

WHITE PAPER · DATABASE CACHING

Learned Cache Replacement

Machine learning for eviction and admission

White Paper 2 · Database Caching series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Learned Cache Replacement: Machine Learning for Eviction and Admission

White Paper 2 · Database Caching series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Database Caching, paper 2 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract.....	4
2. Introduction	4
2.1 Why learn eviction and admission decisions.....	4
2.2 How this paper builds on paper 1	4
2.3 Scope.....	4
2.4 Terms used in this paper	5
3. The target: Belady's policy and its relaxations	5
3.1 The optimal offline policy	5
3.2 Variable object sizes	5
3.3 Relaxing the target	5
4. Families of learned eviction policies.....	5
4.1 Online learning over experts.....	6
4.2 Learning from distributions	6
4.3 Object-level prediction	6
4.4 Heuristic-aided learning	7
4.5 Group-level learning	7
4.6 Learning to tune a heuristic	7
4.7 Imitation and reinforcement learning	7
4.8 The hardware lineage	8
5. Learned admission.....	8
6. Prediction with guarantees	8
7. Learned components in database buffer pools and storage.....	9
8. Engineering a learned cache.....	9
8.1 Features and metadata.....	9
8.2 Labels and training.....	9

8.3 Inference cost and candidate sampling.....	9
8.4 Drift and retraining	10
8.5 Choosing the objective.....	10
8.6 Safe rollout and measurement.....	10
8.7 Traces, tools and reproducibility	10
9. How simple heuristics compare	10
10. Adjacent work in the AI era.....	11
10.1 KV-cache eviction in language-model serving.....	11
10.2 Policies generated by language models	11
11. Comparative analysis.....	11
12. Open challenges and future opportunities.....	12
12.1 The cost of decisions.....	12
12.2 Drift and robustness.....	12
12.3 Choosing the right objective	12
12.4 Generalization across workloads and tenants.....	12
12.5 Fair and reproducible evaluation	12
12.6 Interpretability and operability	12
12.7 Learned policies for database caches.....	13
12.8 Language models as policy designers.....	13
13. What this means for platform decisions	13
References	13

1. Abstract

Every cache must decide which entries to keep and which new entries to admit. Classic policies such as LRU, LFU and their adaptive descendants encode a fixed idea of what makes an entry worth keeping, and each works well on some access patterns and poorly on others. Learned cache replacement replaces or augments that fixed idea with a model trained on the workload's own history. This paper surveys the field. It starts from the optimal offline policy that most learned approaches imitate and the relaxations that make it learnable. It then describes the main families of learned policies: online learning over expert policies, statistical models of object behavior, per-object prediction of the next access, heuristic-aided learning, group-level learning, and learning that tunes a simple heuristic off the critical path. It covers learned admission, prediction-augmented algorithms with worst-case guarantees, and applications to database buffer pools. It then discusses what it takes to run a learned cache in practice (features, delayed labels, inference cost, drift, objectives and safe rollout) and what recent studies show about how simple heuristics compare. It closes with adjacent work on KV-cache eviction for language models and on policies generated by language models, a qualitative comparison, open challenges and notes for platform decisions.

2. Introduction

2.1 Why learn eviction and admission decisions

Paper 1 of this series described the classic eviction and admission policies. Each embodies an assumption: LRU that recent use predicts future use, LFU that frequent use does, ARC and its relatives that a balance of the two can be tuned online. Real workloads mix patterns, change over time and differ between tenants of the same cache, so no fixed assumption fits all of them. Learned policies take a different approach. They collect features of each entry's past behavior, learn from what actually happened next, and use that knowledge to choose victims or admit new entries. Two developments made the idea practical: production traces that show how diverse cache workloads are [1], and research showing how to turn the optimal offline decisions into training labels [2], [3].

2.2 How this paper builds on paper 1

Paper 1 introduced Belady's optimal policy, miss-ratio curves, the adaptive policies ARC, LIRS and 2Q, the TinyLFU admission filter, and the recent FIFO-based designs S3-FIFO and SIEVE. This paper assumes those foundations. Its question is narrower: when, and how, does learning improve on them, and at what cost? It uses the same evidence standard as the rest of the series, and it gives particular weight to studies that compare learned policies against strong simple baselines.

2.3 Scope

This paper covers learned eviction and admission for software caches: CDN and web caches, in-memory key-value caches, flash caches and database buffer pools. It cites hardware cache work where it introduced an idea that software caches later adopted, and it treats KV-cache eviction in language-model serving as adjacent work. It concentrates on peer-reviewed work from 2016 to 2026 and cites foundational work where needed. Preprints and workshop papers are identified as such. It does not report hit ratios, miss-ratio reductions,

throughput or overhead figures; where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.4 Terms used in this paper

A **policy** decides which entry to evict (eviction) or whether to store a new entry (admission). A **feature** is a measured property of an entry, such as the time since its last access, its access count or its size. A **label** is the outcome a model learns to predict, such as the time until the entry's next access; in caching, labels are known only after the fact, so they are **delayed**. An **expert** is a fixed policy, such as LRU, whose advice a learning algorithm may follow. **Regret** is the extra cost of an online decision compared with the best fixed choice in hindsight. **Drift** is a change in the workload that makes a trained model's assumptions out of date.

3. The target: Belady's policy and its relaxations

3.1 The optimal offline policy

Belady's policy evicts the entry whose next use lies furthest in the future [4]. It needs knowledge of future requests, so no online cache can run it, but it can be computed on a recorded trace. That makes it both the yardstick against which policies are measured and, for most learned policies, the teacher: the model learns to predict what Belady's policy would have done.

3.2 Variable object sizes

Belady's policy is optimal only when all objects have the same size. Web and CDN caches hold objects of very different sizes, and with variable sizes, finding the optimal decisions is computationally hard. Berger, Beckmann and Harchol-Balter modeled caching as a min-cost flow problem to compute practical upper and lower bounds on the optimal policy, showing how much room remains above existing policies and providing near-optimal decisions that can serve as training labels [2].

3.3 Relaxing the target

Imitating Belady exactly is hard, because it requires ranking every entry by its next access. LRB relaxes the target: any entry whose next request lies beyond a threshold, which its authors call the Belady boundary, is an acceptable victim [3]. Under that relaxation, a policy need not find the single best victim, only a good one, so it can examine a small random sample of candidates and still find an entry beyond the boundary [3]. LHR takes a related route, computing an online upper bound on optimal hit probability and learning from its decisions [5]. These relaxations are what turned Belady's policy from a theoretical bound into a practical training signal.

4. Families of learned eviction policies

The literature can be grouped by what the model learns and at what granularity. GL-Cache's authors distinguish object-level learning, learning from distributions, and learning from simple experts [6]; later work added heuristic-aided, group-level and parameter-tuning approaches. Table 1 summarizes the families; the subsections give detail.

Family	What the model learns	Examples	Main cost	Main strength
Online learning over experts	Which fixed policy (or setting) to trust now	LeCaR, CACHEUS, Darwin, ALPS	Limited by the experts chosen	Low overhead; adapts quickly
Learning from distributions	Statistical model of behavior by age and size	LHD	Sampling many candidates per eviction	Principled, lightweight model
Object-level prediction	Next-access time, or its distribution, per object	LRB, Raven, 3L-Cache	Per-object features and inference	Closest approximation of Belady
Heuristic-aided learning	Preferences among a few candidates a heuristic proposes	HALP, MAT	A heuristic and a model to maintain	Low inference cost; deployable
Group-level learning	Utility of groups of objects	GL-Cache	Coarser decisions	Amortizes learning over many objects
Learning to tune a heuristic	Cache-level parameters of a simple policy	Learning-Augmented Heuristics (S4-FIFO)	Pre-training across traces	Simple data path; interpretable
Imitation and reinforcement learning	A policy from oracle actions or rewards, or expert weights	Parrot, RL-Cache, Zhou et al.	Training difficulty; model size	Can capture complex patterns

4.1 Online learning over experts

The lightest form of learning keeps a set of fixed policies and learns which one to follow. LeCaR treats LRU and LFU as two experts and uses regret minimization to shift weight toward whichever expert's evictions would have caused fewer misses [7]. CACHEUS extends the idea: it characterizes workloads as mixes of primitive patterns (LRU-friendly, LFU-friendly, scans and churn), introduces scan-resistant and churn-resistant experts, and learns which to trust as the workload shifts [8]. Darwin applies the approach to admission knobs in a CDN's hot object cache, clustering traffic patterns and selecting among candidate settings with a neural bandit, so that the cache adapts when traffic shifts and can optimize different objectives [9]. ALPS uses a light model, fed by compact sketches, to choose among classic heuristics, addressing the objection that learned components keep per-item state and run on every access [10]. Expert-based learning is cheap and robust, but its decisions are limited by the experts in its set.

4.2 Learning from distributions

LHD predicts each object's hit density, its expected hits per unit of space consumed, from conditional probabilities over its age and size that are re-estimated online, and evicts the object with the lowest hit density among a sample [11]. The model is statistical rather than a trained predictor, and it adapts in real time. Its authors implemented it in a memcached-based key-value cache designed to avoid synchronization in the common case [11].

4.3 Object-level prediction

Object-level policies predict, for each object, something close to what Belady's policy needs. LRB regresses the logarithm of each object's time to next request with gradient-boosted trees, using features that generalize those of classic policies: the gaps between consecutive

requests, exponentially decayed counters at several rates, and static attributes such as size [3]. It samples objects from a sliding window, labels each when its next request arrives or when it passes the Belady boundary without one, retrains periodically so the model tracks the workload, and at eviction scores a small random set of candidates in one batch [3]. Its authors chose gradient-boosted trees because they need no feature normalization, tolerate missing values and are cheap to train and evaluate [3]. Raven learns the full distribution of each object's next arrival time with a mixture density network and evicts the object most likely to be requested last, an explicitly probabilistic approximation of Belady [12]. 3L-Cache attacks the overhead of object-level learning directly, by filtering which requests become training data, adapting retraining frequency, sampling candidates with a bias toward less popular objects, and tuning its parameters per workload [13].

4.4 Heuristic-aided learning

Object-level prediction is expensive when every eviction scores many candidates. Heuristic-aided designs let a classic policy propose a few candidates and let a model choose among them. HALP, built for YouTube's content delivery network, has a heuristic such as LRU propose a small candidate set and a small neural network rank the candidates by pairwise comparison; the comparisons become training examples, labeled after the fact by which object was accessed first, and the model is updated online from a replay buffer [14]. Its authors argue that sampling many candidates per eviction costs too much CPU in production, and report that HALP has run as the eviction policy of YouTube's CDN DRAM cache since early 2022 [14]. MAT, a preprint, follows the same pattern, using the heuristic as a filter that supplies both likely victims and useful training samples, so that the model needs only a few predictions per eviction [15].

4.5 Group-level learning

GL-Cache amortizes learning across groups. It groups objects by insertion time, learns a utility for each group online, evicts the least useful groups, and uses lightweight per-object metrics to retain a few valuable objects from evicted groups [6]. Learning per group cuts training, inference and metadata costs, so throughput approaches that of simple heuristics, and group size becomes a knob that trades efficiency against throughput [6].

4.6 Learning to tune a heuristic

The most recent direction moves learning off the request path altogether. Learning-Augmented Heuristics splits the cache into a fast data path that runs a simple policy and a control path in which a model, pre-trained across many production traces, occasionally sets the policy's cache-level parameters from cache-level features [16]. Its concrete policy, S4-FIFO, is a tuned variant of S3-FIFO; its authors present it as avoiding the large regressions of earlier learned policies and as interpretable, with a language model explaining the chosen configurations [16]. This design keeps the per-request cost of a heuristic while letting learning adapt it.

4.7 Imitation and reinforcement learning

Parrot frames eviction as imitation learning: Belady's oracle supplies the expert action, and a neural policy learns to mimic it from past accesses [17]. It is a research model evaluated in simulation, and its authors released an environment for cache-replacement research [17]. Reinforcement learning has been applied mainly to admission: RL-Cache learns a CDN

admission policy with model-free reinforcement learning over a wide feature set, and its authors report that a model trained at one location transferred to others in the same region [18]. RL faces a structural obstacle in caching: the reward for a decision arrives long after it is made, because a large cache holds many objects; Wang et al. shortened the horizon by subsampling the trace and shrinking the cache, then transferring the policy to larger caches [19]. Zhou et al. avoid learning eviction from scratch: their deep reinforcement learning agent learns how to weight two conventional replacement policies according to the workload, for cloud block storage, with offline training constrained so that it does not disturb the live system [20]. Berger argued early that domain-specific supervised learning from optimal decisions is lighter and more robust than generic reinforcement learning for networked caches [21].

4.8 The hardware lineage

Several of these ideas first appeared in processor caches. Hawkeye simulates Belady's policy on past accesses to learn which instructions tend to load lines worth keeping [22]. Glider trained a deep model offline to predict Belady's decisions, then used insights from that model to design a simple online predictor that runs at hardware speed [23]. The lesson carries over to software caches: a heavy model can guide the design of a cheap one, but rarely runs in the hot path itself.

5. Learned admission

Admission decides whether a new object enters the cache at all, and it matters most where space or writes are expensive. AdaptSize, an analytical rather than learned policy, tunes size-aware admission continuously with a Markov model for the small memory cache of a CDN [24], and TinyLFU admits a new item only if its estimated frequency, kept in a compact sketch, exceeds that of the eviction candidate [25]. These are the baselines learned admission must beat. Flash caches add device endurance to the objective. Flashield uses DRAM as a filter and a lightweight learned predictor to admit to flash only objects likely to be read often without being updated [26]. Baleen, evaluated on traces from Meta's storage clusters, learns admission and prefetching together to reduce peak load on backend disks [27]. Its authors describe early ML attempts that failed, a residency model they built to produce correct training labels, and a version that improved hit ratio while worsening the end-to-end metric they cared about, which led them to optimize disk-head time directly [27]. In their setting, gradient-boosted trees outperformed deeper models [27]. LHR also learns admission, pairing an auto-tuned, threshold-based admission model with a simple eviction rule and retraining only when it detects a workload change [5].

6. Prediction with guarantees

A learned policy can fail badly when its predictions are wrong, for example after a workload shift. Theoretical work on learning-augmented algorithms asks how to use predictions without losing worst-case guarantees. Lykouris and Vassilvitskii introduced the framework for caching: each request carries a prediction of the object's next request time, and their algorithm is near-optimal when predictions are good while staying within classical worst-case bounds when they are bad [28]. Rohatgi gave improved algorithms and nearly matching lower bounds on how performance must depend on prediction error [29]. Wei showed that combining a policy that simply follows the predictions with a robust classical algorithm gives a simpler design with

better trade-offs [30], and Antoniadis et al. extended robust use of predictions to a broader class of online problems that includes caching [31]. Chłędowski et al. tested these ideas on real caching data with learned predictors and found that a simple scheme, switching between the predictor-driven policy and a classical robust one whenever one falls behind, costs little when predictions are good and stays competitive when they fail [32]. The practical lesson matches the systems work in Section 4: keep a classical policy in the loop as a fallback.

7. Learned components in database buffer pools and storage

Database buffer pools are caches with their own constraints: pages are fixed-size, the database controls concurrency, and replacement decisions sit on the hot path of every query. Research has applied learning here too. Yuan and Jin proposed retraining a model periodically on historical page requests, as a hot-or-cold classifier or as a regressor, to adapt buffer replacement to workload change, evaluated in simulation [33]. Learning has also been applied to prefetching, the complement of eviction: Hashemi et al. framed memory prefetching as sequence prediction with recurrent neural networks [34], and Chen et al. treated next-page prediction for database prefetching as classification [35]. These works are research prototypes; the database buffer caches described in paper 1, such as InnoDB's midpoint-insertion LRU variant, use heuristic replacement. Production caches outside databases take the same pragmatic route. Redis approximates LRU and LFU by sampling a few keys rather than maintaining full orderings, uses a small probabilistic counter with decay for LFU, and lets operators trade CPU for accuracy by changing the sample size [36]. Any learned policy must beat such approximations on cost as well as on hit ratio.

8. Engineering a learned cache

8.1 Features and metadata

Features describe each entry's past: gaps between accesses, decayed counts, age, size and type [3]. More features improve predictions with diminishing returns, but each costs metadata memory per entry, which competes with the cache's own capacity [3], [10]. Grouping [6] and cache-level features [16] reduce this cost.

8.2 Labels and training

Labels come from the future, so learned caches must wait to learn. LRB labels a sampled object when its next request arrives, or as beyond the boundary when none arrives in time [3]; HALP labels each pairwise comparison by which object was accessed first [14]; Baleen needed a separate residency model to construct correct labels [27]. Training is usually online and periodic so that models follow the workload [3], [14], although Learning-Augmented Heuristics pre-trains a single model across many traces [16].

8.3 Inference cost and candidate sampling

The central obstacle to deployment is the cost of decisions. A cache serves requests at high rates, and a model consulted on every access, or one that scores many candidates per eviction, consumes CPU that the cache needs [10], [14]. The field has answered with sampling [3], [11], heuristic proposals of a few candidates [14], [15], biased sampling [13], grouping [6] and moving learning off the critical path [16].

8.4 Drift and retraining

Workloads change with time of day, releases and tenants. Online retraining lets a model follow them [3], [14], change detection retrains only when needed [5], and expert-based methods adapt by reweighting [7], [8]. Drift remains a risk: the Baleen authors note that learned policies in other settings tended to regress over time even with retraining [27].

8.5 Choosing the objective

Hit ratio is not always the objective that matters. Baleen's experience shows that a policy can raise hit ratio while worsening backend load [27], Flashield and Baleen weigh flash writes [26], [27], and Darwin can target different objectives such as hit rate or disk writes [9]. A learned policy optimizes what it is trained on, so the training target should be the end-to-end metric the system cares about.

8.6 Safe rollout and measurement

Production caches are noisy, so the effect of a policy change is hard to measure. HALP's authors introduced an analysis method for measuring an eviction change's effect under noisy production conditions [14]. Robust fallbacks [32], staged rollout and continuous comparison with a heuristic baseline are the practical safeguards.

8.7 Traces, tools and reproducibility

Public traces and shared tools make learned and heuristic policies comparable. Twitter released anonymized one-week traces from its cache clusters [1], and Wikimedia has published anonymized CDN request traces in several rounds, which LRB's evaluation used [37]. libCacheSim implements classic heuristics, FIFO-based designs, expert-based learners and learned policies in one harness, with LRB, GL-Cache and 3L-Cache available as optional builds [38]. LRB's simulator and prototype are open source [3].

9. How simple heuristics compare

Recent studies set a high bar for learned policies. The Twitter study found that FIFO performed as well as or better than LRU for many production workloads [1]. Yang et al. showed that FIFO-based designs with quick demotion can outperform LRU and several complex policies, while scaling better [39]. S3-FIFO's authors compared it directly with LRB on a sample of traces that included the Wikimedia traces from LRB's own evaluation and found similar efficiency, with S3-FIFO far simpler, and they report that a small FIFO used as a flash admission filter outperformed Flashield's learned filter in their tests on open CDN traces [40]. SIEVE, simpler than LRU, often matches or improves on more complex policies on web cache workloads and was adopted in several caching libraries [41]. W-TinyLFU remains a strong admission-plus-eviction baseline [25]. These results do not show that learning never helps: HALP's production deployment [14] and the gains reported for learned admission on flash [27] show where it can. They do show that a learned policy should be compared with the strongest simple policies, on the workload at hand, with its cost included.

10. Adjacent work in the AI era

10.1 KV-cache eviction in language-model serving

Language-model serving has its own eviction problem: the attention key-value (KV) cache grows with context length and batch size, and keeping all of it is often impossible. Several methods evict KV entries using signals from the model itself rather than a separately trained predictor. H2O keeps recent tokens together with "heavy hitters" that receive much of the attention [42]; Scissorhands relies on the observation that tokens that mattered at one step tend to keep mattering [43]; and SnapKV uses an observation window at the end of the prompt to decide which earlier positions to keep [44]. These are attention-guided heuristics, and they echo the recency and frequency signals of classic caches. Paper 3 of this series covers KV-cache reuse across requests.

10.2 Policies generated by language models

A newer direction uses language models to write cache policies rather than to make cache decisions. PolicySmith searches over code generated by a language model to synthesize heuristics specialized for a particular deployment, applied to web cache eviction on libCacheSim and to congestion control; its output is readable code with no per-request inference cost [45]. A recent preprint applies LLM-guided program evolution to discover KV-cache eviction scorers and reports reward hacking during the search as a failure mode to design against [46]. Generated policies inherit the advantages of heuristics, low cost and inspectable logic, while the search, not the policy, carries the learning.

11. Comparative analysis

Table 2 compares the approaches in this paper. The ratings are qualitative and relative to each other; actual behavior depends on the implementation, workload and hardware.

Approach	Decision cost per request	Adaptivity	Worst-case behavior	Interpretability	Production evidence in the cited work
Strong simple heuristic (S3-FIFO, SIEVE, W-TinyLFU)	Very low	Limited, by design	Predictable	High	Adoption in caching libraries
Online learning over experts	Low	Medium	Bounded by the experts	High	Prototypes
Object-level prediction	High	High	Can regress under drift	Low	Prototypes
Heuristic-aided learning	Low to Medium	High	Bounded by the heuristic's proposals	Medium	Production CDN deployment (HALP)
Group-level learning	Low	Medium to High	Coarser decisions	Medium	Implemented on an open-source production cache code base
Learning to tune a heuristic	Very low on the data path	Medium	Close to the underlying heuristic	High	Pre-trained on production traces

Approach	Decision cost per request	Adaptivity	Worst-case behavior	Interpretability	Production evidence in the cited work
Learned admission for flash	Low to Medium	High	Depends on the objective chosen	Medium	Evaluated on production traces
Prediction with a robust fallback	Depends on the predictor	High	Bounded by the fallback	Medium	Experimental study on real data

12. Open challenges and future opportunities

12.1 The cost of decisions

Challenge. A model that runs on every access or scores many candidates competes with the cache for CPU and memory [10], [14]. **Research direction.** Candidate filtering [14], [15], [13], grouping [6] and moving learning to a control path [16]. **Opportunity.** Learned policies whose data path is as cheap as a heuristic's would remove the main barrier to adoption.

12.2 Drift and robustness

Challenge. Trained models can regress when workloads change [27]. **Research direction.** Prediction-augmented algorithms with worst-case guarantees [28], [30], practical switching to a robust fallback [32], and change detection [5]. **Opportunity.** Learned caches that never do much worse than a strong heuristic would make learning a safe default rather than a gamble.

12.3 Choosing the right objective

Challenge. Optimizing hit ratio can worsen the metric that matters, such as backend load or flash wear [27]. **Research direction.** End-to-end objectives and multi-objective policies [27], [9]. **Opportunity.** Policies trained on cost, latency and device endurance together would align caching with platform goals.

12.4 Generalization across workloads and tenants

Challenge. A model trained on one workload may not transfer to another; some work reports transfer within a region [18], but generality is unproven. **Research direction.** Models pre-trained across many traces [16] and workload characterization [8]. **Opportunity.** Policies that ship pre-trained and adapt lightly would suit multi-tenant platforms.

12.5 Fair and reproducible evaluation

Challenge. Learned and heuristic policies have often been compared on different traces, with different baselines and without cost. **Research direction.** Shared simulators [38], public traces [1], [37], and head-to-head studies against strong simple baselines [40]. **Opportunity.** A common evaluation practice would show where learning helps and where it does not.

12.6 Interpretability and operability

Challenge. Operators must understand and trust a policy before running it on critical systems. **Research direction.** Learned parameters for simple policies [16], and generated policies expressed as readable code [45]. **Opportunity.** Policies whose decisions can be explained and reviewed would fit change-management practice.

12.7 Learned policies for database caches

Challenge. Database buffer pools and query caches have strict concurrency and latency constraints, and published learned approaches remain prototypes [33], [35]. **Research direction.** Lightweight learned replacement and prefetching designed for database engines. **Opportunity.** Engines that adapt replacement and prefetching to their workload without tuning would reduce operational effort.

12.8 Language models as policy designers

Challenge. Policies generated by language models can overfit to the traces used in the search, and search loops can be gamed [46]. **Research direction.** Program search with robust evaluation across traces [45], [46]. **Opportunity.** Generated, inspectable heuristics specialized to a deployment could combine the cost profile of heuristics with the adaptivity of learning.

13. What this means for platform decisions

Start with a strong simple policy and measure it: approximate LRU or LFU as provided by the cache, W-TinyLFU where available, or FIFO-based designs such as S3-FIFO and SIEVE. Record access traces and build miss-ratio curves, as paper 1 describes, so that any change can be evaluated offline first. Consider learning where the stakes justify it: very large caches, flash tiers where admission decides device wear, or workloads that change in ways a fixed policy cannot follow. When adopting a learned policy, prefer designs that keep a heuristic in the loop or tune a heuristic off the critical path, train on the end-to-end objective, keep a robust fallback, and roll out with measurement that can detect regressions.

For in-memory caching, Bassirah works with Redis as a Strategic Technology Partner; Redis's sampled LRU and LFU policies, with tunable sample size and decay, are the kind of efficient approximation that learned policies must outperform on cost as well as hit ratio. Other products and projects named in this paper are ecosystem references. In each case, validate the policy on representative traces, workloads and versions before relying on it.

References

1. Yang, J., Yue, Y. and Rashmi, K. V. (2020). A large scale analysis of hundreds of in-memory cache clusters at Twitter. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 191–208.
(<https://www.usenix.org/conference/osdi20/presentation/yang>)
2. Berger, D. S., Beckmann, N. and Harchol-Balter, M. (2018). Practical Bounds on Optimal Caching with Variable Object Sizes. Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS), 2(2), 1-38 (presented at SIGMETRICS 2018).
(<https://doi.org/10.1145/3224427>)
3. Song, Z., Berger, D. S., Li, K. and Lloyd, W. (2020). Learning Relaxed Belady for Content Distribution Network Caching. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 529-544.
(<https://www.usenix.org/conference/nsdi20/presentation/song>)
4. Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2), 78-101. (<https://doi.org/10.1147/sj.52.0078>)

5. Yan, G., Li, J. and Towsley, D. (2021). Learning from optimal caching for content delivery. Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2021), 344-358. (<https://doi.org/10.1145/3485983.3494855>)
6. Yang, J., Mao, Z., Yue, Y. and Rashmi, K. V. (2023). GL-Cache: Group-level learning for efficient and high-performance caching. 21st USENIX Conference on File and Storage Technologies (FAST 23), 115-134. (<https://www.usenix.org/conference/fast23/presentation/yang-juncheng>)
7. Vietri, G., Rodriguez, L. V., Martinez, W. A., Lyons, S., Liu, J., Rangaswami, R., Zhao, M. and Narasimhan, G. (2018). Driving Cache Replacement with ML-based LeCaR. 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18). (<https://www.usenix.org/conference/hotstorage18/presentation/vietri>)
8. Rodriguez, L. V., Yusuf, F., Lyons, S., Paz, E., Rangaswami, R., Liu, J., Zhao, M. and Narasimhan, G. (2021). Learning Cache Replacement with CACHEUS. 19th USENIX Conference on File and Storage Technologies (FAST 21), 341-354. (<https://www.usenix.org/conference/fast21/presentation/rodriguez>)
9. Chen, J., Sharma, N., Khan, T., Liu, S., Chang, B., Akella, A., Shakkottai, S. and Sitaraman, R. K. (2023). Darwin: Flexible Learning-based CDN Caching. Proceedings of the ACM SIGCOMM 2023 Conference, 981-999. (<https://doi.org/10.1145/3603269.3604863>)
10. Shahout, R. and Friedman, R. (2023). Sketching the Path to Efficiency: Lightweight Learned Cache Replacement. 27th International Conference on Principles of Distributed Systems (OPODIS 2023), LIPIcs 286, 34:1–34:21. (<https://doi.org/10.4230/LIPIcs.OPODIS.2023.34>)
11. Beckmann, N., Chen, H. and Cidon, A. (2018). LHD: Improving Cache Hit Rate by Maximizing Hit Density. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), 389-403. (<https://www.usenix.org/conference/nsdi18/presentation/beckmann>)
12. Hu, X., Ramadan, E., Ye, W., Tian, F. and Zhang, Z.-L. (2022). Raven: Belady-Guided, Predictive (Deep) Learning for In-Memory and Content Caching. Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2022), 72-90. (<https://doi.org/10.1145/3555050.3569134>)
13. Zhou, W., Niu, Z., Xiong, Y., Fang, J. and Wang, Q. (2025). 3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches. 23rd USENIX Conference on File and Storage Technologies (FAST 25), 237-254. (<https://www.usenix.org/conference/fast25/presentation/zhou-wenbin>)
14. Song, Z., Chen, K., Sarda, N., Altınbüken, D., Brevdo, E., Coleman, J., Ju, X., Jurczyk, P., Schooler, R. and Gummadi, R. (2023). HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), 1149-1163. (<https://www.usenix.org/conference/nsdi23/presentation/song-zhenyu>)
15. Yang, D., Berger, D. S., Li, K. and Lloyd, W. (2023). A Learned Cache Eviction Framework with Minimal Overhead. arXiv preprint arXiv:2301.11886. (<https://arxiv.org/abs/2301.11886>)
16. Xia, H., Nixon, W., Marthen, B. D., Bhandari, P. and Yang, J. (2026). Learning-Augmented Heuristics: Simple Yet Smart, Robust and Interpretable Cache Eviction. 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26), 2241-2259. (<https://www.usenix.org/conference/osdi26/presentation/xia>)
17. Liu, E., Hashemi, M., Swersky, K., Ranganathan, P. and Ahn, J. (2020). An Imitation Learning Approach for Cache Replacement. Proceedings of the 37th International Conference on

- Machine Learning (ICML 2020), PMLR 119, 6237-6247.
(<https://proceedings.mlr.press/v119/liu20f.html>)
18. Kirilin, V., Sundarajan, A., Gorinsky, S. and Sitaraman, R. K. (2020). RL-Cache: Learning-Based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications*, 38(10), 2372–2385. (<https://doi.org/10.1109/JSAC.2020.3000415>)
 19. Wang, H., He, H., Alizadeh, M. and Mao, H. (2019). Learning Caching Policies with Subsampling. Workshop paper, NeurIPS 2019 (Machine Learning for Systems workshop; venue inferred, see note). (<https://people.csail.mit.edu/hehaodele/projects/NIPS19-cache/paper.pdf>)
 20. Zhou, Y., Wang, F., Shi, Z. and Feng, D. (2024). An Efficient Deep Reinforcement Learning-Based Automatic Cache Replacement Policy in Cloud Block Storage Systems. *IEEE Transactions on Computers*, 73(1), 164–177. (<https://doi.org/10.1109/TC.2023.3325625>)
 21. Berger, D. S. (2018). Towards Lightweight and Robust Machine Learning for CDN Caching. *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets 2018)*, 134-140. (<https://doi.org/10.1145/3286062.3286082>)
 22. Jain, A. and Lin, C. (2016). Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 78-89. (<https://doi.org/10.1109/ISCA.2016.17>)
 23. Shi, Z., Huang, X., Jain, A. and Lin, C. (2019). Applying Deep Learning to the Cache Replacement Problem. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 413-425. (<https://doi.org/10.1145/3352460.3358319>)
 24. Berger, D. S., Sitaraman, R. K. and Harchol-Balter, M. (2017). AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, 483–498. (<https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger>)
 25. Einziger, G., Friedman, R. and Manes, B. (2017). TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4), 1–31. (<https://doi.org/10.1145/3149371>)
 26. Eisenman, A., Cidon, A., Pergament, E., Haimovich, O., Stutsman, R., Alizadeh, M. and Katti, S. (2019). Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 65–78. (<https://www.usenix.org/conference/nsdi19/presentation/eisenman>)
 27. Wong, D. L.-K., Wu, H., Molder, C., Gunasekar, S., Lu, J., Khandkar, S., Sharma, A., Berger, D. S., Beckmann, N. and Ganger, G. R. (2024). Baleen: ML Admission & Prefetching for Flash Caches. *22nd USENIX Conference on File and Storage Technologies (FAST '24)*, 347–371. (<https://www.usenix.org/conference/fast24/presentation/wong>)
 28. Lykouris, T. and Vassilvitskii, S. (2018). Competitive Caching with Machine Learned Advice. *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, PMLR 80, 3296-3305. (<https://proceedings.mlr.press/v80/lykouris18a.html>). Journal version: Lykouris, T. and Vassilvitskii, S. (2021). Competitive Caching with Machine Learned Advice. *Journal of the ACM*, 68(4), 1-25. (<https://doi.org/10.1145/3447579>)
 29. Rohatgi, D. (2020). Near-Optimal Bounds for Online Caching with Machine Learned Advice. *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, 1834-1845. (<https://doi.org/10.1137/1.9781611975994.112>)
 30. Wei, A. (2020). Better and Simpler Learning-Augmented Online Caching. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*

- (APPROX/RANDOM 2020), LIPIcs vol. 176, 60:1-60:17.
(<https://doi.org/10.4230/LIPIcs.APPROX/RANDOM.2020.60>)
31. Antoniadis, A., Coester, C., Elias, M., Polak, A. and Simon, B. (2020). Online metric algorithms with untrusted predictions. Proceedings of the 37th International Conference on Machine Learning (ICML 2020), PMLR 119, 345-355.
(<https://proceedings.mlr.press/v119/antoniadis20a.html>)
 32. Chłędowski, J., Polak, A., Szabucki, B. and Żotna, K. T. (2021). Robust Learning-Augmented Caching: An Experimental Study. Proceedings of the 38th International Conference on Machine Learning (ICML 2021), PMLR 139, 1920-1930.
(<https://proceedings.mlr.press/v139/chledowski21a.html>)
 33. Yuan, Y. and Jin, P. (2022). Learned Buffer Replacement for Database Systems. Proceedings of the 5th International Conference on Data Storage and Data Engineering (DSDE 2022), ACM, 18-25. (<https://doi.org/10.1145/3528114.3528118>)
 34. Hashemi, M., Swersky, K., Smith, J., Ayers, G., Litz, H., Chang, J., Kozyrakis, C. and Ranganathan, P. (2018). Learning Memory Access Patterns. Proceedings of the 35th International Conference on Machine Learning (ICML 2018), PMLR 80, 1919-1928.
(<https://proceedings.mlr.press/v80/hashemi18a.html>)
 35. Chen, Y., Zhang, Y., Wu, J., Wang, J. and Xing, C. (2021). Revisiting Data Prefetching for Database Systems with Machine Learning Techniques. 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2165-2170.
(<https://doi.org/10.1109/ICDE51399.2021.00218>)
 36. Redis (n.d.). Key eviction. Redis documentation.
(<https://redis.io/docs/latest/develop/reference/eviction/>)
 37. Wikimedia Foundation (n.d.). Data Platform/Data Lake/Traffic/Caching. Wikitech documentation page (releases dated 2008, 2016 and 2019).
(https://wikitech.wikimedia.org/wiki/Data_Platform/Data_Lake/Traffic/Caching)
 38. Yang, J. et al. (n.d., continuously updated). libCacheSim: a high-performance library for building and running cache simulations. GitHub repository.
(<https://github.com/1a1a11a/libCacheSim>)
 39. Yang, J., Qiu, Z., Zhang, Y., Yue, Y. and Rashmi, K. V. (2023). FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23), ACM, 70-79.
(<https://doi.org/10.1145/3593856.3595887>)
 40. Yang, J., Zhang, Y., Qiu, Z., Yue, Y. and Rashmi, K. V. (2023). FIFO queues are all you need for cache eviction. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), ACM, 130-149. (<https://doi.org/10.1145/3600006.3613147>)
 41. Zhang, Y., Yang, J., Yue, Y., Vigfusson, Y. and Rashmi, K. V. (2024). SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24), 1229-1246.
(<https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo>)
 42. Zhang, Z., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C. et al. (2023). H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models. Advances in Neural Information Processing Systems 36 (NeurIPS 2023).
(https://proceedings.neurips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfceb2827f8-Abstract-Conference.html)

43. Liu, Z., Desai, A., Liao, F., Wang, W., Xie, V., Xu, Z., Kyrillidis, A. and Shrivastava, A. (2023). Scissorhands: Exploiting the Persistence of Importance Hypothesis for LLM KV Cache Compression at Test Time. Advances in Neural Information Processing Systems 36 (NeurIPS 2023).
(https://proceedings.neurips.cc/paper_files/paper/2023/hash/a452a7c6c463e4ae8fdbc14c6e983e6-Abstract-Conference.html)
44. Li, Y., Huang, Y., Yang, B., Venkitesh, B., Locatelli, A., Ye, H., Cai, T., Lewis, P. and Chen, D. (2024). SnapKV: LLM Knows What You are Looking for Before Generation. Advances in Neural Information Processing Systems 37 (NeurIPS 2024).
(https://proceedings.neurips.cc/paper_files/paper/2024/hash/28ab418242603e0f7323e54185d19bde-Abstract-Conference.html)
45. Dwivedula, R., Saxena, D., Akella, A., Chaudhuri, S. and Kim, D. (2025). Man-Made Heuristics Are Dead. Long Live Code Generators!. Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets '25), 51–60. (<https://doi.org/10.1145/3772356.3772413>; preprint arXiv preprint 2510.08803)
46. Poudel, P., Wu, Y., Jha, S. and Liu, J. (2026). Discovering KV Cache Eviction Policies via LLM-Guided Program Evolution. arXiv preprint 2608.14555. (<https://arxiv.org/abs/2608.14555>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.