VULCAN: Instance-Specialized, Verifiable Systems Heuristics Through LLM-Driven Search

Rohit Dwivedula
The University of Texas at Austin
Austin, TX, USA

Divyanshu Saxena
The University of Texas at Austin
Austin, TX, USA

Sujay Yadalam
The University of Texas at Austin
Austin, TX, USA

Eric Hayden Campbell
The University of Texas at Austin
Austin, TX, USA

Daehyeok Kim
The University of Texas at Austin
Austin, TX, USA

Aditya Akella
The University of Texas at Austin
Austin, TX, USA

Abstract

Systems resource management tasks rely primarily on hand-designed heuristics. However, growing hardware heterogeneity and workload diversity require heuristics specialized to particular deployment instances, making manual design expensive and difficult to scale. In this paper, we explore how to synthesize systems heuristics using LLMs. The main challenge is ensuring that generated heuristics execute safely, integrate correctly with the surrounding system, and still achieve strong performance. We propose VULCAN, a framework that identifies LLM-friendly interfaces that isolate core decision logic from the rest of the implementation. With VULCAN, LLM-generated code is restricted to simple stateless decision functions, while trusted runtime abstractions provide rich derived statistics for meaningful policy exploration without system-integration bugs. To ensure execution safety, LLMs synthesize heuristics in a restricted language, ANVIL, that guarantees important properties by construction. We evaluate VULCAN across three well-studied domains and demonstrate up to 4.9× higher savings for spot-VM scheduling, up to 2× lower miss ratios for cache eviction, and up to 14% higher application performance for tiered-memory systems, while ensuring execution safety throughout.

CCS Concepts: • **Information systems** → **Information storage systems**; • **Software and its engineering** → **Operating systems**; • **Computing methodologies** → *Randomized search.*

Keywords: resource management heuristics, LLM-aided design, cache eviction, memory tiering, cloud scheduling

ACM Reference Format:
Rohit Dwivedula, Divyanshu Saxena, Sujay Yadalam, Eric Hayden Campbell, Daehyeok Kim, and Aditya Akella. 2027. VULCAN: Instance-Specialized, Verifiable Systems Heuristics Through LLM-Driven Search. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3842654.3848582

EuroSys '27, Rabat, Morocco

ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848582

1 Introduction

Systems resource management has long relied on manually designed heuristics. This approach was sustainable when hardware was slower to evolve, deployment environments were more uniform, and a single carefully engineered policy could remain competitive across many settings. Today, systems operate across rapidly evolving hardware platforms, increasingly heterogeneous memory and compute hierarchies, and workloads that vary across tenants, applications, and time [56, 85, 93, 94, 107, 113]. As a result, the relevant unit of optimization is often an *instance*: a particular combination of hardware, workload mix, deployment setting, and optimization objective. The challenge is that designing and maintaining heuristics for each instance does not scale.

Neural models offered an early route to instance specialization [2, 18, 22, 23, 34, 58]. A learned policy can, in principle, capture complex dependencies in system state and adapt to a target environment better than a fixed hand-written rule. In practice, however, neural approaches remain difficult to deploy in core systems paths: their behavior is opaque, their training and serving pipelines add operational complexity, and integrating them with existing systems code is cumbersome [27, 83]. More broadly, systems heuristics do not exist in isolation. They read state through existing interfaces, maintain auxiliary state over time, and rely on surrounding mechanisms to enact their decisions. Any automated approach must therefore satisfy a substantially higher bar than predictive quality alone: it must integrate correctly with the surrounding system and be safe to execute.

Recent advances in large language models suggest a different route. Unlike neural policies that must be embedded as external predictors, LLMs can synthesize *code*, enabling instance-specialized heuristics to be expressed as executable logic compiled directly into the host implementation. While enticing, this presents difficult challenges. If the LLM has unrestricted control over heuristic generation, it must reason

not only about policy quality but also about state management, memory safety, API semantics, synchronization, and mechanism interaction (§2.2). Our experience, and that of recent work [20, 25], suggests that current LLMs can often discover performant heuristics under such freedom, but do not reliably produce code that satisfies the safety and integration requirements needed for deployment (§2.2.1). Conversely, constraining the LLM too narrowly preserves safety but sharply limits the policy space it can explore (§2.2.2).

This paper asks: *how should resource-management code be organized so that instance-specialized heuristic synthesis is practical?* Our answer is to refactor heuristics around a *synthesis boundary* that isolates core decision logic from the rest of the implementation. The key insight is that a small set of interfaces and runtime abstractions can make such synthesis both expressive and safe inside real systems. Concretely, we show that the core decision logic of a broad class of systems heuristics can be expressed through two interfaces, Rank and Value, and that constructing rich statistics for performant policies can be delegated to reusable feature-store abstractions. This organization permits the synthesis of verified policy code while keeping stateful logic and system-facing interaction in trusted, developer-managed code.

We realize this design in Vulcan, a framework for synthesizing instance-specialized heuristics as safe executable code. Vulcan rests on two key ideas. First, it decouples policy logic from state management. Developers retain responsibility for high-level task decomposition, policy triggers, raw feature collection, and the mechanisms that enact decisions; LLM-synthesized artifacts are restricted to *stateless decision functions* behind Rank and Value interfaces (§4.2). To enable meaningful policy exploration, Vulcan exposes rich derived statistics through `libVulcan`, a library of *listeners*: modular blocks that allow synthesized policies to use moving averages, percentiles, and other aggregates without implementing the data structures themselves (§4.3). Second, Vulcan synthesizes verified decision logic through Anvil, a restricted domain-specific language (DSL) for LLM-generated code (§5). By excluding heap allocation, pointers, recursion, and unbounded loops, Anvil makes important execution safety properties such as memory safety, leak freedom, and termination *provable by construction* for any well-formed program. Notably, the abstractions introduced by Vulcan allow Anvil to remain restrictive while supporting powerful heuristics.

This design occupies a different point in the design space from both prior LLM-based heuristic synthesis and prior ML-for-systems work. Rather than asking the LLM to generate an entire heuristic implementation, Vulcan narrows synthesis to a structured policy layer that remains expressive enough for meaningful heuristic discovery. Rather than embedding an opaque learned model, Vulcan produces explicit policy code that integrates with existing implementations through narrow interfaces and trusted runtime support. We evaluate Vulcan across multiple resource-management settings: deadline-aware spot-VM scheduling, cache eviction, and memory tiering. Our findings show that Vulcan can synthesize both general-purpose and (when required) instance-specialized heuristics that are competitive with, and in several cases outperform, strong human-designed baselines, while satisfying execution-safety properties by construction.

In summary, we make the following contributions:

- We identify a refactoring of systems heuristics around a synthesis boundary and design a framework, Vulcan, that exposes only core decision logic to LLM-driven search while retaining system-facing structure in trusted code.
- We present `libVulcan`, which includes Rank/Value interfaces for decision logic and listener abstractions, which together provide a common substrate for expressive specialized systems heuristics.
- We introduce Anvil, a restricted policy language that yields execution-safety properties by construction, enabling practical deployment of LLM-generated heuristics.
- We evaluate Vulcan for three well-studied domains: for spot VM scheduling, we find a *general* heuristic that outperforms all baselines to yield up to 4.9× savings; for the well-studied cache eviction problem, we find *specialized* heuristics that yield up to a 2× reduction in miss ratios compared to baselines for specific instances; for tiered-memory systems, we find page promotion heuristics that yield up to a 14% improvement in application performance; all while satisfying important execution safety properties.

2 The Pursuit of Instance Specialization

Heuristics have been hand-crafted for various systems management tasks because optimal actions are often intractable to compute online. Action spaces for these tasks are large, decision intervals are fine-grained, and important system variables are often latent or partially observable. Consequently, practitioners design heuristics for specific *instances* of a problem, *i.e.,* particular combinations of workload, hardware, operating conditions, and optimization objectives such as performance, fairness, or utilization. Seen this way, systems research is often a search for *instance-specialized* heuristics rather than universally optimal ones.

2.1 An Example of Specialized Heuristics: Caching

We illustrate this search for specialized heuristics using the example of cache eviction policies. Different heuristics have been proposed for specific workloads, objectives, or deployment scenarios [12, 42, 60, 80, 92, 96, 108, 111]: some [60, 111] perform well for large cache sizes, while others [12, 42] are more suited for smaller caches. Workload characteristics also matter: scan-heavy workloads (mostly new objects) and churn-heavy workloads (mostly repeated objects) require different algorithms [80]. Heuristics have further been

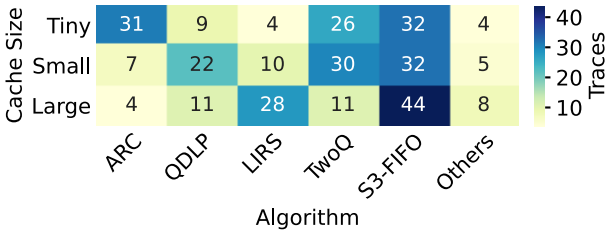


Figure 1. Count of CloudPhysics traces where each heuristic performs best (max. object hit rate). Tiny, small, and large imply cache sizes of 0.1%, 1%, and 10% of trace footprint.

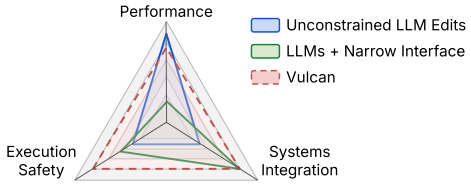


Figure 2. Illustration of three design alternatives. Unconstrained LLM edits result in execution safety and systems integration concerns (§2.2.1); overly constrained synthesis fails to achieve good performance (§2.2.2). Vulcan attempts to satisfy all three.

tailored for end-to-end *objectives* such as tail latency [14] and fairness [49], and for system-level *constraints* such as CPU overhead [92], lock-free design [108], and memory efficiency [26].

To quantify this observation, we executed 17 caching algorithms using libCacheSim [43] on 106 block I/O traces from the CloudPhysics dataset [97], where each trace represents a distinct instance (corresponding to a different tenant). Figure 1 shows our findings: no single algorithm performs best on even half the instances, and the set of competitive algorithms shifts significantly with cache size, *e.g.,* ARC is better for tiny caches while LIRS is better for large caches.

These findings are not limited to cache eviction: policies across the systems stack exhibit the same instance dependence. Congestion control algorithms are tailored for Internet [1, 2, 8, 17] versus datacenter workloads [4, 48, 64], kernel queueing disciplines are tuned for varying performance and fairness objectives [29, 76, 86, 89], and cluster schedulers are fine-tuned for specific application classes such as data processing jobs [30, 31], cloud VM workloads [55], or deep learning training jobs [66, 102].

However, as workloads grow increasingly diverse, performance objectives become multi-dimensional, and hardware heterogeneity evolves rapidly, manually specializing heuristics for each instance is no longer tractable. This raises a natural question: *can heuristic specialization be automated?*

2.2 LLM-Based Synthesis: Requirements and Pitfalls

Recent advances in LLMs and coding agents, particularly in generating executable code [88, 98, 112], refining it using natural-language feedback [105], and carrying out multi-step reasoning [99], make automated heuristic specialization increasingly practical. Indeed, several recent systems [20, 25, 33, 44, 68] have already demonstrated that LLM-based approaches can discover effective resource-management heuristics for specific problems. While these advances are promising, building usable heuristics for practical systems imposes three key requirements.

- **Performance:** They must meet or exceed the performance of existing manually designed heuristics.
- **Execution safety:** They must be *proven* to be free from failures such as crashes, memory errors, leaks, or non-termination. Automated synthesis at scale makes the conventional practice of manual review impractical, so safety verification must be automated as well.
- **System integration:** Heuristics rarely run in isolation: they interact with other system components to read state and enact their decisions. Thus, the generated code must respect the APIs and semantics of the surrounding system.

Naively applying LLMs may not satisfy all three properties; below, we explore the design space to identify an approach that does. Figure 2 summarizes the alternatives, which we discuss in detail below.

2.2.1 Unconstrained Search Space Approaches.

At one extreme of the design space are existing proposals for LLM-driven heuristic synthesis [20, 25, 33, 68]. These approaches place few restrictions on what the LLM generates: starting from decision logic, the LLM agent is free to allocate and manage state and interact with system APIs as it sees fit. This *unconstrained search space* maximizes the LLM's opportunity to discover high-performing heuristics, but also exposes the full implementation surface to errors.

We illustrate this using cache eviction as a running example. We compare two workflows: (i) an *evolutionary loop* [25, 81] that iteratively proposes, evaluates, and refines candidates; and (ii) an *agentic* workflow with access to codebase inspection and other tools. Both approaches used Claude Opus 4.5 [7] to synthesize heuristics for libCacheSim [43]. We evaluate the generated heuristics on ten traces from the CloudPhysics dataset [97]. Full details are provided in Appendix A; here, we discuss the key outcomes.

Evolutionary loop. We ran the search for fifty iterations and found that several synthesized heuristics had **system-integration violations**: several synthesized heuristics (including the best-performing one) under-reported per-object metadata size, claiming to use only 16 bytes while actually allocating up to 40 bytes. libCacheSim uses reported metadata size to check whether the cache has space for a new object; under-reporting causes more objects to be admitted than the

cache capacity allows, artificially inflating the hit rate. We manually fixed these bugs and found that the resulting best heuristic achieved a 1.2%–13.66% improvement over FIFO, which is comparable to several manual baselines.

Agentic workflow. We allowed a Claude Code agent [6] to make edits to the codebase until it reached a fixed cost budget, and repeated this ten times independently. Across these ten heuristics, four modified internal libCacheSim counters that are read-only – another **system-integration violation**. One heuristic also introduced a *double-free bug*, an **execution safety violation**; incidentally, this specific bug was not triggered during testing, demonstrating that simple compile-and-test pipelines alone are insufficient. The improvements for the best heuristic (out of ten) were 0.2–23% over FIFO.

These results confirm that LLMs *can* discover performant heuristics, but do not reliably produce code that satisfies safety and integration requirements. Related work reports similar failures: ADRS [20] noted use-after-erase and null dereference bugs, consistent with broader evidence that LLMs are less reliable on tasks requiring reasoning over complex codebases [40, 45, 71, 115].

2.2.2 Overly Restricted Search Space Approaches.

The problems described in §2.2 arise because the LLM has unrestricted access to the implementation surface. The other extreme is to restrict LLM edits to narrow refinements of an existing heuristic [36], such that system integration and execution safety are preserved by construction. This includes tasks such as adjusting threshold values, adding conditionals, or reweighting existing features. However, shrinking the search space so aggressively forfeits the ability to discover substantially better policies.

To illustrate this, we revisit the evolutionary loop experiment for cache eviction but with a much narrower interface. Inspired by HALP [92], we select eight eviction candidates using a *base heuristic* and then restrict the LLM-written function to choosing the final candidate from within that set. We provide the LLM with each candidate's access count, size, last-access timestamp, and insertion timestamp, plus percentile distributions of these statistics over all cached objects. We experiment with both LRU and FIFO as base heuristics, running evolutionary search for 50 iterations each on the same traces as §2.2. We refer to the resulting best heuristics as Evolved-LRU and Evolved-FIFO, respectively.

Under this interface, the LLM cannot interact directly with libCacheSim, eliminating system-integration violations and substantially narrowing the space of execution-safety failures. Indeed, none of the synthesized heuristics exhibited execution-safety violations. However, the same restriction limits the search space too much to produce meaningful gains: the best Evolved-LRU heuristic achieves a miss-ratio reduction (MRR) over FIFO by 0.05%–7.76%, nearly identical to LRU's 0.04%–7.53%. Similarly, Evolved-FIFO improves over FIFO by only 0.01%–0.19%. These gains are far smaller than those in §2.2.

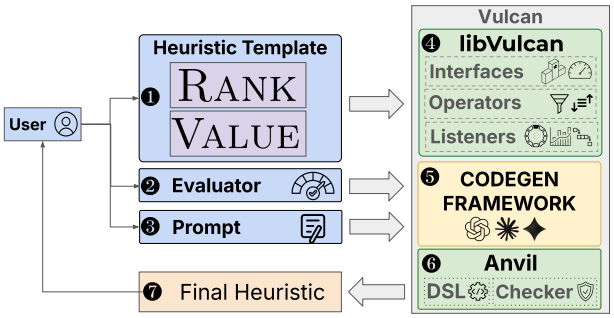


Figure 3. Overview of Vulcan. Users provide task-specific inputs – a template, a prompt, and an evaluator. Vulcan uses a code-generation framework to produce a safe and performant heuristic.

2.3 Towards a Better Division of Labor

While LLM-synthesized heuristics hold promise for instance specialization, approaches at both extremes of the design space fail to meet all three requirements (Figure 2), due to a poor division of labor between the LLM and the developer. In the unconstrained setting (§2.2), the developer delegates everything to the LLM: decision logic, state management, and system interaction alike; in the over-restricted setting (§2.2.2), the developer retains so much control that the LLM cannot meaningfully explore. With Vulcan, we identify abstractions that enable a better division of labor: supporting meaningful policy exploration while preserving execution safety and system integration by construction.

3 Vulcan Overview

We present Vulcan, a framework for synthesizing safe, performant, and integration-compliant heuristics for systems resource-management tasks (see Figure 3).

3.1 Key Ideas

Vulcan rests on two key ideas that together address the three requirements from §2.2.

Idea 1: Abstractions to decouple core decision logic and state management (§4). To explore meaningful heuristics, LLMs must derive useful statistics from raw features and use them to produce a decision. Vulcan decouples these two steps and restricts LLM edits to: (i) generating code for stateless decision interfaces, and (ii) selecting and querying an API for rich derived features. We show that two stateless interfaces, Rank (ranking candidates) and Value (computing a scalar from system state), suffice to express the core decision logic of a broad class of resource-management heuristics (§4.2). A library of *listeners* (§4.3) exposes useful statistics from raw features while handling the underlying state management. All other aspects of system integration remain the responsibility of developers.

Idea 2: Safety by construction via a restricted policy language (§5). As shown in §2.2, compile-and-evaluate pipelines alone are insufficient to ensure safety. Vulcan addresses this through Anvil, a restricted DSL in which LLMs express decision logic. Anvil omits heap allocation, pointer arithmetic, recursion, and unbounded loops; as a result, memory safety, leak freedom, and termination are guaranteed by construction without the need for human review.

3.2 Workflow: Using Vulcan to Create Heuristics

Developers provide three inputs: a *heuristic template* (❶), an *evaluator* (❷), and a natural-language *prompt* (❸) describing the resource-management task, the available raw features, and the evaluation methodology.

The heuristic template (based on Idea 1) exposes the core decision logic through Rank or Value interfaces and declares all raw features of interest. Developers define this template using `libVulcan` (❹), Vulcan's runtime library. The template and prompt are passed to a *code-generation framework* (❺), which synthesizes a heuristic in Anvil (❻) (based on Idea 2) by iteratively generating candidate heuristics, evaluating them on the user-provided evaluator, and using the feedback to improve. LLMs may also attach listeners to raw features, deriving statistics from simple means to arbitrary percentiles. The synthesized heuristic is passed to the evaluator, which scores it and returns feedback to the synthesis loop.

Once the search terminates, Vulcan returns the best-scoring candidate as the final heuristic (❼). Vulcan is agnostic to the code-generation approach; for our case studies (§7), we use two approaches (ShinkaEvolve [51], OpenEvolve [87]).

4 `libVulcan`: Interfaces for Policy Exploration

Vulcan narrows LLM synthesis to the *core decision logic* of a heuristic, delegating state management and system interaction to developer-owned code (§4.1). This division leads to two concrete abstractions: stateless decision interfaces (§4.2) and listener-based feature stores (§4.3).

4.1 Division of Labor in Vulcan

To motivate the division of labor in Vulcan, we examine the typical components of systems heuristics, illustrated in Figure 4 using a memory-tiering system.

A high-level task is typically *decomposed* into a collection of sub-policies; memory tiering, for example, uses separate policies for allocation [78], placement [59, 78], and migration [103] (Figure 4). Within each policy, there are five typical components. The *trigger* determines when the policy runs, *e.g.*, page faults [78] or periodic scans [103]. Then, relevant statistics are *collected*: for the page migration policy in Figure 4, these include page accesses and DRAM bandwidth. Since processing raw signals at decision time can impose

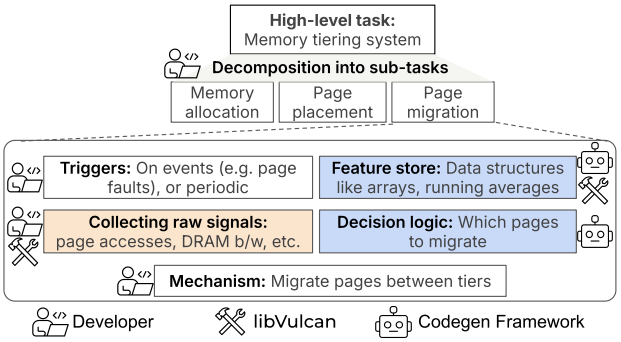


Figure 4. Typical components of systems heuristics (using memory tiering as an example). Vulcan impacts the colored parts; icons show the division of responsibility among developers, `libVulcan`, and LLMs.

high overhead, a third component *stores* them in data structures that support efficient computation of temporal or statistical aggregates; for instance, a windowed average of DRAM bandwidth rather than per-timestamp measurements. The *decision logic* consumes these derived statistics to produce a policy output. Finally, *mechanisms* enact the decision, *e.g.*, swapping pages between memory tiers.

With Vulcan, decomposition, triggers, and mechanisms (uncolored in Figure 4) remain the developer's responsibility. These components require deep knowledge of system semantics: in the memory-tiering example, decomposition depends on how allocators and page fault handlers interact; adding new triggers may require changes to kernel page fault handling; and implementing migration requires correctly managing page metadata, locks, and reference counts. As §2.2 showed, even state-of-the-art LLMs do not reliably handle this complexity.

Vulcan automates the remaining components: feature storage and decision logic. Developers expose raw signals through a heuristic template (❶ in Figure 3), leaving open how those signals are aggregated and used in the decision. The code-generation framework (❺) fills in this logic using `libVulcan`'s APIs (❹) for rich derived statistics.

Insight motivating `libVulcan`. `libVulcan` serves as the interface between developers, synthesized heuristics, and the system. A key observation is that many safety and integration bugs arise when LLMs are responsible for managing low-level feature state: allocating buffers, maintaining data structures, and updating them correctly across events. `libVulcan` eliminates this exposure by providing *stateless interfaces* for decision logic (§4.2) while encapsulating all state management in a *listener library* (§4.3).

4.2 Value and Rank Interfaces

The core decision logic of most resource-management heuristics takes one of two forms: it either *ranks* a set of candidates and selects a subset (*e.g.*, evicting an object from cache), or it

Table 1. Examples of systems resource management tasks and the type of task (VALUE or RANK) they fall into.

Policy	Description	Type
Congestion control [32]	Decide the number of unacknowledged bytes in flight (cwnd).	VALUE: compute the cwnd.
DVFS control [73]	Decide cpu_freq to balance performance & power.	VALUE: compute the frequency.
Cluster autoscaling [47]	Decide n_replicas to provision for a given service.	VALUE: compute replicas per service.
Hardware prefetching [63]	Choose an offset from the current access to prefetch data.	VALUE: compute the offset value.
Cache eviction [91]	Select which cached object(s) to evict.	RANK: all cached objects.
CPU scheduling [70]	Select which thread to schedule next.	RANK: all runnable threads.
Promotion in tiered memory [78, 103]	Select which pages to promote to a higher tier.	RANK: all memory pages.

computes a scalar *value* that drives a resource-management action (*e.g.,* computing a cwnd in congestion control). Table 1 shows that this pattern recurs across a broad class of systems tasks, and libVulcan exposes two corresponding interfaces, RANK and VALUE, for LLM-generated decision logic. When neither interface fits, it can be extended with a small number of additional functions.

Interface: VALUE. A VALUE heuristic, $\text{Value}(X) \rightarrow Y$, is a function where X encodes system state as a set of features and Y is a scalar output corresponding to some control variable. Table 1 lists examples of typical VALUE tasks and their outputs. For instance, a congestion controller maps features such as rtt and loss_rate to a window size. To use the VALUE interface, developers define the function signature `double value_fn (vulcan::store&)`, where store contains the state X and the return value is Y.

Interface: RANK. A RANK heuristic, $\text{Rank}(X, O) \rightarrow \mathcal{R}$, is a function where $O = \{o_i \mapsto f_i\}$ maps N candidate objects to their features and $\mathcal{R} = [o_{i_1}, o_{i_2}, \dots]$ is an ordered list of all objects. The mechanism then selects from this ordered list as needed. Cache eviction, for example, ranks all objects in cache by their utility for retention (Table 1) and evicts the lowest-ranked object. As part of the template for RANK tasks, developers define the function signature for a per-object scoring function: `double score_fn (vulcan::store&, int64_t obj_id)`. libVulcan provides utility functions to sample candidates and sort them by this score_fn in ascending or descending order as needed (§4.5).

4.3 Configurable Feature Stores Using *Listeners*

Stateless decision functions cannot compute derived features such as moving averages, percentiles, or population-level statistics; yet smoothed averages [15, 16, 92, 103], summary statistics [17, 32], and histograms [12] are essential to strong heuristics. Delegating this state management to LLMs reintroduces the safety bugs from §2.2.

VULCAN resolves this tension through *listeners*: modular blocks that the LLM attaches to raw signals to create rich derived features. Once attached, their query APIs can be called inside the decision logic, whether a value function (VALUE) or a scoring function (RANK). Table 2 summarizes the listeners provided by libVulcan. Each listener tracks updates to its signal and maintains internal state for efficient queries. For example, an Average listener maintains running sum and count state and returns their ratio on query. libVulcan also supports more complex aggregates: temporal aggregates (*e.g.,*RollingPercentile) and population-level aggregates over all candidate objects (*e.g.,*PopMean). Temporal aggregates support both interfaces; population-level aggregates primarily support RANK tasks.

Listeners need only be reviewed and verified for execution safety once, and a compact set of listeners can be reused across several systems tasks, as demonstrated by our case studies (§7). Listeners are also composable: multiple listeners can be attached to the same feature, and different features can use different listeners, forming a task-specific feature store without any LLM-written state management.

4.4 An Example of Using libVulcan

Listing 1 demonstrates how the cache eviction task is set up using libVulcan. Lines 1–25 form the *template*, which is one of the inputs to VULCAN. This template is written by the user of VULCAN (*i.e.,* the developer); LLMs in VULCAN then search for instance-specialized heuristics within the confines of this template by deciding which listeners to add (lines 27–29) and defining the decision-making function (lines 31–35).

To create this template, the developer decides which raw signals the policy should use (lines 2–4), sets up instrumentation to collect these signals (within on_insert and on_access), and then formulates the problem using one or more of VULCAN's interfaces (RANK or VALUE). In Listing 1, cache eviction is formulated as a RANK problem (lines 19–23), in which the lowest-ranked object is evicted; the developer can also choose between multiple implementations of RANK provided by libVulcan (line 21, §4.5).

4.5 libVulcan Implementation

libVulcan implements the RANK and VALUE interfaces as well as the listener APIs as a C++ library. C/C++ systems can integrate libVulcan directly; Python-based systems use the same APIs via pybind11 [41] bindings. Our case studies exercise both integration paths: spot VM scheduling (§7.1) uses Python bindings, while cache eviction (§7.2) and memory tiering (§7.3) use the C++ library.

libVulcan provides multiple implementations of RANK that the developer can choose between when setting up

Table 2. Listeners provided by `libVulcan`. The table shows the state each listener maintains, with update/query/space complexities. The first six rows show listeners for temporal aggregates, while the bottom two show listeners for aggregates across the object population.

Listener	Query API(s)	State and data structures	Complexity		
			Update	Query	Space
`Average()`	`get_avg`	`sum`, `count`	$O(1)$	$O(1)$	$O(1)$
`MinMax()`	`get_min`, `get_max`	`min`, `max`	$O(1)$	$O(1)$	$O(1)$
`RollingWindow(N)`	`get_kth`, `get_avg`	ring buffer, sum	$O(1)$	$O(1)$	$O(N)$
`EWMA(`$\{\alpha_0, \alpha_1, \ldots \alpha_M\}$`)`	`get_ewma(`α`)`	current EWMA value for each α	$O(1)$	$O(1)$	$O(1)$
`RollingPercentile(N)`	`get_percentile`	ring buffer, ordered multiset [5]	$O(\log N)$	$O(\log N)$	$O(N)$
`RollingCount(N)`	`get_count`, `contains`	ring buffer, counts map	$O(1)$	$O(1)$	$O(N)$
`PopPercentile()`	`get_pop_percentile`	per-object latest, ordered multiset [5]	$O(\log M)$	$O(\log M)$	$O(M)$
`PopMean()`	`get_pop_mean`	per-object latest, sum	$O(1)$	$O(1)$	$O(M)$

Listing 1. *Template* for cache eviction in `libVulcan` (lines 1–25). LLMs search for policies within this template; lines 27–35 (red) show a sample candidate policy synthesized during the search.

```
1  void init():
2    vulcan::store store = {
3      .obj_signals: [{"insert_time",int}, {"access_time",int}]
4    };
5    setup_listeners(store);
6
7  // LLM evolved functions
8  void setup_listeners(vulcan::store& fs);
9  double score_fn(vulcan::store& fs, int64 id);
10
11 void on_insert(ObjId o):
12   store.add_obj(o);
13   store.update("insert_time", o, now());
14
15 void on_access(ObjId o):
16   store.update("access_time", o.id, now());
17
18 ObjId evict_obj():
19   R = vulcan::Rank(
20     store.all_objs, score_fn,
21     vulcan::OnDemandSort, n_sample=32
22   );
23   victim = R.back();
24   store.remove_obj(victim);
25   return victim;
26
27 void setup_listeners(vulcan::store& fs):
28   fs.add_listener("access_time",vulcan::Latest());
29   fs.add_listener("insert_time",vulcan::PopPercentile());
30
31 double score_fn(vulcan::store& fs, int64 id):
32   auto recency = fs.get_latest("access_time", id)-now()
33   if fs.get_pop_percentile("insert_time", id) > 0.9:
34       return 6.0 * recency
35   else return 1.0 * recency
```

the template: `OnDemandSort` (used in §7.3) scores and ranks candidates at decision time, while `IncrementalSort` (used in §7.2) precomputes ranks by maintaining candidates in a priority queue that is updated as features change, trading decision-time latency for incremental bookkeeping. These implementations also expose configurable parameters; *e.g.,* `OnDemandSort` can approximate ranks via a random sample of candidates rather than all of them (Listing 1, line 21).

Table 3. Execution-safety properties guaranteed by Anvil.

ID	Property
P1	**Memory safety.** No out-of-bounds access, use-after-free, or double-frees.
P2	**Leak freedom.** No unbounded memory growth over time.
P3	**Termination.** Must not run indefinitely; LLM scoring functions are guaranteed to eventually terminate.

`libVulcan` totals ~3.5K lines of C++ code: ~1,300 for the core Rank and Value interfaces, ~1,000 for listeners, and ~250 for Python bindings.

5 Anvil: DSL for LLM-Written Code

The abstractions in §4 reduce the scope for execution-safety bugs but do not eliminate them. Vulcan closes this gap with Anvil: a domain-specific language in which every well-formed program satisfies the execution-safety properties in Table 3 by construction.

Programming model. Anvil is a restricted imperative language based on C++: it supports scalars, arithmetic, conditionals, bounded `for` loops, and built-in math functions (*e.g.,* `anvil::max`), while excluding pointers, arrays, containers, heap allocation, recursion, and unbounded loops. These exclusions map directly to the execution safety properties (Table 3): excluding pointers and arrays prevents unsafe memory accesses (P1); excluding heap allocation and persistent state prevents memory leaks and state-management errors (P2); excluding recursion and unbounded loops prevents non-termination (P3).

State management and trust model. Most useful systems heuristics rely on persistent state, such as histories or derived statistics, but pure Anvil does not contain the constructs needed to maintain such state. To let Anvil programs use stateful functionality implemented outside the language, Anvil supports `extern` declarations for C/C++ functions: the Anvil compiler sees the `extern`'s function signature (name, scalar arguments, and scalar return type),

while its implementation resides in trusted external code. In Vulcan, listener query APIs (Table 2) are exposed to synthesized heuristics through such `extern`s. This defines Anvil's trust model: a well-formed Anvil program satisfies P1–P3 provided every linked `extern` also satisfies P1–P3.

Compiling and executing Anvil code. In Vulcan, code generators write heuristics in Anvil and use `libVulcan` functions to attach and query listeners (declared in Anvil as function prototypes). The LLM-written Anvil code is then compiled in two stages. First, the frontend parses the generated heuristic against the Anvil grammar and rejects disallowed constructs. Next, the Anvil code is lowered to C and linked against C implementations of `libVulcan` functions (for attaching/querying listeners) that were simply declared as `extern`s in the Anvil code. The final C implementation is then compiled with native toolchains (e.g., `gcc`). This lowering of Anvil code into C is made possible because of Anvil's C-like syntax: scalar expressions, conditionals, and bounded loops pass through unchanged, while Anvil built-ins (*e.g.,* `anvil::max`) resolve to equivalent C library calls. Thus, executing Anvil code adds no interpreter or JIT overhead and makes integration with the rest of the codebase seamless. The only overhead is the compilation step that lowers Anvil code to C, which takes under one second per scoring function.

Implementation. Anvil is implemented in OCaml using Menhir [75] for parser generation, totaling approximately 3.5K lines of code: ∼1,000 for the parser and lexer, ∼1,500 for name resolution and class desugaring, and ∼1,000 for the AST and C code generation.

6 Using Vulcan in Practice

Defining the target instance. A key requirement of Vulcan-generated heuristics is high performance, which often requires instance specialization (§2) across workloads, hardware configurations, operating regimes, or performance objectives. When instances differ enough that no single heuristic performs well across them (Figure 1), policies should be specialized to a particular instance, such as a specific cluster, tenant, workload class, or arrival pattern (as we do for cache eviction in §7.2). In contrast, for understudied problems where good solutions are not yet well understood, even a single heuristic targeting average-case performance can yield substantial gains, as in our spot VM scheduling case study (§7.1).

Cost–fidelity tradeoff in evaluators. Vulcan invokes the evaluator hundreds of times during search, so evaluation cost bounds the amount of exploration possible. Evaluating heuristics on real systems can take hours per candidate; in practice, we use cheaper proxies during the search phase: trace-driven simulation (§7.1, §7.2) or execution over a short representative workload (§7.3).

Natural-language prompt. As discussed in §3, users must also provide a natural-language prompt. Besides describing the task at hand, this prompt must also include information about `libVulcan` APIs and may include evaluator details, workload or instance information, and instructions for heuristic design.

LLM API budgets. In our experiments, we use Claude Sonnet 4.5 and Claude Opus 4.5 (80%/20% split) on AWS Bedrock to synthesize heuristics for Vulcan and the baseline [20, 87].

7 Case Studies and Results

We use Vulcan to synthesize heuristics for three resource-management tasks:

- **Spot VM scheduling** (§7.1): decide when a deadline-driven cloud job should use cheaper spot instances to minimize completion cost.
- **Web cache eviction policies** (§7.2): choose which cached objects to evict to maximize object hit rate (*e.g.,* in CDNs).
- **Page promotion policy in tiered-memory systems** (§7.3): choose which pages to promote from slower memory tiers to faster tiers to maximize application-specific performance metrics.

The primary question we seek to answer through our evaluation is whether Vulcan can synthesize heuristics that outperform strong, human-written baselines across these diverse tasks. Our evaluation measures: (i) how Vulcan compares to unconstrained LLM-based synthesis, (ii) how Vulcan's components (interfaces, listeners, and the DSL) affect synthesized heuristics, and (iii) the cost of heuristic synthesis (*e.g.,* API charges and elapsed time) relative to the resulting benefits. In subsequent sections, we will use these case studies and results to discuss the developer effort required to use Vulcan (§8.1) and the scope of its decision interfaces, including how they can be composed or extended to support additional tasks (§8.2).

7.1 Spot VM Scheduling

Cloud providers offer two pricing tiers: on-demand instances are reliably available but expensive, while spot instances are up to 11× cheaper [101] but can be preempted at any time. Deadline-sensitive jobs therefore typically avoid spot instances and pay the on-demand premium for predictability. Recent work [101] proposed a scheduler that cuts this cost by opportunistically switching between spot and on-demand while still meeting deadlines. We use Vulcan to synthesize scheduling heuristics for the single-region (§7.1.1) and multi-region formulations (§7.1.2) of this problem and compare the resulting heuristics against unconstrained evolution [20].

7.1.1 Single-region scheduling. In the single-region formulation [101], the scheduler is initialized with per-job constants: the required VM type, execution time, deadline, and a *changeover delay* cost incurred when the job is switched

Table 4. Summary of use cases, baselines, evaluation metrics, and evaluators.

Use case	Baselines	Metric	Evaluator
Spot VM scheduling (single)	Greedy, Uniform Progress [101], ADRS [20]	Avg. USD saved	Simulator, 5% sample
Spot VM scheduling (multi)	Round-robin uniform progress, ADRS [20]	Avg. USD saved	Simulator, 5% sample
Cache eviction	Nine state-of-the-art algorithms	Object miss-ratio reduction (MRR)	Simulator, 1M requests
Memory tiering	ARMS [103]	Goodput, latency	Emulator, per-workload

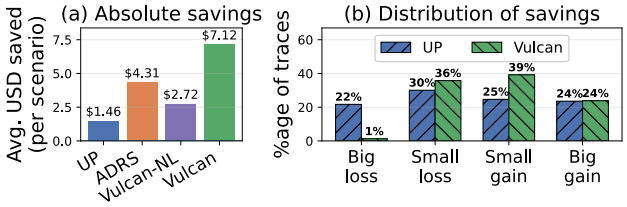


Figure 5. Comparison of schedulers for the single-region spot VM setting. (a) Mean savings; higher is better. (b) Big losses/gains exceed $10; small losses/gains are below $10.

between on-demand and spot instances. At each tick, the scheduler observes the current job status, progress so far, remaining work, and spot availability, then chooses one of three actions: run on spot, run on on-demand, or wait.

Template. Since the per-tick decision in this problem is a single control variable encoding one of three actions, it naturally fits the Value interface, which maps observed state to a scalar output. While Value supports arbitrary scalar outputs, we restrict its output domain here to three values corresponding to the actions above. The per-tick features and per-job constants are passed as inputs to the Value function, and the LLM may attach listeners to the per-tick features.

Manually designed heuristics. We use two baselines from Wu et al. [101]. The first, a *greedy heuristic*, uses spot VMs opportunistically, waits when they are unavailable, and switches to on-demand only when further waiting risks missing the deadline. The second is *Uniform Progress* (UP), the state-of-the-art heuristic for this problem. Uniform Progress keeps the job on a linear progress trajectory from start to finish: it exploits spot VMs when they are available, but if needed, switches back to on-demand to catch up with the linear pace.

Evaluator. We use the simulator and traces from Wu et al. [101]: spot-availability data for four AWS instance types (1- and 8-GPU K80/V100) paired with job configurations to yield 21,600 evaluation cases. During synthesis, candidates are scored by running a fixed 5% sample of all evaluation cases and computing the *average USD savings* over the greedy baseline; all reported results use the full 21,600-case set.

Results. We run OpenEvolve [87] for 100 iterations and compare the resulting Vulcan scheduler against two baselines: (i) UP, the manual baseline above, and (ii) ADRS [20], which runs OpenEvolve with the same budget but allows unconstrained edits to the scheduler policy.

Listing 2. Regime detection in Vulcan's scheduler.

```
double alpha = fs.get_avg(has_spot);
// longest consecutive spot availability streak (L_max)
int64_t L_max = 0, L_curr = 0;
for (int i = 0; i < 18; ++i):
  if (fs.get_kth(has_spot, i) == 1):
    L_curr++; L_max = max(L_max, L_curr);
  else L_curr = 0;
// regime detection: scarce when alpha low or small streaks
bool scarce = (alpha < 0.35)||(L_max<od_ticks && alpha<0.5);
```

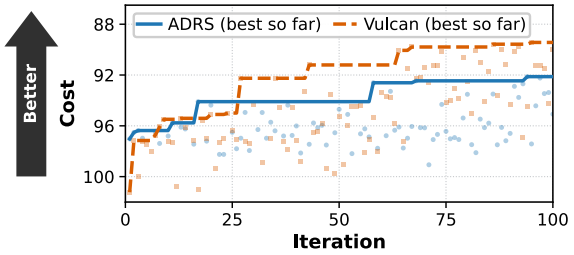


Figure 6. Vulcan vs. unconstrained search (ADRS). Lines: best-performing-heuristic so far; dots: individual candidates.

Main result. Figure 5a shows the mean savings of each scheduler over the greedy baseline. Our synthesized scheduler achieves nearly 4.9× the savings of UP and 1.7× those of ADRS.

How Vulcan outperforms UP. The per-trace distribution (Figure 5b) reveals how Vulcan outperforms UP: UP performs worse than the greedy heuristic (*i.e.,* costs more money) on 52% of the evaluation cases. This pattern stems from UP's core design assumption: it treats spot availability as *worst-case unreliable*. UP switches to on-demand whenever the job falls behind its expected progress schedule, *regardless of spot availability history*. In contrast, the Vulcan heuristic (Listing 2) attaches `RollingWindow` and `Average` listeners to track recent spot availability (`has_spot`), classifying each tick into a *scarce* or *abundant* regime and adapting accordingly: committing to on-demand sooner when spot looks unreliable, but staying patient and returning opportunistically when it does not.

Unconstrained search. Figure 6 compares Vulcan to ADRS under an equal budget. Despite limiting the LLM to a Value function written in Anvil (§5), Vulcan loses no performance to unconstrained search—in fact, it reaches better heuristics faster. A possible explanation is the smaller, structured

Table 5. Cost of heuristic synthesis with Vulcan.

Use case	Tokens (millions)		API cost	Time
	Input	Output		
Spot VM (single)	2.2	0.20	$11	2.5 h
Spot VM (multi)	1.0	0.12	$5	1.2 h
Cache eviction†	0.7–2.0	0.15–0.50	$5–15	2.4–4.4 h
Memory tiering†	1.6–2.2	0.23–0.36	$10–14	1.9–3.0 h

† Minimum–maximum range across instances.

search space: while ADRS must explore the entire heuristic implementation, Vulcan explores only the decision logic, enabling targeted policy changes.

Impact of removing listeners. Without listeners (Vulcan-NL), the heuristic is limited to the latest value of each feature. Vulcan-NL still outperforms UP, but full Vulcan achieves roughly 2.6× the savings of Vulcan-NL. Without listeners, the scheduler cannot use recent spot-availability history to distinguish scarce and abundant regimes, preventing the adaptive behavior used by the full Vulcan heuristic.

Cost of synthesis. Synthesizing the heuristic (Listing 2) with Vulcan incurred $11 in LLM costs and took 2.5 hours (Table 5). Across all 21,600 evaluation scenarios (*i.e.,* simulated jobs), the resulting scheduler saves an average of $7.12 per job relative to the greedy baseline, or $2.81 more than the best ADRS solution. At this average improvement over ADRS, using the synthesized heuristic for just four jobs would recover the LLM synthesis cost on average.

7.1.2 Multi-Region Scheduling.

In the multi-region formulation, the scheduler is allowed to choose which region to run the job in, with each region having different pricing and spot availability. At each tick, in addition to choosing to schedule a job or wait, the scheduler can also choose to switch to a different region. Since the scheduler can observe spot availability only for the *current region*, a good heuristic would have to balance exploration and exploitation when choosing regions.

Template. Unlike the single-region case study, this task does not directly fit the semantics of either Rank or Value: the scheduler output at each tick is a 2-tuple of (action, region). We therefore decompose decision-making into two parts. First, a Value function, invoked once per scheduler tick, chooses the action: use spot instances in the current region, use on-demand instances in the current region, migrate to another region, or wait. Second, if a migration action is selected, a Rank function selects which region to migrate to. Although this region-choice *could* also be expressed as a Value task (with a fixed output domain of region identifiers), destination selection naturally involves *comparing* regions based on their respective observation histories, making Rank's semantics a more natural fit.

Evaluator. We use the multi-region simulator and benchmark from ADRS [20], with 600 evaluation cases with AWS spot availability and prices for V100 GPU instances across nine U.S. regions/zones. During evolution, the evaluator runs each candidate heuristic on a fixed 5% sampled subset of all evaluation cases and returns the average cost per trace; after synthesis, the best policy is evaluated on the full benchmark.

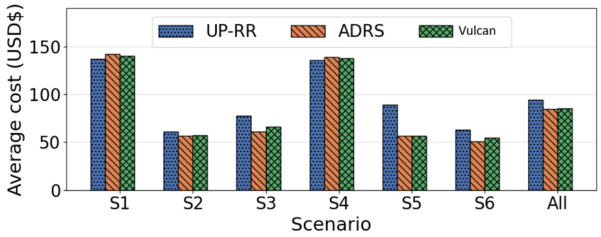


Figure 7. Average workload cost per scenario (lower is better).

Table 6. Raw signals for cache-eviction scoring.

Feature type	Attributes
Per-object (f_i)	Access count, last access timestamp, insertion timestamp, size (bytes)
Global (X)	Current time, recently evicted object IDs (ghost list).

Results. We run OpenEvolve for 100 iterations and compare the result against: (i) UP-RR [20], a round-robin multi-region extension of UP, and (ii) an unconstrained-search baseline (ADRS). We report results for six scenarios [20] covering different combinations of availability zones and geographic regions, each with 100 evaluation cases. As Figure 7 shows, Vulcan outperforms UP-RR in four of six scenarios, demonstrating how performant heuristics can be found even within constrained interfaces that guarantee execution safety. As in the single-region case study, synthesizing this multi-region heuristic required only 1.2 hours on a single node and $5 in LLM API costs (Table 5).

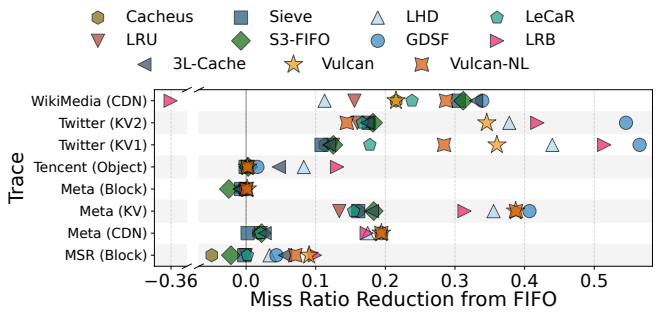
Over all six scenarios, Vulcan achieves only ∼7% smaller savings than unconstrained search, demonstrating that its constrained interfaces can express performant heuristics while guaranteeing execution safety.

7.2 Cache Eviction

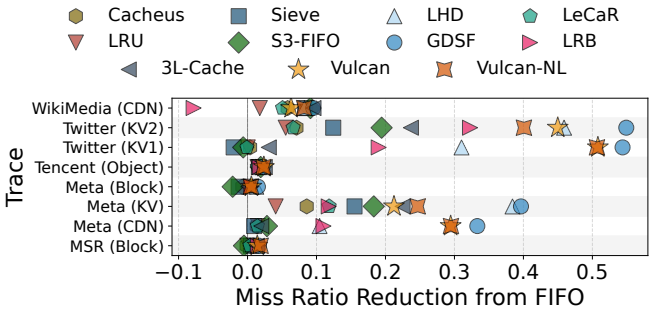
Cache eviction is central to performance in CDNs [9, 14, 38, 69, 91, 109] and in-memory stores [61, 79]. As §2.1 showed, access patterns and object characteristics vary substantially across deployments, making instance-specialized eviction heuristics valuable. Evicting an object from a pool of candidates maps naturally to a Rank task (§4.2): objects are scored by cache utility and the lowest-ranked object is evicted when space is needed. We evaluate whether Vulcan can synthesize such heuristics for individual cache deployments.

Template. We expose a Rank scoring function over a priority queue of cached objects; per-object metadata (age, size, last access time, recent eviction history; see Table 6)[1] is

[1]Note that the caching template shown in Listing 1 was simplified for exposition and contains only two raw signals.



(a) Large cache size (10%), objects of different sizes



(b) Small cache size (0.1%), objects of different sizes

Figure 8. Performance of instance-specialized heuristics versus baselines for two classes of instances. Additional results appear in Appendix B.

stored in a `libVulcan` feature store available to the function. To reduce overhead, we use a lazy update strategy: scores are recomputed only when an object is accessed, not on every cache event.

Evaluator. We use libCacheSim [43] to simulate eight deployment-specific traces from Microsoft [65], Meta [13], Twitter [107], Tencent [113], and Wikimedia [100]. For each trace, we consider two cache sizes (10% and 0.1% of the trace footprint) and two object-size settings (uniform and variable), yielding 32 instances total. During synthesis, candidates are scored on the first 1M requests (with a timeout of 100 s to encourage efficient heuristics); the final heuristic is evaluated over the full trace, measuring miss-ratio reduction (MRR) over FIFO (§2).

Results. Our nine state-of-the-art eviction baselines are GDSF [21], S3-FIFO [108], SIEVE [111], LHD [12], Cacheus [80], LRB [91], 3L-Cache [114], LeCaR [96], and LRU. We use ShinkaEvolve [51] as the code-generation framework for generating heuristics for each of the 32 instances described above. We also run an ablation where we disable listeners (named Vulcan-NL).

Overall results. Across the 32 instances, Vulcan-generated heuristics outperform *all* baselines in two instances and rank among the top three heuristics for 13 instances. Furthermore, Vulcan heuristics are within 1% of the best baseline on six instances and within 5% on an additional six instances. A single implementation achieves all these results automatically after a one-time development effort.

Breakdown of results. Figure 8a and Figure 8b show miss-ratio improvements over FIFO for sixteen instances. For the 10% cache configuration, Vulcan outperforms all baselines on the Meta block trace, matches the performance of the best (GDSF) on the Meta CDN trace, and comes within 1% of the best (LRB) on the MSR trace. On the MSR block trace, the synthesized heuristic (Listing 3) blends raw access count with an EWMA-smoothed count, using a ghost multiplier to emphasize recently evicted objects. For the Meta block trace, it instead measures per-object access intensity and applies an additive penalty for ghost-list objects rather than a multiplicative one. For the 0.1% cache configuration, Vulcan matches the performance of the best baseline (3L-Cache) on the Tencent trace and comes within 1% of the best (GDSF) on the MSR block trace. Vulcan achieves over 2× lower miss ratios than strong baselines such as SIEVE and S3-FIFO on specific instances: for instance, on the Twitter KV1 trace (0.1% cache size, variable object sizes) Vulcan's miss ratio is 35.4%, compared to 73.4% for SIEVE and 72.5% for S3-FIFO. On the Tencent object trace, Vulcan generates a GDSF-inspired heuristic similar to the MSR 10% case, fine-tuning the constants for frequency and size to suit the Tencent workload and smaller cache. Detailed results for the remaining instances are in §B.1 (Figure 12a and Figure 12b).

Listing 3. Simplified snippets of evolved heuristics for MSR and Meta block traces: actual heuristics are 29 and 50 lines of code.

```
// MSR block trace
freq     = 0.7*count + 0.3*EWMA(count, alpha=0.2)
size_pen = log2(size)
ghost_mult = 1 + 2*ghost_count
utility  = (last_access/1e6 + 10*freq - size_pen) *
     ghost_mult

// Meta block trace
intensity  = count / (last_access - insert_time)
freq       = log2(count) * (10*intensity)
size_fac   = 1 / log2(size)
ghost_bonus = 1000 * (1 + log2(ghost_count))
utility    = (0.6*last_access + 400*freq) * size_fac +
     ghost_bonus
```

Impact of listeners. Removing listeners only slightly degrades performance: Vulcan-NL still outperforms all the baselines on two instances but trails the full Vulcan by 0.84% in average MRR. This difference is small for this use case since eviction heuristics are invoked extremely frequently, making complex listeners prohibitively expensive; computationally heavy heuristics are likelier to time out during evolution, biasing the search towards simpler listeners.

Table 7. Computational cost of eviction heuristics. Average (and range) of execution time to run a set of traces (scenarios) in a single-threaded simulator. Higher = more expensive.

Algorithm	Execution time vs. FIFO
LRU	1.0× (0.5–2.1×)
SIEVE [111]	1.1× (0.6–2.0×)
LeCaR [96]	1.8× (0.8–3.0×)
S3-FIFO [108]	2.1× (1.4–3.8×)
GDSF [21]	3.9× (2.5–7.1×)
Cacheus [80]	4.1× (2.3–7.9×)
LHD [12]	4.1× (2.2–7.1×)
Vulcan	**13.4× (7.4–22.5×)**
3L-Cache [114]	14.4× (5.3–38.2×)
LRB [91]	361.2× (19.8–614.6×)

Cost of synthesis. Across all 32 instances, synthesis cost $272 for LLM calls (≈ 36.5M input, 8.6M output tokens) and consumed 85 CPU-hours to evaluate candidate solutions.

Runtime cost of Vulcan heuristics. Table 7 shows that Vulcan policies' runtime overhead is comparable to that of 3L-Cache [114], with which our policies are very competitive. Additionally, LRB [91], which outperforms Vulcan in six instances, takes 27× as long as Vulcan on average. Vulcan's cost can be reduced further by sampling: for instance, approximating the Vulcan eviction policy for Meta KV with variable object sizes (10% cache size) using $k = 5$ samples reduces the cost from 15× FIFO to 5× FIFO while largely preserving miss rates.

7.3 Memory Tiering

Memory tiering expands effective memory capacity by placing pages across fast local DRAM and slower tiers such as CXL-attached memory or NVM. The central policy problem is deciding *which pages* should occupy scarce DRAM: ideally, frequently accessed pages should be promoted to DRAM, while colder pages remain in slower tiers. We use Vulcan to synthesize page promotion heuristics and seek to answer whether they generalize from an emulated tiered-memory setup to real CXL hardware.

Manually designed heuristics. Existing memory tiering systems [53, 59, 78, 103] use sampled access information – from mechanisms such as Intel PEBS [39] or page-table accessed-bit scanning [72] – to estimate page hotness and migrate pages accordingly. Memtis [53] represents one class of designs: it promotes pages whose sampled access count exceeds a dynamically adjusted hotness threshold and demotes pages once their cooled access counts fall below it. ARMS [103] represents another class, using a ranking-based approach instead: it computes per-page scores from multi-timescale access averages, ranks pages by those scores, and migrates the highest-ranked pages to DRAM.

Template. We formulate the page migration problem as a Rank task in Vulcan, similar to ARMS [103]: the goal is to estimate each page's hotness and rank pages accordingly.

Listing 4. Memory tiering heuristic synthesized by Vulcan.

```
fs.add_listeners(accesses, {EWMA({0.95, 0.35})});
double score_fn(vulcan::store& fs, int64_t obj_id):
    double fast = fs.get_ewma(accesses, obj_id, 0.95);
    double slow = fs.get_ewma(accesses, obj_id, 0.35);
    double accel = (slow > 0.001) ? (fast / slow) : 1.0;
    return fast * (1.0 + 0.55 * min(accel, 2.5));
```

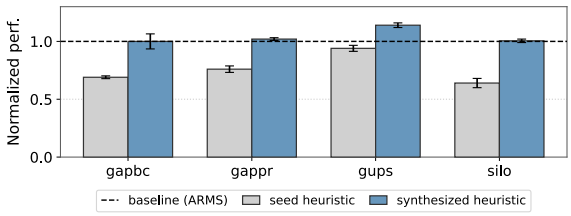


Figure 9. Performance of the Vulcan-synthesized heuristic compared with the no-op seed heuristic.

The raw signals provided for this task include per-page access counts over a time window (500 ms) and the observed bandwidths of the hot and cold tiers. The developer-written template invokes `libVulcan`'s Rank every 1 s.

Evaluator. We score candidate heuristics by running a single workload on a two-tier memory *emulator*. The emulator runs on a dual-socket Xeon CloudLab [24] `c220g5` node: the near tier is socket 0's DRAM, and the far tier is socket 1's DRAM with its uncore frequency – the clock governing the mesh interconnect, last-level cache, and memory controllers – clamped to the minimum ratio. This inflates remote-access latency to *emulate* that of CXL-attached memory, while avoiding the cost of running heuristic search directly on scarce CXL hardware. After synthesis, we analyze whether the resulting heuristic generalizes to other workloads by testing it on our CXL-based testbed.

Evolution setup. We seed the search with a "promote nothing" heuristic that scores every page as 0 (*i.e.,* no migrations happen). We then run OpenEvolve [87] for 100 iterations, where heuristic candidates are scored by their end-to-end goodput on a specific application; we then use the best heuristic in the generalization study.

Generalization to real CXL hardware and unseen workloads. We take this single heuristic synthesized against a single application on an emulator and, without further tuning, evaluate it on a real CXL machine across a diverse set of workloads. Our experimental setup pairs a dual-socket Xeon host with a Micron CZ120 CXL 2.0 memory expander (256 GiB), connected to socket 0 over PCIe Gen5 x8 and exposed as a CPU-less NUMA node. The test workloads span three domains: GUPS [74] (memory microbenchmark used for training), GapBS Betweenness Centrality and PageRank [11] (graph analytics), and Silo TPCC [95] (in-memory database). All results are reported normalized to ARMS [103].

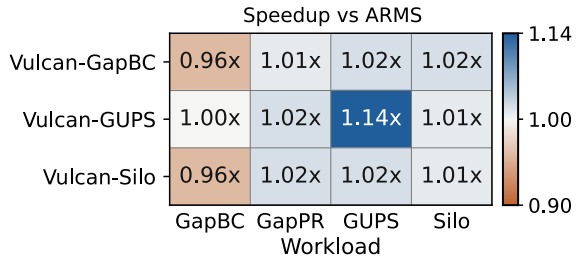


Figure 10. Generalization of Vulcan-synthesized heuristics.

Results. Listing 4 shows the synthesized heuristic, Vulcan-GUPS, which used the GUPS benchmark [74] as the evaluator. Figure 9 shows its performance on the CXL testbed. From Listing 4, we observe that the heuristic uses the ratio of two EWMAs to derive a metric called the *acceleration*, which it uses to make promotion decisions. We observe that this synthesized heuristic outperforms ARMS by 14.2% on GUPS on CXL and matches the performance of ARMS on all other workloads, showing that heuristics synthesized on an emulator can generalize to real hardware.

In Figure 10, we repeat this experiment, using other workloads for training during evolution to obtain heuristics Vulcan-GapBC and Vulcan-Silo. We observe that all three heuristics meet or exceed the performance of ARMS for GapPR, GUPS, and Silo workloads. The only exception to this trend is Vulcan-GapBC, which only achieves 0.96× the performance of ARMS on its training workload of GapBC; we hypothesize that this could be because GapBS-BC is a workload that performs breadth-first search over a graph, and hence can produce random pointer-chasing patterns. Such patterns may not be well captured by heuristics that score pages individually. Synthesizing these heuristics cost $36 in LLM calls (5.8M input, 0.9M output tokens) and required a combined total of 7.8 node-hours on our emulated CXL setup.

8 Discussion

8.1 Developer Effort in Vulcan

Vulcan changes not only the amount of developer effort required for synthesis but also *when* that effort is incurred. With unconstrained synthesis, each synthesized heuristic is a new free-form implementation: before deployment, a developer must determine *where* changes were introduced, which external systems the heuristic interacts with, and whether those interactions are correct. With Vulcan, developers instead define this external-facing boundary before the search begins, through the *template*.

Setting up the template requires developers to specify which raw signals are available to the policy, configure a `libVulcan` feature store to collect and update these signals, and recast the policy's decision logic using Rank/Value(§4). Table 8 quantifies this setup effort for the use cases we study.

Table 8. Developer effort to set up templates.

Metric	Spot VM (single)	Spot VM (multi)	Cache eviction	Memory tiering
Raw signals	8	11	6	3
Template (LoC)	62	104	74	72

Table 8 shows that this refactoring is a modest one-time cost: our case studies required exposing only 3–11 raw signals and writing 62–104 lines of task-specific code for the template. This effort is then amortized across instances requiring specialization. For example, in cache eviction (§7.2), the same 74-line template was reused to synthesize heuristics for all 32 instances.

This fixed synthesis boundary also makes synthesized artifacts more predictable: every heuristic implements the predefined Rank or Value interface and operates over the same declared signals. Thus, even when a developer chooses to inspect a synthesized heuristic in Vulcan, they already know the shape of what they are looking at, substantially reducing the effort required to review each synthesized heuristic.

8.2 Vulcan's Rank and Value Interfaces

8.2.1 Scope and Limitations. While our interfaces capture the core decision logic of a broad class of systems policies (Table 1), we do not claim they are universal. More complex problems can often be decomposed into Rank and Value subproblems: as we demonstrate in our multi-region spot-VM case study (§7.1.2), a Value function decides whether to migrate, and a Rank function then selects the destination region only if the first policy *chooses* to migrate. Such decompositions, however, impose explicit structure on how sub-policies interact: in this setting, the first policy cannot decide to migrate *because* an attractive destination exists. As the number of coupled decisions grows, decomposing them into separate components requires increasingly elaborate coordination between them. For recurring patterns of this kind, extending Vulcan with new interfaces may provide cleaner abstractions than further composition.

8.2.2 Extending Vulcan with New Interfaces. The key requirement for a new interface is that its synthesized decision logic remain stateless. This preserves Anvil's deliberately restrictive design: synthesized programs cannot allocate heap memory or maintain persistent mutable state, enabling strong execution-safety guarantees by construction. For example, a new `COMPARE` interface could choose between two alternatives based on their features jointly, unlike Rank, which scores each candidate independently. Similarly, a `CLASSIFY` interface could return an element from a finite set of actions directly, rather than encoding categorical choices through Value's scalar output. More generally, recurring decision abstractions that admit stateless decision logic could be incorporated into Vulcan as additional interfaces.

9 Related Work

Traditional learning-based policies. Prior ML-based systems policies, particularly neural policies [2, 50, 91, 104, 106, 114], have demonstrated that data-driven approaches can outperform fixed heuristics on specific instances. However, neural approaches introduce significant practical challenges: opaque behavior that complicates debugging [62], complex training and deployment pipelines [3, 27], inference overheads in the control path [27, 110], and safety concerns that limit adoption [83]. Vulcan takes a different stance: it avoids neural inference in the hot path entirely, confining learning to an offline search over small, interpretable LLM-generated code snippets.

AI-driven algorithm and heuristic design. Recent work has explored using LLMs to discover new heuristics via evolutionary search [68, 81]. One line of work modifies existing human-designed algorithms incrementally, such as automatic BBR variants [20, 36]. A complementary line focuses on search strategies: Glia [33] employs multiple specialized LLM agents, and Robusta [44] uses counterexamples to harden candidate heuristics. As we show in §2.2, unconstrained approaches of this kind can produce safety and integration bugs. These search strategies are complementary to Vulcan: any of them, combined with Vulcan's Rank and Value interfaces and listeners, can yield similar benefits.

Domain-specific languages for execution safety. Restricted languages such as Rust and eBPF [28, 84] require complex verifiers and carry steep learning curves; even then, they do not enforce leak-freedom [28, 46], which is too restrictive a guarantee for general programs. General-purpose verifiers [52, 54] can encode execution-safety properties as pre- and post-conditions, but require substantial per-program annotation effort, including loop invariants, termination arguments, and framing conditions [35, 67]. Anvil takes the opposite extreme: by restricting LLM-written code to simple stateless functions implementing Rank or Value, it ensures P1–P3 by construction without any annotations.

10 Conclusion

This paper presents Vulcan, a framework for synthesizing systems resource-management heuristics that are both performant and safe to deploy. Across three use cases, Vulcan discovers heuristics that are competitive with strong human-designed baselines, while avoiding the execution-safety and system-integration problems that arise with unrestricted code generation.

More broadly, Vulcan's results suggest that safe synthesis need not come at the cost of meaningful policy exploration. An important next step is to explore how far this boundary can be expanded, for example by using stronger end-to-end semantic verification to safely expose a larger synthesis surface. Another direction is to formalize what constitutes an *instance* in practice: when to synthesize a new heuristic rather than reuse an existing one, what should trigger re-synthesis, and how to handle transitions between policies.

11 Acknowledgements

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576. This research was also supported by the InfraAI Center at The University of Texas at Austin. Dwivedula was supported by an Amazon AI Fellowship. We are also grateful to our shepherd for their thoughtful feedback, which helped improve the clarity and presentation of this paper.

References

[1] Soheil Abbasloo, Yang Xu, and H Jonathan Chao. 2019. C2TCP: A flexible cellular TCP to meet stringent delay requirements. *IEEE Journal on Selected Areas in Communications* 37, 4 (2019), 918–932.

[2] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. 2020. Classic Meets Modern: a Pragmatic Learning-Based Congestion Control for the Internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 632–647. doi:10.1145/3387514.3405892

[3] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. 2020. KML-IB: Towards machine learning for operating systems. In *Proceedings of the On-Device Intelligence Workshop, co-located with the MLSys Conference*. 1–6.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1851182.1851192

[5] Ami Tavory and Vladimir Dreizin and Benjamin Kosnik. [n. d.]. *Policy-Based Data Structures*. GNU C++ Library, libstdc++, GNU Project. https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ Accessed July 9, 2025.

[6] Anthropic. 2025. Claude Code: Best practices for agentic coding. https://www.anthropic.com/engineering/claude-code-best-practices Accessed: 2026-04-08.

[7] Anthropic. 2025. Introducing Claude Opus 4.5. https://www.anthropic.com/news/claude-opus-4-5 Accessed: 2026-04-08.

[8] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 329–342.

[9] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. 2020. Caching with delayed hits. 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*. 495–513.

[10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[11] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[12] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI*

18). USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann

[13] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[14] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching–Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 195–212.

[15] Zhuohang Bian, Feiyang Wu, Teng Ma, and Youwei Zhuo. 2025. Tokencake: A KV-Cache-centric Serving Framework for LLM-based Multi-Agent Applications. *arXiv preprint arXiv:2510.18586* (2025).

[16] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 499–511.

[17] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-based congestion control. *Commun. ACM* 60, 2 (2017), 58–66.

[18] Jiayi Chen, Nihal Sharma, Tarannum Khan, Shu Liu, Brian Chang, Aditya Akella, Sanjay Shakkottai, and Ramesh K Sitaraman. 2023. Darwin: Flexible Learning-based CDN Caching. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 981–999. doi:10.1145/3603269.3604863

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

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

[21] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, Palo Alto, CA, USA.

[22] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC vivace: Online-Learning congestion control. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 343–356.

[23] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 37–48. doi:10.1145/3307681.3325398

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[25] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators!. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*. 51–60.

[26] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[27] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2023. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 846–861. doi:10.1145/3575693.3575697

[28] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. 2024. The ebpf runtime in the linux kernel. *arXiv preprint arXiv:2410.00026* (2024).

[29] P. Goyal, H.M. Vin, and Haichen Cheng. 1997. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 690–704. doi:10.1109/90.649569

[30] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 65–80. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_altruistic

[31] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 81–97. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_graphene

[32] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[33] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. 2025. Glia: A Human-Inspired AI for Automated Systems Design and Optimization. *arXiv preprint arXiv:2510.27176* (2025).

[34] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. 2020. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 173–190.

[35] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 1–17.

[36] Zhiyuan He, Aashish Gottipati, Lili Qiu, Yuqing Yang, and Francis Y Yan. 2025. Congestion Control System Optimization with Large Language Models. *arXiv preprint arXiv:2508.16074* (2025).

[37] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[38] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating*

Systems Principles. 167–181.

[39] Intel Corporation. 2018. *Intel 64 and IA-32 Architectures Software Developer Manuals*. https://software.intel.com/articles/intel-sdm

[40] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* (2024).

[41] Wenzel Jakob. 2024. pybind11 Documentation.

[42] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450.

[43] Juncheng Yang (1a1a11a). 2023. libCacheSim: a high performance library for building cache simulators. https://github.com/1a1a11a/libCacheSim. Accessed: 2025-06-28.

[44] Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. 2025. Robust Heuristic Algorithm Design with LLMs. *arXiv preprint arXiv:2510.08755* (2025).

[45] Mohammed F Kharma, Soohyeon Choi, Mohammad Alkhanafseh, and David Mohaisen. 2026. Security and quality in llm-generated code: A multi-language, multi-model analysis. *IEEE Transactions on Dependable and Secure Computing* (2026).

[46] Steve Klabnik, Carol Nichols, and Chris Krycho. [n. d.]. Reference Cycles Can Leak Memory. In *The Rust Programming Language*. Chapter 15.6. Rust 1.90.0, Edition 2024. Accessed: 2026-05-08.

[47] Kubernetes. [n. d.]. Horizontal Pod Autoscale. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: Dec 2025.

[48] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. doi:10.1145/3387514.3406591

[49] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 219–234.

[50] Daniar H. Kurniawan, Rani Ayu Putri, Peiran Qin, Kahfi S. Zulkifli, Ray A. O. Sinurat, Janki Bhimani, Sandeep Madireddy, Achmad Imam Kistijantoro, and Haryadi S. Gunawi. 2025. Heimdall: Optimizing Storage I/O Admission with Extensive Machine Learning Pipeline. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1109–1125. doi:10.1145/3689031.3717496

[51] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349* (2025).

[52] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. 2024. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 438–454.

[53] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*.

[54] K Rustan M Leino and Michał Moskal. 2014. Co-induction simply: Automatic co-inductive proofs in a program verifier. In *International Symposium on Formal Methods*. Springer, 382–398.

[55] Jianheng Ling, Pratik Worah, Yawen Wang, Yunchuan Kong, Anshul Kapoor, Chunlei Wang, Clifford Stein, Diwakar Gupta, Jason Behmer, Logan A. Bush, Prakash Ramanan, Rajesh Kumar, Thomas Chestna, Yajing Liu, Ying Liu, Ye Zhao, Kathryn S. McKinley, Meeyoung Park, and Martin Maas. 2025. LAVA: Lifetime-Aware VM Allocation with Learned Distributions and Adaptation to Mispredictions. arXiv:2412.09840 [cs.DC] https://arxiv.org/abs/2412.09840

[56] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. doi:10.1145/3472883.3487003

[57] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 593–608. https://www.usenix.org/conference/atc22/presentation/mansi

[58] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. doi:10.1145/3341302.3342080

[59] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[60] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA, USA.

[61] Memcached. [n. d.]. Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2026-04-22.

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

[63] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.

[64] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 537–550. doi:10.1145/2785956.2787510

[65] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 1–23.

[66] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*

(OSDI 20). USENIX Association, 481–498. https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[67] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 252–269.

[68] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131

[69] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. 2010. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review* 44, 3 (2010), 2–19.

[70] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux Journal* 2009, 184 (2009), 4.

[71] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[72] SeongJae Park, Yunjae Lee, and Heon Y Yeom. 2019. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*.

[73] Padmanabhan Pillai and Kang G Shin. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 89–102.

[74] Steven J Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood, and Mike Davis. 2006. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *2006 IEEE International Conference on Cluster Computing*. IEEE, 1–7.

[75] François Pottier and Yann Régis-Gianas. 2024. Menhir: An LR(1) Parser Generator for OCaml. https://gallium.inria.fr/~fpottier/menhir/

[76] Sriram Ramabhadran and Joseph Pasquale. 2003. Stratified round robin: A low complexity packet scheduler with bandwidth fairness and bounded delay. In *Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications*. Association for Computing Machinery, New York, NY, USA, 239–250. doi:10.1145/863955.863983

[77] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. 2025. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703* (2025).

[78] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[79] Redis. [n. d.]. Redis - an in-memory data structure store, used as a database, cache, and message broker. https://redis.io/. Accessed: 2026-04-22.

[80] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 341–354.

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

[82] Marta Rybczyńska. 2021. Introducing maple trees. LWN.net. https://lwn.net/Articles/845507/ Accessed: 2026-03-22.

[83] Divyanshu Saxena, Jiayi Chen, Sujay Yadalam, Yeonju Ro, Rohit Dwivedula, Eric H Campbell, Aditya Akella, Christopher J Rossbach, and Michael Swift. 2025. How I learned to stop worrying and love learned OS policies. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, New York, NY, USA, 1–7.

[84] Matan Shachnai, Harishankar Vishwanathan, Srinivas Narayana, and Santosh Nagarakatte. 2024. Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel. In *International Static Analysis Symposium*. Springer, Springer Nature Switzerland, Cham, 386–406.

[85] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[86] Erfan Sharafzadeh, Raymond Matson, Jean Tourrilhes, Puneet Sharma, and Soudeh Ghorbani. 2025. Self-Clocked Round-Robin Packet Scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1437–1465.

[87] Asankhaya Sharma. 2025. Openevolve: An Open-Source Evolutionary Coding Agent. https://github.com/codelion/openevolve. Accessed: 2025-12-10.

[88] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[89] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 231–242.

[90] Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2025. *Code Researcher: Deep Research Agent for Large Systems Code and Commit History*. Technical Report MSR-TR-2025-34. Microsoft. https://www.microsoft.com/en-us/research/publication/code-researcher-deep-research-agent-for-large-systems-code-and-commit-history/

[91] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. 2020. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544.

[92] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. 2023. HALP: Heuristic aided learned preference eviction policy for YouTube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1149–1163.

[93] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Esha Choukse, Haoran Qiu, Rodrigo Fonseca, Josep Torrellas, and Ricardo Bianchini. 2025. TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1266–1281. doi:10.1145/3676641.3716025

[94] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: Designing LLM Inference Clusters

for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. doi:10.1109/HPCA61900.2025.00102

[95] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[96] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[97] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, 95–110.

[98] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

[99] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[100] Wikimedia Foundation. [n. d.]. WikiMedia Cache Dataset. https://wikitech.wikimedia.org/wiki/Data_Platform/Data_Lake/Traffic/Caching. Accessed: 2025-05-14.

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

[102] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[103] Sujay Yadalam, Konstantinos Kanellis, Michael Swift, and Shivaram Venkataraman. 2025. ARMS: Adaptive and Robust Memory Tiering System. *arXiv preprint arXiv:2508.04417* (2025).

[104] Chenxi Yang, Divyanshu Saxena, Rohit Dwivedula, Kshiteej Mahajan, Swarat Chaudhuri, and Aditya Akella. 2024. C3: Learning Congestion Controllers with Formal Certificates. *arXiv preprint arXiv:2412.10915* (2024).

[105] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[106] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 115–134. https://www.usenix.org/conference/fast23/presentation/yang-juncheng

[107] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[108] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 130–149.

[109] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. 2022. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1021–1036.

[110] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. 2022. LiteFlow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 414–427. doi:10.1145/3544216.3544229

[111] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. 2024. SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1229–1246. https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo

[112] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

[113] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. 2018. Demystifying Cache Policies for Photo Stores at Scale: A Tencent Case Study. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/3205289.3205299

[114] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 2025. 3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*. USENIX Association, Santa Clara, CA, 237–254.

[115] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).

A Challenges with LLM-Based Heuristic Generation

To evaluate whether current LLMs can synthesize effective systems heuristics that meet these requirements (Table 3), we used a state-of-the-art model, Claude Opus 4.5, to generate caching heuristics in libCacheSim [43].

libCacheSim [43] is an event-driven cache simulator, in which new heuristics are defined by implementing event handlers for cache operations such as accesses, evictions, and insertions – each of which has a specific function signature detailed in the documentation (a systems integration requirement). Additionally, the libCacheSim engine also provides access to commonly used features – such as total object counts, the current timestamp, and the number of hits/misses – via raw pointers, which heuristics are allowed to read but must not mutate (another systems integration requirement).

A.1 Attempt 1: A Simple Evolutionary Loop

We implement an evolutionary loop using a simple LLM agent (inspired by prior work [25, 81]), as shown in the left half of Figure 11. The agent is provided with a prompt describing libCacheSim and its interfaces, which it uses to

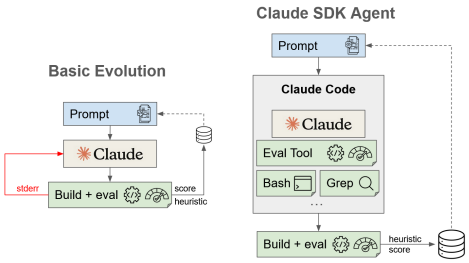


Figure 11. Evolutionary search with a simple agent vs. Claude Code. The dotted line represents the start of a new iteration; every iteration is seeded with samples of the best-performing programs so far to help guide the search.

generate a candidate heuristic. The generated heuristic is then built and evaluated on a suite of ten traces from the CloudPhysics dataset [97]. If a build or runtime error occurs, the agent receives the `stderr` output to iteratively refine and correct its code. Once the heuristic compiles and executes successfully, both the implementation and its performance metrics (*i.e.,* miss rates) are stored in a database. Each such iteration constitutes the *generation* of a single heuristic. This process is repeated for fifty iterations, with higher-performing heuristics sampled from the database and incorporated into subsequent prompts to guide the search toward better designs.

Manual inspection revealed a recurring issue across all fifty heuristics. In libCacheSim, heuristics must report their per-object metadata footprint (in bytes) via a system variable used for memory accounting. Despite introducing additional per-object state, the LLM consistently under-reported this value. As a result, these heuristics exceeded their memory budget, yielding unfair comparisons against baselines that correctly account for memory usage and violating interface requirements (a system-integration bug).

Despite these issues, after fifty iterations, the evolutionary process produced heuristics that "achieved" a 3%–5.1% improvement over FIFO according to the evaluator. However, these gains were partly driven by exploiting gaps in the evaluation harness (*i.e.,* under-reporting metadata usage). Moreover, heuristics with memory safety bugs also passed without crashing, as the harness exercised only limited execution paths. This shows that heuristics which compile, run successfully, and perform well on the evaluator may still violate the desired properties in Table 3.

A.2 Attempt 2: Agentic Code Synthesis

After initial experiments with simple agents showed that LLM-synthesized code often violates desired properties, we asked whether these issues persist in more capable agentic frameworks that augment LLMs with external tools. To investigate, we repeated the heuristic synthesis experiment using Claude Code – a widely used coding agent that equips an LLM with capabilities such as file manipulation, shell execution, and search. Such agents may better avoid interface-compliance bugs observed earlier, as they can spawn subagents to reason about and understand the internals of the systems they interact with more effectively.

To test this hypothesis, we repeated the experiment using the Claude Code Agent SDK (Claude Opus 4.5) to synthesize each candidate heuristic, as shown in the right half of Figure 11. In addition to default tools (bash, search, read, etc.), we provided a custom `evaluate` tool that builds the heuristic, runs it on CloudPhysics traces, and returns a JSON report of its performance relative to FIFO. Although the agent could execute these steps via bash, exposing them as a dedicated tool simplifies its workflow.

Using this more capable agentic setup, we ran the evolutionary process for ten iterations, producing ten heuristics. As in Attempt 1, each iteration generates a candidate via a loop of proposing, evaluating, and refining; however, unlike the earlier setup – where the loop was limited to handling compiler and runtime errors – each iteration here is a richer multi-turn interaction. The agent can explore the codebase, reason about interfaces, modify implementations, and invoke tools such as `evaluate` multiple times before returning a candidate heuristic.

Generating each candidate heuristic with Claude Code took 11–28 minutes, consumed ~2M tokens (~$3.20 via AWS Bedrock), and involved an average of 57.4 tool calls. The resulting heuristics achieved a 36.4%–46.8% improvement over FIFO, as scored by the evaluator, significantly outperforming those found by the simpler agent.

They also exhibited diverse strategies: the best-performing heuristic uses a prioritization function combining frequency tiers, size-cost normalization, and recency protection, while the second-best applies a concave transformation (a sqrt–log blend) to weight access frequency.

Unlike the previous setup – where even modest gains were partly driven by reward hacking (*i.e.,* under-reporting metadata to bypass memory accounting) – the heuristics produced by this agentic framework did not exploit that loophole. However, several still violated interface assumptions: for instance, several heuristics – including the best-performing one – overwrote `next_access_vtime` (per-object state maintained by libCacheSim) to store the current time, violating the system interface by treating a read-only field as writable (a systems integration bug). Additionally, a double-free bug (an execution safety bug) in the second-best heuristic went undetected by the evaluator; libCacheSim exposes two removal paths – `evict()`, which selects and removes objects, and `remove()`, which deletes objects by ID. Since our evaluator exercised only the `evict()` path, the bug in `remove` went undetected.

A.3 Summary

This case study reinforces a growing body of evidence that LLMs routinely introduce bugs and safety violations when

generating complex code [40, 45, 71, 115] – and that more capable agents are no exception, frequently requiring humans in the loop to produce reliable output [77, 90].

While LLMs may be more successful at synthesizing code for self-contained coding tasks [10, 19, 37], systems heuristics are usually more complex: they may maintain internal state in complex data structures [82], involve intricate state-transition logic [32, 60, 108], or be dispersed across multiple files and control paths [57]. Allowing an LLM to synthesize or modify such heuristics end-to-end – reasoning simultaneously about policy logic, state management, and mechanisms – widens the attack surface for subtle bugs and property violations in the synthesized code.

B RANK-based Cache Eviction Heuristics

B.1 Results for Same-Size Instances

Figure 12 reports results for two classes of instances in which cached objects have uniform sizes. As presented in §7.2, we compare these heuristics with nine human-designed baselines and find that VULCAN-generated heuristics are generally competitive against strong manually designed baselines.

B.2 Synthesis Prompt

The natural-language prompt used for the cache eviction case study (§7.2) is shown in Listing 5. The prompt also includes dynamically selected examples of high-performing scoring functions from previous rounds, selected automatically by the evolution frameworks [51, 87].

Listing 5: Prompt for synthesizing cache eviction heuristics

Context. You are a systems researcher designing a new cache eviction policy. Whenever the cache is full and a new object must be inserted, this policy is invoked to select which cached object to evict. The policy has access to per-object features such as object size, insertion time, last access time, and access count, along with global information about recently evicted objects.

Policy Model. The policy is implemented using `vulcan`, a framework that separates *policy* from *mechanism*. Your task is to implement a *scoring function* that returns the **utility** of an object:

$$\text{Higher utility} \Rightarrow \text{more valuable to keep}$$

When eviction is required, select the object with the *lowest utility*.

Lazy Priority Queue Execution Model. The scoring function is evaluated lazily:

- A score is recomputed only when an object is inserted or accessed.
- The resulting score is cached in the priority queue.
- Idle objects are *not* rescored over time.

Therefore, expressions such as `curr_time - last_access` are ineffective because, at rescore time, `curr_time` $\approx$ `last_access`.
Instead, use:

- `last_access` directly for recency,
- `insertion_time` directly for age ordering,
- or other stable per-object signals.

Global Features.

- `f_curr_time (i64)`: Current logical time.
- `f_ghost (i64)`: IDs of recently evicted objects.

Per-Object Features.

- `f_size (i64)`: Object size in bytes.
- `f_insertion_time (i64)`: Insertion timestamp.
- `f_last_access (i64)`: Most recent access timestamp.
- `f_count (i64)`: Access count since insertion.

Listeners. Listeners determine how feature values are stored and queried. Examples include:

- `RollingWindow(N)`
- `EWMA({alpha})`
- `RollingPercentile(N)`
- `RollingCount(N)`

Example listener configuration:

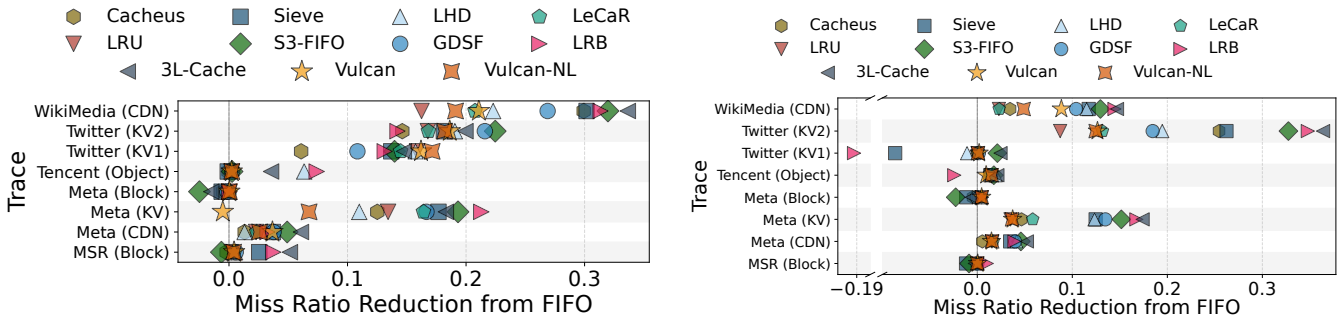
```
fs.add_listeners(
  f_last_access,
  {vulcan::listeners::object::RollingWindow(5)}
);

fs.add_listeners(
  f_curr_time,
  {vulcan::listeners::global::RollingWindow(1)}
);
```

Expected Output Format. The generated file must contain:

1. Listener configuration
2. Scoring function

Example Scoring Function.

```
auto scoring_fn =
  [&](
    vulcan::store& fs,
    int64_t obj_id
  ) -> double {
    double last_acc = fs.get_latest(
      f_last_access, obj_id
    );
    double count = fs.get_latest(
      f_count, obj_id
    );
    double ghost_count = fs.get_count(
      f_ghost, obj_id
    );
    double ghost_penalty = ghost_count * 1e6;
    return last_acc+(count*10)-ghost_penalty;
};
```



(a) Large cache size (10%), objects of the same size

(b) Small cache size (0.1%), objects of the same size

Figure 12. Performance of instance-specialized heuristics versus baselines for two classes of instances.

C Artifact Appendix

Our artifact contains: (a) the final discovered heuristics for each of our use cases, (b) orchestrator and plotting scripts to recreate the main results of the paper, (c) scripts to run heuristic discovery loops for each use case with off-the-shelf evolution frameworks [51, 87], and (d) implementations of `libVulcan` and Anvil and documentation on how to extend and use these libraries to discover heuristics for new tasks. Our artifact is available at the following links:

- Main repository containing code for all four use cases.
  - github.com/ldos-project/vulcan-usecases
- Common libraries:
  - libVulcan (DOI: 10.5281/zenodo.22905536)
  - Anvil (DOI: 10.5281/zenodo.22905483)
- Discovered heuristics and precomputed results (DOI: 10.5281/zenodo.20361337)

Each use case has a separate repository branch. Check out the appropriate branch of `vulcan-usecases` and follow its `README.md`. For each use case, we provide precomputed results (e.g., JSON files and dumps) that were used to create the plots – use these dumps directly or rerun the evaluation.

C.1 Reproducing Spot VM Single-Region Results

- Branch: `cbl-artifact`;
- §7.1.1;
- Plot(s) reproduced: Figure 5.
- Plotting script: `python3 plot_main_result.py`.
- DOI: 10.5281/zenodo.22905385

C.2 Reproducing Spot VM (Multi-Region) Results

- Branch: `cbl-multi-artifact`;
- §7.1.2;
- Plot(s) reproduced: Figure 7.
- Plotting script: `plot_results.py`
- DOI: 10.5281/zenodo.22905389

C.3 Reproducing Caching Results

- Branch: `caching-artifact`;
- §7.2;
- Plot(s) reproduced: Figure 8 and Figure 12.
- Plotting script: `plot_workload_instances.py`
- DOI: 10.5281/zenodo.22905373

C.4 Reproducing Memory Tiering Results

Note: for this use case, the artifact reproduces results from the emulated setup and requires CloudLab.

- Branch: `tiering-artifact`;
- §7.3;
- Plot(s) reproduced: Figure 9.
- Plotting script: `plot_improvement.py`
- DOI: 10.5281/zenodo.22905376

C.5 Running Evolution and Adding Use Cases

Beyond reproducing the final results, each use case also includes the synthesis harness and prompts, allowing users to rerun the full discovery loop with an off-the-shelf evolution framework (OpenEvolve or ShinkaEvolve). Additionally, users who wish to use our framework for their use cases can find documentation in libvulcan.