Distributed Raft Cache

Consensus-Driven Caching with ML-Based Eviction

Jahnavi Kedia   FNU Shamathmika   CMPE 273

Problem & Solution

The Problem

- Single-node caches fail completely, leading to **service outages**
- Network partitions cause **data inconsistency** across replicas
- LRU eviction **doesn't adapt** to complex access patterns
- Difficult trade-off: **Speed vs Consistency vs Fault-tolerance**

Our Solution

01

Raft Consensus

Strong consistency guarantees across distributed nodes

02

Read Lease

105x faster reads by bypassing log replication

03

ML Eviction

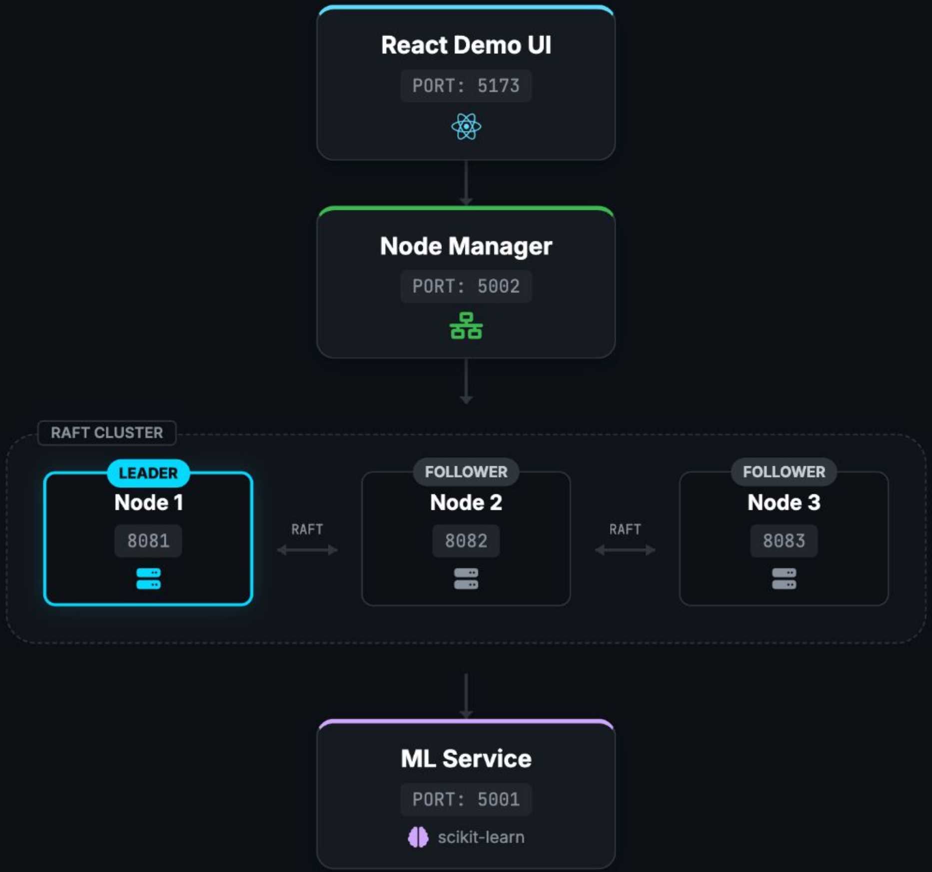
Smart cache eviction using Random Forest prediction

04

Demo UI

Real-time visualization of cluster state & elections

System Architecture



Tech Stack

- Java 17
- Netty
- Flask
- React
- scikit-learn

Raft Consensus Algorithm

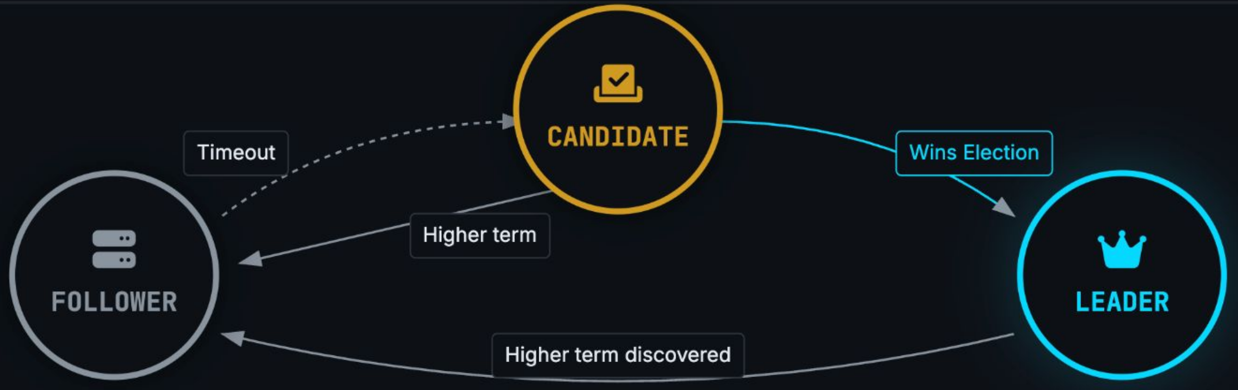
Primary Goals

- Enable multiple servers to agree on shared state reliably
- Ensure system functionality even if minority of servers fail
- Guarantee strong data consistency across the cluster

Three Sub-problems

1. **Leader Election**
   Selecting a coordinator to manage the log replication
2. **Log Replication**
   Accepting entries and replicating them across the cluster
3. **Safety**
   Ensuring state machines apply entries in the same order

Three States of a Raft Node



STATE	ROLE
FOLLOWER	Passive - responds to requests from leaders and candidates
CANDIDATE	Active - seeking votes from other nodes to become leader
LEADER	Dominant - handles all client read/write requests & replicates log

Leader Election Algorithm

raft_election.log

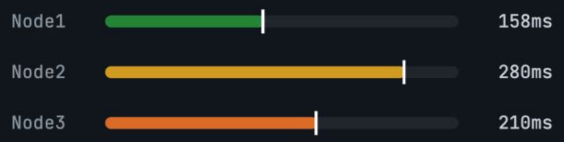
```
LEADER ELECTION PROTOCOL

// 1. Initial State
Follower waits for heartbeat from Leader

// 2. On Timeout (150-300ms randomized)
if (timeout_elapsed) {
 state = CANDIDATE;
 term++;
 voteFor(SELF);
 requestVotes(ALL_NODES);
}

// 3. Election Result
if (votes > majority) {
 state = LEADER;
} else if (higher_term_seen) {
 state = FOLLOWER;
}
```

RANDOMIZED TIMEOUTS



Different timeout durations ensure one node triggers election before others, reducing collision probability.

KEY INSIGHT

"Randomized timeout prevents split votes"

Log Replication Algorithm

REPLICATION_PROTOCOL </>

1. Client sends write request to Leader
2. Leader appends entry to local log
3. Leader sends AppendEntries RPC to followers
4. Wait for majority confirmation
5. Mark entry as COMMITTED
6. Apply command to state machine
7. Respond success to client

LEADER LOG STATE

INDEX	TERM	COMMAND	STATUS
1	T1	x ← 1	Committed
2	T1	y ← 2	Committed
3	T2	z ← 3	Committed
4	T2	w ← 4	*Replicating...*

commitIndex = 3

Entry 4 is currently being replicated to followers. It will be marked committed once a majority acknowledge receipt.

Raft Safety Guarantees

Five fundamental properties ensuring system correctness

Election Safety

At most one leader can be elected in a given term.

Leader Completeness

If log entry is committed in a term, it will be present in logs of leaders for all higher terms.

Leader Append-Only

A leader never overwrites or deletes entries in its log; it only appends new entries.

State Machine Safety

If a server has applied a log entry at a given index, no other server will ever apply a different command for the same index.

Log Matching

If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

THE GUARANTEE

"All nodes eventually have identical data"

Handling Failure

Leader Failure Timeline

`T=0.00s` **Leader Node crashes ❌**
Process termination or network partition

`T=0.20s` **Followers detect timeout**
No heartbeats received for 200ms

`T=0.35s` **Node2 becomes CANDIDATE**
Increments term, votes for self

`T=0.40s` **Node2 gets majority votes**
Received vote from Node3

`T=0.41s` **Node2 is new LEADER ✓**
Cluster operational again

TOTAL DOWNTIME **~500ms**

Follower Recovery

1. **Follower reconnects**
   Node rejoins the network after restart
2. **Leader detects lag**
   Compares `nextIndex` with follower log
3. **Send missing entries**
   Leader streams `AppendEntries` RPCs
4. **Follower catches up**
   Applies all committed entries to state
5. **Consistency Restored**
   Node is fully synchronized and serving

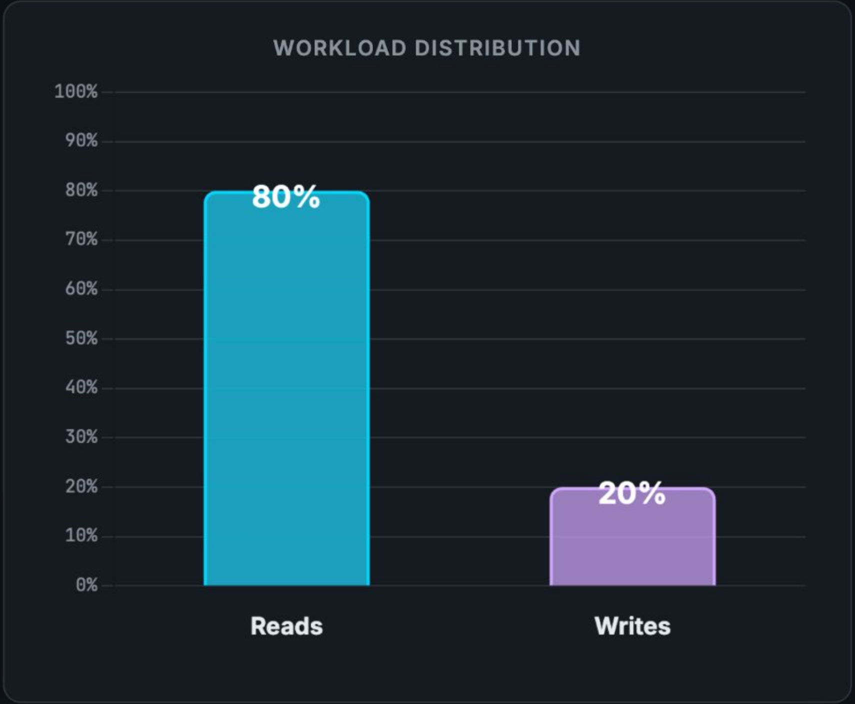
TYPICAL RECOVERY TIME **~10 seconds**

Why Reads Are Slow

NAIVE APPROACH

`"Route ALL reads through Raft log"`

- Every read requires **log append + replication** round-trip
- Read latency degrades to Write latency `(~50-100ms vs optimal <1ms)`
- Massive inefficiency: Reads are **80%** of the total workload!



OUR GOAL

"Fast reads WITHOUT sacrificing consistency"

ReadIndex Protocol

raft_read.log

READINDEX PROTOCOL

```
// 1. Capture State
readIndex = commitIndex;

// 2. Verify Leadership
broadcastHeartbeat();
wait(majority_response); // Confirms still leader

// 3. Wait for State Machine
while (lastApplied < readIndex) {
 wait(); // Ensure data is up-to-date
}

// 4. Local Read & Return
result = stateMachine.lookup(key);
return result;
```

PROTOCOL BENEFITS

No Log Append Needed
Bypasses disk I/O for read operations

Strong Consistency
Guarantees no stale reads are returned

Moderate Latency
~50-100ms (1 RTT required)

KEY INSIGHT

"ReadIndex avoids the heavy Raft log replication path while preserving linearizability."

Read Lease - 105x Faster!

read_lease.py

READ LEASE OPTIMIZATION

```
// 1. Configuration
lease_duration = election_timeout / 2; // 75ms

// 2. Extend Lease on Heartbeat
if (heartbeat_ack_majority) {
 lease_expiry = now() + 75ms;
}

// 3. Handle Read Request
function handleRead(key) {
 if (now() < lease_expiry) {
 return readLocally(key); // FAST!
 } else {
 return useReadIndex(key); // SAFE
 }
}
```

KEY INSIGHT

"If we recently confirmed leadership, we are guaranteed to still be the leader."

WHY IT'S SAFE

Lease (75ms) < Election Timeout (150ms). No other node can become leader while our lease is valid.

0.5ms vs 52ms = 105x Faster!

Consistency Levels

STRONG

- Uses **ReadIndex** protocol
- Guarantees linearizability
- Latency: **50-100ms**
- USE CASE: FINANCIAL DATA

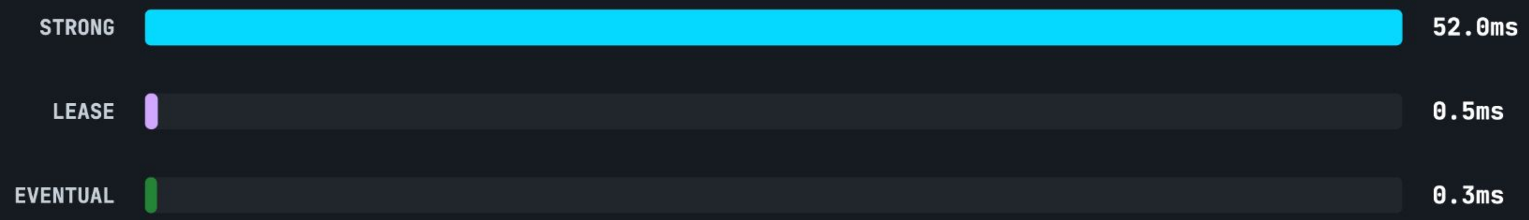
LEASE

- Local read if lease valid
- Falls back to Strong if expired
- Latency: **0.5-1ms**
- USE CASE: USER SESSIONS

EVENTUAL

- Always reads local state
- Might return stale data
- Latency: **0.3ms**
- USE CASE: ANALYTICS

LATENCY COMPARISON (LOWER IS BETTER)



LEASE is 109x faster than STRONG

Why LRU Isn't Enough

STANDARD LRU RULE `"Evict the Least Recently Used item"`

SCENARIO 1 `"holiday-deals"`

LRU Assessment

Last access: 2 days ago — EVICT ⊗

REALITY CHECK
Will be extremely hot next December (Seasonal Pattern)

SCENARIO 2 `"temp-token"`

LRU Assessment

Accessed: 1 hour ago — KEEP ⊗

REALITY CHECK
One-time use token, never accessed again (Scan Pattern)

THE FUNDAMENTAL FLAW

"LRU only sees recency, completely ignoring access PATTERNS."

ML-Based Eviction

CORE IDEA

"Predict which keys will be accessed next"

FEATURE VECTOR

FEATURE NAME	DESCRIPTION
`access_count_hour`	Number of accesses in last 60 mins
`access_count_day`	Number of accesses in last 24 hours
`total_count`	Lifetime access count
`time_since_last`	Seconds since last access
`frequency`	Average accesses per hour

`RandomForest Classifier`

INPUT

Feature vector per cache key (snapshot)

OUTPUT

Probability `P(access)` in next hour

DECISION

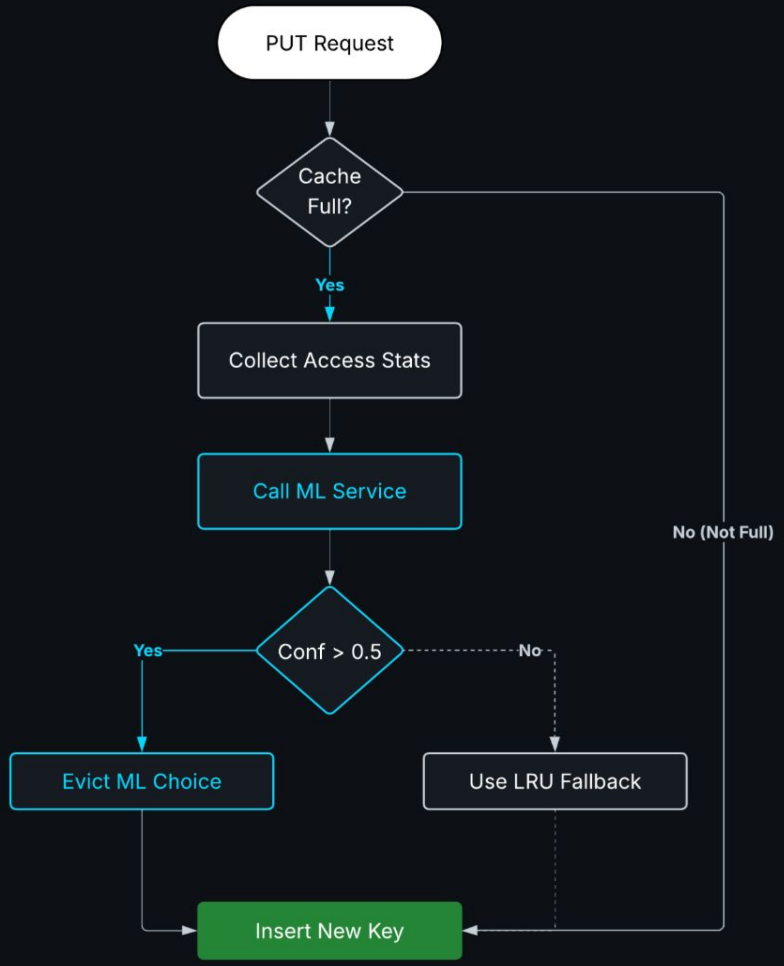
Evict key with **lowest** probability

ML Prediction Algorithm

ml_predict.algo

```
INPUT: All cached keys with stats
1. Build feature vector for each key
     # [hour, day, total, time_since, freq]
2. Send features to ML model
3. Model returns eviction_score
     # Higher score = More likely to evict
4. SELECT key with HIGHEST score
5. RETURN key for eviction
```

Eviction Decision Flow



— ML Optimized Path ---- Legacy Fallback

RandomForest Model

WHY RANDOMFOREST

- Handles **non-linear relationships** in access patterns efficiently
- Works exceptionally well with **small, tabular datasets**
- Provides interpretable **confidence scores** for decision thresholds
- Extremely **fast inference** (<10μs) suitable for cache operations

CONFIGURATION

PARAMETER	VALUE
`n_estimators (Trees)`	100
`max_depth`	10
`min_samples_split`	5

TRAINING PROCESS

TRAINING DATA
Collected from historical access patterns

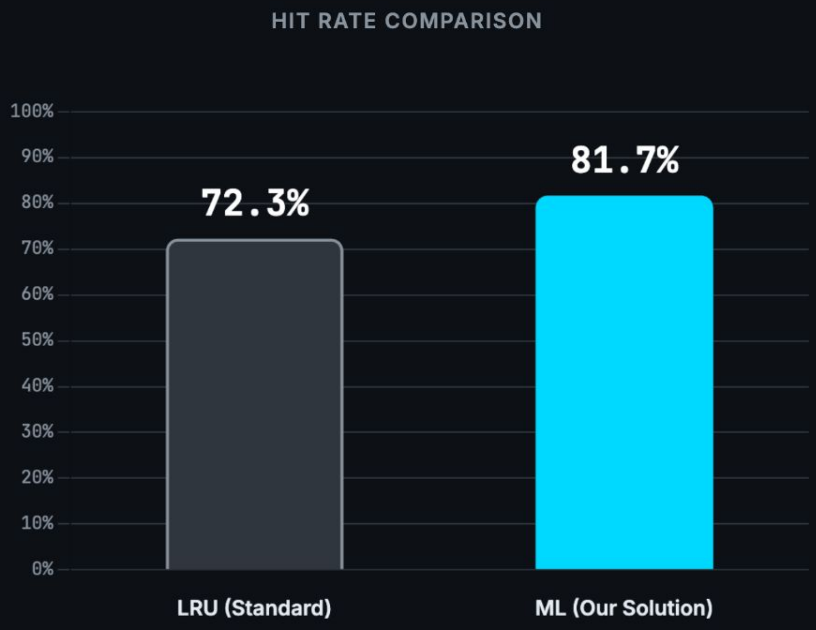
LABELS
1 = accessed within next hour, 0 = not accessed

LIFECYCLE
Retrains periodically as workload patterns change

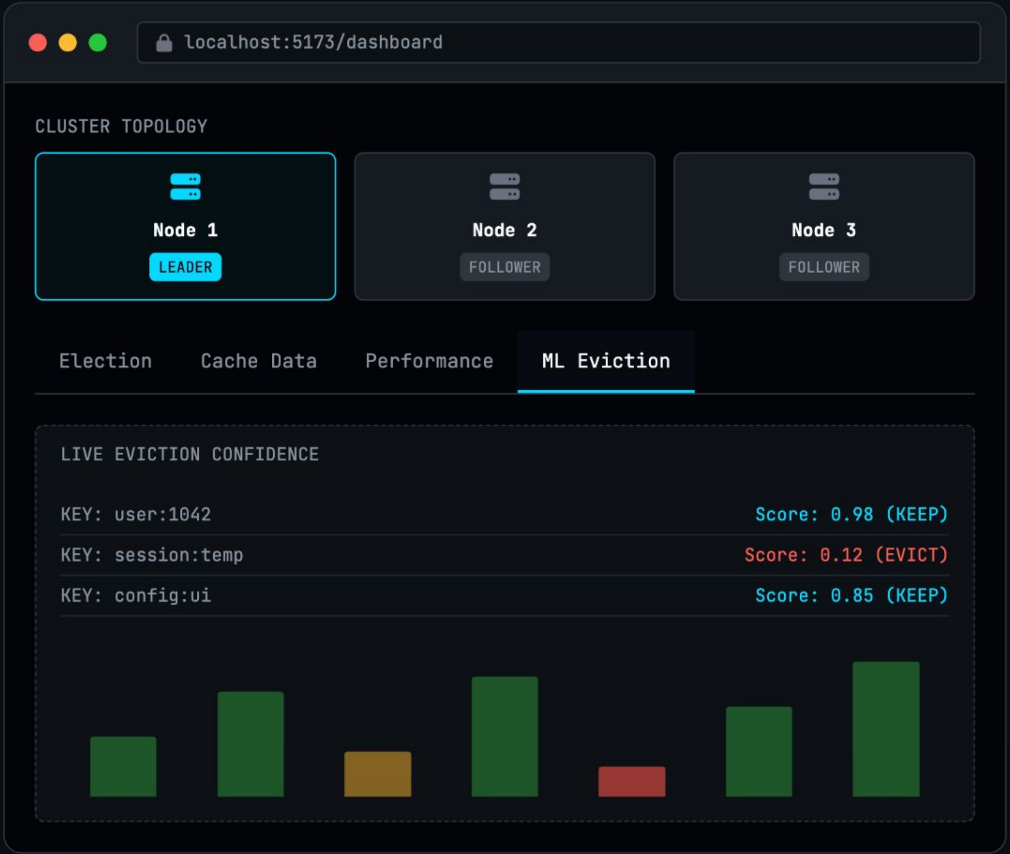
ML vs LRU Results

CACHE SIZE	OPERATIONS	ACCESS PATTERN
100 Keys	10,000	Zipfian Dist.

METRIC	LRU	ML	CHANGE
Hit Rate Cache efficiency	72.3%	81.7%	+13% ✓
Cold Misses Initial fetch failures	284	183	-36% ✓
Latency Processing overhead	0.5ms	2.1ms	+1.6ms



Interactive Demo UI



Real-time Updates

React polling hooks refresh cluster state every 2 seconds

Fault Injection

Kill/Start buttons to simulate node failures instantly

Cache Operations

PUT/GET interface to test replication consistency

Performance Charts

Live latency tracking for Read vs Write operations

ML Visualization

See which keys the model predicts as "hot" or "cold"

Testing Results

Leader Election	✓ PASSED
Follower Recovery	✓ PASSED
Concurrent Writes	✓ PASSED
Read Lease Protocol	✓ PASSED
ML Eviction Accuracy	✓ PASSED

<1 sec
ELECTION TIME

~10 sec
RECOVERY TIME

100%
CONSISTENCY

0%
DATA LOSS

Key Algorithms Summary

LEADER ELECTION

01 Follower timeout (Random 150-300ms)

↓

02 Become CANDIDATE & Request Votes

↓

03 Receive Majority Votes

↓

04 Become LEADER

LOG REPLICATION

01 Leader receives Write & Appends to Log

↓

02 Send `AppendEntries` to Followers

↓

03 Wait for Majority Acknowledgement

↓

04 COMMIT & Apply to State Machine

READ LEASE

01 Client sends Read Request

↓

02 Check: `now < lease_expiry`?

YES (VALID)	NO (EXPIRED)
Return Local Data	Run ReadIndex
(0.5ms)	(50ms)

ML EVICTION

01 Cache Full → Collect Key Stats

↓

02 RandomForest Model Prediction

↓

03 Score = `P(access_next_hour)`

↓

04 Evict Key with Lowest Score

Performance Summary

Election Time	< 1 second
Failover Downtime	~500ms
Recovery Time	~10 seconds
STRONG Read Latency	52ms
LEASE Read Latency	0.5ms 105x faster
ML Hit Rate Improvement	+13% vs LRU
Data Consistency	100%

Strong consistency + High performance

Future Work

Planned roadmap & technical improvements

- [ ] Log compaction & rotation
- [ ] Dynamic cluster membership
- [ ] Multi-region deployment
- [ ] Prometheus metrics
- [ ] Security (TLS, authentication)
- [ ] Production error handling

References

[1] **In Search of an Understandable Consensus Algorithm**
Diego Ongaro & John Ousterhout, USENIX ATC '14 (Stanford University)

[2] **Consensus: Bridging Theory and Practice**
Diego Ongaro, PhD Dissertation, Stanford University (2014)

[3] **The Secret Lives of Data: Raft Visualization**
`thesecretlivesofdata.com/raft`

[4] **CoreOS etcd Raft Implementation**
`github.com/etcd-io/raft`

[5] **Scikit-learn: Machine Learning in Python**
`scikit-learn.org`

PROJECT REPOSITORY: `github.com/jahnavikedia/raft-cache`

Thank You!