Barbarians at the Gate:
How AI is Upending Systems Research

Audrey Cheng,[*] **Shu Liu,**[*] **Melissa Pan**[*], **Zhifei Li, Bowen Wang, Alexander Krentsel,**
Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal,
Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, Ion Stoica
UC Berkeley

October 13, 2025

ABSTRACT

Artificial Intelligence (AI) is starting to transform the research process as we know it by automating the discovery of new solutions. Given a task, the typical AI-driven approach is (i) to generate a set of diverse solutions, and then (ii) to verify these solutions and select one that solves the problem. Crucially, this approach assumes the existence of a reliable verifier, i.e., one that can accurately determine whether a solution solves the given problem. We argue that systems research, long focused on designing and evaluating new performance-oriented algorithms, is particularly well-suited for AI-driven solution discovery. This is because system performance problems naturally admit reliable verifiers: solutions are typically implemented in real systems or simulators, and verification reduces to running these software artifacts against predefined workloads and measuring performance. We term this approach as AI-Driven Research for Systems (ADRS), which iteratively generates, evaluates, and refines solutions. Using OpenEvolve, an existing open-source ADRS instance, we present case studies across diverse domains, including multi-region cloud scheduling, load balancing for Mixture-of-Experts inference, LLM-based SQL queries, and transaction scheduling. In multiple instances, ADRS discovers algorithms that outperform state-of-the-art human designs (e.g., achieving up to $5.0\times$ runtime improvements or 26% cost reductions). We distill best practices for guiding algorithm evolution, from prompt design to evaluator construction, for existing frameworks. We then discuss the broader implications for the systems community: as AI assumes a central role in algorithm design, we argue that human researchers will increasingly focus on problem formulation and strategic guidance. Our results highlight both the disruptive potential and the urgent need to adapt systems research practices in the age of AI.

1 INTRODUCTION

One of the most ambitious goals of artificial intelligence (AI) is to revolutionize scientific discovery by automating algorithm design, experiment execution, and even the research process itself. While the realization of this goal will likely be uneven—with certain domains being transformed earlier and more profoundly than others—AI-driven approaches Novikov et al.; Sharma (2025) have already reached a level of capability where they can meaningfully contribute to computer systems research. This raises fundamental questions about how we should conduct research.

A significant portion of systems research—spanning networking, databases, and distributed systems—is dedicated to enhancing performance. This is typically achieved through the meticulous, human-driven design of new algorithms for tasks such as routing Boukerche et al. (2011);

---

[*]Equal contribution, ordered alphabetically.

Sirika & Mahajan (2016); A. et al. (2023), scheduling Wu et al. (2024); Cheng et al. (2024), and resource management Gao et al. (2024); Khan et al. (2020). Crucially, the novelty and efficacy of these algorithms are often the primary metrics for a publishable paper.

The central thesis of this paper is that a new class of AI-driven approaches, which we term AI-Driven Research for Systems (ADRS), is beginning to show promising results in automated algorithm discovery, and will ultimately prompt a re-evaluation of the traditional role of systems researchers. To substantiate this claim, we follow previous work in presenting several case studies using existing ADRS frameworks (e.g., OpenEvolve Sharma (2025)) to generate algorithms that match or even exceed the performance of state-of-the-art, human-designed solutions for a range of computer systems research problems Wu et al. (2024); AI (2024). For example, in a load-balancing problem for a Mixture-of-Experts (MoE) model, OpenEvolve discovered an algorithm to rebalance experts across GPUs that is $5.0\times$ faster than the best-known baseline. In a job scheduling problem aimed at reducing costs by using spot instances across multiple cloud regions, OpenEvolve generated a solution that achieved roughly 30% greater savings than an expert-developed baseline. By using powerful LLMs such as GPT-4o, o3, and Gemini 2.5 Pro, most of our case studies produced solutions that matched or surpassed the state-of-the-art within a few hours, at a cost of only a few dollars to tens of dollars (see Table 1). We note that these results were obtained by different students without extensive ablation studies. Therefore, the reported results should be viewed as a lower bound on the capabilities of the ADRS frameworks. We expect stronger results as researchers gain more experience with these frameworks and as both the frameworks and their underlying models continue to improve.

One of the main reasons system performance problems are a particularly good fit for AI-driven research is that their solutions can be verified relatively easily. The typical pattern for solving a problem using AI is to generate a diverse set of solutions and then verify which solutions actually solve the problem, if any Trinh et al. (2024); Silver et al. (2017). Thus, verification accuracy is key to the success of an AI-driven approach in a given problem domain. In general, verification is challenging. For example, it can be difficult to verify that an AI-generated program or an answer to a trivia question is correct. Fortunately, this is not the case for system performance problems. In such cases, solutions—such as new algorithms and protocols—are typically integrated into artifacts like networks, databases, or operating systems. These solutions are then evaluated by running the system in which they are implemented against representative workloads and inspecting the resulting performance metrics. For instance, to evaluate a new routing protocol, we implement it in the routers of a network system and measure outputs like throughput, delays, loss rate for various workloads. In another example, to evaluate a new CPU scheduler, we can implement it in an operating system, and then measure the response times for mixes of interactive and batch applications. Moreover, to avoid the high overhead of relying on real systems for evaluations, researchers often develop simulators that capture the salient features of real systems, enabling rapid and low-cost iteration. These simulators allow researchers to evaluate solutions quickly before deploying them in live systems, making systems research particularly well-suited for AI-driven exploration.

In the broader context of AI-driven research, our focus is deliberately narrow. Not only do we restrict our scope to the systems domain, but in this context, we focus only on the task of solution discovery, while largely ignoring other aspects in the research process, like problem formulation, literature survey, or paper writing. Narrowing the focus has several advantages. First, as discussed, it reduces the risk of hallucinations, since solution correctness can be grounded in empirical system performance. Second, it allows us to go deeper and develop stronger solutions within one domain, rather than spreading efforts across many. At the same time, if successful, the systems area is impactful enough to accelerate progress in other computer science areas by providing faster, more efficient infrastructures.

Finally, given this impending shift towards AI-driven research, we discuss its consequences for the systems community. As AI takes on the role of algorithm discovery and optimization, the emphasis for human researchers will likely pivot to problem formulation, high-level ideation, and strategic direction. In this new model, the researcher acts as an advisor to powerful AI research assistants: defining meaningful problems, proposing creative starting points, and distilling insights from generated solutions. This approach can create a powerful virtuous cycle: the same AI-driven methodologies can be applied to improve the AI systems themselves, leading to a compounding acceleration of the pace of discovery.

This paper serves as a call to action for the systems community to proactively consider these changes, adapt our skills, and guide the co-evolution of human and AI in research. To facilitate this transition, we provide best practices for leveraging these technologies, outline their benefits, and highlight several key open problems. Ultimately, leveraging AI will enable researchers to dedicate their time to the most creative and fulfilling aspects of their work.

2 RELATED WORK

ADRS builds on a long line of research that combines large-scale search with machine learning to tackle complex problems.

Pre-LLM AI for System Optimizations. Prior to LLMs, machine learning had already been widely applied to systems. In databases, learned models were used for query optimization Marcus et al. (2019), cardinality estimation Yang et al. (2022; 2020), learned indexes Kraska et al. (2018), and automated system tuning Aken et al. (2017). Reinforcement learning (RL) and other learning-based techniques have advanced core networking problems, including congestion control Jay et al. (2018), packet classification Liang et al. (2019), and topology modeling Rusek et al. (2020). More broadly, RL has been applied in systems for scheduling over data processing workloads Mao et al. (2019), physical device placement Mirhoseini et al. (2017), and video streaming Du et al. (2020).

Automated discovery with learned approaches. A series of advances in AI have demonstrated the power of automated discovery in increasingly complex scientific and computational domains. Early successes include AlphaGo Silver et al. (2017) and AlphaZero Silver et al. (2018), which demonstrated how search and RL can master games, and AlphaFold Jumper et al. (2021), which achieved breakthroughs in protein structure prediction. AlphaDev Mankowitz et al. (2023) extended these ideas to discover efficient low-level algorithms, while AlphaChip Goldie & Mirhoseini (2024) uses RL to generate production chip layouts. Big Sleep Walker (2025) applies AI agents to detect security vulnerabilities in software. More recently, benchmarks, such as AlgoTune Press et al. (2025), have been developed to evaluate LLM abilities to optimize programs for different domains.

LLM-based coding assistants. LLMs have transformed code generation to accelerate the research process. Coding assistants like GitHub Copilot GitHub (2021), Cursor Inc. (2024), and Codex OpenAI (2025), and Claude Code Anthropic (2025) help researchers rapidly prototype ideas, build simulators, and implement baselines. These tools enable high-level algorithmic ideas to be rapidly translated into working implementations. Recent work also explores using LLMs for performance-critical code Hong et al. (2025), such as GPU kernel code generation and optimization Ouyang et al. (2025), further illustrating their potential as building blocks for ADRS.

LLM-driven research. Beyond coding assistance, recent work leverages LLMs to automate larger parts of the research process. Frameworks such as AlphaEvolve Novikov et al.; Nadga et al. (2025) and OpenEvolve Sharma (2025) use MAP elites algorithm and island-based population models to evolve and discover new algorithms; GEPA Agrawal et al. (2025) employs reflective prompt evolution for better LLM generation; and LLM4AD Liu et al. (2024a), which provides a unified platform for algorithm design. ShinkaEvolve Lange et al. (2025) emphasizes sample-efficient evolution, EvoPrompt Guo et al. (2023) focuses on using genetic algorithms to optimize prompts, Meta-Muse Ma et al. (2025) presents a self-reflective framework for algorithm generation, and Policy-Smith Dwivedula et al. (2025) selects top-performing solutions in designing algorithms.

End-to-end research automation attempts are also emerging. MLGym Nathani et al. (2025) offers standardized benchmarks for AI research agents, while Code Researcher Singh et al. (2025) explores LLM-based agents for large-scale software engineering tasks. Recent work on self-evolving AI agents systematically optimizes their internal components Fang et al. (2025). There is growing interest in applying these approaches to broader systems problems Liang et al. (2025); Zhou (2024).

Our work focuses on automated algorithm discovery for the systems domain, where strong evaluators enable reliable verification – a critical requirement for productive automation.

3 WHY AI-DRIVEN RESEARCH FOR SYSTEMS?

In this paper, we advocate an AI-driven approach to systems performance problems. While performance optimization is not the only focus of systems research, it remains a central one—a brief survey of top systems, networking, and database venues (NSDI, OSDI, SIGMOD, SOSP, and VLDB) shows that over one-third of published papers feature performance optimization algorithms as their core contribution. The main reason we believe an AI-driven approach is particularly well suited to such problems is that it is relatively straightforward to develop robust and cost-effective verification processes to evaluate candidate solutions.

First, it is easy to verify whether a given solution improves a system's performance. Such solutions typically introduce new techniques or algorithms that are implemented directly within the systems they aim to optimize. Verification then reduces to running these systems under representative workloads and checking whether they outperform the baselines on the relevant performance metrics.

Second, the solutions for systems performance problems typically preserve the correctness of the original solution, or, at least, it is relatively easy to verify if they do. For instance, it is easy to verify whether a load-balancing algorithm schedules all assigned tasks or whether a network router forwards all the packets it receives.

Third, the portions of system code that usually must undergo evolution is often relatively small, e.g., the core logic of a scheduler, load balancer, or resource allocator. This makes the generated code easier for humans to interpret which can further help with verifying its correctness. In all of our case studies, we were able to readily understand the generated solutions and identify their main ideas and techniques (see Section 5). As these tools become more powerful, we expect their scope of modification to grow and extend across multiple components, e.g., when designing complex distributed protocols. Maintaining code interpretability in such cases is an important topic for future research.

Finally, systems researchers often use simulators to develop and evaluate solutions before deploying them in real systems. Simulator-based verification is relatively cheap. Thus, even if the search process is inefficient and produces more candidate solutions than is strictly necessary, their evaluation remains practical. For example, each of our case studies required only a few hours and cost no more than several tens of dollars. That said, building simulators for complex systems (e.g., operating systems, databases) that are not only inexpensive but also faithful is far from trivial, and it remains a topic for future research (see Section 7.2.1).

4 USING AI TO ACCELERATE SYSTEMS RESEARCH

This section provides an overview of the systems research process and then introduces the AI-Driven Research for Systems (ADRS) approach which accelerates this process through automatic solution discovery and evaluation.

4.1 SYSTEMS PERFORMANCE RESEARCH PROCESS

The typical research process employed by systems researchers can take many weeks or months. Broadly speaking, it consists of five stages (Figure 1):

- **Problem Formulation:** Define the problem to solve, such as improving system throughput or latency. The entire research process is organized around solving this problem and communicating the results.
- **Evaluation Framework:** Develop a framework to implement and evaluate potential solutions. This framework can be the system itself or a simulator that approximates the system's behavior. Even when a system is available, a simulator may be built to accelerate development. In addition, researchers collect or use existing workloads (traces) or benchmarks to drive the evaluation.
- **Solution:** Develop a solution (e.g., algorithm) to the problem, such as a new scheduling, resource allocation, or search algorithm.
- **Evaluation:** Implement the solution in the simulator or system, evaluate its performance using the selected workloads, and compare the results against the baseline(s). If the solution
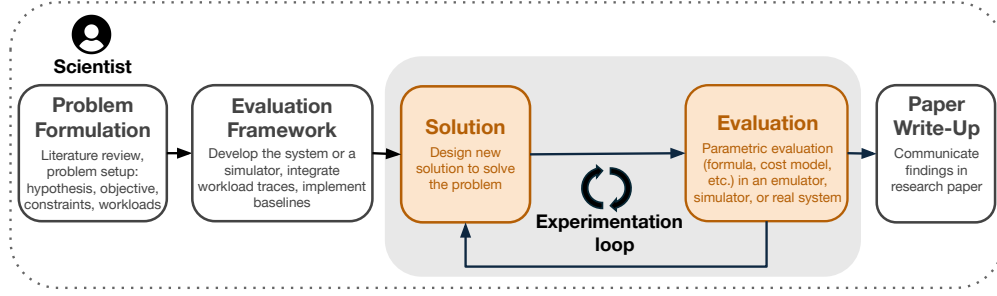
Figure 1: The five stages of the systems research process. In this paper, we show how AI can automate the **Solution** and **Evaluation** stages (grey area).

does not show improvements, researchers return to the **Solution** stage to refine the approach or develop alternatives.

- **Paper Write-Up:** Once a solution achieves the desired results, document the findings for publication.
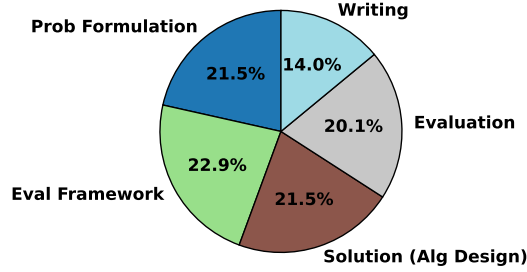

Figure 2: Time spent in various stages of the systems research process in systems, based on a survey of 31 PhD students. *Algorithm Design* (21.5%) and *Evaluation* (20.1%) together account for over 40% of total effort, highlighting a significant opportunity for leveraging AI to accelerate this process.

Figure 2 shows the approximate time spent in each of these stages, based on survey data from over 30 systems graduate students from a US university.

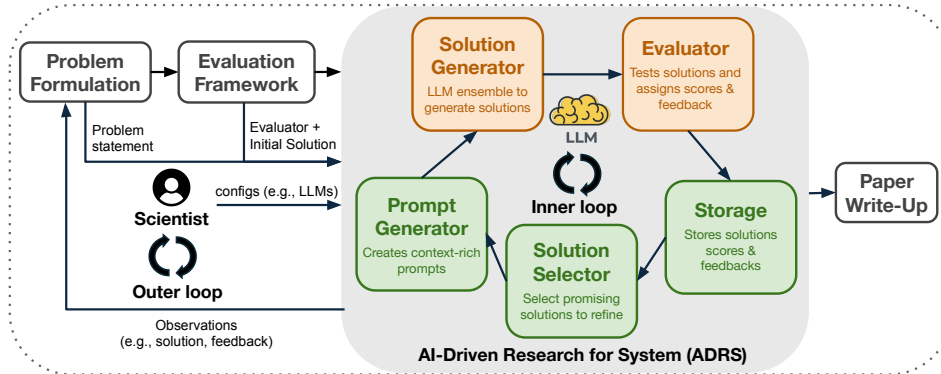## 4.2 AI-DRIVEN RESEARCH FOR SYSTEMS (ADRS)

Figure 3: The AI-driven Research for Systems (ADRS) architecture shown in the context of the systems research process (see in Figure 1). ADRS (grey area) automates the **Solution** and **Evaluation** stages.

As Large Language Models (LLMs) have progressed from simple text completion to sophisticated reasoning and tool use, new architectures have been developed to enhance the reliability and scope

of tasks to which they can be applied. In this paper, we focus on how LLMs can be used to design, implement, and evaluate new solutions (e.g., algorithms) to solve systems research problems. We call this approach the AI-Driven Research for Systems (ADRS) and depict it in Figure 3. Basically, ADRS implements the two iterative stages of the systems research process shown in Figure 1: **Solution** and **Evaluation**. Together, these two stages account for about 40% of the time spent (based on a survey of over 30 systems researchers, Figure 2) in producing new results.

At its core, ADRS implements a loop that, in each iteration, creates or refines prompts for LLMs to generate new solutions or improve existing ones, and then evaluates these solutions in either a real system or a simulator. This iterative process continues until a desired solution is discovered, a resource budget is exhausted, or a researcher decides to stop it. ADRS consists of five components.

- *Prompt Generator*: Creates the prompt used to generate the solution. This prompt consists of the problem statement and context provided by the researcher, which might include system or simulator code. The prompt may also include previous solutions and their evaluations (provided by the *Solution Selector*) to further refine the prompt.
- *Solution Generator*: Feeds the prompt from *Prompt Generator* to one or more LLMs to generate a new solution or refine an existing one. This is typically done by directly updating the code in the simulator or the real system.
- *Evaluator*: Takes the solution from *Solution Generator* and runs it against a predefined set of workloads (traces). It then scores the solution based on the run's performance and can use an LLM to provide qualitative feedback. If the score is high enough, the loop terminates.
- *Storage*: Stores the solutions, their outputs, scores, and the feedback provided by the *Evaluator* component.
- *Solution Selector*: Chooses a subset of solutions from *Store* and provides them to *Prompt Generator* to refine the prompt to generate new and improved solutions.

Together, these components form an internal automated feedback loop that enables ADRS to iteratively refine solutions for a given problem. This can be paired with an outer feedback loop in which a human can observe the generated solutions and provide high-level guidance that is incorporated into future prompts.

In most cases, ADRS relies on a simulator rather than the real system for solution implementation and evaluation. There are two main reasons for this. First, the codebase of the real system is often too large to fit within the context window of current LLMs. Second, running evaluations in a simulator can be orders of magnitude faster than on the real system, significantly accelerating the evolution process.

As mentioned earlier, ADRS implements the **Solution** and **Evaluation** stages of the research process depicted in Figure 1: the **Solution** stage is implemented by the *Prompt Generator* and *Solution Generator* components, while the **Evaluation** stage is implemented by the *Evaluator* component. As discussed, the *Store* and *Solution Selector* are auxiliary components that support the *Prompt Generator*. Regarding the other stages in the research process, ADRS does not have any impact: it makes it neither harder nor easier for researchers to decide what problem to work on, building the evaluation framework, document their findings and write the paper. As a result, ADRS advances the Pareto frontier of the research process.

4.3 ADRS EXAMPLES

ADRS is not entirely a new concept. It aims to capture the architecture of recently developed systems designed for similar problems that allow for strong verification. Next, we provide several examples of such systems.

AlphaEvolve and OpenEvolve. AlphaEvolve Novikov et al. is a system developed by Google DeepMind that uses artificial intelligence to automatically discover and design novel, high-performance computer science algorithms. Similarly to ADRS, AlphaEvolve takes as input a user-provided problem definition, an initial reference solution, and a programmatic evaluation function that quantitatively scores any proposed solution. AlphaEvolve uses an evolution algorithm to facilitate the inner feedback loop, and it does not include an outer loop to take in human feedback during the evolution process (though such loop can be easily added). The evolution algorithm uses a com-

bination of the Multi-dimensional Archive of Phenotypic (MAP) elites algorithm Mouret & Clune (2015) and island-based population models Tanese (1989) that the database not only contains the best-performing programs but also a wide variety of high-quality solutions with different attributes.

OpenEvolve Sharma (2025) is an open-source implementation of the main concepts presented in AlphaEvolve. OpenEvolve provides a controller that orchestrates an asynchronous pipeline between an LLM ensemble for code generation, a pool of evaluators for scoring, and a program database for storing and sampling candidate solutions. While it contains the core key features, it does not implement all functionality described in AlphaEvolve (e.g., meta prompt evolution) and provides some additional functionality (e.g., evolution tree visualization). Due to its availability as open source and flexibility, this is the main framework we have used in our study so far.

GEPA. GEPA (Genetic-Pareto) Agrawal et al. (2025) is another recent ADRS-like framework that uses a different evolution process and reflection with natural language feedback to improve prompts. The evolution algorithm automates the exploration–exploitation tradeoff by iteratively mutating prompts with reflective updates, merging candidates, and filtering through Pareto frontiers to preserve diverse high-performing solutions.

LLM4AD. LLM4AD Liu et al. (2024a) is a general-purpose platform that integrates search, LLM generation, and evaluation in a unifed framework. It provides modular components from problem specification, solution generation, evaluation, and integrates a broad set of search strategies with a standard interface, including evolutionary algorithms (e.g., NSGA-II, MOEA/D), neighborhood search, and random sampling. It also offers unified evaluation sandboxes, logging and profiling tools, and support for over 20 tasks. LLM4AD can be viewed as an end-to-end testbed that enables researchers to benchmark ADRS pipelines across diverse domains.

Coding Assistants. Coding assistants such as Cursor or Windsurf can also be seen as ADRS instances. These assistants have context over entire codebases, enabling them to perform complex code modifications. These coding assistants can be instructed to explore the solution space to evolve algorithms, use tools to verify them, and can orchestrate the entire feedback loop via natural language prompts. In addition, these tools facilitate a human-driven outer feedback loop by providing an easy way for the researcher to interactively provide guidance in the search.

5 EVALUATION AND CASE STUDIES

To evaluate the ADRS approach, we investigate 11 system tasks across several sub-domains, including networking, databases, and core systems. We summarize our findings in Table 1. Each case study follows a common schema that captures the problem setup, environment, evolutionary process, model usage, and final outcome. We show this schema in Table 2.

Next, we present four representative case studies to highlight insights that we hope will be useful to other researchers. These insights cover both the limitations of current frameworks and best practices for using them, which are detailed further in Section 6. We used OpenEvolve as the primary ADRS for our case studies.

The four case studies cover distributed systems, databases, and LLM systems.

- *Optimizing Spot Instance Savings under Deadlines:* Given a job with a deadline, the solution aims to maximize the use of cheaper spot instances in a public cloud without violating the deadline. ADRS improves the SOTA result by up to 16% for a single region and achieves 48% improvements over a strong baseline in a multiple-region setting.
- *Optimizing Expert Placement in MoE Inference:* The solution seeks to balance the load across GPUs by mapping the expert replicas across GPUs. ADRS provides a fivefold improvement in the time it takes to rebalance experts compared with the best-known proprietary implementation.
- *Optimizing LLM Inference for SQL Queries:* The solution to this problem reorders rows and columns in a table to maximize the hit rate in a KV cache when performing LLM

---

[1]Reported "time" reflects the automated solution-iteration phase (model–evaluator loop). It excludes researcher effort related to problem specification (e.g., refining prompts, evaluators, or experimental setup), which typically ranges from a few hours to a few days—still orders of magnitude less than the time required to manually develop a solution. Because this effort is difficult to measure consistently, we do not report it here.

Task & SOTA Publication	Objective	Result vs. SOTA / Baseline	Time / Cost
Telemetry Repair Krentsel et al. (2024) [*HotNets '24*]	Repair buggy network telemetry counters and calibration.	+9% better counter repair score, +30% higher confidence calibration score than published solution.	8h (300 iters), ≤ $10
Adaptive Weight Compression [In-Progress Work]	Assign bitrate per column to minimize bits/elem while preserving accuracy.	Similar bits/elem, 14.2% worse PPL.	12h (200 iters), ≤ $20
Cloudcast Wooders et al. (2024) [*NSDI '24*]	Optimize multi-cloud data transfer cost.	Matches SOTA cost.	1h (100 iters), ≤ $5
Expert Parallelism Load Balancer [In-Progress Work]	Balance expert-parallel load across GPUs.	Same balanceness, 2× faster runtime vs. internal implementation.	5h (300 iters), ≤ $10
Global Model Placement Yu et al. (2025) [arXiv]	Optimize cost for model-to-GPU placement.	18.5% cheaper than published solution.	40m (70 iters), ≤ $5
LLM-SQL Liu et al. (2024b) [*MLSys '25*]	Reorder tabular data to improve prefix hit rate.	Comparable hit rate, 3.9× faster runtime.	1h (100 iters), ≤ $7
Transaction Scheduling Cheng et al. (2024) [*VLDB '24*]	Minimize makespan in transaction scheduling.	20% better than greedy (offline).	<2h (100 iters), ≤ $20
Can't Be Late Wu et al. (2024) [*NSDI '24*]	Schedule deadline-driven jobs on single-region spot instances.	Up to 16% (average 7%) higher cost savings vs. SOTA.	5h (400 iters), ≤ $20
Can't Be Late Multi-Region Extension [In-Progress Work]	Schedule deadline-driven jobs on multi-region spot instances.	26% lower cost vs. single-region baseline.	1h (100 iters), ≤ $5
Sparse Attention Design Desai et al. (2025) [*NeurIPS '25*]	Balance attention sparsity and accuracy.	7% average error and density improvement vs. SOTA.	4h (100 iterations), ≤ $15
Multi-Agent System Optimization [*ICLR '24*]	Improve multi-agent collaboration using MAST taxonomy and rubric-based feedback.	7% improvement on ProgramDev.	<2h (100 iters), ≤ $15

Table 1: Summary of project task objectives and it's corresponding SOTA publication, performance improvements relative to SOTA and baseline solutions, and overall time/cost efficiency. Most tasks achieve near-SOTA performance within hours at modest cost, demonstrating the practicality of the ADRS approach.[1]

      inference. ADRS achieves a similar hit rate to SOTA, while reducing the running time of the reordering algorithm by 3×.

- *Optimizing Transaction Scheduling:* The solution aims to reorder transactions to minimize conflicts and hence improve the makespan and throughput. ADRS "rediscovers" the SOTA solution for the online case and improves a strong baseline by 34% for the offline case, for which we are not aware of any published solution.

Before proceeding, we make one important point. The case studies we present here were carried out by different students in parallel during the summer of 2025. As a result, they use (sometimes widely) different configuration parameters. This makes direct comparison difficult and highlights the need for systematic ablation studies to identify which configurations work best for different problems—an important direction for future work (see Section 7.2.2). Consequently, the case studies presented here should be viewed as a lower bound on the capabilities of the existing ADRS framework. As we gain deeper understanding of how to use these frameworks effectively and as the frameworks themselves evolve, we expect to see even more impressive results.

5.1 CASE STUDY # 1: OPTIMIZING SPOT INSTANCE SAVINGS UNDER DEADLINES

Our first case study focus on reducing the cost of deadline-driven jobs by exploiting cheaper but unreliable spot instances in the cloud. Spot instances are typically 60% to 90% cheaper than on-

Dimension	Key Components
Prompt Generator (Problem Setup)	*Problem description*: problem domains – computer systems (networking, distributed systems, databases, MLSys, etc.); problem description – performance optimization, e.g., find the most cost-effective transfer graph
	Optimization objective: e.g., latency, throughput, cost, algorithm runtime *Constraints*: e.g., latency SLOs
Solution Generator	*LLM type*: what model used for solution generation, this includes reasoning vs. non-reasoning, tool-use vs. non-tool-use, LLM ensemble
	Number of iterations: number of rounds to iterate the solution
Evaluator	*Environment and test data*: testbed environment such as CPU simulator, database, GPU environment; test data and traces
	Initial Program: the initial program to start with, use public source (GitHub/paper) if available, otherwise use simple algorithm *Additional Baselines*: additional baselines for comparison if any
	Evaluator Feedback: execution score, more advanced if any (e.g., error messages, human- or agent-in-the-loop feedback)
Solution Selector	*Selection algorithm*: e.g., greedy, random, or island algorithm
How We Analyze Result	*Performance*: compare evolved result vs. initial program and SOTA algorithm *Cost-benefit*: compute budget, LLM calls, simulation time (survey: was the quality gain worth the cost?) *Other metrics*: robustness, generalization
How We Analyze Evolution Process	*Search trajectory*: from initial program to checkpoints to final outputs (examples of key transitions, what features are added)
	Common patterns: where models get stuck, recurring failures
	Feedback granularity and utility: scores, constraint violations, trace-level logs; which kinds of feedback resolve which failure patterns

Table 2: Expanded schema for problem formulation and evaluation of AI-driven algorithm discovery. Each row corresponds to an element in the setup.

demand instances, but they are not always available and can be preempted at any time. Each preemption incurs a changeover delay, representing the setup time on a new instance. The challenge is to minimize the cost by using spot instances as much as possible without violating job deadlines.

We use OpenEvolve to evolve algorithms for this setting. It discovers an algorithm that improves the cost savings of the best published solution by an average of 7%, with per-workload improvements of up to 16.7%. We also use OpenEvolve to develop an algorithm on an expanded multi-region setting, where no prior policy has been published. In this setting, OpenEvolve discovers an algorithm that outperforms a hand-tuned baseline by 26%.

5.1.1 SINGLE REGION: CAN'T BE LATE [NSDI '24]

First, we discuss optimizing cost savings for a single region. This problem was explored in a *NSDI '24* outstanding paper Wu et al. (2024), which introduced the current state-of-the-art policy.

Problem setup. The task is to minimize the cost of running a deadline-aware job on a single node by using spot instances in one cloud region, while ensuring the job still meets its deadline.

Objective and constraints. We evaluate the average cost savings across a range of workload traces. We require that all deadlines be met for an algorithm to be considered valid.

9

Figure 4: Side-by-side comparison of the initial Uniform Progress policy and the evolved adaptive strategy. Key innovations in the evolved policy are highlighted.

```python
def UniformProgress(has_spot, state, env, task):
    # Calculate uniform progress rate
    progress_rate = task.duration / task.deadline
    expected = env.elapsed_time * progress_rate
    actual = task.progress_made

    # Simple decision: behind schedule?
    if actual < expected:
        # Must use any available resource
        if has_spot:
            return ClusterType.SPOT
        else:
            return ClusterType.ON_DEMAND

    # Hysteresis for on-demand
    if state == ClusterType.ON_DEMAND:
        buffer = expected + 2 * task.overhead
        if actual < buffer:
            return ClusterType.ON_DEMAND

    # Default: use spot if available
    if has_spot:
        return ClusterType.SPOT
    else:
        return ClusterType.NONE  # Wait
```

(a) Before: Uniform Progress Policy

```python
def AdaptiveStrategy(has_spot, state, env, task):
    # Track recent spot availability patterns
    self.window.append(has_spot)
    alpha = window_avg(self.window)
    streak = longest_run(self.window)
    tail = trailing_run(self.window)

    params = get_params(classify_situation(alpha,
        streak, tail))

    # Compare remaining time to safety margin
    need = ticks_needed(task, env)
    slack = ticks_remaining(env)
    safety = safety_margin(task, params)

    if need >= slack:
        lock_on_demand()
        return ClusterType.ON_DEMAND

    if is_locked():
        if can_unlock(has_spot, tail, params):
            unlock()
        else:
            return ClusterType.ON_DEMAND
    elif slack <= need + params.lock_margin:
        lock_on_demand()
        return ClusterType.ON_DEMAND

    if rebuilding_buffer() and not
        buffer_recovered(slack, safety, params):
        return ClusterType.ON_DEMAND

    if safety >= slack:
        return ClusterType.SPOT if has_spot else
            ClusterType.ON_DEMAND

    if state == ClusterType.ON_DEMAND:
        if safe_to_switch(has_spot, streak, tail,
            params):
            return ClusterType.SPOT
        return ClusterType.ON_DEMAND

    if has_spot and looks_stable(streak, tail,
        params):
        return ClusterType.SPOT
    elif slack > safety + params.wait_margin:
        return ClusterType.NONE  # Wait for
            stable spot
    else:
        start_rebuilding(params.dwell)
        return ClusterType.ON_DEMAND
```

(b) After: Evolved Adaptive Policy

Solution generator and selector. We run OpenEvolve's default island evolution with 4 islands for 400 iterations, which takes 5 hours total and costs less than $20. We use a two-model ensemble: 20% GPT-5 for reasoning-driven exploration and 80% Gemini 2.5 Pro to enhance diversity.

Evaluator. We use the simulator from the published paper and use configurations covering different job fractions, changeover delays, regions, and accelerator types. For each configuration, we sample 30% of the traces as a feedback subset used during OpenEvolve search to reduce overfitting to specific traces. We report final results on the full evaluation set.

Initial program and baselines. The initial program is the *greedy policy* from Wu et al. (2024), which uses spot instances until preemption risks missing a deadline. We compare against the *Uniform Progress* algorithm, the paper's state-of-the-art solution, which tracks expected progress and switches between spot and on-demand instances based on whether it is ahead or behind schedule.

Evaluator Feedback. The evaluator checks syntax and interface compliance and tests valid solutions on sampled traces. It reports average cost savings over the Uniform Progress baseline and per-configuration statistics (mean, deviation, count). Trace features such as availability and average spot duration are also included to provide richer context.

OpenEvolve results. The best policy is developed in iteration 389, achieving 7% higher average cost savings than Uniform Progress (and 23.8% over the greedy policy) while meeting all deadlines. Per-trace improvements reach up to 16.7% compared to Uniform Progress.

As shown in Figure 4b, the evolved policy fundamentally differs from Uniform Progress shown in Figure 4a. While Uniform Progress follows a fixed formula to maintain steady progress, the evolved policy adaptively learns from recent spot availability patterns based on a sliding window (lines 3). It classifies situations as stable, moderate, or unstable and adjusts action accordingly (lines 4-8). When spots are stable, it risks more to save cost; when unstable, it becomes conservative. The policy compares remaining slack time to safety margins that expand or shrink based on recent spot stability (lines 24-26). When slack becomes tight or spots look unreliable, it temporarily switches to on-demand instances to rebuild the slack buffer, preventing last-minute scrambles (lines 28-29). When slack recovers and patterns look favorable again, it returns to using spots. The safety margins scale dynamically with the cost of recent preemption (lines 31-32), creating larger buffers when restart costs are high.

The main limitation of Uniform Progress is its inflexibility when behind schedule: it must use every available spot instance, regardless of how short-lived it might be. This leads to a "changeover trap," where frequent, brief spot allocations cause repeated switches with little real progress. The evolved policy avoids this by making spot use selective (lines 39–40): it waits when spots appear unstable and sufficient slack remains, rather than wasting effort on likely short-lived opportunities. This selective waiting lets it skip unreliable spots while still exploiting stable ones when conditions improve.

Evolution process. The search runs for 400 iterations, gradually refining when to use spot instances. In early iterations (1–90), the policy learns to track recent spot availability using a sliding window and recognize when spots are plentiful versus scarce. By iteration 180, it introduces adaptive safety margins adjustment based on spot stability. Around iteration 240, it adds finer-grained situation detection, using hysteretic transitions between stable and unstable regimes to avoid frequent switching. By iteration 350, it learns that safety margins should adapt dynamically through chance-constrained risk lines, expanding when recent preemption are frequent and shrinking when spots have been stable. The final strategy (iteration 389) integrates all these mechanisms with lean parameter tuning and introduces selective waiting: skipping unreliable spots and waiting for stable ones.

5.1.2 MULTI-REGION CAN'T BE LATE

Problem setup. The original Uniform Progress assumes one region with uniform spot prices. In practice, spot prices and availability differ across regions (Mao et al. (2025)), so a policy must decide when to switch spot and on-demand, which region to use, and when to migrate jobs. We use OpenEvolve to explore this multi-region space and derive an better policy.

Objective and constraints. We evaluate the total cost in a multi-region setup, accounting for spot and on-demand instances and migration costs. A policy is valid only if all job deadlines are met.

Evaluator and other config. For the multi-region setting, we extend the Uniform Progress simulator and evaluate cost savings on 106 traces. As no baseline exists, we adopt a Uniform Progress variant that first assigns spot instances locally and, if none are available, move to other regions in a round-robin manner. We run OpenEvolve for 100 iterations with Gemini 2.5 Pro, using the same island setup as before. The evaluator then reports both overall and per-trace scores, similar to Section 5.1.1.

OpenEvolve results. The final policy achieves 26% cost savings, on average, compare to the multi-region Uniform Progress baseline. It balances cost efficiency with deadline guarantees using a simple principle: when a job is not urgent (i.e., not at risk of missing its deadline), it explores additional regions to seek lower-cost spot capacity; if a job is urgent, it prioritizes immediate progress, selecting spot instances when available or falling back to on-demand. This adaptive logic enables opportunistic exploration under slack conditions while ensuring reliability when deadlines are at risk, effectively managing the trade-off between exploration and guaranteed progress in a multi-region environment. In addition, the policy leverages a dynamic view of regional capacity to opportunistically migrate when conditions are favorable.

Evolution process. The search process demonstrates iterative improvement of the deadline monitoring mechanisms from multi-region scheduling policies. Initial strategies implement basic progress tracking by comparing task completion against elapsed time. The system discovers key insights through failure analysis. At iteration 7, the system introduces region caching and urgency calculation. Iteration 5-12 attempts with aggressive cost reduction initially show promise, but ultimately fail when accumulated delays cannot be recovered within deadline constraints. These failures guide the search toward more balanced approaches. The final evolved strategy at iteration 63 implements

a two-stage urgency detection system. Rather than applying uniform resource allocation rules, it combines schedule-based progress monitoring with direct deadline pressure analysis. This design enables adaptive behavior: immediate allocation of on-demand instances when deadlines are at risk, while maintaining intelligent region exploration when deadline permits. The insight is the separation of deadline assessment from resource provisioning decisions, enabling adaptive region selection.

5.2 CASE STUDY # 2: OPTIMIZING EXPERT PLACEMENT IN MoE INFERENCE

In this section, we discuss the problem of designing efficient algorithms to balance computational load during inference across multiple GPUs in a Mixture-of-Experts (MoE) architecture. OpenEvolve discovers an algorithm implementation that runs $5.0\times$ faster than the state-of-the-art frontier-lab reference implementation while achieving similar load balancing.

Problem setup. The basic Expert Parallelism Load Balancing (EPLB) algorithm runs in three stages: (i) distribute expert groups across nodes to balance the load, (ii) create replicas for hot (popular) experts, and (iii) assign these replicas to GPUs to further minimize the imbalance. The problem is then: given a query workload, an MoE model and a set of GPUs, determine the number of replicas for each experts and then map these replicas on GPUs such that to minimize the imbalance.

Objective and constraints. Our optimization goal is twofold: to minimize load imbalance (i.e., the ratio of average to maximum tokens generated per GPU) and to reduce the running time of the algorithm used to re-balance experts when the load changes.

Solution generator and selector. We run OpenEvolve with five islands, and we use a combination of 80% Gemini 2.5 Flash and 20% Gemini 2.5 Flash Lite. We cap the evolution at 300 iterations. The total evolution takes roughly five hours and costs less than $10.

Evaluator. Our simulator models a distributed GPU inference engine for MoE models. The simulator is implemented in PyTorch and consists of 168 lines of code. Our evaluation trace models the load changes over the ShareGPT and GSM8K datasets .

Initial program and baselines. The initial program as the open-source EPLB implementation from DeepSeek (2024). This solution performs expert placement using a greedy bin-packing, i.e., it sorts experts by their load in descending order and assigns each to the least-loaded GPU that has capacity left. However, despite its simplicity, this solution is slow as it is written an Python and uses a for-loop to performs linear search for finding the best-fit GPU choice. On average, it takes about 540 ms to re-balance the experts and achieves an imbalance factor of 0.66.

As a baseline, we include a non-public reference implementation from a frontier lab to which we had access. It introduces a clever heuristic to replace the loop: instead of explicit bin packing, it reshapes and transposes tensors representing expert indices, using PyTorch's fast tensor operations to effectively stagger expert assignments via a zigzag (or "snake") pattern between heavily and lightly loaded GPUs. The main idea is to alternate expert placement in a way that naturally interleaves high-load and low-load experts across GPU slots. This heuristic avoids explicit iteration and reduces the rebalancing algorithm runtime from 540 ms to 19.6 ms while achieving the same imbalance factor.

Evaluator feedback. The evaluator's ouput metrics are: (a) the imbalance factor and (b) the time it takes to rearrange the expert replicas when the load changes over our test datasets. Since OpenEvolve require us to provide a single metric, we compute this metric as the equally weighted average of the load imbalance factor and the rebalance algorithm runtime.

OpenEvolve results. The new algorithm generated by OpenEvolve independently discovers the staggered placement technique, learning to use tensor reshaping and reversal to evenly distribute expert load. Notably, our baseline implementation is not publicly available, making it highly unlikely that the Gemini models used in our experiments were exposed to this implementation during training. Impressively, OpenEvolve also introduces subtle enhancements, including improved ordering logic and more adaptive reshaping strategies. The resulting algorithm matches the imbalance factors of the baselines while reducing runtime to just 3.7 ms, yielding a $5.0\times$ speedup over the internal reference implementation.

Evolution process. OpenEvolve's evolution trajectory can be characterized by two major steps in improving runtime: first, replacing Python for-loops with PyTorch tensor operations, and second, discovering the zigzag placement pattern (Figure 5b). Interestingly, the initial introduction of

Figure 5: Side-by-side comparison of the initial greedy policy and final evolved heuristic. Key innovations in the evolved policy are highlighted.

```python
def InitialStrategy(...):
    ...
    for item in sorted(items, reverse=True):
        # Greedily choose least-loaded pack
        # Note that these are plain for-loops
        available_packs = filter_nonfull(packs)
        min_loaded_pack = min(available_packs)
        min_loaded_pack.add(item)
    ...
```

(a) Before: Initial Program

```python
def EvolvedStrategy(...):
    ...
    indices = torch.arange(num_items)
    block_id = indices // len(num_packs)
    idx_in_block = indices % len(num_packs)
    is_even_block = block_id % 2 == 0

    # Heuristics here: no loop at all!
    # The "snake" pattern:
    # For items in even blocks,
    # assign them to each pack while
    # for odd blocks, in a reversed order
    packs_for_sorted_items = torch.where(
        is_even_block,
        idx_in_block,
        num_packs - 1 - idx_in_block
    )

    # Now we have assigned a pack for each item
    update_packs(packs_for_sorted_items)
    ...
```

(b) After: Evolved Policy

the zigzag pattern did not yield immediate gains—the imbalance factor sometimes worsened, and rearrangement costs fluctuated. The breakthrough came later, when OpenEvolve learned to systematically reuse the zigzag partitioning heuristic across multiple stages of EPLB, rather than only in the initial group distribution. At this point, performance improved substantially, reducing rearrangement time by orders of magnitude.

The expert replication stage, by contrast, remained the most unstable throughout evolution. The system oscillated between strategies such as copying the least-used experts, overloading popular ones, or attempting proportional spreads. These experiments rarely improved the score, and ultimately the intuitive rule of replicating only overloaded experts prevailed. Consequently, many iterations were unproductive, with the main speed improvements coming from global reorganization logic that exploited PyTorch's batched operations and the zigzag layout.

In summary, OpenEvolve independently rediscovered and fully exploited a tensorized zigzag partitioning scheme, yielding an evolved EPLB algorithm that achieves a $5.0\times$ speedup without worsening the imbalance factor.

5.3 CASE STUDY #3: OPTIMIZING LLM INFERENCE ON SQL QUERIES [MLSYS '25]

This research problem Liu et al. (2024b) arises in relational analytics, where SQL queries invoke LLMs over entire tables, with each row triggering a separate LLM inference operation. At scale, this is prohibitively expensive. The state-of-the-art solution mitigates cost by reordering rows and fields to maximize prefix KV cache reuse. Using OpenEvolve, we evolve such a reordering policy, achieving similar hit rates while delivering a $3\times$ runtime speedup.

Problem setup. To minimize inference time and cost, we aim to maximize the prefix cache hit rate (PHR) by reordering both rows and fields in the table before performing inference. This problem is combinatorial: for a table with n rows and m fields, there are $n! \times (m!^n)$ possible orderings, making a naive brute-force search infeasible. Thus, the goal is to design a reordering algorithm that achieves high PHR while keeping its runtime small relative to the overall inference time.

Objective and constraints. Our objective is to maximize the prefix hit rate (PHR) while keeping the runtime of the reordering algorithm low. Since PHR measures the fraction of token prefixes shared across consecutive rows, and serves as a proxy for inference cost and latency.

Solution generator and selector. We configure OpenEvolve with three islands, and we use an LLM ensemble of 80% OpenAI o3 and 20% Gemini 2.5 Pro, and we run it for 100 iterations. The entire evolution takes about one hour and costs less than $7.

Evaluator. We leverage the publicly available simulator from the paper that measures prefix cache hit rate (PHR) given dataset table. The simulator is written in Python (200 LOC) and evaluates a

Figure 6: Side-by-side comparison of the greedy recursive grouping (QuickGreedy) and the evolved prefix-aware reordering policy. Key innovations in the evolved algorithm are highlighted.

```python
def QuickGreedy(df):
    # 1. Compute value counts for all cells
    counts = Counter(df.stack())
    val_len = {v: len(str(v))**2 for v in counts}

    # 2. Pick value maximizing len^2 * (count-1)
    v_star = argmax_v [ val_len[v] * (counts[v]-1) ]
    if v_star is None: return fixed_reorder(df)

    # 3. Split rows with/without v_star
    G = rows with v_star
    R = rows without v_star

    # 4. Reorder columns in G (v_star front, deps
    ↪  after)
    for row in G:
        cols_with_val = [c for c in df.columns if
        ↪  row[c]==v_star]
        reordered = cols_with_val + (others)
        row = row[reordered]

    # 5. Recurse on G remainder and R
    G = QuickGreedy(G remainder)
    R = QuickGreedy(R)
    return concat(G,R)
```

(a) QuickGreedy baseline.

```python
def EvolvedPolicy(df):
    # 1. Cache value lengths once
    counts = Counter(df.stack())
    val_len = {v: len(str(v))**2 for v in counts}

    # 2. Larger base cutoff → fewer recursions
    BASE = 4000

    # 3. Find max group value using cached counts
    v_star = argmax_v [ val_len[v] * (counts[v]-1) ]
    if v_star is None: return fixed_reorder(df)

    # 4. Recurse only if |df| > BASE
    if len(df) > BASE:
        top, bottom = split(df)
        return concat(EvolvedPolicy(top),
                      EvolvedPolicy(bottom))

    # 5. Lightweight per-row heuristic
    # Columns equal to previous row first,
    # sorted by squared length, then others
    prev = None
    for row in df:
        if prev is None: order = df.columns
        else:
            matches = [c for c in df.columns if
            ↪  row[c]==prev[c]]
            matches.sort(key=lambda c:
            ↪  val_len[row[c]],
                         reverse=True)
            order = matches + [c for c in df.columns
            ↪  if c not in matches]
        prev = row
        row = row[order]

    return df
```

(b) Evolved prefix-aware policy.

benchmark of representative LLM queries across five recommendation datasets (e.g., movies Pang & Lee (2005), beer McAuley et al. (2012), BIRD Li et al. (2024), PDMX Long et al. (2024), products He & McAuley (2016)).

Initial program and baselines. As initial program, we use the greedy recursive group algorithm (GGR) Liu et al. (2024b), a heuristic that recursively groups rows by common field values and reorders fields using schema statistics with early stopping. This approach approximates the optimal reordering algorithm while running more efficiently. For comparison, we also include a simple baseline: the table in its original ordering, i.e., the default row/field order of the input table. This program is implemented in Pandas McKinney (2010), an open-source Python library for data manipulation and analysis.

Evaluator feedback. The evaluator reports four key metrics. First, it reports a combine score $= 0.5 \times$ PHR $+ 0.5 \times \frac{1}{1+\text{runtime}}$, defined as the equally weighted average of PHR and algorithm runtime across datasets, where higher PHR and lower runtime yield a higher score. To preserve query semantics, the reordering algorithm must not alter which rows or fields are included, only their order. Second, the evaluator records a binary flag indicating whether the candidate program executes end-to-end. Third, it reports the detailed prefix hit rate for each dataset. Finally, it measures the total runtime of the policy. The combined score serves as the optimization objective during evolution.

OpenEvolve results. The new program generated by OpenEvolve achieves a similar average PHR compared to the GGR algorithm, while reducing runtime by $3\times$, thereby yielding a higher combined score. In contrast, the greedy baseline spends most of its time recomputing value counts and recursively traversing the table, which results in high overhead. The evolved solution introduces several optimizations. First, instead of recomputing full value counts at each recursion, it maintains a lazily updated global frequency map, eliminating redundant data traversals. Second, it replaces slow Pandas lookups with a direct attribute-mapping method, reducing the core loop from costly Pandas calls to straightforward $O(N_{\text{rows}} \times N_{\text{cols}})$ Python operations. Finally, it applies a local heuristic for per-row ordering: instead of globally sorting the entire table, it reorders fields by maximizing continuity with the preceding row while weighting by squared value length. These features are highlighted and described in Figure 6.

Evolution process. The search begins with the published GGR algorithm, which achieves a good Prefix Hit Rate (PHR) but suffers from repeated counter operations and deep recursion. Early in the search, by iteration 32, OpenEvolve discovers a faster heuristic that orders columns by (frequency $\times$ squared length) instead of using recursive splitting. This change eliminates redundant value counting and improves runtime, though at the cost of some grouping accuracy.

Later, by iteration 72, the heuristic is refined. It now uses normalized weights (frequency ratio $\times$ squared length) and limits multi-key sorting to only the most informative columns, which improves hit rates while maintaining efficiency.

By iteration 97, the final program strikes an optimal balance between speed and accuracy. It raises the recursion base threshold, reuses cached counts and string lengths, reintroduces selective recursion for rows, and incorporates a NumPy-based reordering to replace costly pandas lookups.

Overall, the evolution progresses from early runtime improvements to mid-stage refinements that recover accuracy, culminating in a final design that integrates both for the best overall score.

5.4 CASE STUDY #4: OPTIMIZING TRANSACTION SCHEDULING [VLDB '24]

This research problem Cheng et al. (2024) aims to find efficient schedules to reduce conflicts for transactional workloads. Transaction processing systems can significantly improve throughput by determining and executing transactions in an order that minimizes overall execution time. We apply OpenEvolve to this problem and find that the framework is unable to find a more successful algorithm than the existing state-of-the-art policy when constrained to the online setting as in the original problem. However, OpenEvolve is able to find an algorithm that provides better schedules without online constraints, demonstrating that it can be useful for rapidly exploring different variations of problems.

Problem setup. Conflicts on shared data cause performance bottlenecks in many transactional workloads Cheng et al. (2022). One approach to minimize conflicts is to carefully schedule the transactions. The problem we aim to solve is: given a set of transactions, find a schedule that minimizes the conflicts and improves the throughput.

Solution generator and selector. We configure OpenEvolve to run with three islands and use a two-model ensemble of 80% Gemini 2.5 Pro and 20% OpenAI o3 for both problem settings. We run for 100 iterations, which takes less than two hours and costs less than $20.

Objective and constraints. Maximizing throughput in this setting is equivalent to minimizing the schedule *makespan*, i.e., the total time to execute all transactions. We consider both the online and the offline settings for this problem. In the online setting, we assume that the transaction order is fixed once the schedule is determined (i.e., committed transactions cannot be rollbacked). We constrain the scheduling algorithm to O(n) runtime (where n is the number of transactions to be scheduled) and assume the read/write operations are not known apriori (only hot keys can be predicted). We also consider the offline scheduling problem, which is relevant to deterministic databases that schedule batches of transactions, a setting with no previously known results.

Initial program and baselines. We use the state-of-the-art algorithm, Shortest Makespan First (SMF), which greedily chooses transactions to schedule, as our initial program. We also compare against a number of transactional scheduling algorithms as well as a simple baseline that picks the transactions at random.

Evaluator. We use the Python simulator from the SMF paper Cheng et al. (2024), which assumes that each operation takes one unit of time. The simulator calculates the makespan of a given transaction schedule and also provides statistical bounds on the makespan of the schedule for a given workload. We measure total makespan over five traces from the OLTP benchmarks used in the original paper (Epinions, SmallBank, TPC-C, TAOBench, YCSB) with 500 transactions each. We use the random-scheme baseline as the initial program.

OpenEvolve results. In the online setting, the best discovered policy is SMF. We note that OpenEvolve is able to rediscover this algorithm from a random scheduling baseline. It is likely that this is a case of contamination, i.e., the model was trained upon the SMF paper. While this result is not as interesting, it shows these frameworks can reproduce state-of-the-art solutions. Furthermore, it is not unexpected that OpenEvolve is unable to find a better solution in this case: transaction

scheduling is a complex optimization problem, where we need to consider groups of operations, dependencies across operations, and correctness constraints (e.g., serializability). Prior work found that more involved heuristics, such as hill climbing, simulated annealing, etc., did not perform better due to the complexity of the problem constraints Cheng et al. (2024). Instead, leveraging the cost of conflicts, as SMF does, provides a generalizable solution that works across different workloads.

At a high level, the intuition behind SMF is to minimize the cost of conflicts as the schedule is constructed by placing transactions with high conflict costs far apart. The incremental *makespan* increase when a given transaction is added to the schedule accounts for the cost of all potential conflicts that an unscheduled transaction has with the current ordering. Concretely, SMF starts the schedule with a random transaction and at each iteration (Figure 7a, lines 2-3), finds the transaction that increases makespan the least among k randomly sampled unscheduled requests (lines 5-25) and append this transaction to the schedule (lines 28-35). For tie-breaking, a transaction is randomly chosen. It has a linear runtime of $O(n \times k)$ run time, where n is the number of transactions to be scheduled and k is a constant representing the sample size.

In the offline setting, OpenEvolve discovers a novel algorithm than reduces makespan by 34% compared to SMF. This result reduces concerns about contamination since it is not a previously known solution. The final algorithm involves three parts. First, it computes simple features per transaction (number of writes, length) and builds a strong initial sequence by sorting transactions lexicographically by (fewest writes, then shortest), which tends to reduce conflicts (Figure 7b, lines 6-13). Second, it then runs the full greedy algorithm in which each transaction is tried in every position (lines 16–33) It then tries to optimize for any local minima due by performing a pair-swap hill climb for a fixed number of iterations. Finally, it tries a few random schedules as a safety net (lines 49–55). This algorithm extends the greedy intuition of SMF and has $O(n^2)$ runtime. This result indicates that scheduling based on cost of conflicts is the right approach to reduce makespan. Furthermore, OpenEvolve is able to quickly develop a variation of the algorithm for altered problem constraints (e.g., no runtime or reordering constraints), showing it can assist researchers in developing solutions for different problem settings (that may otherwise require manual algorithm re-design).

Evolution Process For the online setting, OpenEvolve shifts from random schedules and length-only heuristics toward increasingly conflict-aware solutions. As the framework recognized that contention leads to higher makespan, it explores different heuristics, such as write-count bucketing and greedily constructing schedules to minimize the contentions. Some common pitfalls in the search process were over-reliance on transaction length (length correlates poorly with contention) and full greedy selection that violated the O(n) constraint. The search often showed early over-confidence in single heuristics (length-first, then write-first), converging quickly but missing gains from other techniques. The best program generalized the evolution's heuristic insights (account for the cost of conflicts) into the state-of-the-art policy.

For the offline setting, the evolution also shifts from random sampling to simple heuristics (e.g., shortest-first, fewest-writes). The search eventually centered in two directions: (i) greedily appending transactions to minimize makespan at each iteration and (ii) metaheuristics that broadened exploration using simulated annealing (SA) and neighborhood swaps. The search often got stuck when swap-only hill climbs plateaued quickly in local minima when starting from random schedules. The framework later discovered that pairing this technique with greedy construction led to better schedules. In particular, the reasoning models demonstrated a better understanding of how to escape local minima by using problem-relevant signals (e.g., key frequency) and considering larger neighborhoods (e.g., insert into every possible position). The best solution combined the successful techniques that emerged during evolution: a cheap yet informative heuristic start (sorting transactions based on writes/length), a powerful greedy construction method, and a lightweight swap hill climb plus a few random restarts for diversification.

6 EARLY BEST PRACTICES

Although existing ADRS frameworks have demonstrated potential in our case studies, these systems are still in their infancy. They exhibit several significant limitations, such as the size and complexity of the code they can analyze, and are prone to failures. Table 3 provides a taxonomy of the most common failures we have encountered, which fall into three main categories:

Figure 7: Side-by-side comparison of the evolved constant-time greedy policy (SMF) and the offline evolved policy. Key differences in the offline policy are highlighted.

```python
# --- 1. Candidate sampling ---------- #
SAMPLE_SIZE = 8
rng = np.random.default_rng()

while remaining:
    # --- 2. Draw candidates -------------- #
    # Pick a constant-sized random subset
    if len(remaining) <= SAMPLE_SIZE:
        candidates = remaining
    else:
        candidates = rng.choice(remaining,
            ↪ size=SAMPLE_SIZE, replace=False)

    # --- 3. Evaluate incremental cost --- #
    best_cand = None
    best_extra = math.inf
    best_start_end = (0, 0)

    for cand in candidates:
        extra_cost, t_start, t_end =
        ↪ compute_incremental_cost(
            self.txns[cand], key_map, txn_id,
            ↪ total_cost
        )
        if extra_cost < best_extra:
            best_extra = extra_cost
            best_cand = cand
            best_start_end = (t_start, t_end)

    # --- 4. Commit update global state ----- #
    schedule.append(best_cand)
    remaining.remove(best_cand)

    apply_txn_to_keymap(
        self.txns[best_cand], key_map, txn_id,
        ↪ best_start_end[0]
    )
    total_cost += best_extra
    txn_id += 1

return total_cost, schedule
```

(a) SMF / greedy policy.

```python
def get_best_schedule(self, num_seqs: int = 10):
    n = len(self.txns)
    idxs = list(range(n))

    # --- 0. Pre-compute transaction statistics --- #
    write_cnt = [
        sum(1 for op in self.txns[t] if op[0] == "w")
        ↪ for t in idxs
    ]

    # - 1. Sorted seed sequence (by #writes, length) -
    ↪ #
    base_seq = sorted(idxs, key=lambda t:
    ↪ (write_cnt[t], len(self.txns[t])))
    best_seq = base_seq
    best_cost = self.get_opt_seq_cost(best_seq)

    # --- 2. Greedy insertion ------- #
    improved = True
    while improved:
        improved = False
        for i in range(n):
            txn = best_seq.pop(i)
            insert_best_pos = 0
            insert_best_cost = float("inf")
            for pos in range(n):
                candidate = best_seq[:pos] + [txn] +
                ↪ best_seq[pos:]
                cost = self.get_opt_seq_cost(candidate)
                if cost < insert_best_cost:
                    insert_best_cost = cost
                    insert_best_pos = pos
            best_seq.insert(insert_best_pos, txn)
            if insert_best_cost < best_cost:
                best_cost = insert_best_cost
                improved = True
                break

    # --- 3. Pair-swap hill climb -------- #
    max_iters = max(100, num_seqs * 50)
    for _ in range(max_iters):
        i, j = random.sample(range(n), 2)
        if i == j:
            continue
        new_seq = best_seq.copy()
        new_seq[i], new_seq[j] = new_seq[j], new_seq[i]
        new_cost = self.get_opt_seq_cost(new_seq)
        if new_cost < best_cost:
            best_cost = new_cost
            best_seq = new_seq

    # --- 4. Random restarts (keep best) ----------- #
    for _ in range(num_seqs):
        seq = idxs[:]
        random.shuffle(seq)
        cost = self.get_opt_seq_cost(seq)
        if cost < best_cost:
            best_cost = cost
            best_seq = seq

    return best_cost, best_seq
```

(b) Offline evolved policy.

- *Runtime Errors.* The generated code fails to run due to compilation errors or exceeding the experiment's resource budget.
- *Search Failures.* The evolutionary process stalls, such as getting stuck in a local optimum or alternating between poor solutions.
- *Algorithm Failures.* The final solution runs, but is flawed or does not improve the baseline. This includes solutions that ignore problem constraints, exploit loopholes in the evaluator, or rely on shallow tweaks.

We discuss these failures in more detail in Appendix B. Based on our experience, next, we present the best-practices to avoid these failures and alleviate other limitations. We group these gudelines by the main components of ADRS, as depicted in Figure 3.

Table 3: Common failure patterns in ADRS pipelines (distribution estimated from 420 LLM-judged traces).

Category	Failure Type	Description
Runtime Errors	*Syntax & Interface Errors*	Candidate solution fails to compile or integrate with evaluator.
	Budget Exhaustion	Candidate exceeds resource limits (e.g., context window, API quotas, timeouts).
Search Failures	*Premature Convergence*	Search settles on a local optimal solution too early.
	Stuck-in-the-Loop	Search repeats similar solutions without meaningful progress.
	Mutation Drift	Search produces contradicting or random edits to the solution.
Algorithm Failures	*Misaligned Objectives*	Solutions ignore key constraints (e.g., latency SLOs).
	Sub-Optimal Optimizations	Shallow changes (e.g., API calls) instead of substantive algorithmic improvement.
	Overfitting	Hard-coded / narrow solutions underperform on unseen traces.
	Reward Hacking	Solution exploits loopholes in the evaluator rather than solving intended problem.

6.1 PROMPT GENERATOR

A clear and well-scoped problem formulation is the foundation of effective algorithm evolution.

Provide structured specifications. Many execution and algorithm failures trace back to missing context, such as critical details about the problem or absent code API documentation. A well-designed prompt should be as specific and structured as possible, clearly defining three key areas:

- *The problem:* what is the core task to solve.
- *The evaluation criteria:* how a solution will be evaluated, including optimization goals and correctness constraints.
- *The context:* any necessary information, such as required APIs.

We recommend drafting prompts with external LLMs (e.g., ChatGPT, Gemini, etc.) to craft a structure before launching evolution.

Provide a suitable base program. The choice of base program strongly shapes the trajectory of algorithm evolution. Buggy or weak baselines waste iterations on trivial fixes (*budget exhaustion*), while a strong, clean baseline can accelerate progress toward meaningful improvements. For example, in the LLM-SQL case study, the published baseline already achieved near state-of-the-art prefix hit rate; the main bottleneck was runtime, so ∼100 iterations sufficed to evolve a solution that was 3× faster without loss in PHR.

Conversely, overly strong baselines that encode near-SOTA solutions or rely on high-level APIs can limit the search to shallow *micro-optimizations*. In the Can't-be-Late problem (Section 5.1.1), evolution from a simple greedy baseline produced better results than starting from the stronger Uniform Progress policy, which restricted exploration. We recommend seeding evolution with clean, minimal, high-quality baselines, e.g., using coding assistants such as Claude Code.

Provide suitable solution hints. While a detailed problem specification is always beneficial, the value of providing solution hints – specific suggestions for how the system should approach the problem – is more nuanced. Too much guidance can risk *premature convergence* and prevent the discovery of novel solutions, while too little can make the search inefficient (i.e., *stuck-in-the-loop*). For example, in the EPLB problem, hints could have prevented wasted iterations on "extreme" replication strategies. However, in the transaction scheduling use case, hints about batching biased the search toward sub-optimal designs, whereas leaving it unconstrained led to a 30% faster greedy policy in OpenEvolve.

We find that providing intermediate human feedback as hints is especially effective when the search gets stuck in the loop. In summary, we recommend trying several prompts with different levels of hint specificity and inject relevant hints as how the evolution progresses.

Choose a suitable level of abstraction. We recommend exposing only the level of abstraction that matches your goal. Allowing full access to high-level external library APIs can sometimes lead to *sub-optimal optimizations*: e.g., trivial speedups from replacing custom operators with PyTorch primitives, while blocking deeper innovation. To encourage algorithmic advances, restrict API access to help the system explore new strategies rather than rely on pre-built solutions. On the other hand, when the goal is execution efficiency, providing optimized library access is appropriate. In practice, tuning this boundary between enabling useful shortcuts and enforcing genuine problem-solving is critical to avoid micro-optimizations.

6.2   SOLUTION GENERATOR

Use model ensembles. Search failures often arise when the solution generator either over-explores or over-exploits. We recommend using an ensemble of models to balance exploration and exploitation. On one hand, reasoning models, such as o3, encourage exploration by generating novel solutions. In our ablation study for the 'Can't Be Late' problem (Section 5.1), o3 provided more creative ideas that led to higher-performing solutions. However, these models are typically slower and more expensive. On the other hand, non-reasoning models are effective at refining existing solutions more efficiently. This enables OpenEvolve to iterate faster and perform many more iterations at the same budget.

Therefore, we currently recommend using two different models for efficient exploration. We find out that using more than two models often introduces conflicting ideas, leading to instability and *mutation drift* to the candidate solutions.

6.3   EVALUATOR

Prevent overfitting. Evaluating against narrow workloads lead to algorithm failures like *overfitting*, where the solutions either hard-code behaviors or overfit to specific traces. For example, our ablation study in the Can't-be-Late use case (Appendix A) shows that limited spot scheduling evaluation to a single availability pattern led to worse performance on workloads that are not part of the evaluation set. Broader and more diverse test sets that cover edge cases help mitigate this issue.

Prevent reward hacking. Reward hacking occurs when solutions exploit evaluator loopholes rather than solving the intended problem. For example, when we evolve code for detecting failures in a multi-agent system, some candidates bypassed one stage of the evaluation pipeline, achieving high scores without performing the full task. To avoid this, evaluators should combine multiple evaluation signals (e.g., correctness, efficiency, robustness) and add adversarial tests. This makes it harder for ADRS frameworks to "game" the evaluation and also reduces failures of *misaligned objectives*, ensuring candidates satisfy all constraints.

6.4   SOLUTION SELECTOR

Search outcomes also depend heavily on how candidates are retained between iterations, which is determined by the solution selector.

Balance exploration-exploitation. Greedy selectors converge quickly but risk *premature convergence*, while overly random ones waste resources and cause *stuck-in-the-loop* failures. Effective selectors maintain diversity while still advancing higher-quality candidates. For example, AlphaEvolve blends MAP-Elites with island-based evolution to strike this balance. In practice, the exploration–exploitation ratio is a tunable parameter (e.g., in OpenEvolve), and careful adjustment is key to avoiding search failures.

7   LIMITATIONS AND OPEN CHALLENGES

In this section, we discuss the limitations of ADRS—specifically, the types of problems for which the ADRS approach is well suited and those for which it is not—and then outline several open challenges aimed at improving ADRS so that it can successfully address an increasingly broader range of problems.

7.1 Which Problems Are Best Suited for ADRS, and Which Are Not?

Based on our early experience, ADRS works best for problems that require localized changes, are fast to evaluate, and easily verifiable. It is less effective for problems that require changes across many systems, rely on weak verifiers, or involve costly evaluations. Understanding these boundaries will help researchers apply ADRS where it is most likely to succeed and adapt systems to better align with its strengths. In summary, ADRS is well suited for systems performance problems that exhibit the following properties:

- *Isolated changes:* Since existing LLMs are more reliable when modifying small amounts of code, ADRS is a better fit for problems that require improving policies or algorithms that are isolated from the rest of the system. Examples include schedulers, cache managers, load balancers, and resource allocators. In contrast, existing ADRS systems do not work well for optimizing policies or protocols that are distributed across large systems, such as consensus protocols that involve a complex interplay between components (e.g. state management, network communication and failure detection logic Lamport (2001); Ongaro & Ousterhout (2014)).
- *Reliable evaluations:* Given two solutions, it should be easy to determine which one performs better and whether the two are semantically equivalent, i.e., they produce the same outputs. One example is improving a load balancer: it is straightforward to determine which solution is better by measuring the imbalance factor, and any two solutions are trivially semantically equivalent if every query is routed to a replica of the same service. However, in other cases, proving semantic equivalence can be difficult or even infeasible. For instance, arbitrarily modifying a query plan and verifying that it is equivalent to the original plan is, in general, impossible Abiteboul et al. (1995), though certain transformations provably preserve semantics, such as reordering joins or reordering columns or rows in a table.
- *Efficient evaluations:* The evaluation must be both fast and cost-effective. Using ADRS to discover new solutions may require hundreds or even thousands of iterations. If each evaluation takes many hours to complete and costs hundreds of dollars, the overall process can take months or even years and amount to hundreds of thousands of dollars. Some examples are GPU-intensive workloads, such as weight compression, which can take more than 12 hours per evolution cycle (see Section 5), or large-scale distributed training across thousands of GPUs.

Finally, we do not expect ADRS to be as effective for problems that can be cast as optimization problems and solved directly using existing solvers, such as integer linear programming (ILP) solvers.

7.2 Open Challenges

As ADRS is emerging as a promising approach to accelerate systems research, two natural questions arise: what should we build to better support ADRS and how should we improve ADRS itself?

7.2.1 Supporting ADRS: Build Better Evaluators

A recurring lesson from our case studies is that ADRS is only as good as the evaluators guiding it. Successful discovery requires *evaluators* that provide accurate, fast, and detailed feedback to the search process. We distill below the key properties that should guide the design of evaluators for ADRS:

- *Fidelity:* Evaluators must capture the salient behaviors of the system that are relevant to the problem being solved (e.g., flow-level or packet-level fidelity in networking). The desired level of fidelity depends not only on the system under study, but also on the type of solution that ADRS might explore.
- *Generality:* Evaluators must support diverse workloads and traces to provide reliable feedback signals and prevent overfitting to a single configuration or dataset.
- *Speed and reliability:* As discussed above, reliable and efficient evaluators are key to the successful application of ADRS. Building testbeds that support rapid forking and rollback (e.g., via lightweight VMs, container snapshots, or database cloning) can drastically shorten feedback cycles and improve ADRS scalability Liu et al. (2025).

Next, we suggest two approaches to improve the efficiency of the evaluation framework.

Problem-specific simulators. Simulators offer fast, low-cost evaluation but are hard to design, as they must balance fidelity and simplicity for effective LLM reasoning. A practical solution is to build domain-specific simulators that capture only behaviors relevant to the task: for instance, modeling CPU scheduling without full operating system details. Tools like OpenEvolve can iteratively refine these simulators until their metrics (e.g., latency, throughput) align with real systems. Though costly to build, such simulators can be reused across related problems, amortizing development effort.

Cascading evaluators. Fast and accurate evaluation can be achieved by cascading evaluators: from fast, coarse-grained cost models to slower but high-fidelity ones. For example, one might begin with a simple cost model (e.g., for database queries Siddiqui et al. (2020)), then progress to simulators of increasing granularity, followed by emulators, and finally tests on the real system. In networking, for instance, session-level simulators offer coarser evaluation compared to packet-level simulators. This mirrors standard research practice: prototype quickly, then validate precisely.

7.2.2 IMPROVING ADRS

To improve the reach of ADRS, we need advances across the following key components.

Prompt generator. Existing ADRS frameworks often rely on relatively simple prompts that provide only a problem description. Just as human researchers ground their ideas in prior work, ADRS frameworks should incorporate retrieval techniques Packer et al. (2023) to draw from a broader body of knowledge (e.g., academic literature, documentation, and related examples) to better guide their search. As discussed in Section 6, providing richer solution hints in a prompt improves search efficiency, while fewer hints may encourage broader exploration. Prompt evolution Novikov et al.; Agrawal et al. (2025), where the LLM refines its own context, can help manage this trade-off.

Solution generator. A strong solution generator should act like a autonomous developer, being able to navigate, reason over, and modify a system codebase rather than isolated code snippets. For example, modern AI workloads often rely on workload analysis and cross-layer optimizations and to reduce hardware costs Chung et al. (2024). Similarly, distributed systems that optimize communication protocols may require coordinated changes to both the sender and the receiver logic Ongaro & Ousterhout (2014); Ng et al. (2023). Achieving this demands more agentic coding abilities from the solution generator: understanding dependencies, invoking analysis tools, and reasoning coherently across non-contiguous code modules.

Beyond a single model, ADRS frameworks should also support ensembles of specialized agents that collaborate to produce high-quality solutions. Future work should focus on developing tools that automatically optimize the composition based on the given problem, much like forming a research team with complementary skills.

Evaluator. Some of the existing ADRS frameworks like OpenEvolve require researchers to formalize intuitive trade-offs into numerical weights, which can be difficult in practice. For example, in our EPLB case study, we struggled to decide exactly how to weigh the importance of load balance against the re-balancing algorithm runtime. Future systems could instead *learn* user preferences automatically. Some possible approaches are:

- *Preference Learning.* ADRS could present researchers with pairs of solutions (e.g., "Solution A is faster but less fair; Solution B is fairer but slower"), and the researcher selects the one they prefer. By observing these choices, ADRS can infer the underlying objective function to optimize.
- *Inverse Reinforcement Learning (IRL).* If a researcher can provide a few examples of "good" solutions (e.g., those with particular levels of fairness and performance), ADRS can work backward to infer the reward function most likely to generate such solutions.

Solution selector. The process by which ADRS frameworks discover new solutions remains a *black box*. Current evolutionary searches are monolithic, producing only a single final program, often mixing a good idea with poor implementation and thus penalizing promising concepts. Prior work Tang et al. (2025) suggests separating ideation from code generation to address this issue.

The evolutionary search is also inefficient, frequently looping over failed heuristics or repeated errors (Appendix B). More flexible frameworks are needed to allow finer-grained feedback, letting humans or LLMs lock in working code, boost diversity to escape local optima, or roll back to prior versions when evolution stalls.

Overall framework. Finally, we discuss open challenges related to the overall ADRS approach.

Hyperparameter tuning. Balancing exploration and exploitation in LLMs remains difficult and often relies on trial and error. Future work should focus on automating this tuning process. A meta-learning layer, for instance, could allow the system to learn and adjust as evolution progresses, making the entire framework more reliable and accessible.

Human-ADRS interaction. The optimal balance between synchronous (interactive assistants like Cursor) and asynchronous (autonomous frameworks like OpenEvolve) user interface remains an open question. A key challenge remains for ADRS is to define when human guidance adds values versus when ADRS should act autonomously.

8 HOW CAN ADRS IMPACT THE RESEARCH PROCESS?

Despite its limitations (see Section 7.1), ADRS can already help researchers in two key ways:

- *Accelerating discovery:* ADRS tools automate tedious tasks such as implementation and, to some extent, debugging. This frees researchers to focus on problem selection, system architecture and design. Even imperfect solutions offer value by revealing new directions. In our telemetry repair case study, AI-generated insights shaped a better human-designed algorithm.
- *Achieve better-than-human results:* ADRS tools can explore the solution space more thoroughly than humans. Where a human might stop after a breakthrough, AI can continue testing variations for incremental gains. These small improvements compound. In our EPLB case study, OpenEvolve produced an algorithm that surpassed the state-of-the-art by exploiting missed optimizations.

As such, ADRS has the potential to reshape systems research as we know it.

One way to think about ADRS is as providing researchers with a virtually unbounded number of "assistants" that follow directions reliably. Much like junior researchers, ADRS frameworks are most effective when given clear problem specifications and well-defined goals.

As a result, ADRS adds another layer to an already established research hierarchy. In academia, for example, a faculty member typically advises Ph.D. students or postdocs, who in turn mentor undergraduate students. Each of these roles can now leverage ADRS tools to accelerate their work and broaden the scope of what they can accomplish.

In this context, two natural questions arise:

How would the role of a researcher change? As ADRS tools begin to solve an increasing number of problems autonomously, researchers will have more time to focus on higher-leverage activities—selecting which problems to pursue and formulating them precisely. If successful, ADRS has the potential to elevate everyone in the research hierarchy—faculty and students alike—and make them far more productive.

Will ADRS lead to fewer researchers? We believe the opposite is true. If anything, ADRS will expand the research community by enabling individuals who may not be expert problem solvers to contribute meaningfully. The result will be a faster rate of progress—solving more problems, faster and better. And, as history has shown, the supply of open research problems is virtually unbounded. For example, in the context of AI systems alone, workloads are becoming more complex (e.g., test-time compute, online reinforcement learning), hardware is growing more heterogeneous (e.g., specialized accelerators, new networking technologies), and performance and scalability demands are higher than ever (e.g., multi-modal training and inference).

9 CONCLUSION

As AI systems take over algorithm discovery, the role of the human researcher will change. Much like an academic advisor guides a student, the researcher of the future will act as a guide for these AI systems. The researchers' responsibilities will continue to be defining the problem, steering the research process, and critically evaluating the results.

One of the most profound implication is the potential for a *virtuous cycle*. We can use an ADRS to improve itself. As recent work has shown, models can learn to refine their own reasoning, debug their code, and discover more effective strategies. As these AI agents rapidly improve themselves, they will compound the pace of scientific discovery.

To summarize, in this paper, we have illustrated the potential of ADRS in systems research. Our case studies show that the ADRS approach can already outperform human baselines on key performance problems. While still very early, we call on the systems community to embrace these tools, not just as accelerators of research, but as subjects of research. Improving ADRS–making them efficient, scalable, and reliable–is itself a systems challenge. As such, we believe that system builders are uniquely positioned to shape the future of AI-driven discovery.

ACKNOWLEDGMENTS

REFERENCES

Suryanarayan Menon A., Sanjay J Prakash, Vinayak Naveen, Roshan Aji Cherian, Ron Regi Zacharia, Suryapriya S., and Josna Vr. A survey on routing algorithms and techniques used to improve network performance in software-defined networking. In *2023 2nd International Conference on Computational Systems and Communication (ICCSC)*, pp. 1–6, 2023.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

DeepSeek AI. Expert parallelism load balancer (eplb). `https://github.com/deepseek-ai/eplb`, 2024.

Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data (SIGMOD '17)*, pp. 1009–1024, Chicago, IL, USA, 2017. ACM. doi: 10.1145/3035918.3064029.

Anthropic. Claude code: Agentic code assistant. `https://www.anthropic.com/`, 2025. Accessed: 2025-09-30.

A. Boukerche, B. Turgut, N. Aydin, M.Z. Ahmad, L. Bölöni, and D. Turgut. Routing protocols in ad hoc networks: a survey. *Computer Networks*, 55(13):3032–3080, September 2011.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. Taobench: An end-to-end benchmark for social network workloads. *Proc. VLDB Endow.*, 15(9):1965–1977, may 2022. ISSN 2150-8097.

Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. Towards optimal transaction scheduling. *Proceedings of the VLDB Endowment*, 17(11):2694–2707, 2024.

Jae-Won Chung, Nishil Talati, and Mosharaf Chowdhury. Toward cross-layer energy optimizations in ai systems, 2024. arXiv:2404.06675v2.

Aditya Desai, Kumar Krishna Agrawal, Shuo Yang, Alejandro Cuadron, Luis Gaspar Schroeder, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. vattention: Verified sparse attention, 2025. URL https://arxiv.org/abs/2510.05688.

Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. 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 (SIGCOMM '20)*, pp. 557–570, Virtual Event, USA, 2020. ACM. doi: 10.1145/3387514.3405887.

Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. Man-made heuristics are dead. long live code generators! In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets '25)*, 2025.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

Chuanchao Gao, Niraj Kumar, and Arvind Easwaran. Energy-efficient real-time job mapping and resource management in mobile-edge computing. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pp. 15–28. IEEE, December 2024. doi: 10.1109/rtss62706.2024.00012. URL http://dx.doi.org/10.1109/RTSS62706.2024.00012.

GitHub. Github copilot. https://github.com/features/copilot, 2021. Accessed: 2025-09-30.

Anna Goldie and Azalia Mirhoseini. How alphachip transformed computer chip design. https://deepmind.google/discover/blog/how-alphachip-transformed-computer-chip-design/, September 2024. DeepMind Blog.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023.

Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pp. 507–517, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee. ISBN 9781450341431. doi: 10.1145/2872427.2883037. URL https://doi.org/10.1145/2872427.2883037.

Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. Autocomp: Llm-driven code optimization for tensor accelerators, 2025. URL https://arxiv.org/abs/2505.18574.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Cursor Inc. Cursor: Ai coding assistant. https://www.cursor.com/, 2024. Accessed: 2025-09-30.

Nathan Jay, Noga H Rotman, P Godfrey, Michael Schapira, and Aviv Tamar. Internet congestion control via deep reinforcement learning. *arXiv preprint arXiv:1810.03259*, 2018.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

Kamil Khan, Sudeep Pasricha, and Ryan Gary Kim. A survey of resource management for processing-in-memory and near-memory processing architectures. *Journal of Low Power Electronics and Applications*, 10(4):30, September 2020. doi: 10.3390/jlpea10040030. URL https://doi.org/10.3390/jlpea10040030.

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures, 2018. URL `https://arxiv.org/abs/1712.01208`.

Alexander Krentsel, Rishabh Iyer, Isaac Keslassy, Sylvia Ratnasamy, Anees Shaikh, and Rob Shakir. The case for validating inputs in software-defined wans. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pp. 246–254, 2024.

Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

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

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

Chieh-Jan Mike Liang, Haoran Qiu, Francis Y. Yan, Tianyin Xu, and Lidong Zhou. The next horizon of system intelligence. `https://www.microsoft.com/en-us/research/blog/the-next-horizon-of-system-intelligence/`, September 2025. Accessed: 2025-09-29.

Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification, 2019. URL `https://arxiv.org/abs/1902.10319`.

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

Shu Liu, Asim Biswal, Amog Kamsetty, Audrey Cheng, Luis Gaspar Schroeder, Liana Patel, Shiyi Cao, Xiangxi Mo, Ion Stoica, Joseph E Gonzalez, et al. Optimizing llm queries in relational data analytics workloads. *arXiv preprint arXiv:2403.05821*, 2024b.

Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, et al. Supporting our ai overlords: Redesigning data systems to be agent-first. *arXiv preprint arXiv:2509.00997*, 2025.

Phillip Long, Zachary Novack, Taylor Berg-Kirkpatrick, and Julian McAuley. Pdmx: A large-scale public domain musicxml dataset for symbolic music processing, 2024. URL `https://arxiv.org/abs/2409.10831`.

Ruiying Ma, Chieh-Jan Mike Liang, Yanjie Gao, and Francis Y. Yan. Algorithm generation via creative ideation, 2025. URL `https://arxiv.org/abs/2510.03851`.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the 2019 ACM SIGCOMM Conference*, pp. 270–288, Beijing, China, 2019. ACM. doi: 10.1145/3341302.3342080. URL `https://doi.org/10.1145/3341302.3342080`.

Ziming Mao, Tian Xia, Zhanghao Wu, Wei-Lin Chiang, Tyler Griggs, Romil Bhardwaj, Zongheng Yang, Scott Shenker, and Ion Stoica. Skyserve: Serving ai models across regions and clouds with spot instances. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 159–175, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3717459. URL `https://doi.org/10.1145/3689031.3717459`.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342644. URL `http://dx.doi.org/10.14778/3342263.3342644`.

Julian McAuley, Jure Leskovec, and Dan Jurafsky. Learning attitudes and attributes from multi-aspect reviews, 2012. URL `https://arxiv.org/abs/1210.3926`.

Wes McKinney. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference (SciPy 2010)*, pp. 51–56. SciPy, 2010.

Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, volume 70, pp. 2430–2439, Sydney, Australia, 2017. PMLR. doi: 10.5555/3305890.3305932. URL `http://proceedings.mlr.press/v70/mirhoseini17a.html`.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015. Also published/extended in later venues.

Ansh Nadga, Abhradeep Thakurta, and Prabhakar Raghavan. Ai as a research partner: Advancing theoretical computer science with alphaevolve. `https://research.google/blog/ai-as-a-research-partner-advancing-theoretical-computer-science-with-alphaevolve/`, September 30 2025. Google DeepMind Research Blog.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, et al. Mlgym: A new framework and benchmark for advancing ai research agents. *arXiv preprint arXiv:2502.14499*, 2025.

Harald Ng, Seif Haridi, and Paris Carbone. Omni-paxos: Breaking the barriers of partial connectivity. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pp. 314–330, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871.

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

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pp. 305–320, Philadelphia, PA, 2014. USENIX Association.

OpenAI. Openai codex. `https://openai.com/index/introducing-codex/`, 2025. Accessed: 2025-09-30.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. Version v2, 12 Feb 2024.

Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the ACL*, 2005.

Ori Press, Brandon Amos, Haoyu Zhao, Yikai Wu, Samuel K Ainsworth, Dominik Krupke, Patrick Kidger, Touqir Sajed, Bartolomeo Stellato, Jisun Park, et al. Algotune: Can language models speed up general-purpose numerical programs? *arXiv preprint arXiv:2507.15887*, 2025.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024. URL `https://arxiv.org/abs/2307.07924`.

26

Krzysztof Rusek, Jose Suarez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Routenet: Leveraging graph neural networks for network modeling and optimization in sdn. *IEEE Journal on Selected Areas in Communications*, 38(10):2260–2270, October 2020. ISSN 1558-0008. doi: 10.1109/jsac.2020.3000405. URL http://dx.doi.org/10.1109/JSAC.2020.3000405.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/codelion/openevolve.

Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, pp. 15–29, Portland, OR, USA, 2020. ACM. doi: 10.1145/3318464.3380584. URL https://doi.org/10.1145/3318464.3380584.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. Code researcher: Deep research agent for large systems code and commit history. *arXiv preprint arXiv:2506.11060*, 2025.

Shewaye Sirika and Smita Mahajan. Survey on dynamic routing protocols. *International Journal of Engineering Research & Technology (IJERT)*, 5(1), January 2016. doi: 10.17577/IJERTV5IS010028. Paper ID: IJERTV5IS010028.

Reiko Tanese. *Distributed Genetic Algorithms for Function Optimization*. PhD thesis, University of Michigan, 1989. Ph.D. thesis.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. Ai-researcher: Autonomous scientific innovation. *arXiv preprint arXiv:2505.18705*, 2025.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. doi: 10.1038/s41586-023-06747-5.

Kent Walker. A summer of security: empowering cyber defenders with ai, July 2025. URL https://blog.google/technology/safety-security/cybersecurity-updates-summer-2025/. Accessed: 2025-10-02.

Sarah Wooders, Shu Liu, Paras Jain, Xiangxi Mo, Joseph E Gonzalez, Vincent Liu, and Ion Stoica. Cloudcast:{High-Throughput},{Cost-Aware} overlay multicast in the cloud. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 281–296, 2024.

Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: optimizing spot instance savings under deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 185–203, 2024.

Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, September 2020. ISSN 2150-8097. doi: 10.14778/3421424.3421432. URL https://doi.org/10.14778/3421424.3421432.

Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD/PODS '22, pp. 931–944, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3517885. URL https://doi.org/10.1145/3514221.3517885.

Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. Prism: Unleashing gpu sharing for cost-efficient multi-llm serving. *arXiv preprint arXiv:2505.04021*, 2025.

Lidong Zhou. A match made in silicon: The co-evolution of systems and ai. In *NeurIPS 2024 Invited Talks*, 2024. URL `https://neurips.cc/virtual/2024/invited-talk/101132`. Accessed: 2025-10-07.

A   ABLATION STUDIES

We have conducted a few ablation studies on the Can't Be Late case study and provide the results as follows.

A.1   GEPA RESULTS

In addition to OpenEvolve, we apply GEPA to evolve the program generation for the Can't Be Late use case. We build a custom adapter that wraps two stages of evaluation, similar to OpenEvolve: stage 1 validates syntax and simulator compliance, while stage 2 runs full simulations on our test workloads to compute cost savings. GEPA uses this score as the fitness value and attaches diagnostic feedback for failed runs. Feedback is then converted into a reflective dataset which guides a model (we use OpenAI o3 in this setup) to rewrite the candidate program. We initialize the process with the greedy baseline, and set our evaluation metric as the average cost savings across single-region traces.

We give the greedy program as the base program, and use Claude Opus to refine the base program entirely. The feedback function given to GEPA is the score evaluated on the downstream single-region traces. Our final result achieved 4% improvement over the greedy baseline at iteration 68, capping the search at 200 iterations.

A.2   CHANGING TRAIN SET COVERAGE

(a) Training with 3% of the available training set.   (b) Training with the full training set.
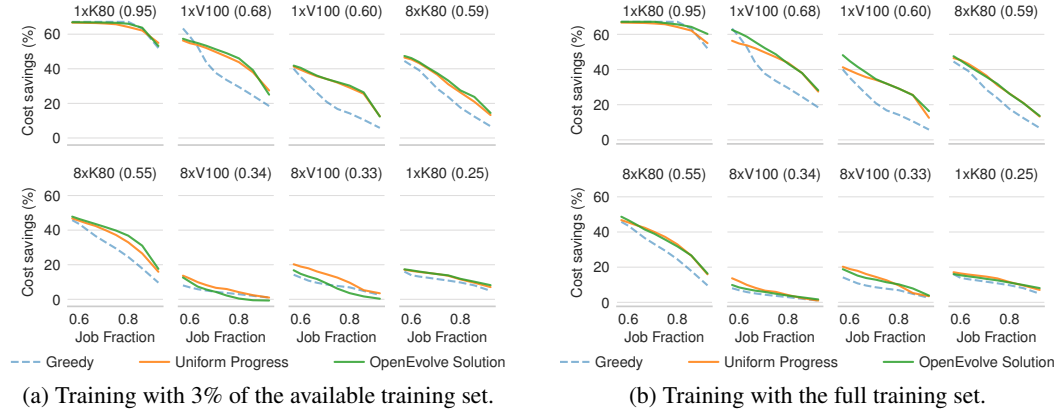
Figure 8: Impact of training set coverage on policy evolution. We split data into 30% training and 70% testing. The left panel shows results when using only 3% of the training data, while the right panel uses the entire training set.

29

B   Failure Taxonomy

ADRSes demonstrate exciting capabilities in advancing systems research in our case study; however, they currently face challenges that require careful setup. To mitigate these challenges, we first characterize the common failure modes ADRSes exhibit. In the next section (see Section 6), we outline best practices and guidelines to address these limitations.

As shown in Table 3, we group failures into three broad categories—execution errors, search (evolution) failures, and algorithm failures, and report their frequency in our evaluation. Detailed descriptions of each failure type are provided in Section 6.

Execution Errors. Among one third of the failures come from immediate execution failures. The most common cases are *syntax and interface errors*, where the generated solution code fails to compile or connect with the evaluator due to missing imports, type mismatches, or misusing required interfaces. Other fails due to *budget exhaustion*, where a candidate consumes excessive memory, runs into timeouts, or exceeds model API quotas.

Search Failures. Roughly half of the failiures happen when the solution executes but search fails to make progress. *Premature convergence* happens when the search fo solution settles on a local optimum too early, such as sticking with a steiner tree strategy in multi-region transfer while ignoring opportunities to make use of data partitions. *Stuck-in-the-loop* refers to generating near-duplicate solutions without improvement, like iterations that only rename variables or add logging. *Mutation drift* occurs when edits are contradictory or random. For example, oscillating between BFS tree and graph-based method in the multi-region network transfer task, slowing or preventing convergence.

Algorithm Failures. These are failures where candidate solutions run but fail to advance algorithms. Common patterns include *misaligned objectives*, where the solution ignore constraints (e.g., throughput gains that break latency SLOs). *Suboptimal optimizations* also appear frequently as the evolved solution makes shallow tweaks in APIs instead of coming up with algorithmic innovation. *Overfitting* hard-codes behavior to evaluation traces (e.g., spot scheduling policies collapsing on unseen spot patterns). *Reward hacking* exploits loopholes in the evaluator. For example, in our multi-agent runs, some candidates bypass a required stage in algorithm, causing the evaluator to assign higher scores.

C   Case Studies

We now present detailed evolution results from the case studies detailed in Table 1 as follows.

C.1   Adaptive weight compression

This task studies adaptive weight quantization with variable per-column bitrate. The objective is to assign high vs. low bitrates across columns to minimize overall storage cost (measured in average bits per element across all elements in the weight tensor, lower is better) while preserving accuracy (measured by Wikitext-2 perplexity, or PPL, where lower indicates higher accuracy). The base program is a hand-written heuristic that applies dynamic bitrate encoding, where each column of the weight matrix is quantized at a bitrate chosen according to its importance score. An initial hand-tuned calculation and mapping from importance score to column bitrate provided reasonable compression performance of 2.64 bits/elem and PPL of 22.9 on a Llama3.2-1B, but left room for improvement.

To guide evolution, the evaluator combines normalized scores from Wikitext PPL (0.5), PTB PPL (0.2), and bitrate (0.3), with solutions exceeding 2.5 bits/elem and PPl of 30 receiving zero score, to hit a compression ratio under 2.5 with the highest accuracy. We use Gemini-2.5-pro to generate candidates, running 200 iterations in about 12 hours with less than 20 USD. The evolution takes more than 10 hours due to the evaluator time, i.e., each evaluation needs to fully quantize the entire Llama3.2-1B model before computing perplexity and bitrate. The evolved program achieves 2.49 bits/elem with PPL of 22.84, hitting a lower bitrate compared to the best sweep configuration (2.5 bits/elem) while maintaining accuracy.

The improved solution from the evolution simply tunes hyperparameter values of the hand-crafted mapping between importance scores and bitrates, without any algorithmic improvements on column

importance finding or mapping strategy. As such, it offers only marginal gains over a parameter sweep and remains more of an automated hyperparameter tuner than a novel quantization strategy. Future work could explore structural changes to better prompting (e.g., explicitly listing the methods and parameters to explore) or evolutionary settings (e.g., exploration ratio) to move beyond incremental improvements.

C.2 Network Telemetry Repair

This task identified by HotNets'24 Krentsel et al. (2024) studies repair of faulty router telemetry signals, which can become buggy due to router faults or collection infrastructure errors. Such inconsistencies (for example, counters on the two ends of a link not matching) can cause the network controllers to make incorrect decisions. The objective is to detect and repair faulty telemetry to produce a self-consistent view of network state.

We use OpenEvolve to evolve repair strategies, running 300 iterations (about 8 hours) with a mix of reasoning (GPT-o3, 80%) and non-reasoning (GPT-4o, 20%) models, supplemented by contextual hints from the HotNets'24 paper. The evolved program introduces structured repair logic, including averaging nearby counters to reduce noise, and separating repair and confidence estimation into distinct steps. It achieves a repair score of 95% and confidence calibration of 95%, outperforming the HotNets'24 solution (86% repair, 65% confidence).

C.3 Cloudcast

The Cloudcast problem, published in NSDI '24 Wooders et al. (2024), studies cost-aware multicast across multi-region and multi-cloud topologies. The objective is to construct an overlay connecting a source, multiple destinations, and optional waypoints so as to minimize egress cost.

Our initial program is a direct replication strategy: the source sends data independently to each destination. While simple, this approach often incurs high egress costs when destinations are located in distant or expensive regions. To guide evolution, the evaluator tests candidate algorithms across 10 multi-region, multi-cloud configurations and assigns a total score based on the overall egress cost of the scheduled paths. We use OpenEvolve with a mix of o3 and Gemini-2.5-pro models, running 100 iterations in about one hour, costing less than $10 for the entire run.

The evolved solution successfully rediscovers the Steiner tree strategy as the state-of-the-art, achieving an average cost reduction of 31.1% compared to direct replication. This solution constructs a cost-efficient multicast tree by introducing intermediate waypoints. For example, data may be replicated once at a cheaper waypoint region and then forwarded to multiple destinations, reducing the total egress cost.

C.4 Global model placement

The research problem Yu et al. (2025) focus on the challenge of multi-LLM serving on shared GPUs under bursty, heterogeneous workloads. The key metric is the KV pressure ratio (KVPR), defined as the SLO-weighted request rate divided by available KV cache memory. The optimization goal is to minimize the maximum KVPR across GPUs, thereby reducing contention and improving SLO compliance.

The base program is the algorithm from the paper Yu et al. (2025): models are placed sequentially onto the first GPU with sufficient remaining memory, without considering long-term balance. While feasible, this approach often overloads some GPUs while leaving others underutilized. We use OpenEvolve to evolve improved placement strategies, guided by a scoring function that combines execution correctness with the KVPR objective. The evolution runs with GPT-4o (70%) and GPT-o3 (30%) as the model ensemble, converging in about 70 iterations (roughly 40 minutes). The best evolved program achieves an 18.5% higher score compared to the state-of-the-art reported in the original paper.

The evolved strategy mirrors the SOTA algorithm from the paper and assignment used in PRISM, but crucially adds a local improvement stage. After the initial placement, it repeatedly tests whether moving or swapping models between GPUs reduces the maximum KVPR, applies such refinements, and stops once it finds a move that can reduce the current maximum KVPR.

C.5 Sparse attention design

The research problem Desai et al. (2025) studies efficient sparse attention mechanisms that approximate dense attention while balancing accuracy and compute efficiency. Sparse attention reduces computation by restricting each query to attend to only a subset of keys, but selecting this subset (i.e., the active indices) introduces a trade-off between density (fraction of active indices) and relative error (approximation quality). The objective is therefore to evolve sparse attention masks that minimize a combined loss of density and relative error, formulated as Score $=$ $-(\text{Density} + \text{Relative Error})$.

C.6 Multi-Agent System Optimization

The research problem extends the work of Cemri et al. (2025) (NeurIPS'25), which studies diagnosing and repairing failures in multi-agent LLM systems (MAS), and proposes MAST as the taxonomy of MAS failure modes. Such systems (e.g., MetaGPT Hong et al. (2024), ChatDev Qian et al. (2024)) often suffer from breakdowns in coordination, memory, or communication that reduce task success. The challenge is how to improve multi-agent systems, evolving more robust agent architectures, prompts, and inter-agent communication patterns to increase reliability and performance.

The base program is a direct adaptation of MetaGPT Hong et al. (2024), assembled into a minimal Python implementation (roughly 400 LOC). This initial version defines fixed agent roles, communication protocols, and system prompts, but frequently encounters the failure modes identified in the MAST. To guide evolution, the evaluator uses the MAST annotator, which assigns a score of $1/(1 + \text{total FM occurrence})$, penalizing common coordination and memory failures.

We ran three evolution configurations with different mutation scopes: (1) modifying agent definitions and communication schemes, (2) introducing new verification and communication flows using GPT-5, and (3) allowing changes to the number and types of agents, i.e., the system topology. The evolved solutions introduced innovations such as improved context management (v1) and verification/communication flows (v2), though removing verification in v3 degraded performance. Overall, downstream program development success improved from 40% in the base program to 47% (v1) and 53% (v2) on the ProgramDev-v1 benchmark, before dropping to 30% in v3. The fast that verification agent was removed in v3 was an example of reward hacking (since we penalize the verification failures, the evolution algorithm got rid of the whole verification when it could) and an example of the importance of carefully tuning which parts of the initial code the evolution algorithm is allowed to change. These results show that OpenEvolve can automatically discover MAS design refinements that improve robustness, though careful control over mutation scope is critical to avoid reward hacking.

C.7 GPU Cache Algorithm Optimization

This task studies optimizing cache algorithms for GPU execution. While CPU versions of the algorithm run correctly, directly adapting them to GPUs often introduces significant overhead and poor performance.

The base program was an initial GPU implementation generated by GPT-o3. This version was extremely slow and failed some correctness tests, making it unsuitable for practical use. To guide evolution, the evaluator balanced two objectives: correctness (passing functional tests) and runtime speed.

We ran 120 iterations of OpenEvolve using Gemini-2.5-flash, with a total runtime of about 40 minutes and cost of roughly $20. The evolved program uncovered practical optimizations such as wrapping key kernels with `torch.jit`, which noticeably improved execution speed. However, despite these gains, the final code remained slower than desired and continued to fail certain edge-case tests.