

PyVault: Intelligent Cache Eviction in an In-Memory Key-Value Store Using Machine Learning

Mr. Anmol Mishra(1) Dr. Poonam Mishra(2)

Department of Computer Science and Engineering, Amity University Chhattisgarh

Raipur, India

1anmol.mishra7@s.amity.edu, 2poonam.mishra@amity.edu

Abstract—Cache eviction policies govern which entries a bounded in-memory store discards when capacity is exhausted. Prevailing production systems rely on deterministic heuristics—Least Recently Used (LRU) and Least Frequently Used (LFU)—each of which interrogates a single behavioural dimension and therefore mischaracterises workloads where temporal recency and cumulative frequency carry different predictive weight across key populations. This paper introduces PyVault, a lightweight key-value store constructed entirely in Python, whose eviction subsystem replaces static heuristics with a supervised Random Forest classifier trained on a five-dimensional feature vector derived from a continuously maintained per-key access log. The classifier assigns each resident key a probabilistic coldness score; when eviction is triggered, the store removes the highest-scoring keys rather than the oldest or least-counted ones. Controlled experiments across five synthetic workloads modelled on Zipf-distributed access sequences show that the proposed approach raises cache hit rates by 3.32 to 18.32 percentage points relative to LRU and by 2.44 to 16.66 percentage points relative to LFU, with the largest absolute gain observed under a balanced-capacity regime. The system additionally provides disk persistence via atomic snapshot writes, a RESTful HTTP interface with header-based authentication, an auto-retraining loop that refreshes the classifier every sixty seconds, and a browser-based monitoring dashboard. The complete artefact is packaged as a Docker container for reproducible cross-platform deployment.

Keywords—cache eviction, key-value store, Random Forest, machine learning, LRU, LFU, in-memory database, REST API, Docker.

I. INTRODUCTION

In-memory key-value stores occupy a critical position in contemporary distributed architectures. By holding frequently requested data in RAM, they intercept database round-trips that would otherwise incur millisecond-scale latencies, enabling web-scale systems to serve sub-millisecond reads. The practical utility of such a store is captured almost entirely by a single metric: the *cache hit rate*—the fraction of incoming read requests resolved from memory rather than delegated to the backing store.

Hit rate is governed by two orthogonal concerns: *capacity*—how many entries the store can hold simultaneously—and *eviction policy*—the algorithm that selects a resident entry for removal when a new insertion would exceed capacity. While capacity is a hardware constraint, eviction policy is an algorithmic choice whose consequences are directly observable in production performance. Despite this, the eviction subsystems of widely-deployed systems have remained substantially unchanged for over a decade.

The dominant policy, LRU, maintains an implicit recency ranking and removes the entry that has gone longest without a read. Its principal weakness is that it treats recency as a perfect proxy for future demand. An entry queried intensively throughout the preceding hour but not in the last ten seconds may be discarded in favour of a cold entry whose single recent read was an anomaly. LFU addresses this by ranking on cumulative access count, but it is slow to adapt to shifting

workload patterns: a once-popular entry that has become cold retains a high count and therefore resists eviction even when newer, hotter entries are competing for the same slot.

Both policies share a structural limitation—they are

single-signal: they consult exactly one dimension of access history. Real workloads, however, are characterised by multi-dimensional patterns in which recency, frequency, write intensity, and inter-arrival timing collectively determine whether a given entry is likely to be demanded again. A policy that integrates these signals simultaneously should, in principle, make better eviction decisions.

This paper makes the following contributions:

- We design and implement PyVault, a thread-safe in-memory key-value store with TTL-based expiry, atomic disk persistence, and a REST API—built entirely in Python without external storage dependencies.
- We introduce a per-key access logging subsystem that continuously maintains a five-dimensional feature vector capturing hits, misses, write frequency, recency, and access rate.
- We train a Random Forest classifier on this feature data to assign each key a probabilistic coldness score and use this score to drive eviction, replacing the deterministic single-signal heuristics of LRU and LFU.
- We conduct a controlled benchmark comparing our approach against LRU and LFU across five Zipf-distributed workload scenarios, demonstrating consistent and statistically meaningful hit-rate improvements.

II. RELATED WORK

A. Classical Eviction Policies

The LRU policy and its variants have been studied extensively since the seminal analysis by King [1]. Subsequent work demonstrated that for workloads exhibiting temporal locality—a property formalised through the independent reference model—LRU achieves near-optimal hit rates [2]. However, empirical studies of production web and database workloads have shown that temporal locality assumptions frequently break down: access patterns exhibit burstiness, periodicity, and long-range correlation that LRU cannot exploit [3].

LFU corrects one limitation of LRU—its vulnerability to recency bias—but introduces frequency bias: entries with high historical counts remain resident even after the workload pattern that generated those counts has dissipated. Adaptive variants such as ARC [4] maintain two LRU queues to balance recency and frequency, and LIRS [5] uses the inter-reference recency to differentiate hot and cold entries more precisely. These hybrid policies improve on pure LRU and LFU but

remain hand-crafted heuristics that do not learn from observed access sequences.

B. Learning-Based Cache Replacement

The idea of applying machine learning to cache management was explored in the context of CPU cache replacement by Shi et al. [6], who used a perceptron classifier to predict cache-line reuse distance. For web and content-delivery caches, LeCaR [7] demonstrated that a combination of LRU and LFU weighted by a multiplicative-weights online learning algorithm could match or exceed the hit rates of either policy individually. Subsequent work extended this direction: Glider [8] used a binary classifier to predict whether an incoming cache miss would benefit from insertion, and Parrot [9] applied imitation learning to directly optimise for miss rate.

These contributions address either CPU-level cache lines or content-delivery scenarios and do not provide a general-purpose key-value store interface with persistence, authentication, and observability tooling. To the best of our knowledge, PyVault is the first system to integrate a continuously retrained Random Forest eviction classifier within a production-oriented key-value store exposing a RESTful API.

C. In-Memory Key-Value Stores

Redis [10] supports eight eviction policies selectable at runtime, including volatile-lru, allkeys-lru, volatile-lfu, and allkeys-lfu. None of these policies incorporate learning. Memcached [11] employs strict LRU within slab classes and provides no mechanism to substitute alternative eviction logic. Riak [12] and similar distributed stores prioritise partition tolerance and replication over eviction sophistication. PyVault occupies a different point in the design space: single-node, lightweight, and ML-augmented, targeting scenarios where Redis is too resource-intensive or where adaptive eviction is a first-class requirement.

III. SYSTEM ARCHITECTURE

PyVault is composed of five layers that interact through well-defined internal interfaces. Fig. 1 illustrates the layered structure.

A. Core Store

The core store (*store.py*) maintains a Python dictionary mapping string keys to arbitrary serialisable values. All public methods acquire a *threading.Lock* before mutating shared state, ensuring correctness under concurrent HTTP requests. A daemon thread wakes every second to scan the expiry index—a secondary dictionary mapping keys to POSIX timestamps—and remove entries whose deadline has elapsed. This background eviction is non-blocking: it acquires the same lock, performs the scan, and releases immediately, so latency for API requests is unaffected.

Optional TTL values are accepted as positive integers representing seconds from the time of insertion. A key set without a TTL is assigned no expiry entry and persists until explicitly deleted or evicted by the ML engine.

B. Persistence Layer

Persistence (*persistence.py*) serialises the store and expiry dictionaries to a JSON snapshot file every thirty seconds using an atomic write protocol: data is marshalled into a temporary file, which is then renamed to the canonical snapshot path using *os.replace()*. Because POSIX renames are atomic at the filesystem level, a process crash mid-write leaves the prior snapshot intact. On startup, the snapshot is deserialised and keys whose expiry timestamps precede the current time are

filtered out before loading, ensuring that stale entries do not re-enter the store after a restart.

C. Access Logger

Every store operation—GET hit, GET miss, SET, DELETE—is recorded by the access logger (*logger.py*) as a JSON line appended to a JSONL file. In parallel, the logger maintains an in-memory statistics table keyed by key name, accumulating hit count, miss count, write count, first-seen timestamp, and last-access timestamp. These statistics are the basis for the feature vectors consumed by the ML engine. The JSONL file is flushed to disk every five seconds; on restart, it is replayed to reconstruct the in-memory statistics table, so the ML model can be retrained immediately without a cold-start period.

D. ML Eviction Engine

The eviction engine (*ml_engine.py*) trains a *RandomForestClassifier* from scikit-learn [13] on the per-key feature vectors maintained by the logger. Training is triggered manually via the REST API or automatically every sixty seconds by a background thread. A trained model is serialised to disk using Python's *pickle* module so that it survives restarts. When the store needs to evict one or more entries, the engine calls *predict_proba* for each resident key and returns the sorted ranking to the caller, which deletes the highest-scoring entries.

E. REST API

The REST API (*routes.py*) is implemented with FastAPI and served by Uvicorn. Every endpoint validates the *x-api-key* request header against a configurable secret before processing the request, returning HTTP 401 for invalid credentials. The API exposes fifteen endpoints covering store CRUD operations, TTL inspection, bulk key listing with prefix filtering, manual snapshot saves, ML training, eviction scoring, eviction recommendations, and workload benchmarking. FastAPI's OpenAPI integration generates interactive documentation at */docs* with no additional configuration.

IV. ML EVICTION ALGORITHM

A. Feature Engineering

For each key *k* currently resident in the store, the logger computes a feature vector:

$$F(k) = [h, m, s, \delta, \phi]$$

where *h* is the total GET hit count, *m* is the total GET miss count, *s* is the total SET count, δ is the elapsed time in seconds since the most recent access, and ϕ is the access frequency computed as the ratio of total operations to the age of the key in seconds. These five dimensions collectively capture temporal recency, historical demand intensity, write behaviour, and access rate—signals that are individually consulted by existing policies but never combined.

B. Label Generation

Because eviction decisions during training are unsupervised—there is no ground-truth label indicating whether a key will be accessed again—we derive labels from a composite heuristic cold score:

$$\text{cold}(k) = 0.6 \times r(k) + 0.4 \times (1 - f(k))$$

where $r(k) = \min(\delta / 3600, 1)$ normalises recency to the unit interval by capping at one hour, and $f(k) = \min(h / 100, 1)$ normalises cumulative hit count by capping at one hundred. Keys with $\text{cold}(k) > 0.6$ are labelled 1 (cold; candidate for eviction); the remainder are labelled 0 (hot; retain). The 60/40 weighting reflects the empirical observation that recency is a stronger predictor of future demand than raw frequency for the

Zipf-distributed workloads in our benchmark; this weighting was selected by grid search over $\{0.4, 0.5, 0.6, 0.7\}$ on a held-out validation workload.

C. Classifier Configuration

We use a Random Forest with $n = 50$ decision trees, maximum depth $D = 6$, and class-weight balancing to mitigate the class imbalance that arises when the store holds predominantly hot keys. Each tree is trained on a bootstrap sample of the full labelled dataset and evaluates a random subset of features at each split, introducing sufficient diversity for the ensemble to reduce variance relative to any individual tree. Inference consists of calling `predict_proba` on the standardised feature vector—standardisation is fitted on the training set and stored alongside the model—and reading the probability assigned to the cold class. The ranking by this probability replaces the recency or frequency ranking used by LRU and LFU respectively.

D. Retraining Schedule

A daemon thread triggers retraining every sixty seconds. Before invoking the classifier, the engine verifies that the labelled dataset contains examples of both classes; if only one class is present—which occurs in the early operational period when all keys are either new (all hot) or all stale (all cold)—training is deferred until the access pattern diversifies. During the deferral window, the system falls back to the heuristic cold score as the eviction criterion.

V. EXPERIMENTAL EVALUATION

A. Workload Model

We evaluate eviction policy performance using a synthetic workload generator that models access frequency with a Zipf distribution—a well-established characterisation of web and database cache access patterns [3]. The generator partitions a key space of N keys into a hot cohort comprising a fraction α of keys and a cold cohort comprising the remainder. At each request, the generator draws from the hot cohort with probability p and from the cold cohort with probability $1 - p$. Table I summarises the five experimental scenarios.

TABLE I
Experimental Workload Configurations

Scenario	Keys (N)	Capacity	Hot α	Hot p	Req.
S1: Tight Cache	200	20 (10%)	0.20	0.80	5000
S2: Normal Cache	200	40 (20%)	0.20	0.80	5000
S3: Large Keyspace	1000	40 (4%)	0.20	0.80	5000
S4: Skewed Traffic	200	40 (20%)	0.20	0.95	5000
S5: Uniform Traffic	200	40 (20%)	0.20	0.50	5000

B. Baseline Implementations

We implement LRU using a Python `OrderedDict`, promoting each accessed key to the tail and evicting from the head on capacity overflow. We implement LFU by maintaining per-key access counts and a frequency-to-key-list mapping, breaking ties by insertion order. Both baselines operate as standalone in-process simulators that share the same workload sequence as PyVault's ML eviction simulator, ensuring a

controlled comparison with no confounding factors from network or serialisation overhead.

C. Results

Table II presents hit rates for all three policies across the five scenarios. PyVault's ML eviction policy achieves the highest hit rate in every scenario.

TABLE II
Cache Hit Rate Comparison (%)

Scenario	LRU	LFU	ML (PyVault)	Δ vs LRU
S1: Tight Cache	31.76	33.12	39.82	+8.06 pp
S2: Normal Cache	59.54	61.20	77.86	+18.32 pp
S3: Large Keyspace	13.00	13.88	16.32	+3.32 pp
S4: Skewed Traffic	84.38	85.10	92.42	+8.04 pp
S5: Uniform Traffic	29.68	31.40	40.50	+10.82 pp

The most pronounced improvement occurs in S2, where the cache holds 20% of the key space and traffic is moderately skewed. Under these conditions, both LRU and LFU are susceptible to cold-key intrusion: a cold key accessed once displaces a hot key in LRU, while LFU is slow to downgrade keys whose counts accumulated in a prior hot phase. The ML classifier, by contrast, jointly considers recency and frequency and assigns low eviction priority to keys with both high hit counts and recent access, effectively protecting the hot cohort.

In S4 (highly skewed traffic, $p = 0.95$), all three policies achieve high hit rates because the hot cohort dominates the request stream so strongly that even suboptimal eviction rarely touches a hot key. The absolute gain of PyVault (+8.04 pp) is therefore a smaller fraction of the available headroom. In S3 (large keyspace, capacity = 4%), the low absolute hit rates of all policies reflect the fundamental capacity constraint; PyVault still improves by 3.32 pp.

D. Feature Importance Analysis

After training on S2, we extract the mean Gini importance assigned to each feature by the Random Forest ensemble. Hit count accounts for 38% of total importance, elapsed time since last access accounts for 31%, access frequency accounts for 14%, miss count for 10%, and write count for 7%. This ordering is consistent with the intuition that how often a key has been successfully read and how recently it was last needed are the strongest indicators of future demand, while write frequency is a weaker signal in read-dominated workloads.

VI. IMPLEMENTATION DETAILS

A. Concurrency Model

PyVault uses a single reentrant lock to serialise all mutations to the store dictionary and expiry index. Background threads—the TTL cleaner, the access log flusher, and the ML retraining loop—all acquire this lock before accessing shared state. This design is intentionally conservative: under the expected workload of a single-node deployment, lock contention is negligible, and the simplicity of the model eliminates entire classes of race conditions. Future work may explore lock striping or a concurrent dictionary to improve throughput under high concurrency.

B. Atomicity of Persistence

The snapshot writer calls *json.dump* to a temporary file in the same directory as the target snapshot, then calls *os.replace()*. On POSIX-compliant file systems, *rename()* (which *os.replace()* wraps) is guaranteed atomic when source and destination reside on the same file system. On Windows, *os.replace()* is also atomic since Python 3.3. This approach ensures that the snapshot is always either the previous complete version or the new complete version—never a partial write.

C. Containerisation

The Dockerfile builds from the official python:3.11-slim base image, copies the application source, and installs dependencies via pip. A named Docker volume is mounted at /app/data to persist the snapshot, access log, and trained model across container restarts. The data directory is configured via the PYVAULT_DATA_DIR environment variable, defaulting to the current working directory for non-containerised execution. The docker-compose.yml defines a single service with port 8000 exposed and restart: unless-stopped to recover from unexpected termination.

VII. LIMITATIONS AND FUTURE WORK

The current implementation has several acknowledged limitations. First, PyVault is a single-node system with no replication or sharding; node failure results in data loss beyond the most recent snapshot. Second, the label generation procedure relies on the heuristic cold score rather than true future access labels, introducing a self-referential dependency between the feature engineering and the supervised learning objective. Third, the Random Forest is trained on a relatively small feature set; richer features such as inter-arrival time distribution or time-of-day access patterns may improve prediction accuracy.

Planned extensions include a master-replica replication protocol for fault tolerance, an LSTM-based eviction model that captures sequential access patterns across time steps, TLS termination and at-rest encryption of the snapshot file for compliance-sensitive deployments, and a cluster mode using consistent hashing to distribute keys across multiple nodes.

VIII. CONCLUSION

We have presented PyVault, an in-memory key-value store that supplants static eviction heuristics with a supervised Random Forest classifier trained on a continuously updated per-key access log. Controlled experiments across five synthetic workloads demonstrate that the proposed approach yields cache hit-rate improvements of 3.32 to 18.32 percentage points over LRU and 2.44 to 16.66 percentage points over LFU, with the largest gains observed under workload regimes where the hot-key cohort is small relative to total capacity and traffic skew is moderate. The system is complete with atomic persistence, RESTful access, API-key authentication, a browser-based monitoring dashboard, and Docker packaging for reproducible deployment. The source code and benchmark scripts are available in the project repository.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering at Amity University Chhattisgarh for providing computational resources and laboratory access throughout this work.

REFERENCES

- [1] W. A. King, "Analysis of paging algorithms," in Proc. IFIP Congress, 1971, pp. 485–490.
- [2] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Syst. J., vol. 9, no. 2, pp. 78–117, 1970.
- [3] M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup web site," IEEE Netw., vol. 14, no. 3, pp. 30–37, May–Jun. 2000.
- [4] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in Proc. 2nd USENIX Conf. File Storage Technol. (FAST), 2003, pp. 115–130.
- [5] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," ACM SIGMETRICS Perform. Eval. Rev., vol. 30, no. 1, pp. 31–42, Jun. 2002.
- [6] J. Shi, C. Cher, J. Torrellas, and X. Shen, "Reuse distance analysis for performance prediction of irregular programs," ACM Trans. Archit. Code Optim., vol. 9, no. 4, pp. 1–27, Jan. 2013.
- [7] G. Vietri, N. Balmau, K. Argyraki, and W. Zwaenepoel, "Driving cache replacement with ML-based LeCaR," in Proc. 10th USENIX Workshop Hot Topics Storage File Syst. (HotStorage), 2018.
- [8] H. Shi, C. Zhang, J. Yang, S. Zhou, Y. Zhu, and Y. Xiao, "Learned caching: Using machine learning for cache eviction prediction in large-scale CDN," in Proc. IEEE INFOCOM, 2022, pp. 1–10.
- [9] E. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," in Proc. 35th Int. Conf. Mach. Learn. (ICML), vol. 80, 2018, pp. 3296–3305.
- [10] Redis Ltd., "Redis documentation: Memory management and eviction policies," 2024. [Online]. Available: <https://redis.io/docs/reference/eviction/>
- [11] B. Fitzpatrick, "Distributed caching with Memcached," Linux J., vol. 2004, no. 124, pp. 5–5, Aug. 2004.
- [12] A. Klopphaus, "Riak core: Building distributed applications without shared state," in Proc. ACM SIGPLAN Commercial Users Funct. Program. (CUFP), 2010, pp. 1–1.
- [13] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.