

VULCAN: Instance-Optimal Systems Heuristics Through LLM-Driven Search

Rohit Dwivedula Divyanshu Saxena Sujay Yadalam Daehyeok Kim Aditya Akella
The University of Texas at Austin

Abstract

Resource-management tasks in modern operating and distributed systems continue to rely primarily on hand-designed heuristics for tasks such as scheduling, caching, or active queue management. Designing performant heuristics is an expensive, time-consuming process that we are forced to continuously go through due to the constant flux of hardware, workloads and environments.

We propose a new alternative: synthesizing *instance-optimal* heuristics – specialized for the exact workloads and hardware where they will be deployed – using code-generating large language models (LLMs). To make this synthesis tractable, VULCAN separates policy and mechanism through *LLM-friendly*, task-agnostic interfaces. With these interfaces, users specify the inputs and objectives of their desired policy, while VULCAN searches for performant policies via evolutionary search over LLM-generated code. This interface is expressive enough to capture a wide range of system policies, yet sufficiently constrained to allow even small, inexpensive LLMs to generate correct and executable code.

We use VULCAN to synthesize performant heuristics for cache eviction and memory tiering, and find that these heuristics outperform all human-designed state-of-the-art algorithms by upto 69% and 7.9% in performance for each of these tasks respectively.

1 Introduction

Systems research has long treated heuristic design as a manual craft. Performance-critical systems rely on hand-written heuristics – tuned for typical conditions and deployed as static, one-size-fits-all policies. However, decades of work across caching [20, 126], congestion control [77], kernel queueing disciplines [94], memory tiering [109], and more – demonstrate the same lesson: there is no universal heuristic. Different settings demand different heuristics, and specialization is essential for performance.

So, we continually tune and redesign (§2.1): we tweak congestion control algorithms for new network environments [104], tailor prefetching or caching policies for emerging workloads [17] or hardware [116], and evolve queueing disciplines for new performance targets [25]. Yet despite this retuning, we continue to find instances of sub-optimal performance from these manually-designed heuristics [9, 57, 89]. The heuristic design space continually shifts with context-dependent behaviors and changing performance trade-offs, making it increasingly difficult to discover and maintain the ‘right’ (*i.e.*, performant) heuristics for each scenario. This raises a fundamental question: **how can we dis-**

cover “the right heuristics” fast enough, to keep pace with ever-evolving deployment conditions?

We aim to tackle this ever-lasting challenge using VULCAN, a framework in which we recast the heuristic design problem as an *an automated search problem* using large language models (LLMs): instead of hand-crafting heuristics, we repeatedly invoke generative models to produce *instance-optimal heuristics* tailored to each deployment context. Our framework is inspired by recent advances in LLM-powered algorithm synthesis [19, 71, 86], which combine generative models with evolutionary search to discover expressive code that maximizes a target quantitative reward function.

Directly applying these search-based techniques to synthesize systems heuristics, however, is non-trivial (§2.3). Modern systems heuristics tightly couple *policy* (high-level decision logic) with *mechanism* (low-level state, data structures, and control paths). As a result, modifying or improving a heuristic requires coordinated reasoning about both intent and implementation. In this setting, simply prompting an LLM to synthesize a complete end-to-end heuristic proves futile (see §2.3). On the other hand, constraining the search to tiny tweaks on top of known algorithms [36] limits the opportunity for genuine innovation.

The central challenge, therefore, is identifying how to leverage generative models where they are strongest today – reasoning over structured signals and expressing high-level decision logic – without requiring them to correctly implement complex mechanisms or manage intricate system state. VULCAN resolves this tension by placing LLM-driven evolutionary search at the core of heuristic design, while exposing a simple, structured **three-step pipeline** (Figure 3 in §3) that cleanly separates policy from mechanism and makes the search tractable.

In the first step of the pipeline, users define the search space by casting the problem into a narrow interface (§3.1). Our key observation is that many systems heuristics can be expressed in one of two forms: (i) a function that computes a value from system state (e.g., congestion controllers computing a congestion window), which we term a **VALUE-type heuristic**; or (ii) a function that ranks a set of objects and selects among them (e.g., schedulers ranking runnable tasks), which we term a **RANK-type heuristic**. This interface captures the policy logic cleanly. The corresponding mechanism – the concrete implementation that executes these decisions – is developed separately. This separation exposes a rich mechanism design space for both policy interfaces, which we systematically explore in this paper (§3.1.1 and §3.1.2).

Second, users define the instance for which they seek a specialized heuristic – either by specifying a concrete work-

load–hardware pair, or by configuring an automated instance generator (§3.2) that algorithmically delineates instances based on observed workload characteristics and system signals. Today, heuristics are redesigned only when workloads or objectives change substantially, not because smaller changes are unimportant, but because manual heuristic design is expensive and time-consuming; consequently, even redesigned heuristics are typically made as broadly applicable as possible. VULCAN fundamentally changes this cost model: by dramatically reducing the human cost of heuristic creation, it becomes practical to synthesize heuristics for much more narrower instances, enabling specialization even for modest shifts such as changes in input parameters, evolving access patterns, or transient workload phases. In effect, VULCAN enables *instance-optimal policies*: rather than relying on coarse, one-size-fits-all heuristics that underperform across heterogeneous conditions, specialization becomes the default.

Finally, users specify how to invoke the evolutionary search process (§3.3). This includes describing the system context, objectives, and available state in natural language, and, critically, defining the *evaluation harness* used to assess candidate heuristics. As with earlier stages of the pipeline, this step exposes important design trade-offs, such as balancing evaluation speed against fidelity, choosing appropriate metrics of interest, and combining multiple objectives into a single optimization target. We discuss these considerations and illustrate, through concrete examples, how to effectively invoke the search process.

In this paper, we instantiate VULCAN for two policies: cache eviction (§4) and page promotion in tiered-memory systems (§5). We discuss specific choices we made for both of these policies in the design space exposed by the aforementioned three-step process. Across ten different *instances*, cache eviction policies synthesized by VULCAN either nearly match state-of-the-art baselines or outperform them by 1.94% to 69%. Memory tiering policies designed with VULCAN yield 2.5-7.9% improvement over state-of-the-art baselines, for a variety of workloads.

While VULCAN automates the search for effective heuristics, it does not obviate the role of human designers. Our position is that human expertise is most valuable not in hand-crafting policy minutiae, but in structuring the problem so algorithmic search can succeed. We argue that this structure arises from defining clean interfaces that control the space in which heuristics are discovered, providing the scaffolding that makes LLM-guided design both tractable and meaningful.

Conversely, VULCAN advocates using LLMs where it is most effective today: generating code that embodies candidate heuristics while respecting the interface and its invariants. This stands in contrast to prior work on neural policies, whose opaque representations and substantial runtime overheads make them difficult to deploy in performance-critical systems. By having LLMs synthesize concise, human-readable decision logic, VULCAN enables rapid exploration of the heuristic

design space without sacrificing interpretability or implementation efficiency.

2 Background and Motivation

Systems research has long relied on hand-crafted heuristics to implement policies for tasks such as cache eviction, congestion control, and queue scheduling. Heuristics arise because the optimal action is either (1) fundamentally intractable to compute, due to large action spaces, fine-grained decision intervals, and NP-hard underlying problems, or (2) effectively unknowable because key system variables are latent or unobservable. Consequently, researchers continually design new heuristics that approximate optimal behavior under specific workload, hardware, and operating conditions, each tuned for distinct objectives such as performance, fairness, utilization, or scalability. In practice, this iterative process is a sustained search for *instance-optimal* heuristics: policies that perform well, not in general, but for the particular deployment context at hand.

2.1 The pursuit of instance-optimality

Taking the concrete example of eviction policies in web caches, different heuristics for specific workloads, objectives, or deployment scenarios [13, 44, 62, 85, 96, 102, 115, 120] have been proposed – *e.g.*, some [62, 120] perform well for large cache sizes, while others [13, 44] are more suited for smaller caches. It has also been demonstrated that depending on whether a workload consists of mostly new objects (“scan workloads”) or mostly repeated objects (“churn workloads”), different algorithms perform better [85]. Additionally, these eviction heuristics have been tailored for end-to-end *objectives* (or their combinations) such as tail latency [15] and fairness [52], or system-level *constraints* such as CPU overhead [96], lock-free design [115], and memory efficiency [28]. To demonstrate this more concretely, Figure 1 summarizes the results of running 17 caching algorithms [45] on 106 traces from the CloudPhysics dataset [103] on caches of varying sizes – we observe that *no single algorithm* performs the best on even half of the traces and different algorithms perform better for different cache sizes. Even though all of the traces in this dataset are from the same broad workload class – block I/O traces collected over a one week period – we find that one-size-fits-all is not true even within this single class.

These findings are not limited to caching eviction heuristics alone: system policies across the stack require similar instance-dependent heuristics. Congestion control algorithms are often tailored for the Internet [2, 3, 10, 18] versus data-center workloads [8, 51, 65], kernel queueing disciplines are tuned for performance and fairness objectives [31, 81, 89, 92], and cluster schedulers are designed for specific applications, such as data processing [32, 33], VMs [56], or deep learning [67, 108].

Historically, this search for instance-optimality¹ has been

¹We use the term *instance-optimal* to refer to the best-performing heuristic

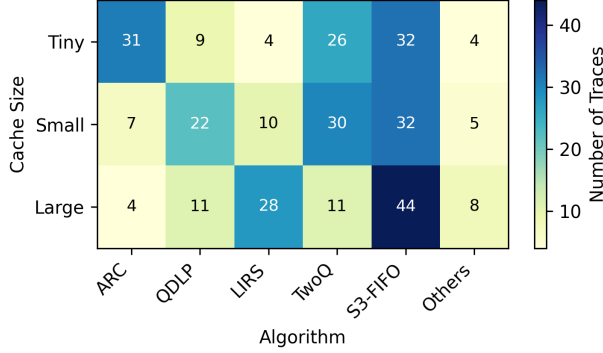


Figure 1: **Count of CloudPhysics traces where each heuristic performs best (highest object hit rate).** Tiny, small, and large caches correspond to 0.1%, 1%, and 10% of the trace footprint (*i.e.*, number of unique objects in a trace); we ignore object sizes, similar to how [120] and [115] report their results. The column “Others” includes SR_LRU [85], CR_LFU [85], Sieve [120], and GDSF [23] all of which had at least one trace in which they were the best heuristic.

conducted manually, by researchers and developers. As workloads grow increasingly diverse, performance objectives become multi-dimensional, and hardware evolves rapidly toward greater heterogeneity, such manual exploration is no longer tractable. This brings to the fore a clear need for an automated search for systems heuristics. Recently, neural models have been seen as an alternative to this search, learning from large datasets to predict near-optimal decisions across varied inputs [3, 56, 60]. However, these approaches come at a steep cost: opaque behavior [63], complex training and deployment pipelines [6, 29], inference overheads [29, 119], and safety concerns [88] make their operation and adoption challenging.

2.2 LLMs as heuristic-generators

We argue that recent advances in LLMs enable a new paradigm of finding such instance-optimal heuristics. This is based on the insight that finding an instance-optimal heuristic can be thought of as a *code-generation* problem: we seek a program that satisfies the input–output specification of the target policy while meeting some non-trivial performance objectives for the given instance. And with the ability of LLMs to generate executable code, refine it from natural-language instructions, and employ chain-of-thought reasoning [106], generating and refining heuristics has become a much faster process.

2.3 Can LLMs be used to synthesize entire heuristics?

Unlike conventional coding tasks that are largely self-contained (*e.g.*, generating boilerplate, or solving programming-contest problems [12, 21, 37]), systems heuristics are not amenable to automated synthesis. They maintain internal state [24, 62], involve intricate state-transition

within a defined search space for a specific workload, objective or hardware; distinct from the theoretical offline optimal (*e.g.*, Belady’s MIN [14]).

logic [115], and are intertwined with mechanisms spread across multiple files and control paths [59]. This complexity stems from the tight coupling between the *high-level intent* of the policy and the *low-level mechanisms* that realize it, making heuristic design a joint reasoning problem in *current* systems. For example, Linux’s CFS scheduler couples its policy with a red–black tree mechanism [76]; I/O schedulers such as BFQ [99] encode responsiveness through per-process queues; DRR-style qdiscs [89, 92] achieve fairness by maintaining per-flow subqueues. In all these cases, even if the policy logic is conceptually simple, modifying or improving it requires coordinating intent with the surrounding mechanism – unfortunately, this is where current LLMs struggle, as shown by multi-function, stateful code-generation benchmarks [41, 125].

While coding agents [91, 105, 111, 122] are increasingly getting better at making changes to large code bases, they often require human supervision. Efforts to automate such agents for large systems codebases require sophisticated mechanisms for exploration and instruction tuning and still only achieve close to 50% success at best [83, 93]. Together, these limitations preclude using current LLMs for automated search over full, mechanism-entangled heuristic implementations.

2.4 Common interface separating policy and mechanism

A key question then is how to design an interface that separates policy from mechanism to enable automated search for instance-optimal heuristics. Our insights here is that many systems policies can be expressed in one of two forms: (i) *computing a value based on the current system state* (VALUE), or (ii) *ranking a set of objects* and selecting the top-K (RANK). If the underlying mechanism is engineered to efficiently consume these abstractions, the search space for an LLM collapses to learning a single function: a `value()` function for VALUE tasks, or a `rank()` function for RANK tasks.

Table 1 provides examples for both of these categories. For example, congestion control [34], horizontal autoscaling [1], cluster admission control [87] and DVFS [79] (among many others) are examples of policies that attempt to *predict a value* using some system state as the input. On the other hand, tasks such as block I/O prefetching [112], scheduling [76] and memory tiering [84, 109] involve *selecting* a subset of objects out of a larger list, and then performing an action on these selected objects.²

A key advantage of using such a narrow interface is that it greatly simplifies correctness checking. Using an LLM to implement an entire heuristic – including both policy and mechanism – opens the door to a wide array of catastrophic failure modes: null-pointer dereferences, incorrect state updates, broken invariants, and subtle memory

²It is interesting to note that the same resource management task could either be a VALUE-type policy or RANK-type policy depending on the problem formulation: for example the two prefetching policies [64, 112] in Table 1 demonstrate this.

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

Policy	Description	Type
Congestion control [34]	Decide how many unacknowledged bytes may be in flight (cwnd).	VALUE: compute the cwnd
DVFS control [79]	Decide what frequency to run the CPU at to balance performance and power.	VALUE: compute the frequency.
Cluster autoscaling [1]	Decide how many replicas to provision to meet workload demand.	VALUE: compute replicas per service.
Hardware prefetching [64]	Choose an offset from the current access to prefetch data.	VALUE: compute the offset value.
Block I/O prefetching [112]	Select blocks to prefetch.	RANK: all historically accessed blocks.
Cache eviction [95]	Select which cached object(s) to evict.	RANK: all cached objects.
CPU scheduling [76]	Select which thread to schedule next.	RANK: all runnable threads.
Page promotion in tiered-memory systems [84, 109]	Select which pages to promote to higher tier.	RANK: all memory pages.

errors. Existing LLM-based code-generation approaches that attempt to synthesize such complex, mechanism-entangled logic typically rely on extensive runtime testing [82], automated verification and invariant checking [66], or detailed human-written semantic specifications [50] to mitigate these risks. In contrast, by sharply constraining the "attack surface" of the LLM to a single, stateless function that returns a numerical value, we make validation nearly trivial. Indeed, every function that returns a real value is a well-formed policy under our interfaces – it may be a poor policy, but it cannot be an invalid one.

Generality of these common interfaces. Table 1 illustrates how diverse resource-management tasks – across a wide spectrum of operating systems and distributed systems – can be expressed in our common interfaces, allowing VULCAN to autonomously search for new, performant heuristics. We now attempt to understand how broadly this observation holds by systematically examining recent systems literature.

Specifically, we surveyed *all* OSDI and NSDI papers published between 2021 and 2025 ($N = 660$). We first filtered for papers whose primary contributions include a *resource-management policy or algorithm*. For each such paper, we then attempted to see if the task(s) in the paper could be formulated and expressed using either the VALUE or RANK interfaces.

Appendix A describes the LLM-assisted paper analysis pipeline we used to identify in-scope papers, extract resource management tasks, and then classify them. Our pipeline identified 191 papers where a new heuristic or algorithm was the key contribution. From these 191 papers, our pipeline identified 234 resource management tasks in total, out of which 71 could be modeled as a VALUE tasks and 158 as RANK tasks. These results suggest that VULCAN’s interfaces are general and widely applicable.

3 Design

VULCAN is a framework that automates the discovery of instance-optimal heuristics. As shown in Figure 3, its workflow consists of three phases (left to right). First, the user defines the target resource-management task and provides a template that constrains and guides the search space. Next, VULCAN uses these inputs in an offline search loop that iteratively generates candidate heuristics, compiles them, and evaluates their performance on a simulator or testbed. Finally, once a performant heuristic is identified, it is deployed onto the production system to handle live workloads.

Users of VULCAN need to follow a *three-step pipeline* to use our framework for their tasks (Figure 2). We structure the remainder of this section around the three key steps, with each step elaborated in a dedicated subsection.

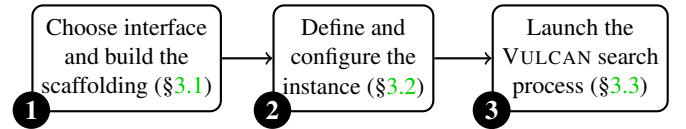


Figure 2: **Three-step user-facing pipeline for instantiating VULCAN.** All of these are part of the *problem specification phase* (Phase I) shown in Figure 3.

3.1 Interface and Scaffolding

The first step in using VULCAN is to cast the ‘instance-optimal heuristic search’ problem into a form that is amenable to LLM-driven search. This requires the users to implement three components – depicted in Figure 4 and described below. Although we describe these as three components for conceptual clarity, VULCAN only requires that policy logic be isolated; in practice, users may tightly couple data collection and action logic if that better matches their system’s structure.

Data collection logic. The user defines the set of *signals* that are exposed to the decision-making policy module. These signals form the entire observation space available to synthe-

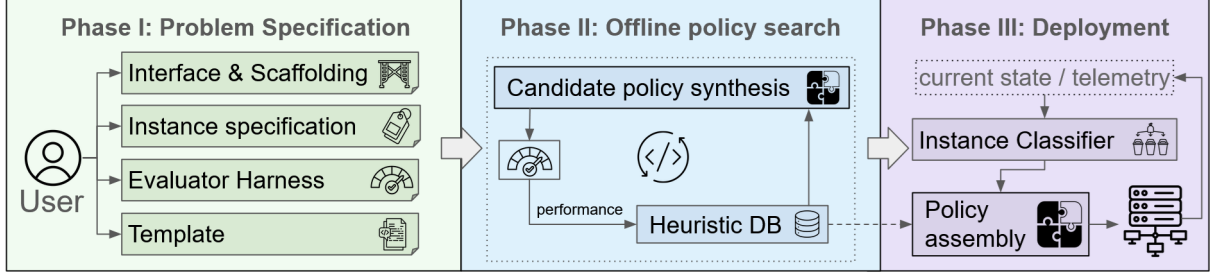


Figure 3: **End-to-end overview of VULCAN.** Each of the inputs defined in Phase I are used by subsequent phases.

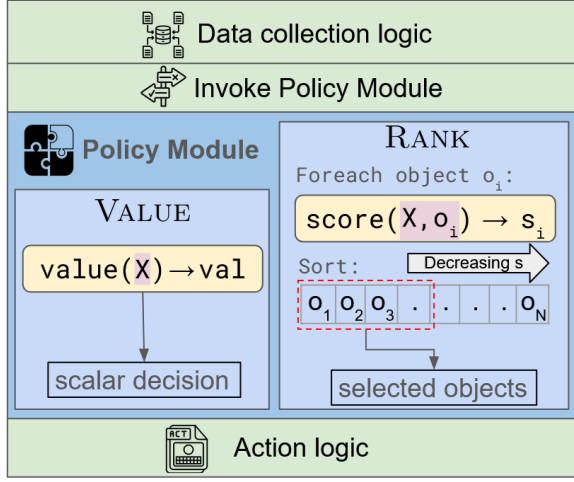


Figure 4: **Division of labour in VULCAN:** users are responsible for implementing mechanisms for data collection and action (in green); LLMs synthesize the scoring functions (in yellow); VULCAN uses templates to assemble these LLM-generated functions into an executable *policy module* (in blue).

sized policies, and thus directly shape the space of heuristics that VULCAN can discover. Unlike traditional heuristic design, where human ingenuity in designing complex features – for instance, via normalization [53], moving averages [110] or exponential decay counters [95] – is often essential, VULCAN does not always require such feature engineering to be manually performed. Even if users expose raw, minimally processed signals, the LLM-guided search can often determine how these signals should be combined, as we show in §5.

The policy module. In VULCAN, the policy module performs all the policy choices and is completely synthesized through automated search and templates. VULCAN requires the user to formulate their problem using common interfaces that collapse the decision making required to a stateless *scoring function* – by casting the problem as either VALUE-type or RANK-type (these interfaces merit detailed discussion, provided in §3.1.1 and §3.1.2). In Figure 4, these scoring functions – are LLM-generated (in yellow). VULCAN dynamically assembles this LLM-generated function into a *policy module* that is injected by recompiling the scaffolding with the synthesized

Table 2: Examples of VALUE tasks and their inputs/outputs

Policy	Typical inputs	Value (output)
Congestion control	RTT, loss, ACK timing, prev_cwnd	next_cwnd (int)
DVFS	CPU util, QoS, power/thermal info	next_freq (float)
Autoscaler	CPU usage, latency, queue length	n_replicas (int)

code, dynamically linking or overriding symbols at runtime, or – for certain kernel-resident policies – injecting logic via eBPF probes.

Action logic. Once the *policy module* makes its decision, that is, a new value in a VALUE task or a subset of objects for a RANK task, the user-defined scaffolding is responsible for consuming this decision and performing the associated actions. For example, for congestion control, the action operator would set the new cwnd to the value returned by the scoring function and ensure that this is the number of unacknowledged packets in flight for the next time window; for cache eviction, the objects selected by rank() are erased from the cache.

With the workflow for this step concretized, we can now discuss the semantics and design details of the VALUE (§3.1.1) and RANK (§3.1.2) interfaces.

3.1.1 Interface: VALUE

Interface definition. VALUE heuristics take as input a set of features (X) that encode the system state, run a function to compute $\text{value}(X)$, and return that value. LLMs are used to synthesize the $\text{value}(X)$ function. Typical inputs and outputs for prototypical VALUE policies are listed in Table 2.

Creation of the policy module. Once the LLM synthesizes this function, VULCAN injects it directly into the scaffolding using the selected integration mechanism. Since the policy consists of a single stateless function that maps system state to a scalar decision, no additional machinery is required.

3.1.2 Interface: RANK

Interface definition. RANK-based heuristics, at a high level, select the topK objects from a larger pool of N candidate objects (objects), using some global features (X) and per-object features – $f_{o_i} \forall o_i \in \text{objects}$

While we could use LLMs to synthesize this entire ranking logic, our RANK interface cleanly separates this into three

Table 3: Examples of RANK tasks

	Memory Tiering	CPU scheduling
objects	memory pages	runnable tasks
global features	bandwidth, memory stalls	load avg, number of tasks
object features	recency, frequency, size	niceness, CPU util
score	represents page hotness	represents task urgency
action	promote to fast tier	schedule task

stages, with LLMs being used only in the first stage: (1) computing a *per-object score* for each candidate, and (2) applying a $\text{sort}(\cdot)$ operation to order these candidates in descending order of their score, and (3) selecting the topK objects from the sorted list. This decomposition significantly simplifies the LLM’s task by restricting it to generating a stateless scoring function, while all sorting and selection semantics are handled by reusable mechanisms. Beyond the benefits of simplification for LLMs (§2.3), this design is pragmatic: sort-and-select logic is largely invariant across tasks, making it natural to factor it out once as a common interface rather than repeatedly rediscovering it through search.

Table 3 summarizes how memory tiering and CPU scheduling would be represented in this interface. In memory tiering page promotion policies, for instance, all pages in lower tiers are candidates for promotion (*i.e.*, objects); X represents a set of global features (*i.e.*, not per-object) such as the bandwidth utilization and stall cycles. Using these global features, along with per object features (o), the function $\text{score}(X, o_i)$ computes a page hotness score, and then uses the topK hottest pages, promoting them to the higher tier (the action).

Creation of the policy module. To instantiate a policy module for RANK tasks, the LLM-generated scoring function $\text{score}(X, o)$ must be paired with a mechanism that selects the top-ranked objects. In this paper, we discuss and use three such mechanisms—FullSort, SampleSort, and PriorityQueue—and their decision-time and update costs.

FullSort evaluates and sorts all N objects at decision time, offering simplicity but with an $O(N \log N)$ latency that is acceptable for infrequent decisions (e.g., memory tiering) but too costly for high-frequency tasks (e.g., cache eviction). We use this mechanism in our instantiation of the page promotion policy in tiered memory systems (see §5). SampleSort reduces decision-time work by selecting a random subset of $S \ll N$ objects, scoring only this subset, and picking the top- K among them. This lowers complexity to $O(S \log S)$ while introducing an approximation – useful when exact global optimality is unnecessary, and latency is the primary driving factor. Such approaches have also been used in prior work to offset overheads [13, 16, 74, 95, 96]. PriorityQueue shifts work off the critical path by maintaining a globally ordered structure over all objects. Score updates happen incrementally: when an object’s metadata changes, its score is recomputed

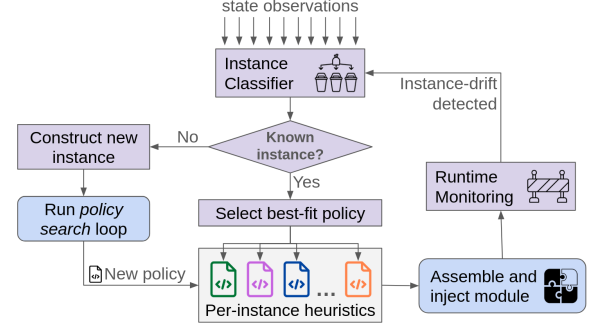


Figure 5: Automated instance generation and runtime policy selection with VULCAN.

and the structure is updated, incurring $O(\log N)$ (or worst-case $O(N)$) cost. At decision-time, selection requires only retrieving the top- K entries, providing $O(K \log N)$ complexity – making this approach suitable when decisions are frequent but updates occur outside the hot path. We find that this approach is fast enough to make the search for cache eviction policies tractable (see §4).

Together, these mechanisms span a spectrum of tradeoffs between selection latency and background maintenance, allowing the system designer to tailor the policy module to the demands of the task.

3.2 Defining an instance

A key design question for the users of VULCAN lies in defining *what an instance means* for their task. At a high level, an instance can be thought of as a specific pair of (workload, hardware) for which you are trying to search for new resource management heuristics. In some cases, it may be easy to delineate instances manually. For instance, in memory tiering page promotion policies, distinct applications running on the tiered memory system [4, 80, 97, 98] can be thought of as distinct instances – this is how we define instances in one of our case studies (§5). Similarly, recent work in LLM-driven search for GPU kernels [75, 107] defines every specific *hardware class* of GPUs (*e.g.*, Nvidia A100 vs H100) as an independent search problem (*i.e.*, an instance).

However, neat static boundaries such as this are oftentimes not possible: as we’ve seen in §2.1, for cache eviction, what people model as “one workload” class oftentimes has many sub-instances inside it. To address this, we demonstrate one way of building an automated instance generation process that can be used in conjunction with VULCAN, that we demonstrate in the other case study (§4).

3.2.1 Automated instance generation

Figure 5 shows an overview of our automated instance generation pipeline. The core piece of this system is the **instance classifier**, which is initialized with a set of *initial instances*, their distributional properties, and *instance-optimal policies* for each of these instances. At runtime, the instance classifier

uses signals from the current workload (*i.e.*, state measurements) to predict which distribution the current workload is the most similar to. If none of the instances are a good match, this classifier can collect data about the current instance and trigger an offline VULCAN policy search loop. Once the search is complete, if a new policy is found for this instance, it is added to the pool of available policies that the classifier chooses from.

We discuss a concrete implementation of this instance classifier for the cache eviction problem in §4.1.2. While a wide range of methods [7, 20, 69, 100, 118] have been proposed for these kinds of workload classification and runtime adaptation systems, we implement a simple classification system that uses unsupervised clustering.

3.3 Evolutionary search

With the policy formulation modified to meet the requirements of VULCAN, and the instance defined, users can now launch the evolutionary search component of VULCAN. VULCAN features an iterative, LLM-driven, evolutionary search process to find heuristics, summarized in Figure 6. At a high level, this evolutionary search algorithm maintains a population of candidate heuristics (for the `value(.)` or `rank(.)` functions; referred to combined as ‘scoring functions’). It then repeatedly generates more heuristics by learning from the existing ones, and uses a task-specific evaluator to score each variant on the instance of interest. Over successive generations, high-performing candidates are preferentially selected, allowing the search to gradually discover increasingly effective heuristics without requiring explicit supervision.

In this step, the user is responsible for setting up the search by providing a *template* and an *evaluation harness*. The template contains a natural language description of the requirements and objectives of the heuristic being designed and is consumed by a *generator* – a large language model. The generator uses this template along with example heuristics seeded by the operator, to synthesize code for a new *candidate heuristic*.

This process of generating new scoring functions, compiling them into a module, and evaluating their performance, is repeatedly performed, each time using a new set of samples from the increasing pool of heuristics in the database. The search loop terminates when a “good-enough” heuristic is found, or when new heuristic performance has plateaued.

Below, we describe the two main components of this step – the template and the evaluator.

3.3.1 The template

In addition to the natural-language description of the task and the available features (*i.e.*, inputs), the template contains a *prototype definition* of the scoring function, which the generator is prompted to implement in a code block while preserving the exact function signature.

Beyond the function prototype, the template can specify

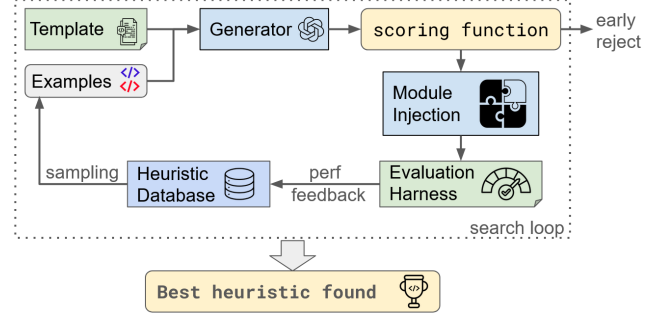


Figure 6: **Search process in VULCAN.** Green boxes (template, evaluation harness) are specified by users of VULCAN; blue boxes (generator, heuristic DB, module injection) are implemented by VULCAN.

additional constraints in natural language that the generated function must respect. These could include limits on memory usage or scratch space, mandates that certain inputs be treated as read-only, and task-specific semantics such as requiring that higher scores correspond to higher selection priority in RANK tasks. Many of these constraints can be checked via static analysis or enforced at runtime (e.g., using eBPF), allowing some invalid candidates to be rejected early in the search loop without explicitly evaluating them. Collectively, these rules bound the search space and provide the structural guidance needed to generate valid scoring functions

3.3.2 The evaluator harness

As shown in Figure 6, the harness receives a new candidate heuristic in the form of a *policy module* (described previously in §3.1.1 and §3.1.2) and evaluates it on the instance of interest.

During evaluation, any compilation or runtime errors act as early failures and the heuristic is discarded. For heuristics that run successfully, the evaluator collects a suite of performance metrics and derives a *single optimization metric* representing how well this new heuristic performed on the target instance – these performance scores guide the evolutionary search process in VULCAN. The heuristic and the corresponding performance measurements are then recorded as ‘performance feedback’ in a local *heuristic database*. The same approach can also target heuristics that try to optimize multiple objectives – here the metric reflects the heuristic developer’s preference on how to balance the objectives.

Designing an effective evaluation harness requires balancing two competing objectives: *efficiency* and *fidelity*. An efficient evaluator enables rapid iteration by cheaply scoring hundreds of candidate heuristics, while a high-fidelity evaluator more accurately reflects real-world deployment behavior. Table 4, which shows the different ways to implement evaluation harnesses, illustrates this tradeoff. Evaluators optimized for efficiency typically rely on simulators or emulators [5, 30, 45, 70, 101], enabling candidate policies to be evaluated quickly; in contrast, high-fidelity evaluators may

Table 4: Policy module implementations and evaluation harnesses across systems tasks.

Task	Policy module implementation	Evaluation harness...
Web caching (§4)	C++ function compiled directly into the code	... uses a simulator (libcachesim [45]) to measure <i>object hit rate</i> on a specific workload encoded in a set of traces (e.g., [103] or [68]).
Memory tiering (§5)	C++ function dynamically injected via LD_PRELOAD	... uses a CloudLab [26] node with remote NUMA memory to emulate CXL. We run a real workload on this system, and measure latency/throughput against for this workload.
Congestion control	eBPF probe attached dynamically at runtime	... uses the Mahimahi emulator [70] to measure latency and bandwidth on a benchmark of various network conditions [117]

require running full workloads on real hardware to capture realistic interactions, making them orders of magnitude more expensive.

In practice, users of VULCAN select a point in this design space based on available computational resources and the expected benefits from improved heuristics. Importantly, the VULCAN search loop may require evaluating dozens to hundreds of candidate heuristics before converging to a strong policy. Consequently, evaluation costs can accumulate rapidly: even modest increases in per-evaluation latency can translate into hours or days of additional search time – rendering the overall process impractical – especially when policies must be independently specialized for many distinct target instances. Users should therefore anticipate running hundreds of evaluations across tens to hundreds of instances, and design and select evaluation modules with their computational budget in mind.

4 Case study: Cache Eviction

Web caching systems – such as CDNs [72] – are widely deployed on the internet today to improve latency for end users [11, 15, 116] and minimize bandwidth usage [38, 95]. By storing popular content in limited-capacity *cache servers* closer to users, these systems improve latency, bandwidth and availability for end users. The fundamental resource management decision that caches need to make is to decide which objects should remain in cache and which should be evicted: given the limited amount of space available in cache servers, ensuring that only objects likely to be reused remain cached is essential for performance.

Researchers over the last two decades have developed a wide range of heuristics [23, 24, 28, 42–44, 62, 73, 115, 120] and ML-based methods [13, 85, 95, 102, 113] to tackle this challenge of deciding which objects to evict. These policies use features such as recency of accesses [43, 62], frequency [23, 28, 62], sizes of objects, “ghost lists” containing items that were recently evicted from cache [43, 62, 115], and object inter-arrival times [43, 73], among others, to make these decisions. As we have seen in §2.4 and from Figure 1, even within a single workload class, there is no “one-size-fits-all” heuristic, motivating the need for identifying instance-optimal heuristics with VULCAN.

Table 5: Data provided to `score(·)` by our scaffolding.

Type of feature	Attributes
Per-object (f_{o_i})	Number of accesses (count), last access time, time added to cache, object size
Global (X)	Percentiles over access counts, ages, or sizes (e.g., p50 size in bytes of all objects in cache). List of recently evicted objects, along with (timestamp, access count, age) at eviction.

4.1 Searching for new eviction heuristics

Following the workflow discussed earlier in Figure 2, we first discuss how to formulate the problem in VULCAN’s common interfaces and build the scaffolding (§4.1.1). Then, we define what an *instance* means in this use-case (§4.1.2). Finally, we instantiate a policy search (§4.1.3) and discuss the results (§4.1.4) on how these new heuristics performed.

4.1.1 Interface and scaffolding

The core decision being made in cache eviction is to *select* an eviction candidate out of pool of objects – this naturally fits the semantics of the RANK interface (§3.1.2). For this case study, we use a `PriorityQueue` to implement the RANK operation.

In this interface, all objects in the cache are in a *logical* priority queue, where an object’s position reflects its likelihood of eviction; objects near the top (i.e., front) of the priority queue are likelier to be evicted. Positions in the queue are updated using a `score(·)` function every time they are accessed. Each object in this priority queue is also annotated with metadata collected by our scaffolding (Table 5), allowing `score(·)` to incorporate per-object counters, access history, and information about recently evicted objects – features commonly used across a wide range of eviction policies.

When an eviction decision needs to be made, our scaffolding merely looks at the topK elements in this priority queue and evicts them – a simple mechanism that doesn’t even invoke the `score(·)` function at decision time since we have a fully sorted list at all times. Using a priority queue, however, introduces an $O(\log N)$ overhead every time an object is accessed – this is because the object needs to be rescored and reinserted into the priority queue. We adopt this structure

because it provides a clean upper bound on the performance envelope for approaches in this design space, while still remaining far more efficient than the computational cost of many ML-based eviction strategies (*e.g.*, [95]). The mechanism itself is not fundamental: it could easily be replaced by SampleSort, which offers substantially lower overhead at the cost of some degradation in ranking accuracy.

4.1.2 An instance generator for caching

We now build an *instance generator* (§3.2.1) to delineate the boundaries of instances for the cache eviction problem. For this case study, we use the CloudPhysics dataset [103] which consists of 106 block I/O traces collected over a span of a week. From each trace, we collect the first 50,000 requests (representing <1% of total requests in the trace even for the smallest trace) and extract fifteen *features* from each trace. These features include information such as the number of unique objects, percentage of one hit wonders, average/max object sizes, and interarrival times. Using these features, we cluster the 106 traces in this dataset into 10 clusters using KMeans. The sizes and characteristics of these ten clusters (C_0 to C_9) are summarized in Table 6 and visualized using t-SNE [58] in Figure 7. In this case study, each of these clusters is an *instance*.

Table 6: **Summary statistics for the ten clusters in the CloudPhysics dataset.** *Footprint (obj)* is the number of unique objects seen in this cluster; *Footprint (GB)* is the sum of sizes of those unique objects; *total requests* is the total number of object accesses.

Metric	C_0	C_1	C_2	C_3	C_4
Footprint (obj)	274M	16M	4M	92M	17M
Footprint (GB)	5751	327	82	8192	1646
Total requests	982M	41M	5M	351M	100M
Metric	C_5	C_6	C_7	C_8	C_9
Footprint (obj)	48M	26M	10M	0.4M	5M
Footprint (GB)	3740	1336	126	8	1241
Total requests	235M	180M	164M	45M	12M

4.1.3 Policy search for cache eviction

Now that we have implemented the scaffolding and defined the notion of an instance for this task, we move on to the final steps before beginning the search: *writing a template* and *implementing an evaluation harness*.

The template for this use case includes an overview of the task (eviction heuristics for web caches), an explanation of all the available features that can be used to compute the score (Table 5), and ends with a prompt asking the LLM to “think” and design a new scoring function. The prompt also includes a function prototype for score, an explanation that objects with a higher score are likelier to be evicted and an example `score(.)` function that implements LRU. The full template used for this case study, along with one of the heuristics dis-

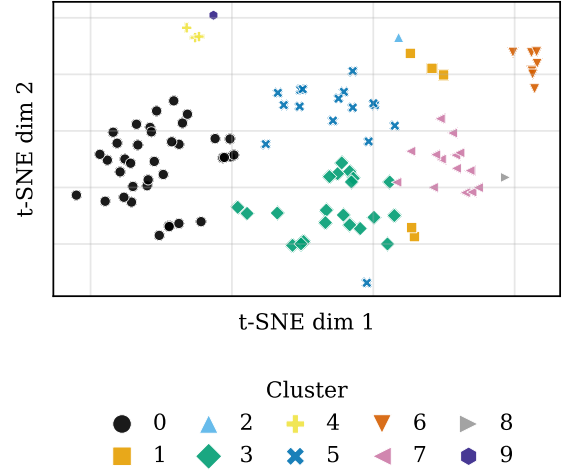


Figure 7: **t-SNE visualization of the CloudPhysics dataset**, with points colored by cluster assignment (0 – 9).

covered through the search, is provided in Appendix B.

For our evaluator harness, we opted to use a simulator – lib-cachesim [45] – which runs the newly generated heuristic and measures the *object hit rate* of the policy. During the search phase, the evaluator harness uses one trace from within the cluster to score candidate solutions – the heuristic identified at the end of the search is then evaluated on all traces within this cluster.

Implementing the evolutionary search loop. We use the gpt-4o-mini model via the OpenAI API to synthesize candidate solutions for our evolutionary search loop. The initial round is seeded with an example of a simple baseline heuristic (LRU, in this case study). In each round, the generator produces 25 candidate heuristics that are evaluated using the *evaluation harness* described above. The top two performing heuristics across all rounds are retained and used as examples in the next round, replacing earlier examples and progressively biasing generation toward higher-performing regions of the search space. This process is repeated for a fixed number of rounds, broadly following prior work on evolutionary search for code synthesis [71, 86].³

4.1.4 Results

We use thirteen eviction algorithms as our baselines: GDSF [23], S3-FIFO [115], SIEVE [120], LIRS [43], LHD [13], Cacheus [85], FIFO-reinsertion [24], LeCaR [102], SR-LRU [85], CR-LFU [85], LRU, MRU, and FIFO. For each caching heuristic, we compute the *miss rate reduction* (MRR) over FIFO, averaged over all traces in a cluster, and use that as the metric to compare performance – this is similar to how [120] and [115] report their results.

³ Code used to implement this evolutionary search loop is available in [27]. For the other case studies in the paper (§4.2, §5), we use OpenEvolve [90] to implement the search loop. In practice, any of these frameworks [27, 54, 90] can be used to implement the search phase of VULCAN.

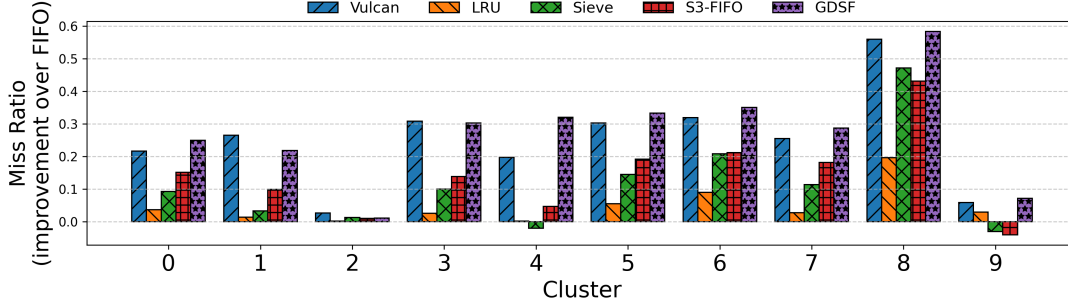


Figure 8: Performance of VULCAN-synthesized cache eviction heuristics compared against baselines, per cluster ($C_0 - C_9$). Higher is better.

We instantiated a VULCAN search on each of the clusters (*i.e.*, instances) independently, and discovered a specialized heuristic for each of the clusters. Figure 8 shows the performance (*i.e.* MRR) of these specialized heuristics compared against some representative baselines, per-cluster. In clusters C_1 , C_2 , and C_3 , the VULCAN-discovered heuristic achieves the best overall performance, exceeding the strongest baseline by 21.4%, 69%, and 1.94%, respectively. In clusters C_5 , C_6 , C_8 , and C_9 , the synthesized heuristic ranks second, trailing only GDSF. In the remaining clusters, it consistently ranks third, outperforming eleven out of thirteen baseline heuristics. Overall, these results show that VULCAN either outperforms or closely matches state-of-the-art eviction policies across all clusters.

4.2 Efficiency by design: Queue Topology

In this case study so far, we formulated the cache eviction problem as a RANK-style task, and then used a “learned” scoring function to rank and select eviction candidates from a priority queue. While expressive, this approach incurs non-trivial overhead: maintaining a priority queue requires $O(\log N)$ work on every object update; even lower-overhead alternatives such as the SampleSort mechanism (§3.1.2) must repeatedly sample, score, and partially sort objects at each eviction decision.

In contrast, recent work – most notably S3-FIFO [115] – demonstrates that highly efficient eviction policies can be built from extremely simple primitives (FIFO queues), and that such efficiency is critical for a wide range of real-world cache deployments operating at scale. Broadly, we observe that a large number of caching algorithms over the years – *e.g.*, 2Q [44], ARC [62], LIRS [43] and many others [48, 73, 124] – are fundamentally queue-based: they compose a small number of base primitives (FIFO, LRU, stack) and define rules for moving objects between them. Many other policies [24, 42, 120] simply apply minor structural tweaks to these base primitives.

Motivated by this recurrence, we ask whether it is possible to systematically search this space of *queue-based heuristics* to discover new, instance-specialized eviction policies that are **efficient by design**. We call this search space the *queue*

topology class, which we define in the next section.

4.2.1 Definition of a Queue Topology cache

A *queue-topology (QT) cache* consists of a fixed set of M FIFO or LRU queues ($M \leq 5$), together with rules that govern how objects transition between these queues over time. The total cache capacity is L objects, which is partitioned across these M queues. At any time, each object present in the cache resides in exactly one of these queues.

In addition, the QT cache may also include a *ghost queue* \mathcal{G} , implemented as a FIFO of capacity at most L , storing *meta-data* of recently evicted objects. Formally, a queue topology cache is specified by defining the following components:

- **Queue configurations.** The set of queues $Q = \{q_0, \dots, q_{M-1}\}$, the type of each queue (FIFO or LRU), and the size of each queue (such that the total capacity sums to L), the capacity of the ghost queue (at most L).
- **Per-object features.** Each object in the cache is associated with a fixed feature vector whose components are defined by the policy designer (*e.g.*, access counts or recency measures). Feature values are updated on cache events, persist across queue transitions (including placement in the ghost queue), and constitute the sole inputs to the placement and transition rules.
- **Initial placement function.** A function that determines which queue a newly inserted object is placed into:

$$f_{\text{init}}(\text{new_obj}) \rightarrow q_{\text{init}} \in Q.$$

If the selected queue is full, inserting the object evicts the object at the tail of that queue.

- **Transition functions.** For each queue $q \in Q$, a transition function that determines how an object evicted from the tail of that queue is handled:

$$f_{\text{trans}}^{(q)}(\text{obj}) \rightarrow q' \in Q \cup \{\mathcal{G}, \mathcal{T}\}.$$

This function may decide to move the object to another queue ($q' \in Q$), evict the object while storing its metadata in the ghost queue \mathcal{G} , or bypass the ghost queue and directly delete (*i.e.*, *trash*) both the object and its metadata directly (\mathcal{T}).

All of the queue-based algorithms discussed at the beginning of §4.2 can be viewed as individual points within the space of

heuristics defined by this queue-topology formulation. In the next section, we describe how to systematically explore this space to identify new, instance-specialized eviction policies.

4.2.2 Searching the Queue Topology Space

Following the three-step pipeline discussed earlier (Figure 2), we formulate the Queue Topology cache search in VULCAN.

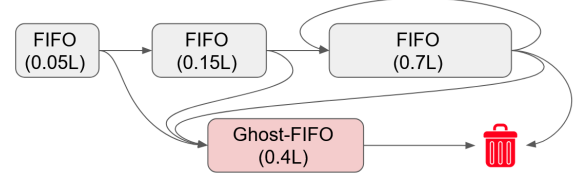
Interface. The initial placement function f_{init} and the transition functions $f_{\text{trans}}^{(q)}$ of a queue-topology cache naturally fit the VALUE interface, as each function takes as input a set of features describing the object in question and outputs a scalar routing decision. In this setting, the scalar routing decision corresponds to a destination resident queue index in $\{0, \dots, M-1\}$; for transition functions, it may additionally specify a move to the ghost queue (-1) or direct deletion from the cache (-2). A queue topology is fully specified only by the joint definition of the initial placement function and all transition functions. Accordingly, the LLM must co-design these interacting VALUE functions simultaneously, which is precisely the approach we take here.

For the initial placement function, the inputs (i.e. X_{init}) are: (i) a boolean indicator of whether the object is currently in the ghost queue, and (ii) boolean indicators for each of the M resident queues, indicating if they are currently full. For the transition functions, we provide features (X_{trans}) capturing the object’s access history and temporal behavior, including global access counts (since insertion into the cache), per-queue access counts (since insertion into the current queue), insertion times at the cache and queue levels, and the time of the most recent access.

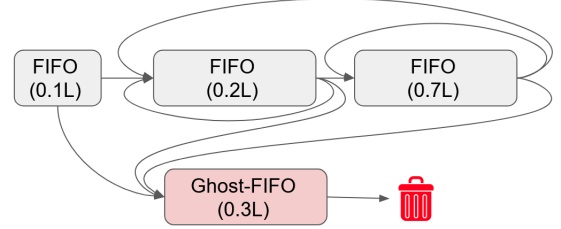
Scaffolding. Following the policy–mechanism separation described in §3.1, we isolate all cache mechanics from the decision-making logic. The scaffolding, implemented on top of libCacheSim [45], is responsible for instantiating the queues that are a part of the topology (FIFO/LRU), enforcing queue capacities, maintaining per-object metadata, and executing insertions, evictions, and queue transitions. The only components synthesized by VULCAN are the initial placement function and one transition function per queue; all routing decisions produced by these functions are then implemented by the fixed scaffolding.

Instance. We use the same instance definition methodology as in §4.1.2, where instances are clusters of traces with similar characteristics. For this case study, we specialize heuristics for two instances: C_7 and C_8 .

Launching the search. We implement the evolutionary search using OpenEvolve [90] and use gpt-4o-mini as the underlying language model. The search is run for 100 iterations, and candidate topologies are evaluated using *object hit rate* as the optimization objective.



(a) Topology discovered for C_7



(b) Topology discovered for C_8

Figure 9: Two cache queue topologies discovered by VULCAN. Each topology partitions a cache of capacity L objects across three FIFO queues. Directed edges indicate possible object transitions between queues: the exact route or edge that a specific object takes is conditioned on object-specific features.

4.2.3 Experimental Setup

Similar to the results for the RANK-based cache eviction search (§4.1.4), we evaluate heuristics synthesized by VULCAN against a suite of baseline caching policies on the CloudPhysics dataset (*per-instance*).

CloudPhysics traces include both object access times and object sizes. In §4.1.4, we use both signals: cache capacity is measured in bytes, and eviction decisions account for variable object sizes, making size-aware policies such as GDSF [23] and LHD [13] the strongest baselines. In contrast, the queue-topology experiments in this subsection use the same access traces but ignore object sizes, treating all objects as occupying a single cache slot and measuring cache capacity in number of objects.

We adopt this size-agnostic setting to fairly compare against state-of-the-art multi-queue caching literature (e.g., S3-FIFO [115], SIEVE [120]), which targets slab-based allocators where eviction decisions are restricted to classes of similarly sized objects.

Under this size-agnostic setup, we compare the VULCAN-synthesized heuristics against seventeen baselines. This set includes all baselines evaluated in §4.1.4, along with four additional queue-based baselines: ClockPro [42], ARC [62], QDLP [114], and TwoQ [44].

4.2.4 Results

We use VULCAN to search for eviction heuristics within the *queue-topology* design space for clusters C_7 and C_8 . The topologies discovered by the search are visualized in Figure 9. Source code for the VULCAN synthesized topology for C_7 is available in Appendix C.

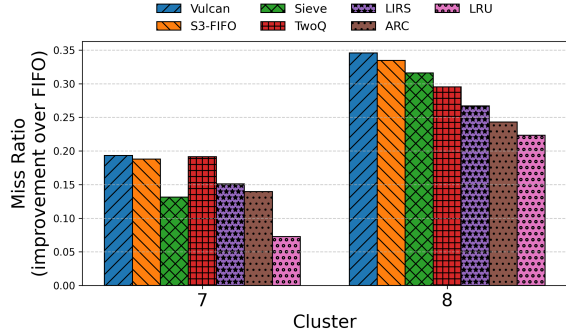


Figure 10: Performance of VULCAN-synthesized Queue Topology heuristics compared against queue based baselines on clusters C_7 and C_8 .

In both clusters C_7 and C_8 , the heuristic synthesized by VULCAN outperforms all seventeen baselines considered. Specifically, VULCAN improves over the strongest baseline by 1.0% on C_7 (TwoQ) and by 3.2% on C_8 (S3-FIFO). Figure 10 compares the performance of the VULCAN-synthesized heuristics against a selection of queue-based baselines.

By reformulating cache eviction as a search over queue topologies, we demonstrate that VULCAN can synthesize instance-specialized policies that are *efficient by design*. All heuristics in this interface execute using only constant-time queue operations, yet the discovered policies outperform strong, hand-designed multi-queue baselines. This result highlights a key strength of VULCAN: by choosing an interface that embeds structural efficiency, we can systematically explore rich policy spaces while preserving the performance guarantees required in high-throughput cache deployments.

5 Case study: Memory Tiering

To meet the memory demands of modern workloads, memory tiering expands effective memory capacity by incorporating slower tiers such as CXL-attached memory or NVM alongside local DRAM. Unlike traditional caches, data resides in exactly one tier and is directly accessible from any tier. Because these additional tiers have higher latency and lower bandwidth, careless data placement can degrade performance when frequently accessed data remains in slow memory. Tiering systems, therefore, aim to keep frequently accessed *hot* data in fast DRAM and demote infrequently accessed *cold* data to slower tiers.

Current heuristics. Several systems have been developed to transparently place and migrate memory pages in tiered memory systems [55, 61, 84, 109]. These systems gather application memory accesses via hardware event sampling such as Intel PEBS [39] or through page table Accessed (A) bit scanning [78]. They then employ heuristics based on access frequency and recency to classify pages and make page migration decisions. We briefly describe two state-of-the-art

systems below.

Memtis [55] uses Intel PEBS to track per-page access counts. It relies on a *hotness threshold* to identify hot and cold pages. If a page receives more than `hot_threshold` samples, it is deemed hot and promoted to the fast tier. In order to adapt to workload phase changes, Memtis periodically “cools” (decays) page access counts. If a page access count drops below the `hot_threshold`, the page is considered cold and demoted to the slow tier. Crucially, Memtis attempts to adapt to different workloads by adjusting the `hot_threshold` based on the observed workload global access distribution.

ARMS [109], a more recent tiering system, eliminates the absolute `hot_threshold` in favor of relative page scoring. ARMS maintains multi-timescale moving averages of the page access counts, capturing both historical and recent behavior. These averages are combined into per-page hotness scores, which are used to rank pages. Periodically, ARMS ranks pages by this score and migrates the top-ranked pages to the fast tier and the bottom-ranked pages to the slow tier.

Need for instance-optimality. While heuristics used in systems like Memtis [55] and ARMS [109] demonstrate success in general scenarios, they are not inherently instance-optimal. For example, prior work finds that frequency-based systems such as Memtis and ARMS work well for workloads with a Zipfian access distribution but perform sub-optimally for workloads with latest access distribution, such as YCSB-D or TPC-C [47]. The primary limitation of these state-of-the-art approaches is their reliance on fixed underlying policies, even when they include dynamic parameters.

For instance, although Memtis dynamically adjusts its `hot_threshold`, the adjustment algorithm itself is static and assumes specific distribution properties. As a result, Memtis might fail to converge on a stable threshold or react slowly to workload changes, leading to mis-classification of pages [47]. Similarly, ARMS uses a fixed scoring function to balance recent and historical accesses. If this function’s weights are not tuned for the specific workload instance — for example, weighing recent access too heavily — the system may fail to identify truly hot data in sparse access patterns.

5.1 Memory Tiering Policies with VULCAN

5.1.1 Interface and scaffolding

As discussed in §3.1, users express their problem using either the VALUE or RANK interface. Memory tiering admits both views: Memtis’ `hot_threshold` corresponds to a VALUE interface, while ARMS hotness-score ranking naturally fits a RANK interface. We describe how these interfaces can be instantiated for tiering and why we ultimately favor the latter for evolutionary search.

With a VALUE-style framing (as in Memtis), one could compute a dynamic threshold based on various statistics, then compare each page’s access count against it. However, ARMS argues that any global threshold – no matter how well tuned –

Table 7: Evaluator objectives for each application instance.

App	Parameters	Metrics of Interest	Objective
GUPS	hotset size, array size	Updates per second (U) Completion time (T)	max U/T
GapBS (BC/PR)	num threads	Completion time (T)	min T
Silo (TPCC)	num ops	Goodput (G) Average latency (L)	max G

inevitably misclassifies some hot pages or triggers wasted migrations. Threshold-based mechanisms thus lack the expressiveness needed to capture richer policies. Thus, we instead use the ARMS mechanism, as explained below.

ARMS consists of four components: (i) page-access tracking, (ii) hotness-score computation, (iii) hotset-change detection, and (iv) migration mechanism. ‘Data-collection’ logic lies in (i), ‘policy’ lies in the (ii)-(iii), and the ‘action’ lies in (iv) (see §3.1). In particular, the hotness-score computation is an instance of the RANK-interface, with the migration mechanism implementing the FullSort sorting mechanism (see §3.1.2). We replace the policy components with a single policy module, while leaving the action component untouched and modifying data collection only minimally.

Vanilla ARMS only tracks access counts over a single window (500 ms) and aggregates them into short- and long-range averages. To give evolutionary search the ability to explore alternative temporal features, VULCAN maintains a longer access history: each page stores a 20-window circular buffer (10 s) of access counts. These richer per-page histories, along with global signals such as DRAM and NVM bandwidth usage, form the feature set exposed to VULCAN’s evolutionary search.

5.1.2 Defining the instance

In the context of memory tiering, an instance is defined by the specific combination of an application (characterized by its access pattern) and the underlying hardware characteristics, such as the latency/bandwidth difference between tiers. In practice, it is easy to specify instances by selecting the workload and platform of interest. We evaluate VULCAN on four such instances: GUPS (a memory microbenchmark), GapBS Betweenness Centrality and PageRank (graph analytics), and Silo TPCC (an in-memory database).

5.1.3 Policy search for tiering policies

Building the evaluator. Different applications prioritize different metrics: GUPS targets update rate and runtime, GapBS focuses on end-to-end completion time, and Silo measures goodput and average latency. We define a specific evaluator for each instance (summarized in Table 7), supplying each of these metrics as feedback and constructing a simple optimization objective. Notably, the objective can combine multi-dimensional performance metrics; for example, in Silo, the ob-

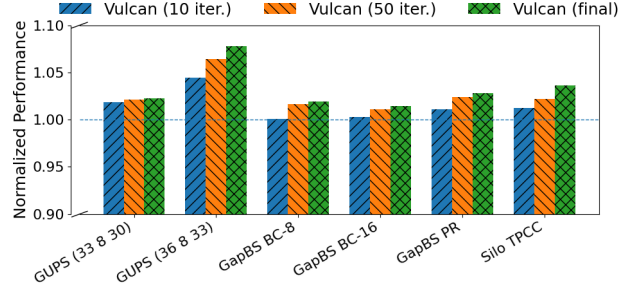


Figure 11: **Improvement in VULCAN memory tiering policies.** Performance improvement normalized against metrics achieved for the same workloads with the vanilla ARMS policy, for the best heuristic after 10 iterations of VULCAN’s search process versus the final best heuristic.

jective function simultaneously maximizes throughput while minimizing the tail latency.

5.2 Results

We instantiate the VULCAN search for different applications given in Table 7, and run the search loop 150 times for each instance. We use the OpenEvolve [90] implementation for the evolutionary search, with a combination of GPT-5.1-mini and GPT-4 models. We seed the search with a very simple implementation of the rank(.) function, that simply ranks pages based on the sum total of accesses across all 20 windows.

Overall improvements. For each execution, we collect the metrics of interest and show the performance improvement normalized against vanilla ARMS policy in Figure 11. Specifically, for applications that are latency-sensitive (such as GapBS workloads), we plot $\text{runtime_arms} / \text{runtime_vulcan}$, while for throughput-oriented applications (such as GUPS and Silo), we plot $\text{tput_vulcan} / \text{tput_arms}$. We find that across *all* applications, we see a 2.5-7.9% improvement in the performance, with GapBS (BC) seeing 2.5% and GUPS seeing a 7.9% improvement.

Further, these improvements come at a shallow cost: the total cost of executing the 150 iterations across all these applications was around \$37 USD. In fact, for several workloads, just 50 and in some cases, even 10 iterations are enough to find a heuristic close enough to the best-performing one (see Figure 11) – driving the cost of discovery down further.

Deep-dive into the generated heuristics. We find that VULCAN can discover interesting ways of fine-tuning heuristics. Often times, these heuristics exceed 150 lines of code – showcasing that VULCAN’s evolutionary search can find complex, yet interpretable heuristics. Even though we started the evolutionary search with a simple aggregate over all 20 windows, all VULCAN-generated heuristics use some form of weighted averages over the windows, or some notion of recency – something that ARMS also utilizes, but uses the same constants for all workloads.

Further, we noticed that VULCAN can identify workload characteristics and specialize the heuristic for the workload at hand. For instance, the VULCAN-discovered heuristic for GUPS adds a penalty for very high NVM bandwidth usage and NVM writes (see Listing 1). On the other hand, the VULCAN heuristic for Silo instead scores recent windows highly, and introduces a *phase-transition* based scoring mechanism – where, if a page is seeing a bursty access pattern (and hence, the variance between the min and max access counts over the last 20 windows is high) the score is penalized (see Listing 2). This is especially important for a database workload like Silo, where transactions can cause spurious bursty accesses.

Listing 1: Bandwidth-aware penalties to avoid NVM and DRAM saturation. `g_stats` is a data-structure storing global statistics

```
/* Add global NVM bandwidth penalty factor:
 * penalize moving hot pages if NVM is saturated */
float nvm_bw_penalty = 1.0f;
float dram_bw_penalty = 1.0f;
float nvm_std_penalty = 1.0f;

if (g_stats) {
    if (g_stats->nvm_bw_ewma > 0.92 * NVM_RD_BW_KNEE) {
        nvm_bw_penalty = 0.55f;
    } else if (g_stats->nvm_bw_ewma > 0.75 * NVM_RD_BW_KNEE) {
        nvm_bw_penalty = 0.8f;
    } else if (g_stats->nvm_bw_ewma > 0.6 * NVM_RD_BW_KNEE) {
        nvm_bw_penalty = 0.92f;
    }
    if (g_stats->dram_bw_ewma > 0.97 * DRAM_BW_KNEE) {
        dram_bw_penalty = 0.7f;
    }
    if (g_stats->nvm_bw_std > 2.5) {
        nvm_std_penalty = 0.85f;
    }
}
```

Listing 2: Phase-aware penalty to avoid promotions during bursty access patterns. `min_total` and `max_total` track the minimum and maximum accesses in a window (total implies it is a sum of DRAM and NVM reads/writes).

```
/* Novelty: capture recent phase transitions
 * (spikes or drops in total access) */
float phase_penalty = 1.0f;

if (max_total > 8.0f &&
    min_total / (max_total + 1e-3f) < 0.18f) {
    /* Very bursty phase: do not promote */
    phase_penalty = 0.65f;
} else if (max_total > 8.0f &&
    min_total / (max_total + 1e-3f) < 0.35f) {
    /* Moderately bursty phase */
    phase_penalty = 0.8f;
}
```

6 Related Work

Learning-based specialization. ML-based approaches [95, 113, 123]—especially neural policies for systems tasks [3, 53]—have been proposed as a way to automatically specialize policies to workload structure. These systems demonstrate that data-driven models can outperform fixed heuristics in domains such as caching and congestion control. However, neural approaches carry well-known costs: opaque behavior that complicates debugging and validation [63]; complex training, tuning, and deployment pipelines [6, 29]; substantial inference overheads in the control path [29, 119]; and safety concerns that limit adoption in production environments [88]. VULCAN takes a different stance: it avoids neural inference in the hot path entirely and confines learning to an offline search over small, interpretable code snippets written against narrow interfaces.

AI-driven algorithm and heuristic design. Recent work has explored the use of LLMs to generate new systems heuristics. One line of work modifies existing human-designed algorithms incrementally, such as automatic BBR variants [36] or the ADRS-style search explored by *Barbarians* [22], which demonstrates that LLM-guided mutation and evaluation can produce heuristics competitive with or superior to expert-crafted designs. A complementary line focuses on search strategies and prompting structure rather than the algorithms themselves; for example, Glia [35] employs multiple specialized LLM agents and Robusta [49] uses counterexamples to harden candidate heuristics. These approaches are orthogonal to VULCAN and can be layered atop it. What they do not provide is a principled articulation of *what* the search should target. VULCAN contributes that missing piece by defining VALUE and RANK as minimal, mechanism-agnostic interfaces that structure the design space, shrink the search surface, and allow mechanisms (e.g., FullSort versus PriorityQueue scheduling) to be cleanly separated from policy logic.

Systems use of LLMs. Several projects explore placing LLMs or other large models directly in the systems hot path [46, 121], or as kernel-level assistants [6, 29]. Such approaches are powerful but face stringent latency, safety, and predictability constraints. VULCAN is deliberately conservative: LLMs participate only in the offline loop that generates candidate policy code. At runtime, the system executes simple, deterministic heuristics identical in structure to those used in traditional systems, ensuring predictable overheads and integration safety.

7 Discussion and Conclusion

ML-for-systems work often attributes its gains to access to richer data or higher-quality features than those available to traditional heuristics. While additional visibility can help, the intellectually difficult part is rarely the modeling itself. Instead, it is deciding which signals matter, how they should be represented, and how they interact with underlying mech-

anisms. As a result, humans remain responsible for feature selection, data engineering, and reconciling policy logic with system constraints. This suggests that meaningful progress hinges less on increasingly complex models and more on principled abstractions that clarify what a policy should compute and how it should interface with the system.

VULCAN embodies this shift in perspective. By reducing heuristic design to narrow, mechanism-agnostic interfaces, it enables automated search over compact, interpretable policies while preserving the safety and predictability required in performance-critical settings. Our evaluation shows that synthesized heuristics for caching and memory tiering consistently match or outperform expert-crafted designs, delivering instance-specific benefits without neural inference, large feature sets, or heavyweight training pipelines. These results illustrate that focusing on the structure of decisions – not the sophistication of learning machinery – can make automated specialization both practical and effective, and point toward broader opportunities for principled, search-driven systems design.

References

- [1] Horizontal pod autoscale. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Accessed: Dec 2025.
- [2] Soheil Abbasloo, Yang Xu, and H Jonathan Chao. C2tcp: A flexible cellular tcp to meet stringent delay requirements. *IEEE Journal on Selected Areas in Communications*, 37(4):918–932, 2019.
- [3] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. 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*, SIGCOMM ’20, page 632–647, New York, NY, USA, 2020. Association for Computing Machinery.
- [4] Reto Achermann and Ashish Panwar. Mitosis workload btree, 2019.
- [5] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for llm inference. *Proceedings of Machine Learning and Systems*, 6:351–366, 2024.
- [6] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. Kmlib: Towards machine learning for operating systems. In *Proceedings of the On-Device Intelligence Workshop, co-located with the MLSys Conference*, pages 1–6, 2020.
- [7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.
- [8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM ’10*, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.
- [9] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference, SIGCOMM ’22*, page 177–192, New York, NY, USA, 2022. Association for Computing Machinery.
- [10] Venkat Arun and Hari Balakrishnan. Copa: Practical {Delay-Based} congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, Renton, WA, 2018. USENIX Association.
- [11] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. 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*, pages 495–513, 2020.
- [12] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [13] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, 2018. USENIX Association.
- [14] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.
- [15] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. {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)*, pages 195–212, Carlsbad, CA, 2018. USENIX Association.

- [16] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.
- [17] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 369–378, Santa Clara, CA, July 2024. USENIX Association.
- [18] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Communications of the ACM*, 60(2):58–66, 2017.
- [19] Angelica Chen, David Dohan, and David So. Evoprompting: Language models for code-level neural architecture search. *Advances in neural information processing systems*, 36:7787–7817, 2023.
- [20] Jiayi Chen, Nihal Sharma, Tarannum Khan, Shu Liu, Brian Chang, Aditya Akella, Sanjay Shakkottai, and Ramesh K Sitaraman. Darwin: Flexible learning-based cdn caching. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM ’23, page 981–999, New York, NY, USA, 2023. Association for Computing Machinery.
- [21] 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 Hebbgen 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. Evaluating large language models trained on code, 2021.
- [22] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, et al. Barbarians at the gate: How ai is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025.
- [23] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, Palo Alto, CA, USA, 1998.
- [24] Fernando J Corbato. *A paging experiment with the multics system*. Massachusetts Institute of Technology, Cambridge, MA, 1968.
- [25] Jonathan Corbet. Let them run cake, Jun 2018.
- [26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.
- [27] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. Man-made heuristics are dead. long live code generators! In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, pages 51–60, 2025.
- [28] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.
- [29] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 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*, ASPLOS 2023, page 846–861, New York, NY, USA, 2023. Association for Computing Machinery.
- [30] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.
- [31] P. Goyal, H.M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, 1997.
- [32] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, November 2016. USENIX Association.

- [33] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and Dependency-Aware scheduling for Data-Parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.
- [34] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.
- [35] Pouya Hamadanian, Pantea Karimi, Arash Nasr-Esfahany, Kimia Noorbakhsh, Joseph Chandler, Ali ParandehGheibi, Mohammad Alizadeh, and Hari Balakrishnan. Glia: A human-inspired ai for automated systems design and optimization. *arXiv preprint arXiv:2510.27176*, 2025.
- [36] Zhiyuan He, Aashish Gottipati, Lili Qiu, Yuqing Yang, and Francis Y Yan. 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. 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. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.
- [39] Intel Corporation. Intel 64 and ia-32 architectures software developer manuals. 2018.
- [40] Amazon Artificial General Intelligence. Amazon nova premier: Technical report and model card. 2025.
- [41] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- [42] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.
- [43] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.
- [44] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB ’94*, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [45] Juncheng Yang (1a1a11a). libcachesim: a high performance library for building cache simulators. <https://github.com/1a1a11a/libCacheSim>, 2023. Accessed: 2025-06-28.
- [46] Aditya K Kamath and Sujay Yadalam. Herding llamas: Using llms as an os module. *arXiv preprint arXiv:2401.08908*, 2024.
- [47] Konstantinos Kanellis, Sujay Yadalam, Shivaram Venkataraman, and Michael Swift. Striking the right chord: Parameter tuning in memory tiering systems. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems*, pages 1–9, 2025.
- [48] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.
- [49] Pantea Karimi, Dany Rouhana, Pooria Namyar, Siva Kesava Reddy Kakarla, Venkat Arun, and Behnaz Arzani. Robust heuristic algorithm design with llms. *arXiv preprint arXiv:2510.08755*, 2025.
- [50] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. Crust-bench: A comprehensive benchmark for c-to-safe-rust transpilation. *arXiv preprint arXiv:2504.15254*, 2025.
- [51] Gautam Kumar, Nandita Dukkupati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 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, SIGCOMM ’20*, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.
- [52] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shvinnath Babu. Robus: fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 219–234, New York, NY, USA, 2017. Association for Computing Machinery.

- [53] 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. Heimdall: Optimizing storage i/o admission with extensive machine learning pipeline. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25*, page 1109–1125, New York, NY, USA, 2025. Association for Computing Machinery.
- [54] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.
- [55] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.
- [56] 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. Lava: Lifetime-aware vm allocation with learned distributions and adaptation to mispredictions, 2025.
- [57] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.
- [58] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.
- [59] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. CBMM: Financial advice for kernel memory managers. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 593–608, Carlsbad, CA, July 2022. USENIX Association.
- [60] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery.
- [61] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanauja, and Prakash Chauhan. 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*, 2023.
- [62] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, USA, 2003. USENIX Association.
- [63] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. 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, SIGCOMM '20*, page 154–171, New York, NY, USA, 2020. Association for Computing Machinery.
- [64] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480. IEEE, 2016.
- [65] Radhika Mittal, Vinh The Lam, Nandita Dukkhipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.
- [66] Prasita Mukherjee and Benjamin Delaware. Towards automated verification of llm-synthesized c programs, 2025.
- [67] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.
- [68] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.
- [69] Usama Naseer and Theophilus A. Benson. Configanator: A data-driven approach to improving CDN performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1135–1158, Renton, WA, April 2022. USENIX Association.

- [70] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, Santa Clara, CA, USA, 2015. USENIX Association.
- [71] Alexander Novikov, Ngân Vù, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025.
- [72] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.
- [73] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, 22(2):297–306, June 1993.
- [74] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 69–84, 2013.
- [75] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.
- [76] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.
- [77] Lorenzo Pappone, Alessio Sacco, and Flavio Esposito. Mutant: Learning congestion control from existing protocols via online reinforcement learning. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1507–1522, 2025.
- [78] SeongJae Park, Yunjae Lee, and Heon Y Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, 2019.
- [79] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, 2001.
- [80] Steven J Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood, and Mike Davis. A simple synchronous distributed-memory algorithm for the hpcc randomaccess benchmark. In *2006 IEEE International Conference on Cluster Computing*, pages 1–7. IEEE, 2006.
- [81] Sriram Ramabhadran and Joseph Pasquale. 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*, pages 239–250, New York, NY, USA, 2003. Association for Computing Machinery.
- [82] Nikitha Rao, Elizabeth Gilbert, Harrison Green, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. Diffspec: Differential testing with llms using natural language specifications and code artifacts. *arXiv preprint arXiv:2410.04249*, 2024.
- [83] 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. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.
- [84] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.
- [85] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with {CACHEUS}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, 2021.
- [86] 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. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.
- [87] Sultan Mahmud Sajal, Luke Marshall, Beibin Li, Shandan Zhou, Abhisek Pan, Konstantina Mellou, Deepak Narayanan, Timothy Zhu, David Dion, Thomas Moscibroda, et al. Kerveros: Efficient and scalable cloud

- admission control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 227–245, 2023.
- [88] Divyanshu Saxena, Jiayi Chen, Sujay Yadalam, Yeonju Ro, Rohit Dwivedula, Eric H Campbell, Aditya Akella, Christopher J Rossbach, and Michael Swift. How i learned to stop worrying and love learned os policies. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 1–7, New York, NY, USA, 2025. Association for Computing Machinery.
- [89] Erfan Sharafzadeh, Raymond Matson, Jean Tourrilhes, Puneet Sharma, and Soudeh Ghorbani. {Self-Clocked}{Round-Robin} packet scheduling. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1437–1465, Philadelphia, PA, 2025. USENIX Association.
- [90] Asankhaya Sharma. Openevolve: An open-source evolutionary coding agent. <https://github.com/codelion/openevolve>, 2025. Accessed: 2025-12-10.
- [91] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.
- [92] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.
- [93] Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Code researcher: Deep research agent for large systems code and commit history. Technical Report MSR-TR-2025-34, Microsoft, June 2025.
- [94] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No silver bullet: extending sdn to the data plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*, pages 1–7, New York, NY, USA, 2013. Association for Computing Machinery.
- [95] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, 2020. USENIX Association.
- [96] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gumadi. {HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1149–1163, Boston, MA, 2023. USENIX Association.
- [97] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.
- [98] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.
- [99] Paolo Valente and Arianna Avanzini. Evolution of the bfq storage-i/o scheduler. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 15–20. IEEE, 2015.
- [100] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.
- [101] Andras Varga. Omnet++. In *Modeling and tools for network simulation*, pages 35–59. Springer, 2010.
- [102] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.
- [103] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient {MRC} construction with {SHARDS}. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110. USENIX Association, 2015.
- [104] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, Changxi Zheng, Jing Wang, and Honghao Liu. Pudica: Toward Near-Zero queuing delay in congestion control for cloud gaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 113–129, Santa Clara, CA, April 2024. USENIX Association.

- [105] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024.
- [106] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- [107] Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv eprints*, pp. *arXiv-2507*, 2025.
- [108] 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. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.
- [109] Sujay Yadalam, Konstantinos Kanellis, Michael Swift, and Shivaram Venkataraman. Arms: Adaptive and robust memory tiering system. *arXiv preprint arXiv:2508.04417*, 2025.
- [110] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghuvaran, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.
- [111] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- [112] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.
- [113] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association.
- [114] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 70–79, 2023.
- [115] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, New York, NY, USA, 2023. Association for Computing Machinery.
- [116] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. {CacheSack}: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, 2022. USENIX Association.
- [117] Chen-Yu Yen, Soheil Abbasloo, and H. Jonathan Chao. Computers can learn from the heuristic designs and master internet congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM ’23, page 255–274, New York, NY, USA, 2023. Association for Computing Machinery.
- [118] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD ’19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.
- [119] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM ’22, page 414–427, New York, NY, USA, 2022. Association for Computing Machinery.
- [120] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. 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)*, pages 1229–1246, Santa Clara, CA, April 2024. USENIX Association.
- [121] Yibo Zhao and Cheng Tan. Can llms replace time-tested system policies? perhaps. In *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 168–175, 2025.

- [122] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.
- [123] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 3l-cache: Low overhead and precise learning-based eviction policy for caches. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, pages 237–254, Santa Clara, CA, 2025. USENIX Association.
- [124] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.
- [125] 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. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.
- [126] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. cache_ext: Customizing the page cache with ebpf. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP ’25*, page 462–478, New York, NY, USA, 2025. Association for Computing Machinery.

A LLM-driven survey of resource-management papers

The goal of this survey is to assess whether the two interfaces proposed by VULCAN (VALUE and RANK) are broadly applicable abstractions for expressing resource-management tasks. We use recent systems literature (last five years of OSDI and NSDI papers; 660 papers total) as a representative sample to evaluate the expressiveness of these interfaces across a wide range of real-world systems tasks that researchers are working on today. For each paper, we determine whether its primary contribution includes a novel resource-management policy or algorithm for some systems task. If so, we attempt to recast each such task using one of VULCAN’s interfaces.

A.1 LLM Assisted Pipeline

To scale this analysis, we use an LLM-assisted pipeline. For each paper, we provide the full camera-ready PDF along with a structured prompt describing the scope criteria and the classification task. We use the Amazon Nova Premier model [40] via the AWS Bedrock API, which can consume complete PDFs directly using vision-based document understanding.

The prompt is divided into two stages. The first stage determines whether a paper is in scope - *i.e.*, whether it proposes a resource-management policy or decision-making algorithm (Listing 1). For papers deemed in scope, the second stage asks the model to identify all distinct resource-management tasks and classify each task according to whether it can be expressed using the VALUE or RANK interface (Listing 2). The prompt for the second stage also includes the definitions of both the interfaces (§3.1.1, §3.1.2) as well as some illustrative examples of tasks that fit within each use case (Table 1) – these parts of the prompt are omitted from Listing 2 for brevity).

After these two stages, in the ending of the prompt, the model is instructed to reason about its decisions and emit its results in a fixed, structured JSON format, which we parse to extract scope decisions, task lists, and interface classifications. Across the full survey (660 papers), the analysis consumed 21.22M input tokens and 0.36M output tokens. Using Nova Premier pricing (input: \$2.50 per million tokens; output: \$12.50 per million tokens), the total API cost of this analysis was around \$58.

Listing 1: Prompt for scope check

You are an expert Systems Researcher assisting in a meta-analysis of past research papers (NSDI, SIGCOMM, SOSP, OSDI). You will be provided a paper and you are required to read and analyze it.

STEP 1: SCOPE CHECK - Is this paper primarily about a resource-management heuristic, policy or decision-making system?

Before any analysis, read the paper and consider the following question: does the paper’s primary contribution involve proposing, analyzing, or justifying a new resource-management policy or decision rule? Resource management policies decide how to allocate or prioritize system resources - such as scheduling, caching, memory management, etc - between different processes or tasks running on a system.

A paper is OUT OF SCOPE if:

- The paper is not about a resource management heuristic. Many papers in systems conferences might cover something else - such as systems security (e.g. describing security vulnerabilities), cryptography, new hardware design, etc.
- The policy logic is reused, standard, or incidental. (*i.e.*, policy is not a central contribution of the paper)
- The contribution is primarily mechanism design, system architecture, hardware, or performance engineering, even if it uses resource-management policies.
- One way to think about this is: if the paper would remain interesting (and useful) to read even if the policy were replaced by a very simple algorithm (e.g. random, FIFO, etc) - policy is probably not the centerpiece of this paper and this paper is out of scope.

Listing 2: Prompt for task classification

STEP 2 - ABSTRACT TASK CLASSIFICATION

Perform this analysis only if a paper is IN SCOPE.

- Identify the fundamental resource decision(s) being made (e.g., "which machine gets the job?", "what bitrate to assign?", "which page to evict?") - call this the TASK.
- Distinguish Problem from Solution: Completely ignore *how* the paper solves the problem (the APPROACH) - describe the resource management TASK only. (e.g. "selecting a page to evict" is the TASK, APPROACH could be LRU / LFU / etc).
- Focus only on the ****primary**** resource management task(s) or heuristic(s) that are the main contribution of the paper. Ignore minor / secondary / auxiliary heuristics that are not the primary point of study of the paper.
- Classify if the TASK can be thought of as (i) a VALUE-type task, (ii) RANK-type policy, or (iii) NEITHER.
- VALUE and RANK are defined below.
- If you think your paper has multiple primary tasks, do this analysis for all primary task(s).

B RANK-based cache heuristic search: prompt and sample output

This appendix shows the template used for the evolutionary search (Listing 3) and the identified *score(.)* function identified for cluster C_1 (Listing 4).

Listing 3: Prompt used for cache eviction heuristic synthesis

You are designing a new heuristic for an eviction-based caching system. This system uses a priority queue to store cached objects. The object with the lowest numeric priority is evicted when space is needed. On every access, the caching system:

- Removes the accessed object from the priority queue.
- Calls a priority function to compute the new priority of the object being accessed.
- Reinserts the object into the priority queue with the updated priority.

The priority function signature is:

```
'''cpp
int priority(
    uint64_t current_time, obj_id_t obj_id, pq_cache_obj_info& obj_info,
    CountsInfo<int32_t>& counts, AgeInfo<int64_t> ages, SizeInfo<int64_t>& sizes,
    History& history
);
'''
```

This function returns an integer priority. Higher values are more likely to stay in cache; lower values are more likely to be evicted. The inputs to this function are:

- `current_time` (`uint64_t`): the current virtual timestamp.
- `obj_id` (`obj_id_t`): unique identifier for the object being accessed
- `obj_info`: includes:
 - `obj_info.count` (`int32_t`)
 - `obj_info.last_access_vtime` (`int64_t`)
 - `obj_info.size` (`int64_t`)
 - `obj_info.addition_to_cache_vtime` (`int64_t`)
- `counts`, `ages`, and `sizes`: provide aggregate statistics on objects in cache. They all support `.percentile(p)` for $p \in [0.0, 1.0]$. For example, `ages.percentile(0.5)` returns the median age of all objects currently in cache (same unit as `vtime`).
- `history`: stores recently evicted objects. Use `history.contains(obj_id)` to check if `curr` was recently evicted and readded to cache. Additionally, if `obj_id` is in history, `auto info = history.get_metadata(obj_id)` fetches information on the object:
- `history`: stores recently evicted objects. Use `history.contains(obj_id)` to check if `curr` was recently evicted and re-added to cache. Additionally, if `obj_id` is in history, `auto info = history.get_metadata(obj_id)` fetches information on the object: specifically `info->count` (count of how many times it was accessed before eviction) and `info->age_at_eviction_time` (how long - in time - was the object present in cache before it's previous eviction).

You do not have to define function prototypes or include any headers - just write the implementation of `'priority()'`. You can choose to use all of these features or a subset of them.

REPLY FORMAT: think about the provided inputs, brainstorm possible scoring functions, weigh the possibilities, and then select one of them (ideally, the most promising one). Describe the idea you have chosen in plain English, and what you are hoping the heuristic achieves. After this text, write the code for `priority()` in a single code block as shown below. For inspiration here is how you would define the priority functions for some simple heuristics:

```
return current_time; // LRU
return obj_info.addition_to_cache_vtime; // FIFO
return obj_info.count; // LFU
```

Code:

```
'''cpp
// priority implementation here
'''
```


Listing 4: score(.) function discovered by VULCAN for cluster C_1

```
1 int priority(
2     uint64_t current_time, obj_id_t obj_id, pq_cache_obj_info& obj_info,
3     CountsInfo<int32_t>& counts, AgeInfo<int64_t> ages, SizeInfo<int64_t>& sizes,
4     History& history
5 ) {
6     int32_t base_priority = obj_info.count * 20;
7     int64_t time_since_last_access = current_time - obj_info.last_access_vtime;
8     base_priority -= static_cast<int32_t>(time_since_last_access / 300);
9     base_priority -= static_cast<int32_t>(obj_info.size / 500);
10    if (history.contains(obj_id)) {
11        auto info = history.get_metadata(obj_id);
12        base_priority += (info->count * 15) + static_cast<int32_t>(info->age_at_eviction_time / 150);
13    }
14    else base_priority -= 40;
15
16    int32_t recent_age_threshold = ages.percentile(0.75);
17    int32_t large_size_threshold = sizes.percentile(0.75);
18
19    // Penalizing for objects that are accessed too old
20    if (obj_info.last_access_vtime < recent_age_threshold) base_priority -= 30;
21
22    // Encouraging smaller objects, especially when they are not frequently accessed
23    if (obj_info.size > large_size_threshold) base_priority -= 25;
24    else base_priority += 10;
25
26    if (time_since_last_access < 1000) base_priority += 25;
27
28    int32_t access_count_threshold = counts.percentile(0.70);
29    if (obj_info.count > access_count_threshold) base_priority += 50; else base_priority -= 5;
30
31    // Additional penalty for objects that have not been accessed multiple times
32    if (obj_info.count < 3) base_priority -= 15;
33
34    return base_priority;
35 }
```

Listing 4 shows the best performing policy discovered for cluster C_1 . All of the code in the block, except the function prototype, was generated completely by the LLM. We see that the LLM uses the features available to the scoring function in interesting ways: such as penalizing objects that are old (lines 8, 20) or big (lines 9, 23) and preserving small objects (line 24) or frequent (lines 12, 29).

C Queue Topology-based cache heuristic discovered for C_7

Listing 5 shows the code for the queue topology heuristic discovered by VULCAN for cluster C_7 .

Listing 5: Queue Topology heuristic discovered by VULCAN for cluster C_7

```
1 #include "template.h"
2 #define Q0_PROMOTE_THRESHOLD 2
3 #define Q1_PROMOTE_THRESHOLD 1
4 #define Q2_STALE_AGE 100000
5
6 void get_topology_info(struct BaseTopologyInfo& topo_inf) {
7     topo_inf.num_queues = 3; // Tiny probation to filter one-hit objects, reasonable mid-stage, large main
8     topo_inf.queue_sizes.push_back(0.05); // Probation
9     topo_inf.queue_types.push_back("FIFO");
10    topo_inf.queue_sizes.push_back(0.15); // Mid-stage
11    topo_inf.queue_types.push_back("FIFO");
12    topo_inf.queue_sizes.push_back(0.80); // Main
13    topo_inf.queue_types.push_back("FIFO");
14    // Slightly larger ghost to better capture recently-evicted objects for resurrection.
15    topo_inf.ghost_queue_size = 0.40;
16    // Limit transitions to reduce thrashing between queues.
17    topo_inf.max_transitions_allowed = 4;
18 }
19
20 int get_initial_queue(const bool in_ghost, const int obj_size, std::vector<bool>& is_full) {
21     // Resurrected objects get a second chance in mid-stage so they don't immediately evict.
22     if (in_ghost) return 1;
23
24     // If probation is full but mid-stage has room, start new objects in mid-stage to reduce
25     // promotion storm when probation cycles quickly.
26     if (is_full[0] && !is_full[1]) return 1;
27     if (is_full[0] && is_full[1] && !is_full[2]) return 2;
28     return 0;
29 }
30
31 /* Probation FIFO (Q0)
32 - Aggressively promote small, re-referenced objects.
33 - Also promote items with modest global popularity to avoid repeatedly cycling the same object.
34 */
35 int transition_logic_0(obj_info_t& inf) {
36     // Probation: require local evidence of a re-reference to promote.
37     if (inf.queue_access_counter >= Q0_PROMOTE_THRESHOLD) return 1; // Promote to Q1
38     else return -1; // Evict to ghost
39 }
40
41 /* Mid-stage FIFO (Q1)
42 - Require stronger local signal or clear global popularity to reach main.
43 - Otherwise keep in mid-stage if not stale (reduces needless evictions).
44 */
45 int transition_logic_1(obj_info_t& inf) {
46     // Mid-stage: a single local re-reference is sufficient to reach main.
47     if (inf.queue_access_counter >= Q1_PROMOTE_THRESHOLD) {
48         return 2; // Promote to main
49     }
50     return -1; // Evict to ghost if it reaches tail without promotion
51 }
52
53 /* Main FIFO (Q2)
```

```

54 - Keep objects that show any activity. If very stale, perform direct eviction
55   to free space; otherwise push to ghost so we remember them briefly.
56 */
57 int transition_logic_2(obj_info_t& inf) {
58     // Require local activity in main to keep objects long-term; avoid treating one remote access
59     // recorded long ago as reason to keep cold items forever.
60     if (inf.queue_access_counter > 0) return 2; // stay in main
61     if ((inf.access_timestamp_counter - inf.queue_insertion_timestamp) > Q2_STALE_AGE)
62         return -2; // direct eviction for very stale items
63     return -1; // evict to ghost otherwise
64 }

```