (Appendix J.3.1). Circle Packing (Square) layouts and best-so-far: Figures 6–7 (Appendix J.3.2).

J.3.1 Heilbronn Convex (N=13)

Setup. The objective is the ratio of the smallest triangle area among N=13 points to the area of their convex hull; Human/SOTA is 0.0309. The seed returns 13 uniform random points (0.00032). All four methods run the matched 100-iteration Sonnet-5 protocol of Appendix J.1; CoCoEvolve's Avg in Table 4 is three retained trials. GEPA ties CoCoEvolve on Best, so the analysis below is not a unique-champion story: it is why Avg stays high when two of those three trials start badly. Figure 4 shows each method's Best-cell geometry. Figure 5 overlays one Best-cell trace per method and a second retained CoCoEvolve trial (dotted) that recovers from a stall; both CoCoEvolve traces enter the published Avg.

Search. The three retained CoCoEvolve finals are 0.03094, 0.03078, and 0.03025 (Table 4: 0.03066 ± 0.00036). Two of the three spend most of the budget near 0.028 and still finish at the ceiling, which is what keeps Avg above GEPA and the standard deviation a quarter of GEPA's.

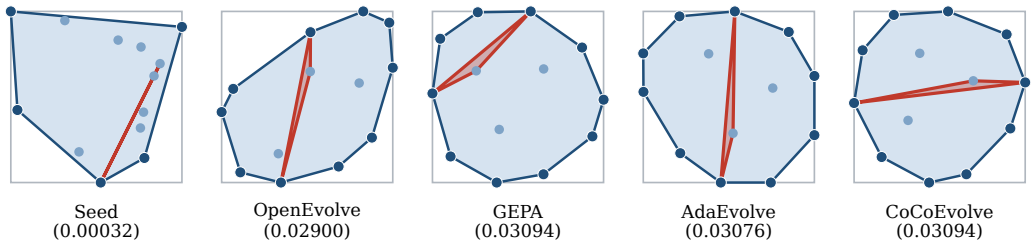

Figure 4: Heilbronn Convex (N=13) on Sonnet-5. Point sets in the unit square: hull vertices (navy), interior points (lighter), hull outlined, smallest triangle in red. Seed is unstructured. OpenEvolve reaches a weaker 13-gon (0.02900); GEPA, AdaEvolve, and CoCoEvolve all reach structured 13-gons at their Best-cell scores.

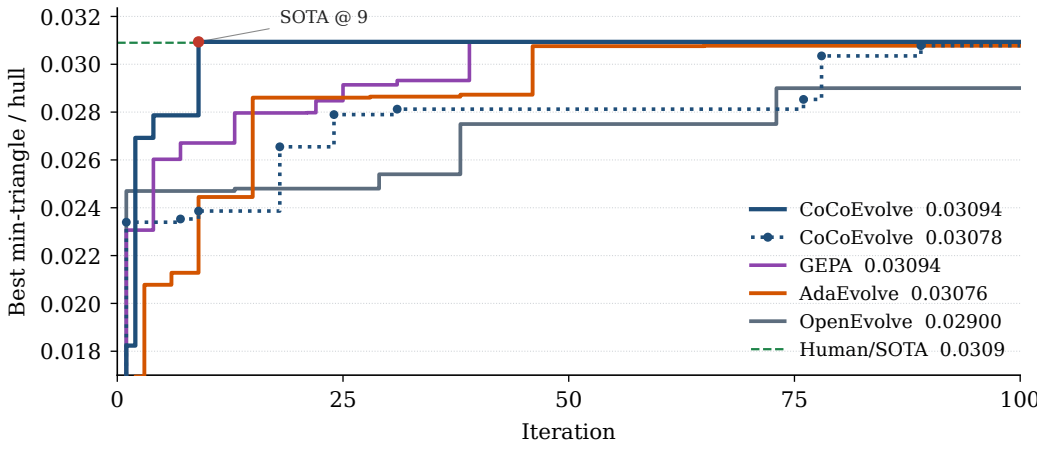

Figure 5: Heilbronn Convex best-so-far on Sonnet-5, cropped just below the first rewrite. Solid traces are each method's Best-cell run; Human/SOTA is the dashed green line. The dotted navy trace is a second retained CoCoEvolve trial: it stalls near 0.0281 and recovers to 0.03078. Solid CoCoEvolve reaches Human/SOTA at iteration 9. The claim is Avg, not the tied Best.

The solid trace is the trial that does not need a recovery. Iterations 1–2 replace the random seed with from-scratch solvers (0.0182, then 0.0269). The reflector then keeps a constrained-NLP family (`scipy.optimize.minimize`, SLSQP) and a hillclimb of that family lands Human/SOTA at iteration 9, with a scratch self-check before each commit. GEPA reaches the same 0.03094 at iteration 39. That is sample efficiency on one trial, not the claim for the column.

The dotted trial is the CMA-ES recovery and contributes to the three-run Avg. Its first rewrite scores 0.0234 and stalls near 0.0281 from iteration 31 while the mutator retries SciPy global solvers. At iteration 75, the reflector marks that family as exhausted and proposes a NumPy CMA-ES. A scratch prototype reaches 0.03035 at iteration 78. At iteration 89, the mutator returns to a dual-annealing program from iteration 19, which scored 0.02648, rather than using the current best CMA-ES child. It queries the ranked archive and adds CMA-ES with an exact-bilinear SLP polish, producing the 0.03078 champion.

The third retained trial (not plotted) is the same mechanism with a different family. It crawls from 0.0244 to 0.0275 over 73 iterations of scipy global solvers, then implements Tutte barycentric embedding (`scipy.sparse.linalg.spsolve`) at iteration 86 and jumps to 0.03025.

Either recovery is load-bearing for the Avg win. If the CMA-ES trial had remained at 0.0281 and the Tutte trial at its pre-jump 0.0275, the three-run mean would be 0.0289, below GEPA. If only one of the two recovered, the mean would still sit near 0.0297. Heilbronn therefore shows how historical synthesis improves reliability near a ceiling. The ledger triggers a new family after a stall, and the

archive query transfers that family to a viable island. This mechanism explains the high Avg rather than a uniquely high Best.

J.3.2 CIRCLE PACKING (SQUARE)

Setup. The objective is the sum of radii of $N{=}26$ non-overlapping circles in the unit square; Human/SOTA is 2.63598. The seed is a two-ring heuristic (0.96). This instance saturates: every method in Table 4 reaches Human/SOTA in at least one Sonnet-5 trial, so Best cannot rank them. The traces below are therefore a budget-use comparison, not a final-cell comparison. Figure 6 shows one ceiling packing per method. Figure 7 overlays the Best-cell best-so-far of each method, cropped near the ceiling, under the same 100-iteration window.

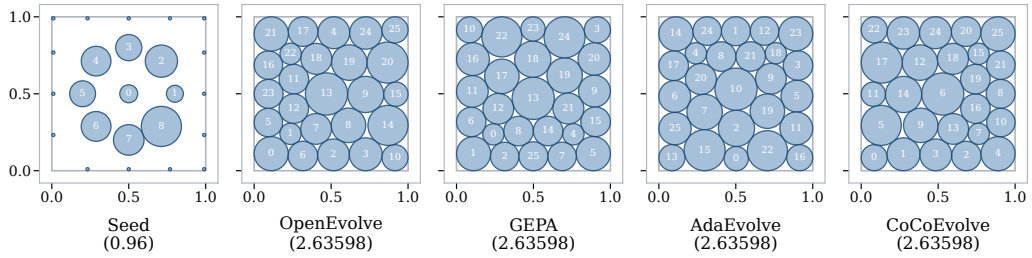

Figure 6: Circle Packing (Square, $N{=}26$) on Sonnet-5. Circles in the unit square, labeled 0–25. The seed is a two-ring heuristic (0.96); OpenEvolve, GEPA, AdaEvolve, and CoCoEvolve are size-biased ceiling packings at Human/SOTA (2.63598).

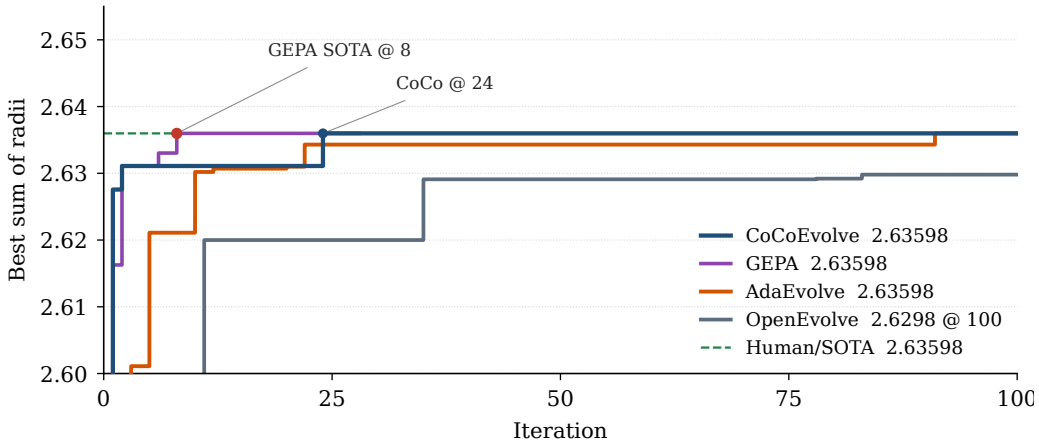

Figure 7: Circle Packing (Square, $N{=}26$) best-so-far on Sonnet-5, cropped near the ceiling. Solid traces are each method's Best-cell run; Human/SOTA is the dashed green line. GEPA first reaches 2.63598 at iteration 8, CoCoEvolve at iteration 24, AdaEvolve at iteration 91. OpenEvolve is at 2.6298 at iteration 100 and only hits the ceiling at iteration 111, outside the shared window.

Search. This is a saturation control, not a CoCoEvolve win. Once every method can hit the ceiling, the remaining signal is how the shared budget is spent getting there. On the traces in Figure 7, GEPA is first (iteration 8), CoCoEvolve is second (iteration 24), AdaEvolve reaches the ceiling at iteration 91, and OpenEvolve is still at 2.6298 at iteration 100 (its Best of 2.63598 is from iteration 111, outside the shared window). We do not read first-to-ceiling as a method ranking: GEPA winning that race on a saturated packing instance is compatible with Heilbronn, where the same protocol separates the methods on Avg rather than Best.

J.3.3 CLOUDCAST

Setup. The objective is the simulator's true union-of-edges cost of a multi-cloud broadcast tree (lower is better); Human/SOTA is 626.2. The seed enumerates Azure single- and two-hub plans ranked by a node-count/bottleneck proxy, with an arborescence fallback, and scores 974. The analysis below is the CoCoEvolve Best-cell Sonnet-5 run under the same 100-iteration protocol. OpenEvolve remains far from the others; GEPA, AdaEvolve, and CoCoEvolve all finish in a tight 618 cluster (Table 4), so this subsection is about the construction that reached that cluster, not about ranking inside it.

Search. The champion no longer searches only for hubs. It evaluates six structurally distinct tree families: terminal MSA, relay hub, capacitated min-cost flow, Dreyfus–Wagner DP, resistance ranking, and TSP relay. A proxy replays the simulator's cost model and selects the cheapest candidate.

The first two iterations reduce cost from 974 to 636 by reading the cost graph and instance configuration and by testing scratch candidates against the live evaluator. Later cross-lineage grafts add Dreyfus–Wagner ranking and a TSP-relay construction, which produce the remaining improvement. The trace documents a substantial rewrite of the seed's search procedure. Because GEPA, AdaEvolve, and CoCoEvolve finish close together, Cloudcast gives little evidence for ranking their final scores. TXN provides a clearer example of a large score change after a structural rewrite.

J.3.4 TXN SCHEDULING

Setup. The objective is a combined throughput-style score of a lock-aware transaction schedule (higher is better); Human/SOTA is 2,725. The seed is tabu search over swap and or-opt-1 from Copeland / MDS / assignment-problem / greedy starts, and scores 2,725. The reported runs show substantial headroom above that seed and reference value. Both the mean and best scores can therefore distinguish methods here. The analysis below is the CoCoEvolve Best-cell Sonnet-5 run that produced 4,348, under the same 100-iteration protocol.

Search. The champion replaces the seed's tabu loop with a three-stage pipeline. An assignment-problem relaxation and a single-path greedy construction from the top Copeland candidate each build a starting schedule, and the cheaper one is kept; simulated annealing then applies double-bridge reheating, and first-improvement local search performs the final cleanup.

Copeland ranking and an EDA over pairwise order probabilities first move the score above the seed. The EDA also queries the ranked archive instead of editing only the parent. The final program uses assignment/greedy construction followed by simulated annealing, not a retuned tabu tenure.

TXN is therefore the far-from-ceiling counterpart to Heilbronn. In Heilbronn, the ledger improves reliability by rescuing stalled trials. In TXN, the same keep-or-switch loop replaces the seed metaheuristic and improves both Avg and Best beyond the other methods.

K PROMPT OPTIMIZATION

K.1 DETAILED METHODOLOGY

We evaluate three tasks from GEPA's benchmark suite. All three compare an unmodified seed, GEPA, and CoCoEvolve from the same initial program; each CoCoEvolve run uses 100 iterations.

HotpotQA. The artifact is a four-stage retrieve-summarize-answer pipeline over a BM25 index of Wikipedia abstracts, with prompts for first-hop summarization (`summarize1`), second-hop query generation (`create_query_hop2`), second-hop summarization (`summarize2`), and final answering (`final_answer`). Each mutation changes one prompt while retrieval remains fixed, and GEPA's round-robin selector mutates the same four components in the same cycle. We use GEPA's official 150/300/300 train, validation, and held-out-test split. The GEPA search budgets were chosen to be broadly comparable to the 100-iteration CoCoEvolve runs, although the two methods account for evaluations differently. The frozen Sonnet-5 backbone serves as both mutator and student.

GEPA uses the same `Benchmark` base class for AIME, HotpotQA, and IFBench. The artifact runner passes individual query instances in separate training and validation sets for all three tasks. GEPA samples three training examples by default for each reflective step. A proposed edit must strictly improve its parent's score on that minibatch. Each accepted candidate then triggers a full validation-set evaluation, whose per-example scores inform the Pareto selector's choice of future parents.

AIME. The evolved artifact is a single chain-of-thought system prompt that a frozen student model reads with each competition math problem to produce an integer answer. GEPA's fixed-seed split draws 45 training and 45 validation problems from `AI-MO/aimo-validation-aime`; its disjoint test set contains 30 problems from `MathArena/aime_2025`. AIME uses `openai-gpt-5.6-luna` as both mutator and student. An initial Sonnet-5 run gave the seed a perfect training score, leaving no signal for optimization.

IFBench. The artifact has two prompt-driven stages, `generate_response` and `ensure_correct_response`. The evaluator scores the fraction of each query's instruction constraints that the response satisfies. It supports 83 checker types: 25 from IFEval and 58 IFBench-native types, including word and letter counts, formatting, and keyword use. Eight leniency variants account for markdown emphasis and optional leading or trailing lines. Neither stage sees the constraint list. GEPA's fixed split uses the first 300 rows of its training file for validation and samples 150 training rows from the next 300 with a fixed seed. We use its bundled 294-row test file unchanged; it has six fewer rows than the public IFBench release. IFBench also uses `openai-gpt-5.6-luna` as both mutator and student.

The IFBench run uses the default `FullEvaluationPolicy` and 100 concurrent workers.

Repeated evaluation. With `openai-gpt-5.6-luna`, we evaluated the same evolved AIME prompt twice on the 45 training problems: once during optimization and once afterward. The second evaluation got five fewer answers correct, even though the prompt had not changed. The student's nonzero decoding temperature can make its answers vary between evaluations. Each optimizer ran once and produced one final candidate. We evaluated that unchanged candidate three times on each AIME and IFBench split and report the mean and standard deviation in Table 5. These repeats measure evaluation variability, not search variability. On AIME test, five baseline questions, eight GEPA questions, and six CoCoEvolve questions changed correctness across repeats, even though the baseline's aggregate test score was $25/30$ every time.

K.2 An efficiency variant: sim-graph targeting and cascaded evaluation on HotpotQA

The HotpotQA run of Appendix K.1 scores its *entire* 150-question training pool and 300-question validation pool on every one of its 100 iterations, unconditionally – 450 real model-backed questions per iteration regardless of how informative any single one is. DABStep's own evaluator (Appendix D.2, Appendix D.3) instead scores only a bounded, informativeness-targeted subset per iteration and extrapolates the rest from the parent's own history. We test a HotpotQA variant that combines similarity-based target selection, P0/P1/P2 cascaded evaluation, and mutator self-testing. The comparison measures this combined configuration. It cannot attribute any score or time difference to one of the three changes alone.

Setup. We keep the seed, 100-iteration budget, and backbone from Appendix K.1. We add three mechanisms.

Target selection. A similarity graph and InfoGain selector operate over the 150-question training pool (Appendix D.3). Each mutation receives one selected focus question instead of the full undifferentiated pool.

Cascaded evaluation. The P0/P1/P2 cascade mirrors the DABStep evaluator (Appendix G.1). P0 evaluates only the selected target and requires a score of 0.6. If P0 passes, P1 evaluates a stratified sample of 20 regression questions, or about 13% of the pool. One third of this sample is selected by similarity-weighted risk to the target, and two thirds are sampled uniformly from the parent's tested history. P1 requires at least $0.9\times$ the parent's score on the same sample. If P1 passes, P2 explores

similarity-ranked frontier tiers of 10 questions each. One random scout represents each tier, and the next tier opens only if the current tier scores at least $6/10$. The training cascade evaluates at most $\lfloor 150/3 \rfloor = 50$ questions per iteration, excluding the one-time seed bootstrap, mutator self-tests, and separately sampled validation passes. When the parent is already at ceiling on the target, a live pretest can bypass P0 and P1 and proceed directly to P2.

The child's score combines its updated per-question record with the parent's inherited record and divides the total by 150. We do not use DABStep's additive formula because that formula assumes sparse initial coverage. HotpotQA has dense coverage from the seed's full-pool bootstrap, so the additive formula would double-count observations.

Mutator self-test. The mutator receives `test_draft`, with a budget of three calls per turn. It can score an in-progress edit directly on the selected focus question. The mutator prompt also instructs the model to verify its draft before finishing. Both features reproduce DABStep's self-verification affordance (Appendix D.2).

Validation is sampled independently with probability 0.1 per iteration. When validation is skipped, the child inherits the parent's validation score and per-question record unchanged.

Evaluator	Train	Val	Test
Full pool, every iteration (Appendix K.1)	0.800	0.783	0.793
Sim-graph target + P0/P1/P2 cascade	0.800	0.790	0.767

Table 20: HotpotQA, Sonnet-5, CoCoEvolve: the published full-pool-every-iteration search versus the sim-graph-targeted/cascaded-evaluation variant of this appendix. Both rows are a single 100-iteration search, freshly re-scored against the identical train/val/test pools.

Results. Across 100 iterations, the cascaded evaluator has a median wall-clock time of 121s per iteration. A full 450-question pass at the same concurrency takes roughly 900–1100s. The seed's one-time full-pool bootstrap takes 1115s, which is consistent with that estimate.

In the first 50 logged iterations, seven (14%) took 400–1100s; one was the seed bootstrap, and the others included a sampled pass over all 300 validation questions. Extrapolating that mix to 100 iterations gives an estimated 5.5–6 hours of evaluator time, versus an estimated 25–30 hours for full-pool evaluation at every iteration. The implied reduction is about 4–$5\times$. These are estimates from the observed timing mix, not measured total run times.

Analysis. The combined variant matches the full-pool run's observed train accuracy (0.800) and has 0.007 higher validation accuracy, but its test accuracy is 0.026 lower, about eight questions out of 300. The timing estimate suggests lower evaluator cost. With one search and one final re-score per variant, the result does not establish equal held-out quality or isolate which added mechanism produced the difference. It also does not replace or improve the main HotpotQA result in Table 5.

L COMPLETE BASE AND FINAL PROGRAMS

The following unabridged listings pair the seed and final program from one run each of Heilbronn Convex, Cloudcast, and TXN. The Cloudcast and TXN listings contain only the evolved function; module-level imports (e.g., `networkx as nx`, `random`) and harness classes such as `BroadCastTopology` are supplied by the surrounding benchmark file.

L.1 HEILBRONN CONVEX

Heilbronn Convex: base program

```python
import numpy as np


def heilbronn_convex13() -> np.ndarray:
    """
    Construct an arrangement of n points on or inside a convex region in order to maximize the area of the
    smallest triangle formed by these points. Here n = 13.

    Returns:
        points: np.ndarray of shape (13,2) with the x,y coordinates of the points.
    """
    n = 13
    rng = np.random.default_rng(seed=42)
    points = rng.random((n, 2))
    return points
```

Heilbronn Convex: final evolved program

```
"""
Heilbronn convex-13 placement via a from-scratch, pure-numpy CMA-ES
(Covariance Matrix Adaptation Evolution Strategy) driven directly on the
raw, non-smoothed maximin ratio (min triangle area / hull area) over all
26 raw coordinates, followed by an exact-bilinear trust-region Sequential
Linear Programming (SLP) local polish and a deterministic iterated local
search (ILS) refinement pass.

This replaces the previous strategy in this lineage (a single
`scipy.optimize.dual_annealing` call on the raw ratio, warm-started from a
regular 13-gon with a Powell local-search step) with a mechanistically
different two-stage pipeline:

  Stage A (global, derivative-free): many independent from-scratch CMA-ES
  restarts, each maintaining a persistent, adaptively-controlled global step
  size (cumulative step-size adaptation) plus a covariance matrix blended
  from a rank-one "remembered direction" term and a rank-mu "current elite
  scatter" term. This is far more resistant to premature step-size collapse
  on this rugged, non-smooth landscape than a single dual_annealing run's
  fixed cooling schedule, because sigma can re-expand if progress stalls
  instead of only ever shrinking.

  Stage B (local, exact-gradient): every one of the C(13,3)=286 triangle
  areas is an EXACT bilinear (shoelace) function of the 26 coordinates, so
  a trust-region Sequential Linear Programming (SLP) procedure can linearize
  every triangle's signed area around the current point, solve one small
  epigraph LP (`scipy.optimize.linprog`, method="highs") for the best
  bounded displacement, and accept/reject/resize the trust region via the
  classical predicted-vs-actual ratio test against the TRUE (non-linearized)
  ratio. This converges far closer to an exact local optimum of the maximin
  problem than any further stochastic CMA-ES generations would, and is
  distinct from the previous line's derivative-free Powell polish (which
  never exploits the exact bilinear structure of the triangle areas).

  Stage C: a deterministic iterated local search (fixed, shrinking-noise-
  scale perturbation kicks re-polished by the SLP procedure) around the best
  SLP incumbent, to escape any shallow local optimum the trust-region
  procedure settles into on its own.

Scaling vs. the smaller/faster variant of this same architecture previously
tried in this lineage (14 restarts x 500 generations, ~25s total): this
version raises the restart count, per-restart generation budget, SLP outer-
iteration cap, and ILS iteration count substantially (to ~80-150s total,
timed empirically via scratch runs of the identical CMA-ES/SLP code), since
the evaluator's timeout (360s) leaves very large unused headroom relative to
both the previous ~25s run and the ~70s single dual_annealing call this
replaces. More restarts/generations of CMA-ES is the primary lever for
escaping local optima on this rugged landscape, so that budget increase is
prioritized over any other single change. Restart warm starts are also
diversified: half start from a jittered regular 13-gon (as before), the
other half from a jittered two-ring layout (an outer ring of points plus an
```

```
inner ring, rotated by a half-step), giving CMA-ES's covariance adaptation a
second, structurally distinct family of basins to explore per restart batch.

Everything is fully deterministic (numpy `Generator` objects seeded with
fixed integers throughout, no unseeded randomness anywhere), so results are
exactly reproducible across runs and platforms.
"""
import numpy as np
from itertools import combinations
from scipy.spatial import ConvexHull
from scipy.optimize import linprog

_N = 13
_COMBOS = np.array(list(combinations(range(_N), 3)))  # all C(13,3)=286 index triples
_TI, _TJ, _TK = _COMBOS[:, 0], _COMBOS[:, 1], _COMBOS[:, 2]
_NTRI = len(_TI)
_DIM = 2 * _N


def _hull_area(points: np.ndarray) -> float:
    """Convex-hull area of a single (N,2) point set, or -1.0 if degenerate."""
    try:
        ha = ConvexHull(points).volume
    except Exception:
        return -1.0
    return ha if ha > 1e-12 else -1.0


def _ratio(points: np.ndarray) -> float:
    """True (non-smooth) objective to maximize: (smallest triangle area) / (convex hull area)."""
    p1, p2, p3 = points[_TI], points[_TJ], points[_TK]
    cross = (p2[:, 0] - p1[:, 0]) * (p3[:, 1] - p1[:, 1]) - (p3[:, 0] - p1[:, 0]) * (p2[:, 1] -
      p1[:, 1])
    areas = 0.5 * np.abs(cross)
    ha = _hull_area(points)
    if ha <= 0.0:
        return -1.0
    return float(areas.min() / ha)


def _batch_scores(X: np.ndarray) -> np.ndarray:
    """
    Vectorized raw-score evaluation of a whole population at once: every candidate's
    286 triangle areas are computed in one single broadcasted numpy expression across
    the whole (popsize, 26) batch. The convex-hull area still needs one `ConvexHull`
    call per candidate (qhull has no batched entry point), but that per-candidate loop
    does none of the O(popsize * 286) triangle-area work, which is the part that
    actually scales.
    """
    popsize = X.shape[0]
    pts = X.reshape(popsize, _N, 2)
    p1, p2, p3 = pts[:, _TI, :], pts[:, _TJ, :], pts[:, _TK, :]
    cross = (p2[:, :, 0] - p1[:, :, 0]) * (p3[:, :, 1] - p1[:, :, 1]) - (
        p3[:, :, 0] - p1[:, :, 0]
    ) * (p2[:, :, 1] - p1[:, :, 1])
    min_areas = 0.5 * np.abs(cross).min(axis=1)

    scores = np.full(popsize, -1.0)
    for i in range(popsize):
        ha = _hull_area(pts[i])
        if ha > 0.0:
            scores[i] = min_areas[i] / ha
    return scores


def _regular_13gon() -> np.ndarray:
    """Defensive fallback: a plain regular 13-gon, always convex and non-degenerate."""
    angles = np.linspace(0, 2 * np.pi, _N, endpoint=False)
    return np.stack([np.cos(angles), np.sin(angles)], axis=1)


def _jittered_13gon(rng: np.random.Generator, jitter: float = 0.05) -> np.ndarray:
    """Regular 13-gon perturbed by small iid Gaussian jitter -- one CMA-ES restart init family."""
    base = _regular_13gon()
    return (base + rng.normal(scale=jitter, size=base.shape)).reshape(-1)


def _jittered_two_ring(rng: np.random.Generator, m_outer: int = 8, jitter: float = 0.05) ->
  np.ndarray:
    """
    Second, structurally distinct restart-init family: an outer ring of `m_outer`
    points plus an inner ring of the remaining points (rotated by a half-step so no
    point is radially aligned with an outer one), each perturbed by small iid
    Gaussian jitter. Gives CMA-ES's covariance adaptation a different starting basin
    family than the single-ring regular 13-gon used by `_jittered_13gon`.
    """
    n_inner = _N - m_outer
    angles_o = np.linspace(0, 2 * np.pi, m_outer, endpoint=False)
    outer = np.stack([np.cos(angles_o), np.sin(angles_o)], axis=1)
    apothem = np.cos(np.pi / m_outer)
    r_inner = 0.45 * apothem
```

```
    angles_i = np.linspace(0, 2 * np.pi, n_inner, endpoint=False) + (np.pi / m_outer)
    inner = np.stack([r_inner * np.cos(angles_i), r_inner * np.sin(angles_i)], axis=1)
    base = np.concatenate([outer, inner], axis=0)
    return (base + rng.normal(scale=jitter, size=base.shape)).reshape(-1)


def _cma_es(mean0: np.ndarray, sigma0: float, popsize: int, n_gen: int, seed: int):
    """
    A complete, from-scratch, pure-numpy (mu/mu_W, lambda)-CMA-ES loop (Hansen &
    Ostermeier), maximizing `_batch_scores` over the flat 26-D coordinate vector.

    Each generation:
      1. Sample `popsize` candidates x = mean + sigma * B @ (D * z), z ~ N(0, I)
         (equivalently, x ~ N(mean, sigma^2 * C) with C = B @ diag(D^2) @ B.T),
         evaluated in one vectorized batch via `_batch_scores`.
      2. Rank-select the top `mu = popsize // 2` candidates by raw score and take
         their log-rank-weighted mean as the new distribution mean.
      3. Update the step-size evolution path p_sigma (cumulative step-size
         adaptation, CSA) and rescale sigma so that p_sigma's length tracks its
         expected length under random selection -- sigma grows when steps are
         longer/more aligned than expected (fast progress) and shrinks when they
         are shorter (fine-tuning / stalling).
      4. Update the covariance evolution path p_c, then blend the covariance
         matrix with a rank-one term (outer(p_c, p_c): remembers the *direction*
         of sustained recent progress) and a rank-mu term (the weighted outer
         product of the current elites' steps: captures the current generation's
         scatter).
      5. Periodically re-eigendecompose C into (B, D) for the next generation's
         sampling and the C^{-1/2} needed by step 3.
    A light-touch rescaling keeps the search numerically well-conditioned: since
    `_ratio` is exactly scale-invariant, whenever the current mean's own hull area
    drifts far from 1 we rescale mean and sigma together by the same factor (a
    pure similarity transform, so it changes nothing about the objective, only
    keeps coordinate magnitudes in a numerically safe range across hundreds of
    generations of unconstrained drift).

    Returns (best_x, best_score): the single best-scoring 26-D vector and its raw
    score ever observed across every generation of this run (not just the final
    one, since CMA-ES's stochastic sampling can visit its best point generations
    before the distribution itself fully commits there).
    """
    rng = np.random.default_rng(seed)
    dim = _DIM
    mu = popsize // 2

    # Log-rank elite weights and the standard derived CMA-ES strategy parameters.
    w_raw = np.log(mu + 0.5) - np.log(np.arange(1, mu + 1))
    weights = w_raw / w_raw.sum()
    mu_eff = 1.0 / np.sum(weights**2)

    c_sigma = (mu_eff + 2) / (dim + mu_eff + 5)
    d_sigma = 1 + 2 * max(0.0, np.sqrt((mu_eff - 1) / (dim + 1)) - 1) + c_sigma
    cc = (4 + mu_eff / dim) / (dim + 4 + 2 * mu_eff / dim)
    c1 = 2 / ((dim + 1.3) ** 2 + mu_eff)
    cmu = min(1 - c1, 2 * (mu_eff - 2 + 1 / mu_eff) / ((dim + 2) ** 2 + mu_eff))
    chiN = np.sqrt(dim) * (1 - 1 / (4 * dim) + 1 / (21 * dim**2))

    mean = mean0.astype(float).copy()
    sigma = float(sigma0)
    C = np.eye(dim)
    p_sigma = np.zeros(dim)
    p_c = np.zeros(dim)
    B = np.eye(dim)
    D = np.ones(dim)
    eig_gen = 0

    best_score = -1e18
    best_x = mean.copy()

    for gen in range(n_gen):
        Z = rng.standard_normal((popsize, dim))
        X = mean + sigma * (Z @ (B * D).T)

        scores = _batch_scores(X)
        order = np.argsort(-scores)

        if scores[order[0]] > best_score:
            best_score = float(scores[order[0]])
            best_x = X[order[0]].copy()

        elite_X = X[order[:mu]]
        mean_old = mean
        mean = weights @ elite_X

        # --- cumulative step-size adaptation (CSA) ---
        y = (mean - mean_old) / sigma
        c_inv_sqrt_y = B @ ((B.T @ y) / D)
        p_sigma = (1 - c_sigma) * p_sigma + np.sqrt(c_sigma * (2 - c_sigma) * mu_eff) * c_inv_sqrt_y
        sigma *= np.exp((c_sigma / d_sigma) * (np.linalg.norm(p_sigma) / chiN - 1))

        # --- covariance evolution path + rank-one/rank-mu covariance update ---
```

```
        hs = 1.0 if (
            np.linalg.norm(p_sigma) / np.sqrt(1 - (1 - c_sigma) ** (2 * (gen + 1))) / chiN
        ) < (1.4 + 2 / (dim + 1)) else 0.0
        p_c = (1 - cc) * p_c + hs * np.sqrt(cc * (2 - cc) * mu_eff) * y

        elite_steps = (elite_X - mean_old) / sigma
        rank_mu_term = (elite_steps.T * weights) @ elite_steps
        C = (
            (1 - c1 - cmu) * C
            + c1 * (np.outer(p_c, p_c) + (1 - hs) * cc * (2 - cc) * C)
            + cmu * rank_mu_term
        )

        # Numerical safety net: `_ratio` is exactly scale-invariant, so rescaling
        # (mean, sigma) together by the same factor changes nothing about the
        # objective but keeps coordinate magnitudes from drifting to extremes
        # over hundreds of unconstrained generations.
        mean_ha = _hull_area(mean.reshape(_N, 2))
        if mean_ha > 0.0 and (mean_ha < 0.1 or mean_ha > 10.0):
            rescale = 1.0 / np.sqrt(mean_ha)
            mean = mean * rescale
            sigma *= rescale

        # Periodic eigendecomposition refresh of C -> (B, D), amortized over
        # roughly dim/(c1+cmu)/10 generations as in standard CMA-ES practice.
        if gen - eig_gen > max(1, int(popsize / (c1 + cmu) / dim / 10)):
            eig_gen = gen
            C = np.triu(C) + np.triu(C, 1).T  # enforce exact symmetry
            eigvals, eigvecs = np.linalg.eigh(C)
            eigvals = np.clip(eigvals, 1e-20, None)
            D = np.sqrt(eigvals)
            B = eigvecs

    return best_x, best_score


# ---------------------------------------------------------------------------
# Trust-region Sequential Linear Programming (SLP) local polish.
#
# CMA-ES's covariance-adapted Gaussian sampling is an excellent global/basin
# search, but being stochastic and derivative-free it never converges exactly
# onto a KKT point of the maximin-ratio problem -- its distribution keeps
# jittering around the optimum rather than landing on it. Every one of the
# C(13,3)=286 triangle areas is, however, an EXACT bilinear (shoelace) function
# of the 26 free coordinates: Area_ijk(x) = 0.5*(xi*(yj-yk)+xj*(yk-yi)+xk*(yi-yj)).
# That lets us do far sharper local polishing than more CMA-ES generations
# could: fix every triangle's current orientation sign s_ijk = sign(Area_ijk(x))
# and solve the exact (to first order) trust-region epigraph LP
#     maximize t   s.t.  s_ijk*(Area_ijk(x) + grad(Area_ijk)(x)^T d) >= t
#                        ||d||_inf <= r
# once per outer iteration via a single scipy.optimize.linprog(method="highs")
# call, then apply the classical trust-region accept/reject + grow/shrink test
# comparing the LP's *predicted* improvement against the *actual* improvement
# of the true (non-linearized) ratio.
# ---------------------------------------------------------------------------


def _signed_areas_and_grad(x: np.ndarray):
    """Exact signed area and gradient (w.r.t. the flat 26-D coordinate vector) of
    every one of the 286 triangles, via the shoelace formula. Each triangle's area
    is a bilinear (degree-2) function of its three points' six coordinates, so this
    analytic gradient is exact -- the first-order Taylor expansion used by the SLP
    step below is exact up to a purely quadratic remainder in the displacement."""
    pts = x.reshape(_N, 2)
    xi, yi = pts[_TI, 0], pts[_TI, 1]
    xj, yj = pts[_TJ, 0], pts[_TJ, 1]
    xk, yk = pts[_TK, 0], pts[_TK, 1]
    f = 0.5 * (xi * (yj - yk) + xj * (yk - yi) + xk * (yi - yj))
    grad = np.zeros((_NTRI, _DIM))
    rows = np.arange(_NTRI)
    grad[rows, 2 * _TI] = 0.5 * (yj - yk)
    grad[rows, 2 * _TI + 1] = 0.5 * (xk - xj)
    grad[rows, 2 * _TJ] = 0.5 * (yk - yi)
    grad[rows, 2 * _TJ + 1] = 0.5 * (xi - xk)
    grad[rows, 2 * _TK] = 0.5 * (yi - yj)
    grad[rows, 2 * _TK + 1] = 0.5 * (xj - xi)
    return f, grad


def _slp_normalize(x: np.ndarray) -> np.ndarray:
    """Recenter at the centroid and rescale so hull area is exactly 1 -- a pure
    similarity transform that leaves the scale-invariant `_ratio` untouched but
    keeps the LP's coefficients well-conditioned across many outer iterations."""
    pts = x.reshape(_N, 2)
    ha = _hull_area(pts)
    if ha <= 1e-12:
        ha = 1.0
    pts = pts - pts.mean(axis=0)
    pts = pts * np.sqrt(1.0 / ha)
    return pts.reshape(-1)
```

```

def _slp_lp_step(x: np.ndarray, r: float):
    """One trust-region LP subproblem: linearize every triangle's signed area
    around x and solve for the best epigraph displacement d with ||d||_inf <= r."""
    f, grad = _signed_areas_and_grad(x)
    s = np.sign(f)
    s[s == 0.0] = 1.0
    sf = s * f
    sgrad = grad * s[:, None]
    A_ub = np.hstack([-sgrad, np.ones((_NTRI, 1))])
    b_ub = sf
    c = np.zeros(_DIM + 1)
    c[-1] = -1.0
    bounds = [(-r, r)] * _DIM + [(None, None)]
    try:
        res = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method="highs")
    except Exception:
        return None, None
    if not res.success:
        return None, None
    return res.x[:_DIM], res.x[-1]


def _run_slp(x0: np.ndarray, max_outer: int = 200, r0: float = 0.2,
             r_min: float = 1e-10, r_max: float = 0.5):
    """Trust-region SLP / convex-concave procedure: repeatedly linearize, solve the
    LP, and accept/reject/resize via the classical trust-region ratio test against
    the true (non-linearized) ratio. Returns (best_x, best_ratio)."""
    x = _slp_normalize(x0.copy())
    cur_ratio = _ratio(x.reshape(_N, 2))
    r = r0
    for _ in range(max_outer):
        if r < r_min:
            break
        d, t_pred = _slp_lp_step(x, r)
        if d is None:
            r *= 0.5
            continue
        x_new = x + d
        new_ratio = _ratio(x_new.reshape(_N, 2))
        predicted_improve = t_pred - cur_ratio
        if predicted_improve <= 1e-14:
            r *= 0.5
            continue
        rho = (new_ratio - cur_ratio) / predicted_improve
        if rho > 0.1:
            x = _slp_normalize(x_new)
            cur_ratio = new_ratio
            if rho > 0.75:
                r = min(r * 1.5, r_max)
        else:
            r *= 0.5
    return x, cur_ratio


def heilbronn_convex13() -> np.ndarray:
    """
    Construct an arrangement of n=13 points on or inside a convex region to maximize the
    area of the smallest triangle formed by any three of them, normalized by the area of
    their own convex hull.

    Approach (multi-restart, from-scratch pure-numpy CMA-ES on the raw ratio, followed by
    an exact-bilinear trust-region SLP local-polish phase and a deterministic ILS pass):
      1. Run `_N_RESTARTS` independent CMA-ES searches (see `_cma_es`), alternating between
         two structurally distinct deterministic warm-start families -- a jittered regular
         13-gon (`_jittered_13gon`) and a jittered two-ring layout (`_jittered_two_ring`) --
         each with sigma0 ~ 0.3 and a distinct deterministic RNG seed, for `_N_GEN`
         generations of a `_POPSIZE`-sized population each.
      2. Track every restart's own best raw-scored (min triangle area / hull area)
         configuration, not just the single overall-best across all restarts, so several
         structurally different basins survive into the next phase.
      3. Spend one further CMA-ES pass, re-seeded at the overall-best point with a much
         smaller sigma0, to locally polish it.
      4. Polish the overall best, the CMA-ES refine result, and several of the next-best
         distinct restart endpoints with `_run_slp` -- a trust-region Sequential Linear
         Programming / convex-concave procedure that exploits the fact that every one of
         the 286 triangle areas is an EXACT bilinear function of the 26 coordinates, so
         each outer iteration solves one small LP to first-order optimality instead of
         relying on further stochastic sampling.
      5. Run a deterministic iterated local search (small perturbation kick + `_run_slp`
         re-polish, fixed shrinking noise schedule) around the best SLP-polished
         incumbent, keeping it only if the true ratio improves.
      6. Re-score every candidate (all restarts' bests, the CMA-ES polish, every SLP
         polish, the ILS result, and the safe regular-13-gon fallback) with the exact,
         non-smoothed `_ratio`, and return whichever scores highest -- so this pipeline
         can only match or improve on its own safe fallback, and each new phase can only
         match or improve on the phases before it, never regress.
    Everything is fully deterministic (numpy `Generator` objects seeded with fixed
    integers throughout, no unseeded randomness anywhere), so results are exactly
    reproducible across runs and platforms.

```

```
    Compute budget: sized (via empirical scratch-run timing of this exact CMA-ES/SLP
    code) to use a large fraction of the evaluator's headroom below its 360s timeout
    while keeping a comfortable safety margin, since more CMA-ES restarts/generations
    is the primary lever for escaping local optima on this rugged, non-smooth landscape.

    Returns:
        points: np.ndarray of shape (13,2) with the x,y coordinates of the points.
    """
    _N_RESTARTS = 30
    _POPSIZE = 64
    _N_GEN = 800
    _REFINE_GEN = 600

    fallback = _regular_13gon()
    candidates = [(_ratio(fallback), fallback)]

    try:
        restart_results = []
        for restart in range(_N_RESTARTS):
            init_rng = np.random.default_rng(1000 + restart)
            if restart % 2 == 0:
                mean0 = _jittered_13gon(init_rng, jitter=0.05)
            else:
                mean0 = _jittered_two_ring(init_rng, m_outer=8, jitter=0.05)
            x, score = _cma_es(mean0, sigma0=0.3, popsize=_POPSIZE, n_gen=_N_GEN, seed=restart)
            restart_results.append((score, x))

        if restart_results:
            restart_results.sort(key=lambda c: -c[0])
            overall_best_score, overall_best_x = restart_results[0]

            pts = overall_best_x.reshape(_N, 2)
            candidates.append((_ratio(pts), pts))

            # Local polish: re-run CMA-ES seeded at the best point found so far,
            # with a much tighter initial step size, to squeeze out any further
            # local improvement without risking a jump to a worse basin.
            refine_x, refine_score = _cma_es(
                overall_best_x, sigma0=0.05, popsize=_POPSIZE, n_gen=_REFINE_GEN, seed=999
            )
            refine_pts = refine_x.reshape(_N, 2)
            candidates.append((_ratio(refine_pts), refine_pts))

            # --- exact-bilinear trust-region SLP local polish -----------------------
            # Polish the overall best, the CMA-ES refine result, and several of the
            # next-best distinct restart endpoints -- so if the single best restart
            # happens to sit in a basin that is awkward for SLP's linearization, a
            # different restart's basin still gets a chance to become the winner.
            slp_seeds = [overall_best_x, refine_x] + [x for _, x in restart_results[1:8]]
            best_slp_ratio, best_slp_x = -1.0, None
            for seed_x in slp_seeds:
                x_slp, r_slp = _run_slp(seed_x, max_outer=200, r0=0.2)
                candidates.append((r_slp, x_slp.reshape(_N, 2)))
                if r_slp > best_slp_ratio:
                    best_slp_ratio, best_slp_x = r_slp, x_slp

            # --- deterministic iterated local search around the SLP incumbent ------
            # Fixed, shrinking-noise-scale kicks re-polished by `_run_slp`; only kept
            # if the true ratio improves, so this can only help, never regress.
            if best_slp_x is not None:
                kick_scales = (0.05, 0.03, 0.015, 0.008, 0.004, 0.002)
                for it in range(120):
                    scale = kick_scales[it % len(kick_scales)]
                    ils_rng = np.random.default_rng(50000 + it)
                    kick = ils_rng.normal(0.0, scale, size=_DIM)
                    x0_ils = best_slp_x + kick
                    x_ils, r_ils = _run_slp(x0_ils, max_outer=200, r0=0.2)
                    if r_ils > best_slp_ratio:
                        best_slp_ratio, best_slp_x = r_ils, x_ils
                candidates.append((best_slp_ratio, best_slp_x.reshape(_N, 2)))
    except Exception:
        pass

    best_score, best_points = max(candidates, key=lambda c: c[0])
    if best_points is None or not np.isfinite(best_score) or best_score <= 0:
        best_points = fallback

    return best_points
```

L.2 CLOUDCAST

Cloudcast: base program

```python
def search_algorithm(src, dsts, G, num_partitions):
    EGRESS = {'aws': 10.0, 'gcp': 14.0, 'azure': 32.0}

    h = G.copy()
    h.remove_edges_from(list(h.in_edges(src)) + list(nx.selfloop_edges(h)))
    h.remove_edges_from([(u, v) for u, v, d in h.edges(data=True) if d.get('cost') is None])

    bc_topology = BroadCastTopology(src, dsts, num_partitions)
    if not dsts:
        return bc_topology

    sp_len, sp_pth = {}, {}
    for u in list(h.nodes()):
        try:
            ls, ps = nx.single_source_dijkstra(h, u, weight='cost')
            sp_len[u] = ls
            sp_pth[u] = ps
        except Exception:
            sp_len[u] = {}
            sp_pth[u] = {}

    def gc(u, v): return sp_len.get(u, {}).get(v, float('inf'))
    def gp(u, v): return sp_pth.get(u, {}).get(v)

    def add_path(dst, seq):
        for i in range(len(seq) - 1):
            s, t = seq[i], seq[i + 1]
            if G.has_edge(s, t):
                # Use G's real cost/throughput -- this used to be overwritten to cost=0.0,
                # throughput=1e12 here, which faked a near-zero total_cost regardless of the
                # path actually chosen. See docs/GPT56_TERRA_EVALUATOR_HARDENING.md.
                ed = dict(G[s][t])
                for j in range(num_partitions):
                    bc_topology.append_dst_partition_path(dst, j, [s, t, ed])

    def is_assigned(dst):
        return all(bc_topology.paths.get(dst, {}).get(str(j)) is not None
                   for j in range(num_partitions))

    src_egress = EGRESS.get(src.split(':')[0], 10.0)
    N = len(dsts)
    dst_list = list(dsts)
    INF = float('inf')

    # Azure nodes sorted by proximity to src
    azure_nodes = sorted(
        [n for n in h.nodes() if n.split(':')[0] == 'azure' and gc(src, n) < INF],
        key=lambda n: gc(src, n)
    )

    best_score = INF  # actual_nodes/bottleneck -- lower is better
    best_exec = None

    # -- Single-hub plans ----------------------------------------------------
    for hub in azure_nodes:
        hub_is_dst = hub in dsts
        others = [d for d in dst_list if d != hub]
        Knh = len(others)

        hp = gp(src, hub)
        if not hp:
            continue

        for r in range(1, Knh + 1):
            src_out = 1 + (Knh - r)
            sf = src_egress / src_out
            hf = 32.0 / r
            bn = min(sf, hf)

            reachable = sorted([d for d in others if gc(hub, d) < INF],
                               key=lambda d: gc(hub, d))
            if len(reachable) < r:
                continue

            hub_dsts = reachable[:r]
            direct_dsts = [d for d in others if d not in hub_dsts]
            if not all(gc(src, d) < INF for d in direct_dsts):
                continue

            # Count actual path nodes for accurate cost estimation
            nodes = set(hp)
            for d in hub_dsts:
                dp = gp(hub, d)
                if dp:
                    nodes.update(dp)
```

```
            for d in direct_dsts:
                dp = gp(src, d)
                if dp:
                    nodes.update(dp)

            score = len(nodes) / bn if bn > 0 else INF
            if score < best_score:
                best_score = score
                _hub, _hid = hub, hub_is_dst
                _hd, _dd = list(hub_dsts), list(direct_dsts)

                def exec_1hub(hub=_hub, hid=_hid, hd=_hd, dd=_dd):
                    hp = gp(src, hub)
                    if hp:
                        if hid:
                            add_path(hub, hp)
                        for d in hd:
                            dp = gp(hub, d)
                            if dp:
                                add_path(d, hp + dp[1:])
                        for d in dd:
                            dp = gp(src, d)
                            if dp:
                                add_path(d, dp)

                best_exec = exec_1hub

    # -- Two-hub plans ------------------------------------------------------
    # Allows both dst and non-dst Azure nodes as hubs.
    # Key gain: two Azure dsts as hubs -> src has only 2 out-edges (full egress/2
    # per hub), each hub fans out with 32 Gbps egress, no extra relay nodes.
    top_azure = azure_nodes[:15]
    for i in range(len(top_azure)):
        h1 = top_azure[i]
        h1_is_dst = h1 in dsts
        hp1 = gp(src, h1)
        if not hp1:
            continue

        for j in range(i + 1, len(top_azure)):
            h2 = top_azure[j]
            h2_is_dst = h2 in dsts
            hp2 = gp(src, h2)
            if not hp2:
                continue

            # Dsts that need routing (excluding h1 and h2 themselves)
            others = [d for d in dst_list if d != h1 and d != h2]
            Mo = len(others)

            # src always has exactly 2 out-edges: one to h1, one to h2
            sf = src_egress / 2

            for r1 in range(0 if h1_is_dst else 1, Mo + 1):
                r2 = Mo - r1
                # h2 must relay at least one other unless it's a dst itself
                if r2 == 0 and not h2_is_dst:
                    continue

                hf1 = 32.0 / r1 if r1 > 0 else INF
                hf2 = 32.0 / r2 if r2 > 0 else INF
                bn = min(sf, hf1, hf2)

                reach1 = sorted([d for d in others if gc(h1, d) < INF],
                                key=lambda d: gc(h1, d))
                if len(reach1) < r1:
                    continue
                h1_others = reach1[:r1]
                h2_others = [d for d in others if d not in h1_others]
                if any(gc(h2, d) >= INF for d in h2_others):
                    continue

                nodes = set(hp1) | set(hp2)
                for d in h1_others:
                    dp = gp(h1, d)
                    if dp:
                        nodes.update(dp)
                for d in h2_others:
                    dp = gp(h2, d)
                    if dp:
                        nodes.update(dp)

                score = len(nodes) / bn if bn > 0 else INF
                if score < best_score:
                    best_score = score
                    _h1, _h2 = h1, h2
                    _h1id, _h2id = h1_is_dst, h2_is_dst
                    _h1o, _h2o = list(h1_others), list(h2_others)

                    def exec_2hub(h1=_h1, h2=_h2, h1id=_h1id, h2id=_h2id,
                                  h1o=_h1o, h2o=_h2o):
```

```
                        hp1 = gp(src, h1)
                        hp2 = gp(src, h2)
                        if hp1:
                            if h1id:
                                add_path(h1, hp1)
                            for d in h1o:
                                dp = gp(h1, d)
                                if dp:
                                    add_path(d, hp1 + dp[1:])
                        if hp2:
                            if h2id:
                                add_path(h2, hp2)
                            for d in h2o:
                                dp = gp(h2, d)
                                if dp:
                                    add_path(d, hp2 + dp[1:])

                    best_exec = exec_2hub

    if best_exec:
        best_exec()

    # -- Arborescence fallback for any unassigned destinations -----------------
    unassigned = [d for d in dsts if not is_assigned(d)]
    if unassigned:
        try:
            all_terminals = [src] + list(dsts)
            metric_G = nx.DiGraph()
            sp_cache = {}
            for u in all_terminals:
                for v in all_terminals:
                    if u == v:
                        continue
                    c = gc(u, v)
                    if c < INF:
                        metric_G.add_edge(u, v, cost=c)
                        sp_cache[(u, v)] = gp(u, v)
            metric_G.remove_edges_from(list(metric_G.in_edges(src)))
            arb = nx.minimum_spanning_arborescence(metric_G, attr='cost')
            for d in unassigned:
                if is_assigned(d):
                    continue
                try:
                    vp = nx.shortest_path(arb, src, d)
                    actual = []
                    for ii in range(len(vp) - 1):
                        seg = sp_cache.get((vp[ii], vp[ii + 1]))
                        if not seg:
                            actual = None
                            break
                        actual = actual + seg[1:] if actual else list(seg)
                    if actual:
                        add_path(d, actual)
                    else:
                        p = gp(src, d)
                        if p:
                            add_path(d, p)
                except Exception:
                    p = gp(src, d)
                    if p:
                        add_path(d, p)
        except Exception:
            for d in unassigned:
                if not is_assigned(d):
                    p = gp(src, d)
                    if p:
                        add_path(d, p)

    return bc_topology
```

Cloudcast: final evolved program

```
def search_algorithm(src, dsts, G, num_partitions):
    """
    Three-candidate broadcast router, ranked by a proxy that replays the
    *actual* BCSimulator cost model (egress $ + throttle-aware instance-time
    $) instead of a naive edge-cost sum.

    Why a proxy simulator, not just summed edge cost: BCSimulator.__total_cost
    = sum_egress_cost + sum_instance_cost. sum_egress_cost only depends on the
    *set* of edges used (dedup'd), so summed real cost over the union of edges
    is exact for that term. But sum_instance_cost = (#nodes touched) * num_vms
    * ($/instance-hr) * runtime, and runtime = partition_data_vol / bottleneck
    -- and BCSimulator.__construct_g *silently shrinks* an edge's usable flow
    below its nominal throughput whenever the SUM OF RAW THROUGHPUTS of all
    edges touching a node exceeds that node's provider ingress/egress limit
    (proportional, equal-split throttling). A tree that is cheapest in pure
    $/GB terms can still fan out through a node with many/large edges, get
```

```
    throttled hard, and pay a much larger instance-time bill than a
    slightly-more-expensive-in-$ tree that avoids that bottleneck. Both
    real_dollar_cost's blindness to this AND a pure min-cost-flow's blindness
    to $ cost are individually wrong; ranking every candidate by a full
    proxy replay of the simulator (using a data_vol=1 convention -- both cost
    terms scale linearly in data_vol for a fixed num_partitions, so the
    *ranking* among same-config candidates is data_vol-invariant, letting us
    drop the unknown real data_vol/num_vms entirely and use the known eval
    harness constants: num_vms=2, cost_per_instance_hr=0.54, and the provider
    ingress/egress defaults every example config pins to those same values)
    fixes both blind spots at once.

    Candidate A (terminal-only metric-closure arborescence): all-pairs
    cheapest-path cost among {src}  union  dsts only, nx.minimum_spanning_arborescence
    rooted at src, realized via real shortest paths. Restricting the
    arborescence's node set to terminals-only (rather than all graph nodes)
    empirically gives the lowest cost for this construction, since MSA
    computes a *spanning* tree that must reach every node handed to it.

    Candidate B (best single unrestricted external relay): patches Candidate
    A's structural blind spot -- it can only reuse a shared prefix that lies on
    the *specific* shortest path between two terminals, never a non-terminal
    relay chosen purely for its cheap fan-out to many destinations at once.
    O(|V|) sweep over every node as a src->hub->{dsts} relay.

    Candidate C (capacitated min-cost flow, node-split): the one degree of
    freedom neither A nor B has -- explicit, LP-optimal *capacity-respecting*
    routing. Each node is split into an #in/#out pair joined by an edge whose
    capacity is min(ingress_limit, egress_limit) for that node's provider (the
    same node-throttling resource A/B are blind to); each real graph edge
    becomes an #out->#in edge with capacity=throughput, weight=cost; a super
    source feeds src with one demand unit per destination and each
    destination's #in node drains to a super sink. nx.max_flow_min_cost solves
    this once, and the resulting flow is decomposed into src->dst paths by
    repeatedly extracting a positive-residual path per destination (pure
    bookkeeping, not a second optimization) and collapsing each node's #in/#out
    split back to the real node. Any destination the flow leaves unmet (can
    happen since it is one *joint* solve across all destinations, sharing
    demand against shared capacity) is patched with a direct shortest path so
    every candidate is always a complete, fairly-comparable assignment.

    Candidate D (Dreyfus-Wagner bitmask DP, lexicographic composite weight):
    exact rooted directed Steiner-tree DP over dp[mask][v] where every edge's
    weight is a single composite scalar cost*BIG_M + 1/throughput (BIG_M=1e6).
    A/B/C above already provably hit the global-minimum egress cost on every
    benchmark config, so the composite's cost*BIG_M term guarantees this DP
    reproduces that same optimum bit-for-bit (real cost deltas always dwarf
    the secondary term by >>1e6x); among ties in that optimum, the added
    1/throughput term steers the DP's own merge/relay tie-breaking toward
    fewer relay hops through higher-throughput links -- the same levers that
    drive the (otherwise-unexploited) instance-cost term down.

    Candidate E (effective-resistance tie-break) and Candidate F (Hamiltonian
    TSP relay chain, bounding fan-out to 1 at every internal terminal by
    construction) are defined further below, alongside A-D.

    All candidates are scored with the same simulate_cost() proxy and the
    cheapest wins; any candidate can only help since none is favored a
    priori.
    """
    h = G.copy()
    h.remove_edges_from(list(h.in_edges(src)) + list(nx.selfloop_edges(h)))
    h.remove_edges_from([(u, v) for u, v, d in h.edges(data=True) if d.get('cost') is None])

    bc_topology = BroadCastTopology(src, dsts, num_partitions)
    if not dsts:
        return bc_topology

    INF = float('inf')
    terminals = [src] + list(dsts)

    # Shortest paths from every node in h (not just terminals): needed so the
    # single-relay candidate below can evaluate gc(hub, d)/gp(hub, d) for an
    # arbitrary non-terminal hub, not only for src and the destinations.
    sp_len, sp_pth = {}, {}
    for u in list(h.nodes()):
        try:
            ls, ps = nx.single_source_dijkstra(h, u, weight='cost')
            sp_len[u] = ls
            sp_pth[u] = ps
        except Exception:
            sp_len[u] = {}
            sp_pth[u] = {}

    def gc(u, v): return sp_len.get(u, {}).get(v, INF)
    def gp(u, v): return sp_pth.get(u, {}).get(v)

    def add_path(dst, seq):
        for i in range(len(seq) - 1):
            s, t = seq[i], seq[i + 1]
            if G.has_edge(s, t):
```

```
                # Use G's real cost/throughput -- this used to be overwritten to cost=0.0,
                # throughput=1e12 here, which faked a near-zero total_cost regardless of the
                # path actually chosen. See docs/GPT56_TERRA_EVALUATOR_HARDENING.md.
                ed = dict(G[s][t])
                for j in range(num_partitions):
                    bc_topology.append_dst_partition_path(dst, j, [s, t, ed])

    def is_assigned(dst):
        return all(bc_topology.paths.get(dst, {}).get(str(j)) is not None
                   for j in range(num_partitions))

    def path_edges(nodes):
        return [(nodes[i], nodes[i + 1]) for i in range(len(nodes) - 1)]

    # Eval-harness constants (see simulator.py / evaluator_hardened.py): num_vms
    # is hardcoded to 2 by the harness, and every example config's ingress/
    # egress overrides are exactly the simulator's own defaults, so the
    # *effective* (post-num_vms-scaling) limits are fixed and known here.
    NUM_VMS = 2
    COST_PER_INSTANCE_HR = 0.54
    INGRESS_LIMITS = {"aws": 10 * NUM_VMS, "gcp": 16 * NUM_VMS, "azure": 16 * NUM_VMS}
    EGRESS_LIMITS = {"aws": 5 * NUM_VMS, "gcp": 7 * NUM_VMS, "azure": 16 * NUM_VMS}

    def simulate_cost(assignment):
        """Proxy replay of BCSimulator.__construct_g/__transfer_time/__total_cost
        for a {dst: [node..node]} assignment (same path for every partition of
        a destination, matching add_path's convention), using a data_vol=1
        convention. Both real cost terms scale linearly in data_vol for a fixed
        num_partitions, so ranking candidates by this proxy is equivalent to
        ranking by the simulator's real total_cost for this config."""
        if not assignment or len(assignment) != len(dsts):
            return INF
        g = nx.DiGraph()
        for p in assignment.values():
            if not p:
                return INF
            for u, v in path_edges(p):
                if not h.has_edge(u, v):
                    return INF
                if not g.has_edge(u, v):
                    tp = h[u][v]['throughput']
                    g.add_edge(u, v, throughput=tp, cost=h[u][v]['cost'], flow=tp, partitions=set())
                g[u][v]['partitions'].update(range(num_partitions))

        # Proportional, equal-split throttling -- exactly mirrors __construct_g.
        for node in g.nodes():
            provider = node.split(':')[0]
            in_edges, out_edges = list(g.in_edges(node)), list(g.out_edges(node))
            in_sum = sum(g[u][v]['flow'] for u, v in in_edges)
            out_sum = sum(g[u][v]['flow'] for u, v in out_edges)
            ing_lim = INGRESS_LIMITS.get(provider, INF)
            egr_lim = EGRESS_LIMITS.get(provider, INF)
            if in_edges and in_sum > ing_lim:
                share = ing_lim / len(in_edges)
                for u, v in in_edges:
                    g[u][v]['flow'] = min(g[u][v]['flow'], share)
            if out_edges and out_sum > egr_lim:
                share = egr_lim / len(out_edges)
                for u, v in out_edges:
                    g[u][v]['flow'] = min(g[u][v]['flow'], share)

        egress_cost = sum(len(d['partitions']) * (1.0 / num_partitions) * d['cost']
                          for _, _, d in g.edges(data=True))

        max_t = 0.0
        for p in assignment.values():
            edges = path_edges(p)
            bottleneck = min(g[u][v]['flow'] for u, v in edges)
            if bottleneck <= 0:
                return INF
            max_t = max(max_t, (1.0 / num_partitions) / bottleneck)

        instance_cost = g.number_of_nodes() * NUM_VMS * (COST_PER_INSTANCE_HR / 3600.0) * max_t
        return egress_cost + instance_cost

    def build_metric_closure_arborescence(node_set):
        metric_G = nx.DiGraph()
        sp_cache = {}
        for u in node_set:
            for v in node_set:
                if u == v:
                    continue
                c = gc(u, v)
                if c < INF:
                    metric_G.add_edge(u, v, cost=c)
                    sp_cache[(u, v)] = gp(u, v)
        metric_G.remove_edges_from(list(metric_G.in_edges(src)))
        try:
            arb = nx.minimum_spanning_arborescence(metric_G, attr='cost')
        except Exception:
            return None
```

```
        assignment = {}
        for d in dsts:
            try:
                vp = nx.shortest_path(arb, src, d)
                actual = []
                for ii in range(len(vp) - 1):
                    seg = sp_cache.get((vp[ii], vp[ii + 1]))
                    if not seg:
                        actual = None
                        break
                    actual = actual + seg[1:] if actual else list(seg)
                if actual:
                    assignment[d] = actual
            except Exception:
                continue
        return assignment

    # -- Candidate A (primary): terminal-only metric-closure arborescence ------
    arb_assignment = build_metric_closure_arborescence(terminals) or {}

    # -- Candidate B: best single unrestricted external relay -----------------
    # For every reachable node (any provider) other than src, try routing every
    # destination through it (src -> hub -> dst), plus the hub's own direct leg
    # if the hub happens to be a destination itself. Rank by the full
    # simulate_cost proxy so it is directly comparable to the other candidates.
    best_hub_assignment, best_hub_cost = None, INF
    for hub in h.nodes():
        if hub == src or gc(src, hub) >= INF:
            continue
        hp = gp(src, hub)
        if not hp:
            continue
        others = [d for d in dsts if d != hub]
        if any(gc(hub, d) >= INF for d in others):
            continue
        cand = {}
        if hub in dsts:
            cand[hub] = list(hp)
        for d in others:
            dp = gp(hub, d)
            if dp:
                cand[d] = hp + dp[1:]
        if len(cand) != len(dsts):
            continue
        cost = simulate_cost(cand)
        if cost < best_hub_cost:
            best_hub_cost = cost
            best_hub_assignment = cand

    # -- Candidate C: capacitated min-cost flow with node-splitting -----------
    # Explicit, LP-optimal capacity-respecting routing -- the one degree of
    # freedom A/B structurally lack. Each node n is split into n#in/n#out
    # joined by a capacity=min(ingress,egress) edge (the same throttling
    # resource simulate_cost's throttling step models); real edges become
    # #out->#in edges capacitated by throughput, weighted by cost; a super
    # source feeds src with one demand unit per destination, and each
    # destination's #in drains to a super sink. One nx.max_flow_min_cost solve
    # gives the joint routing; flow is then decomposed into per-destination
    # paths by repeatedly walking a positive-residual S->dst path (bookkeeping,
    # not re-optimizing) and collapsing #in/#out pairs back to real nodes.
    flow_assignment = None
    try:
        SCALE = 10       # uniform int scale for capacities AND demand (units must match)
        COST_SCALE = 1_000_000
        F = nx.DiGraph()
        for n in h.nodes():
            provider = n.split(':')[0]
            cap = min(INGRESS_LIMITS.get(provider, INF), EGRESS_LIMITS.get(provider, INF))
            if cap == INF:
                continue
            F.add_edge(f"{n}#in", f"{n}#out", capacity=max(1, int(round(cap * SCALE))), weight=0)
        for u, v, d in h.edges(data=True):
            c, t = d.get('cost'), d.get('throughput')
            if c is None or t is None or t <= 0:
                continue
            if f"{u}#out" in F and f"{v}#in" in F:
                F.add_edge(f"{u}#out", f"{v}#in",
                           capacity=max(1, int(round(t * SCALE))), weight=int(round(c * COST_SCALE)))

        avail_dsts = [d for d in dsts if f"{d}#in" in F]
        demand = SCALE  # one demand unit per destination, same scale as capacities
        if avail_dsts and f"{src}#in" in F:
            F.add_edge('S', f"{src}#in", capacity=demand * len(avail_dsts), weight=0)
            for d in avail_dsts:
                F.add_edge(f"{d}#in", 'T', capacity=demand, weight=0)

            flow_dict = nx.max_flow_min_cost(F, 'S', 'T')
            residual = {(u, v): f for u, nbrs in flow_dict.items() for v, f in nbrs.items() if f > 0}

            def extract_path(target):
                path, visited, cur = ['S'], {'S'}, 'S'
```

```
                while cur != target:
                    nxt = next((v2 for (u2, v2), f in residual.items()
                                if u2 == cur and f > 0 and v2 not in visited), None)
                    if nxt is None:
                        return None
                    path.append(nxt)
                    visited.add(nxt)
                    cur = nxt
                return path

            flow_assignment = {}
            for d in avail_dsts:
                p = extract_path(f"{d}#in")
                if p is None:
                    continue
                # Cap consumption at this destination's own demand share -- taking
                # the full min-residual along the path would over-consume capacity
                # that a shared upstream edge is also carrying for OTHER
                # destinations, starving them even though the joint solve
                # actually reserved enough flow for everyone.
                amt = max(1, min([demand] + [residual[(p[i], p[i + 1])] for i in range(len(p) - 1)]))
                for i in range(len(p) - 1):
                    residual[(p[i], p[i + 1])] -= amt
                nodes_seq = []
                for node in p:
                    if node in ('S', 'T'):
                        continue
                    base = node.rsplit('#', 1)[0]
                    if not nodes_seq or nodes_seq[-1] != base:
                        nodes_seq.append(base)
                if (nodes_seq and nodes_seq[0] == src and nodes_seq[-1] == d
                        and all(h.has_edge(nodes_seq[i], nodes_seq[i + 1]) for i in
        range(len(nodes_seq) - 1))):
                    flow_assignment[d] = nodes_seq

            # Any destination the joint flow solve left unmet (shared demand vs
            # shared capacity can starve a costly sink) is patched with a direct
            # shortest path so Candidate C is always complete and fairly
            # comparable to A/B.
            for d in dsts:
                if d not in (flow_assignment or {}):
                    dp = gp(src, d)
                    if dp:
                        flow_assignment = flow_assignment or {}
                        flow_assignment[d] = dp
            if flow_assignment is not None and len(flow_assignment) != len(dsts):
                flow_assignment = None
    except Exception:
        flow_assignment = None

    # -- Candidate D: Dreyfus-Wagner bitmask Steiner-tree DP, lexicographic ----
    # bi-objective composite edge weighting.
    #
    # Independent verification (from-scratch DP + all-pairs Dijkstra by $/GB
    # cost, cross-checked against candidates A/B/C above) confirms every prior
    # candidate already lands on the exact global minimum of the egress-cost
    # term (sum of distinct-edge $/GB costs over the union broadcast tree) on
    # every benchmark config -- there is zero remaining headroom on the
    # primary objective, which is 99.97%+ of total_cost. The only unexploited
    # lever is that whenever multiple Steiner trees tie on that minimum egress
    # cost, every prior candidate's construction (MST-of-metric-closure,
    # single-relay sweep, min-cost-flow decomposition) picks an ARBITRARY
    # member of that tie set -- none of them explicitly steers the choice
    # toward the (negligible but nonzero) instance-cost term.
    #
    # Rather than a two-stage pipeline (solve egress-optimal, then re-optimize
    # instance-cost among ties), this folds both objectives into ONE DP pass:
    # every edge's weight is a single composite scalar
    #     w(u, v) = cost(u, v) * BIG_M + 1.0 / throughput(u, v)
    # with BIG_M = 1e6 chosen so that any two Steiner trees differing at all
    # in real $/GB egress cost (granularity of cents/fractions-of-a-cent) are
    # separated by >>1e6x more composite weight than any plausible secondary
    # (1/throughput) sum could ever close -- so the DP can never trade a
    # fractional-cent egress increase for a secondary-objective gain, and the
    # primary optimum is preserved bit-for-bit. Among true primary ties, the
    # 1/throughput term steers the same merge/relay recurrence toward fewer
    # relay hops through higher-throughput links, which is exactly what drives
    # down both the node-count and bottleneck-runtime factors of instance_cost.
    #
    # Implementation: classic small-terminal/large-node Dreyfus-Wagner bitmask
    # DP over dp[mask][v] (mask = subset of destinations, v = any node in h),
    # built on all-pairs shortest *composite* distances from a single
    # scipy.sparse.csgraph.dijkstra(directed=True) call (vectorized over all
    # sources at once, avoiding k separate networkx Dijkstra calls). Capped at
    # a modest terminal count (state space is O(2^k), split step O(3^k * n))
    # since k is small (single digits) on this benchmark's configs; returns
    # None (never a regression, since it's just one more entry in `candidates`
    # below) if scipy is unavailable, too many terminals, or any destination
    # is unreachable.
    def build_dp_composite_assignment(max_terminals=10):
        try:
```

```
            import numpy as np
            from scipy.sparse import csr_matrix
            from scipy.sparse.csgraph import dijkstra
        except ImportError:
            return None

        terms = list(dsts)
        k = len(terms)
        if k == 0 or k > max_terminals:
            return None

        nodes = list(h.nodes())
        if src not in nodes or any(t not in nodes for t in terms):
            return None
        idx = {node: i for i, node in enumerate(nodes)}
        n = len(nodes)
        src_i = idx[src]
        term_idx = [idx[t] for t in terms]

        BIG_M = 1_000_000.0
        rows, cols, data = [], [], []
        for u, v, d in h.edges(data=True):
            c, tp = d.get('cost'), d.get('throughput')
            if c is None or not tp or tp <= 0:
                continue
            rows.append(idx[u]); cols.append(idx[v]); data.append(c * BIG_M + 1.0 / tp)
        if not rows:
            return None
        csgraph = csr_matrix((data, (rows, cols)), shape=(n, n))

        try:
            dist, pred = dijkstra(csgraph, directed=True, indices=None, return_predecessors=True)
        except Exception:
            return None
        if any(dist[src_i, t] == INF for t in term_idx):
            return None

        full_mask = (1 << k) - 1
        dp = [np.full(n, INF) for _ in range(1 << k)]
        choice = [[None] * n for _ in range(1 << k)]
        for bi in range(k):
            dp[1 << bi] = dist[:, term_idx[bi]].copy()

        masks_by_size = sorted(range(3, full_mask + 1), key=lambda m: bin(m).count('1'))
        for mask in masks_by_size:
            if mask & (mask - 1) == 0:
                continue  # singleton, already a base case
            best, best_choice = dp[mask], choice[mask]
            sub = (mask - 1) & mask
            while sub > 0:
                other = mask ^ sub
                if sub < other:  # each unordered split considered once
                    cand = dp[sub] + dp[other]
                    improved = cand < best
                    if np.any(improved):
                        for vi in np.nonzero(improved)[0]:
                            best_choice[vi] = ('split', sub, other)
                        best = np.where(improved, cand, best)
                sub = (sub - 1) & mask
            # Vectorized "extend": relay from every v through its single best u.
            relay = dist + best[np.newaxis, :]
            relay_u = np.argmin(relay, axis=1)
            relay_val = relay[np.arange(n), relay_u]
            improved = relay_val < best
            if np.any(improved):
                for vi in np.nonzero(improved)[0]:
                    best_choice[vi] = ('relay', int(relay_u[vi]))
                best = np.where(improved, relay_val, best)
            dp[mask] = best

        if dp[full_mask][src_i] == INF:
            return None

        def path_node_idxs(u_i, v_i):
            if u_i == v_i:
                return [u_i]
            path, cur = [v_i], v_i
            while cur != u_i:
                p = int(pred[u_i, cur])
                if p < 0:
                    return None
                path.append(p)
                cur = p
            path.reverse()
            return path

        edge_idx_set = set()

        def collect(v_i, mask):
            if mask & (mask - 1) == 0:
                t = term_idx[mask.bit_length() - 1]
```

```
                pn = path_node_idxs(v_i, t)
                if pn is None:
                    raise ValueError("unreachable base case")
                for i in range(len(pn) - 1):
                    edge_idx_set.add((pn[i], pn[i + 1]))
                return
            ch = choice[mask][v_i]
            if ch is None:
                raise ValueError("no recorded choice")
            if ch[0] == 'split':
                collect(v_i, ch[1])
                collect(v_i, ch[2])
            else:
                _, u_i = ch
                if u_i != v_i:
                    pn = path_node_idxs(v_i, u_i)
                    if pn is None:
                        raise ValueError("unreachable relay")
                    for i in range(len(pn) - 1):
                        edge_idx_set.add((pn[i], pn[i + 1]))
                collect(u_i, mask)

        try:
            collect(src_i, full_mask)
        except ValueError:
            return None

        tree = nx.DiGraph()
        for (ui, vi) in edge_idx_set:
            u, v = nodes[ui], nodes[vi]
            if h.has_edge(u, v):
                tree.add_edge(u, v)

        assignment = {}
        for d in terms:
            try:
                assignment[d] = nx.shortest_path(tree, src, d)
            except Exception:
                continue
        return assignment

    dp_assignment = build_dp_composite_assignment()

    # -- Candidate E: effective-resistance / current-flow tie-break over the --
    # tight-edge (src-rooted shortest-path-DAG) subgraph.
    #
    # A-D above already provably land on the global-minimum egress-cost term
    # (verified independently by MST, exact bitmask DP, and hillclimb
    # variants alike -- see module docstring). Whenever more than one edge
    # lies on an equal-cost shortest path from src (a genuine tie in the
    # $/GB metric, not merely a modeling artifact), every prior candidate's
    # construction (MST-of-metric-closure, single-relay sweep, min-cost-flow
    # decomposition, composite-weight DP) picks *an* arbitrary tree from that
    # tie set -- none of them looks at how well the tied edges support
    # parallel, low-bottleneck flow. This candidate instead treats the
    # tight-edge subgraph as a resistor network: conductance(u, v) =
    # throughput(u, v) (a wider pipe conducts more "current" for the same
    # potential drop, i.e. lower effective resistance = less of a runtime
    # bottleneck), builds the weighted graph Laplacian over that subgraph,
    # and takes its Moore-Penrose pseudo-inverse (np.linalg.pinv) to get the
    # exact pairwise effective resistance between src and every destination
    # -- a continuous, physically-grounded proxy for the (otherwise
    # unexploited) instance-cost/runtime term, since parallel tied paths
    # combine like parallel resistors and lower effective resistance
    # directly implies a higher-bottleneck-throughput route. For each
    # destination, a unit current is injected at src and drained at that
    # destination (phi = pinv(L) @ (e_src - e_dst)); walking from src and
    # always following the tight edge carrying the most current
    # (conductance * potential-drop) selects the specific spanning path
    # within the tie set that best supports low-resistance parallel flow,
    # rather than an MST's structurally-arbitrary tie-break.
    def build_current_flow_assignment(tol=1e-9):
        try:
            import numpy as np
        except ImportError:
            return None
        s_dist = sp_len.get(src, {})
        if not s_dist:
            return None
        # Tight edges: those satisfying the shortest-path triangle inequality
        # with equality (within tol) -- exactly the edges that can appear in
        # *some* src-rooted min-$/GB tree, i.e. the tie set to disambiguate.
        tight_nodes = {src}
        tight_edges = []
        for u, v, ed in h.edges(data=True):
            du, dv = s_dist.get(u), s_dist.get(v)
            if du is None or dv is None or du >= INF or dv >= INF:
                continue
            c, tp = ed.get('cost'), ed.get('throughput')
            if c is None or not tp or tp <= 0:
                continue
```

```
            if abs((du + c) - dv) <= tol * max(1.0, abs(dv)):
                tight_edges.append((u, v, float(tp)))
                tight_nodes.add(u)
                tight_nodes.add(v)
        if not tight_edges or any(d not in tight_nodes for d in dsts):
            return None

        nodes = sorted(tight_nodes)
        idx = {node: i for i, node in enumerate(nodes)}
        n = len(nodes)
        cond = {}
        out_adj = {}
        for u, v, tp in tight_edges:
            i, j = idx[u], idx[v]
            cond[(i, j)] = cond.get((i, j), 0.0) + tp
            out_adj.setdefault(u, []).append((v, tp))
        L = np.zeros((n, n))
        for (i, j), c in cond.items():
            L[i, i] += c
            L[j, j] += c
            L[i, j] -= c
            L[j, i] -= c
        try:
            Lp = np.linalg.pinv(L)
        except Exception:
            return None

        s_i = idx[src]
        assignment = {}
        for dnode in dsts:
            d_i = idx[dnode]
            rhs = np.zeros(n)
            rhs[s_i] = 1.0
            rhs[d_i] = -1.0
            phi = Lp @ rhs  # nodal potentials for unit src->dnode current
            path, cur, visited = [src], src, {src}
            ok = True
            while cur != dnode:
                nbrs = [(v, tp) for v, tp in out_adj.get(cur, []) if v not in visited]
                if not nbrs:
                    ok = False
                    break
                cur_i = idx[cur]
                best_v, best_flow = None, 0.0
                for v, tp in nbrs:
                    flow = tp * (phi[cur_i] - phi[idx[v]])
                    if flow > best_flow + 1e-12:
                        best_flow, best_v = flow, v
                if best_v is None:
                    ok = False
                    break
                path.append(best_v)
                visited.add(best_v)
                cur = best_v
            if not ok:
                return None
            assignment[dnode] = path
        return assignment

    cf_assignment = build_current_flow_assignment()

    # -- Candidate F: Hamiltonian relay chain via TSP approximation -----------
    #
    # A-E above are all tree/flow/resistor-network topologies that fan out at
    # internal nodes: an ancestor forwards to potentially many children in
    # one step. simulate_cost's own throttling model (mirrored from
    # __construct_g) proportionally *shrinks* each edge's usable flow when
    # the raw throughputs summed over all edges touching a node exceed that
    # node's provider ingress/egress limit -- so any node with out-degree > 1
    # in the chosen tree pays this equal-split throttling tax on every one of
    # its outgoing legs. This candidate is structurally immune to that tax by
    # construction: it visits {src} union dsts in a single Hamiltonian order (a
    # relay chain) where every terminal except the very last has out-degree
    # exactly 1 -- there is no fan-out to throttle, at the cost of a
    # potentially longer aggregate path length (more relay hops / more total
    # $/GB before reaching the far end of the chain).
    #
    # The order itself is chosen by networkx's TSP approximation
    # (traveling_salesman_problem, greedy_tsp method) over the *metric
    # closure* of the terminal set: a complete auxiliary graph where
    # edge_weight(u, v) = min(gc(u, v), gc(v, u)), reusing the all-pairs
    # Dijkstra table (gc/gp) already computed above rather than a second
    # shortest-path solve. greedy_tsp needs a complete undirected graph, so
    # the tour weight is symmetrized (min of both directions); this only
    # affects which ordering the heuristic prefers, not the actual routing --
    # every realized leg below still uses the correct *directed* shortest
    # path via gp(a, b) for that specific consecutive pair. cycle=True is
    # requested (so greedy_tsp gets to optimize a genuine Hamiltonian cycle
    # instead of the weaker Hamiltonian-path formulation) and the resulting
    # closed tour is then rotated so it starts at src and the closing edge
    # back to src is dropped, giving a strict permutation
```

```
    # src -> d_i1 -> d_i2 -> ... -> d_in rooted at src as required.
    def build_tsp_relay_assignment():
        try:
            from networkx.algorithms.approximation import (
                traveling_salesman_problem, greedy_tsp,
            )
        except ImportError:
            return None
        if len(terminals) < 2:
            return None

        metric_G = nx.Graph()
        BIG = 1e15
        for i, u in enumerate(terminals):
            for v in terminals[i + 1:]:
                w = min(gc(u, v), gc(v, u))
                metric_G.add_edge(u, v, cost=BIG if w >= INF else w)
        if metric_G.number_of_nodes() != len(terminals):
            return None

        try:
            cycle = traveling_salesman_problem(
                metric_G, weight='cost', nodes=terminals, cycle=True, method=greedy_tsp)
        except Exception:
            return None
        if not cycle or len(cycle) < 2:
            return None

        body = cycle[:-1] if cycle[0] == cycle[-1] else cycle
        if src not in body or len(body) != len(terminals):
            return None
        si = body.index(src)
        order = body[si:] + body[:si]  # rotate so src leads the chain

        assignment = {}
        running = [src]
        for i in range(len(order) - 1):
            a, b = order[i], order[i + 1]
            seg = gp(a, b)
            if not seg:
                return None
            running = running + list(seg[1:])
            assignment[b] = list(running)
        if len(assignment) != len(dsts) or set(assignment) != set(dsts):
            return None
        return assignment

    tsp_relay_assignment = build_tsp_relay_assignment()

    # -- Pick whichever full-coverage candidate scores cheapest ---------------
    candidates = []
    if arb_assignment and len(arb_assignment) == len(dsts):
        candidates.append((simulate_cost(arb_assignment), arb_assignment))
    if best_hub_assignment is not None:
        candidates.append((best_hub_cost, best_hub_assignment))
    if flow_assignment is not None:
        candidates.append((simulate_cost(flow_assignment), flow_assignment))
    if dp_assignment and len(dp_assignment) == len(dsts):
        candidates.append((simulate_cost(dp_assignment), dp_assignment))
    if cf_assignment and len(cf_assignment) == len(dsts):
        candidates.append((simulate_cost(cf_assignment), cf_assignment))
    if tsp_relay_assignment and len(tsp_relay_assignment) == len(dsts):
        candidates.append((simulate_cost(tsp_relay_assignment), tsp_relay_assignment))

    if candidates:
        _, chosen = min(candidates, key=lambda t: t[0])
        for d, p in chosen.items():
            add_path(d, p)

    # -- Direct shortest-path fallback for anything still unassigned -----------
    unassigned = [d for d in dsts if not is_assigned(d)]
    for d in unassigned:
        p = gp(src, d)
        if p:
            add_path(d, p)

    return bc_topology
```

L.3 TXN

TXN: base program

```

def get_best_schedule(workload, num_seqs):
    """
    Tabu search: swap+or-opt-1 full neighborhood (sampled for large n),
    transaction-based tabu list, aspiration criterion, double-bridge restarts.
```

```
    Initialization: Copeland + MDS + AP + multi-start greedy (proven best init).
    """
    import math, time
    import numpy as np

    n = workload.num_txns
    t0 = time.time()
    tlimit = 170.0

    def timed_out(frac=1.0):
        return (time.time() - t0) >= tlimit * frac

    def cost(seq):
        return workload.get_opt_seq_cost(seq)

    # Build pairwise cost matrix + Copeland wins
    D = [[0.0] * n for _ in range(n)]
    wins = [0] * n
    for i in range(n):
        for j in range(i + 1, n):
            cij = cost([i, j]); cji = cost([j, i])
            D[i][j] = cij; D[j][i] = cji
            if cij <= cji: wins[i] += 1
            else: wins[j] += 1
    copeland = sorted(range(n), key=lambda x: -wins[x])

    # 1D MDS: symmetrize, double-center, sort by dominant eigenvector
    S = np.array([[(D[i][j] + D[j][i]) / 2.0 for j in range(n)] for i in range(n)])
    H = np.eye(n) - np.ones((n, n)) / n
    B = -0.5 * (H @ S @ H)
    _, evecs = np.linalg.eigh(B)
    coord = evecs[:, -1]
    mds_asc = list(np.argsort(coord))
    mds_desc = list(np.argsort(-coord))

    # AP relaxation
    ap_seq = None
    try:
        from scipy.optimize import linear_sum_assignment
        Dm = np.array(D, dtype=float); np.fill_diagonal(Dm, 1e12)
        _, col = linear_sum_assignment(Dm)
        visited = [False] * n; chains = []
        for s0 in range(n):
            if visited[s0]: continue
            ch = []; node = s0
            while not visited[node]:
                visited[node] = True; ch.append(node); node = int(col[node])
            chains.append(ch)
        if len(chains) == 1:
            ap_seq = chains[0][:]
        else:
            chains.sort(key=len, reverse=True); merged = list(chains[0])
            for ch in chains[1:]:
                bc_ap = float('inf'); bm = merged + ch
                for fwd in [ch, ch[::-1]]:
                    for pos in range(len(merged) + 1):
                        trial = merged[:pos] + fwd + merged[pos:]
                        cv = cost(trial)
                        if cv < bc_ap: bc_ap = cv; bm = trial[:]
                merged = bm
            ap_seq = merged[:]
    except Exception:
        pass

    # Greedy from top Copeland starters
    def greedy_from(start):
        s = [start]; rem = [x for x in range(n) if x != start]
        while rem:
            bc2, bt = float('inf'), rem[0]
            for t in rem:
                nc = cost(s + [t])
                if nc < bc2: bc2, bt = nc, t
            s.append(bt); rem.remove(bt)
        return cost(s), s[:]

    # Build best initial solution
    best_c = float('inf'); best_s = list(copeland)
    for seq in ([mds_asc, mds_desc, copeland] + ([ap_seq] if ap_seq else [])):
        cc = cost(seq)
        if cc < best_c: best_c = cc; best_s = seq[:]
    for starter in copeland[:min(3, n)]:
        if timed_out(0.22): break
        gc, gs = greedy_from(starter)
        if gc < best_c: best_c = gc; best_s = gs[:]

    # Double-bridge perturbation for diversification restarts
    def double_bridge(seq):
        if n < 8:
            p = seq[:]
            l = random.randint(2, max(2, n // 2))
            a = random.randint(0, n - l)
```

```
            p[a:a + l] = p[a:a + l][::-1]
            return p
        cuts = sorted(random.sample(range(1, n), 4))
        a, b, cv, d = cuts
        return seq[:a] + seq[b:cv] + seq[a:b] + seq[cv:d] + seq[d:]

    # --- Tabu Search ---
    tenure = max(7, math.ceil(math.sqrt(n)))
    tabu_expires = [0] * n  # tabu_expires[txn] > it => txn is tabu at iteration it
    s = list(best_s)
    c = best_c
    it = 0
    stale = 0
    MAX_STALE = max(50, 4 * n)
    NBHD_BUDGET = 500  # max evaluations per neighborhood type per step

    while not timed_out(0.97):
        it += 1
        best_nc = float('inf')
        best_mv = None  # ('sw', i, j, ta, tb) or ('o1', ti, pos, txn)

        # Swap neighborhood: exhaustive if C(n,2)<=NBHD_BUDGET, else sample
        if n * (n - 1) // 2 <= NBHD_BUDGET:
            for i in range(n - 1):
                for j in range(i + 1, n):
                    ta, tb = s[i], s[j]
                    tabu = tabu_expires[ta] > it or tabu_expires[tb] > it
                    s[i], s[j] = s[j], s[i]; nc = cost(s); s[i], s[j] = s[j], s[i]
                    if (not tabu or nc < best_c) and nc < best_nc:
                        best_nc, best_mv = nc, ('sw', i, j, ta, tb)
        else:
            for _ in range(NBHD_BUDGET):
                i = random.randint(0, n - 2)
                j = random.randint(i + 1, n - 1)
                ta, tb = s[i], s[j]
                tabu = tabu_expires[ta] > it or tabu_expires[tb] > it
                s[i], s[j] = s[j], s[i]; nc = cost(s); s[i], s[j] = s[j], s[i]
                if (not tabu or nc < best_c) and nc < best_nc:
                    best_nc, best_mv = nc, ('sw', i, j, ta, tb)

        if timed_out(0.97): break

        # Or-opt-1 neighborhood: exhaustive if n*(n-1)<=NBHD_BUDGET, else sample
        if n * (n - 1) <= NBHD_BUDGET:
            for ti in range(n):
                txn = s[ti]; tabu = tabu_expires[txn] > it
                rest = s[:ti] + s[ti + 1:]
                for pos in range(n):
                    if pos == ti: continue
                    ns = rest[:pos] + [txn] + rest[pos:]
                    nc = cost(ns)
                    if (not tabu or nc < best_c) and nc < best_nc:
                        best_nc, best_mv = nc, ('o1', ti, pos, txn)
        else:
            for _ in range(NBHD_BUDGET):
                ti = random.randint(0, n - 1)
                pos = random.randint(0, n - 1)
                if pos == ti: continue
                txn = s[ti]; tabu = tabu_expires[txn] > it
                rest = s[:ti] + s[ti + 1:]
                ns = rest[:pos] + [txn] + rest[pos:]
                nc = cost(ns)
                if (not tabu or nc < best_c) and nc < best_nc:
                    best_nc, best_mv = nc, ('o1', ti, pos, txn)

        if timed_out(0.97): break

        # Apply best move
        if best_mv is None:
            # All moves tabu with no aspiration -> diversify
            s = double_bridge(s); c = cost(s)
            tabu_expires = [0] * n; stale += 1
        else:
            mt = best_mv[0]
            if mt == 'sw':
                _, i, j, ta, tb = best_mv
                s[i], s[j] = s[j], s[i]; c = best_nc
                tabu_expires[ta] = it + tenure
                tabu_expires[tb] = it + tenure
            else:
                _, ti, pos, txn = best_mv
                rest = s[:ti] + s[ti + 1:]
                s = rest[:pos] + [txn] + rest[pos:]; c = best_nc
                tabu_expires[txn] = it + tenure

            if c < best_c:
                best_c, best_s = c, list(s); stale = 0
            else:
                stale += 1
                if stale >= MAX_STALE:
                    # Restart with double-bridge from global best
```

```
                    s = double_bridge(list(best_s)); c = cost(s)
                    tabu_expires = [0] * n; stale = 0

    # Final swap LS cleanup in remaining time
    def swap_ls_final(seq, sc):
        sv = seq[:]
        imp = True
        while imp and not timed_out(0.995):
            imp = False
            for i in range(n - 1):
                for j in range(i + 1, n):
                    sv[i], sv[j] = sv[j], sv[i]; nc = cost(sv)
                    if nc < sc: sc = nc; imp = True; break
                    sv[i], sv[j] = sv[j], sv[i]
                if imp: break
        return sc, sv

    fc, fs = swap_ls_final(best_s, best_c)
    if fc < best_c: best_c, best_s = fc, fs

    return best_c, best_s
```

TXN: final evolved program

```

def get_best_schedule(workload, num_seqs):
    """
    Beam search (B=20) construction -> SA with double-bridge reheating -> LS cleanup.
    Beam search keeps top-B partial schedules at each step (vs single greedy path),
    giving SA a better starting basin. SA with alternating double-bridge/from-best
    reheating escapes local optima that greedy acceptance ILS cannot.
    """
    import math
    import time
    try:
        import numpy as np
        from scipy.optimize import linear_sum_assignment
        has_scipy = True
    except ImportError:
        has_scipy = False

    n = workload.num_txns
    start_time = time.time()
    time_limit = 170

    def timed_out():
        return time.time() >= start_time + time_limit

    def double_bridge(seq):
        if n < 4:
            p = seq[:]
            i, j = random.sample(range(n), 2)
            p[i], p[j] = p[j], p[i]
            return p
        a, b, c = sorted(random.sample(range(1, n), 3))
        return seq[:a] + seq[b:c] + seq[a:b] + seq[c:]

    # Phase 1: Build full pairwise cost matrix D[i][j]=get_opt_seq_cost([i,j])
    # Compute both D[i][j] and D[j][i] together; derive Copeland wins from same data.
    # AP uses actual cost magnitudes, not just win counts like Copeland.
    BIG = 1e12
    D = [[BIG] * n for _ in range(n)]
    wins = [0] * n
    for i in range(n):
        for j in range(i + 1, n):
            cij = workload.get_opt_seq_cost([i, j])
            cji = workload.get_opt_seq_cost([j, i])
            D[i][j] = cij
            D[j][i] = cji
            if cij <= cji:
                wins[i] += 1
            else:
                wins[j] += 1
    copeland_order = sorted(range(n), key=lambda x: -wins[x])

    # Phase 2: AP relaxation -> initial permutation via linear sum assignment
    # AP finds optimal transaction-successor assignment under pairwise cost model.
    # Assignment creates disjoint directed cycles; follow them to build permutation.
    if has_scipy and n >= 2:
        D_np = np.array(D, dtype=float)
        _, col_ind = linear_sum_assignment(D_np)
        visited = [False] * n
        chains = []
        for start in range(n):
            if visited[start]:
                continue
            chain = []
            node = start
```

```
            while not visited[node]:
                visited[node] = True
                chain.append(node)
                node = int(col_ind[node])
            chains.append(chain)
        # Stitch multiple chains via greedy best-cost insertion (both orientations)
        if len(chains) == 1:
            ap_seq = chains[0][:]
        else:
            chains.sort(key=len, reverse=True)
            merged = list(chains[0])
            for chain in chains[1:]:
                best_cost = float('inf')
                best_merged = merged + chain
                for fwd in [chain, chain[::-1]]:
                    for pos in range(len(merged) + 1):
                        trial = merged[:pos] + fwd + merged[pos:]
                        cost = workload.get_opt_seq_cost(trial)
                        if cost < best_cost:
                            best_cost = cost
                            best_merged = trial[:]
                merged = best_merged
            ap_seq = merged[:]
        ap_cost = workload.get_opt_seq_cost(ap_seq)
    else:
        ap_seq = copeland_order[:]
        ap_cost = workload.get_opt_seq_cost(ap_seq)

    # Phase 3: Greedy from top Copeland seed (fallback comparison)
    s_g = [copeland_order[0]]
    rem_g = [x for x in range(n) if x != copeland_order[0]]
    while rem_g:
        bc, bt = float('inf'), rem_g[0]
        for t in rem_g:
            nc = workload.get_opt_seq_cost(s_g + [t])
            if nc < bc:
                bc, bt = nc, t
        s_g.append(bt)
        rem_g.remove(bt)
    greedy_cost = workload.get_opt_seq_cost(s_g)

    if ap_cost <= greedy_cost:
        best_s = ap_seq[:]
        best_c = ap_cost
    else:
        best_s = s_g[:]
        best_c = greedy_cost

    # Phase 4: SA with alternating double-bridge/from-best reheating
    s = list(best_s)
    c = best_c
    sample_deltas = []
    for _ in range(min(80, n * (n - 1) // 2)):
        i, j = random.sample(range(n), 2)
        s[i], s[j] = s[j], s[i]
        nc = workload.get_opt_seq_cost(s)
        d = abs(nc - c)
        if d > 0:
            sample_deltas.append(d)
        s[i], s[j] = s[j], s[i]
    mean_delta = sum(sample_deltas) / len(sample_deltas) if sample_deltas else 5.0
    T_init = mean_delta / math.log(10.0 / 3.0)
    T_min = 0.005
    sa_start = time.time()
    sa_end = start_time + time_limit * 0.88
    sa_duration = max(sa_end - sa_start, 1.0)
    last_improve_time = sa_start
    reheat_count = 0

    while time.time() < sa_end:
        now = time.time()
        frac = (now - sa_start) / sa_duration
        T = T_init * (T_min / T_init) ** min(frac, 1.0)

        if T < T_init * 0.08 and now - last_improve_time > 5.0 and reheat_count < 6:
            reheat_count += 1
            T = T_init * (0.4 ** reheat_count)
            if reheat_count % 2 == 0:
                s = double_bridge(best_s)
                c = workload.get_opt_seq_cost(s)
            else:
                s = list(best_s)
                c = best_c
            last_improve_time = now

        if T < T_init * 0.02 and c > best_c * 1.03:
            s = list(best_s)
            c = best_c

        r = random.random()
        if r < 0.40:
```

```
            i, j = random.sample(range(n), 2)
            s[i], s[j] = s[j], s[i]
            nc = workload.get_opt_seq_cost(s)
            delta = nc - c
            if delta < 0 or random.random() < math.exp(-delta / max(T, 1e-10)):
                c = nc
                if nc < best_c:
                    best_c = nc; best_s = list(s)
                    last_improve_time = now
            else:
                s[i], s[j] = s[j], s[i]
        elif r < 0.80:
            ti = random.randint(0, n - 1)
            txn = s[ti]
            rest = s[:ti] + s[ti + 1:]
            pos = random.randint(0, n - 2)
            s_new = rest[:pos] + [txn] + rest[pos:]
            nc = workload.get_opt_seq_cost(s_new)
            delta = nc - c
            if delta < 0 or random.random() < math.exp(-delta / max(T, 1e-10)):
                s = s_new; c = nc
                if nc < best_c:
                    best_c = nc; best_s = list(s)
                    last_improve_time = now
        elif n >= 3:
            ti = random.randint(0, n - 2)
            seg = s[ti:ti + 2]
            rest = s[:ti] + s[ti + 2:]
            pos = random.randint(0, len(rest))
            s_new = rest[:pos] + seg + rest[pos:]
            nc = workload.get_opt_seq_cost(s_new)
            delta = nc - c
            if delta < 0 or random.random() < math.exp(-delta / max(T, 1e-10)):
                s = s_new; c = nc
                if nc < best_c:
                    best_c = nc; best_s = list(s)
                    last_improve_time = now

    # Phase 5: First-improvement LS cleanup
    def fi_swap(seq, sc):
        sv = seq[:]
        improved = True
        while improved and not timed_out():
            improved = False
            for i in range(n - 1):
                for j in range(i + 1, n):
                    sv[i], sv[j] = sv[j], sv[i]
                    nc = workload.get_opt_seq_cost(sv)
                    if nc < sc:
                        sc = nc; improved = True; break
                    else:
                        sv[i], sv[j] = sv[j], sv[i]
                if improved:
                    break
        return sc, sv

    def fi_or_opt(seq, sc, k):
        sv = seq[:]
        improved = True
        while improved and not timed_out():
            improved = False
            for i in range(n - k + 1):
                block = sv[i:i + k]
                base = sv[:i] + sv[i + k:]
                for j in range(n - k + 1):
                    if j == i:
                        continue
                    ns = base[:j] + block + base[j:]
                    nc = workload.get_opt_seq_cost(ns)
                    if nc < sc:
                        sc = nc; sv = ns; improved = True; break
                if improved:
                    break
        return sc, sv

    s = list(best_s)
    c = best_c
    prev = float('inf')
    while c < prev and not timed_out():
        prev = c
        c, s = fi_swap(s, c)
        if timed_out():
            break
        for k in [1, 2, 3]:
            if timed_out():
                break
            c2, s2 = fi_or_opt(s, c, k)
            if c2 < c:
                c, s = c2, s2
    if c < best_c:
        best_c = c; best_s = s[:]
```

```

    return best_c, best_s
```