# Man-Made Heuristics Are Dead. Long Live Code Generators!

Rohit Dwivedula Divyanshu Saxena Aditya Akella Swarat Chaudhuri
Daehyeok Kim
The University of Texas at Austin
Austin, TX, USA

## Abstract

Policy design for various systems controllers has conventionally been a manual process, with domain experts carefully tailoring heuristics for the specific instance in which the policy will be deployed. In this paper, we re-imagine policy design via a novel automated search technique fueled by recent advances in generative models, specifically Large Language Model (LLM)-driven code generation. We outline the design and implementation of PolicySmith, a framework that applies LLMs to synthesize instance-optimal heuristics. We apply PolicySmith to two long-standing systems policies - web caching and congestion control, highlighting the opportunities unraveled by this LLM-driven heuristic search. For caching, PolicySmith discovers heuristics that outperform established baselines on standard open-source traces. For congestion control, we show that PolicySmith can generate safe policies that integrate directly into the Linux kernel.

## CCS Concepts

• **Information systems** → **Cloud based storage**; • **Networks** → *Transport protocols*; • **Computing methodologies** → *Heuristic function construction.*

## Keywords

program synthesis, LLM-driven code generation, caching, congestion control.

**ACM Reference Format:**
Rohit Dwivedula Divyanshu Saxena Aditya Akella Swarat Chaudhuri Daehyeok Kim. 2025. Man-Made Heuristics Are Dead. Long Live Code Generators!. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3772356.3772413


*HotNets '25, November 17–18, 2025, College Park, MD, USA*

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772413

## 1 Introduction

Systems research has long treated policy design as a manual craft. Whether it is congestion control, caching, or scheduling, performance-critical systems rely on heuristics handwritten by domain experts, optimized for typical conditions, and deployed as a static policy. Yet these policies are brittle, degrading under new workloads, hardware, or objectives.

The standard response is to continually tune and reimplement heuristics: we tweak congestion control algorithms for new network environments [62], tailor prefetching and caching policies for emerging workloads [10] or hardware [65], and evolve queueing disciplines for new performance targets [17]. But this is an arms race we are losing. We face rapid evolution in workloads, deployment settings, and heterogeneous hardware. As such, the design space of heuristics for most problems is complex and shifting in a context-dependent manner. Human developers often cannot discover "the right heuristics" fast enough.

Meanwhile, learning-based systems have shown that it is possible to approach *instance-optimality*, i.e., finding the best policy for a given context by learning from data. Prior work in learned congestion control [2] and caching [58, 63, 70], for example, demonstrates that data-driven policies, represented as neural networks, can significantly outperform fixed heuristics. However, neural approaches come at a steep cost: opaque behavior [37], complex training and deployment pipelines [4, 19], inference overheads [19, 68], and safety concerns [50] that preclude adoption in many environments.

Alternatively, we find that recent advancements in coding agents such as FunSearch [49], AlphaEvolve [43], and EvoPrompting [13] present a powerful new opportunity for policy design. These agents use a form of LLM-guided evolutionary computation to synthesize expressive code that maximizes quantitative reward functions. We propose to use this approach to automatically synthesize system policies.

In our approach, *policy design is re-imagined as an automated search problem, solved as often as needed using generative models to produce instance-optimal policy code.* This means that "intelligence" in systems no longer needs to reside in hard-coded rules or opaque neural network weights. Instead, it could reside in the *process* by which policies are generated. This enables a move away from deploying fixed

logic to a pipeline that can automatically generate, evaluate, and produce optimized code for each context. By pairing LLMs with evolutionary search and test-time feedback, we can explore vast policy spaces and discover high-performing code that would be infeasible to author manually. This approach also avoids opaque learned models in favor of policy code that is safer and more interpretable.

In the limit, this shift can unlock a future where systems come with tailor-made, self-evolving policy logic, allowing developers to simply control high-level specifications that include metrics they care about, such as performance and an upper bound on the costs incurred to "search" for the policy, among other things.

**One instantiation.** To ground this idea, we sketch PolicySmith, a prototype system that uses LLMs to synthesize instance-optimal policies. Given an objective and test harness, it generates, evaluates, and evolves policy code offline. However, our broader contribution is not a tool—it is a call to rethink the boundary between systems and machine learning, and to embrace a new model of policy design.

## 2 Heuristic Design: Status Quo and Vision

Systems research has seen a long history of heuristics, such as cache eviction, congestion control, and queue scheduling, being developed, modified, and fine-tuned for a specific "context" which is defined by the workload (application and traces) being supported by the heuristic, the desired objectives (*e.g.,* performance, utilization, fairness, scalability, etc.), and the environment (*e.g.,* hardware) where the heuristic is running.

Taking the concrete example of web caching, different eviction heuristics for specific traces, objectives, or deployment scenarios [7, 31, 36, 48, 59, 60, 64, 69]. ARC [36] and SIEVE [69] perform well for large cache sizes by balancing new and old objects. In contrast, TwoQ [31] and LHD [7] perform well for smaller caches due to their ability to quickly remove low-value objects. Cacheus [48] shows that depending on whether a workload consists of mostly new objects ("scan workloads") or mostly repeated objects ("churn workloads"), different expert algorithms perform better. Additionally, heuristic design often takes into account end-to-end *objectives* such as tail latency [9] and fairness [33], or system-level *constraints* such as CPU overhead [59], lock-free design [64], and memory efficiency [18].

Because no single heuristic performs well across all contexts, experts routinely invest significant time adapting or inventing new heuristics for new workloads, objectives, and environments. This process—designing new algorithms [7, 31, 64, 69], engineering feature sets for learned models [59, 60, 63], and evaluating across diverse traces [59] using simulators [32] or real-world systems [8]—is manual and painstakingly slow. This problem is not limited to caching alone; it is widespread across several systems policies. As a result, complex systems (*e.g.,* the Linux Kernel) often continue running suboptimal policies for years, because discovering better ones requires developers with kernel expertise and is too labor-intensive. The ability to automatically discover context-specific, incrementally better heuristics can yield meaningful gains in terms of both performance metrics of interest and manual effort involved.

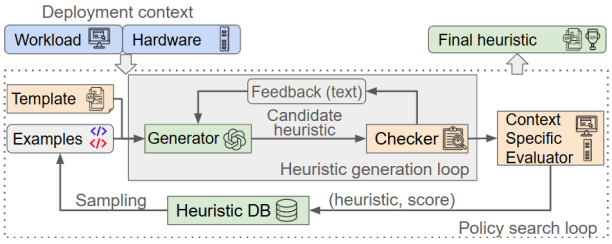



**Figure 1: PolicySmith overview**

**Our vision** is to automate the *heuristic design process* so humans can focus on what matters most: defining *what* they want. This includes both the *intent*—the goals/objectives of the heuristic—and the *space* it operates in, shaped by context and constraints. These choices require high-level reasoning about system goals, tradeoffs (like fairness or liveness), and operator needs—which are difficult to formalize and automate, and deeply benefit from human experience and intuition. Once the intent and space are clear, generating heuristics becomes a *search* problem that can be scaled, guided by feedback, and automated.

In particular, automating this search process is feasible today because large language models (LLMs) are remarkably effective *generators*: they can quickly produce a wide range of candidate heuristics, remixing and adapting known techniques across domains. Many state-of-the-art heuristics, such as SIEVE [69], ARC [36], and Cacheus [48], are delicate recombinations and improvements of existing approaches. Because LLMs have been pretrained on code and patterns from across the stack, they are well-positioned to generate *candidate* heuristics inspired by these recurring structures; though inventing entirely new structures and principles may still require human insight.

## 3 PolicySmith

To turn the vision of automated heuristic design into practice, we build the PolicySmith framework (Fig 1) that separates specification (user-defined) from policy search (automated).

The user is responsible for designing a Template, which defines the space of programs to search, and an Evaluator that runs candidate heuristics and returns a numeric *score* indicative of how well it performed in the current deployment context. The Template contains a function signature

or a partial code stub that specifies the semantics of what the heuristic must implement. For example, in a caching system, this might be a function that decides which object to evict from a set of eviction candidates; for congestion control, it could be event-driven functions that adjust the *cwnd* on packet ACK or packet loss.

The Template, used by an LLM-based Generator to produce candidate heuristics, also contains natural-language *constraints* that guide generation. These constraints describe the constructs allowed in the heuristic, including which libraries may be imported, the states it can access, and any behavioral or performance requirements. For example, congestion control heuristics on the critical path in the kernel must avoid floating-point arithmetic, locking, or unbounded loops. In caching, constraints may require `O(log N)` complexity, ruling out full-cache scans. Overall, the Template (including the constraints) acts like a "design spec", ensuring generated heuristics are efficient and deployable.

The LLM, of course, may produce code that does not honor these constraints, due to hallucination [28, 44], producing plausible yet non-conforming or incorrect code. To catch such violations, users define a Checker that enforces syntactic and semantic rules, and provides structured feedback to help generators stay within spec.

With these components defined, the framework begins its search loop. Motivated by recent coding agents such as AlphaEvolve [43] and FunSearch [49], we leverage evolutionary search to explore the large space of programs defined by the Template. This choice is motivated by the fact that the space of "good" heuristics is discrete, combinatorially large, and sparsely populated – making gradient-based methods impractical and local search [51] limited in efficacy.

In our search process (Fig 1), an LLM-based Generator synthesizes multiple candidate heuristics based on the Template, which are scored by the context-specific Evaluator, and the best-performing candidates are fed back as examples in the next round. This loop continues for several iterations, gradually steering the generator toward better-performing heuristic code. At the end, PolicySmith outputs a final heuristic tailored to the target context.

## 3.1 Responding to Context Shifts

PolicySmith generates instance-optimal heuristics for each *context*, defined as a combination of the workload, hardware, and the desired objectives. However, how is a context actually delineated in practice? How do we identify when the deployment context has drifted enough to warrant re-synthesis?

*3.1.1 Explicit context shifts.* Some context changes might be obvious. For instance, upgrading from HDDs to SSDs usually requires a new block I/O scheduling heuristic [54] even if nothing else changes (hardware change); different application classes on the same hardware benefit from different process scheduling [20] and networking queuing heuristics [57] (workload change). In such scenarios, PolicySmith can be invoked manually for each context.

*3.1.2 Implicit context shifts.* Context changes are not always readily apparent. A CDN server, for instance, may experience shifts in access patterns due to the time of day, even if the hardware and objectives remain unchanged. A cloud provider may not have visibility into application-level changes that affect the performance of heuristics. These drifts degrade the performance of previously specialized heuristics without triggering any explicit event.

To address this, prior work has developed runtime adaptation systems that leverage online techniques such as reinforcement learning [67], bayesian optimization [5], and multi-armed bandits [14, 41] to identify context changes and respond to them, by selecting a new heuristic or configuration from a pre-defined set. These systems rely on lightweight monitoring infrastructure (*e.g.,* guardrails [50]) or clustering-based approaches [14] to detect context shifts.

PolicySmith complements these approaches: the same monitoring signals that guide these systems can also trigger PolicySmith automatically, allowing it to re-synthesize a heuristic offline. Until a new, better heuristic is ready, the existing one continues to be used with degraded performance. Over time, this enables building a *library* of PolicySmith-generated heuristics, providing better options for an adaptation system to choose from.

This paper does not focus on designing context-detection or runtime-adaptation systems, and rather assumes such triggers (manual or automated) are available. Given such a trigger, the core question then becomes: can we reliably synthesize effective heuristics for the new context? We will explore this in two case studies[1].

First (§4), we use PolicySmith to discover instance-optimal cache eviction heuristics, illustrating its ability to generate high-performing, context-specific policies. In the second case study (§5), we use PolicySmith to evolve Linux kernel code to synthesize congestion control heuristics, demonstrating the feasibility of our vision even in highly constrained, safety-critical systems such as the Linux kernel.

# 4 Case Study: Web Caching

We now describe an instantiation of PolicySmith to find instance-optimal eviction policies for web caches. Our prototype is built on libCacheSim [32], a high-performance web cache simulator with an event-driven interface. We describe our prototype design (§4.1), followed by results (§4.2).

[1]All code used for these case studies is available at https://github.com/ldos-project/policysmith

| Type | Attributes |
|---|---|
| Per object | Number of accesses (count), last access time, time added to cache, object size |
| Aggregates | Percentiles over access counts, ages, or sizes (*e.g.,* p50 size in bytes of all objects in cache). |
| History | List of recently evicted objects, along with (timestamp, access count, age) at eviction. |

**Table 1: Features available to `priority()`.**

## 4.1 Design

*4.1.1 Tradeoffs in template design.* Designing an effective Template for heuristic generation in PolicySmith requires deciding which parts of the heuristic should be exposed to the Generator and which to hold fixed. Many heuristics (including web caching) naturally consist of two parts: *state management* (e.g., tracking metadata) and *decision-making logic* (e.g., choosing what to evict).

One option is to use the Generator to synthesize both. However, this requires the LLM to coordinate logic and state across multiple interdependent functions. This increases the size of each candidate heuristic, increases the computational cost, and is more likely to result in errors due to complexity. At the other extreme, we could fix the state management—*i.e.,* the data structures used to track cache metadata—to match a known policy (*e.g.,* GDSF [15], ARC [36]) and evolve only the decision logic. While this simplifies code generation, it severely limits what can be discovered: it becomes impossible to discover heuristics that depend on richer signals like access history, temporal patterns, or global cache statistics.

*4.1.2 Template definition.* For this case study, we use a Template that favors simplicity and ease of generation. In our design, object metadata is stored in a priority queue, with the position determined by a customizable `priority()` function, that is synthesized by the Generator. This `priority` function is invoked on each access or insertion and updates the object's priority score; when needed, the object with the lowest score is evicted. To support diverse strategies, `priority()` has access to a rich set of features (Table 1) designed as a superset of the features used by existing caching policies. This enables the discovery of varied heuristics without changing the underlying queue structure or interface.

The resulting heuristic may incur higher overhead than policies like LRU or FIFO, due to `O(log N)` priority updates on each access. This may be acceptable for caches with relatively fewer objects (our focus in this case study). For caches where the overhead is prohibitive, alternative Template designs with stricter constraints, such as using approximate structures like soft heaps [12], may be needed.

*4.1.3 Generator.* Because our Template is narrow and self-contained, we use a lightweight LLM, GPT-4o mini [29] via the OpenAI API, for heuristic generation. Using a smaller model like this keeps generation fast and inexpensive; more broadly, the complexity of the Template governs the sophistication of the required Generator. As templates become richer or span multiple functions, they may require larger, more capable models to handle reasoning and coordination. As LLMs continue to improve, we expect this design space to widen, enabling the evolution of heuristics in more complex templates. The narrow Template implies that most errors surface as build failures, allowing for a simple Checker that automatically feeds errors back to the Generator.

*4.1.4 Context-specific evaluator.* In caching, the *context* is defined by the cache size (typically constrained by hardware) and the *access patterns of requests* (*i.e.,* the workload). For our case study, we use two real-world block I/O trace datasets: CloudPhysics [61] (with 105 week-long traces from diverse VMs) and Microsoft Research Cambridge (MSR) dataset [40] (with 14 traces from production servers). In our case study, the Evaluator scores candidate heuristics by running them on *a single* block I/O trace from these datasets, for a fixed cache size (10% of the trace footprint). Each pair (trace, cache size) defines a distinct *context*, and the objective is to minimize the object miss rate. While we define context narrowly for simplicity, one could alternatively treat an entire distribution of traces—*e.g.,* a sample from all 105 CloudPhysics traces—as a single, broader context, evaluating heuristics based on an aggregate metric such as average hit rate.

## 4.2 Results

*4.2.1 Methodology.* We select **one** CloudPhysics trace (`w89`) and use it as the *context*. The prompt to the Generator includes: a natural language description of our priority queue interface and available features (Table 1), the function signature for `priority()`, and example priority functions seeded at the start of the search—namely, for LRU and LFU. In each round of PolicySmith, we prompt the Generator repeatedly to generate 25 candidate heuristics, which are then evaluated by the Evaluator. The top two performing heuristics across all previous rounds are then used as examples in the next round. This process is repeated 20 times, yielding a total of 500 heuristics. The best-performing heuristic from this search, referred to as Heuristic $\mathcal{A}$, is shown in Listing 1. We repeat this process independently for three more CloudPhysics traces to obtain heuristics $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$, and on four MSR traces to obtain heuristics $\mathcal{W}$, $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$.

*4.2.2 Baselines and Metrics.* We use fourteen eviction algorithms as our baselines: GDSF [15], S3-FIFO [64], SIEVE [69], LIRS [30], LHD [7], Cacheus [48], FIFO-Reinsertion [16] (denoted FIFO-Re), LeCaR [60], SR-LFU [48], CR-LRU [48], LRU, MRU, and FIFO. For each caching heuristic, we report the

improvement in the miss ratios over a fixed baseline (FIFO), similar to how [64, 69] report their results.

*4.2.3 Instance-optimality of synthesized heuristics.* All heuristics produced by PolicySmith ($\mathcal{A}$–$\mathcal{D}$ and $\mathcal{W}$–$\mathcal{Z}$) either match or outperform **all** 14 baselines for their original context (*i.e.,* the trace used in that instance's Evaluator), demonstrating that PolicySmith can tailor heuristics for each context. Additionally, we evaluate each PolicySmith-generated heuristic not just on its original trace, but across all other traces within the same dataset. Table 2 summarizes this: for instance, Heuristic $\mathcal{A}$, generated using trace `w89`, outperforms all baselines on 48% of CloudPhysics traces; similarly, Heuristic $\mathcal{X}$ does so for 64% of MSR traces. This suggests that traces within a dataset may share structural similarities (*e.g.,* request access patterns), allowing heuristics tuned on one to remain competitive on others.

While generalization is not our goal, future instantiations of PolicySmith could also define context more broadly, *e.g.,* by using multiple similar traces in the Evaluator. Our findings also indicate that the discovered heuristics do not overfit to a single trace.

*4.2.4 Performance of PolicySmith.* Figure 2 shows the performance of baselines and the PolicySmith-synthesized heuristics on all traces from CloudPhysics and MSR datasets. Notably, Heuristic $\mathcal{A}$ and Heuristic $\mathcal{Y}$ achieve the second-highest average performance (surpassed only by GDSF) for CloudPhysics and MSR datasets, respectively.

The figures also include two *oracles*: (1) the baseline oracle (B-Oracle), which selects the best-performing baseline (§4.2.2) for each trace, and (2) the PolicySmith-oracle (PS-Oracle), which selects the best heuristic from both the baselines and PolicySmith-synthesized heuristics. These oracles model ideal runtime adaptation (§3.1), perfectly identifying context and selecting the best heuristic for each trace. They serve as *upper bounds* on expert performance. The PS-Oracle has a 2% higher improvement over FIFO than the baseline oracle, demonstrating the additional performance gains possible with the addition of PolicySmith-generated heuristics.

*4.2.5 Example of evolved heuristic.* Listing 1 shows Heuristic $\mathcal{A}$, the best performing heuristic we found for CloudPhysics. 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 provided (Table 1) in interesting ways: such as penalizing objects that are old (lines 4, 13) or big (lines 5, 15) and preserving small objects (line 16) or frequent (lines 8, 18). As discussed in §4.2.1, the initial seed heuristics provided to PolicySmith are simple algorithms—LRU and LFU—that can be implemented in the priority function in a single line, *e.g.,* by returning `obj_info.count` and `obj_info.last_accessed` for LFU and LRU, respectively.

*4.2.6 Computational cost.* The search for heuristic $\mathcal{A}$ required 5.5 CPU-hours to evaluate all candidate heuristics. It also used 800k input tokens and 300k output tokens with the GPT-4o-mini model. The total cost of the OpenAI API for the eight runs in this section was approximately USD $7.

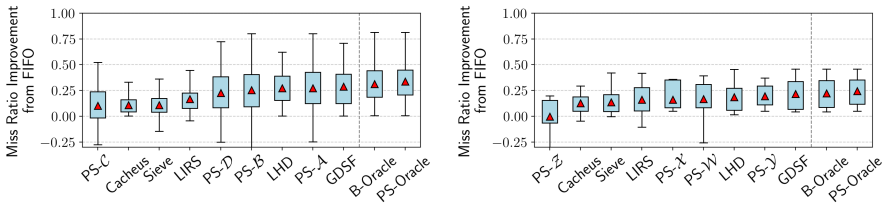
## 5 Case Study: Congestion Control

In this section, we explore whether PolicySmith can be extended to a more demanding setting: evolving heuristics in the Linux kernel. Modern kernels house several policies like TCP congestion control [1, 6, 11, 25], packet scheduling [22, 46, 53, 57], and block I/O scheduling [27, 47, 54], that have been shaped by decades of manual tuning for specific contexts.

Instantiating PolicySmith to perform policy search in the kernel is particularly challenging for two core reasons. First, the kernel programming environment is highly constrained (*e.g.,* floating-point ops disallowed, idiosyncratic patterns to access registers, BPF maps, etc). These constraints make code generation especially difficult for the Generators, which struggle to produce syntactically and semantically valid code in such narrow, domain-specific contexts [24]. Second, kernel development comes with strict safety and performance requirements: bugs can lead to kernel panics, and even minor inefficiencies can degrade system-wide behavior. This makes the design of the Checker and Evaluator critical. To evaluate whether these challenges can be overcome, we instantiate PolicySmith in the context of TCP congestion control. This case study is not aimed as a search for new instance-optimal algorithms, but as a focused case study to test whether PolicySmith can navigate the constraints and risks of kernel-space policy generation.

*5.0.1 Template design.* The Linux kernel requires congestion control algorithms to implement five event-driven callbacks that update the congestion window (cwnd) in response to packet-level events. Following the Template design used for caching (§4.1.2), we isolate decision logic from state management, exposing only the decision making function (`cong_control`) to the Generator. This function is provided with a rich set of features, such as previous cwnd, minimum RTT, inflight bytes, among others. To enable reasoning over history, our Template also provides *history arrays* - time series arrays that capture smoothed versions of these metrics over the last 10 RTT intervals [66].

*5.0.2 Template and Checker.* We implement our Template as a Linux kernel module. However, rather than compiling LLM-generated code directly into the kernel, we offload the generated logic to a dynamically attached eBPF probe that is attached to the `cong_control` Linux kernel function. At runtime, whenever this function is invoked, the probe is triggered as well. The eBPF program executes the generated




(a) CloudPhysics Traces (b) MSR Traces

**Figure 2: Miss ratio improvements over FIFO for all traces in a dataset (higher is better, triangles indicate mean, heuristics ordered left to right by increasing average). Top five performing baselines shown in plot for brevity.**

| Dataset | %age of traces | |
|---|---|---|
| CloudPhysics | $\mathcal{A}$ (48%) | $\mathcal{B}$ (42%) |
| | $\mathcal{C}$ (14%) | $\mathcal{D}$ (31%) |
| MSR | $\mathcal{W}$ (57%) | $\mathcal{X}$ (64%) |
| | $\mathcal{Y}$ (57%) | $\mathcal{Z}$ (21%) |

**Table 2: Performance of discovered heuristics: proportion of traces in the dataset where synthesized heuristic outperforms all baselines.**

logic, computes the updated decision (*i.e.,* cwnd), and writes the result to a BPF map. This design ensures that all candidate programs pass the eBPF verifier [52] before execution - which acts as the Checker in our framework. This pattern – using a kernel module for scaffolding and template logic, combined with an eBPF probe for executing generated code and communicating via a BPF map – is a general mechanism that can be applied to other kernel components such as packet schedulers, I/O controllers, or memory managers.

*5.0.3 Preliminary Results.* We generated 100 candidate congestion control heuristics and attempted to compile them into eBPF programs. Only 63% of the candidates passed the eBPF verifier on the first try, and an additional 19% successfully compiled after the Generator was provided with the `stderr`. The most common causes of errors were the use of floating-point arithmetic and missing checks for division by zero. This compilation rate for kernel code is substantially lower than what we observed for caching: where 92% of candidates compiled in the first pass itself.

```
priority(now, obj_id, obj_info, counts, ages,
    sizes, history):
  score = obj_info.count * 20;
  age   = now - obj_info.last_accessed;
  score -= age/300;
  score -= obj_info.size/500;
  if (history.contains(obj_id)){
    h = history.get_metadata(obj_id);
    score += h->count*15;
    score += h->age_at_eviction_time/150;
  }
  else score-=40;
  recent = ages.percentile(0.75);
  if (obj_info.last_accessed<recent) score-=30;
  big = sizes.percentile(0.75);
  if (obj_info.size > big) score-=25;
  else score+=10;
  frequent = counts.percentile(0.7);
  score += (obj_info.count>frequent) ? 50:-5;
  if (age < 1000) score+=25;
  if (obj_info.count < 3) score -= 15;
  return score;
```

**Listing 1: A heuristic discovered by PolicySmith.**

We evaluated the heuristics that compiled successfully on a 12 Mbps, 20ms delay emulated link [42]. The resulting behaviors varied widely: bandwidth utilizations ranged from 23% to 98%, and average queuing delays spanned from 2ms to 40ms. This variance illustrates the diversity of policies that can be explored using automated strategies like PolicySmith.

## 6 Discussion

*Per-Instance Specialization in Practice.* Our thesis explicitly abandons the pursuit of "universal" heuristics. Instead, we target *instance-optimality*—generating code tailored to each workload, hardware target, and objective function. This specialization raises new challenges: How do we reliably detect when the instance has changed? Can we incrementally update policies? What abstractions enable policy reuse across similar contexts? These are tantalizing open questions, with only a few having initial answers (e.g., guardrails [50]).

*Per-Policy vs. End-to-End Coordination.* Policy synthesis today often targets individual components (e.g., a congestion control algorithm or cache eviction strategy). But in real systems, these policies interact—coordinating across the network stack, storage hierarchy, and compute resources. A key research direction is to extend synthesis to reason about *interactions between synthesized policies*, or even synthesize policies end-to-end across components to achieve global objectives. Can we express cross-cutting goals and constraints? Can LLMs or other program synthesis tools understand and generate such coordinated logic?

*Evaluation Without Deployment.* Offline policy synthesis hinges on the ability to evaluate candidate policies without full deployment. This raises questions about test harness fidelity, simulation accuracy, and robustness to distribution shift. For safety- or latency-critical systems, policy evaluation must ensure that synthesized logic is not just performant, but correct. Integrating synthesis with formal methods, fuzzing, or worst-case scenario testing may help bridge this gap.

*Trade-offs between Expressivity and Interpretability.* Models like transformers can capture complex feature interactions

via attention, enabling behaviors beyond traditional human-designed heuristics, which are often shallow and simplified. LLM-generated code may offer a middle ground – more expressive than typical handwritten logic, yet still interpretable and efficient. Moreover, LLMs can be tuned to produce simpler code, preserving interpretability when needed.

*Tools, Workflow, and Developer Experience.* In this new paradigm, the "programmer" becomes a supervisor of synthesis, not the author of logic. This shift requires a rethink of developer tools: how to prompt, debug, validate, and evolve synthesized policies. Understanding how systems engineers interact with these tools—and how to guide synthesis with expert insight—is a key opportunity.

## 7 Related Work

Prior work, such as [38, 56, 57], have hinted at the lack of a universal heuristic and that instance-optimal heuristics are needed for various domains. While these efforts propose programmable interfaces, they still rely on developers to craft optimal policies. In contrast, PolicySmith automatically discovers effective heuristics given such an interface.

Other approaches use solvers [3, 21] or program search [35] to generate code, but require detailed system models. PolicySmith avoids these challenges by using high-level specifications, relying on Generator and Evaluator to search without needing a formal model.

There is a substantial body of recent work on coding agents that combine LLM queries and evolutionary search. For example, FunSearch [49] uses such an approach for automated discovery of mathematically interesting artifacts; LaSR [23] and LLMSR [55] use the approach for scientific discovery; EvoPrompting uses it for neural architecture search; and AlphaEvolve [43] uses it for a range of tasks from mathematical discovery to the synthesis of scheduling heuristics. Our proposed approach is the first to generalize these methods into a unified approach to system policies.

In the context of systems policies, prior work has explored the use of LLMs to automate specific pieces of policy design: such as feature engineering [26], explaining heuristic behavior [34, 45], or synthesizing router configurations [39].

## 8 Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) Grant Numbers 2326576 and 2212559. Dwivedula was supported with an Amazon AI Fellowship.

## References

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

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

[3] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2022. Automating network heuristic design and analysis. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (Austin, Texas) *(HotNets '22)*. Association for Computing Machinery, New York, NY, USA, 8–16. https://doi.org/10.1145/3563766.3564085

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

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

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

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann

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

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

[10] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2024. FetchBPF: Customizable Prefetching Policies in Linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 369–378. https://www.usenix.org/conference/atc24/presentation/cao

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

[12] Bernard Chazelle. 2000. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)* 47, 6 (2000), 1012–1027.

[13] Angelica Chen, David Dohan, and David So. 2023. Evoprompting: Language models for code-level neural architecture search. *Advances in neural information processing systems* 36 (2023), 7787–7817.

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

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

[16] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Massachusetts Institute of Technology, Cambridge, MA.

[17] Jonathan Corbet. 2018. Let them run cake. https://lwn.net/Articles/758353/

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

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

[20] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 72–88. https://doi.org/10.1145/3582016.3582036

[21] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 72–88. https://doi.org/10.1145/3582016.3582036

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

[23] Arya Grayeli, Atharva Sehgal, Omar Costilla Reyes, Miles Cranmer, and Swarat Chaudhuri. 2024. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems* 37 (2024), 44678–44709.

[24] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the Effectiveness of Large Language Models in Domain-Specific Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 3, Article 78 (Feb. 2025), 22 pages. https://doi.org/10.1145/3697012

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

[26] Zhiyuan He, Aashish Gottipati, Lili Qiu, Xufang Luo, Kenuo Xu, Yuqing Yang, and Francis Y. Yan. 2024. Designing Network Algorithms via Large Language Models. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 205–212. https://doi.org/10.1145/3696348.3696868

[27] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 301–314. http://www.usenix.org/conference/atc19/presentation/hedayati-queue

[28] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[29] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[30] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.

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

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

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

[34] Franck Le, Mudhakar Srivatsa, Raghu Ganti, and Vyas Sekar. 2022. Rethinking data-driven networking with foundation models: challenges and opportunities. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, USA, 188–197. https://doi.org/10.1145/3035918.3064018

[35] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. Association for Computing Machinery, New York, NY, USA, 122–126. https://doi.org/10.1145/36206.36194

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

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

[38] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. Universal Packet Scheduling. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (Philadelphia, PA, USA) *(HotNets-XIV)*. Association for Computing Machinery, New York, NY, USA, Article 24, 7 pages. https://doi.org/10.1145/2834050.2834085

[39] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to synthesize correct router configurations?. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, USA, 189–195.

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

[41] Usama Naseer and Theophilus A. Benson. 2022. Configanator: A Data-driven Approach to Improving CDN Performance.. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1135–1158. https://www.usenix.

org/conference/nsdi22/presentation/naseer

[42] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, USA, 417–429.

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

[44] Pedro A. Ortega, Markus Kunesch, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Joel Veness, Jonas Buchli, Jonas Degrave, Bilal Piot, Julien Perolat, Tom Everitt, Corentin Tallec, Emilio Parisotto, Tom Erez, Yutian Chen, Scott Reed, Marcus Hutter, Nando de Freitas, and Shane Legg. 2021. Shaking the foundations: delusions in sequence models for interaction and control. arXiv:2110.10819 [cs.LG] https://arxiv.org/abs/2110.10819

[45] Sagar Patel, Dongsu Han, Nina Narodytska, and Sangeetha Abdu Jyothi. 2024. Toward Trustworthy Learning-Enabled Systems with Concept-Based Explanations. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, USA, 60–67.

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

[47] Zebin Ren, Krijn Doekemeijer, and Animesh Trivedi. 2024. A Systematic Configuration Space Exploration of the Linux Kyber I/O Scheduler. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering* (London, United Kingdom) *(ICPE '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 167–173. https://doi.org/10.1145/3629527.3651416

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

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

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

[51] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

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

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

[54] Kai Shen and Stan Park. 2013. {FlashFQ}: A fair queueing {I/O} scheduler for {Flash-Based}{SSDs}. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 67–78.

[55] Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. 2024. Llm-sr: Scientific equation discovery via programming with large language models. *arXiv preprint arXiv:2404.18400* (2024).

[56] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 44–57. https://doi.org/10.1145/2934872.2934899

[57] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. 2013. No silver bullet: extending SDN to the data plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*. Association for Computing Machinery, New York, NY, USA, 1–7.

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

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

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

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

[62] 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. 2024. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 113–129. https://www.usenix.org/conference/nsdi24/presentation/wang-shibo

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

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

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

[66] Chen-Yu Yen, Soheil Abbasloo, and H. Jonathan Chao. 2023. Computers Can Learn from the Heuristic Designs and Master Internet Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 255–274. https://doi.org/10.1145/3603269.3604838

[67] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

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

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

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