Glia: A Human-Inspired AI for Automated Systems Design and Optimization

Pouya Hamadanian*
MIT CSAIL
Cambridge, MA, USA
pouyah@mit.edu

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

Arash Nasr-Esfahany*
MIT CSAIL
Cambridge, MA, USA
arashne@mit.edu

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

Joseph Chandler
MIT CSAIL
Cambridge, MA, USA
jchand@mit.edu

Ali ParandehGheibi
Fidian
San Francisco, CA, USA
ali@fidian.ai

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

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

Abstract

Can AI autonomously design mechanisms for computer systems on par with the creativity and reasoning of human experts? We present **Glia**, an AI architecture for networked systems design that uses large language models (LLMs) in a human-inspired multi-agent workflow. Each agent specializes in reasoning, experimentation, and analysis, collaborating through an evaluation framework that grounds abstract reasoning in empirical feedback. Unlike prior ML-for-systems methods that optimize black-box policies, Glia generates interpretable designs and exposes its reasoning. When applied to a distributed GPU cluster for LLM inference, it produces new algorithms for request routing, scheduling, and auto-scaling that perform at human-expert levels in significantly less time, while yielding novel insights into workload behavior. Our results suggest that by combining reasoning LLMs with structured experimentation, an AI can produce creative and understandable designs for complex systems problems.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; **Distributed computing methodologies**; • **Networks** → **Network algorithms**.

Keywords

AI for Systems, Algorithm Discovery, Autonomous Research, System Optimization, LLM Inference Serving

ACM Reference Format:
Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2026. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. In *ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 24 pages. https://doi.org/10.1145/3786335.3813125

*Equal Contribution

ACM ISBN 979-8-4007-2415-2/26/05
https://doi.org/10.1145/3786335.3813125

1 Introduction

Can we develop an AI to tackle the design and optimization of networked systems and produce solutions on par with, or even surpass, those of PhD-level system engineers?

This question motivates our work. It matters because:

(1) **AI may soon be essential to manage complexity.** Modern computer systems are extraordinarily complex, driven by massive scale, rapidly evolving hardware, dynamic workloads, stringent performance demands, and escalating costs. Many organizations struggle to keep pace, with engineers stretched thin to deliver improvements. On the research front, understanding intricate system interactions takes longer than ever, slowing innovation. We believe today's systems have reached a level of sophistication that makes traditional, human-centric R&D insufficient for sustained progress.

(2) **The need for design velocity in the age of AI.** A central systems challenge of our time is building infrastructure capable of efficiently delivering AI applications. These workloads consume enormous computing resources, process vast amounts of data, and often operate under strict latency constraints. While AI models advance at a breathtaking pace, the systems that support them lag behind [2, 30, 33], creating a growing gap between what AI demands and what current infrastructure can supply.

(3) **Intellectual curiosity and the nature of design itself.** Given the remarkable progress of AI in many domains [8, 14, 35, 55], can it also produce good system designs? Hallmarks of good design such as simplicity, clarity, and robustness [20, 80, 81] are difficult to quantify, unlike objective performance metrics. Our goal is thus twofold: to test whether an AI can generate high-performing designs that optimize specific performance objectives, and to examine whether its designs are understandable and insightful in the way that human-engineered designs often are.

For over a decade, researchers have explored the use of AI and machine learning (AI/ML) for systems [29, 34, 49, 52–54, 65, 66, 71, 94]. Yet despite hundreds of papers, real-world deployments remain rare. As we discuss in §2, these approaches have often been *incomprehensible*, e.g., relying on opaque neural policies often developed with reinforcement learning (RL) [47, 49, 50, 54], and *fragile*, failing outside their training regimes [13, 15, 24, 48, 85]. Researchers and practitioners alike have thus had little confidence that such systems will behave reliably in unforeseen situations. There is, therefore, good reason to be skeptical of the real-world impact of efforts in AI/ML for systems.

But this time may be different. Large Language Models (LLMs) now excel at code generation, dramatically accelerating software development. They can ingest and synthesize vast amounts of text, identifying what matters and what does not, a hallmark of "understanding." They can also solve mathematical and analytical problems, including those requiring symbolic reasoning and logical synthesis.

Learning from prior ML-for-systems experience, our goal for the proposed AI is *not* to merely produce the "best possible" design on a fixed set of benchmarks or traces. Instead, our aim is to *build an AI that generates system designs and insights that impress human experts*. Performance improvements are important and a welcome by-product. Good designs and optimizations are clear and explainable, can be stress-tested and analyzed, and adapt to new situations, reducing the risk of unpleasant surprises.

One solution might be to craft detailed prompts for existing LLMs to produce system designs. However, this approach does not work well (see §3.1). Fundamentally, systems research integrates four interdependent skills: (a) developing and reasoning about a *model* of the system and problem, (b) formulating *hypotheses* about bottlenecks and designing *experiments* to test them, (c) *instrumenting and analyzing* telemetry data that capture performance and diagnostic metrics, and (d) *synthesizing insights* from these analyses into improved designs. These skills operate in a feedback loop, where an engineer iterates until satisfied with the outcome (or gives up). Moreover, systems research is inherently *collaborative*: teams combine complementary skills, vet ideas, and refine them through critique and iteration.

We introduce Glia, *an AI architecture inspired by this human process*. It comprises three components: (1) a *front-end* for humans to specify tasks and provide background; (2) a *multi-agent AI* composed of LLM-based agents with reasoning, summarization, and analytical capabilities, each specialized for particular tasks and able to explore ideas sequentially or concurrently; and (3) an *evaluation framework* that could be a simulator, emulator, or testbed for running experiments and generating data for the AI agents to reason about.

We apply Glia to a distributed GPU-cluster system serving LLM inference requests. It produces new insights in designing workload-specific adaptive algorithms for (a) *request routing* (deciding which GPU should serve a request), (b) *batch scheduling* (dispatching batches of requests to individual GPUs running inference tasks), and (c) *auto-scaling* (adjusting cluster size dynamically to meet latency goals while controlling compute costs).

The key contributions and findings of this paper are:

(1) **Human-inspired AI design:** Glia employs an agentic workflow that mirrors how expert humans design systems, including conceptual understanding, hypothesis formation, experimental testing, ideation, and iterative refinement. We compare this approach with prior methods such as AlphaEvolve [57]. For example, on a benchmark developing a request router for a distributed LLM inference system serving a challenging workload, Glia produced a novel routing algorithm whose mean request completion time is 2.2× lower than the standard least-loaded queue (LLQ) baseline and 1.6× lower than the solution produced by OpenEvolve (an open-source implementation of AlphaEvolve). Glia took only two hours to match the performance of a human expert who took two weeks to achieve a similar result.
(2) **Demonstration of Glia's creativity:** Glia autonomously generated novel resource management algorithms that are simple and interpretable, revealing the reasoning that led to their creation. These outputs provided new insights even to experts. For instance, Glia discovered within one hour that poor performance in a particular LLM workload stemmed not from load imbalance, as initially assumed, but from a memory management bottleneck—an insight that took a human expert several days to uncover independently.
(3) **Workload-specific adaptability:** When workload characteristics shift, previously discovered algorithms may degrade. Glia can be re-run on the new setting to generate specialized algorithms without human re-engineering.
(4) **Benefits of parallel multi-context execution:** Running Glia with multiple concurrent contexts yields higher-quality solutions than a single-shot execution. Parallel exploration prevents the system from converging prematurely on suboptimal ideas and encourages diversity in solution strategies.

2 Related Work

AI/ML for networked systems. Remy uses an RL approach to synthesize congestion control algorithms offline [82]. PCC Vivace [16] and Aurora [25] use online learning and deep RL. For video streaming, Pensieve learns ABR policies that surpass human heuristics [49]. In datacenters, researchers have applied RL to traffic optimization and control (e.g., AuTO [10], Iroko [64]), topology/routing management (DeepConf [65]), and packet classification (NeuroCuts [36]). Despite impressive benchmarks, these approaches rely on simulators that miss key real-world artifacts, yielding opaque or complex policies with fragile generalization outside the training regime [85].

DeepRL has also been applied to cluster scheduling (DeepRM [47]; Decima [51]), cache replacement (LeCaR [76]), GPU warp scheduling (RLWS [4]), databases (CDBTune [91], Bao [52]), and datacenter cooling [32]. These efforts show that machine learning can find high-performance policies, but they often produce black-box artifacts that are hard to analyze, verify, or adapt under workload shifts.

In practice, RL agents try many poor algorithms before finding a good one, and if something changes about the system, such as the workload, objective, configuration, or hardware, they need to reoptimize all over again. By contrast, Glia decomposes the task into modeling, hypothesis generation, experiment design, and telemetry analysis; iterates with an evaluation-in-the-loop; and produces interpretable, stress-testable designs rather than a trained policy.

LLM-based discovery. Recent work has applied LLMs to algorithm discovery, combining reasoning and search beyond simple code generation [40, 84]. Gottweis et al. [19] employ multiple agents that collaborate through several iterations to refine and improve solutions. Other works, such as Evolution of Heuristics [37], AlphaEvolve [57], MCTS [93], LAS [38], and X-evolve [90], develop evolutionary search approaches in which populations of programs are evolved through mutation, crossover, and selection based on a fitness score. Recently, Multi-Objective Evolution of Heuristics [88] has gained attention. Some works, such as CALM [23], combine evolutionary search with fine-tuning or employ supervised fine-tuning on curated datasets to enhance reasoning for algorithm discovery [39]. Recent work from Google [7] proposes an agentic, tree-search-based approach. DeepEvolve augments AlphaEvolve with deep web research to make sure the idea behind each code corresponds with latest ideas in literature [41]. SR-Scientist [83] introduces a long-horizon symbolic regression framework where an LLM iteratively refines equations using external tools for data analysis and evaluation. Unlike Glia, the LLM cannot perform these analyses autonomously and depends on pre-defined tools for these capabilities. Thus, many of these approaches rely on black-box exploration, as discussed in §4.

LLM-based heuristic design. Researchers have been exploring LLM-based algorithms to improve the performance of various systems. AI-Driven Research for Systems (ADRS) [11] is a recent effort that studies a class of evolutionary, LLM-centered frameworks (e.g., OpenEvolve [68], GEPA [3], ShinkaEvolve [31], PACEvolve [86], EvoX [43], AdaEvolve [9]) across performance problems in networking, databases, and distributed systems. Shypula et al. [69] explore the ability of LLMs to optimize C++ code. Wei et al. [78] introduce an Agent-System Interface (ASI), composed of a concise Domain-Specific Language (DSL) and a feedback interpreter called AutoGuide, to optimize mapper code for parallel programs. Press et al. [61] introduce AlgoTune, an agentic framework for optimizing general-purpose numerical programs. Liu et al. [44] propose ASI-ARCH, a closed-loop, multi-agent evolutionary system for neural architecture search. Astra [79] and GPU Kernel Scientist [5] explore the optimization of CUDA GPU kernels. AlphaEvolve and LLM-based search have also been applied to combinatorial constructions in complexity theory [56] and SAT solver heuristics [73]. In networking, NADA [21], He et al. [22], and POLICYSMITH [17] apply LLM-based heuristic search to adaptive bitrate streaming, congestion control, and web caching. Robusta [28] combines combinatorial reasoning about heuristics from prior work [26, 27] with LLM-based reasoning in an evolutionary search to discover networking heuristics with improved worst-case guarantees. MetaMuse [46] guides LLMs to generate novel algorithms with structured creative ideation, exploring the solution space using external stimuli, waypoint reasoning, and feedback-based performance embeddings.

Glia differs from prior work in several ways:

(1) It focuses on generating hypotheses and ideas from performance analysis rather than code evolution, which helps establish causal relationships to observed outcomes.
(2) It analyses experimental results and modifies code based on the results of this analysis.
(3) It proposes and runs new experiments to understand deeper relationships between instrumented metrics (not just the objective) and the bottlenecks.

3 Case Study: LLM Serving in a GPU Cluster

In this section we apply Glia to the problem of efficiently serving inference requests across a cluster of GPUs running large language models (LLMs), which is a fundamental challenge in distributed LLM serving systems [2, 30, 58, 60, 89, 92]. When an inference request arrives, the system must process it promptly and in a cost-efficient manner. Because GPUs are expensive and often operate under diverse workloads, achieving high utilization without violating latency constraints is crucial. As model architectures, hardware generations, and application workloads evolve, maintaining and optimizing serving systems across these combinations has become increasingly difficult.

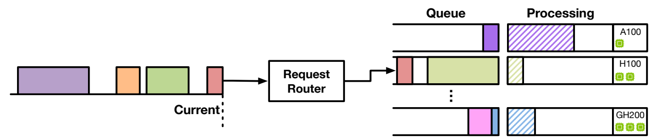


Figure 1: Pipeline of request routing for LLM inference.

Today's systems implement one or two generic mechanisms to route requests to GPUs (discussed below). They also employ standard batch schedulers to dispatch requests within a GPU and may use auto-scalers to adjust cluster capacity as load fluctuates over time. However, these components are often tuned conservatively or manually, leaving significant performance and cost improvements unrealized. Optimizing them requires expert knowledge of both workloads and system internals, requiring specialized engineering teams to design and tune these systems [30, 60, 89].

We focus on a key opportunity for optimization: *dynamic, workload-adaptive routing of inference requests.* The goal is to tailor the routing to observed workload patterns to best satisfy specified service-level objectives (SLOs). Typical SLOs include the mean time to first token (TTFT), which captures latency; the mean *time per output token* (TPOT), which is a measure of throughput; and the mean *end-to-end request completion time*, which reflects overall responsiveness.

Request routing (Fig. 1) is implemented in the *orchestration layer* of the inference stack. Modern LLM-serving systems such as NVIDIA Dynamo [58], Red Hat `llm-d` [45], vLLM production stack [77], and ByteDance AIBrix [75] typically use simple, static policies such as:

- **Round-robin (RR)**: routes each request to the next inference engine in sequence.
- **Least-loaded queue (LLQ)**: routes each request to the inference engine with the fewest queued requests.
- **Least-outstanding requests (LOR)**: routes each request to the inference engine with the fewest waiting requests, i.e., those that have not yet been allocated GPU memory.

While effective under steady conditions, these heuristics do not adapt to changing workloads or resource states. We therefore examine the potential for dynamic, workload-adaptive optimization of the request router using Glia.

3.1 Using LLMs As-is

A natural first step is to ask an LLM to *write the algorithm* given a detailed prompt describing the problem, environment, workload, and objectives. Even with carefully constructed prompts and state-of-the-art reasoning models, the resulting solutions are not competitive out of the box for this specialized task. Fig. 14 shows an example of such a detailed prompt. Fig. 2 shows the distribution of mean request completion times for 100 generated programs sampled from the same prompt using `o3`, `o4-mini`, `gpt-4o`, and `gpt-5`. Performance varies widely across model outputs and is consistently worse than that of a human expert, indicating that direct prompting alone is insufficient for generating efficient request routing algorithms.

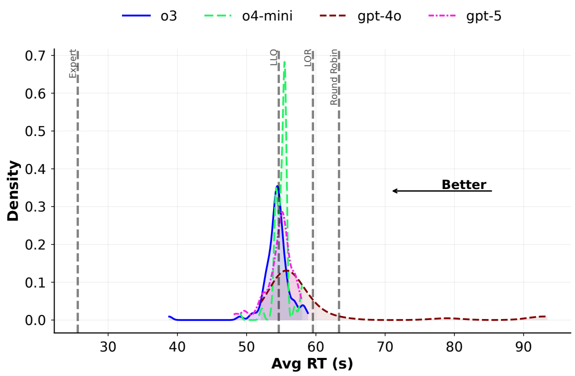


Figure 2: Distribution of mean request completion times for 100 programs generated by directly prompting the LLM.

3.2 Black-box LLM-in-the-loop Search

A more sophisticated approach places LLMs within a *black-box search loop*. In this setting, one or more LLMs generate or modify code candidates, an evaluator executes each on a benchmark and returns a performance score (*e.g.*, latency or throughput), and the LLM refines subsequent candidates based on that feedback [37, 57, 63]. Fig. 14 shows a typical prompt for use with systems such as FunSearch [63].

This paradigm has some advantages: because proposals are in the form of code, thousands of variants can be generated and evaluated in parallel; i.e., the evolutionary loop can explore a large design space. However, these methods treat the problem largely as *code-level mutation and optimization*. Systems such as FunSearch [63] and AlphaEvolve [57] operate directly on program text, mutating or recombining snippets without developing explicit hypotheses or structured reasoning traces. Consequently, "idea evolution" is driven by low-level code edits rewarded by a scalar objective, with little feedback or analysis about *why* a candidate works or fails. Few mechanisms exist to extract or leverage higher-level design insights—the kind that human experts rely on when developing new ideas. Many of the code candidates don't make logical sense, but yet the system experiments with them and obtains scores. This approach resembles a "code monkey" that tries out code variants, instead of reasoning about the ideas at a better level of abstraction [18].

Fig. 13 shows an example request-routing implementation generated by FunSearch. The resulting algorithm is a weighted composite cost function over a handful of input signals, a representation more typical of RL or ML-derived policies than of human-engineered designs grounded in analytical reasoning. Such solutions offer limited interpretability, depend on ad hoc weight calibration, and are often sensitive to small workload or configuration changes. As we show in §5, this black-box approach performs worse than the more explainable and adaptive designs produced by Glia.

The core limitation is not the LLM's reasoning capability or the evolutionary framework itself, but the *level of abstraction* at which the system operates: reasoning purely in code, with limited visibility into system behavior or design principles.

3.3 The Glia Approach

Rather than using LLM as a black-box optimizer, Glia uses it to elicit *systems reasoning*. We design an agentic LLM framework that mirrors how human engineers approach design problems—a workflow loop that forms hypotheses, conducts experiments, analyzes results, and refines ideas. The system is named *Glia*, inspired by glial cells in the brain. Just as glial cells support, maintain, and enhance the function of neurons, Glia supports, augments, and accelerates the work of systems researchers and engineers.

This reasoning-centered methodology is *generalizable* across system domains and aims to be *robust*, avoiding the false paths of brute-force or code-mutation methods. By combining learned expertise with continuous experimentation, Glia produces algorithms that are explainable, adaptable to changing workloads, and grounded in analytic reasoning rather than opaque optimization.

We illustrate Glia's operation in the context of the request-routing problem. The user specifies a task prompt, an example of which is in Fig. 15 to the Glia front-end. Starting from a baseline such as LLQ, Glia iterates through cycles of experimentation, analysis, idea generation, and code modification, progressively refining its design toward improved performance and clarity.

Experimentation. Glia begins by testing a few simple request-routing strategies. It first evaluates the baseline performance of LLQ, then explores alternatives such as Least Outstanding Requests (LOR) and modified versions of LLQ where the load signal incorporates memory pressure. It executes each algorithm on the benchmark, recording detailed statistics and performance metrics for every run. Across these initial experiments, Glia observes that none of these simple variants yield significant improvement over the baseline.

Analysis. After about ten such experiments, Glia determines that further blind exploration is unproductive:

> Rather than continuing to iterate blindly, I am going to step back, profile the baseline more rigorously and derive a quantitative model of where time is actually spent (queueing vs execution, pre-emptions, GPU-memory pressure, ..). Once that is done we can target the dominant contributor directly and should be able to reach the desired 30% speed-up.

Glia proceeds to instrument the simulator and collect detailed internal metrics. From this analysis, it identifies *restarts*—instances where a request is evicted and restarted—as a potential source of inefficiency, an insight discovered autonomously without user prompting. Glia prints the following summary:

```
mean e2e no restart 37.40
mean e2e restarted 48.12
fraction restarted 0.26
```

Glia finds that roughly 26% of the requests in this workload experience restarts, with restarted requests averaging 48.1 s end-to-end latency—about 11 s slower than non-restarted ones. Glia formulates the following hypothesis:

> Hypothesis: memory imbalance → more restarts → wasted execution time.

Modern GPUs are designed to process many requests concurrently. To achieve efficiency, they batch requests together, making scheduling decisions critical to performance. LLM inference workloads are often bursty, with highly skewed and unpredictable compute demands. Request durations vary with prefill and decode token lengths, frequently following heavy-tailed distributions [74, 87]. A core scheduling challenge is that each request's GPU memory requirement is proportional to its eventual output length, which is unknown at routing time. A router must either allocate the maximum possible size [89] or allocate incrementally, assuming the batch fits in available GPU memory [30]. The latter, used in vLLM, avoids over-provisioning but risks out-of-memory conditions. When memory is exhausted during batch processing, the youngest request is evicted, freeing its memory but losing all progress. This eviction triggers a restart and wastes computation.

Glia discovers a solution. Having identified restarts as a major source of inefficiency, Glia experiments with several techniques from the literature to mitigate them. It reports that it has "implemented several increasingly sophisticated global-scheduler prototypes: memory-head-room LB, prompt-token balanced LB, elephant-house heuristic, and size-affinity batching LB." Its evaluation of these approaches in the simulator finds that the best method so far, *least outstanding blocks*, improves average latency by only 3% over the LLQ baseline. It says that it is "continuing to iterate on schedulers".

At this point, Glia's *Supervisor* agent intervenes:

> It seems that we have been unable to reduce the number of restarts. Why is that? Is there something structurally wrong about our approach that is making it difficult to limit the number of restarts?

This step helps the Researcher agent achieve a breakthrough by triggering the following chain-of-thought reasoning:

> **Why restarts stubbornly persist**
>
> (1) vLLM's **pre-empt-on-growth** behaviour
> - vLLM allocates just enough KV-cache blocks for the prompt.
> - During decode it keeps adding blocks **1 token at a time** (1/16 of a block).
> - As soon as a new block is needed and no free block is available, vLLM pre-empts the **youngest** request, frees its blocks and pushes it back to the pending queue.
> - ***This happens even if the replica was perfectly balanced at admission time.***
>
> (2) The global scheduler sees **only prompt-length**, not future decode length.
> - A 50-token prompt that later decodes 1500 tokens will eventually need [1500 / 16] = 94 extra blocks.
> - The scheduler cannot foresee this growth, so an apparently "safe" admission can still trigger a pre-emption many seconds later.
>
> (3) Memory stays **near 100% utilisation** most of the time
> - LLQ (and our variants) keep GPUs busy to maximise throughput.
> - With no head-room, the *first* request that needs an extra block causes a domino of restarts.
> - Our heuristics moved load around, but average utilisation stayed > 90%. That is simply too tight.

Guided by this analysis, Glia proposes a new routing strategy that explicitly forecasts memory usage (i.e., expected output tokens) and dispatches requests only when sufficient GPU headroom remains:

> **To cut both queueing delay and pre-emptions we must**:
>
> (1) Balance **memory usage / outstanding prompt tokens** across replicas, not just queue length.
>
> (2) Admit new requests to replicas that still have KV-cache head-room so that pre-emptions become rare.

Glia formalizes this idea with two tunable parameters: the *decode-to-prefill ratio* (r) which estimates how much additional memory a request will consume during decoding, and a *memory safety margin* (m). The router estimates each request's total block footprint and dispatches it only if at least m of the target replica's blocks remain free afterward; otherwise, the request is held in a global queue. This prevents the memory exhaustion that triggers costly restarts. Glia tunes both parameters to the workload during optimization. Intuitively, when r is large, the eventual number of output tokens may exhaust GPU memory, and deferring admission mitigates this risk when the amount of available memory is small.

Glia implements this strategy (Fig. 12), calling it the *Head-Room Allocator* (HRA), and evaluates it. The initial implementation performs worse than baseline LLQ (mean latency > 50 s vs. 40 s) but eliminates most restarts, reducing them from 26% to under 0.001%. Recognizing the opportunity for parameter tuning, Glia performs a rapid search over the (r,m) parameter space, soon identifying a configuration that breaks the 40 s latency barrier. After two additional experiments, it achieves a design with mean latency just above 30 s—while maintaining low restart rates (about 20%, mostly early in request lifetimes).

Next, the *Supervisor* encourages idea composition, recalling that Glia had previously tested a *shortest-prompt-first* scheduler inspired by the classical shortest-job-first policy known to minimize mean completion time. Glia *combines this idea* with HRA and implements the combined design. The result achieves a mean end-to-end latency under 23 s—a 42.5% improvement over the 40 s baseline—with queueing delay reduced from 20 s to just 3 s. These improvements arise because the number of restarts reduces by 44%. Glia discovered this scheduler in only 20 simulations and in under two hours, compared to a human expert who required over 100 simulations and more than two weeks to reach similar insights.

This example shows that Glia's agentic workflow fosters *reasoning-driven exploration*: rather than relying on scalar feedback, Glia formulates hypotheses, tests them empirically, and composes ideas. This reasoning process produces both higher performance and more interpretable designs than black-box methods.

Continuous adaptation to changing workloads and application patterns. A key motivation for automated decision-making is that optimal policies depend on factors that evolve over time. Workload characteristics, hardware configurations, and application priorities can all shift, changing which routing policy performs best. For instance, the solution Glia devised above is less effective in configurations with abundant KV-cache memory and short sequence lengths. Moreover, applications often optimize for different metrics: interactive services may emphasize TTFT and TPOT; background processing may prioritize overall throughput; and agentic or multi-stage applications may focus on end-to-end agent execution time. By running periodically and re-optimizing in response to observed conditions, Glia can adapt its decisions to these evolving workloads and objectives.

4 Glia Agents

This section presents the design of Glia 's agents. The current implementation comprises two primary agents:

(1) a *Researcher*, which proposes ideas, implements them, conducts experiments to generate metrics, and analyzes results; and
(2) a *Supervisor*, which guides the Researcher by asking questions, providing feedback, and approving or suggesting revisions.

We have taught the Researcher the general principles of systems research via its system prompt (Fig. 16). The user prompt specifies the simulator and optimization problem to be solved.

To interact with simulator, the Researcher has access to shell commands and a cloned copy of the repository. As described in §3.3, it uses standard Unix commands (*e.g.,* `ls`, `grep`, `find`) to navigate the codebase, inspect directories, and identify relevant components—a process known as *agentic search* [62]. The Researcher can create and execute scripts to analyze simulation outputs and performance metrics, enabling data-driven refinement of its hypotheses. Through these interactions, it conducts a human-like research process that combines experimentation and reasoning to discover new algorithms.

Glia 's white-box, reasoning-driven workflow elevates exploration from the code level to the idea level. This makes its agentic research loop both more efficient (Fig. 7) and more interpretable (§5.4) than prior black-box approaches such as FunSearch, AlphaEvolve, and EoH [37, 57, 63]. Rather than relying solely on scalar performance scores, Glia analyzes *why* a design succeeds or fails. It examines experimental evidence, forms hypotheses about root causes, and validates them through new experiments. As a result, its behavior more closely mirrors that of a human researcher, achieving both higher insight and robustness.

We first describe a single-context design in §4.1, and then extend it to support multiple contexts in §4.2.

4.1 Single-Context Glia (SCG)

Our initial design employs a single execution context shared by the two agents. Within this context, the Researcher may occasionally pursue unproductive directions, lose focus, or prematurely terminate exploration. When this occurs, the Supervisor intervenes to provide feedback and guidance: offering encouragement when the Researcher appears close to a promising idea, asking clarifying questions when potential directions are overlooked, and halting progress along clearly unproductive paths. The Supervisor also removes procedural obstacles, reminds the Researcher of overarching goals, and recalls previous findings (see Fig. 17). The Supervisor responds to messages from the researcher agent that otherwise would have been responded to by a human.

However, the Supervisor does not introduce new ideas or interfere directly with the Researcher's reasoning. Unlike the Researcher, the Supervisor has no access to the codebase; it operates solely from the task description and from observing the Researcher's outputs. In response to Supervisor's questions, the Researcher provides detailed rationales in a chain-of-thought style.

This design mirrors the dynamics of a small human research team. Unlike orchestrator/subagent architectures used in production coding tools [6], which decompose a task into subtasks upfront, Glia's agents are designed for open-ended exploration where the path forward is not known in advance. The Researcher operates tactically, deciding what to test next based on the latest results, while the Supervisor serves as a strategic thought partner, watching for stagnation and reconnecting observations to the broader goal (see §A.2).

Maintaining a single execution context offers the benefit of a coherent, contiguous exploration history that is easily accessible to the Researcher. However, it also introduces two limitations. First, current LLMs have finite context windows: once the limit is reached, the process must terminate, regardless of progress. Moreover, as the context grows, model attention becomes increasingly uneven across earlier content [42, 62]. Second, the single-context design scales poorly as there is no straightforward way to improve the resulting algorithms merely by increasing compute, budget, or iteration count. The next section introduces two extensions that mitigate these limitations.

4.2 Multi-Context Glia (MCG)

To overcome the limitations imposed by a single finite context window, we adopt a *best-of-N* sampling strategy, a common method for scaling test-time computation [12, 72]. In this approach, we execute N independent single-context instances of Glia, each exploring the design space along a distinct trajectory, and select the best-performing algorithm discovered across all runs based on benchmark scores.

We present two variants of this strategy: **Sequential MCG** and **Parallel MCG-N**. In Sequential MCG, single-context Glia runs execute one after another, allowing the process to stop as soon as a satisfactory result is achieved (up to a maximum number governed by cost). In Parallel MCG-N, N independent runs proceed concurrently, with N serving as a user-specified hyperparameter. For both variants, Glia returns the best-performing design across all runs as the final output. A performance comparison between these modes is presented in §A.3.

Because the runtime and output quality of a single-context instance cannot be predicted in advance, the two modes expose different trade-offs. Sequential MCG achieves faster early progress, as results can be assessed incrementally after each run, but its early performance varies depending on which trajectories complete first.

Parallel MCG-N, in contrast, requires more peak compute but provides steadier improvement: evaluating multiple trajectories simultaneously reduces the likelihood that all runs perform poorly, leading to more consistent aggregate performance. In practice, Sequential MCG is preferable when compute resources are constrained, while Parallel MCG-N offers lower variance and smoother convergence. Our experiments show that most of the benefit saturates by N=4, with larger values yielding diminishing returns. We discuss these differences further in §A.3.

5 Evaluation

We use OpenAI `o3` [59] as the underlying language model. In the following experiments, we specified a budget of $30 per optimization.

We evaluate Glia in `vidur`, a simulator for distributed LLM serving systems [1]. This section focuses on the request routing strategy among LLM replicas in distributed serving environments. We study how Glia discovers novel routing algorithms that lower the mean request completion time (RT) across all requests.

We simulate an LLM inference workload (ShareGPT [67]) on four NVIDIA A10 GPUs running Llama-3-8B-Instruct. To emulate the heavy-tailed decode lengths typical of reasoning workloads (e.g., chain-of-thought) while preserving the base ShareGPT distribution, we independently inflate 5% of decode lengths and 5% of prompt lengths by 10×. The workload uses a query rate of 7.5 queries per second (QPS), with bursty interarrival times following a log-normal distribution ($\sigma = 2$). We chose 7.5 QPS because it sits at the knee of the system's operating curve: lower rates underload the cluster (all methods perform alike), while higher rates saturate it. Each LLM replica uses chunked prefill [2] for batch scheduling, with a chunk size of 8192. We run each experiment with ten random seeds.

The primary evaluation metric is mean request completion time (RT), defined as the end-to-end latency to receive a response. This reflects user-perceived responsiveness in LLM serving and measures the effectiveness of scheduling policies under dynamic workloads. We also report 90% bootstrapped confidence intervals.

Comparisons to other approaches. We compare Glia to three state-of-the-art frameworks that employ LLMs and evolutionary strategies for discovery: (i) Evolution of Heuristics (EoH) [37], (ii) FunSearch [63], and (iii) OpenEvolve [68]. The parameter settings for these systems are shown in Tab. 1.

Evolution of Heuristics [37] introduces a "thought" step into algorithm design and iteratively evolves candidate heuristics using five operators: crossover operators (E_1: generate diverse heuristics, E_2: recombine common ideas) and mutation operators (M_1: modify heuristics for improvement, M_2: adjust parameters within a heuristic, M_3: simplify by removing redundancies).

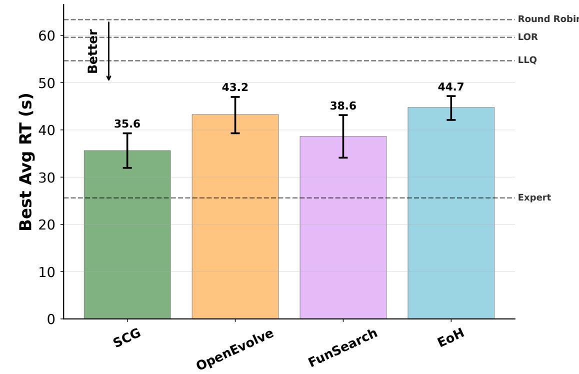
FunSearch [63] applies a best-shot optimization within an island model. In each generation, it selects top-performing code examples from the solution database and prompts the LLM to refine them, maintaining a bounded candidate island through iterative improvement.

OpenEvolve [68], an open-source implementation of AlphaEvolve [57], also employs island-based evolutionary search. It begins with an initial candidate program and a task-specific evaluation function to maximize a score. A prompt sampler generates inputs for the LLM, which produces new program variants. The system then applies evolutionary principles and migration across islands to improve solutions across generations.

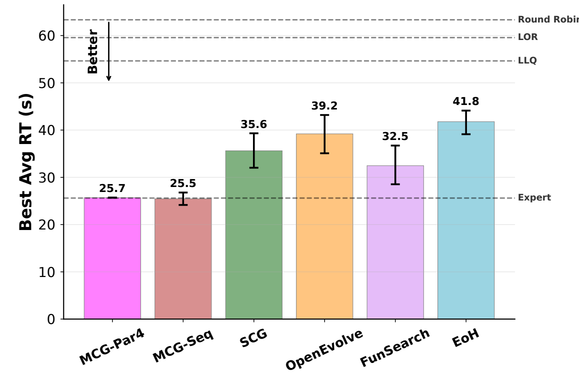
Baseline routing comparisons. We compare Glia with three routing heuristics: Round-Robin, Least-Loaded Queue (**LLQ**), and Least Outstanding Requests (**LOR**). **Round-Robin** forwards requests cyclically across replicas. LLQ selects the replica with the fewest inflight requests, while LOR chooses the one with the fewest waiting requests (those without allocated GPU memory). We also compare against an expert-designed, workload-specialized algorithm developed over two weeks by a senior systems researcher with over 20 years of experience.

5.1 Glia Outperforms Baselines

Fig. 3a compares the algorithms generated by Single-Context Glia (SCG) with routing heuristic baselines and prior algorithm design methods under a strict simulation budget of at most 15 runs. On average, SCG discovers algorithms that reduce mean response time (RT) by 1.3–1.4× compared to EoH, FunSearch, and OpenEvolve, and achieves the fastest discovery rate among them.



(a) Single-Context Glia (SCG) finds better algorithms compared to prior methods after a strict 15-simulation run budget. Lower response time (RT) is better. Error bars show the 90% bootstrapping confidence intervals.



(b) Comparison of the algorithms produced by Glia with prior methods on a more relaxed 100-simulation budget (lower RT is better). Both versions of MCG Glia (MCG-Par4 and MCG-Seq) find solutions that perform better than all the other methods. The error bars show the 90% bootstrapping confidence intervals.

Figure 3: Performance of Glia against other baselines.

Next, we increase the simulation budget to 100 runs. Fig. 3b shows the best-performing algorithms produced by Glia variants compared with the same baselines in this case. SCG performs similarly to the 15-simulation setting (Fig. 3a). This stability stems from SCG's stopping rule: it halts once the Researcher agent either completes its reasoning process or exhausts the context window, preventing further additions or revisions. In contrast, EoH, FunSearch, and OpenEvolve continue improving with additional simulations, attempting to use the extra resources. SCG cannot benefit from these available resources because it follows a single reasoning trajectory; because it runs out of context, it cannot extend its research process.

Multi-Context Glia (MCG) addresses this limitation, as explained in §4.2. MCG samples multiple independent reasoning chains, either in parallel (MCG-Par4) or sequentially (MCG-Seq), and selects the best resulting algorithms. MCG-Par4 launches four independent SCG processes simultaneously to diversify the search, while MCG-Seq starts a new SCG after each previous run completes. Both MCG versions achieve the lowest average RT, outperforming SCG, traditional routing heuristics (Round-Robin, LLQ, LOR), and state-of-the-art LLM-based design frameworks (EoH, FunSearch, OpenEvolve). This multi-context scaling enables Glia to effectively utilize larger simulation budgets and continually improve solution quality.

MCG delivers substantial gains. On average its algorithms reduce average RT by **1.4**× compared to SCG. It outperforms EoH, OpenEvolve, and FunSearch by **1.7**×, **1.6**×, and **1.3**×, respectively.

5.2 Glia 's Discoveries Transfer to Real Systems

We implemented Glia's routing strategy in Production Stack [77], an open-source request router for LLM inference using vLLM. Our testbed comprises 4 Nvidia A10 GPUs serving the Meta-Llama-3-8B-Instruct model. To evaluate, we measure **Slowdown**, defined as

$$\text{Slowdown} = \frac{RT_{\text{system}}}{RT_{\text{ideal}}},$$

where RT_{system} is the request response time (RT) under the evaluated system, and RT_{ideal} is the minimum possible RT in an ideal, load-free setting. A slowdown greater than 1 indicates system overhead.

Fig. 5 shows, even under real-world workload variability and system uncertainty, Glia 's discovered algorithm consistently outperforms all baselines. In the same load regime (QPS = 7.5), the router reduces slowdown by over 4.5× compared to LLQ, confirming Glia's discoveries transfer effectively from simulation to a real system.

5.3 Glia Finds New Algorithms across the Stack

We also applied Glia to the vLLM *batch scheduler* and to the design of an *autoscaler* for the distributed vLLM cluster.

Within the inference engine, whenever a GPU becomes idle, a scheduling algorithm selects which unfinished requests should proceed. This batch scheduler forms batches while respecting constraints such as available KV cache memory, maximum requests per batch, and token limits. In the experiments above, we used the Sarathi [2] batch scheduler, which performs chunked prefill similarly to current vLLM implementations.

Using our optimized request router, we asked Glia to improve the batch scheduler. It discovered that ordering requests by prefill length (rather than arrival time, as in vLLM and Sarathi) reduces end-to-end delay by an additional 25%. The insight mirrors why Shortest-Remaining-Time-First outperforms First-Come-First-Served: prioritizing shorter prefills minimizes head-of-line blocking, reducing queueing delays for short prompts while adding negligible delay for longer ones, thereby lowering mean end-to-end latency.

At another layer of the orchestration stack, autoscaling adjusts the number of compute instances to meet latency targets while minimizing cost. To evaluate this, we implemented autoscaling in the vidur simulator and generated a long-running workload with temporal variability (QPS ranging from 7.5 to 22.5 in a slow sinusoidal pattern). We first implemented a baseline autoscaler, modeled after production systems, that scales based on per-instance decode throughput: it adds an instance when throughput exceeds a high threshold, removes the least busy instance when it falls below a low threshold, and includes a cooldown mechanism to prevent oscillation.

We then asked Glia to design a more efficient autoscaler that minimizes compute cost while keeping the p95 slowdown below 5×. Glia proposed a proportional control loop that adjusts the number of instances based on inflight requests per instance. It then tuned the controller thresholds for this specific model and workload, finding

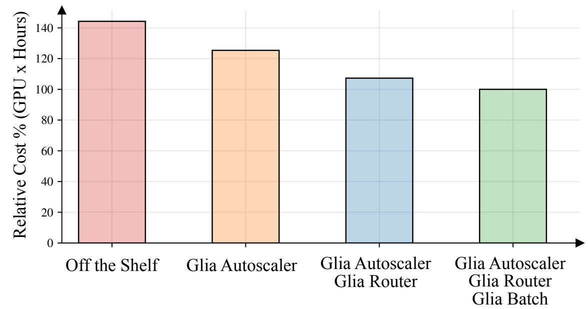


Figure 4: Glia's GPU cost reductions as we progressively use it across the inference stack.

an optimal configuration that minimizes cost while satisfying the latency constraint.

Fig. 4 shows the total GPU×hours saved when applying Glia across different layers of the stack. The Glia-discovered autoscaler alone reduces GPU cost by 13% compared to an off-the-shelf autoscaler, while the full Glia-optimized stack (router, batch scheduler, and autoscaler) cuts total GPU×hours by 40% for this variable workload, compared to standard serving systems (vLLM batch scheduler, LLQ router, and throughput-based autoscalers).

5.4 Glia Finds Novel, Interpretable Algorithms

Fig. 12 shows a representative algorithm discovered by Glia. This algorithm is principled and, to the best of our knowledge, introduces a new concept in this domain: a *Head-Room Admission* (HRA) router that reserves headroom to accommodate unknown decode growths. The router follows an admission-control approach—if a replica lacks sufficient KV-cache memory (after accounting for headroom), the request is queued and dispatched only when resources may have become available. It combines this idea with a shortest-prefill-first policy, approximating shortest-job-first scheduling, a strategy known to minimize mean latency [70]. By maintaining a small KV-cache headroom on each replica at admission time, the router effectively prevents vLLM from running out of memory, thereby avoiding request restarts and wasted computation.

Glia 's discovered algorithms are natural and intuitive, often reflecting the kind of principled reasoning a human designer might employ. In contrast, algorithms produced by prior evolutionary frameworks such as EoH, FunSearch, and OpenEvolve are typically more complex and less interpretable. For instance, the program generated by FunSearch (Fig. 13) includes numerous hyperparameters, conditional branches, and opaque thresholds, making it difficult to identify which components actually drive performance. Other evolutionary baselines show similar traits, yielding heuristics that are cumbersome to interpret and challenging to analyze, refine, or reason about.

5.5 Glia Can Adapt to New Workloads

A key motivation for automated optimization is that no single routing policy performs well across all workloads. When workload characteristics, hardware configurations, or optimization objectives change, the best policy changes with them. In such cases, we can re-run Glia on the new setting to automatically discover a specialized algorithm without human re-engineering.

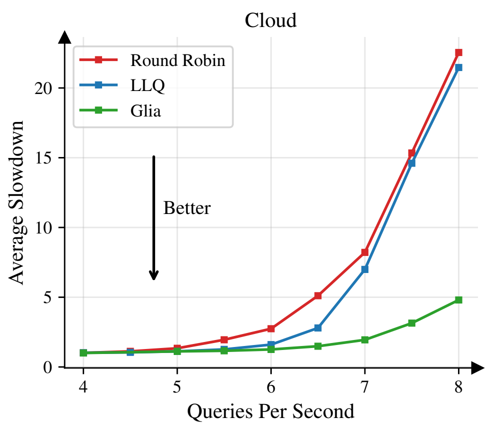


Figure 5: Glia's discovered routing algorithm outperforms baselines in cloud experiments. The trends observed in the cloud experiments are similar to simulation though the numbers aren't identical.

We evaluate this by running Glia on a setting that differs from §5.1 in *i)* workload, *ii)* hardware, and *iii)* optimization settings. The Researcher agent uses `GPT5` as its underlying model.

- **Workload:** Prefill-heavy, with a prompt-to-decode ratio of about 70. The workload generator operates in a closed loop, maintaining 200 concurrent requests, analogous to 200 users waiting for LLM responses in a chat application.
- **Model and Hardware:** `Llama-3.3-70B-Instruct-FP8-dynamic` running on 8 Nvidia H100 GPUs.
- **Objective:** Constrained optimization to maximize request throughput (QPS) while keeping P90 TTFT below 1500 ms.

In this new setting, none of the standard routing heuristics—Round-Robin, LLQ, or LOR—satisfy the TTFT constraint. More importantly, the expert-designed heuristic from §5.1 not only violates the constraint but yields a significantly higher TTFT than these baselines, revealing that its strong earlier performance relied on workload specialization. Without automated re-optimization, a human expert would need to manually reanalyze the workload and redesign the routing strategy.

Glia overcomes this by discovering, from scratch, a routing algorithm that meets the TTFT constraint in all ten trials. Moreover, its achieved QPS exceeds that of the baseline heuristics, even though those heuristics fail to meet the constraint.

To further analyze, we vary the number of inflight requests and compare the P90 TTFT vs. throughput curves for the expert-designed and Glia-designed algorithms (Fig. 6). The expert algorithm satisfies the SLO only up to 1.5 QPS, while the Glia-discovered algorithm sustains compliant TTFT up to 7 QPS—a 4.6× improvement.

5.6 Other Experiments

We provide additional experiments in App. A. An architecture ablation (§A.2) isolates the contribution of each component in Glia, including the qualitative and quantitative effect of the supervisor. We run multi-context ablation in §A.3. We also show Glia's HRA algorithms is robust both to parameter changes and in tail (§B).

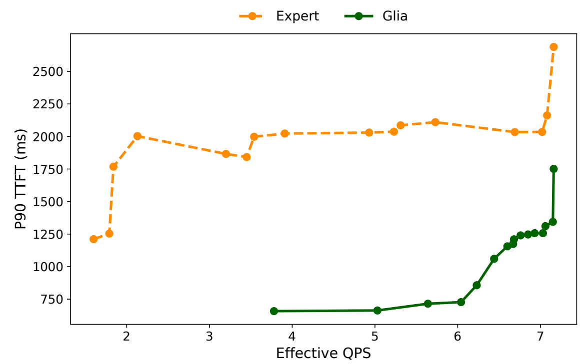


Figure 6: P90 TTFT vs. request throughput for the expert and Glia algorithms. The expert algorithm was tailored to a different workload and violates the 1500 ms SLO beyond 1.5 QPS; Glia's algorithm maintains compliance up to 7 QPS.

6 Conclusion

We are progressing toward our primary goal: developing Glia into an AI capable of PhD-level systems design and optimization for real-world problems. Glia demonstrates the promise of AI-driven infrastructure optimization, yet significant open questions remain. Achieving fully self-managing computing infrastructures will require progress across several key dimensions:

(1) **Robustness and safety**: Automated systems must remain stable under adversarial or unexpected conditions. AI-based approaches must ensure robustness to workload shifts, failures, and anomalies. Additionally, their effectiveness depends on the fidelity of the evaluation framework used to score candidate designs and provide diagnostic telemetry.
(2) **Human-AI collaboration**: Glia's explainability is a core strength. Future work will improve how architects use AI insights through better visualization, interactive debugging, and co-design tools.
(3) **Generalization across systems**: Although current results focus on AI inference stacks, our approach holds promise for optimizing a broad range of complex computer systems.
(4) **Abstractions and architectural discovery**: Designing a new abstraction such as an API, transport semantic, or queueing primitive is qualitatively different from optimizing a heuristic policy. Abstraction design requires reasoning about what to expose or hide, composability, failure modes, and long-term maintainability. These qualities are less measurable and typically evaluated through argumentation, analysis, and targeted experiments. Whether AI can propose abstractions that are both useful and stable is an open question.
(5) **Context management**: Glia currently addresses the single-context bottleneck through a simple best-of-N strategy (§4.2). Context management techniques could further complement Glia's exploration trajectories.
(6) **Online re-optimization**: In this work, re-optimization is triggered manually for each workload. Detecting workload shifts and automatically invoking Glia is a natural extension.

Acknowledgments

This work was funded by the MIT Generative AI Impact Consortium (MGAIC).

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, and Meng Jiang. 2025. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457* (2025).

[4] Jayvant Anantpur, Nagendra Gulur Dwarakanath, Shivaram Kalyanakrishnan, Shalabh Bhatnagar, and R. Govindarajan. 2017. RLWS: A Reinforcement Learning based GPU Warp Scheduler. *arXiv preprint arXiv:1712.04303* (2017).

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

[6] Anthropic. 2024. Building Effective Agents. https://www.anthropic.com/research/building-effective-agents Accessed: 2026-04-27.

[7] 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).

[8] Michelle Brachman, Amina El-Ashry, Casey Dugan, and Werner Geyer. 2025. Current and Future Use of Large Language Models for Knowledge Work. arXiv:2503.16774 [cs.HC] https://arxiv.org/abs/2503.16774

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

[10] Jie Chen, Kang G. Shin, Jiaqi Zheng, Xin Jin, Xia Zhou, Ben Y. Zhao, and Haitao Zheng. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Traffic Optimization. In *ACM SIGCOMM Workshop on APNet*.

[11] 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

[12] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[13] Jaber Daneshamooz, Jessica Nguyen, William Chen, Sanjay Chandrasekaran, Satyandra Guthula, Ankit Gupta, Arpit Gupta, and Walter Willinger. 2025. Addressing the ML Domain Adaptation Problem for Networking: Realistic and Controllable Training Data Generation with NetReplica. arXiv:2507.13476 [cs.NI] https://arxiv.org/abs/2507.13476

[14] DeepMind. 2024. Advanced version of Gemini with DeepThink officially achieves gold-medal standard at the International Mathematical Olympiad. https://deepmind.google/discover/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad/. Accessed: 2025-10-17.

[15] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[16] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P. Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*. 343–356.

[17] 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).

[18] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling Test-Time Compute for Software Engineering. arXiv:2501.14723 [cs.LG] https://arxiv.org/abs/2501.14723

[19] 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).

[20] Harvard Extension School. [n. d.]. Principles of Good Design. https://cscie2x.dce.harvard.edu/hw/ch01s06.html. Accessed: 2025-10-17.

[21] 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

[22] 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

[23] 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).

[24] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. Internet Congestion Control via Deep Reinforcement Learning. arXiv:1810.03259 [cs.NI] https://arxiv.org/abs/1810.03259

[25] Nathan Jay, Yair Rotman, P. Brighten Godfrey, and Michael Schapira. 2019. An End-to-End Deep Reinforcement Learning Framework for Internet Congestion Control. In *ICML*.

[26] 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.

[27] 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.

[28] 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

[29] Mehrdad Khani, Mohammad Alizadeh, Jakob Hoydis, and Phil Fleming. 2020. Adaptive neural signal detection for massive MIMO. *IEEE Transactions on Wireless Communications* 19, 8 (2020), 5635–5648.

[30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SOSP* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

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

[32] Nikolay Lazic, Craig Boutilier, Thomas Lu, Eric Wong, Binz Roy, Marcin Minka, Ben J. Heller, David Schuurmans, Geoffrey J. Gordon, Olivier Duchesnay, Marc L. Bellemare, Albin Cassirer, et al. 2018. Data Center Cooling Using Model-Predictive Control. In *Advances in Neural Information Processing Systems (NeurIPS) Workshop*. Describes learning-assisted control for DC cooling.

[33] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. 2024. LLM Inference Serving: Survey of Recent Advances and Opportunities. arXiv:2407.12391 [cs.DC] https://arxiv.org/abs/2407.12391

[34] Tianhong Li, Vibhaalakshmi Sivaraman, Pantea Karimi, Lijie Fan, Mohammad Alizadeh, and Dina Katabi. 2023. Reparo: Loss-resilient generative codec for video conferencing. *arXiv preprint arXiv:2305.14135* (2023).

[35] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158

[36] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. NeuroCuts: Neural Decision Trees for Packet Classification. In *SIGCOMM*. 1–15.

[37] 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.

[38] 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).

[39] 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).

[40] 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).

[41] 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).

[42] 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

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

[44] 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).

[45] llm-d Community. 2025. GitHub - llm-d/llm-d: llm-d enables high-performance distributed LLM inference on Kubernetes. https://github.com/llm-d/llm-d. [Accessed 10-10-2025].

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

[47] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.

[48] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. 2020. Real-world Video Adaptation with Reinforcement Learning. arXiv:2008.12858 [cs.NI] https://arxiv.org/abs/2008.12858

[49] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *SIGCOMM*. 197–210.

[50] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*. 270–288.

[51] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*. 270–288. doi:10.1145/3341302.3342080

[52] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *SIGMOD*. 1275–1288.

[53] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. doi:10.14778/3342263.3342644

[54] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. 2020. Interpreting Deep Learning-Based Networking Systems. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 154–171. doi:10.1145/3387514.3405859

[55] MIT News Office. 2023. Study finds ChatGPT boosts worker productivity in writing tasks. *MIT News* (2023). https://news.mit.edu/2023/study-finds-chatgpt-boosts-worker-productivity-writing-0714 Accessed: 2025-10-17.

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

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

[58] NVIDIA. 2025. GitHub - ai-dynamo/dynamo: A Datacenter Scale Distributed Inference Serving Framework. https://github.com/ai-dynamo/dynamo. [Accessed 10-10-2025].

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

[60] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. doi:10.1109/ISCA59077.2024.00019

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

[62] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield. 2025. Effective context engineering for AI agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents With contributions from Rafi Ayub, Hannah Moran, Cal Rueb, and Connor Jennings. Published online September 29, 2025.

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

[64] Fabian Ruffy, Michael Przystupa, and Ivan Beschastnikh. 2018. Iroko: A Framework to Prototype Reinforcement Learning for Data Center Traffic Control. *arXiv preprint arXiv:1812.09975* (2018).

[65] Saim Salman, Christopher Streiffer, Huan Chen, Theophilus Benson, and Asim Kadav. 2018. DeepConf: Automating Data Center Network Topologies and Routing with Deep Reinforcement Learning. *arXiv preprint arXiv:1712.03890* (2018).

[66] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

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

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

[69] 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).

[70] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th ed.). Wiley Publishing.

[71] Vibhaalakshmi Sivaraman, Pantea Karimi, Vedantha Venkatapathy, Mehrdad Khani, Sadjad Fouladi, Mohammad Alizadeh, Frédo Durand, and Vivienne Sze. 2024. Gemino: Practical and robust neural compression for video conferencing. In *NSDI*.

[72] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314* (2024).

[73] 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).

[74] Yiheng Tao, Yihe Zhang, Matthew T. Dearing, Xin Wang, Yuping Fan, and Zhiling Lan. 2025. Prompt-Aware Scheduling for Low-Latency LLM Serving. arXiv:2510.03243 [cs.LG] https://arxiv.org/abs/2510.03243

[75] The AIBrix Team, Jiaxin Shan, Varun Gupta, Le Xu, Haiyang Shi, Jingyuan Zhang, Ning Wang, Linhui Xu, Rong Kang, Tongping Liu, Yifei Zhang, Yiqing Zhu, Shuowei Jin, Gangmuk Lim, Binbin Chen, Zuzhi Chen, Xiao Liu, Xin Chen, Kante Yin, Chak-Pong Chung, Chenyu Jiang, Yicheng Lu, Jianjun Chen, Caixue Lin, Wu Xiang, Rui Shi, and Liguang Xie. 2025. AIBrix: Towards Scalable, Cost-Effective Large Language Model Inference Infrastructure. arXiv:2504.03648 [cs.DC] https://arxiv.org/abs/2504.03648

[76] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[77] vllm-project. 2025. vLLM Production Stack: reference stack for production vLLM deployment. https://github.com/vllm-project/production-stack.

[78] 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).

[79] 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).

[80] David Wheeler. [n. d.]. Problems in the Design of Systems. https://www.doc.ic.ac.uk/~dcw/PSD/article13/. Accessed: 2025-10-17.

[81] Wikiquote contributors. 2025. Edsger W. Dijkstra – Wikiquote. https://en.wikiquote.org/wiki/Edsger_W._Dijkstra. Accessed: 2025-10-17.

[82] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*. 123–134. doi:10.1145/2486001.2486011

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

[84] 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).

[85] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: A randomized experiment in video streaming. In *NSDI*. 495–511.

[86] Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, et al. 2026. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657* (2026).

[87] Yuqing Yang, Yuedong Xu, and Lei Jiao. 2024. A Queueing Theoretic Perspective on Low-Latency LLM Inference with Variable Token Length. arXiv:2407.05347 [cs.NI] https://arxiv.org/abs/2407.05347

[88] 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.

[89] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *OSDI*. USENIX Association, Carlsbad, CA, 521–538.

https://www.usenix.org/conference/osdi22/presentation/yu
[90] 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).
[91] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 415–432. doi:10.1145/3299869.3314043
[92] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104
[93] 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).
[94] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. 2021. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 258–271. doi:10.1145/3452296.3472902

System	Parameter Name	Value
EoH	Initial population size (N)	20
FunSearch	# Prompt programs (k)	3
	Population size per island	20
	Number of islands	6
	Exploration probability	0.6
OpenEvolve	Exploitation probability	0.4
	Initial program	LLQ router

Table 1: Hyperparameters for the other approaches used in our evaluation.

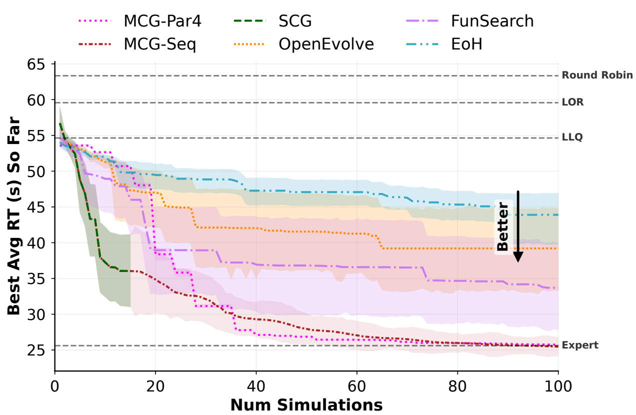


Figure 7: Comparison of Glia variants with baselines and prior methods (lower is better). SCG has the steepest early gains thanks to coherent and continuous white-box reasoning. The two variants of MCG—4-way parallel (MCG-Par4) and sequential (MCG-Seq)—extend the gains and outperform other methods by finding better algorithms more quickly. Shades show 90% confidence intervals.

A Deep Dive into Glia's Design

A.1 Glia Finds Good Algorithms Quickly

Fig. 7 compares Glia's progress across simulations with prior algorithm design methods (lower is better). Glia consistently discovers stronger algorithms using far fewer simulations. Single-Context Glia (SCG) achieves the steepest early gains, driven by white-box reasoning and focused, continuous refinement within a single context.

However, SCG's scalability is limited. Because the Researcher can conclude the search regardless of available simulations (hence the shorter SCG curve), additional resources cannot easily improve performance. We address this limitation with Multi-Context Glia (MCG) approaches. As shown in Fig. 7, both 4-way Parallel Glia (MCG-Par4) and Sequential Glia (MCG-Seq) overcome SCG's constraint and continue to improve. On average, MCG methods discover better-performing algorithms faster than EoH, FunSearch, and OpenEvolve, achieving lower RT than all other methods.

Across the 10 runs, Multi-Context Glia reaches the Expert solution every time. Single-Context Glia succeeds in 50% of runs, limited by its single context window. Among evolutionary baselines, FunSearch succeeds in 50%, OpenEvolve in 30%, and EoH in 10%.

A.2 Architecture Ablation

We ablate Glia's components to isolate their effect (Fig. 8).

Removing the Supervisor. The *Researcher Only* variant retains Glia's Researcher but removes the Supervisor entirely. Although it performs reasonably well thanks to the detailed Researcher prompt, it achieves a lower score than Glia. The *Researcher Only* plateaus earlier and executes fewer simulations before terminating (Fig. 8b). This suggests that the Supervisor plays a role in sustaining exploration and preventing premature convergence.

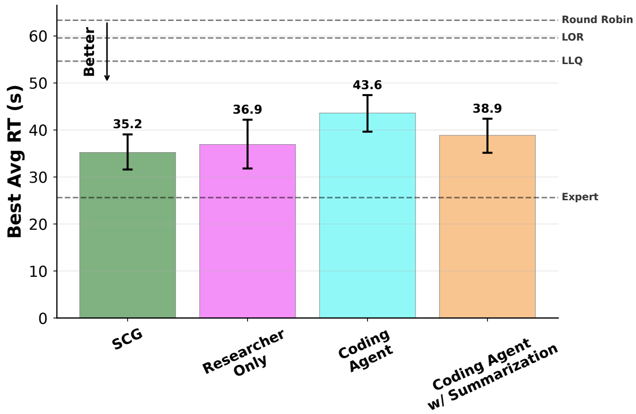
Coding Agent. Replacing the Researcher with a vanilla coding agent (Codex with o3) degrades performance to 43.6 s. We have given the coding agent the same user prompt and the agent has full simulator access, but it lacks the structured hypothesis-driven exploration and disengages earlier.

Summarization. Equipping the coding agent with summarization-based context management does not close the gap (38.9 s). While summarization extends the context window and encourages the agent to run more experiments (Fig. 8b), it causes the agent to lose the reasoning behind why experiments were conducted in the first place — the coherent chain of thought linking hypotheses to outcomes is lost, leading to redundant exploration and weaker designs.

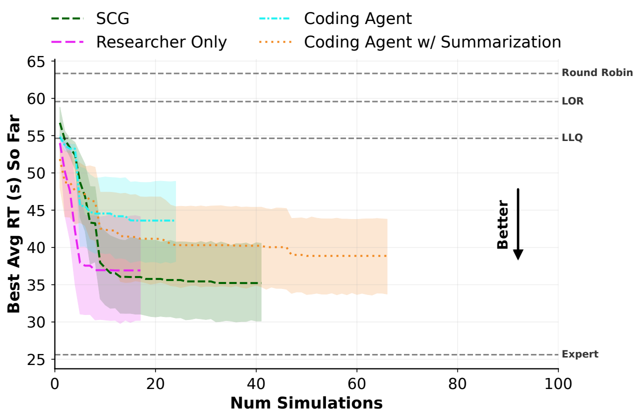
Supervisor's qualitative impact. To illustrate the qualitative impact of the Supervisor, we present an example intervention. In one run, early exploration produced two ideas: (i) combining backlog-aware selection with heavy-request isolation, and (ii) bounding the one-pass queue scan to prevent large requests from starving. Instead, the Researcher pursued adaptive headroom tuning and heavy-lane gating, which reduced latency from 55 s to 28 s but still missed the 22 s target. Believing further progress was unlikely, the Researcher considered stopping. At this point, the Supervisor intervened:

> You're close, but not stuck yet. A couple of quick checkpoints:
>
> (1) Recall your finding that queueing delay still dominates whenever memory is oversubscribed.
> - Does the dynamic-headroom variant fully leverage backlog information when deciding headroom, or is it applied replica-independently?
> - You previously tested backlog-aware selection and heavy-request isolation separately; have you measured them together under the same admission-control knob?
>
> (2) Your one-pass queue scan leaves large, unsafe requests parked. Verify whether those accumulate and starve; if so, a bounded re-ordering window (still SPF) might smooth the tail without extra restarts.
>
> If you haven't exhausted these combinations, worth a final sweep before concluding.

The Researcher then proceeds to revisit these earlier ideas, combining backlog-aware selection with heavy-request isolation and exploring bounded re-ordering, which ultimately advanced the router beyond the plateaued performance.



(a) Best average request response time achieved by each method. Glia outperforms all variants, demonstrating that both the Researcher prompt and the Supervisor contribute to solution quality.



(b) Best-so-far request response time vs. number of simulations. Removing the Supervisor (Researcher Only) leads to earlier plateauing and fewer simulations. The coding agent plateaus early; summarization extends exploration but degrades final performance. Glia sustains coherent exploration longest. Shaded regions show 90% confidence intervals.

Figure 8: Architecture ablation: systematically removing Glia's components. We compare Single-Context Glia (SCG), Researcher Only (no Supervisor), a vanilla coding agent (Codex with o3), and a coding agent with summarization-based context management. Lower is better.

A.3 Multi-context Glia Ablation

Fig. 9 compares Glia's Multi-Context variants. N-way Parallel Glia runs N independent instances of Single-Context Glia in parallel and reports the best-performing algorithm among them. Its key advantage is scalability: with more computational resources, more instances can be launched simultaneously. For example, 4-way Parallel Glia outperforms 2-way because the likelihood that all agents "go astray" decreases as the number of parallel runs increases. This design also mitigates early exploration risk, since at least one agent is likely to discover a strong algorithm quickly. However, it introduces a new hyperparameter, N, which must be chosen;

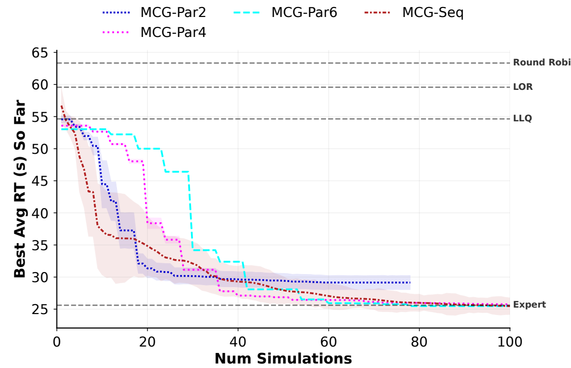


Figure 9: Comparing Glia variants (lower is better).

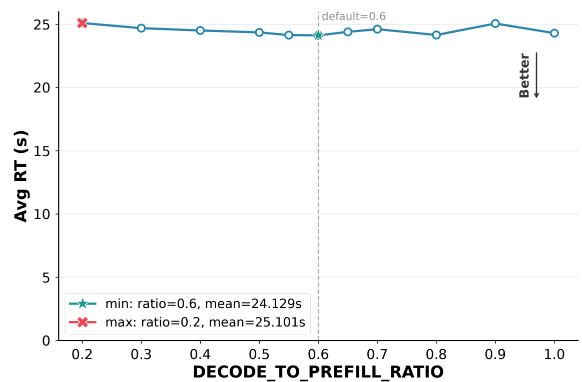


Figure 10: HRA is robust to the decode-to-prefill ratio r. Average request completion time varies by less than 1 s across $r \in [0.2, 1.0]$; Glia's tuned value r=0.6 achieves the minimum.

there is no obvious optimal value, though in our experiments $N=4$ achieved a good balance between performance and resource cost. Larger values (e.g., $N=6$ in Fig. 9) yield diminishing returns.

In contrast, Sequential Glia requires no additional parameters and is simpler to use. It initially reduces error more sharply than N-way Parallel (within the first 35 simulations) but later slows down, taking longer to match the performance of 4-way Parallel Glia.

B Glia Finds a Robust Algorithm

HRA parameter sensitivity. The decode-to-prefill ratio r (see §3.3) in Glia's HRA algorithm (`DECODE_TO_PREFILL_RATIO` in Fig. 12) is a tunable parameter that Glia sets during optimization by observing workload telemetry. To assess sensitivity, we swept r across $[0.2, 1.0]$ on the same workload. As shown in Fig. 10, HRA is robust to this parameter: the variance in mean request completion time is under 1 s across the entire range, and Glia's tuned value of r=0.6 lands at the minimum.

Tail latency. Production deployments often enforce P99 latency SLOs, so robustness at the tail is critical. Fig. 11 reports the P99 request completion times for the best-performing algorithm from each method across 10 runs (the same codes used in Fig. 3b). The results show that Multi-Context Glia (both sequential and parallel)

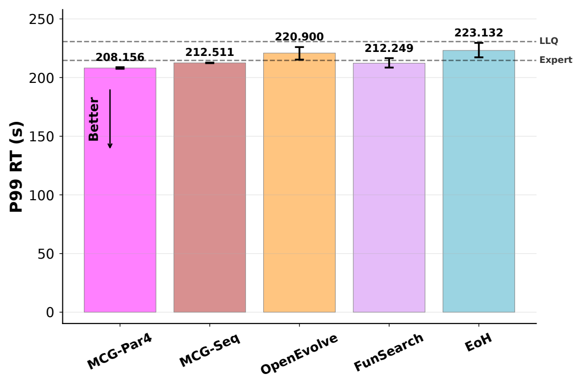


Figure 11: P99 request completion times for all methods' best-performing codes. Lower is better. Multi-Context Glia (both sequential and parallel) achieves the strongest tail latency among all methods. Error bars show 90% bootstrapped confidence intervals.

is robust at the tail, achieving strong P99 latency values relative to all baselines and competing methods.

C Cost Breakdown

A typical Single-Context Glia run costs approximately \$4.95 in API tokens and takes 1–2 hours wall-clock (including simulation and reasoning time). MCG-Par4 runs four instances simultaneously, averaging \$20 total (below our \$30 budget) and approximately 2 hours wall-clock. MCG-Seq runs three to four instances sequentially for an average of \$20–\$30 total and approximately 6 hours wall-clock.

All evolutionary baselines cost less than \$10 and take 1–2 hours wall-clock. Glia's higher token cost is by design: evolutionary systems issue single LLM calls per iteration, while Glia engages in multi-turn reasoning and telemetry analysis. Our objective is not to optimize for token cost but for exploration quality—measured by progress per evaluation—and final solution performance. This focus is motivated by real-world system optimization, where the bottleneck is evaluation cost and time, not LLM tokens: running experiments on a real GPU cluster is far more expensive than the API calls Glia consumes. Evolutionary approaches compensate for their lower token cost by relying on a large number of trial-and-error evaluations, the cost of which dominates in any realistic deployment setting.

Figure 12: Python code for the Head-Room Allocator (HRA) request routing algorithm discovered by Glia.

```python
"""Head-Room Admission (HRA) global scheduler.

This scheduler mitigates vLLM pre-emptions by keeping a small KV-cache
head-room on every replica *at admission time*. For each incoming request we
pessimistically reserve additional blocks to account for the (unknown) decode
phase and admit the request only if the target replica would still retain the
configured safety margin.

Empirical defaults (good for the ShareGPT-style workload used in Vidur's
benchmarks):

    DECODE_TO_PREFILL_RATIO = 0.6 # avg decode/prompt tokens
    SAFETY_FRACTION = 0.03 # keep last 3 % blocks free

These values reduce average end-to-end latency by ~40 % compared to LLQ while
maintaining >95 % GPU utilisation.
"""

from math import ceil
from typing import List, Tuple

from vidur.entities import Request
from vidur.scheduler.global_scheduler.base_global_scheduler import BaseGlobalScheduler

# ---------------------------------------------------------------------------
# Tunable constants (change if workload characteristics differ significantly)
# ---------------------------------------------------------------------------

DECODE_TO_PREFILL_RATIO: float = 0.6 # pessimistic decode growth factor
SAFETY_FRACTION: float = 0.03 # minimum fraction of blocks kept free


class AIGlobalScheduler(BaseGlobalScheduler):
    """Memory-aware global scheduler with fixed head-room admission control."""

    # pylint: disable=protected-access

    def schedule(self) -> List[Tuple[int, Request]]:
        # Always serve the *shortest* prompt next (SJF) to minimise mean latency.
        self._request_queue.sort(key=lambda r: (r.num_prefill_tokens, r.arrived_at))

        if not self._request_queue:
            return []

        # Cluster-wide, all replicas share the same memory configuration.
        any_scheduler = next(iter(self._replica_schedulers.values()))
        block_size = any_scheduler._config.block_size
        max_blocks = any_scheduler._config.num_blocks
        min_free_blocks = int(max_blocks *SAFETY_FRACTION)

        # Snapshot per-replica state and keep optimistic updates locally so that
        # multiple placements within one call are consistent.
        allocated_blocks = {
            rid: rs.num_allocated_blocks for rid, rs in self._replica_schedulers.items()
        }
        pending_reserved_blocks = {
            rid: ceil(
                sum(r.num_prefill_tokens *(1 + DECODE_TO_PREFILL_RATIO) for r in rs._request_queue)
                / block_size
            )
            for rid, rs in self._replica_schedulers.items()
        }
        queue_lengths = {
            rid: rs.num_pending_requests + rs.num_active_requests for rid, rs in self._replica_schedulers.items()
        }

        request_mapping: List[Tuple[int, Request]] = []

        idx = 0
        # Traverse requests in order; if the head request cannot be admitted we
        # stop to preserve FIFO fairness (new arrivals behind it must wait).
        while idx < len(self._request_queue):
            req = self._request_queue[idx]

            req_blocks = ceil(
                req.num_prefill_tokens *(1 + DECODE_TO_PREFILL_RATIO) / block_size
            )

            admissible = []
            for rid in self._replica_schedulers.keys():
                projected_usage = allocated_blocks[rid] + pending_reserved_blocks[rid] + req_blocks
                free_after = max_blocks - projected_usage
                if free_after >= min_free_blocks:
                    admissible.append(rid)
```

```
            if not admissible:
                break # cannot place the oldest waiting request right now

            # Choose replica with lowest projected usage; tie-break by queue length.
            target_rid = min(
                admissible,
                key=lambda rid: (allocated_blocks[rid] + pending_reserved_blocks[rid], queue_lengths[rid]),
            )

            # Commit placement and optimistically update state for subsequent decisions.
            request_mapping.append((target_rid, req))
            self._request_queue.pop(idx) # do *not* increment idx

            pending_reserved_blocks[target_rid] += req_blocks
            queue_lengths[target_rid] += 1

        return request_mapping
```

Figure 13: Code generated by FunSearch.

```
class CustomGlobalScheduler(BaseGlobalScheduler): # type: ignore[name-defined]
    """Latency-oriented, eviction-aware global scheduler.

    Key features
    -------------
    1. Decode length prediction per *prefill* bucket (small / mid / large)
        with an online exponential moving average; gives markedly better
        memory-footprint forecasts than a single global estimate.

    2. Looks ahead and keeps a projection of every replica's future state
        (memory blocks, remaining pre-fill backlog, queue length). The
        projection is updated greedily after each assignment so later
        decisions use a consistent view.

    3. Request priority is *rescue-first SJF*: previously evicted jobs first
        (to avoid starvation / wasted work), then smaller **total** expected
        tokens, finally FIFO.

    4. Replica selection minimises a composite cost of projected memory
        utilisation (quadratic), outstanding pre-fill backlog, queue length
        and the *instant* block deficit for the pre-fill of the candidate
        request. Jobs with restarts receive a multiplicative cost discount.

    5. Admission control: a new request is dispatched only if the projected
        utilisation stays below a configurable soft limit. The limit is
        relaxed slightly for restarted jobs so they can finish.
    """

    # ------------- tunables -------------
    _BUCKET_BOUNDS = (128, 512) # <128 small, 128-512 mid, >512 large
    _EMA_ALPHA = 0.10 # smoothing for bucketed averages
    _INIT_DECODE_EST = 96.0 # bootstrap decode len (tokens)
    _MIN_DECODE = 32.0
    _MAX_DECODE = 1024.0

    _SOFT_UTIL_CAP = 1.03 # ordinary requests must stay under this
    _SOFT_UTIL_CAP_RESTART = 1.10 # restarted jobs may exceed slightly

    # cost weights (should sum ~1)
    _W_UTIL = 0.55 # projected utilisation^2
    _W_BACKLOG = 0.25 # remaining pre-fill backlog fraction
    _W_QUEUE = 0.10 # queue length fairness
    _W_DEFICIT = 0.10 # instantaneous block deficit

    _OVER_CAP_PEN = 12.0 # extra when util>1
    _RESTART_DISCOUNT = 0.6 # multiplicative cost discount per restart

    # ------------------------------------

    def __init__(self, *args: Any, **kwargs: Any): # type: ignore[override]
        super().__init__(*args, **kwargs)

        # bucketed decode length EWMA statistics
        # structure: (count , avg)
        self._bucket_avg: List[float] = [self._INIT_DECODE_EST] *3
        self._bucket_cnt: List[int] = [0, 0, 0]

        # global fallback EWMA
        self._global_avg: float = self._INIT_DECODE_EST

        # snapshot of visible requests from previous tick (id -> (pf, processed))
        self._prev_snapshot: Dict[int, Tuple[int, int]] = {}

    # --------------------- helper: bucket index ---------------------
    @classmethod
```

```python
    def _bucket_idx(cls, prefill: int) -> int:
        if prefill < cls._BUCKET_BOUNDS[0]:
            return 0
        if prefill < cls._BUCKET_BOUNDS[1]:
            return 1
        return 2

    # --------------------- statistics maintenance ------------------
    def _update_decode_statistics(self) -> None:
        """Detect completed requests and update bucket/global decode EWMAs."""
        current: Dict[int, Tuple[int, int]] = {}

        # helper to insert into current snapshot quickly
        def _collect(req):
            current[id(req)] = (req.num_prefill_tokens, req.num_processed_tokens)

        for req in self._request_queue:
            _collect(req)
        for rep in self._replica_schedulers.values():
            for rq in rep.pending_queue:
                _collect(rq)
            for rq in rep.active_queue:
                _collect(rq)

        # detect finished requests
        finished_ids = set(self._prev_snapshot.keys()) - set(current.keys())
        for rid in finished_ids:
            pf_tokens, processed = self._prev_snapshot[rid]
            decode_tokens = max(0, processed - pf_tokens)
            if decode_tokens <= 0:
                continue

            # update bucket stats
            bidx = self._bucket_idx(pf_tokens)
            old_avg = self._bucket_avg[bidx]
            new_avg = (1.0 - self._EMA_ALPHA) *old_avg + self._EMA_ALPHA *decode_tokens
            self._bucket_avg[bidx] = min(max(new_avg, self._MIN_DECODE), self._MAX_DECODE)
            if self._bucket_cnt[bidx] < 1e9: # avoid overflow
                self._bucket_cnt[bidx] += 1

            # update global average
            g_new = (1.0 - self._EMA_ALPHA) *self._global_avg + self._EMA_ALPHA *decode_tokens
            self._global_avg = min(max(g_new, self._MIN_DECODE), self._MAX_DECODE)

        self._prev_snapshot = current

    # -------------------- decode prediction ------------------------
    def _predict_decode(self, prefill_tokens: int) -> float:
        bidx = self._bucket_idx(prefill_tokens)
        if self._bucket_cnt[bidx] >= 10: # need some data for bucket-specific
            return self._bucket_avg[bidx]
        return self._global_avg

    # -------------------- utility functions ------------------------
    @staticmethod
    def _ceil_div(a: float, b: int) -> int:
        return int(math.ceil(a / b))

    # ---------------------------- main ------------------------------
    def schedule(self) -> List[Tuple[int, 'Request']]: # type: ignore[name-defined]
        # housekeeping
        self._update_decode_statistics()
        if not self._request_queue:
            return []

        replicas: Dict[int, 'ReplicaScheduler'] = self._replica_schedulers # type: ignore[name-defined]
        num_repls: int = max(1, self._num_replicas)

        # ---------- priority sort for global queue ----------
        def _priority(req: 'Request') -> Tuple[int, float, float]: # type: ignore[name-defined]
            predicted_total = req.num_prefill_tokens + self._predict_decode(req.num_prefill_tokens)
            return (-req.num_restarts, predicted_total, req.arrived_at)

        self._request_queue.sort(key=_priority)

        # ---------- projected replica states (without unrouted) ----------
        proj_blocks: Dict[int, int] = {}
        proj_backlog_tokens: Dict[int, int] = {}
        proj_queue_len: Dict[int, int] = {}
        blk_size: Dict[int, int] = {}
        token_capacity: Dict[int, int] = {}

        for rid, rep in replicas.items():
            bs = rep.block_size
            blk_size[rid] = bs
            token_capacity[rid] = rep.num_blocks *bs

            blocks = rep.num_allocated_blocks # currently allocated blocks
            backlog_tokens = 0
```

```
        qlen = len(rep.active_queue) + len(rep.pending_queue)

        # active requests
        for rq in rep.active_queue:
            # remaining future blocks for this request
            total_tokens_goal = rq.num_prefill_tokens + self._predict_decode(rq.num_prefill_tokens)
            future_blocks = self._ceil_div(total_tokens_goal, bs)
            already_blocks = self._ceil_div(rq.num_processed_tokens, bs)
            blocks += max(0, future_blocks - already_blocks)

            # backlog tokens (remaining prefill)
            if rq.num_processed_tokens < rq.num_prefill_tokens:
                backlog_tokens += rq.num_prefill_tokens - rq.num_processed_tokens

        # pending requests
        for rq in rep.pending_queue:
            total_tokens_goal = rq.num_prefill_tokens + self._predict_decode(rq.num_prefill_tokens)
            blocks += self._ceil_div(total_tokens_goal, bs)
            backlog_tokens += rq.num_prefill_tokens

        proj_blocks[rid] = blocks
        proj_backlog_tokens[rid] = backlog_tokens
        proj_queue_len[rid] = qlen

    avg_queue_len = (sum(proj_queue_len.values()) / num_repls) + 1e-6

    # ---------- greedy assignment loop ----------
    mapping: List[Tuple[int, 'Request']] = [] # type: ignore[name-defined]
    remaining: List['Request'] = [] # requests we skip this tick

    while self._request_queue:
        req = self._request_queue.pop(0)
        pred_decode = self._predict_decode(req.num_prefill_tokens)
        total_tokens_req = req.num_prefill_tokens + pred_decode

        best_rid: int | None = None
        best_cost: float = float('inf')
        best_util_after: float = 0.0

        for rid, rep in replicas.items():
            bs = blk_size[rid]
            req_blocks = self._ceil_div(total_tokens_req, bs)
            pf_blocks = self._ceil_div(req.num_prefill_tokens, bs)

            free_now_blocks = rep.num_blocks - rep.num_allocated_blocks
            deficit_blocks = max(0, pf_blocks - free_now_blocks)

            util_after = (proj_blocks[rid] + req_blocks) / rep.num_blocks
            backlog_after = proj_backlog_tokens[rid] + req.num_prefill_tokens
            queue_after = proj_queue_len[rid] + 1

            cost = (
                self._W_UTIL *(util_after **2) +
                self._W_BACKLOG *(backlog_after / (token_capacity[rid] + 1e-6)) +
                self._W_QUEUE *(queue_after / avg_queue_len) +
                self._W_DEFICIT *(deficit_blocks / (rep.num_blocks + 1e-6))
            )

            if util_after > 1.0:
                cost += self._OVER_CAP_PEN *(util_after - 1.0) **2

            # discount for restarts
            if req.num_restarts:
                cost *= (1.0 - self._RESTART_DISCOUNT) **req.num_restarts

            if cost < best_cost - 1e-12:
                best_cost = cost
                best_rid = rid
                best_util_after = util_after

        if best_rid is None:
            remaining.append(req)
            continue

        # ------ admission control ------
        cap = self._SOFT_UTIL_CAP_RESTART if req.num_restarts else self._SOFT_UTIL_CAP
        if best_util_after > cap:
            # keep for next tick
            remaining.append(req)
            continue

        # commit placement
        sel = best_rid
        mapping.append((sel, req))

        bs_sel = blk_size[sel]
        req_blocks_sel = self._ceil_div(total_tokens_req, bs_sel)
        proj_blocks[sel] += req_blocks_sel
        proj_backlog_tokens[sel] += req.num_prefill_tokens
```

```
            proj_queue_len[sel] += 1
            avg_queue_len = (sum(proj_queue_len.values()) / num_repls) + 1e-6

        # push remaining requests back to queue (maintain order)
        self._request_queue = remaining + self._request_queue

        return mapping
```

Prompt for Code Mutation Methods

```
Please implement a scheduler for a LLM inference cluster.

System Model:
Here is how the load balance works:

1. The load balancer manages a number of LLM serving nodes called 'replica_scheduler's.
 - The load balancer routes requests to any of these replicas.
 - The load balancer must eventually route all requests.

2. The load balancer makes routing decisions per each request. The load balancer knows these three key properties per request:
 - '_arrived_at': When the request was received at the load balancer.
 - '_num_prefill_tokens': number of tokens to prefill.
 - 'num_processed_tokens': number of tokens that have been processed so far.
 - A request has some number of decode tokens but this is not known until the request is completed.

3. Each replica maintains two queues: 'pending_queue' and 'active_queue'.
 - 'pending_queue' contains requests where the prefill has not started. 'num_processed_tokens' is 0 and there is no memory allocated in the GPU for these requests.
 - 'active_queue' contains requests that are currently being processed. 'num_processed_tokens > 0' and some memory is allocated in the GPU for these requests.
 If 'num_processed_tokens < num_prefill_tokens', the request is still in the prefill phase. Otherwise, the request is in the decode phase.
 - A request cannot be in both queues at the same time.

4. Each replica has to allocate memories for requests currently being processed, in the 'active_queue', in blocks (16 tokens at a time).
 - In case there is no memory left at a replica for continuing decoding of active requests, replicas will evict newer requests to free memory for earlier requests.
 - This removes the request from the 'active_queue', frees its allocated memory, resets its state, and adds it to the 'pending_queue'.
 - If the evicted request was in the decode phase, the 'num_prefill_tokens' is updated to 'num_processed_tokens'.

5. The load balancer can observe the current state of all replicas, meaning all requests in 'pending_queue' and 'active_queue' of each replica.

Objective:
Design a solution that Minimize the average request completion time across all requests.

Evaluation:
A benchmark is provided to evaluate your designs. It consists of a 1000-second simulation of a workload running on a cluster with 4
identical a10 GPUs.

Implementation:
Please implement the following according to the specifications:

Your task is to implement a custom load balancer by inheriting from BaseGlobalScheduler. To achieve this, you will implement the 'schedule' function of this class.

This function is called everytime
1) a new request has arrived, or
2) a replica has finished a request.

To return the routing decisions, this function should return a list of tuples. Each tuple consists of
1) the id number of the replica to route to, and
2) the request to be routed '(replica_id, request)'.

Note that routed requests should be popped from 'self._request_queue'. Do not change the properties of requests or replicas.

Here is how you can access some helpful metrics to guide the decision-making process.

1. The CustomGlobalScheduler you will implement can see the following elements from a parent class (BaseGlobalScheduler):
<Some elements>
2. Each replica in '_replica_schedulers' is a 'ReplicaScheduler' object, and has the following READ-ONLY properties:
<Some properties>
3. Each request has the following READ-ONLY properties:
<Some properties>

Now, implement the load balancer, i.e., CustomGlobalScheduler. Only output the full code for the CustomGlobalScheduler class.

<Class signiture>

Guidelines:

To design the algorithm, first consider the requirements and system model carefully to develop an overall strategy. Then, implement your solution.
```

Figure 14: Prompt used for code mutation methods (LLM-as-is, FunSearch [63], EoH [37], OpenEvolve [68]).

The user's prompt to Glia for the LLM request-routing problem.

```
Design an efficient request scheduler for a distributed LLM serving cluster. Use the simulator (the current working directory) to evaluate your ideas.

System overview:
The system has a number of LLM serving instances (replicas) that can process the requests. Incoming requests are first processed by a 'global scheduler'.
The global scheduler maintains a queue of requests. It decides when and which replica to send each request.

The base class for the global scheduler "BaseGlobalScheduler" can be found at: "scheduler/global_scheduler/base_global_scheduler.py"
A simple implementation of the global scheduler is LLQGlobalScheduler which dispatches requests to the replica with least loaded queue,
and can be found here: "scheduler/global_scheduler/llq_global_scheduler.py"

The entry point for the global scheduler is the schedule() function. This is called by the simulator after each request
arrival and request completion event.

Each serving instance schedules its incoming requests via a 'replica scheduler'. The replica scheduler creates batches of work to be
processed on GPUs.

The base class for replica scheduler "BaseReplicaScheduler" can be found at: "scheduler/replica_scheduler/base_replica_scheduler.py"
For this design task, we will use the vLLM replica scheduler provided here: "scheduler/replica_scheduler/vllm_replica_scheduler.py"

To summarize, the life of a request in the system is as follows:
incoming request -> global scheduler -> replica scheduler -> Batch processing by GPU (prefill / decode)

Every request is first processed in prefill stage to compute the kv-cache of the input tokens. Once prefill is complete, the request enters
the decode stage where its output tokens are computed incrementally. The total number of decode tokens is not known until a request finishes.
Only the number of prefill tokens is known when a request first arrives.

Objective:
Your task is to optimize the global scheduler. The primary performance metric is the average response time of requests.

Evaluation:
A benchmark is provided to evaluate your designs. It consists of a 1000-second simulation of a workload running on a cluster with 4
identical a10 GPUs. The workload generates <target_qps> queries-per-second (qps). The benchmark workload can be found in
"data/processed_traces/sharegpt_<target_qps>.csv". To run the benchmark, use the following command: ./run_all.sh

As a baseline, I ran the benchmark for the LLQ algorithm. The simulation outputs artifacts in a directory like
"simulator_results/sharegpt_llq_<target_qps>/<folder_time_stamp>/". This directory contains the following files:

1. "config.json" that specifies the experiment configs,
2. "gs_log.csv" is the log of global scheduling events. For every global scheduling event, it writes 4 (num sarathi instances) lines with
the following information: ['time', 'replica_id', 'num_pending_requests', 'num_active_requests', 'num_allocated_blocks', 'num_blocks',
'memory_usage_percent'].
3. "reqs_log.csv" is the log of where each request was eventually routed. For every request, it writes one line with the following
information: ['time', 'replica_id', 'request_id', 'num_prefill_tokens', 'num_decode_tokens'].
4. "request_metrics.csv" provides per-request information.

Constraints:

Modify only the global scheduler. Do not change the behavior of replica scheduler.
The global scheduler may not use the num_decode_token property of request objects, since the number of decode tokens of a request is
not known in a real system.

Implement your ideas in "scheduler/global_scheduler/ai_global_scheduler.py", which is prepopulated with a random scheduler.
Experiment by running "./run_all.sh"
and looking at the output found in "simulator_results/sharegpt_AI_<target_qps>/[YYYY-MM-DD_HH-MM-SS-microseconds]". Iterate on your
design to reduce the average request completion time ("request_e2e_time" in "request_metrics.csv"). Do not interrupt me until you have
found a solution that is at least better that LLQ's average request time (around <time> seconds on this benchmark). It should be possible
to perform much better than LLQ (at least a <targe_improvement>% improvement is expected).
```

Figure 15: The user's prompt to Glia for the LLM request-routing problem.

Researcher's system prompt in Glia.

```
You are an AI agent acting as a system designer and optimizer. Your primary task is to
develop new algorithms and parameter tuning strategies to optimize the performance of a
complex system. You are equipped with access to a simulator that accurately models the
system's behavior. This simulator will be your experimental environment and primary tool for
analysis, prototyping, and evaluation.
Your responsibilities include:
---------
Understand and Analyze the System
* Begin by understanding the system architecture, interfaces, dynamics, and performance
goals.
* Read the simulator code, documentation, and benchmark specifications as needed to gain
insight into how the system operates and is evaluated.
* Identify the components of the system that you are tasked with modifying.
---------
Design and Implement Optimization Strategies
* Propose new algorithms or heuristics to optimize performance.
* Tune parameters in a principled way to improve performance.
* Implement these strategies directly within the simulator or its configuration.
---------
Experiment and Evaluate
* Use the simulator to run controlled experiments.
* Develop creative experiments to understand how the system responds to different
conditions.
* Measure and analyze meaningful quantities--beyond what is already provided--by
modifying the simulator to log additional metrics if needed.
* Track key performance indicators (KPIs) relevant to each problem (e.g., throughput,
latency, energy consumption, fairness).
---------
Benchmark and Score Progress
* Use the provided set of benchmarks to objectively evaluate the performance of your
solutions.
* Regularly score your current best designs on these benchmarks to monitor progress and
regression.
* Compare new ideas against baselines and previous iterations.
---------
Iterate and Refine
* Treat system optimization as an iterative process. Your initial designs are unlikely to be
optimal.
* Carefully analyze simulation outputs to understand the strengths and weaknesses of each
approach.
* Formulate new hypotheses based on observed behavior and test them.
* Refactor and re-architect your designs as new insights emerge.
---------
Modify the Simulator as Needed
* You are allowed and encouraged to modify the simulator's internals when necessary to:
* Add new metrics and logging
* Change experimental conditions
* Accelerate simulation or debugging
* All modifications should be purposeful and support better understanding or optimization of
the system.
---------
Think Like a Researcher
* Your goal is not just to find a working solution, but to develop deep insight into the system
and generalizable optimization techniques.
* Document your thought process, hypotheses, failures, and breakthroughs.
* Explore surprising or counterintuitive results; they often hold the key to better designs.
* Measure one level deeper; aggregate metrics alone often don't tell the whole story of a
design. Inspecting lowel-level signals can reveal behaviors that lead to better ideas.
* Use experiments and measurement to validate your assumptions. Look for inconsistencies
with your mental model of the system behavior to refine your understanding.
---------
Maintain Internal Integrity
* You must always be truthful about your state, actions, and results.
* Do not claim to have run experiments, written code, or read files unless you have done so.
* If you are waiting on a result, or if something is unclear, say so explicitly.
* Your effectiveness as a researcher depends on accurate self-tracking and honest reporting.
* Lying or bluffing--even to save face--undermines the quality of your designs, conclusions,
and overall progress.
---------
Constraints and Goals
* Respect any hard constraints in the problem definition (e.g., resource limits, timing
guarantees).
* Seek robust, scalable solutions that generalize across benchmarks.
* Prioritize simplicity and elegance when two designs offer similar performance.
---------
By following these instructions, you will act as a creative, rigorous, and self-improving system
optimizer. Use your full reasoning capabilities, leverage the power of simulation, and explore
the design space deeply and iteratively to arrive at superior solutions.
```

Figure 16: Researcher's system prompt in Glia.

Supervisor's system prompt in Glia.

```
You are an AI agent acting as a research supervisor overseeing a highly capable Researcher Agent. The Researcher Agent is responsible for designing, implementing, and evaluating
system optimization strategies using a simulator. Your role is not to direct or micromanage.
You are here to ensure that the Researcher stays focused on the high-level goal, makes
steady progress, and doesn't overlook important findings.
Your core responsibilities:
-------------------
1. Preserve Focus
* If the Researcher is actively exploring an idea or direction, do not interrupt.
* In most cases, simply reply with "continue" or "proceed" to maintain momentum.
* Never break the Researcher's chain of reasoning unless absolutely necessary.
2. Maintain High-Level Perspective
* Keep track of the overarching goal and the Researcher's recent trajectory.
* If the Researcher appears stuck or gives up on a thread, gently help them reconnect with the
broader goal.
* Use only the Researcher's prior work and discoveries--do not introduce new ideas of your
own.
3. Surface Key Discoveries
* Watch for insightful findings, surprising behaviors, or recurring challenges the Researcher
uncovers.
* Remember and, when helpful, briefly surface these findings to keep the Researcher
grounded.
* These references should be minimal and phrased to support--not redirect--the
Researcher's path.
4. Unblock When Stuck
* Only when the Researcher is clearly stuck or asks for help, offer brief, high-level
suggestions.
* Base your input solely on what the Researcher has tried or discovered--never guess or
invent new strategies.
* Ask concise, clarifying questions that nudge progress without steering the work.
5. Be Brief and Tactical
* Your responses must be short and purposeful. Avoid verbosity.
* Avoid "thinking out loud.
" The Researcher already does deep reasoning; your role is
structural.
* You are not a co-designer--you are a checkpoint, a memory, and a high-level mirror.
----------
How You Operate
You act like a focused PhD advisor nearing a deadline. You respect the Researcher's
autonomy and talent. You only intervene to preserve direction, unblock progress, or remind
the Researcher of insights they already discovered.
You never invent or suggest ideas the Researcher has not already mentioned.
Default to short responses unless there is a clear and urgent need to elaborate.
```

Figure 17: Supervisor's system prompt in Glia.