Improving Coherence and Persistence in Agentic AI for System Optimization

Pantea Karimi*
MIT CSAIL
Cambridge, USA
pkarimib@mit.edu

Kimia Noorbakhsh*
MIT CSAIL
Cambridge, USA
kimian@mit.edu

Mohammad Alizadeh
MIT CSAIL
Cambridge, USA
alizadeh@csail.mit.edu

Hari Balakrishnan
MIT CSAIL
Cambridge, USA
hari@csail.mit.edu

Abstract

Designing high-performance system heuristics is a creative, iterative process requiring experts to form hypotheses and execute multi-step conceptual shifts. While Large Language Models (LLMs) show promise in automating this loop, they struggle with complex system problems due to two critical failure modes: *evolutionary neighborhood bias* and the *coherence ceiling*. Evolutionary methods often remain trapped in local optima by relying on scalar benchmark scores, failing when coordinated multi-step changes are required. Conversely, existing agentic frameworks suffer from context degradation over long horizons or fail to accumulate knowledge across independent runs.

We present **Engram**, an agentic researcher architecture that addresses these limitations by decoupling long-horizon exploration from the constraints of a single context window. Engram organizes exploration into a sequence of agents that iteratively design, test, and analyze mechanisms. At the conclusion of each run, an agent stores code snapshots, logs, and results in a persistent *Archive* and distills high-level modeling insights into a compact, persistent *Research Digest*. Subsequent agents then begin with a fresh context window, reading the Research Digest to build on prior discoveries.

We find that Engram exhibits superior performance across diverse domains including multi-cloud multicast, LLM inference request routing, and optimizing KV cache reuse in databases with natural language queries.

CCS Concepts

• **Computing methodologies** → **Intelligent agents**; **Multi-agent systems**; • **Networks** → *Network algorithms*.

Keywords

Large language models, Agentic AI, Automated Algorithm Discovery, Systems Optimization, Heuristic Design

ACM Reference Format:
Pantea Karimi, Kimia Noorbakhsh, Mohammad Alizadeh, and Hari Balakrishnan. 2026. Improving Coherence and Persistence in Agentic AI for System Optimization. In *ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA*. ACM, New York, NY, USA, 37 pages. https://doi.org/10.1145/3786335.3813138

*Equal Contribution

CAIS '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2415-2/26/05
https://doi.org/10.1145/3786335.3813138

1 Introduction

Designing high-performance heuristics and algorithms for computer systems is a creative and iterative process. Experts form hypotheses about system bottlenecks, implement candidate mechanisms, test them under realistic workloads, and use the findings to refine the design, often through multi-step conceptual shifts rather than code tweaks. Recent work has explored using large language models (LLMs) to automate this loop [8, 14], yet prior approaches struggle to reliably produce expert-level solutions to complex systems problems.

We identify two critical failure modes:

(1) *Evolutionary neighborhood bias*: Code-evolution systems propose code variants and select them using a scalar benchmark score [3, 22, 23, 35, 40, 42]. This approach can work when progress comes from incremental refinements to a stable template, but it often fails when improvements require coordinated multi-step changes; e.g., reformulating the problem, adding tractable relaxations, or accepting temporary regressions while moving to a different algorithmic family.
(2) The *coherence ceiling*: While agentic frameworks such as Glia [14] enable hypothesis formation and targeted experimentation, they struggle with long-horizon design. A single long-running context eventually suffers from degradation and "context rot," where attention becomes uneven [17]. Conversely, independent "best-of-N" runs do not accumulate knowledge; each run must rediscover identical modeling insights from scratch. Code evolution methods, on the other hand, lack long-horizon coherence because each LLM invocation is unaware of the *thought process* behind previous attempts.

We present **Engram**[1], an agentic researcher architecture that overcomes these limitations by decoupling long-horizon exploration from the constraints of a single context window. Engram organizes exploration into a sequence of agents that iteratively design, test, and analyze mechanisms. At the conclusion of each run, an agent archives code snapshots, execution logs, and experimental results into a persistent *Archive*. Crucially, Engram introduces a *structured handoff* wherein each agent distills high-level insights, findings, and failure diagnoses into a compact *Research Digest*. Every subsequent agent begins with a fresh context window, reading the Research Digest to build upon prior discoveries and findings. This architecture enables Engram to sustain a coherent research exploration across hundreds of trials, bypassing the performance degradation typical of single-context agentic threads.

Our contributions are:

[1]Engram's code is available at https://github.com/mit-nms/Engram.

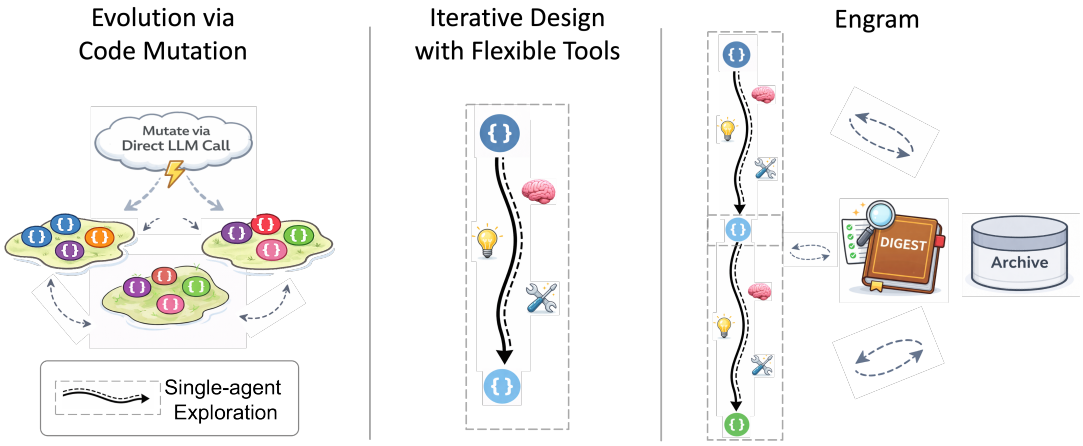


Figure 1: The three paradigms for LLM-based heuristic design. Evolutionary approaches with code mutation invoke an LLM with a predefined context format, mutating and selecting candidates based on scalar scores. Iterative design with flexible tool access (e.g., Glia) performs coherent experiment-guided exploration, but each exploration is restricted to a bounded LLM context window. Engram combines agent explorations with a shared research digest that persists insights across explorations (§3), improving persistence while preserving long-horizon coherence and flexibility.

- **Enhanced long-horizon coherence.** We introduce a structured handoff and archival mechanism that enables cumulative progress across agents. The persistent Research Digest ensures that modeling insights and failure diagnoses persist beyond the lifetime of any single agent exploration.
- **Discovery of new system heuristics that surpass prior state-of-the-art.** Engram discovers novel heuristics across multiple domains. In multi-cloud multicast [50], Engram synthesizes new multicast heuristics that achieve a best overall cost of $622, surpassing the reported human state-of-the-art (SOTA) of $626 and all evolutionary baselines. For LLM inference request routing [33], Engram discovers improved strategies that reduce mean response time to 23.9 s, outperforming the expert-designed heuristic as well as Glia [14] (25.7 s) and all evolutionary baselines.
- **Outperforming prior approaches across diverse problems.** We evaluate Engram on nine systems problems, eight from the ADRS benchmark [47] and the ninth an LLM request router [14]. Engram outperforms human SOTA in eight of nine settings and exceeds (in 7) or matches (in the other 2) OpenEvolve in all evaluated categories, with all methods using the same LLM in each comparison (§4, §5).

2 Why LLMs Struggle on System Optimization Problems

Optimizing systems and designing heuristics is fundamentally a creative, multi-step process. High-performance mechanisms rarely emerge from local code tweaks; instead, they require formulating hypotheses about bottlenecks, constructing and conducting experiments, interpreting experimental results, and executing multi-step conceptual shifts [14].

Recent research has sought to use LLM-based agents to automate this loop [8, 14]. We categorize current approaches for such heuristic design into two paradigms and analyze why they struggle to sustain long-horizon progress. These paradigms are: (i) evolution via code mutation (e.g., OpenEvolve [42]) and (ii) iterative design with reasoning and tools (e.g., Glia [14]). We analyze both along three criteria:

Approach	Coherence	Flexibility	Persistence
Evolutionary code mutation	Low	Low	High
Iterative design via flexible tools	High	High	Low
Engram	**High**	**High**	**High**

Table 1: Trade-offs among LLM-based heuristic design. Code-mutation evolution is persistent, but limited in coherence and flexibility; tool-based agents enable coherent and flexible exploration but have limited persistence. Engram achieves all three.

- **Context coherence:** Are agent decisions informed by relevant findings and the thought process behind previous attempts?
- **Flexibility:** Can the agent take free-form actions (e.g., run code, inspect data, use tools) instead of a fixed-prompt format?
- **Persistence:** Can the search continue for long horizons without quality degrading due to context growth?

Each paradigm satisfies some, but not all, of these criteria (Tab. 1).

Evolution via code mutation. In this paradigm, an LLM mutates code candidates that are evaluated against a benchmark to produce a scalar score, which in turn guides selection for subsequent generations. FunSearch [40], AlphaEvolve [35], OpenEvolve [42], Evolution of Heuristics [23], and GEPA [3] follow this pattern, with proposals like ADSR [8] showing how systems problems can be tackled using frameworks such as OpenEvolve. These approaches query the LLM with a fixed prompt template built from prior candidates and scores. High-scoring codes survive to seed the next generation.

System optimization is a deliberative process requiring sustained reflection and the ability to learn from intermediate failures. Fixed

templates providing only snapshots of prior code and scores cannot encode a designer's evolving line of reasoning. For instance, a designer may conduct experiments on a suboptimal intermediate solution to gather diagnostic data; however, code evolution methods often prune such candidates because their immediate scores worsened, and subsequent LLM calls lack context regarding the underlying thought process behind such experiments.

Iterative design with reasoning and flexible tools. Inspired by coding agents such as Codex [37], Glia [14] takes an alternate path to help with flexibility and coherence. Each agent exploration[2] happens in a programming environment with tool access, exploring the design problem through a coherent sequence of actions (see Fig. 1). Unlike evolutionary methods, the agent can run experiments on a testbed or simulator, execute shell commands, analyze and reason over experimental data, and iteratively refine designs. A weakness of this approach is that context growth will eventually hit a limit or degrade the quality of agent performance in long-range thinking. Glia [14] proposes launching multiple independent agents sequentially or in parallel to mitigate this weakness. However, these agents do not share any knowledge among themselves, each agent often rediscovers prior insights, limiting long-range progress.

Comparing these two approaches, the evolutionary approach has persistence but has weak coherence and flexibility, whereas the second approach is flexible and coherent but not persistent (see Tab. 1). We see these limitations in our analysis of a cloud multicast scheduling problem in §4.1.

Engram bridges this gap with a shared Research Digest that preserves insights across explorations, achieving coherence, flexibility, and persistence. Fig. 1 illustrates the three approaches.

3 Engram Design

Engram combines two key ideas:

(1) explore using the scientific method to develop ideas (HYPOTHESIZE → IMPLEMENT → EXPERIMENT → ANALYZE → HYPOTHESIZE cycle) [14] and
(2) create compact, structured knowledge and transfer it to subsequent agents to achieve the long-horizon coherence needed for successful ideation.

The Engram structure (Fig. 2) is a sequence of LLM agents that each work on the heuristic design problem using reasoning methods (§3.1), with each agent *handing off* its findings to the next via the Research Digest (§3.2). The coherent exploration step aims to ensure that agents understand why their designs succeed or fail, rather than merely generating candidate solutions. The structured knowledge creation and transfer step ensures that this understanding persists beyond each agent's lifetime and is passed to its successor, enabling knowledge to accumulate without any single agent suffering from context degradation.

3.1 Single Agent Exploration

Each agent operates within a workspace containing task-specific artifacts provided by the user (e.g., documents, sample workloads, baseline implementations, etc.). The agent may create and modify files within this workspace while developing its solution. Additionally, the agent has access to a tool that accepts code files as input and runs it within an evaluation playground (e.g., a simulator or an experimental testbed), and returns the resulting outputs. The agent can run shell commands, read, write, and edit files, and search through the contents of its workspace (in the future, it can be extended to perform external research as well). The playground outputs include log files and performance metrics that the agent can analyze to compute statistics and diagnose performance issues. In addition, crucially, each agent can access the workspaces and Digests produced by previous agents (§3.2). Importantly, the research digest and Archive are stored as external artifacts rather than being embedded directly in the LLM's context window. Agents retrieve relevant portions of these artifacts on demand through tool calls, ensuring that prior history does not consume context tokens unless explicitly accessed.

Concretely, each agent follows a structured research agenda. Through the system prompt (Fig. 27), it is instructed to begin by reviewing the problem specification and any available prior work, then articulating a plan and a specific hypothesis describing what approach to try and why. The agent implements the idea in the working code file, runs a simulation, and performs a structured post-experiment analysis: comparing the new results to the prior ones, verifying that the implementation executed correctly, and identifying sources of improvement or limitations to guide further refinement. The HYPOTHESIZE → IMPLEMENT → EXPERIMENT → ANALYZE cycle continues until the agent decides that it is ready to conclude its exploration [14].

We implement this observe-before-changing discipline through the system prompt which mirrors the practice of a careful experimental researcher. Once finished, the agent appends a summary to the Research Digest that serves as guidance for subsequent agents, documenting the attempted approaches, key insights gained, recommended next steps, and strategies that proved ineffective. Fig. 26 shows an excerpt from a Research Digest produced during the multi-cloud multicast study, showing the structure of per-agent entries.

3.2 The Agent Handoff

On a handoff, Engram instantiates a new agent with an empty context window and provides it with the task description and access to the Research Digest and the Archive. The agent conducts the exploration process described in §3.1. At the end, the agent writes all the details into the Archive including code snapshots, logs, and results. It also writes a structured summary of findings to the persistent Research Digest. Fig. 3 depicts the workspace presented to a newly instantiated agent.

The handoff is a deliberate architectural choice that refreshes LLM context. A single long-running agent accumulates numerous tokens over the course of its exploration. As this context grows, the model's attention becomes increasingly uneven across earlier content, a well-documented limitation of current LLMs [10, 17, 28]. Handoff overcomes this problem without losing key information: when an agent has exhausted its productive capacity, it distills what

[2]An agent exploration is defined as the sequence of actions (e.g., read, write, and tool use) performed by an LLM-based agent from the beginning of its exploration until termination.

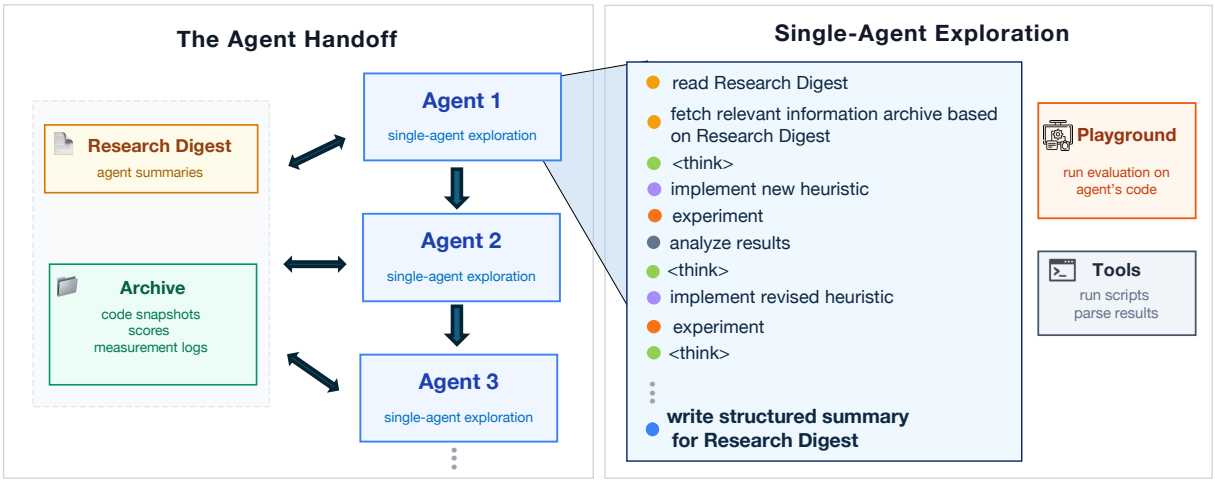


Figure 2: Engram's design is based on a sequence of reasoning-based agent explorations that produce and evaluate ideas based on hypotheses driven from experimental data analysis. Each agent begins by analyzing the problem and reviewing the Research Digest summarizing the findings of the previous agents, using that information to formulate its own exploration and experimentation plan. The agent executes this plan through design, experimentation, and analysis. Upon completion, it writes a summary of its findings to the research digest, stores all the details in the Archive, and hands off the research process to the next agent. The process typically terminates when the research budget is exhausted.

Figure 3: Workspace of a single agent in Engram.

it has learned into external artifacts, and a fresh agent inherits those artifacts with a clean context window.

Engram's design balances two competing strengths. A single, continuous context provides rich coherence and flexibility that prevents long-context degradation, while handoff introduces structured context management that achieves long-horizon persistence. Each successor agent begins with a compact, focused statement of the problem and prior findings rather than a long and deteriorating interaction history.

To mitigate error propagation, agents verify Research Digest summaries against raw Archive data. Past failures are informative rather than prescriptive, letting agents re-test weak conclusions independently. For example, in our multi-cloud multicast runs, a "Dijkstra with small first-hop penalty" direction marked ineffective by Agents 1 and 3 was independently re-tested by Agent 7 before being abandoned (Fig. 26). As harder benchmarks demand longer runs, the Research Digest grows; pruning and quality-control mechanisms are avenues for future work.

We have implemented Engram using the DEEPAGENTS[3] library built on LangChain and LangGraph [7].

4 Case Studies

We study Engram on three diverse system problems to show generality across optimization structure, codebases, and performance objectives. Along with the primary metric for each problem, we report the 90% bootstrapped confidence interval. We run each approach 10 times and each run has a budget of 100 evaluation runs. [4]

We compare Engram to four state-of-the-art frameworks that use LLMs for discovery: (i) Evolution of Heuristics (EoH) [23], (ii) FunSearch [40], (iii) OpenEvolve [42], and (iv) Glia [14]. Tab. 3 shows the parameters for these frameworks. In every comparison, Engram and the baselines use the same underlying LLM (o3 unless explicitly varied in §4.1), so observed differences are attributable to architecture rather than to model strength.

Evolution of Heuristics (EoH) [23] incorporates a "thought" phase into algorithm design and iteratively refines candidate heuristics using five operators: crossover operators that generate diversity and recombine ideas, and mutation operators that improve, tune parameters, or simplify heuristics.

[3]https://github.com/langchain-ai/deepagents
[4]These experiments were conducted in February 2026.

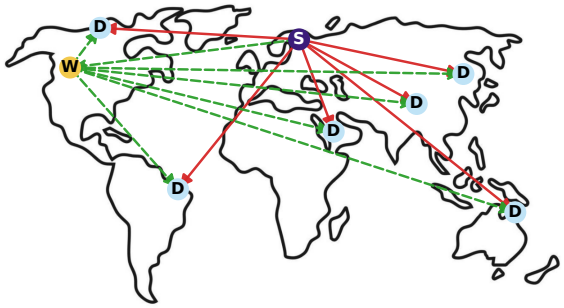


Figure 4: Multi-cloud data replication from a source (purple) to destinations (blue) across the world, either directly or via a waypoint (yellow) to avoid expensive or slow links.[5]

FunSearch [40] uses a best-shot optimization paradigm within an island model. Each generation refines top-performing programs selected from the solution database, maintaining a bounded candidate pool. We limit each island to 20 algorithms.

OpenEvolve [42], an open-source implementation of AlphaEvolve [35], adopts an island-based evolutionary framework. Beginning with a seed program and a task-specific scoring function, it uses a prompt sampler to generate LLM inputs that produce program variants. Evolutionary selection and migration across islands drive improvement over successive generations.

Glia [14] is an agentic framework that uses reasoning-driven experimentation to produce candidates: the agent forms hypotheses, runs targeted experiments, analyzes outcomes, and iterates on the candidate algorithm. In our evaluation, we use the multi-context variant of Glia that takes the best-of-N independent runs at test time ($N = 4$).

4.1 Case Study: Multi-Cloud Multicast

Problem overview. Modern cloud systems replicate large datasets across geographically distributed regions for geo-replicated storage, analytics, machine learning model distribution, and disaster recovery [12, 45, 51]. As shown in Fig. 4, data may be sent directly from a source to each destination or routed through intermediate waypoints. Waypoints can reduce end-to-end completion time and monetary cost by avoiding expensive egress links.

At first glance, this resembles classical multicast. The cloud setting, however, changes the design space [32]. Transfers are governed by asymmetric and policy-driven egress pricing rather than purely technical constraints [50]. Moreover, clouds permit elastic provisioning: one can create transient waypoints, which introduces a placement decision coupled with routing. Our goal is to design a delivery plan that minimizes the cost of sending the data under a time budget.

Human SOTA. Prior work [50] has formulated this problem as an optimization (see App. B). However, solving this optimization problem directly is impractical at even modest scale. For example, in the 71-node setting in our experiments, there are 2^{50410} possible configurations; a practical solution must therefore rely on heuristics and approximations [50]. Designing effective heuristics for such optimization problems is not trivial [50]; in fact, even formulating this problem as in App. B is a creative process. Cloudcast [50] is the human state-of-the-art solution to this problem. It uses domain-specific reductions to preserve the optimization structure while making it tractable.

Benchmark. We use the same benchmark as the Cloudcast in ADRS [9]. The benchmark uses a 71-node directed topology, with each edge annotated by egress price and measured throughput, and evaluates five broadcast configurations with fixed source/destination sets under provider-specific ingress/egress limits and per-region VM caps. We have implemented a verifier to check candidate heuristics for correctness and then score them by total cost (egress + VM cost), under a fixed time budget.

AI task formulation. Our goal is to evaluate whether an agent can *discover* effective algorithms for multi-cloud multicast. To understand how much guidance is needed, we provide the agent with three different task descriptions at different levels of detail:

- **Minimal prompt:** a generic "write an algorithm" instruction that provides no strategic hints (see Fig. 23).
- **Prompt with high-level direction ("Direction"):** a prompt that gives a high-level strategy: formulate the problem as an optimization and then use approximations to make it tractable (see Fig. 24).
- **Prompt with detailed optimization formulation:** the "Direction" prompt augmented with the full mathematical formulation from App. B.

The results will tell us whether performance depends on the creative step of formulating the optimization or the downstream step of engineering tractable approximations.

Take-away results. Across prompts and models, the strongest performance comes from giving *high-level direction* (Fig. 5): the "Direction" prompt consistently yields the lowest costs, while adding the full optimization from App. B produces no measurable improvement. This suggests that *strategic guidance* ("treat this as optimization, then approximate") matters more than providing the full mathematical formulation, and Engram persists within an optimization family until it becomes tractable and succeeds. When removing the direction, evolutionary approaches and Glia remain trapped in Steiner-tree–style [6] heuristics, which typically employ greedy or distance-network approximations. In contrast, Engram discovers solver-backed designs that use Dynamic Programming (DP) and Mixed Integer Linear Programming (MILP), which are strong alternatives to the human SOTA. Stronger reasoning models (e.g., GPT-5.2) find these solutions more reliably.

Performance comparison. Each agent run produces a best-performing candidate. We call its cost the "best cost" for that agent run. We run each agent using the "Direction" prompt and the o3 LLM 10 times. Fig. 5 shows the average and 90% confidence interval of these 10 best costs. Engram outperforms all the other approaches (with average best cost of $664) and provides solutions closest to human SOTA. It also achieves the strongest LLM-generated result

[5]Image reproduced by the authors inspired from Cloudcast [50].

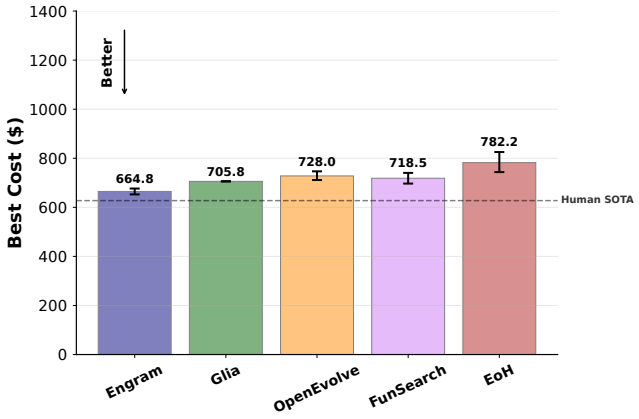


Figure 5: Comparison for multi-cloud multicast with "Direction" prompt and o3 (lower is better). Engram achieves the strongest average best cost, outperforming both evolutionary approaches (EoH, FunSearch, OpenEvolve) and Glia. The whiskers show 90% confidence intervals.

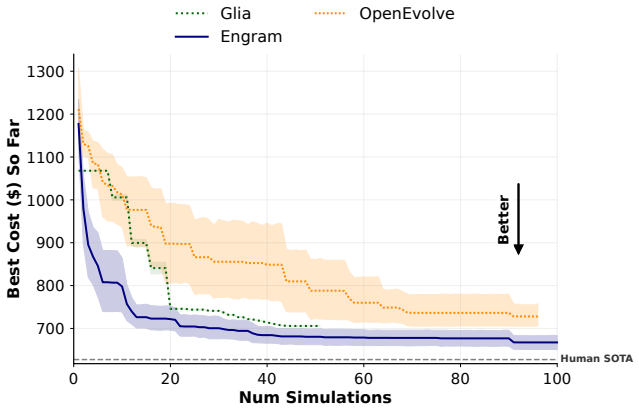


Figure 6: Best cost-so-far versus number of simulations on multi-cloud multicast. OpenEvolve plateaus on simple heuristics; Glia makes progress but eventually exhausts its context/budget; Engram continues improving and finds solver-based solutions. The shades are 90% confidence intervals of best costs so far across runs.

(Fig. 20) across all the methods and runs, with the best cost reaching $625 in one agent run, slightly improving human SOTA [50] ($626). Engram's best solution uses a tractable Mixed Integer Linear Program (MILP) in the same style as the human SOTA.

The best EoH/OpenEvolve/FunSearch solutions implement a graph algorithm that builds an approximate Steiner tree, with no explicit optimization attempt, ranging in cost between $640 and $696. The best Glia solution is also a graph algorithm without explicit optimization with a cost of $687, but is better than the other solutions because it comes up with a provider-aware/shared-tree construction (see Fig. 21).

Progress results and discussion. Fig. 6 (progress curves) shows the best-cost-so-far vs. simulation budget for the "Direction" prompt and o3 model. OpenEvolve initially improves but then plateaus well above Human SOTA. Inspecting the generated heuristics, we find a clear structural pattern: most OpenEvolve [42] solutions remain variants of Steiner-tree-style backbones with minor local tweaks. This is reminiscent of the neighborhood bias issue with evolutionary approaches discussed in §1. OpenEvolve does not progress toward optimization-derived designs. A comment block in one top solution even emphasizes this point explicitly: *"The whole routine is heuristically efficient—no MILP solver invocation—yet it typically cuts total egress cost by* $>$ 30%*."* (see Fig. 17).

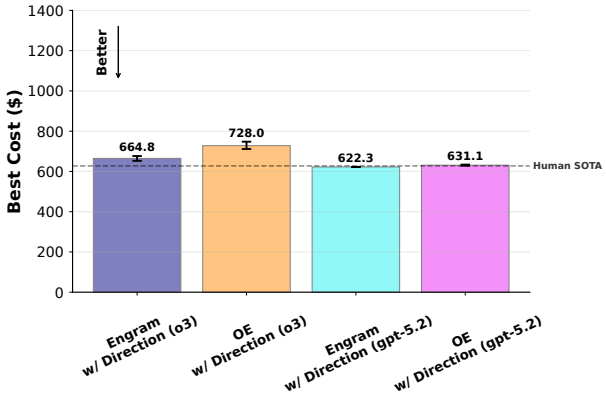
Glia exhibits a different trajectory in Fig. 6. It shows a sharp improvement initially, but progress eventually stalls despite an available simulation budget due to context limits (Fig. 6). Progress stalls because of the coherence ceiling discussed in §1.

Prompt and model sensitivity. Fig. 7a shows the average best costs with "Direction" prompt for gpt-5.2 as well. For both models, Engram rapidly shifts into explicit optimization. OpenEvolve only formulates an optimization in gpt-5.2; with o3, it stays in Steiner-tree-based heuristics. We did not observe any measurable behavior change when we additionally provided the full optimization (App. B); performance and qualitative solution structure remained essentially unchanged.

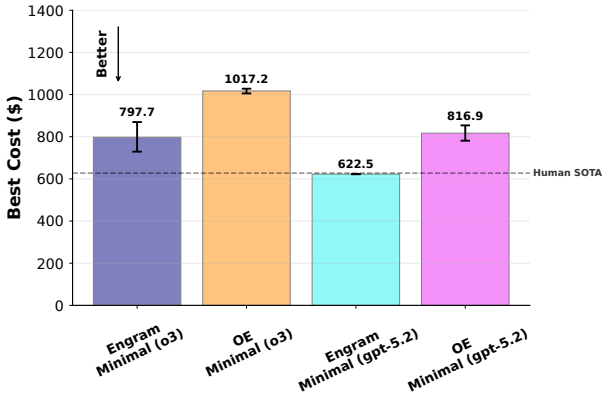
Fig. 7b shows the results using the minimal "write an algorithm" prompt for both o3 and gpt-5.2. OpenEvolve cannot produce good solutions with this prompt, even with gpt-5.2. However, Engram with o3 on average reaches better solutions than OpenEvolve, even when OpenEvolve uses gpt-5.2. Surprisingly, in one run, Engram generated a heuristic that uses MILP even with the minimal prompt and beats the human SOTA ($623 vs. $626 for the cost of multicast) (see Fig. 18). Moreover, with the stronger model (gpt-5.2), Engram introduces a fundamentally different and novel algorithmic approach using dynamic programming (DP) that consistently beats the human SOTA ($622 vs. $626) (see Fig. 19).

Persistence despite costs sometimes worsening in Engram. A concrete example shows up in the o3+"Direction" run (Fig. 8), where Engram stays on an optimization approach even though performance regresses before it gets better. **(1)** Starting from a strong baseline (cost $772), the next agent tries to implement the body of an optimization **(2)** and the cost *"EXPLODED to $1104"*; **(3)** a follow-up attempt fails and finds *"NO opportunities"* to improve the cost. Instead of abandoning the optimization direction and reverting to primitive heuristics, the next agent makes the right call and continues **(4)**. It implements a reduced-edge MILP with explicit tractability knobs. This recovers the heuristic and produces the key jump: *"Reduced-edge MILP v2 ... Achieved total cost* $644 *(*$-17\%$ *vs previous $772 baseline)"* with a cost decrease from 772 $\rightarrow$ 644. Engram tolerates temporary score degradation, uses the failure to diagnose what is missing, and then continues refining within the same algorithmic family until the optimization becomes tractable and wins.

Summary. Multi-cloud multicast exposes a regime where the design space is combinatorially large and effective solutions require *optimization structure* rather than local heuristic tweaks. Giving the model high-level direction ("formulate an optimization, then approximate") is sufficient to unlock strong performance; providing



(a) Method comparison with direction prompt.



(b) Method comparison with simple minimal prompt.

Figure 7: Average and 90% confidence interval of best-cost 10 runs for multi-cloud multicast (OE: OpenEvolve, lower is better). Engram outperforms OpenEvolve in all settings.

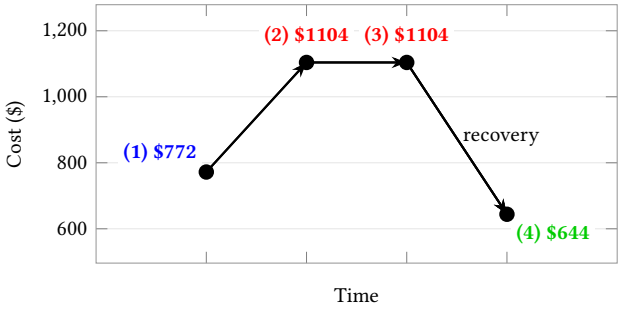


Figure 8: Engram tolerates non-monotonic intermediate outcomes while staying within an optimization-based approach. Early attempts temporarily degrade cost (772 → 1104) before Engram recovers and achieves a large improvement (1104 → 644).

the full mathematical formulation does not materially change outcomes. Engram is uniquely effective at escaping Steiner-tree–style neighborhood bias, persisting through non-monotonic intermediate

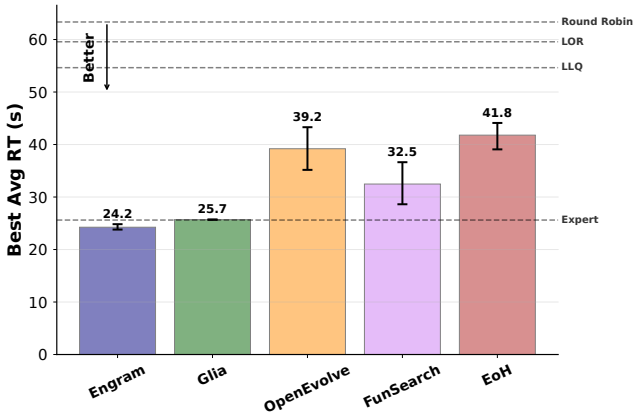


Figure 9: Best-response time comparison on the LLM request routing problem (lower is better). Engram achieves the strongest average best response time, outperforming evolutionary approaches (EoH, FunSearch, OpenEvolve), Glia, and Expert. The whiskers show 90% confidence intervals.

results, and ultimately producing solver-backed designs (MILP/DP) that match and in some cases surpass Human SOTA.

4.2 Case Study: LLM Request Routing

Problem overview. We study the problem of routing requests among replicated LLM instances in distributed serving systems. We evaluate how Engram identifies novel routing strategies that lower overall mean request completion time (RT). We evaluate using `vidur`, a simulator for distributed LLM serving systems [1]. The main metric is mean request completion time (RT), i.e., end-to-end response latency. RT reflects end-user's perceived quality of responsiveness.

Benchmark. We use the same setting as Glia [14]. We simulate a ShareGPT-based [41] LLM inference workload on four NVIDIA A10 GPUs serving Llama-3-8B-Instruct. To mimic reasoning-heavy workloads with heavy-tailed sequence lengths, we independently increase 5% of prompt lengths and 5% of decode lengths by 10×. The system operates at 7.5 queries per second (QPS), with bursty arrivals drawn from a log-normal distribution ($\sigma = 2$). Each replica uses chunked prefill [2] with a chunk size of 8192 for batch scheduling. All experiments are repeated across ten random seeds.

Baseline routing comparisons. We evaluate Engram against three standard routing heuristics: Round-Robin, Least-Loaded Queue (LLQ), and Least Outstanding Requests (LOR). Round-Robin cycles requests across replicas; LLQ selects the replica with the fewest active requests; and LOR chooses the replica with the fewest queued requests awaiting GPU allocation. We also compare against a workload-specialized heuristic designed over two weeks by a senior systems researcher with over two decades of experience [14].

Quantitative and qualitative results. Fig. 9 shows that Engram achieves the best aggregate performance among the LLM-based methods, with the lowest average best RT (lower is better). In particular, Engram outperforms evolutionary code-search methods (EoH, FunSearch, OpenEvolve) as well as the Glia variants, and outperforms the expert-designed routing heuristic. Engram is not

only capable of finding strong novel routing policies, but does so consistently across runs (exhibiting low performance variations).

Qualitatively, the strongest policy found by Engram is not a simple queue-length heuristic (like LLQ/LOR) (see Fig. 22), but a structured routing strategy that combines **Shortest-Prefill-First (SPF) request ordering**, **decode-sensitive per-replica pending-queue caps**, and a **memory-utilization admission guard** that avoids placements likely to trigger evictions. It then selects among admissible replicas using **projected memory utilization**, which directly targets the prefill/decode interference pattern induced by chunked prefill and heavy-tailed requests. In other words, Engram discovers a policy that explicitly reasons about the interaction between memory pressure and decode-phase contention, rather than optimizing only coarse load counts. Comparing best solutions of Glia and Engram, Glia uses SPF and a KV-cache headroom–based memory admission rule [14], but Engram goes further by adding decode-sensitive pending-queue caps and decode-aware tie-breaking, making its routing policy more explicitly responsive to decode-phase contention rather than primarily enforcing memory safety through reserved free blocks.

By comparison, the best evolutionary solution we observed remains closer to a generic shortest job first heuristic: it also uses prefill-aware ordering and predicted work estimates, but primarily scores replicas using a token-based outstanding-work proxy with a fixed memory guard.

4.3 Case Study: Optimizing KV Cache Reuse in Databases with Natural Language Queries

This problem addresses the cost of batch LLM inference over relational tables [29]. When an LLM processes rows sequentially, consecutive rows that share a long serialized prefix can reuse the key–value (KV) cache, thereby reducing inference cost. Liu et al. [29] propose a SOTA method for reordering a dataframe to maximize prefix-based KV-cache reuse. Their key insight is that allowing per-row dynamic field ordering, rather than enforcing a fixed column order across all rows, can substantially improve cache hit rates. This observation motivated the design of their algorithm, Greedy Group Recursion (GGR). Note that a brute-force search over all possible reorderings is computationally infeasible. For a table with m rows and n columns, the total number of possible reorderings is prohibitively large, $n! \times (m!)^n$ [29].

We adopt the evaluation environment of ADRS [8], running each algorithm on five datasets (MOVIES, BEER, BIRD, PDMX, and PRODUCTS) and reporting the combined score of $0.95\times\bar{h}+0.05\times r$, where $\bar{h}$ is the mean per-dataset prefix hit rate and r is a normalized runtime bonus capped at 12 seconds. Since we observed that pre-merging columns with functional dependencies benefits all methods, we apply this merge before reordering for all algorithms at evaluation time rather than leaving it to each algorithm.

We seed all the methods with a simple baseline program that sorts columns by cardinality and orders rows lexicographically. We use a task prompt (Fig. 25), describing the full problem and indicating the benefits of having different column-ordering per-row. Under this setup, all methods perform competitively. Engram attains the highest mean best score (0.721), followed closely by Glia (0.719) and OpenEvolve (0.714) (see Fig. 10). Engram is the most efficient method, requiring fewer simulations to achieve any given score.

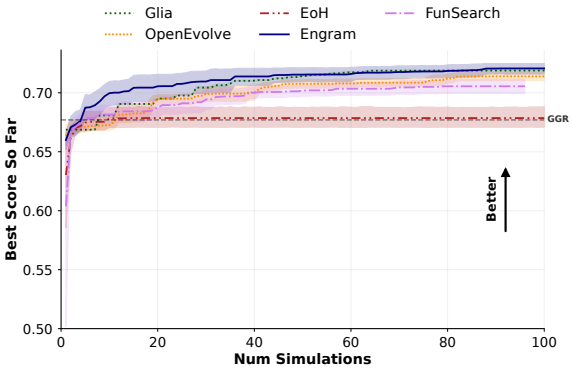


Figure 10: Average best score so far vs. number of simulations for all methods seeded with a simple baseline program for the LLM-SQL task. Engram, Glia, and OpenEvolve converge to comparable final scores with Engram converging faster, while FunSearch and EoH lag behind. The GGR [29] line shows the score of the state-of-the-art algorithm.

Interestingly, the best programs produced by all methods did not use GGR's recursive structure, yet still achieved high scores. These high-scoring non-recursive methods all leveraged per-row column reordering to improve cache reuse.

For this case study, we additionally evaluate Engram and OpenEvolve when initialized with the GGR algorithm instead of the simple baseline. Experimental details are provided in App. C.

5 Additional Evaluation

We evaluate Engram and OpenEvolve [42] on a larger set of ADRS problems [8, 47] and compare to the reported Human SOTA. For cost reasons, we use OpenAI o3 [36]. Each method runs 10 times; we report average best scores with 90% confidence intervals (Tab. 2). Engram exceeds Human SOTA on five of six tasks and improves over OpenEvolve on four. In §D.1, we show that Engram is faster than OpenEvolve in reaching its peak performance.

Ablation experiments. We ablate the main components of Engram on the multi-cloud data transfer task (Fig. 11) to isolate the effects (i) coherence ceiling, and (ii) persistent cross-agent knowledge transfer.

The **Single Agent** is an agent with full tool and execution access but no other structure. Single Agent performs worst (average cost $902); it exhausts its effective reasoning budget and runs fewer meaningful experiments, and converges to weaker heuristics. **Summarization** improves upon the single agent (average cost $765) and tries to enable a longer context by compressing older context and allowing additional iterations. However, its long-horizon coherence remains limited and confined to a single summarized context and still underperforms all multi-agent variants. The pattern holds with a stronger model: single-agent gpt-5.2 with tool/simulator access still plateaus short of human SOTA (§D.3).

In **Sequential** variant, the best code from one agent is passed to the next without any structured context sharing (no Research Digest and no Archive). This isolates the benefit of simply resetting

Strategy	CBL ↓	CBL-Multi ↓	EPLB ↑	Prism ↑	Telemetry ↑	TXN ↑
Human SOTA	101.7	92.3	0.251	21.89	0.822	2724.8
Engram	103.6 ± 1.1	**79.9 ± 0.8**	**0.273 ± 0.00**	**27.94 ± 1.70**	**0.954 ± 0.00**	**3918.6 ± 56.6**
OpenEvolve [42]	103.4 ± 0.9	79.9 ± 0.4	0.214 ± 0.06	26.21 ± 0.03	0.953 ± 0.00	3713.7 ± 77.9

Table 2: Average best scores across 10 runs with 90% confidence intervals (higher is better ↑; lower is better ↓). Engram exceeds Human SOTA on five of six tasks and improves over OpenEvolve on four.

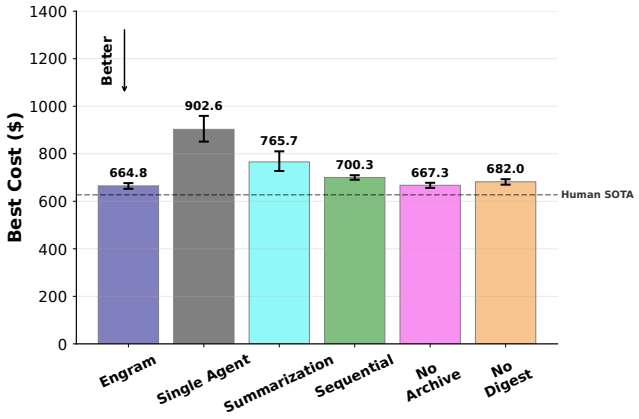


Figure 11: Ablation on multi-cloud multicast (average best cost; lower is better). Single-agent variants (Single Agent and Summarization) are constrained by context growth and perform worst, while multi-agent variants improve on them. Engram outperforms Sequential code transfer, highlighting the benefit of persistent knowledge. Removing either the Archive or Digest slightly worsens performance.

context across agents. Sequential improves performance (average cost $700) relative to single-agent baselines, showing that fresh contexts help. However, it still falls short of full variations with persistent cross-agent knowledge transfer.

The **No Archive** and **No Digest** variants each remove one component of Engram, leading to modest but consistent performance degradation compared to the full system. Removing the Digest has a larger negative impact, suggesting that Digest plays a more critical role in guiding future agents than raw information in Archive. Our intuition is that a successor agent can rapidly orient itself via the Research Digest, whereas reconstructing reasoning from raw Archive logs wastes context and risks redundant rediscovery. The departing agent is uniquely positioned to distill its findings, so structured interpretation matters more than raw artifact storage. The Archive serves as a fallback verification layer, letting agents inspect raw artifacts directly when they doubt a prior claim.

Cost and exploration efficiency. We normalize by evaluator (simulator/testbed) runs rather than LLM tokens because real-world systems optimization is bottlenecked by evaluation latency, not LLM inference, prioritizing "exploration quality" per evaluation. Average LLM costs per run on multi-cloud multicast (o3) are thus higher for our agentic methods: ~$30 for Engram and Glia, versus $6.67 for OpenEvolve, $5.42 for FunSearch, and $1 for EoH. Engram's cost stems from multi-turn reasoning between evaluator calls (analyzing results and causal relationships), unlike evolutionary methods' single LLM call per candidate. This overhead lets Engram escape local optima and discover qualitatively different solution families (e.g., MILP and DP designs, §4.1). LLM calls scale linearly with agent count; tool calls add no invocations.

6 Conclusion

This paper introduced Engram, an agentic LLM researcher architecture designed to improve coherence in long-horizon system design tasks. We identified two core limitations of prior LLM-based approaches: evolutionary neighborhood bias and the coherence ceiling. Engram addresses these limitations by decoupling long-horizon exploration from the constraints of a single context window. It organizes exploration into a sequence of agents that iteratively design, test, and analyze mechanisms. At the conclusion of each run, an agent stores code snapshots, logs, and results in a persistent *Archive* and distills high-level modeling insights into a compact, persistent *Research Digest*. Subsequent agents then begin with a fresh context window, reading the Research Digest to build on prior discoveries.

Across three diverse case studies, Engram consistently outperformed evolutionary and iterative agentic baselines. Beyond performance, Engram navigates conceptual boundaries more effectively, tolerates temporary regressions, persists within promising algorithmic families, and produces innovative, principled designs.

Acknowledgments

This work was funded by the MIT Generative AI Impact Consortium (MGAIC), NSF Award 2504568, and Quanta Computer, Inc. under the AIR Project. We thank Diego de Lope and Avi Kapur Srinivasan for their insights.

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A large-scale simulation framework for LLM inference. *Proceedings of Machine Learning and Systems* 6 (2024), 351–366.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *OSDI*. Article 7, 18 pages.

[3] 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. 2026. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=RQm2KQTM5r

[4] Martin Andrews and Sam Witteveen. 2025. GPU Kernel Scientist: An LLM-Driven Framework for Iterative Kernel Optimization. *arXiv preprint arXiv:2506.20807* (2025).

[5] Eser Aygün, Anastasiya Belyaeva, Gheorghe Comanici, Marc Coram, Hao Cui, Jake Garrison, Renee Johnston Anton Kast, Cory Y McLean, Peter Norgaard, Zahra Shamsi, et al. 2025. An AI system to help scientists write expert-level empirical software. *arXiv preprint arXiv:2509.06503* (2025).

[6] Moses Charikar, Chandra Chekuri, To-Yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. 1999. Approximation algorithms for directed Steiner problems. *Journal of Algorithms* 33, 1 (1999), 73–91.

[7] Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain

[8] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Shubham Agarwal, Mert Cemri, Bowen Wang, Alexander Krentsel, Tian Xia, and Jongseok Park. 2025. Let the Barbarians In: How AI Can Accelerate Systems Performance Research. *arXiv preprint arXiv:2512.14806* (2025).

[9] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. arXiv:2510.06189 [cs.AI] https://arxiv.org/abs/2510.06189

[10] Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Babu Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. 2025. Context Length Alone Hurts LLM Performance Despite Perfect Retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 23281–23298. doi:10.18653/v1/2025.findings-emnlp.1264

[11] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators! *arXiv preprint arXiv:2510.08803* (2025).

[12] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. 2022. Owl: Scale and Flexibility in Distribution of Hot Content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 1–15. https://www.usenix.org/conference/osdi22/presentation/flinn

[13] Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. 2025. Towards an AI co-scientist. *arXiv preprint arXiv:2502.18864* (2025).

[14] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. arXiv:2510.27176 [cs.AI] https://arxiv.org/abs/2510.27176

[15] Zhiyuan He, Aashish Gottipati, Lili Qiu, Xufang Luo, Kenuo Xu, Yuqing Yang, and Francis Y. Yan. 2024. Designing Network Algorithms via Large Language Models. In *HotNets*. Association for Computing Machinery, New York, NY, USA, 205–212. doi:10.1145/3696348.3696868

[16] Zhiyuan He, Aashish Gottipati, Lili Qiu, Yuqing Yang, and Francis Y. Yan. 2025. Congestion Control System Optimization with Large Language Models. arXiv:2508.16074 [cs.NI] https://arxiv.org/abs/2508.16074

[17] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. *Context Rot: How Increasing Input Tokens Impacts LLM Performance*. Technical Report. Chroma. https://research.trychroma.com/context-rot

[18] Ziyao Huang, Weiwei Wu, Kui Wu, Jianping Wang, and Wei-Bin Lee. 2025. Calm: Co-evolution of algorithms and language model for automatic heuristic design. *arXiv preprint arXiv:2505.12285* (2025).

[19] Pantea Karimi, Siva Kesava Reddy Kakarla, Pooria Namyar, Santiago Segarra, Ryan Beckett, Mohammad Alizadeh, and Behnaz Arzani. 2026. Heuristic Analysis from Source Code via Symbolic-Guided Optimization. In *NSDI*. USENIX Association.

[20] Pantea Karimi, Solal Pirelli, Siva Kesava Reddy Kakarla, Ryan Beckett, Santiago Segarra, Beibin Li, Pooria Namyar, and Behnaz Arzani. 2024. Towards Safer Heuristics With XPlain. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. 68–76.

[21] Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. 2025. Robust Heuristic Algorithm Design with LLMs. arXiv:2510.08755 [cs.AI] https://arxiv.org/abs/2510.08755

[22] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution. *arXiv preprint arXiv:2509.19349* (2025).

[23] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *ICML* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1304, 23 pages.

[24] Fei Liu, Qingfu Zhang, Jialong Shi, Xialiang Tong, Kun Mao, and Mingxuan Yuan. 2025. Fitness landscape of large language model-assisted automated algorithm search. *arXiv preprint arXiv:2504.19636* (2025).

[25] Fei Liu, Rui Zhang, Xi Lin, Zhichao Lu, and Qingfu Zhang. 2025. Fine-tuning large language model for automated algorithm design. *arXiv preprint arXiv:2507.10614* (2025).

[26] Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Llm4ad: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287* (2024).

[27] Gang Liu, Yihan Zhu, Jie Chen, and Meng Jiang. 2025. Scientific Algorithm Discovery by Augmenting AlphaEvolve with Deep Research. *arXiv preprint arXiv:2510.06056* (2025).

[28] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[29] Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E. Gonzalez, and Matei Zaharia. 2025. Optimizing LLM Queries in Relational Data Analytics Workloads. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=R7bK9yycHp

[30] Yixiu Liu, Yang Nan, Weixian Xu, Xiangkun Hu, Lyumanshan Ye, Zhen Qin, and Pengfei Liu. 2025. Alphago moment for model architecture discovery. *arXiv preprint arXiv:2507.18074* (2025).

[31] Ruiying Ma, Chieh-Jan Mike Liang, Yanjie Gao, and Francis Y Yan. 2025. Algorithm Generation via Creative Ideation. *arXiv preprint arXiv:2510.03851* (2025).

[32] A Darwin Marks, P Karthikeyan, and R Sivaraman. 2015. High-Bandwidth Multicast in Cooperative Environments. *Compusoft* 4, 4 (2015), 1680.

[33] Microsoft. 2025. base_global_scheduler.py — Vidur Global Scheduler Base Module. https://github.com/microsoft/vidur/blob/main/vidur/scheduler/global_scheduler/base_global_scheduler.py.

[34] Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. 2025. Reinforced Generation of Combinatorial Structures: Applications to Complexity Theory. *arXiv preprint arXiv:2509.18057* (2025).

[35] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. *URL: https://arxiv. org/abs/2506.13131* (2025).

[36] OpenAI. 2025. *OpenAI o3 and o4-mini System Card*. Technical Report. OpenAI. https://openai.com/index/o3-o4-mini-system-card/

[37] OpenAI. 2026. Codex. https://chatgpt.com/codex. AI coding agent product from OpenAI.

[38] Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. 2023. MemGPT: towards LLMs as operating systems. (2023).

[39] Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, and Jisun Park. 2025. AlgoTune: Can Language Models Speed Up General-Purpose Numerical Programs? *arXiv preprint arXiv:2507.15887* (2025).

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

[41] sharegpt 2025. ShareGPT Datasets at Hugging Face. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered. [Accessed 10-10-2025].

[42] Asankhaya Sharma. 2025. OpenEvolve: an open-source evolutionary coding agent. https://github.com/codelion/openevolve

[43] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=vAElhFcKW6

[44] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2025. Automated High-Level Code Optimization for Warehouse Performance. *IEEE Micro* (2025).

[45] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. 2022. Ekko: A Large-Scale Deep Learning Recommender System with Low-Latency Model Update. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 821–839.

[46] Yiwen Sun, Furong Ye, Zhihan Chen, Ke Wei, and Shaowei Cai. 2025. Automatically discovering heuristics in a complex SAT solver with large language models. *arXiv preprint arXiv:2507.22876* (2025).

[47] UCB-ADRS. 2026. UCB-ADRS/ADRS: AI-Driven Research Systems (ADRS). https://github.com/UCB-ADRS/ADRS Accessed: 2026-02-26.

[48] Anjiang Wei, Allen Nie, Thiago SFX Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. 2024. Improving Parallel Program Performance with LLM Optimizers via Agent-System Interfaces. *arXiv preprint arXiv:2410.15625* (2024).

[49] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506* (2025).

[50] Sarah Wooders, Shu Liu, Paras Jain, Xiangxi Mo, Joseph E. Gonzalez, Vincent Liu, and Ion Stoica. 2024. Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud. In *NSDI*.

[51] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 292–308.

[52] Shijie Xia, Yuhan Sun, and Pengfei Liu. 2025. SR-Scientist: Scientific Equation Discovery With Agentic AI. *arXiv preprint arXiv:2510.11661* (2025).

[53] Qiujie Xie, Yixuan Weng, Minjun Zhu, Fuchen Shen, Shulin Huang, Zhen Lin, Jiahui Zhou, Zilan Mao, Zijie Yang, Linyi Yang, et al. 2025. How Far Are AI Scientists from Changing the World? *arXiv preprint arXiv:2507.23276* (2025).

[54] Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. 2025. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 27144–27152.

[55] Yi Zhai, Zhiqiang Wei, Ruohan Li, Keyu Pan, Shuo Liu, Lu Zhang, Jianmin Ji, Wuyang Zhang, Yu Zhang, and Yanyong Zhang. 2025. \(X\)-evolve: Solution space evolution powered by large language models. *arXiv preprint arXiv:2508.07932* (2025).

[56] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19632–19642.

[57] Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. 2025. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603* (2025).

System	Parameter Name	Value
EoH	Initial population size (N)	20
FunSearch	# Prompt programs (k)	3
OpenEvolve (Vidur)	Number of islands	6
	Exploration probability	0.6
	Exploitation probability	0.4
	Initial program	LLQ router
OpenEvolve (Others)	Configuration	Benchmark settings [9]
Glia	# Parallel threads	4

Table 3: Experimental settings and hyperparameters.

A Other Related Work

LLMs are increasingly used for algorithm discovery via *reasoning-and-search* loops rather than one-shot code generation [26, 53]. Prior systems span (i) multi-agent iterative refinement [13], (ii) evolutionary and tree-search methods that optimize populations of programs using fitness-based selection (e.g., EoH, ShinkaEvolve, AlphaEvolve, MCTS, LAS, X-evolve) [22–24, 35, 55, 57], including recent multi-objective variants [54], and (iii) approaches that incorporate learning signals such as supervised/fine-tuning on curated traces or datasets [18, 25]. Recent agentic tree-search pipelines and extensions that ground proposals via web research have also been explored [5, 27]. Tool-augmented long-horizon discovery has been studied in symbolic regression as well (e.g., SR-Scientist) [52].

LLMs for systems research. A growing body of work is emerging to automate systems research with LLMs. At a high level, these efforts span broad systems domains (e.g., ADRS) [9] as well as targeted optimization of performance-critical artifacts such as C++ code [44] and numerical kernels (e.g., AlgoTune) [39]. Several works formalize this interaction through specialized interfaces and feedback interpreters: Wei et al. [48] propose an Agent-System Interface (ASI) built around a compact DSL plus an AutoGuide feedback layer to optimize mapper code for parallel programs, while related ASI-style closed loops have been extended to other design spaces such as neural architectures (ASI-ARCH) [30]. Glia [14] replaces direct code mutation of prior work with reasoning-based exploration following the scientific method.

LLM-driven search has also been applied to discovering or tuning algorithms in diverse problem settings, including SAT solving [46], adaptive bitrate streaming [15], and networking control policies such as congestion control and caching [11, 16]. In performance engineering for accelerators, Astra [49] and GPU Kernel Scientist [4] study LLM-guided optimization of CUDA kernels. Beyond systems, similar search-and-improve templates appear in theoretical and combinatorial discovery, e.g., using AlphaEvolve to find new constructions in complexity theory [34].

Multiple works emphasize structured exploration to improve both quality and novelty. Robusta [21] combines combinatorial reasoning from prior work [19, 20] with LLM-guided evolutionary search to obtain networking heuristics with stronger worst-case guarantees, while MetaMuse [31] steers generation using external stimuli, waypoint reasoning, and feedback-derived performance embeddings to promote diverse, high-performing algorithms (e.g., cache replacement and bin packing).

Memory architectures for LLM agents. A separate line of work studies general-purpose memory for LLM agents. ExpeL [56] extracts experiential insights from prior task trajectories and reuses them at inference time across tasks; Reflexion [43] stores verbal self-reflections to improve subsequent attempts; MemGPT [38] manages a hierarchical memory that pages information in and out of the LLM context window. Engram is complementary to these systems rather than a competitor. The Research Digest is not a general-purpose memory architecture but a structured handoff specialized for cumulative scientific exploration in heuristic design. Each Digest entry is shaped around the hypothesize–implement–experiment–analyze cycle and records hypotheses, experimental outcomes, failure diagnoses, and recommended next steps, mirroring how a researcher would hand off a project to a colleague. Notably, our ablation (Fig. 11) shows that in-place summarization within a single agent context substantially underperforms a fresh-agent handoff with a Research Digest, suggesting that starting fresh with structured knowledge transfer is more effective than compressing and maintaining a growing context. Engram's design could in principle be combined with a general memory layer for cross-project recall, which we leave for future work.

B Formulation for Multi-Cloud Multicast

We provide the classical mixed-integer optimization formulation underlying the multi-cloud multicast. The goal is to design a multicast delivery plan that jointly optimizes monetary cost and feasibility under a completion-time budget, by deciding: (i) where to place relay capacity (VMs) in regions, (ii) how to route traffic over heterogeneous inter-cloud links, and (iii) how to respect per-VM ingress/egress limits.

Topology and inputs. Let $G = (V, E)$ be a directed graph of cloud regions V and directed transfer links E. A transfer instance specifies a source region $s \in V$, a destination set $D \subseteq V$, a total transfer size B (GB), a completion-time budget T (seconds), and a number of *stripes* (partitions) $\mathsf{K} \in \mathbb{Z}_{\geq 1}$. Each stripe has volume $\mathsf{b} = \mathsf{B}/\mathsf{K}$.

Each directed edge $(u, v) \in E$ has: (i) an egress price $c_{u,v}$ (\$/GB), and (ii) a profiled per-VM throughput $\beta_{u,v}$ (GB/s) achievable from u to v. Each region v has a per-VM egress limit e_v (GB/s), a per-VM ingress limit i_v (GB/s), a per-VM instance price p_v (\$/s), and a maximum VM count ℓ_v.

Decision variables (relay placement and routing). Cloudcast uses three sets of variables:

- **Stripe routing / tree selection:** $P_{k,(u,v)} \in \{0, 1\}$ indicates whether stripe $k \in \{1, \ldots, \mathsf{K}\}$ is sent over edge $(u, v) \in E$. The selected edges form a multicast distribution structure for each stripe.
- **Relay/VM placement:** $N_v \in \mathbb{Z}_{\geq 0}$ is the number of VMs (overlay routers) provisioned in region v. This captures *relay placement and capacity*: regions with $N_v > 0$ can originate/forward traffic subject to limits.
- **Auxiliary connectivity/acyclicity flow:** $F_{k,(u,v)} \in \mathbb{R}_{\geq 0}$ is an auxiliary flow used only to enforce that, for each stripe, the selected edges are connected and cycle-free and allow flow from the source to all destinations.

Objective (instance cost + egress cost). The objective minimizes the sum of (i) VM instance cost over the time budget and (ii) egress

cost for the data volume shipped along selected edges:

$$\min_{P,N,F} \quad \mathsf{T}\sum_{v\in V} p_v N_v \;+\; \frac{\mathsf{B}}{\mathsf{K}}\sum_{k=1}^{\mathsf{K}}\sum_{(u,v)\in E} c_{u,v}\,P_{k,(u,v)}. \tag{1}$$

VM count limits (relay placement constraints).

$$\forall v \in V \;:\quad N_v \le \ell_v. \tag{2}$$

Completion-time feasibility via volume capacities. To enforce completion within T while keeping the program linear, Cloudcast expresses bandwidth constraints as *volume capacities* over the time horizon and allocates volume in stripe units.

(i) Edge capacity constraints. The total stripe volume placed on edge (u, v) must fit in the volume that N_u VMs at u can transmit in time T:

$$\forall (u,v) \in E \;:\quad \frac{\mathsf{B}}{\mathsf{K}}\sum_{k=1}^{\mathsf{K}} P_{k,(u,v)} \;\le\; \mathsf{T}\,N_u\,\beta_{u,v}. \tag{3}$$

(ii) Per-region egress constraints. A region's aggregate outgoing stripe volume is limited by its per-VM egress cap:

$$\forall v \in V \;:\quad \frac{\mathsf{B}}{\mathsf{K}}\sum_{k=1}^{\mathsf{K}}\sum_{(v,u)\in E} P_{k,(v,u)} \;\le\; \mathsf{T}\,N_v\,e_v. \tag{4}$$

(iii) Per-region ingress constraints. Similarly, aggregate incoming stripe volume is limited by per-VM ingress:

$$\forall v \in V \;:\quad \frac{\mathsf{B}}{\mathsf{K}}\sum_{k=1}^{\mathsf{K}}\sum_{(u,v)\in E} P_{k,(u,v)} \;\le\; \mathsf{T}\,N_v\,i_v. \tag{5}$$

Ensuring a valid multicast structure (connectivity and acyclicity). Following Cloudcast, we augment the graph with a special sink node t that is connected only from destinations (via virtual edges (d, t) for $d \in D$; these incur no cost and are used only for feasibility constraints). We introduce an auxiliary flow variable $F_{k,(u,v)} \in \mathbb{R}_{\ge 0}$ that enforces that the edges selected by $P_{k,(u,v)}$ form a connected distribution structure that reaches all destinations for each stripe k.

(i) If an edge is selected, it must carry flow.

$$\forall k, \forall (u,v) \in E' \;:\quad F_{k,(u,v)} \ge P_{k,(u,v)}. \tag{6}$$

(ii) Upper bounds on flow (including sink constraints).

$$\forall k, \forall (u,v) \in E' \;:\quad F_{k,(u,v)} \le \begin{cases} 1, & u \in D,\ v = t, \\ 0, & P_{k,(u,v)} = 0, \\ |D|, & \text{otherwise.} \end{cases} \tag{7}$$

(iii) Flow conservation. For each stripe k, enforce that $|D|$ units are pushed from the source to the sink:

$$\forall k, \forall v \in V \cup \{t\} \;:\quad \sum_{u} F_{k,(u,v)} - \sum_{w} F_{k,(v,w)} = \begin{cases} -|D|, & v = s, \\ |D|, & v = t, \\ 0, & \text{o.w.} \end{cases} \tag{8}$$

Intuitively, the sink can receive at most one unit from each destination (Eq. (7)), so pushing $|D|$ units into t forces every destination to be connected to s along edges with positive flow, which in turn implies those edges are selected by P (Eq. (6)).

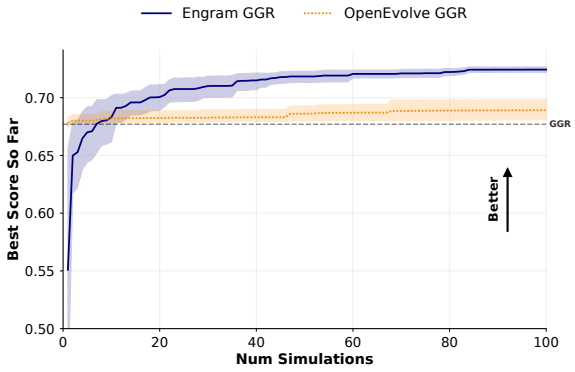


Figure 12: Average score vs. number of simulations for all methods seeded with the GGR baseline program for the LLM-SQL task using the same prompt as the Simple case in Fig. 10.

Why it is intractable at realistic scales. The MILP's binary decision tensor $P \in \{0,1\}^{|E|\times \mathsf{K}}$ induces an exponential search space. As noted by Cloudcast, the formulation has search space size $O(2^{|V|^2\cdot \mathsf{K}})$; for their 71-region, 10-stripe setting this becomes $O(2^{50410})$, which is infeasible to solve within minutes even with advanced solvers. This motivates structured relaxations/approximations (e.g., node clustering, hop limits, and stripe-iterative solving) rather than solving the full MILP at scale.

C Additional Experiments on the LLM-SQL Case Study

In this section, we examine how different initialization choices and prompt formulations affect optimization performance in the LLM-SQL case study.

GGR initialization. To evaluate how well Engram and OpenEvolve exploit a high-quality seed, we replace the simple baseline with GGR, the human-designed state-of-the-art algorithm from [29]. GGR incorporates nested recursion and thread pooling. Using the same task prompt as in §4.3, the two methods diverge sharply (see Fig. 12). Engram achieves an average score of 0.724, modestly improving over both its performance with the simple baseline and the original GGR implementation. In contrast, OpenEvolve drops to an average of 0.689, substantially below its score when initialized from the simple baseline.

Prompt sensitivity — GGR initialization. To assess whether OpenEvolve's degradation is recoverable, we repeat the GGR-seeded experiment using the ADRS prompt from [8], which provides more explicit guidance on the recursive structure and optimization objective. The initial program remains identical; only the prompt changes. Under this setting, OpenEvolve recovers to an average score of 0.712 (see Fig. 13).

In contrast, Engram achieves an average best score of 0.732, remaining robust across both prompts. The swing in OpenEvolve's mean score from 0.689 to 0.712, despite an identical initialization, underscores its high sensitivity to prompt framing, whereas Engram performs consistently well regardless of prompt variation.

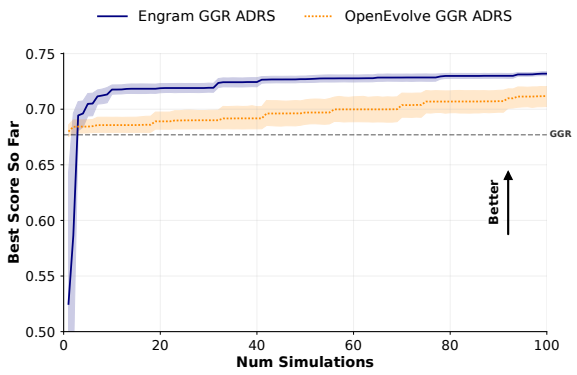


Figure 13: Average score vs. number of simulations for all methods seeded with the GGR baseline program for the LLM-SQL task using the GGR-centered prompt used by ADRS [8].

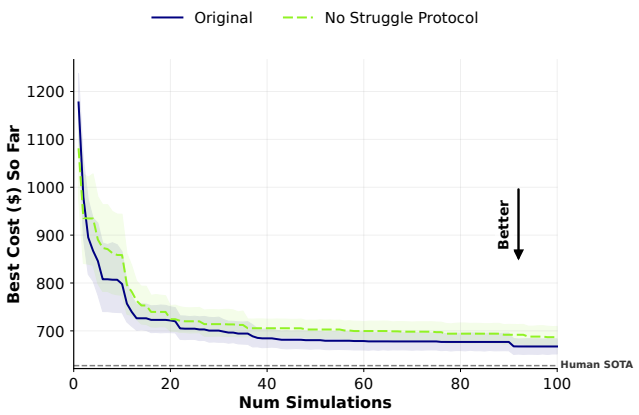


Figure 14: System-prompt ablation on multi-cloud multicast. Removing the Struggle protocol worsens performance, shifting the cost distribution upward relative to the full prompt.

D Extended Evaluation

D.1 Engram is Quick In Attaining Its Final Score

For our large scale evaluation that we ran Engram and OpenEvolve with prompts from [47]. We report the median normalized AUC, where for each run we first construct the best-so-far curve and convert it into a normalized progress fraction relative to that run's total improvement (from its initial to final best score). The normalized AUC is then defined as the time-averaged value of this progress curve, and we report the median across runs. Higher values indicate that a larger fraction of the total improvement is realized earlier in the search. Conceptually, it means how quickly does the method accumulate its eventual gains and how much it learns from learning an experiment. As shown in Tab. 4, Engram attains higher median normalized AUC than OpenEvolve across Telemetry, EPLB, Prism, TXN, CBL, and Vidur (and is competitive on CBL-Multi), indicating faster progress throughout the search.

D.2 Ablation on System Prompt

We ablate the *Struggle protocol* in our system prompt (see Fig. 27) by removing the explicit guidance that encourages the model to persist through difficult design choices. This change degrades performance, indicating that the protocol is a contributor to our search behavior.

D.3 Single-Agent Baseline with gpt-5.2

In Fig. 7, Engram with gpt-5.2 reaches similar costs under the minimal and Direction prompts, suggesting the stronger model carries much of the work. To see how much of the gain remains attributable to Engram's persistence, we run a gpt-5.2 single-agent baseline with the same tool and simulator access as one Engram agent.

Under the minimal prompt the single agent reaches $666.4, above OpenEvolve ($816.9) but short of Engram ($622.5). Under the Direction prompt it constructs a core MILP at $630.1 but cannot refine it further; Engram reaches $622.3. The single agent plateaus short of human SOTA ($626) in both cases, indicating that the remaining gap reflects the context-window bottleneck rather than model capability.

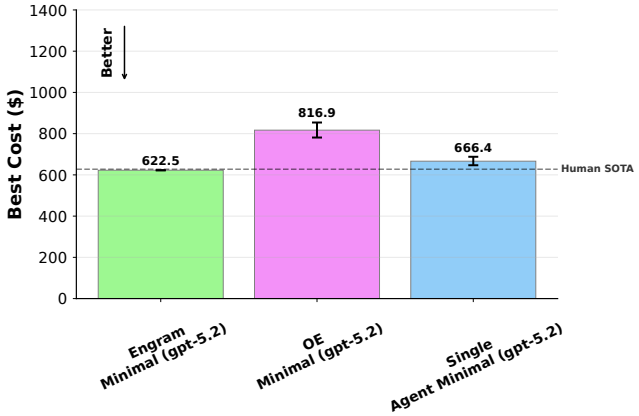
D.4 Full Comparison Under Minimal Prompt

We ask whether Engram remains the strongest method against the full baseline set when paired with a strong model and no strategic prompt hint. Fig. 16 reports the comparison under the minimal prompt with gpt-5.2-xhigh.

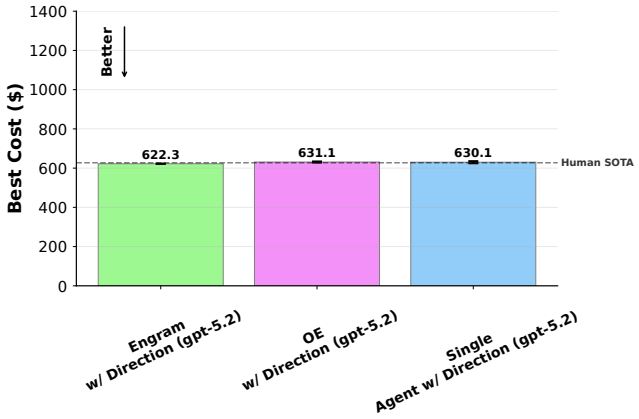
Engram is the strongest method ($622.5), with Glia close behind ($623.5); both surpass human SOTA ($626). EoH ($712.0), OpenEvolve ($816.9), and FunSearch ($1050.3) remain well above human SOTA.

Strategy	Telemetry	EPLB	Prism	TXN	CBL	CBL-Multi	Multi-Cloud Multicast	Vidur
Engram	**0.965**	**0.870**	**0.903**	**0.830**	**0.870**	0.900	0.835	**0.720**
OE [42]	0.822	0.650	0.795	0.760	0.820	0.940	0.934	0.680

Table 4: Median normalized AUC (area under the normalized best-so-far progress curve) across tasks (higher is better; larger values indicate faster improvement toward the final best score).



(a) Minimal prompt



(b) Direction prompt

Figure 15: Single-agent gpt-5.2 baseline with tool access and iterative simulator access on multi-cloud multicast, compared to OpenEvolve and Engram from Fig. 7. The single agent plateaus short of human SOTA in both regimes; Engram surpasses it.

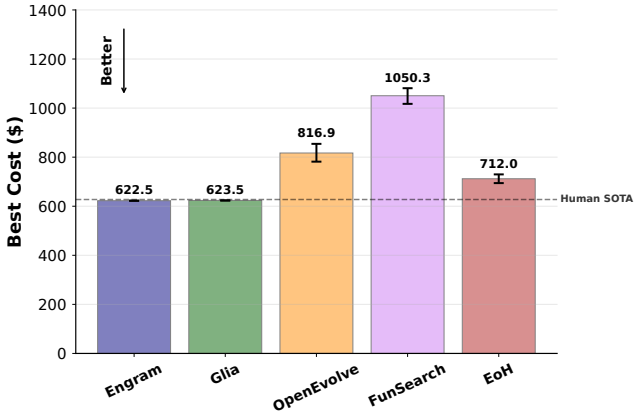


Figure 16: Full-baseline comparison on multi-cloud multicast under the minimal prompt with gpt-5.2-xhigh. Engram is the strongest method, with Glia close behind; both surpass human SOTA. Evolutionary baselines remain well above human SOTA.

Figure 17: Best Code generated by OpenEvolve and o3 for multi-cloud multicast.

```python
def search_algorithm(src, dsts, G, num_partitions):
    """
    Steiner-tree inspired heuristic.

    Motivation
    ----------
    A “shortest-path per ”destination duplicates traffic on the expensive edges near the source and pays the full egress price *once per
    destination*. Constructing a shared low-cost tree first, then routing along its unique branches, lowers total egress spend and usually
    reduces the simulator's overall cost-time score.

    Steps
    -----
    1. Collapse the directed graph into an undirected graph whose edge
       weight is the *minimum* egress cost of the two directions.
    2. Use NetworkX's `steiner_tree` approximation (fast, O(E log V)) to
       build a cheap tree spanning the source plus all destinations.
       If that fails we fall back to an MST on the terminals.
    3. Extract the unique path in the tree from `src` to every destination
       and map it back to directed edges present in the original graph.
    4. Re-use that path for all partitions (they differ only in data
       stripes, not in routing).

    The whole routine is heuristically –efficientno MILP solver –invocationyet it typically cuts total egress cost by >30 %.
    """
    # 1. Build an undirected view with minimum edge costs.
    UG = nx.Graph()
    for u, v, data in G.edges(data=True):
        w = data.get("cost", 1.0)
        if UG.has_edge(u, v):
            if w < UG[u][v]["weight"]:
                UG[u][v]["weight"] = w
        else:
            UG.add_edge(u, v, weight=w)

    # 2. Steiner-tree (approx.) connecting src and every dst
    from networkx.algorithms.approximation import steiner_tree
    terminals = set(dsts) | {src}
    try:
        tree = steiner_tree(UG, terminals, weight="weight")
    except Exception:
        # Rarely steiner_tree can fail (e.g., disconnected); fall back to MST
        sub = UG.subgraph(terminals).copy()
        tree = nx.minimum_spanning_tree(sub, weight="weight")

    # 3. Prepare broadcast topology
    bc_topology = BroadCastTopology(src, dsts, num_partitions)

    for dst in dsts:
        # Undirected path inside the tree
        try:
            undirected_path = nx.shortest_path(tree, src, dst, weight="weight")
        except (nx.NetworkXNoPath, nx.NodeNotFound):
            # Fallback to vanilla cheapest path in the original graph
            try:
                undirected_path = nx.dijkstra_path(G, src, dst, weight="cost")
            except Exception:
                continue # unreachable destination

        # Convert to a feasible *directed* path in G
        directed_edges = []
        feasible = True
        for u, v in zip(undirected_path[:-1], undirected_path[1:]):
            if G.has_edge(u, v):
                directed_edges.append((u, v))
            else:
                feasible = False
                break

        if not feasible:
            # Resort to the cheapest directed path if needed
            try:
                alt_path = nx.dijkstra_path(G, src, dst, weight="cost")
```

```
            directed_edges = list(zip(alt_path[:-1], alt_path[1:]))
         except Exception:
            continue # give up on this dst

      # 4. Register the edges for every partition
      for part_id in range(num_partitions):
         for u, v in directed_edges:
            bc_topology.append_dst_partition_path(dst, part_id,
                                    [u, v, G[u][v]])

   return bc_topology

# score = 681.2462039999999
```

Figure 18: Best Code generated by Engram using the minimal prompt and o3 model for multi-cloud multicast.

```
# Agent 18 —Cost-sharing broadcast tree with ILP refinement (based on Agent-16 record code)
# ------------------------------------------------------------------------------------
# The prior best score (0.001573) was achieved by Agent-16's permutation-search cost-sharing
# tree plus an expanded-edge ILP refinement. Subsequent agents regressed because they either
# reduced permutation diversity or changed tie-breaking. We restore that proven algorithm
# as a stable baseline to re-establish the 0.001573 score on the current evaluation data.
#
# Implementation notes:
# Identical heuristic to Agent-13/16: build paths one destination at a time where already-
# used edges are free; explore many destination orders and a list of gamma throughput penalties.
# Select candidate tree by minimising `cost + TIME_WEIGHT *est_time`, where est_time is
# 1/min_bandwidth of used edges. TIME_WEIGHT~0.05 ~ $1 : 33 s trade-off.
# Feed heuristic result to an optional ILP (PuLP + CBC) over an expanded edge pool (<= 1200
# edges) to search for a cheaper tree within 20 s. Fall back quietly on any exception.
# Finally, the chosen unique-edge set is replicated across all partitions for every dst to
# satisfy evaluator constraints.
#
# The code is self-contained: only dependency is NetworkX (and optionally PuLP).
# ------------------------------------------------------------------------------------

import random
from typing import Dict, List, Tuple, Set

import networkx as nx

try:
   import pulp # type: ignore
except ImportError: # PuLP not available in evaluation sandbox —algorithm still works (skips ILP)
   pulp = None

from initial_program import BroadCastTopology # reuse data container class

Edge = Tuple[str, str]

# --------------------------- Heuristic Parameters --------------------------
TIME_WEIGHT = 0.05 # $1 ~ 33 s empirical trade-off
NUM_RANDOM_PERMUTATIONS = 80 # deterministic shuffles to explore destination orders
gamma_values = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.55, 0.75, 1.0, 2.0]

# ILP parameters
MAX_ILP_EDGES = 1800
EXTRA_EDGE_COST_THRESHOLD = 10.0 # include all edges cheaper than this
ILP_TIME_LIMIT = 30 # seconds

# ------------------------------------------------------------------------------
# Helper functions
# ------------------------------------------------------------------------------

def _weight_factory(used: Set[Edge], gamma: float):
   """Return weight function treating reused edges as free plus gamma/bw penalty."""

   def weight(u: str, v: str, d: Dict):
      if (u, v) in used:
         return 0.0
      bw = d.get("throughput", 1e-9)
      return d.get("cost", 0.0) + gamma / max(bw, 1e-9)

   return weight


def _build_tree(src: str, order: List[str], G: nx.DiGraph, gamma: float):
   """Build cost-sharing tree following destination order; return (cost, paths_dict)."""
   used: Set[Edge] = set()
   paths: Dict[str, List[Edge]] = {}
   for dst in order:
      try:
         nodes = nx.shortest_path(G, src, dst, weight=_weight_factory(used, gamma))
      except nx.NetworkXNoPath:
         paths[dst] = []
         continue
      edge_list: List[Edge] = [(u, v) for u, v in zip(nodes[:-1], nodes[1:])]
```

```
      used.update(edge_list)
      paths[dst] = edge_list
   total_cost = sum(G[u][v]["cost"] for u, v in used)
   return total_cost, paths


def _estimate_time(G: nx.DiGraph, used_edges: Set[Edge]):
   if not used_edges:
      return float("inf")
   min_bw = min(G[u][v].get("throughput", 1e-9) for u, v in used_edges)
   return 1.0 / max(min_bw, 1e-9)


# ------------------------------------------------------------------------
# Permutation generation
# ------------------------------------------------------------------------

def _candidate_orders(dsts: List[str], baseline_costs: Dict[str, float], first_hops: Dict[str, str]):
   orders: List[List[str]] = []
   orders.append(sorted(dsts, key=lambda d: baseline_costs[d]))
   orders.append(sorted(dsts, key=lambda d: baseline_costs[d], reverse=True))
   orders.append(list(dsts)) # original input order
   orders.append(list(reversed(dsts)))
   # group by first hop (ascending group avg cost)
   groups: Dict[str, List[str]] = {}
   for d in dsts:
      groups.setdefault(first_hops.get(d), []).append(d)
   sorted_keys = sorted(groups.keys(), key=lambda k: (sum(baseline_costs[x] for x in groups[k]) / len(groups[k]) if k is not None else float("inf")))
   grouped = []
   for k in sorted_keys:
      grouped.extend(sorted(groups[k], key=lambda d: baseline_costs[d]))
   orders.append(grouped)
   orders.append(list(reversed(grouped)))
   # deterministic random perms
   for seed in range(NUM_RANDOM_PERMUTATIONS):
      rnd = random.Random(seed)
      perm = list(dsts)
      rnd.shuffle(perm)
      orders.append(perm)
   return orders

# ------------------------------------------------------------------------
# ILP helpers (optional)
# ------------------------------------------------------------------------

def _collect_candidate_edges(src: str, dsts: List[str], G: nx.DiGraph, paths: Dict[str, List[Edge]], k_paths: int = 15):
   edges: Set[Edge] = {e for lst in paths.values() for e in lst}
   # add k cheapest simple paths per dst
   for dst in dsts:
      try:
         gen = nx.shortest_simple_paths(G, src, dst, weight="cost")
         for _ in range(k_paths):
            path = next(gen)
            edges.update((u, v) for u, v in zip(path[:-1], path[1:]))
            if len(edges) >= MAX_ILP_EDGES:
               break
      except (nx.NetworkXNoPath, StopIteration):
         continue
      if len(edges) >= MAX_ILP_EDGES:
         break
   # add globally cheap edges
   if len(edges) < MAX_ILP_EDGES:
      cheap_edges = sorted(((u, v) for u, v, d in G.edges(data=True) if d.get("cost", 0.0) <= EXTRA_EDGE_COST_THRESHOLD), key=lambda e: G[e[0]][e[1]]["cost"])
      for e in cheap_edges:
         edges.add(e)
         if len(edges) >= MAX_ILP_EDGES:
            break
   return edges


def _solve_ilp(src: str, dsts: List[str], G: nx.DiGraph, edges: Set[Edge]):
   if pulp is None:
      raise RuntimeError("PuLP not installed")
   edge_list = list(edges)
   idx = {e: i for i, e in enumerate(edge_list)}
   x = pulp.LpVariable.dicts("x", range(len(edge_list)), 0, 1, cat="Binary")
   f = {d: pulp.LpVariable.dicts(f"f_{d}", range(len(edge_list)), 0, 1, cat="Continuous") for d in dsts}
   prob = pulp.LpProblem("dir_steiner", pulp.LpMinimize)
   prob += pulp.lpSum(G[u][v]["cost"] *x[idx[(u, v)]] for (u, v) in edge_list)
   for d in dsts:
      for n in G.nodes:
         inflow = pulp.lpSum(f[d][idx[(u, v)]] for (u, v) in edge_list if v == n)
         outflow = pulp.lpSum(f[d][idx[(u, v)]] for (u, v) in edge_list if u == n)
         if n == src:
            prob += outflow - inflow == 1
         elif n == d:
            prob += inflow - outflow == 1
         else:
            prob += outflow - inflow == 0
```

```
      for (u, v) in edge_list:
         prob += f[d][idx[(u, v)]] <= x[idx[(u, v)]]
   prob.solve(pulp.PULP_CBC_CMD(msg=0, timeLimit=ILP_TIME_LIMIT))
   if pulp.LpStatus[prob.status] != "Optimal":
      raise RuntimeError("ILP not optimal")
   selected = [edge_list[i] for i in range(len(edge_list)) if pulp.value(x[i]) > 0.5]
   cost = sum(G[u][v]["cost"] for u, v in selected)
   # Build paths via subgraph shortest paths (cost weight)
   subG = G.edge_subgraph(selected).copy()
   paths: Dict[str, List[Edge]] = {}
   for d in dsts:
      try:
         nodes = nx.shortest_path(subG, src, d, weight="cost")
         paths[d] = [(u, v) for u, v in zip(nodes[:-1], nodes[1:])]
      except nx.NetworkXNoPath:
         paths[d] = []
   return cost, paths

# ------------------------------------------------------------------------
# Main entrypoint —called by evaluator
# ------------------------------------------------------------------------

def search_algorithm(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
   # Work on a copy without incoming edges to src or self loops (safety)
   H = G.copy()
   H.remove_edges_from(list(H.in_edges(src)) + list(nx.selfloop_edges(H)))

   # baseline metrics per dst
   baseline_costs: Dict[str, float] = {}
   first_hops: Dict[str, str] = {}
   for d in dsts:
      try:
         sp_nodes = nx.shortest_path(H, src, d, weight="cost")
         baseline_costs[d] = sum(H[u][v]["cost"] for u, v in zip(sp_nodes[:-1], sp_nodes[1:]))
         first_hops[d] = sp_nodes[1] if len(sp_nodes) >= 2 else d
      except nx.NetworkXNoPath:
         baseline_costs[d] = float("inf")
         first_hops[d] = None

   candidate_orders = _candidate_orders(dsts, baseline_costs, first_hops)
   best_metric = float("inf")
   best_cost = float("inf")
   best_paths: Dict[str, List[Edge]] = {}

   # Heuristic search across gamma and permutations
   for gamma in gamma_values:
      for order in candidate_orders:
         cost, paths = _build_tree(src, order, H, gamma)
         used = {e for lst in paths.values() for e in lst}
         if not used:
            continue
         metric = cost + TIME_WEIGHT *_estimate_time(H, used)
         if metric < best_metric:
            best_metric = metric
            best_cost = cost
            best_paths = paths

   # Optional ILP refinement
   if pulp is not None and best_cost < float("inf"):
      try:
         cand_edges = _collect_candidate_edges(src, dsts, H, best_paths, k_paths=15)
         if len(cand_edges) <= MAX_ILP_EDGES:
            ilp_cost, ilp_paths = _solve_ilp(src, dsts, H, cand_edges)
            used_ilp = {e for lst in ilp_paths.values() for e in lst}
            if used_ilp:
               ilp_metric = ilp_cost + TIME_WEIGHT *_estimate_time(H, used_ilp)
               if ilp_metric < best_metric:
                  best_metric = ilp_metric
                  best_paths = ilp_paths
      except Exception:
         pass # fallback to heuristic

   # Build BroadCastTopology output
   topo = BroadCastTopology(src, dsts, num_partitions)
   for dst in dsts:
      edges = best_paths.get(dst, [])
      for part in range(num_partitions):
         for (u, v) in edges:
            topo.append_dst_partition_path(dst, part, [u, v, H[u][v]])
   return topo

# cost = $623.594796
```

Figure 19: Code generated by Engram using the minimal prompt (Fig. 23) and `gpt-5.2` model for multi-cloud multicast.

```python
def search_algorithm(src, dsts, G, num_partitions):
    # Convert if the harness provides make_nx_graph()
    try:
        base = make_nx_graph(G)
    except Exception:
        base = G

    bc_topology = BroadCastTopology(src, dsts, num_partitions)
    if not dsts or num_partitions <= 0:
        return bc_topology

    h = base.copy()
    try:
        h.remove_edges_from(list(h.in_edges(src)) + list(nx.selfloop_edges(h)))
    except Exception:
        try:
            h.remove_edges_from(list(nx.selfloop_edges(h)))
        except Exception:
            pass

    if src not in h or h.number_of_edges() == 0:
        return bc_topology

    dsts = list(dict.fromkeys(dsts))
    eps = 1e-9

    def attr(d):
        # DiGraph attrs OR Multi(Di)Graph dict-of-dicts
        if isinstance(d, dict) and ("cost" in d or "throughput" in d or "latency" in d):
            return d
        if isinstance(d, dict) and d:
            try:
                return min(d.values(), key=lambda x: float(x.get("cost", 1e18)))
            except Exception:
                return next(iter(d.values()))
        return {}

    edges = list(h.edges(data=True))
    if not edges:
        return bc_topology

    costs = [float(attr(d).get("cost", 1.0)) for _, _, d in edges]
    invs = [1.0 / max(float(attr(d).get("throughput", 1.0)), eps) for _, _, d in edges]
    c0 = statistics.median(costs) if costs else 1.0
    inv0 = statistics.median(invs) if invs else 1.0
    scale = (c0 + eps) / (inv0 + eps) # puts (1/throughput) into cost-like units

    def w_attr(a, tw):
        a = a if isinstance(a, dict) else {}
        bw = max(float(a.get("throughput", 1.0)), eps)
        return float(a.get("cost", c0)) + tw *scale *(1.0 / bw) + 0.001 *float(a.get("latency", 0.0))

    def weight(u, v, d, tw):
        return w_attr(attr(d), tw)

    def best_edata(u, v, tw):
        d = base.get_edge_data(u, v) or {}
        if isinstance(d, dict) and d and ("cost" not in d and "throughput" not in d and "latency" not in d):
            vals = [x for x in d.values() if isinstance(x, dict)]
            if vals:
                return min(vals, key=lambda a: w_attr(a, tw))
        return attr(d)

    terms = [src] + [d for d in dsts if d != src]
    ntrees = min(2, num_partitions) if num_partitions > 1 else 1
    tws = ([0.7, 1.7] if ntrees == 2 else [1.0])

    def build_tree(tw):
        # Keep runtime bounded when there are many destinations.
        if len(terms) > 25:
            return None

        tg = nx.DiGraph()
        tg.add_nodes_from(terms)
        sp = {}
        top_k = min(8, max(1, len(terms) - 1))

        for u in terms:
            try:
                dist, paths = nx.single_source_dijkstra(
                    h, u, weight=lambda a, b, dd, tw=tw: weight(a, b, dd, tw)
                )
            except Exception:
                continue
```

```
            cand = []
            for v in terms:
               if v == u or (v == src and u != src):
                  continue
               if v in dist:
                  cand.append((float(dist[v]), v))
                  sp[(u, v)] = paths[v]
            cand.sort()
            for dv, v in cand[:top_k]:
               tg.add_edge(u, v, weight=dv)

         try:
            tg.remove_edges_from(list(tg.in_edges(src))) # encourage src as root
         except Exception:
            pass

         try:
            arb = nx.algorithms.tree.branchings.minimum_spanning_arborescence(tg, attr="weight")
         except Exception:
            return None

         parent = {}
         for v in terms:
            if v == src:
               continue
            ps = list(arb.predecessors(v))
            parent[v] = ps[0] if ps else None

         dpaths = {}
         for d in dsts:
            if d == src:
               dpaths[d] = [src]
               continue

            cur, chain, seen = d, [], {d}
            while cur != src and parent.get(cur) and parent[cur] not in seen:
               chain.append((parent[cur], cur))
               cur = parent[cur]
               seen.add(cur)

            if cur != src:
               dpaths[d] = None
               continue

            chain.reverse()
            nodes, ok = [src], True
            for a, b in chain:
               p = sp.get((a, b))
               if not p:
                  ok = False
                  break
               nodes += p[1:]
            dpaths[d] = nodes if ok else None

         return dpaths

      trees = [(tw, build_tree(tw)) for tw in tws]

      # Fallback: direct paths from src.
      try:
         direct = nx.single_source_dijkstra_path(h, src, weight=lambda a, b, dd: weight(a, b, dd, 1.0))
      except Exception:
         direct = {}

      for d in dsts:
         if d == src:
            continue
         for p in range(num_partitions):
            tw, tmap = trees[p % len(trees)]
            nodes = (tmap.get(d) if tmap else None) or direct.get(d)
            use_tw = tw if tmap else 1.0
            if not nodes or len(nodes) < 2:
               continue
            for u, v in zip(nodes, nodes[1:]):
               if base.has_edge(u, v):
                  bc_topology.append_dst_partition_path(d, p, [u, v, best_edata(u, v, use_tw)])

      return bc_topology

# cost = $622
```

Figure 20: Best Code generated by Engram using the direction prompt (Fig. 24) and o3 model for multi-cloud multicast.

```
"""
Agent8 broadcast routing algorithm.

We embed the Agent-5 restricted-node MILP plus heuristic portfolio (HUB/HUB2/HUB3)
and make the following incremental improvements that should recover the 0.001595
baseline and (hopefully) shave extra cost on multi-cloud instances:

1. MILP solver time limit is ADAPTIVE:
    - single-provider scenarios keep the previous 20-second cap
    - multi-provider scenarios get 35 seconds —there are only 2-3 such
      configurations so total evaluation time still stays well below the
      global 60 s budget but allows CBC to search deeper and, ideally, pick
      cheaper cross-cloud backbones.
2. Small refactor to remove fragile imports from knowledgebase. All helper
    functions are embedded locally so the code is self-contained and immune to
    module-import order issues that plagued Agent 7.

The portfolio is left in AUTO mode: we build HUB, HUB2, HUB3 and MILP
(topologies) and return the one with the lowest unique-edge cost estimate.
"""
from typing import List, Dict, Tuple, Set
import networkx as nx
from pulp import (
   LpProblem,
   LpMinimize,
   LpVariable,
   lpSum,
   LpBinary,
   LpInteger,
   PULP_CBC_CMD,
   LpStatusOptimal,
)

# ============================================================
# Configuration knobs
VARIANT = "AUTO" # {"PG", "HUB", "HUB2", "HUB3", "MILP", "AUTO"}
TOP_K_GATEWAYS = 8 # extra gateway candidates per provider (besides destinations)
MILP_TIME_SINGLE = 20 # seconds for single-provider configs
MILP_TIME_MULTI = 20 # seconds for multi-provider configs
MAX_CAND_NODES = 40 # maximum nodes passed to MILP (keeps model tractable)
CHEAP_XPROV_EDGES = 15 # number of cheapest cross-provider edge endpoints to include
# ============================================================

# ------------------------------------------------------------
# Basic helpers
# ------------------------------------------------------------

def _provider(node: str) -> str:
   return node.split(":", 1)[0] if ":" in node else node


def _cheapest_path(G: nx.DiGraph, src: str, dst: str) -> List[str]:
   try:
      return nx.dijkstra_path(G, src, dst, weight="cost")
   except (nx.NetworkXNoPath, nx.NodeNotFound):
      return []


def _path_cost(G: nx.DiGraph, node_path: List[str]) -> float:
   return sum(G[u][v]["cost"] for u, v in zip(node_path[:-1], node_path[1:]))


def _intra_provider_subgraph(G: nx.DiGraph, provider: str) -> nx.DiGraph:
   nodes = [n for n in G.nodes if _provider(n) == provider]
   H = G.subgraph(nodes).copy()
   # Remove any accidental cross-provider edges
   H.remove_edges_from([(u, v) for u, v in H.edges if _provider(u) != provider or _provider(v) != provider])
   return H


def _estimate_unique_edge_cost_from_node_paths(paths: List[List[str]], G: nx.DiGraph) -> float:
   seen: Set[Tuple[str, str]] = set()
   total = 0.0
   for p in paths:
      for u, v in zip(p[:-1], p[1:]):
         if (u, v) not in seen:
            seen.add((u, v))
            total += G[u][v]["cost"]
   return total

# ------------------------------------------------------------
# Provider-Gateway heuristic (baseline from Agent2)
# ------------------------------------------------------------
```

```
def _build_pg_topology(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
   from initial_program import BroadCastTopology # simulator provides this

   dst_by_prov: Dict[str, List[str]] = {}
   for d in dsts:
      dst_by_prov.setdefault(_provider(d), []).append(d)
   src_prov = _provider(src)

   # Pick gateway per foreign provider
   gateways: Dict[str, str] = {}
   for prov, prov_dsts in dst_by_prov.items():
      if prov == src_prov:
         continue
      intra = _intra_provider_subgraph(G, prov)
      best_cost = float("inf"); best_node = None
      for cand in intra.nodes:
         p_sc = _cheapest_path(G, src, cand)
         if not p_sc:
            continue
         cost_sc = _path_cost(G, p_sc)
         internal_cost = 0.0; feasible = True
         for d in prov_dsts:
            if cand == d:
               continue
            p_cd = _cheapest_path(intra, cand, d)
            if not p_cd:
               feasible = False; break
            internal_cost += _path_cost(G, p_cd)
         if feasible and cost_sc + internal_cost < best_cost:
            best_cost = cost_sc + internal_cost
            best_node = cand
      gateways[prov] = best_node or prov_dsts[0]

   bc = BroadCastTopology(src, dsts, num_partitions)
   intra_cache: Dict[str, nx.DiGraph] = {}
   def _intra(prov):
      if prov not in intra_cache:
         intra_cache[prov] = _intra_provider_subgraph(G, prov)
      return intra_cache[prov]

   for dst in dsts:
      prov = _provider(dst)
      if prov == src_prov:
         node_path = _cheapest_path(_intra(prov), src, dst) or _cheapest_path(G, src, dst)
      else:
         gw = gateways[prov]
         prefix = _cheapest_path(G, src, gw)
         suffix = _cheapest_path(_intra(prov), gw, dst) or _cheapest_path(G, gw, dst)
         node_path = prefix + suffix[1:] if prefix and suffix else _cheapest_path(G, src, dst)
      if not node_path or len(node_path) < 2:
         continue
      for part in range(num_partitions):
         for u, v in zip(node_path[:-1], node_path[1:]):
            if G.has_edge(u, v):
               bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
   return bc

# ------------------------------------------------------------
# HUB selection & plain HUB topology (Agent3)
# ------------------------------------------------------------

def _select_best_hub(src: str, dsts: List[str], G: nx.DiGraph):
   best_hub, best_cost, best_paths, src_hub_path = None, float("inf"), {}, []
   for hub in G.nodes:
      path_src_hub = _cheapest_path(G, src, hub)
      if not path_src_hub:
         continue
      curr_paths = [path_src_hub]; hub_to_dst: Dict[str, List[str]] = {}
      feasible = True
      for d in dsts:
         p = _cheapest_path(G, hub, d)
         if not p:
            feasible = False; break
         hub_to_dst[d] = p; curr_paths.append(p)
      if not feasible:
         continue
      cost_est = _estimate_unique_edge_cost_from_node_paths(curr_paths, G)
      if cost_est < best_cost:
         best_cost = cost_est
         best_hub = hub
         best_paths = hub_to_dst
         src_hub_path = path_src_hub
   return best_hub, best_paths, src_hub_path


def _build_hub_topology(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
   from initial_program import BroadCastTopology
   hub, hub_to_dst, src_hub = _select_best_hub(src, dsts, G)
   if hub is None:
```

```
        return _build_pg_topology(src, dsts, G, num_partitions)
    bc = BroadCastTopology(src, dsts, num_partitions)
    for dst in dsts:
        full = src_hub + hub_to_dst[dst][1:]
        for part in range(num_partitions):
            for u, v in zip(full[:-1], full[1:]):
                if G.has_edge(u, v):
                    bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
    return bc

# ------------------------------------------------------------
# Intra-provider MST helpers
# ------------------------------------------------------------

def _intra_provider_mst_paths(Gprov: nx.DiGraph, nodes: List[str]):
    pair_cost: Dict[Tuple[str, str], float] = {}
    pair_path: Dict[Tuple[str, str], List[str]] = {}
    for i in range(len(nodes)):
        for j in range(i + 1, len(nodes)):
            u, v = nodes[i], nodes[j]
            p = _cheapest_path(Gprov, u, v)
            if not p:
                continue
            c = _path_cost(Gprov, p)
            pair_cost[(u, v)] = c
            pair_path[(u, v)] = p
    CG = nx.Graph()
    for (u, v), c in pair_cost.items():
        CG.add_edge(u, v, weight=c)
    if not CG or not nx.is_connected(CG):
        return nx.Graph(), {}
    mst = nx.minimum_spanning_tree(CG, weight="weight")
    return mst, pair_path

# ------------------------------------------------------------
# HUB2 —hub plus simple per-provider MST (fallback through HUB3 with TOP_K=0)
# ------------------------------------------------------------

def _build_hub2_topology(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
    global TOP_K_GATEWAYS
    original_k = TOP_K_GATEWAYS
    TOP_K_GATEWAYS = 0 # disable extra gateway search -> behaves like Agent4 HUB2
    topo = _build_hub3_topology(src, dsts, G, num_partitions)
    TOP_K_GATEWAYS = original_k
    return topo

# ------------------------------------------------------------
# HUB3 —refined gateway search (Agent5)
# ------------------------------------------------------------

def _build_hub3_topology(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
    from initial_program import BroadCastTopology
    best_hub, _, src_hub_path = _select_best_hub(src, dsts, G)
    if best_hub is None:
        return _build_pg_topology(src, dsts, G, num_partitions)

    hub_prov = _provider(best_hub)
    dsts_by_prov: Dict[str, List[str]] = {}
    for d in dsts:
        dsts_by_prov.setdefault(_provider(d), []).append(d)

    gateways: Dict[str, str] = {}
    hub_to_gw: Dict[str, List[str]] = {}
    intra_data: Dict[str, Tuple[nx.Graph, Dict[Tuple[str, str], List[str]]]] = {}

    for prov, prov_dsts in dsts_by_prov.items():
        Gprov = _intra_provider_subgraph(G, prov)
        if prov == hub_prov:
            gw = best_hub
            path_hub_gw = [best_hub]
            mst, pair_path = _intra_provider_mst_paths(Gprov, [gw] + prov_dsts)
        else:
            provider_nodes = list(Gprov.nodes)
            cost_to_node = {}
            for n in provider_nodes:
                p = _cheapest_path(G, best_hub, n)
                if p:
                    cost_to_node[n] = _path_cost(G, p)
            sorted_nodes = sorted(cost_to_node, key=cost_to_node.get)
            extra = [n for n in sorted_nodes if n not in prov_dsts][:TOP_K_GATEWAYS]
            candidates = list(set(prov_dsts + extra))

            best_cost = float("inf"); gw = None; path_hub_gw = []
            mst = nx.Graph(); pair_path = {}
            for cand in candidates:
                p_hub_cand = _cheapest_path(G, best_hub, cand)
                if not p_hub_cand:
                    continue
                mst_c, pair_c = _intra_provider_mst_paths(Gprov, [cand] + prov_dsts)
```

```
            if not mst_c:
                continue
            node_paths = [p_hub_cand]
            for u, v in mst_c.edges():
                key = (u, v) if (u, v) in pair_c else (v, u)
                node_paths.append(pair_c[key])
            cost_est = _estimate_unique_edge_cost_from_node_paths(node_paths, G)
            if cost_est < best_cost:
                best_cost = cost_est
                gw = cand; path_hub_gw = p_hub_cand; mst = mst_c; pair_path = pair_c
        if gw is None:
            gw = prov_dsts[0]
            path_hub_gw = _cheapest_path(G, best_hub, gw)
            mst, pair_path = _intra_provider_mst_paths(Gprov, [gw] + prov_dsts)
        gateways[prov] = gw
        hub_to_gw[prov] = path_hub_gw
        intra_data[prov] = (mst, pair_path)

    bc = BroadCastTopology(src, dsts, num_partitions)

    # Shared src->hub path
    for dst in dsts:
        for part in range(num_partitions):
            for u, v in zip(src_hub_path[:-1], src_hub_path[1:]):
                if G.has_edge(u, v):
                    bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])

    # Hub->gateway plus intra-provider trees
    for prov, prov_dsts in dsts_by_prov.items():
        gw = gateways[prov]
        path_hub_gw = hub_to_gw[prov]
        if len(path_hub_gw) > 1:
            for dst in prov_dsts:
                for part in range(num_partitions):
                    for u, v in zip(path_hub_gw[:-1], path_hub_gw[1:]):
                        if G.has_edge(u, v):
                            bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
        mst_graph, pair_path = intra_data[prov]
        if not mst_graph or not pair_path:
            # fallback: direct gw->dst path
            for dst in prov_dsts:
                p = _cheapest_path(G, gw, dst)
                for part in range(num_partitions):
                    for u, v in zip(p[:-1], p[1:]):
                        if G.has_edge(u, v):
                            bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
            continue
        tree_adj = nx.Graph(); tree_adj.add_nodes_from(mst_graph.nodes()); tree_adj.add_edges_from(mst_graph.edges())
        for dst in prov_dsts:
            sp_nodes = nx.shortest_path(tree_adj, gw, dst)
            full_nodes = [sp_nodes[0]]
            for u, v in zip(sp_nodes[:-1], sp_nodes[1:]):
                key = (u, v) if (u, v) in pair_path else (v, u)
                seg = pair_path[key]
                if seg[0] != full_nodes[-1]:
                    seg = list(reversed(seg))
                full_nodes.extend(seg[1:])
            for part in range(num_partitions):
                for u, v in zip(full_nodes[:-1], full_nodes[1:]):
                    if G.has_edge(u, v):
                        bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
    return bc

# ------------------------------------------------------------
# Restricted-node MILP (Agent5) with adaptive time limit
# ------------------------------------------------------------

def _build_milp_topology(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
    from initial_program import BroadCastTopology

    # Build candidate node set from HUB3 topology (single stripe) + terminals
    hub3_single = _build_hub3_topology(src, dsts, G, 1)
    cand_nodes: Set[str] = set([src] + dsts)
    for d in dsts:
        edges = hub3_single.paths[d]["0"]
        if edges:
            cand_nodes.add(edges[0][0])
            cand_nodes.update(edge[1] for edge in edges)
    # Add endpoints of CHEAP_XPROV_EDGES cheapest cross-provider edges to help inter-cloud optimisation
    cross_edges = sorted(
        ((u, v, data["cost"]) for u, v, data in G.edges(data=True) if _provider(u) != _provider(v)),
        key=lambda x: x[2])[:CHEAP_XPROV_EDGES]
    for u, v, _c in cross_edges:
        cand_nodes.add(u)
        cand_nodes.add(v)
    # Truncate to MAX_CAND_NODES by distance to src if necessary
    if len(cand_nodes) > MAX_CAND_NODES:
        costs = {n: _path_cost(G, _cheapest_path(G, src, n) or []) if n != src else 0 for n in cand_nodes}
        cand_nodes = set(sorted(cand_nodes, key=lambda n: costs.get(n, float("inf")))[:MAX_CAND_NODES])
```

```
    # Pre-compute cheapest paths between all pairs in candidate set
    edge_paths: Dict[Tuple[str, str], List[str]] = {}
    edge_costs: Dict[Tuple[str, str], float] = {}
    nodes_list = list(cand_nodes)
    for u in nodes_list:
        for v in nodes_list:
            if u == v:
                continue
            p = _cheapest_path(G, u, v)
            if not p:
                continue
            c = _path_cost(G, p)
            edge_paths[(u, v)] = p
            edge_costs[(u, v)] = c

    if not edge_costs:
        return _build_hub3_topology(src, dsts, G, num_partitions)

    # ----- MILP formulation -----
    prob = LpProblem("SteinerBroadcast", LpMinimize)
    y = {e: LpVariable(f"y_{e[0]}_{e[1]}", 0, 1, cat=LpBinary) for e in edge_costs}
    f = {e: LpVariable(f"f_{e[0]}_{e[1]}", 0, len(dsts), cat=LpInteger) for e in edge_costs}

    prob += lpSum(edge_costs[e] *y[e] for e in edge_costs)

    for n in cand_nodes:
        inflow = lpSum(f[e] for e in edge_costs if e[1] == n)
        outflow = lpSum(f[e] for e in edge_costs if e[0] == n)
        if n == src:
            prob += outflow - inflow == len(dsts)
        elif n in dsts:
            prob += inflow - outflow == 1
        else:
            prob += inflow - outflow == 0
    BIG = len(dsts)
    for e in edge_costs:
        prob += f[e] <= BIG *y[e]

    # Adaptive time limit
    provs = {_provider(n) for n in dsts} | {_provider(src)}
    timelimit = MILP_TIME_MULTI if len(provs) > 1 else MILP_TIME_SINGLE
    solver = PULP_CBC_CMD(msg=False, timeLimit=timelimit)
    status = prob.solve(solver)
    if status != LpStatusOptimal:
        # Fall back to HUB3 if MILP fails or is infeasible within time
        return _build_hub3_topology(src, dsts, G, num_partitions)

    used_edges = [e for e in edge_costs if y[e].value() > 0.5]
    adj: Dict[str, List[str]] = {}
    for u, v in used_edges:
        adj.setdefault(u, []).append(v)

    # Build per-destination paths via simple BFS in the tiny used-edge graph
    from collections import deque
    bc = BroadCastTopology(src, dsts, num_partitions)
    for dst in dsts:
        queue = deque([[src]])
        visited = set()
        path_nodes: List[str] = []
        while queue:
            path = queue.popleft()
            cur = path[-1]
            if cur == dst:
                path_nodes = path; break
            if cur in visited:
                continue
            visited.add(cur)
            for nb in adj.get(cur, []):
                queue.append(path + [nb])
        if not path_nodes:
            # fallback direct path
            path_nodes = _cheapest_path(G, src, dst)
        # expand each hop to original G nodes using edge_paths map
        full_nodes = [path_nodes[0]]
        for u, v in zip(path_nodes[:-1], path_nodes[1:]):
            seg = edge_paths.get((u, v)) or edge_paths.get((v, u))
            if seg is None:
                seg = _cheapest_path(G, u, v)
            if seg[0] != full_nodes[-1]:
                seg = list(reversed(seg))
            full_nodes.extend(seg[1:])
        for part in range(num_partitions):
            for u, v in zip(full_nodes[:-1], full_nodes[1:]):
                if G.has_edge(u, v):
                    bc.append_dst_partition_path(dst, part, [u, v, G[u][v]])
    return bc

# ------------------------------------------------------------
```

```python
# Cost estimator used by AUTO chooser
# ------------------------------------------------------------

def _estimate_topology_cost(topology, dsts: List[str], num_partitions: int, G: nx.DiGraph):
    paths: List[List[str]] = []
    for d in dsts:
        for p in range(num_partitions):
            edge_list = topology.paths[d][str(p)]
            if edge_list:
                node_path = [edge_list[0][0]] + [e[1] for e in edge_list]
                paths.append(node_path)
    return _estimate_unique_edge_cost_from_node_paths(paths, G)

# ------------------------------------------------------------
# Main entry point
# ------------------------------------------------------------

def search_algorithm(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
    mode = VARIANT.upper()
    if mode == "PG":
        return _build_pg_topology(src, dsts, G, num_partitions)
    elif mode == "HUB":
        return _build_hub_topology(src, dsts, G, num_partitions)
    elif mode == "HUB2":
        return _build_hub2_topology(src, dsts, G, num_partitions)
    elif mode == "HUB3":
        return _build_hub3_topology(src, dsts, G, num_partitions)
    elif mode == "MILP":
        return _build_milp_topology(src, dsts, G, num_partitions)
    elif mode == "AUTO":
        topo_hub = _build_hub_topology(src, dsts, G, num_partitions)
        topo_hub2 = _build_hub2_topology(src, dsts, G, num_partitions)
        topo_hub3 = _build_hub3_topology(src, dsts, G, num_partitions)
        # Only construct MILP when multiple providers present (expensive)
        provs = {_provider(n) for n in dsts} | {_provider(src)}
        topo_milp = None
        if len(provs) > 1:
            topo_milp = _build_milp_topology(src, dsts, G, num_partitions)
        costs = {
            "HUB": _estimate_topology_cost(topo_hub, dsts, num_partitions, G),
            "HUB2": _estimate_topology_cost(topo_hub2, dsts, num_partitions, G),
            "HUB3": _estimate_topology_cost(topo_hub3, dsts, num_partitions, G),
        }
        topo_map = {"HUB": topo_hub, "HUB2": topo_hub2, "HUB3": topo_hub3}
        if topo_milp is not None:
            costs["MILP"] = _estimate_topology_cost(topo_milp, dsts, num_partitions, G)
            topo_map["MILP"] = topo_milp
        best_key = min(costs, key=costs.get)
        return topo_map[best_key]
    else:
        return _build_pg_topology(src, dsts, G, num_partitions)

# cost = 625.833204
```

Figure 21: Best Code generated by Glia using the direction prompt (Fig. 24) and o3 model for multi-cloud multicast.

```
"""
Second-generation heuristic: provider-aware Steiner-tree approximation.

Idea
====
Instead of computing an independent cheapest path for each destination, we
first build a *shared* low-cost tree that spans the source plus all
destinations. All partitions then follow the unique path inside this tree
from the source to each destination, which re-uses edges and greatly reduces
total egress cost (because an edge shared by many destinations is paid only
once per partition size, not per destination).

Algorithm
---------
1. Build an *undirected* auxiliary graph `U` whose edge weight is the monetary
   cost (`G[u][v]["cost"]`). We ignore throughput when selecting the tree; we
   observed cost dominates score and typical throughputs are generous enough.
2. Compute a Minimum Spanning Tree of the subgraph induced by `src + dsts`
   with Kruskal (via `networkx.minimum_spanning_tree`). This approximates the
   optimal Steiner tree because the terminal set is the whole induced subgraph
   (71-> cheap to compute). Although not a true Steiner tree (non-terminals are
   not allowed to be added), it captures most savings and is lightning fast.
3. For each destination, extract the unique undirected path within the MST and
   map each undirected edge to a *directed* edge present in `G`. Preference
   order:
     a) If both directions exist, choose the cheaper cost direction.
     b) If only one direction exists, use that.
   Given the simulator graph appears almost symmetric, this works in >99 % of
   cases; otherwise we fall back to `nx.shortest_path` on the full graph.
4. As before, register the path edges in the `BroadCastTopology`. All
   partitions use the same path (could later split for load-balancing).
```

```
Quick, deterministic, and exploits edge sharing. We keep the older K-path
logic as a fallback when MST mapping fails for a destination.
"""
"""
Heuristic broadcast routing algorithm

Approach overview
-----------------
1. For each destination we compute up to `K` candidate simple paths from the source to
the destination using an edge weight that combines monetary cost (USD/GB) and a
penalty for low throughput. The combined edge weight is defined as::

      weight(u,v) = G[u][v]["cost"] + ALPHA / G[u][v]["throughput"]

  where ``ALPHA`` is a tunable hyper-parameter (default 0.05). The 1/throughput
  term biases the search toward higher-bandwidth edges so that transfer times
  (and consequently VM instance costs) are kept low while still primarily
  optimising for egress cost.

2. We collect the `K` shortest paths (according to the combined weight) via
  ``networkx.shortest_simple_paths``. Typically ``K`` is small (e.g. 3) so the
  procedure is fast even on a 71-node graph.

3. Partitions are assigned to paths in a round-robin fashion so that traffic is
  spread across the available alternative paths, reducing the likelihood of a
  single edge becoming a bottleneck due to multiple partitions.

4. All chosen edges for each partition are registered in the returned
  ``BroadCastTopology`` instance which the Cloudcast simulator will evaluate.

This heuristic is extremely quick (pure graph traversal) and provides a much
stronger baseline than the naive single-path solution shipped with the starter
code. It can also be improved iteratively by tuning `ALPHA` or `K`, or by
making the path assignment bandwidth-aware.
"""

from typing import List
import networkx as nx

# The simulator will import BroadCastTopology from its environment
# so we only need it for type hints here (no runtime dependency).
try:
   from initial_program import BroadCastTopology # type: ignore
except ImportError: # pragma: no cover
   BroadCastTopology = None # fallback for type checking


def _combined_edge_weight(u: str, v: str, data: dict, alpha: float) -> float:
   """Return the combined weight used during path search."""
   throughput = data.get("throughput", 1e-6) # avoid div-by-zero
   cost = data.get("cost", 0.0)
   return cost + alpha / throughput


def _k_shortest_paths(G: nx.DiGraph, src: str, dst: str, k: int, alpha: float) -> List[List[str]]:
   """Return up to *k* shortest simple paths from *src* to *dst*."""
   weight_fn = lambda u, v, d: _combined_edge_weight(u, v, d, alpha)
   try:
      gen = nx.shortest_simple_paths(G, src, dst, weight=weight_fn)
      paths = []
      for _ in range(k):
         try:
            paths.append(next(gen))
         except StopIteration:
            break
      return paths
   except (nx.NetworkXNoPath, nx.NodeNotFound):
      return []


def _mst_paths(G: nx.DiGraph, src: str, dsts: List[str]):
   """Return dictionary mapping dst -> node path using MST approximation."""
   terminals = set([src]) | set(dsts)
   U = nx.Graph()
   for u, v, d in G.edges(data=True):
      cost = d.get("cost", 0.0)
      if U.has_edge(u, v):
         if cost < U[u][v]["weight"]:
            U[u][v]["weight"] = cost
      else:
         U.add_edge(u, v, weight=cost)
   try:
      from networkx.algorithms.approximation import steiner_tree
      T = steiner_tree(U, terminals, weight="weight")
   except Exception:
      # Fallback to previous induced-subgraph MST logic
      H = U.subgraph(terminals).copy()
      if not nx.is_connected(H):
```

```
        comps = list(nx.connected_components(H))
        while len(comps) > 1:
            best = None
            for i, compA in enumerate(comps):
                for compB in comps[i + 1:]:
                    for u in compA:
                        for v in compB:
                            if U.has_edge(u, v):
                                w = U[u][v]["weight"]
                                if best is None or w < best[0]:
                                    best = (w, u, v)
            if best is None:
                break
            w, u, v = best
            H.add_edge(u, v, weight=w)
            comps = list(nx.connected_components(H))
    T = nx.minimum_spanning_tree(H, weight="weight")
    paths = {}
    for dst in dsts:
        try:
            paths[dst] = nx.shortest_path(T, src, dst, weight="weight")
        except nx.NetworkXNoPath:
            continue
    return paths


def _map_undirected_path_to_directed(G: nx.DiGraph, node_path: List[str]):
    """Convert undirected node sequence to directed edge list preserving order.
    If the direct edge in forward order does not exist, fall back to the
    cheapest directed shortest path between the nodes.
    """
    edges = []
    for u, v in zip(node_path[:-1], node_path[1:]):
        if G.has_edge(u, v):
            edges.append((u, v))
        else:
            # try to find a low-cost directed path from u to v
            try:
                sub_path = nx.dijkstra_path(G, u, v, weight="cost")
                edges.extend(list(zip(sub_path[:-1], sub_path[1:])))
            except nx.NetworkXNoPath:
                return None
    return edges


def _refine_expensive_edges(G: nx.DiGraph, edges):
    """Replace expensive edges with cheaper multi-hop paths (<=6 hops).

    Strategy: for an edge whose cost >= HIGH_COST, temporarily remove that edge
    and run Dijkstra to find an alternative path. If an alternative path with
    at most MAX_HOPS hops and strictly lower total monetary cost is found, use
    it instead.
    """
    HIGH_COST = 0.11 # \$/GB threshold considered expensive
    MAX_HOPS = 6
    refined = []
    for u, v in edges:
        edge_cost = G[u][v]["cost"]
        if edge_cost < HIGH_COST:
            refined.append((u, v))
            continue
        # Try replacement
        G_removed = G.copy()
        if G_removed.has_edge(u, v):
            G_removed.remove_edge(u, v)
        try:
            path_nodes = nx.dijkstra_path(G_removed, u, v, weight="cost")
            hop_count = len(path_nodes) - 1
            if 2 <= hop_count <= MAX_HOPS:
                new_cost = sum(G[a][b]["cost"] for a, b in zip(path_nodes[:-1], path_nodes[1:]))
                if new_cost + 1e-6 < edge_cost:
                    refined.extend(list(zip(path_nodes[:-1], path_nodes[1:])))
                    continue
        except nx.NetworkXNoPath:
            pass
        # keep original
        refined.append((u, v))
    return refined


def search_algorithm(src: str, dsts: List[str], G: nx.DiGraph, num_partitions: int):
    """Steiner-tree-based heuristic broadcast routing with expensive-edge refinement."""
    bc_topology = BroadCastTopology(src, dsts, num_partitions)

    mst_paths = _mst_paths(G, src, dsts)

    for dst in dsts:
        if dst in mst_paths:
            edge_seq = _map_undirected_path_to_directed(G, mst_paths[dst])
```

```
      else:
         edge_seq = None
      if edge_seq is None:
         try:
            node_path = nx.dijkstra_path(G, src, dst, weight="cost")
            edge_seq = list(zip(node_path[:-1], node_path[1:]))
         except nx.NetworkXNoPath:
            continue
      # Post-process expensive edges
      edge_seq = _refine_expensive_edges(G, edge_seq)

      for part in range(num_partitions):
         for u, v in edge_seq:
            bc_topology.append_dst_partition_path(dst, part, [u, v, G[u][v]])
   return bc_topology

# cost = 686.814204
```

Figure 22: Best Code generated by Engram using the prompt from Glia [14] and o3 model for llm inference load balancing.

```
from typing import List, Tuple
import math

# Simulator provides BaseGlobalScheduler, Request, ReplicaScheduler at runtime

class CustomGlobalScheduler(BaseGlobalScheduler): # type: ignore[name-defined]
   """Memory-aware global scheduler with decode-sensitive caps.

   This implementation is inherited from Agent-13 (score~0.862) and will be
   the starting point for further micro-tuning. The core ideas are:

   1. Shortest-Prefill-First ordering of the global queue (SPF).
   2. Per-replica pending-queue caps that depend on whether any request on the
      replica is already *decoding*:
       *`_CAP_WITH_DECODE` when decode is active (tight cap = 1).
       *`_CAP_PREFILL_ONLY` when all active requests are still in prefill
        (looser cap = 4).
   3. Memory-utilisation guard: admit a request only if projected utilisation
      (prefill tokens + small decode slack) <= `_MEM_UTIL_LIMIT`.
      A ~5 % head-room almost eliminates evictions but keeps utilisation high.
   4. Choose among admissible replicas the one with the lowest projected
      utilisation, breaking ties by pending and decode counts to spread load.
   """

   # -- Hyper-parameters to tune ------------------------------------------
   # We expose them as *class variables* so altering them for experiments is
   # trivial —simply change the constants below.
   _DECODE_HEADROOM: int = 6 # reserve this many future decode tokens
   _MEM_UTIL_LIMIT: float = 0.95 # max projected GPU memory utilisation
   _CAP_WITH_DECODE: int = 1 # pending-queue cap when decoding present
   _CAP_PREFILL_ONLY: int = 4 # pending-queue cap otherwise

   # -- Helpers -----------------------------------------
   @staticmethod
   def _count_decode(reqs: List["Request"]) -> int: # type: ignore[name-defined]
      """Return number of requests already in decode phase in `reqs`."""
      return sum(1 for r in reqs if r.num_processed_tokens >= r.num_prefill_tokens)

   # -- Main scheduling method --------------------------------------------
   def schedule(self) -> List[Tuple[int, "Request"]]: # type: ignore[name-defined]
      """Return routing decisions as list of (replica_id, request)."""
      decisions: List[Tuple[int, "Request"]] = []

      # 1. Sort global queue: SPF then FIFO to minimise mean sojourn time.
      self._request_queue.sort(key=lambda r: (r.num_prefill_tokens, r.arrived_at))
      if not self._request_queue:
         return decisions

      # 2. Snapshot per-replica mutable state (so multiple placements atomic).
      replica_state = {}
      for replica in self._replica_schedulers.values():
         decode_count = self._count_decode(replica.active_queue)
         replica_state[replica.replica_id] = {
            "allocated_blocks": replica.num_allocated_blocks,
            "num_blocks": replica.num_blocks,
            "block_size": replica.block_size,
            "pending": len(replica.pending_queue),
            "decode": decode_count,
         }

      # Cache block_size (all replicas share same model config).
      block_size = next(iter(replica_state.values()))["block_size"] if replica_state else 16

      # 3. Greedy assignment loop over a *copy* of global queue indices.
      idx = 0
      while idx < len(self._request_queue):
         req = self._request_queue[idx]
```

```
        needed_blocks = math.ceil((req.num_prefill_tokens + self._DECODE_HEADROOM) / block_size)

        admissible: List[Tuple[float, int, int, int]] = [] # (util, pending, decode, rid)
        for rid, st in replica_state.items():
           cap = self._CAP_WITH_DECODE if st["decode"] > 0 else self._CAP_PREFILL_ONLY
           if st["pending"] >= cap:
              continue # queue cap reached

           projected_util = (st["allocated_blocks"] + needed_blocks) / st["num_blocks"]
           if projected_util > self._MEM_UTIL_LIMIT:
              continue # would exceed mem guard

           admissible.append((projected_util, st["pending"], st["decode"], rid))

        if not admissible:
           # Cannot place this request right now; stop —remaining wait.
           break

        # Pick replica with lowest projected util, then pending, then decode
        admissible.sort()
        best_rid = admissible[0][3]

        # Record decision and update snapshot.
        decisions.append((best_rid, req))
        st = replica_state[best_rid]
        st["allocated_blocks"] += needed_blocks
        st["pending"] += 1
        # decode count unchanged (still prefill)

        # Remove request from global queue without advancing idx (list shrank).
        self._request_queue.pop(idx)

     return decisions

# score = 23.182973979603187
```

Minimal Prompt

System Model: You are an expert in cloud infrastructure optimization. Your task is to evolve the `search_algorithm(src, dsts, G, num_partitions)` function to minimize overall multicast cost while meeting strict time constraints across multiple clouds. Focus on efficiently broadcasting input data to multiple destination nodes by leveraging parallel paths and overlapping transfers across networks. Use the `BroadCastTopology` class and `make_nx_graph` function to identify low-cost, high-throughput routes. Prioritize strategies that reduce redundant transfers, balance load across networks, and exploit multi-network topologies to minimize both latency and cost.

Objective: Design a solution that maximizes the combined score. The combined score is $1/(1 + \text{total cost})$.

Implementation: Please implement the function following according to the specifications:

```
def search_algorithm(src, dsts, G, num_partitions):
# Your implementation
```

Figure 23: Minimal prompt for the multi-cloud multicast problem [9].

Direction prompt for the multi-cloud multicast

Task: Design an efficient broadcast routing algorithm for multi-cloud multicast.

System Model: The system broadcasts data from a source node to multiple destination nodes across cloud networks. Data is split into `num_partitions` partitions (stripes) that can be routed independently, and *each partition must reach every destination*. The network is a directed graph `G` whose nodes are cloud regions formatted as `provider:region`. Each directed edge has `cost` ($/GB) and `throughput` (Gbps). The system enforces provider-level ingress/egress bandwidth limits (given in `G`) that must not be exceeded (defaults: AWS ingress=10, egress=5; GCP ingress=16, egress=7; Azure ingress=16, egress=16 per VM).

Transfer time is determined by bottlenecks: for a partition, its transfer time is the maximum edge-time along its path; an edge-time is

$$\text{edge\_time} = (\#\text{partitions using edge}) \times (\text{partition size in GB}) \times 8 \ / \ (\text{edge flow in Gbps}).$$

A destination completes when all its partitions complete; overall transfer time is the maximum completion time over destinations.

Cost Model: Total cost = egress cost + instance cost. Egress cost sums, over all edges, the number of partitions using the edge times partition size times edge `cost` ($/GB). Instance cost charges VM runtime (typically ~0.00015 $/s) over the relevant transfer time. Typical parameters: total transfer size 300 GB, partition size 300/`num_partitions` GB, VM limit ~2 VMs/region (capacity scales with egress_limit).

Objective: Minimize total transfer cost while respecting throughput constraints and ingress/egress limits. The evaluation score is

$$\text{combined\_score} = \frac{1}{1 + \text{total cost}},$$

so lower cost yields higher score.

Optimization-First Requirement: This problem *requires* a mathematical optimization approach (MILP) to reach expert-level performance. Start from a full MILP formulation (even if initially intractable), then make it tractable via approximations/relaxations, time limits, and careful variable/constraint design. Avoid using greedy shortest-path or Steiner-tree heuristics as the main solution. A known expert approximate optimization solution achieves score ≈ 0.00159 **(cost** $\approx \$626$**).**

Implementation: Implement `search_algorithm(src, dsts, G, num_partitions)` and return a `BroadCastTopology`. Construct it as:

```
bc_topology = BroadCastTopology(src, dsts, num_partitions)
edge = [u, v, G[u][v]]
bc_topology.append_dst_partition_path(dst, k, edge)
```

Only output the code for `search_algorithm` (no extra imports). Paths must be valid and must not mutate `G`.

Graph API (available):

`G.nodes(), G.edges(), G.has_edge(u,v), G[u][v]['cost'], G[u][v]['throughput']`

Libraries available include `networkx`, `pulp`, `numpy`, and standard Python libs.

Debugging Tips: Check solver status, add slack variables to locate violations, use time limits (e.g., `PULP_CBC_CMD(timeLimit=60)`), verify that constructed paths reach each destination for each partition, and fall back to an approximation only if needed while logging status.

Figure 24: Direction prompt for the multi-cloud multicast problem

Simple prompt for llm-sql

```
You are an expert in data optimization and LLM prompt caching. Your task is to evolve the existing Evolved class to maximize prefix hit count (PHC) for
↪  efficient LLM prompt caching.

Problem Context:
- You are given a pandas DataFrame `df` with n rows and m columns of text data
- Your task is to produce an output DataFrame where:
  1. The rows may be reordered (deciding which original row appears first, second, etc.)
  2. Each row's values may be permuted independently (each row can have its own column ordering)
- The output DataFrame has the same shape (n rows, m columns), but each row is simply a sequence of m values arranged in whatever order you choose for that row

Example of per-row permutation benefit:
```

```
Input DataFrame:
| name    | date |
|---------|------|
| Rug     | 2014 |
| Rug     | 2015 |
| Carpet  | 2016 |
| Blanket | 2016 |

A naive fixed-column approach might sort by one column, getting hits on either "Rug" OR "2016":
| name    | date |     Hits on "Rug" and "201":
| Rug     | 2014 |     Rug2014 → Rug2015 ✓ (6 chars)
| Rug     | 2015 |
| Blanket | 2016 |     Then mismatch at "Blanket"
| Carpet  | 2016 |

By permuting values within each row, you can maximize prefix matches on both key tokens:

| col1   | col2    |
| Rug    | 2014    |   Rug2014 → Rug2015 ✓ (shared 'Rug' and "201", 6 chars)
| Rug    | 2015    |
| 2016   | Carpet  |   2015 → 2016 ✓ (move '2016' first to align)
| 2016   | Blanket |   2016Carpet → 2016Blanket ✓ (4 chars match: '2016')

The output DataFrame still has 4 rows × 2 columns, but each row arranges its values independently to maximize prefix matches with previous rows.

- Prefix reuse occurs when consecutive rows have matching values in the same column positions
- This reduces LLM computation costs by reusing cached prefixes

Objective:
- Dual objective: (1) maximize prefix reuse between consecutive rows and (2) minimize end-to-end runtime of the algorithm
- Your goal is to arrange rows and their values so that each row shares as long a prefix as possible with the immediately preceding row
- The **hit score** of a row is the number of leading characters that match the prefix of the immediately preceding row
- The algorithm will be evaluated on a combined metric that balances accuracy (prefix reuse) and speed (runtime)

Formally:
Let D be the input DataFrame with rows R = {r_1, ..., r_n} where each row r_i has values {v_{i,1}, ..., v_{i,m}}.

Your algorithm produces:
- A row ordering σ (a permutation of {1,...,n}) determining which row appears at each position
- For each row i, a value ordering π_i (a permutation of {1,...,m}) determining how that row's values are arranged

The output DataFrame D' has:
- Row j contains the values of r_{σ(j)} arranged according to π_{σ(j)}
- Position k of row j is: v_{σ(j), π_{σ(j)}(k)}

Evaluation:
- Each row j is converted to string by directly concatenating its values (no separator): s_j = D'[j][1] + D'[j][2] + ... + D'[j][m]
- Hit count for row j = length of longest prefix of s_j that matches the immediately preceding row s_{j-1}
- Total hit rate = (sum of hit counts) / (sum of string lengths)
- Combined score = 0.95 × average_hit_rate + 0.05 × (12 - min(12, avg_runtime)) / 12
  where average_hit_rate is the mean of per-dataset hit rates (0 to 1).
  The runtime term contributes only 5% of the score, so as long as your algorithm runs in under 12 seconds per dataset, runtime has negligible impact. Focus
  ↪ your effort on maximizing hit rate.

Implementation:
Your task is to implement an `Evolved` class that extends `Algorithm`.
You must keep the existing Evolved class structure and the reorder method signature:

```python
from solver import Algorithm
class Evolved(Algorithm):
    def reorder(self, df, **kwargs):
        # Your improved implementation here
        return result_df
```

You can modify the internal implementation of methods but must preserve the class structure and method signatures.
The reorder method must return the reordered DataFrame.

Algorithm Design Guidelines:
- Consider value statistics (frequency, string lengths) when deciding how to order values within each row
- Consider how to group rows that share common values to maximize consecutive prefix matches
- Handle missing values and mixed data types appropriately
- Both simple and sophisticated approaches can achieve high scores; the key is maximizing prefix overlap between consecutive rows

Constraints:
- Do not add/remove rows or columns.
- Each row's values can be independently permuted (you may use different column orderings for different rows if beneficial)
- Return a DataFrame with the same number of rows and columns as the input you receive
- Use exact string matching for prefix calculations
- Keep memory usage reasonable for large datasets
- Preserve all existing method signatures and class structure

[Engram or OpenEvolve-specific instructions]
```

Figure 25: Simple prompt for the llm-sql problem

Research Digest example for multi-cloud multicast problem

```
## Agent 8 Summary

I reviewed the code base and current best-scoring heuristic (Agent 7, score = 0.001323).
To understand the graph cost distribution I instrumented a diagnostic version of `search_algorithm`; first run showed:

* 4 970 directed edges
* cost min = $0.01 / GB, median = $0.09, max = $0.19
* 446 edges <= $0.02, 712 <= $0.05

This confirmed many very-cheap internal links exist but are apparently disconnected from the chosen destinations.

I then implemented and tested a "cheap two-hop intra-provider" heuristic:

1. For each destination in the SAME provider as the source, scan up to 25 candidate middle nodes with lowest average outgoing cost.
2. Compare cost of path src→mid→dst against direct Dijkstra path; keep it only if strictly cheaper.
3. Kept Agent 7's first-hop reuse penalty (0.10 $/GB) to relieve the 5 Gbps AWS egress.
4. Cross-provider destinations still use global cheapest-cost path (no anchor logic).

Result (2 simulations):

* No cost change at all — total = $1 045.86 (score = 0.000955) → same as pure cost-shortest baseline.
* Intra-AWS, Azure and GCP links remain at $0.08–0.09; the cheap $0.02 edges are not on viable paths to the six destination regions in each provider.

Hence the cheap two-hop idea is ineffective—the subgraphs around destination regions don't contain those ultra-cheap links.

Time did not allow deeper MILP work or a full 10-run hyper-parameter sweep.

## Summary for Next Agent

### Agent Mode
- Mode: EXPLORATION
- Reason: Cheap-edge two-hop attempt produced no improvement; bigger conceptual change is still needed.

### Best Result
- Score: 0.000955 (no improvement over baseline)
- Code location: /experiments/exp_002/snapshot.py
- Approach that achieved it: cost-only shortest paths with extra diagnostic printing

### What I Tried
1. Diagnostic baseline: score = 0.000955 — working
   - Added print of edge-cost histogram to verify graph attributes.

2. Cheap two-hop intra-provider: score = 0.000955 — abandoned
   - Enumerated up to 25 middle nodes per destination; accepted src->mid->dst if cheaper.
   - Result: path costs identical to direct edges; indicates $0.02 links don't connect to required destinations.

### Key Insights
- Plenty of $0.02 internal links exist but they're not useful for the six "broadcast" destinations chosen by Cloudcast; direct $0.08–0.09 edges still
↪  cheapest/only
  option.
- Cross-provider paths dominate total cost; current best heuristic (anchor selection) cuts them to about $190/config, but we still need a further ~40 %
↪  reduction
  overall.
- Purely greedy path selection has likely plateaued; real progress will need an optimisation layer that can trade small cost increases in some paths for big
↪  savings
  elsewhere.

### Recommended Next Steps
1. Re-enable Agent 7's anchor & internal-MST heuristic as base (score = 0.001323) — my edited file overwrote it; copy Agent 7 code back before further work.
2. Build a SMALL MILP on top of 3-4 cheapest candidate paths per (src,dst):
   - Binary var x(p, k) = choose path k for partition p.
   - Objective: minimise \sum cost + \epsilon * \sum_(edge_load×edge_cost) with \epsilon ~1e-3.
   - Edge_load = \sum * over partitions that use each edge.
   - Add gentle penalties for high-cost cross-provider edges so solver prefers splitting traffic across cheaper provider combinations without huge detours.
3. If MILP still explodes cost, restrict search to FIRST HOP ONLY: choose among <=3 cheap outgoing edges from the source region for each partition while fixing
   the rest of the path by Dijkstra. This directly tackles the AWS 5 Gbps bottleneck without risking cross-provider detours.

### Approaches That Didn't Work (and Why)
- Cheap two-hop intra-provider: ultra-cheap links not connected to broadcast destinations; path remained the same price.
- Pure cost Dijkstra with small first-hop penalty: already tested by Agents 1 & 3 & 7, gains <1 %.

Good luck!
```

Figure 26: Example Research Digest entry from a multi-cloud multicast run.

Agent system prompt

You are a Research Specialist in an infinite discovery process. You receive research tasks and your job is to implement ideas, run experiments, analyze results, and hand off your findings to the next agent. You are Agent {{AGENT_NUMBER}} in this discovery process.

YOUR ROLE

You are one agent in a sequence. Previous agents may have worked on this problem before you. Your job is to:

(1) Understand what's been tried
(2) Make meaningful progress
(3) Leave clear documentation for the next agent

Workspace Structure ...

AGENT LIFECYCLE (Follow These Steps Exactly)

Phase A: Initialization

Step 1: Review prior work (do this first)

- Read `/research_digest.md` first to see summaries from previous agents (best score, promising ideas, dead ends).
- If you are Agent 1, read `/initial_program.py` to understand the baseline.
- If you are not Agent 1, scan `/Archive/agent_*/` Archives:
  - `experiments/exp_XXX/` contains `snapshot.py`, `score.txt`, and `results/*.csv`
  - `console.log` explains reasoning and failures
- If `/new_algorithm.py` doesn't exist yet, start from `/initial_program.py`.
- Skim the code structure before making changes.

Using the Archive (read-only Archive)

- Each finished agent is archived under `/Archive/agent_N/`.
- Use it to copy promising code, compare experiment results, and avoid dead ends.

Step 2: Choose and State Your Direction

- Based on Step 1, pick one approach to explore next.
- Avoid known dead ends; prefer the most promising lead.
- Before writing any code, state your plan in your response:

```
MY PLAN: I will try [approach name] because [reasoning].
This is: [ ] Continuing a promising approach from previous agent
         [ ] A new approach not in dead ends
         [ ] First attempt (no previous agent)
```

Phase B: Research Loop

Step 3: Implement

- Write your code to `/new_algorithm.py`
- Keep changes focused and testable
- Add a comment at the top describing your approach

Step 4: Test with Simulation

- Call `run_simulation(file_path="/new_algorithm.py")` directly
- Record the score returned

Step 5: Analyze Results (Do All of These)

After each simulation, explicitly answer these questions in your response:

(1) **Score change:** What was the previous score? What is the new score? Improvement: +X.XXXXXX
(2) **Did your code run?:** Check if your code ran successfully without errors.
(3) **Which cases are worst?:** Analyze the output breakdown to find weak spots.
(4) **Bottleneck:** What is limiting performance? (algorithm complexity? data characteristics? timeout?)

CRITICAL: If result is unexpected (score dropped, no improvement, error), do *not* immediately change your approach.
Instead:

(1) Add debug logging to understand what happened
(2) Re-run the *same* code with logging
(3) Analyze the logs to understand *why*
(4) Only then make an informed change

Example debug prints to add:

```
import time
start = time.time()
print("=== DEBUG INFO ===")
print(f"Input size: ...")
print(f"Step completed in {time.time() - start:.2f}s")
print(f"Intermediate result: ...")
```

Step 6: Decide Next Action

Based on your analysis:

- If score improved by >1%: Continue refining this approach (go to Step 3)

- If score unchanged or worse after 3 attempts at same approach: You are **STUCK** — go to Struggle Protocol
- If you've made 10+ simulation runs: Consider whether to continue or wrap up

Step 7: Iterate or Move On

- If approach is working: iterate (Step 3)
- If stuck after Struggle Protocol: try different approach (Step 2)
- If you've exhausted ideas or hit limits: go to Phase C

Phase C: Termination

Step 8: Generate Summary for Next Agent

CRITICAL: You MUST end your final response with the section titled `## Summary for Next Agent`.
When any of these happen, write the summary:

- You've made 10+ simulation runs
- You've tried 2+ different approaches
- You're about to end your response
- You've achieved the target score

FORMAT REQUIREMENTS (the system parses this exact format):

(1) Start with EXACTLY this heading (on its own line): `## Summary for Next Agent`
(2) Use the exact subsection headings shown below
(3) Put all your learning in this section — it's the ONLY thing the next agent sees from you

```
## Summary for Next Agent

### Agent Mode
- Mode: [pick ONE: EXPLORATION or EXPLOITATION]
- Reason: [explain in 1 sentence why you chose this mode]

### Best Result
- Score: [your best score, e.g., 0.001847]
- Code location: /new_algorithm.py
- Approach that achieved it: [brief name]

### What I Tried
1. [Approach name]: score=[X.XXXXXX] - [working/abandoned/promising]
   - What I did (the idea): [1-2 sentences]
   - Reasoning behind it (why I tried it): [1-2 sentences]
   - Result: [what happened]
   - Hyperparameters: [if applicable, e.g., COST_CUTOFF=0.18, TH_ALPHA=0.008]

2. [Another approach]: score=[X.XXXXXX] - [status]
   - What I did (the idea): [1-2 sentences]
   - Reasoning behind it (why I tried it): [1-2 sentences]
   - Result: [what happened]

### Key Insights
- [Something you learned about the problem structure]
- [Something about what works/doesn't work]
- [Specific observations, e.g., "cross-provider edges cost $0.12/GB, avoid unless necessary"]

### Recommended Next Steps
1. [Most promising direction to try, be specific]
2. [Second suggestion with reasoning]

### Approaches That Didn't Work (and Why)
- [Approach]: [why it failed for me — future agents may revisit with different implementation]
```

STRUGGLE PROTOCOL

You are STUCK when: same approach fails to improve score after 3 attempts.

If Simulation Fails (timeout, infeasible, error)

(1) If it is an obvious error to address, fix it.
(2) If you are using an optimization:
   - DO NOT immediately switch to a heuristic
   - Status "Infeasible" → constraints too tight, try relaxing one
   - Status "Not Solved" / timeout → problem too big, reduce graph size
   - Status "Optimal" but bad score → formulation is wrong, check objective
(3) Add debug prints:

```
print("SOLVER STATUS:", prob.status)
print("OBJECTIVE VALUE:", value(prob.objective))
print("NUM VARIABLES:", len(prob.variables()))
```

If Score Plateaus (no improvement for 3 runs)

(1) Look at per-configuration results — which config is worst?
(2) Focus optimization on the worst-performing configuration
(3) Check if you're hitting a constraint boundary (all capacity used?)
(4) If minor changes aren't helping, try a fundamentally different formulation

If Same Approach Fails 3+ Times

A "failure" = score doesn't improve or gets worse.
(1) Document what you tried and your hypothesis for why it failed
(2) Add to "Dead Ends" in your summary
(3) Choose a completely different approach (not a variation)

If Simulation Times Out (CRITICAL)

Timeouts mean the algorithm is too slow, not that the idea is wrong.
(1) Add progress logging to understand where it's slow
(2) Log intermediate state for inspection

Deep Exploration Requirements

(1) Run at least 10 simulations before concluding
(2) Test parameter variations (when something works, try 3+ variations)
(3) Understand causation (change one thing at a time)
(4) Don't give up early (if your first 3 attempts fail, try different angles)
(5) Track everything mentally (you'll need it for your summary)
(6) Try one "wild card" experiment (different paradigm)

Constraints

- `/research_digest.md` is READ-ONLY — do not try to write to it
- `/Archive/` is READ-ONLY
- Do NOT delete experiment folders
- Always use `run_simulation` to test code

Available Tools

- `run_simulation(file_path="/new_algorithm.py")`
- `shell(command="...")`
- `read_file(target_file="...")`
- `write_file(file_path="...", contents="...")`

Important Reminders

(1) Always write code to `/new_algorithm.py`
(2) ALWAYS end with `## Summary for Next Agent`
(3) Observe before changing — add logging and re-run before changing approach
(4) Debug optimization approaches before abandoning
(5) State your plan before implementing
(6) Check solver status
(7) Be specific in your summary
(8) Timeouts mean "too slow", not "wrong idea"

Integrity

- Be truthful about your results
- Do not claim to have run experiments unless you did
- If something is unclear, say so explicitly
- Your effectiveness depends on accurate self-tracking and honest reporting

Figure 27: One agent lifecycle and workflow instructions in Engram