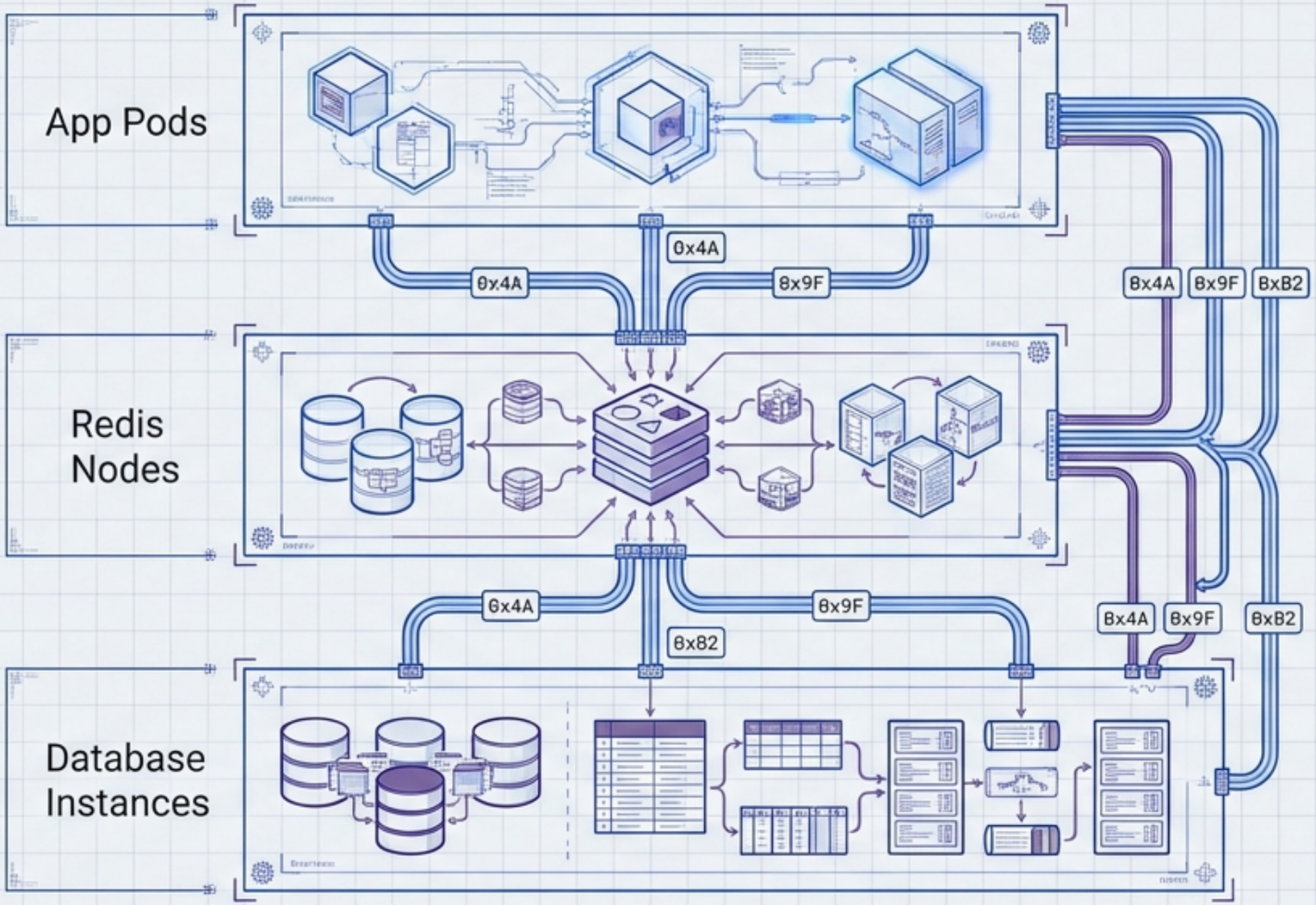
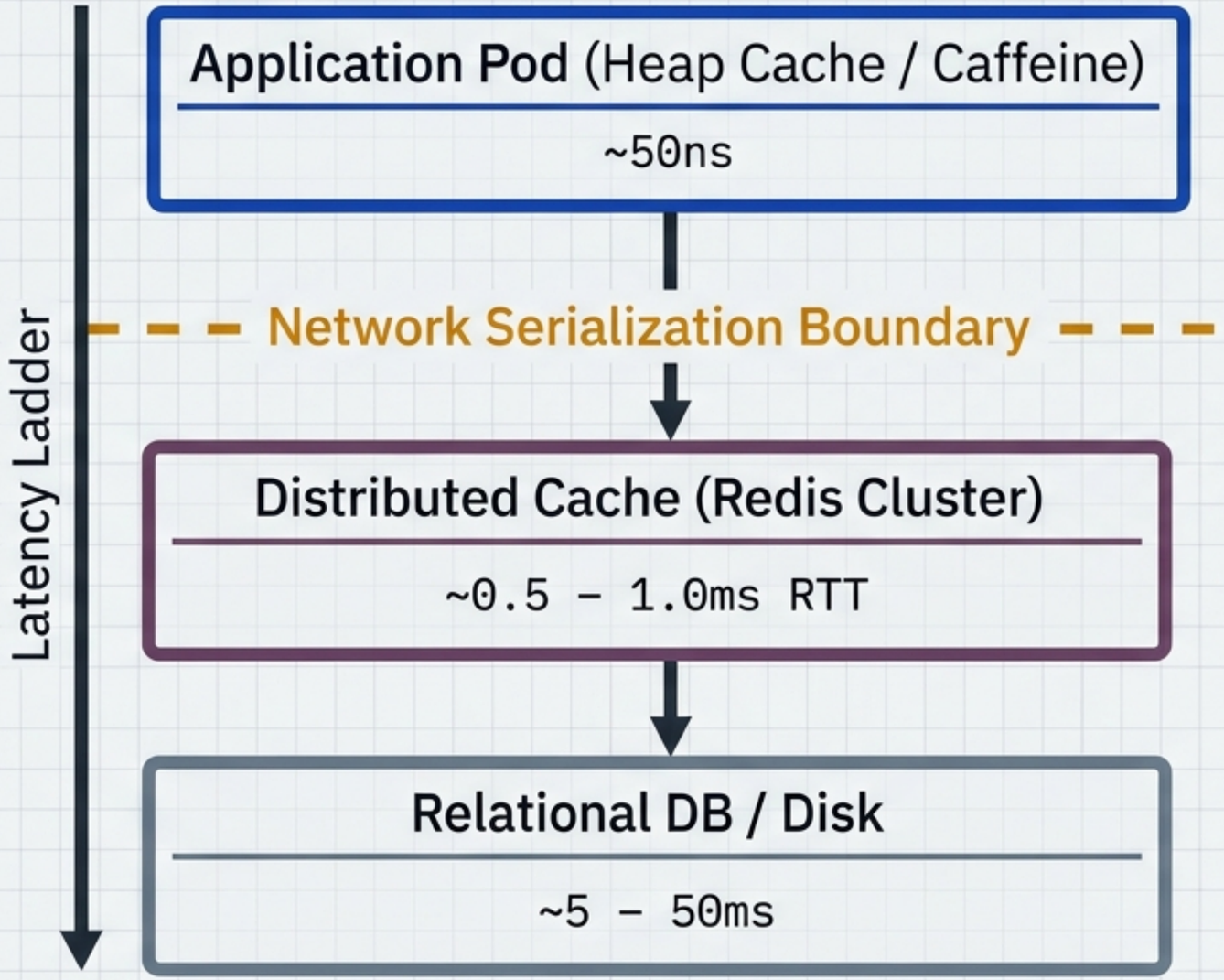
# Web Architecture — S02E07: Data-Tier Scaling

## Redis Internals, Cache Topologies, Stampede Defense & Connection Multiplexing




Gemini Notebook

# Tiered Application Caching: Physics & Topologies




L1 avoids serialization and network hops entirely, but risks fleet-wide inconsistency.

L2 is strongly coherent, but introduces network round-trips.

Gemini Notebook

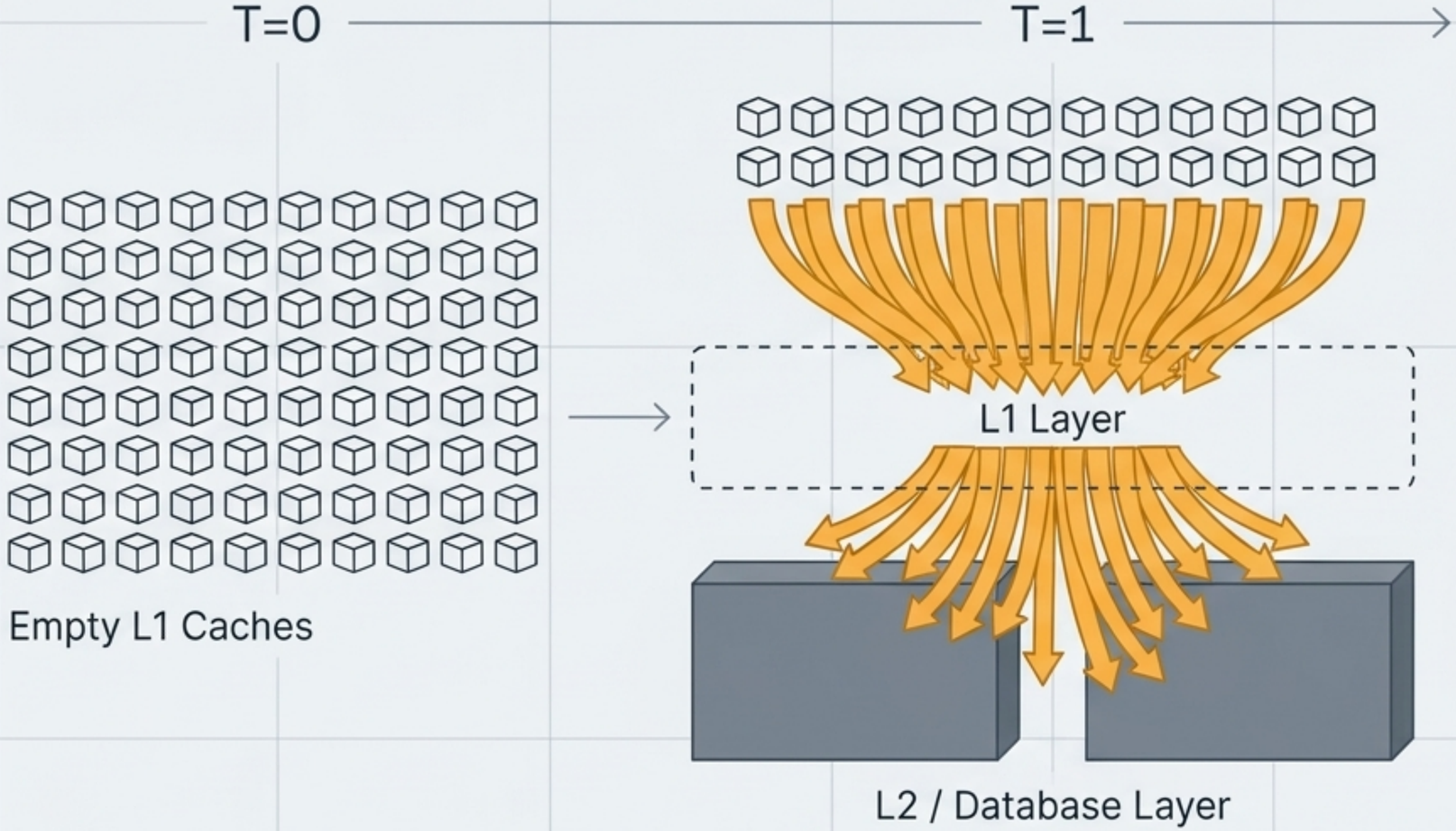
# Trade-offs in Caching Layers

| Dimension | L1 (In-Process) | L2 (Shared Redis) |
| --- | --- | --- |
| Latency | Ultra-fast (ns) | Fast (ms) |
| Coherence | Highly incoherent across fleet | Strongly coherent (Single source of truth) |
| Network Cost | Zero bandwidth | High bandwidth (Serialization overhead) |
| Blast Radius | Low (Cold starts isolated to 1 pod) | High (Cluster failure = system brownout) |

The ultimate architecture often requires both:
L1 for ultra-hot static data, L2 for shared, mutating state.

Gemini Notebook

# The Danger of L1 Fleet-Wide Flushes

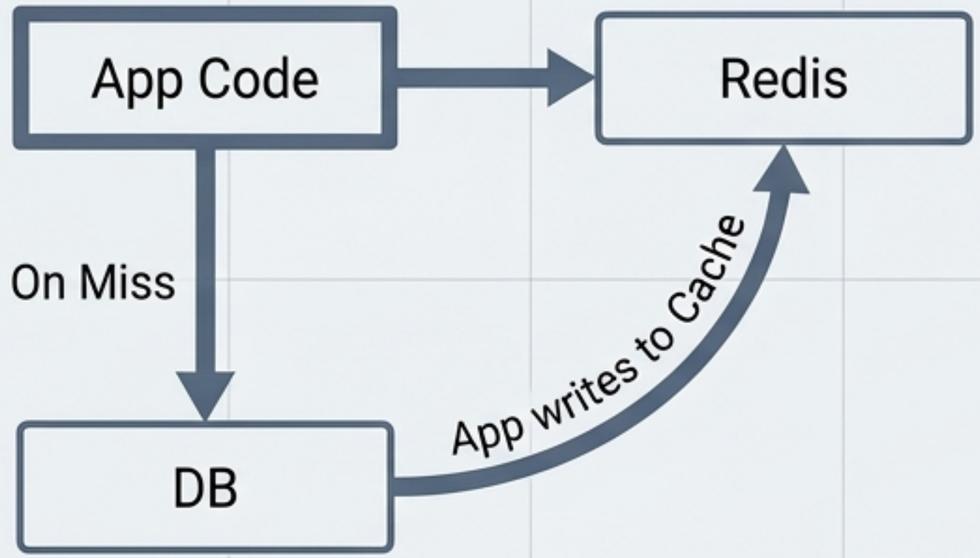



Local caches tie downstream load directly to fleet size. A deployment that flushes the L1 cache creates an immediate, severe burst to dependent services.

Gemini Notebook

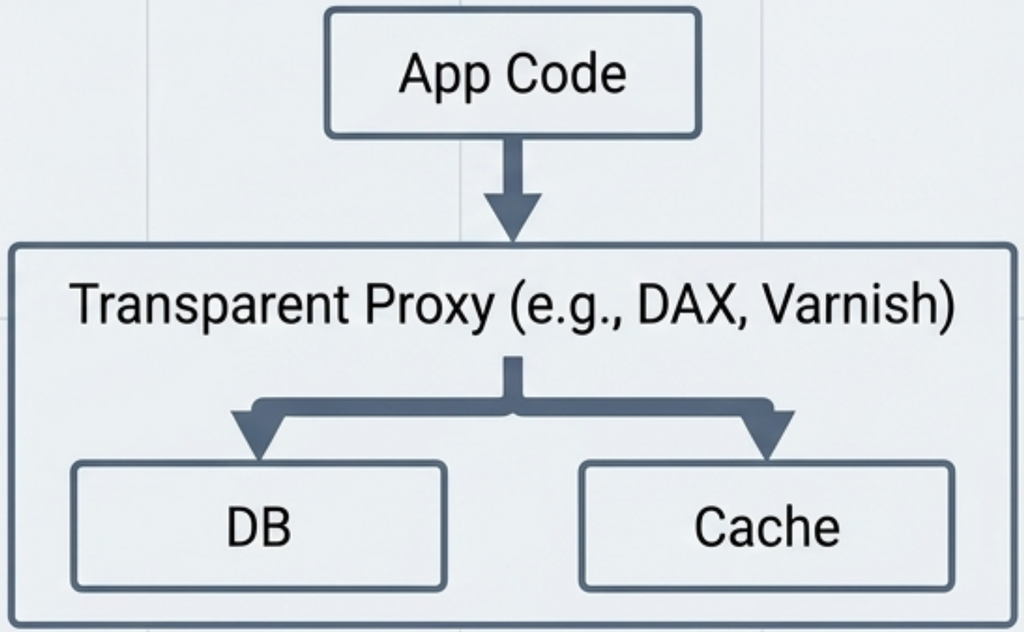
# Architecture: Inline vs. Side Caches

**Side Cache (Cache-Aside)**




Requires complex application logic, but offers fallback resilience if the cache drops.

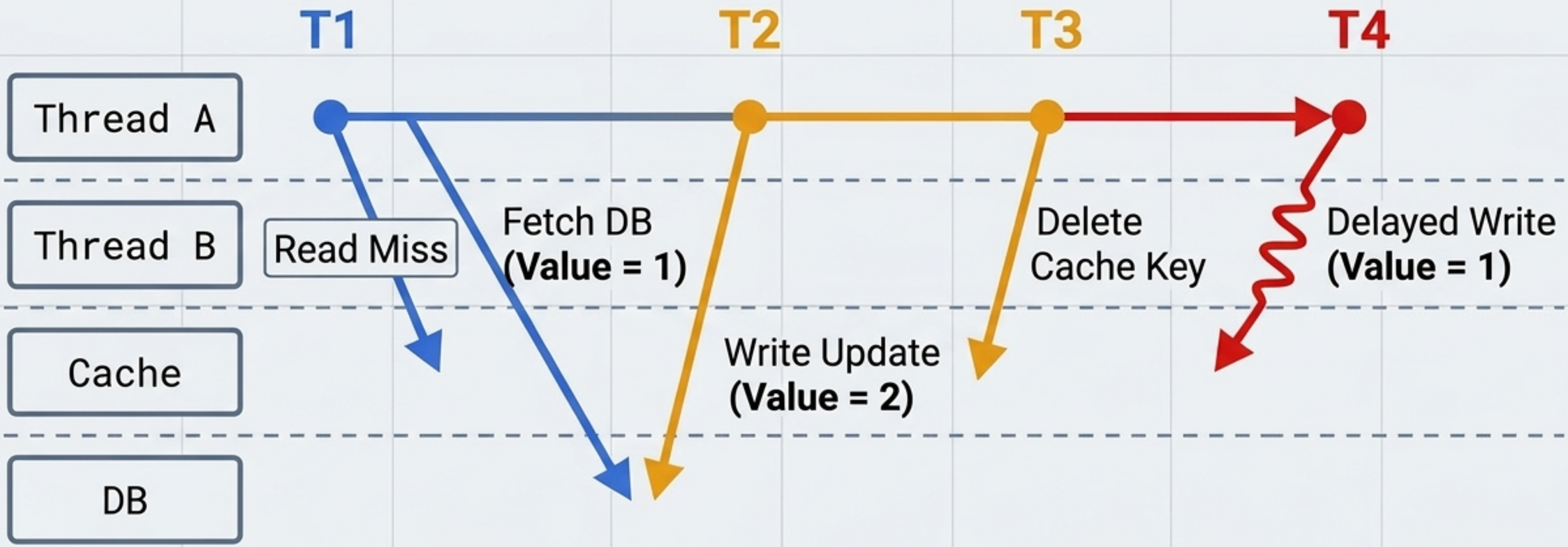
**Inline Cache (Read-Through)**




Simplifies client code, but acts as a hard availability dependency. If the inline cache fails, it’s a total outage.

Gemini Notebook

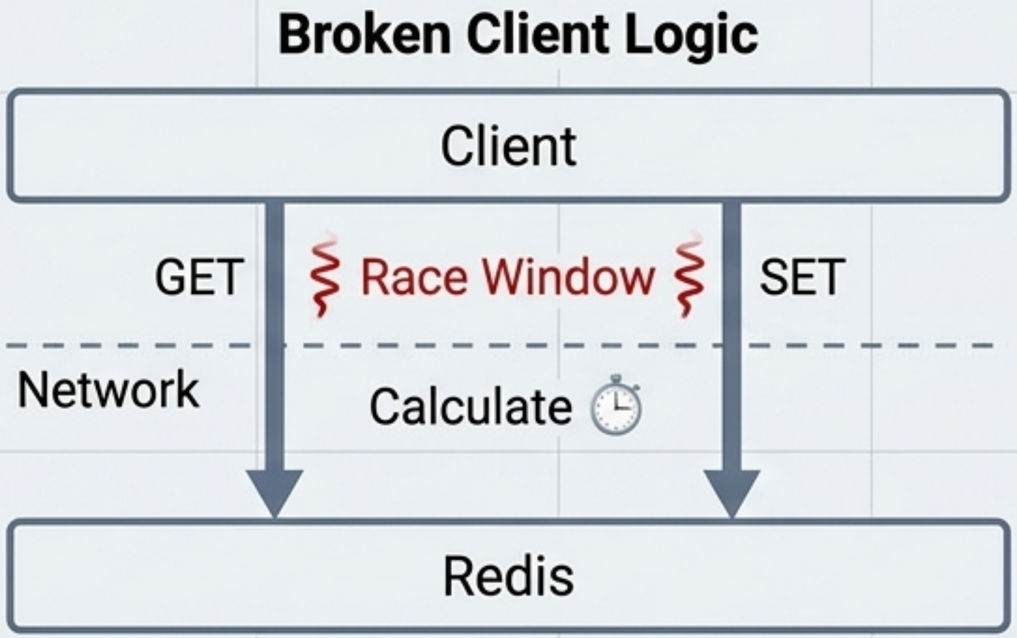
# Concurrency Hazard: Stale-Data Overwrite

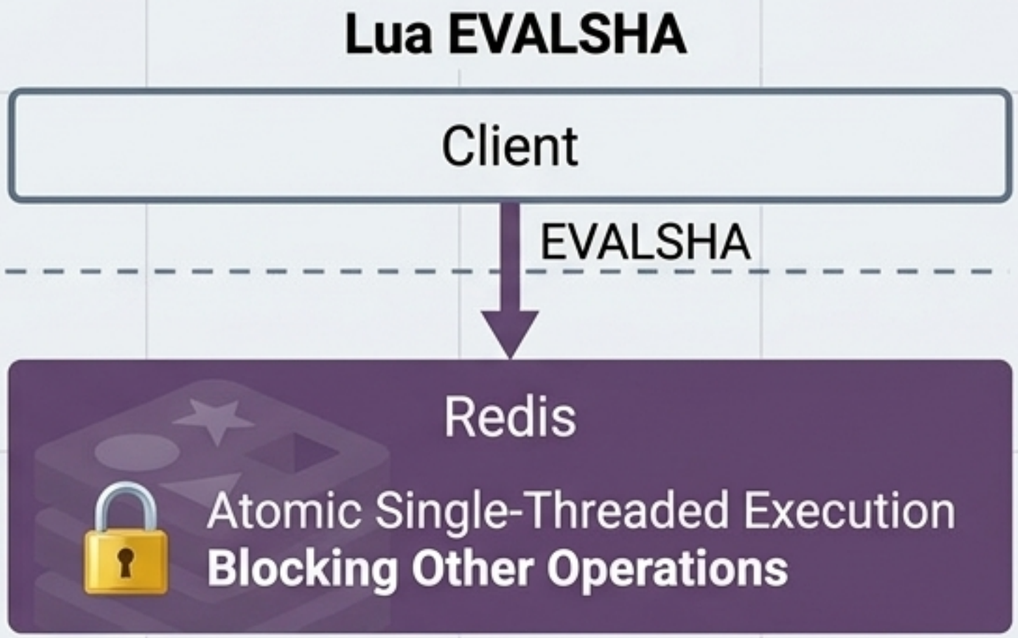



The cache now permanently holds stale data until the next TTL expiry.
The classic Read-Miss + Write-Delete race.

Gemini Notebook

# Lua Scripting: The Antidote to Multi-Key Races



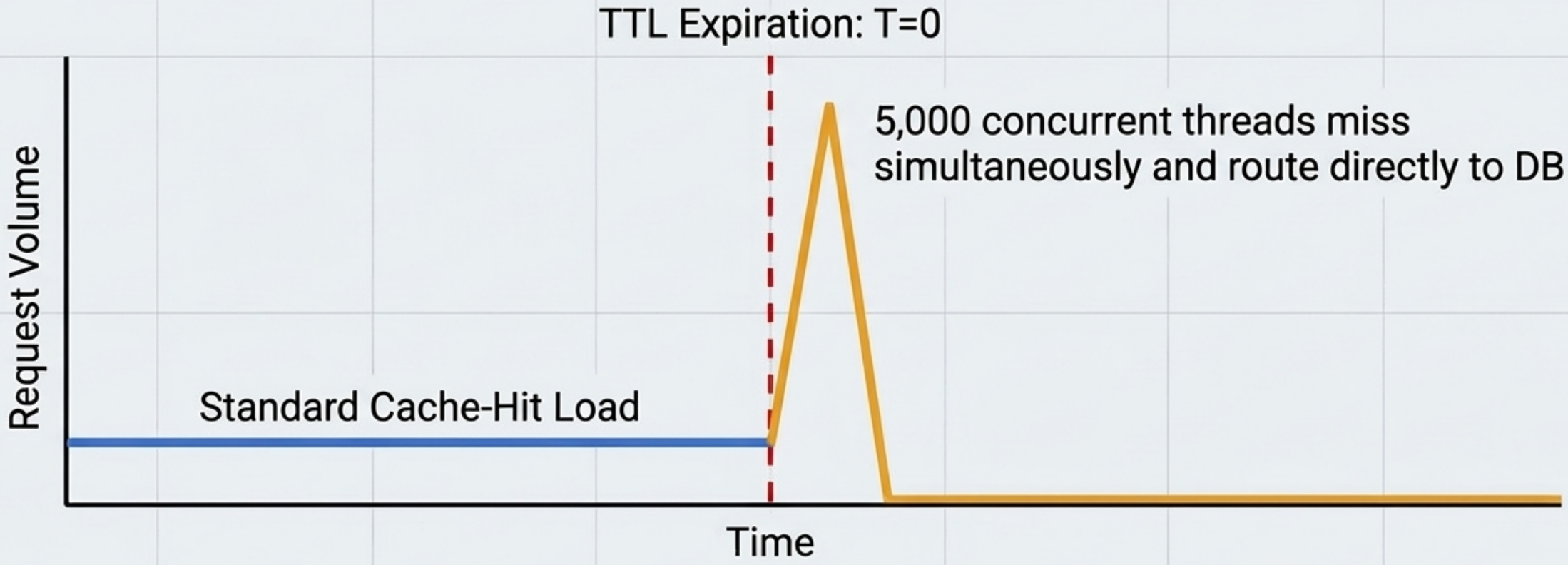




```
#!lua flags=no-writes
-- Parameterized for deterministic replication
local val = redis.call('GET', KEYS[1])
-- Logic executed atomically without race windows
```

Redis guarantees the script's atomic execution. Pass explicit KEYS[1] to ensure cluster compatibility and proper effects replication.

Gemini Notebook

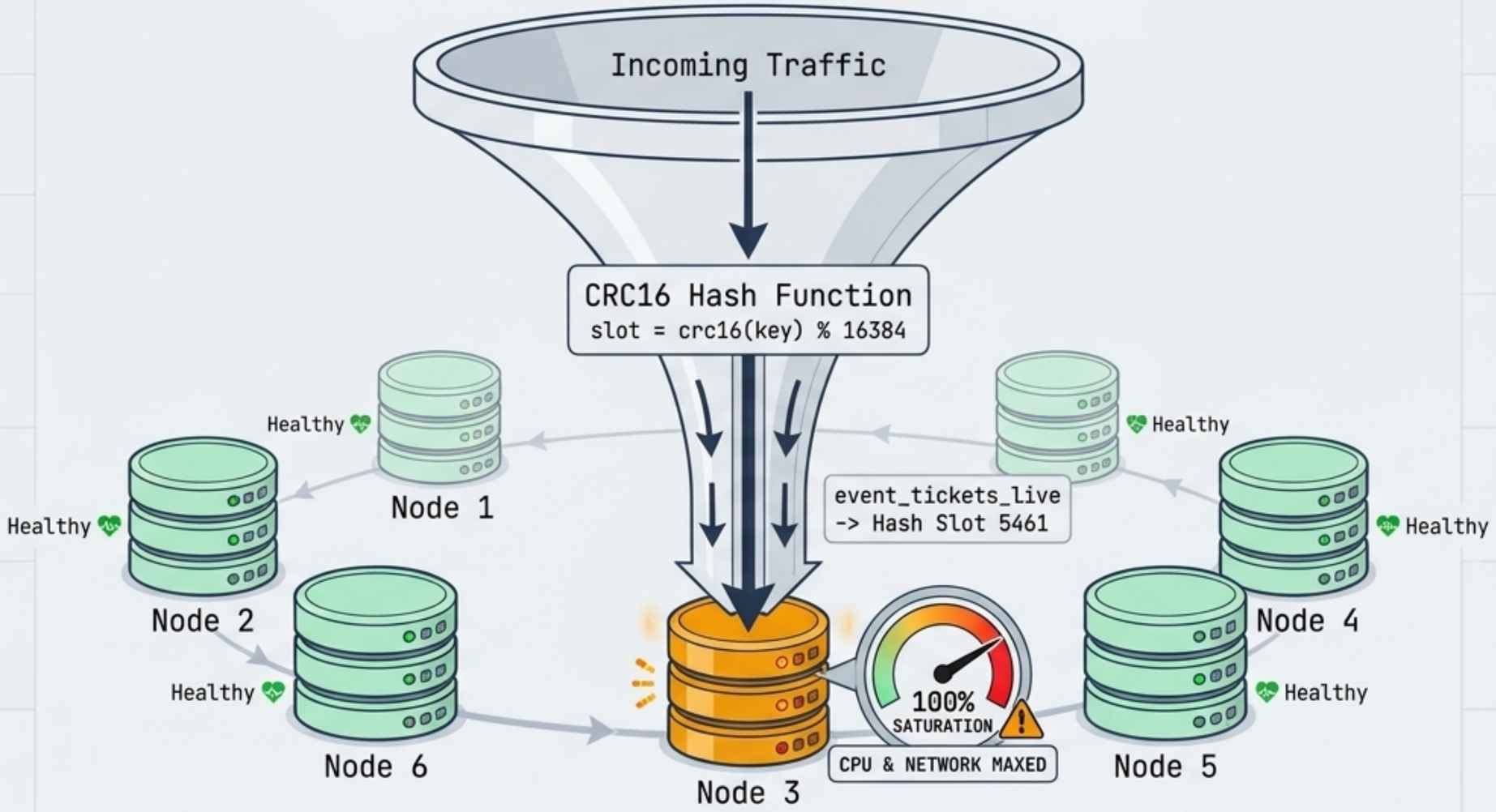
# The Thundering Herd: TTL Expiration




When an expensive, highly concurrent key expires, waiting threads become a thundering herd. They overwhelm the data store before the first thread can repopulate the cache.

Gemini Notebook

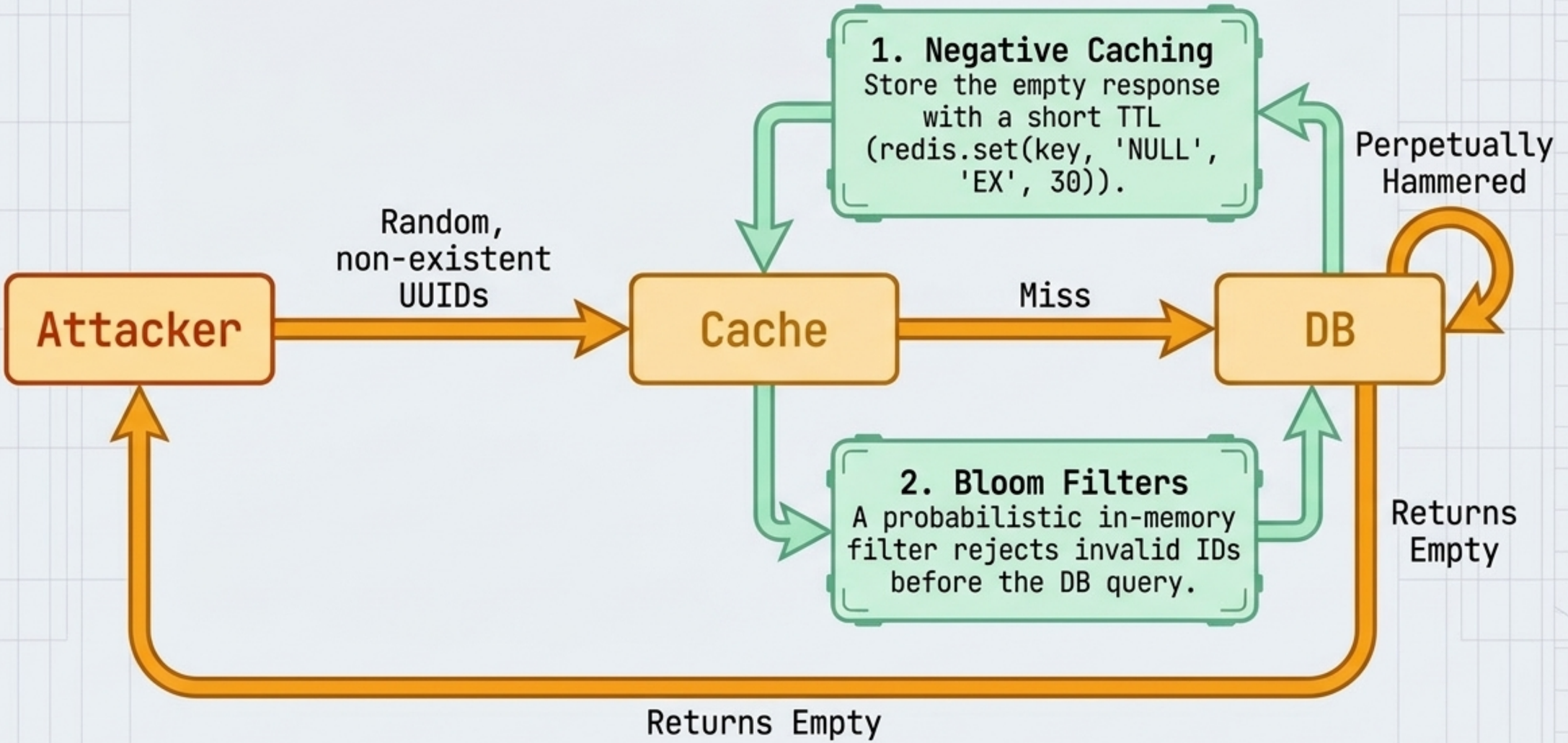
# Hot-Key Saturation in Redis Clusters




**The Single-Slot Bottleneck:** No matter how many nodes you add to a cluster, a single hot key routes to a single shard, maxing out its physical CPU and network interface.

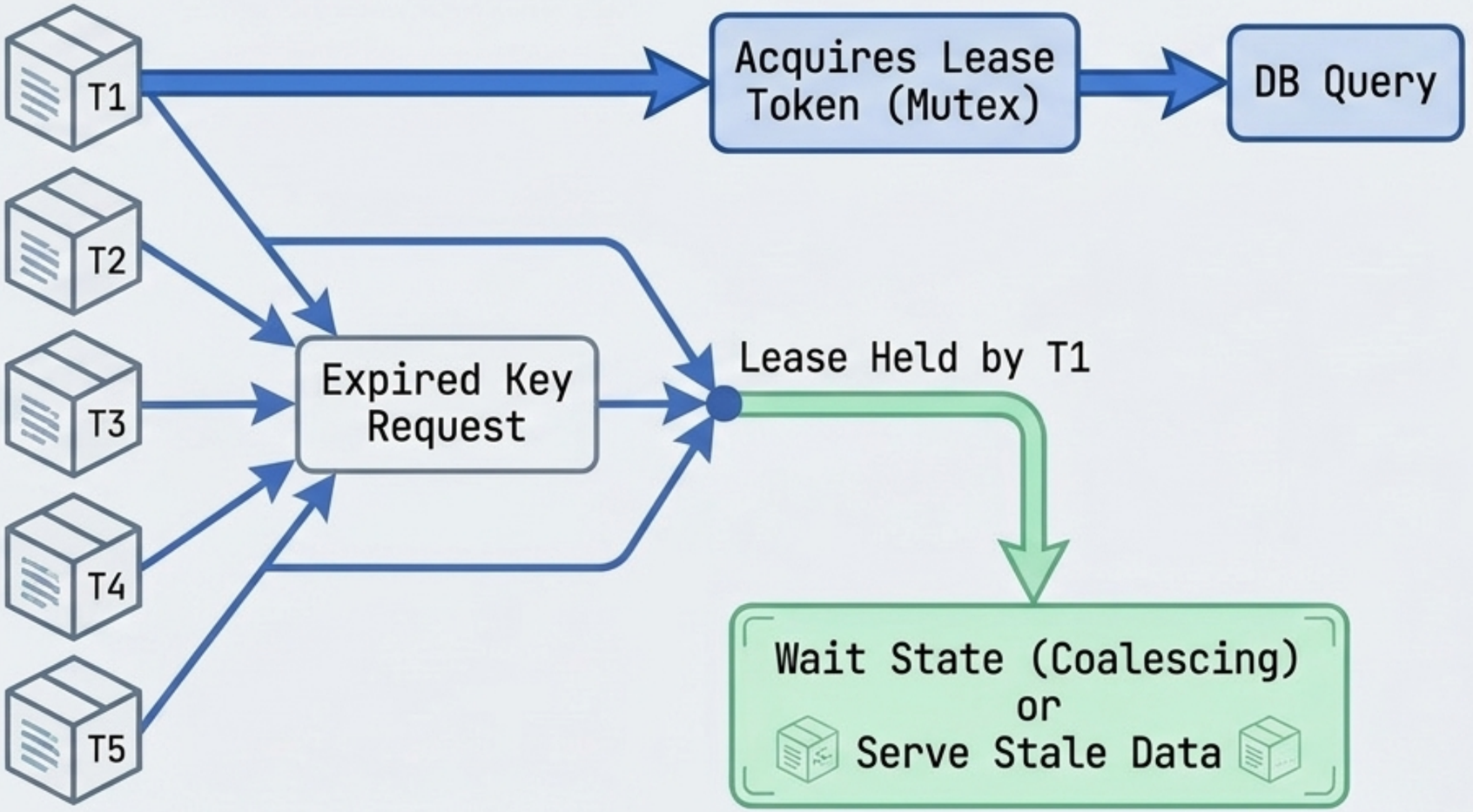
Gemini Notebook

# Malicious Cache Penetration




Gemini Notebook

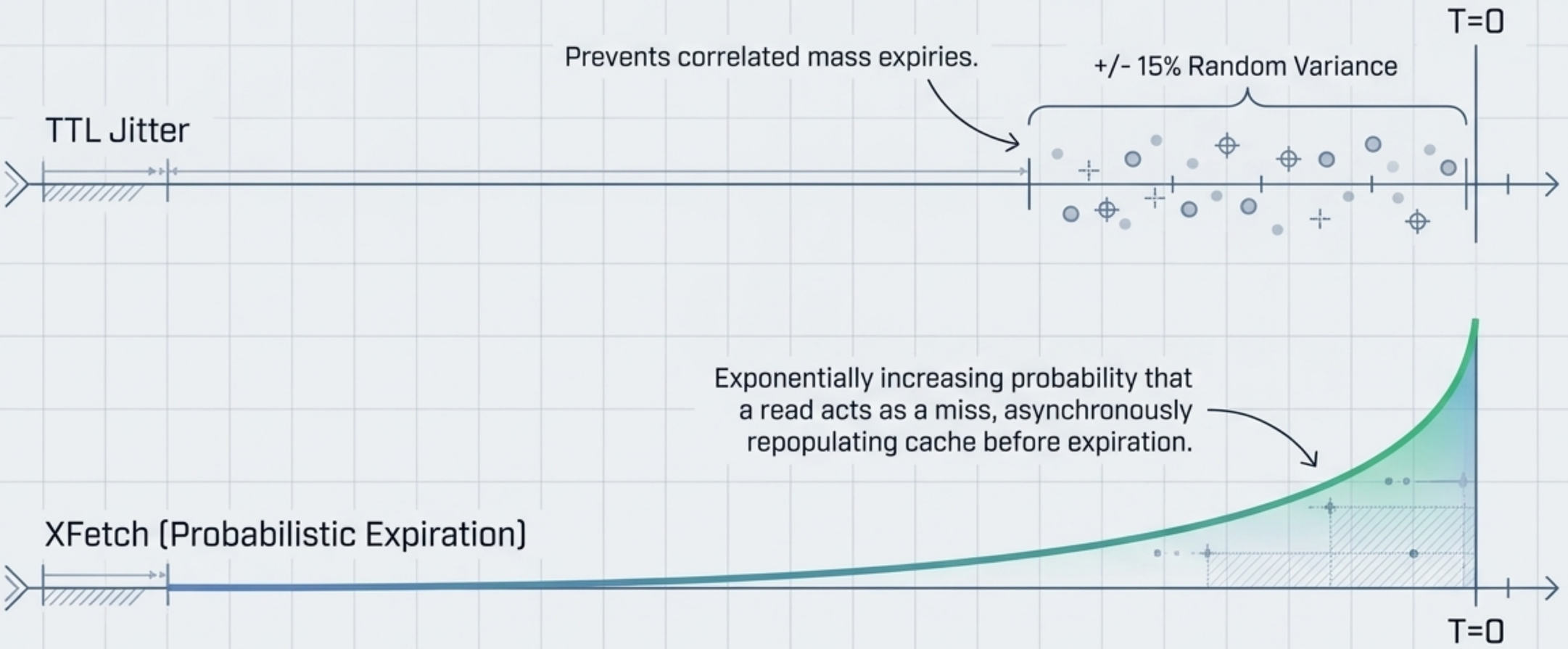
# Stampede Defense: Request Coalescing




**Coalescing** turns 5,000 DB queries into 1 DB query and 4,999 waiting (or stale) reads, shielding the database entirely.

Gemini Notebook

# Stampede Defense: Jitter & Probabilistic Expiration

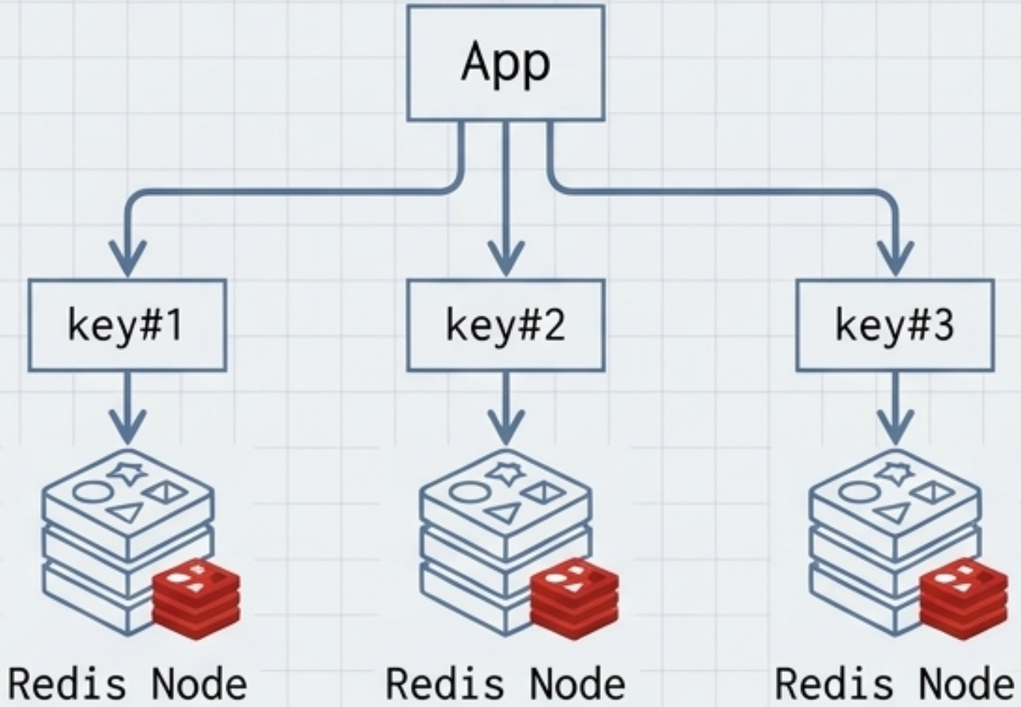



> "Why do people always forget that you need to add a little jitter?"
> **— Ade Oshineye, Google SRE**

Gemini Notebook

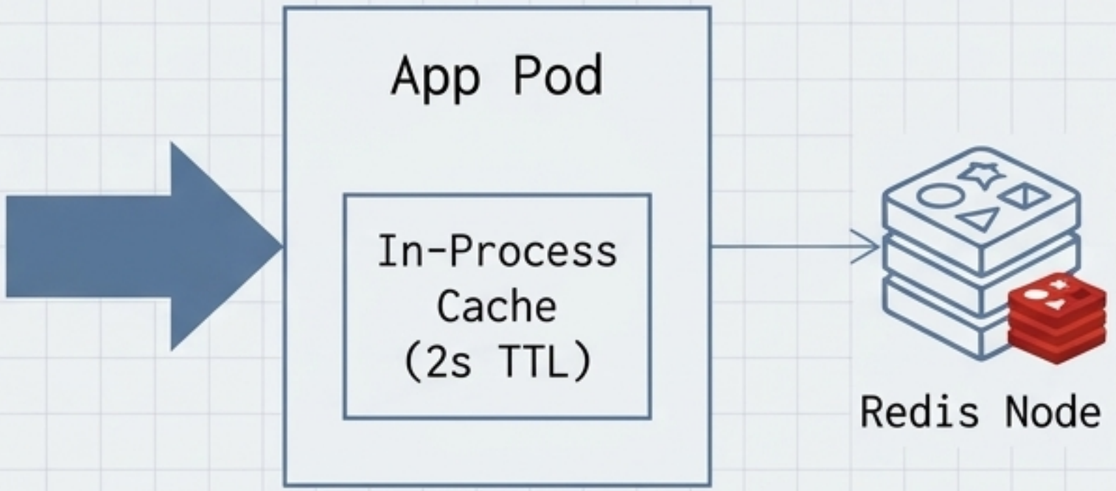
# Defeating the Single-Shard Bottleneck

## Key Replication




Reads randomly select a replica suffix, spraying hot-key load across shards.

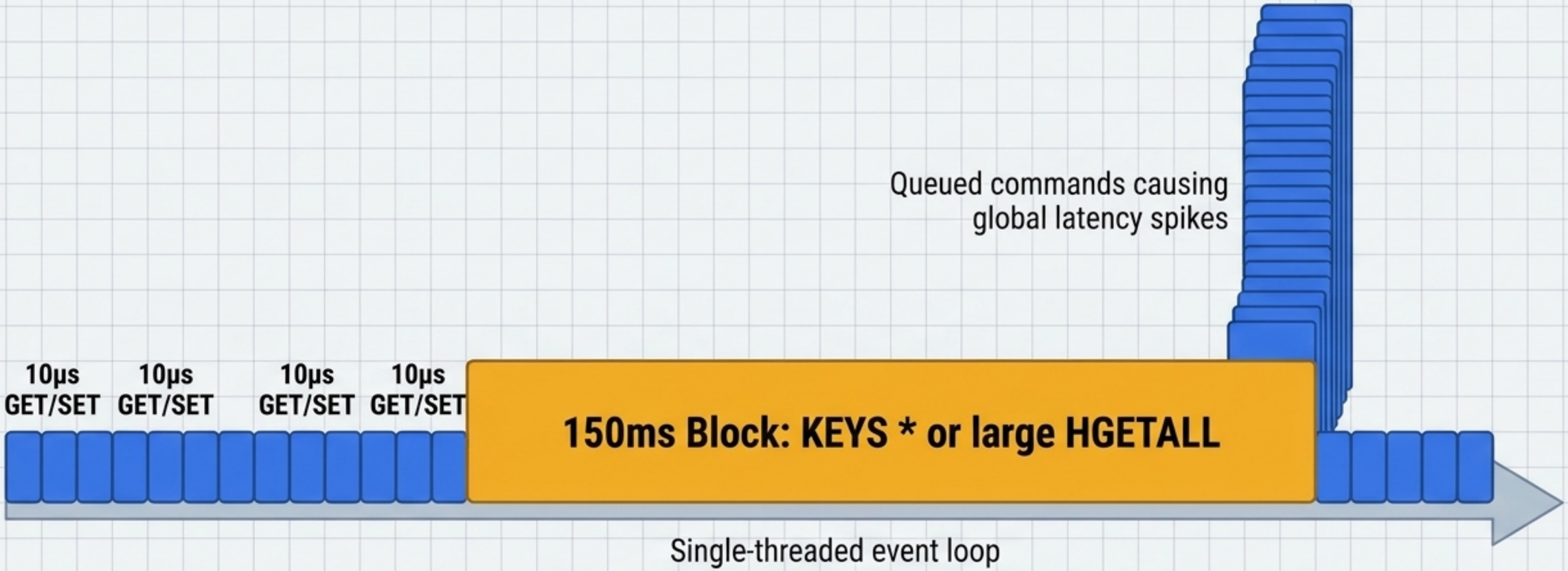
## L1 Micro-Caching




A 2-second in-process TTL for a viral hot key absorbs 99% of read throughput at the application layer, shielding the Redis network interface from complete saturation.

Gemini Notebook

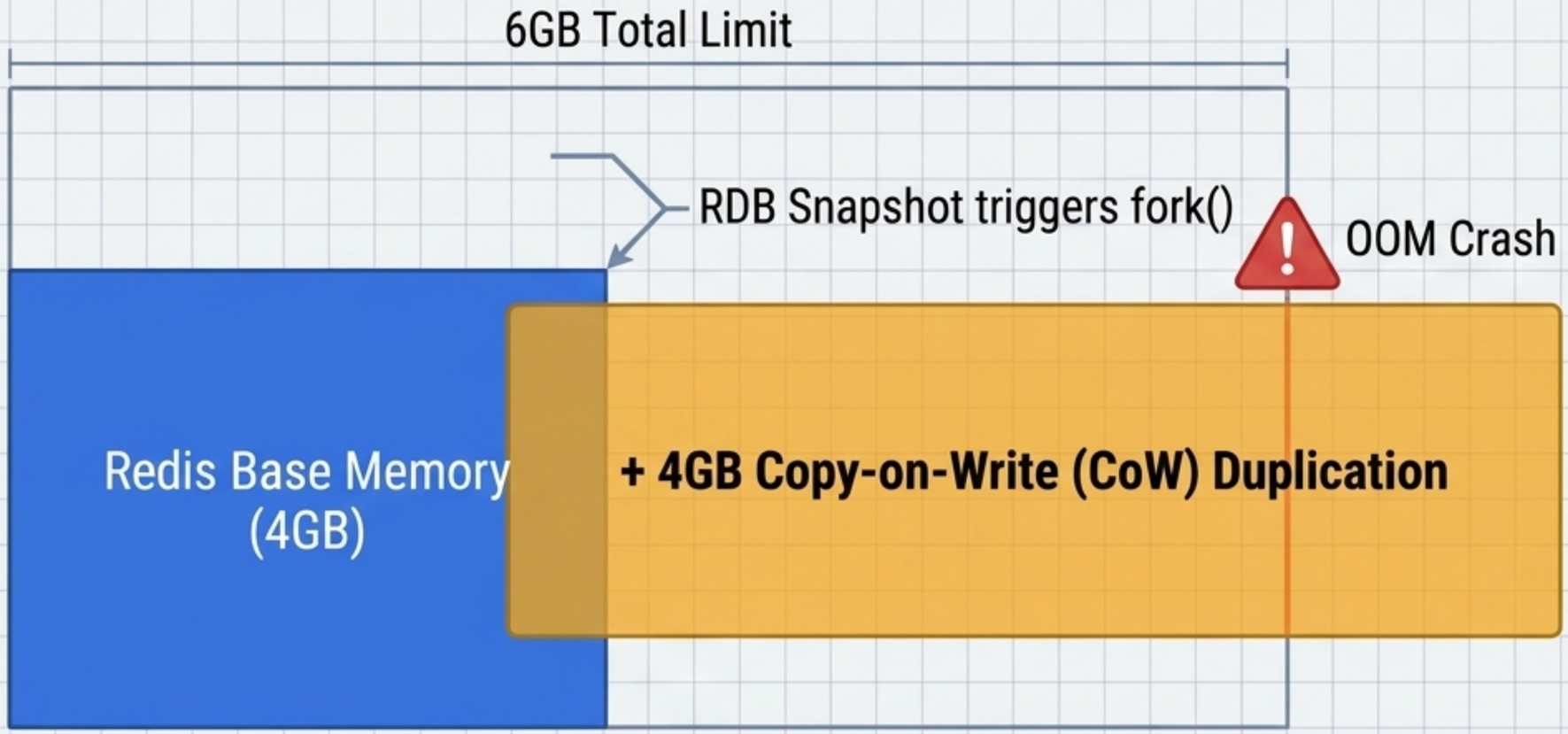
# Redis Event-Loop Hazards




```
[HAZARD] O(N) operations halt the world.
[FIX] Use SCAN instead of KEYS.
[FIX] Use async UNLINK instead of synchronous DEL for large objects.
```

Gemini Notebook

# BGSAVE: The fork() Memory Hazard




```
echo 1 > /proc/sys/vm/overcommit_memory
Always leave a substantial RAM buffer (up to 50%) for
Copy-on-Write overhead when using RDB persistence.
```

Gemini Notebook

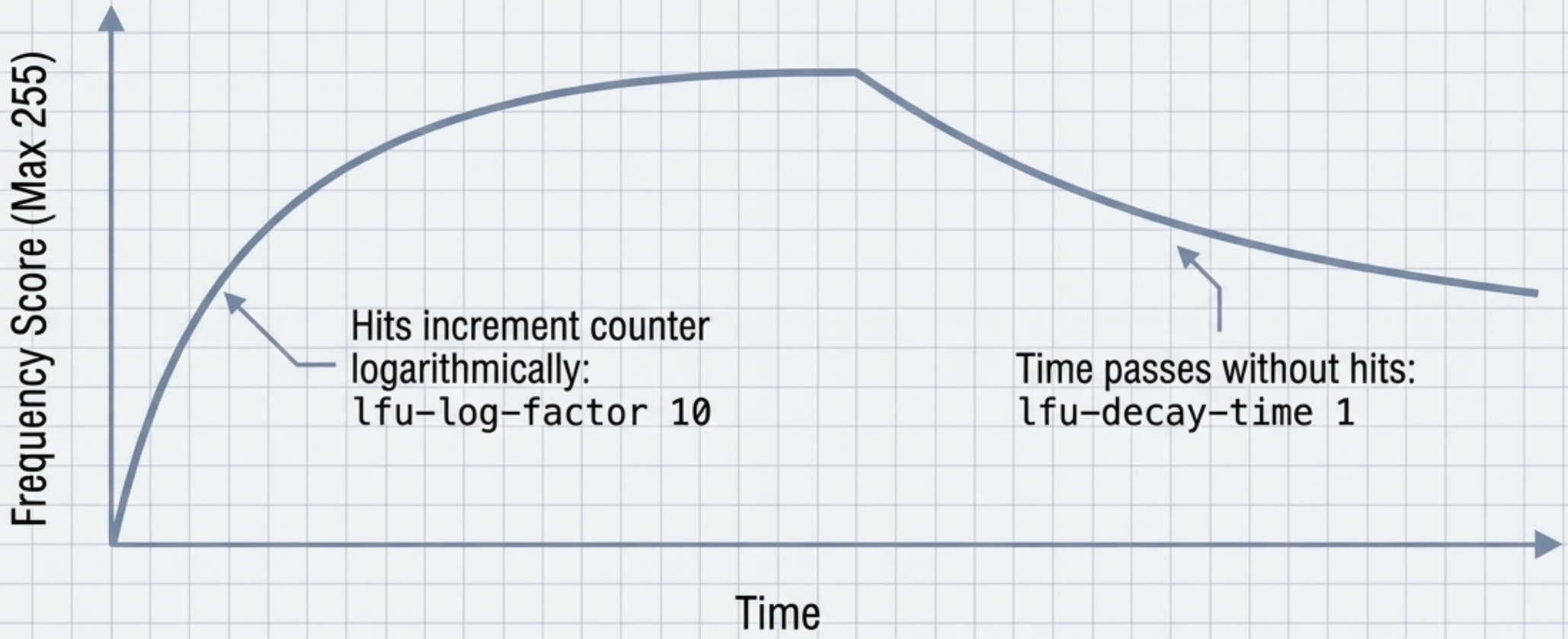
# Cache Eviction: How Redis Discards Data

| maxmemory-policy | Mechanism | Optimal Use Case |
| --- | --- | --- |
| noeviction | Fails new writes (OOM) | Primary datastores. Fatal for caches. |
| allkeys-lru | Approximated Least Recently Used | Pareto/power-law access patterns. Good default. |
| volatile-ttl | Evicts keys with shortest remaining TTL | App knows exactly what to expire. |
| allkeys-lrm | Least Recently Modified | Read-heavy workloads preserving active updates. |

```
Redis LRU is approximated to save memory.
Setting 'maxmemory-samples 5' (or 10) tunes the algorithm
closely to true LRU without massive memory overhead.
```

Gemini Notebook

# The LFU Morris Counter Internals

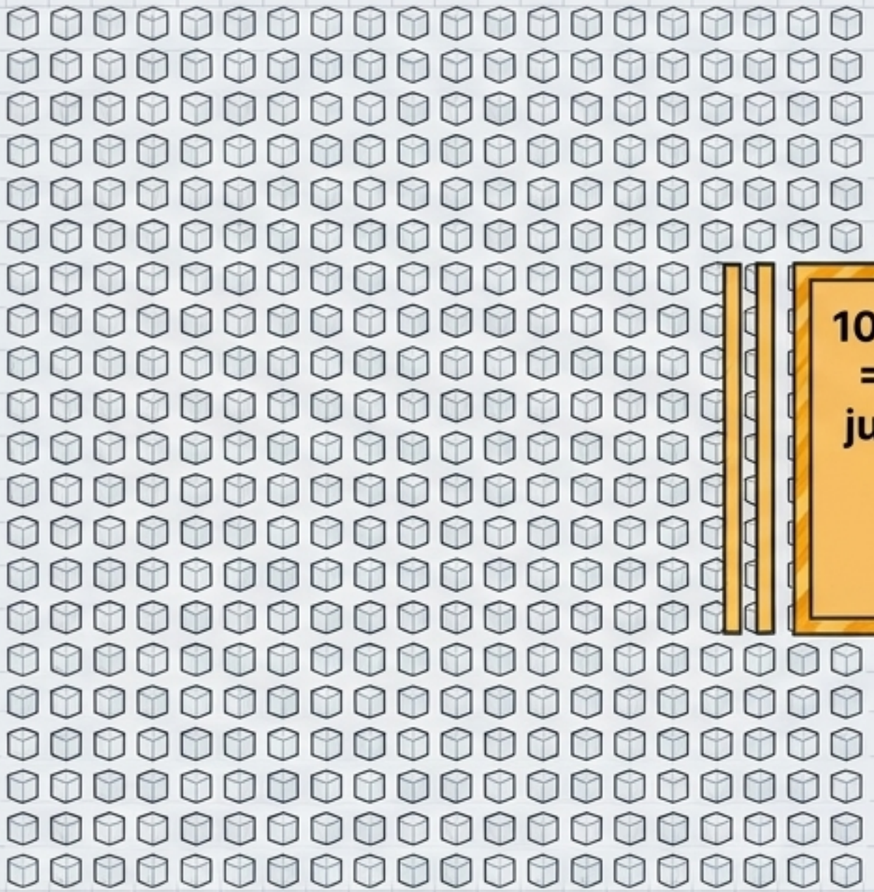



LFU uses a probabilistic Morris counter. The decay mechanism ensures that past virality doesn't permanently lock a key in memory, adapting dynamically to shifting traffic patterns.

Gemini Notebook

# Why Databases Break Before Caches

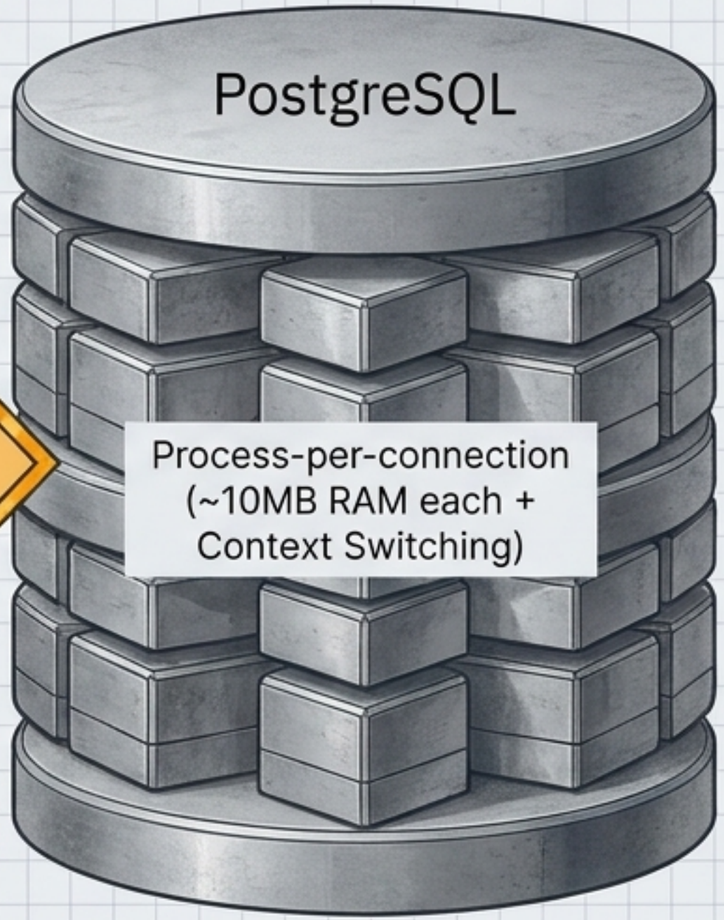
## The Scale-Out App



10,000 ephemeral K8s pods / Serverless Lambdas

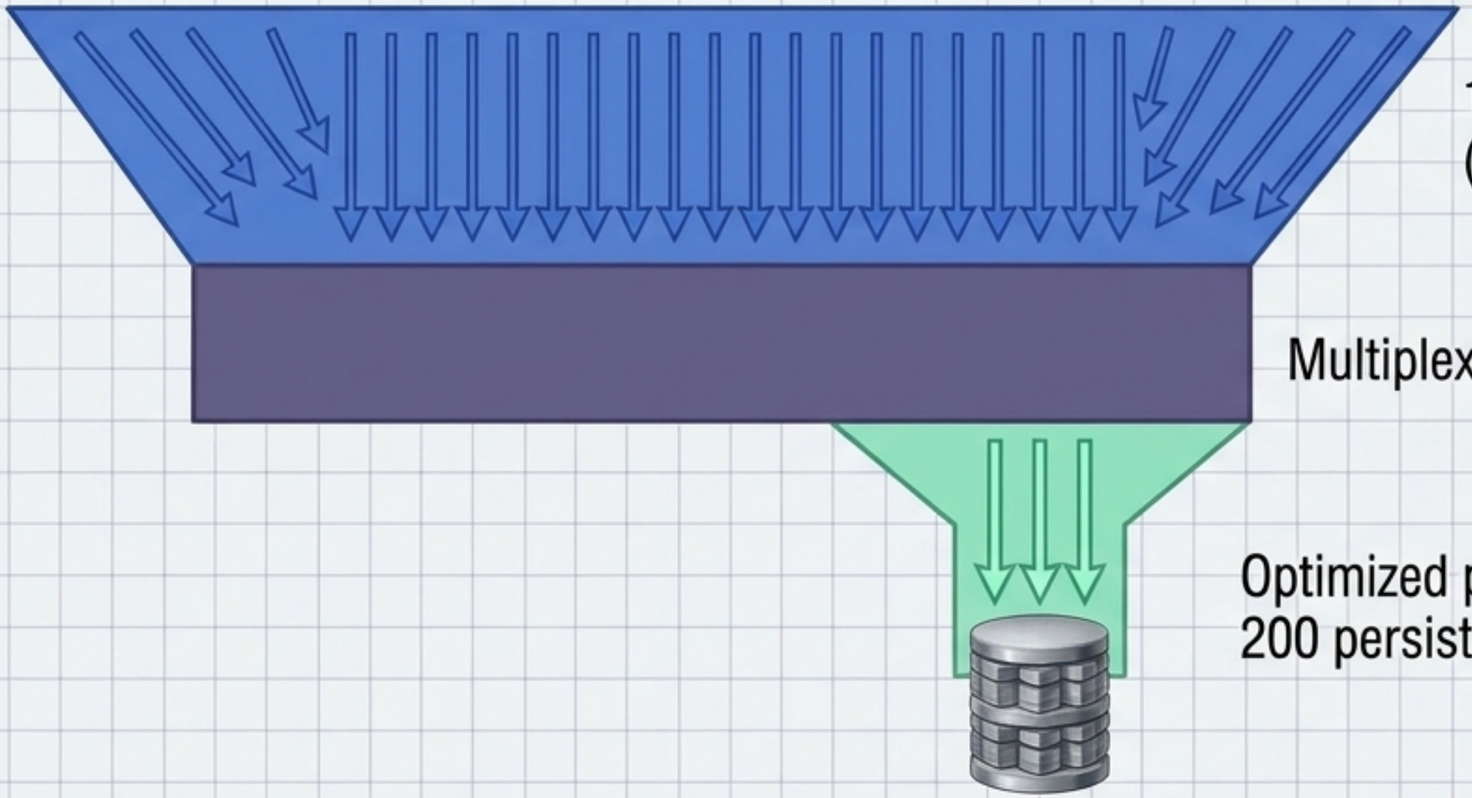
**10,000 direct connections = 100GB RAM overhead just for connection state. DB instantly OOMs before executing a single query.**

## The DB Constraint




Gemini Notebook

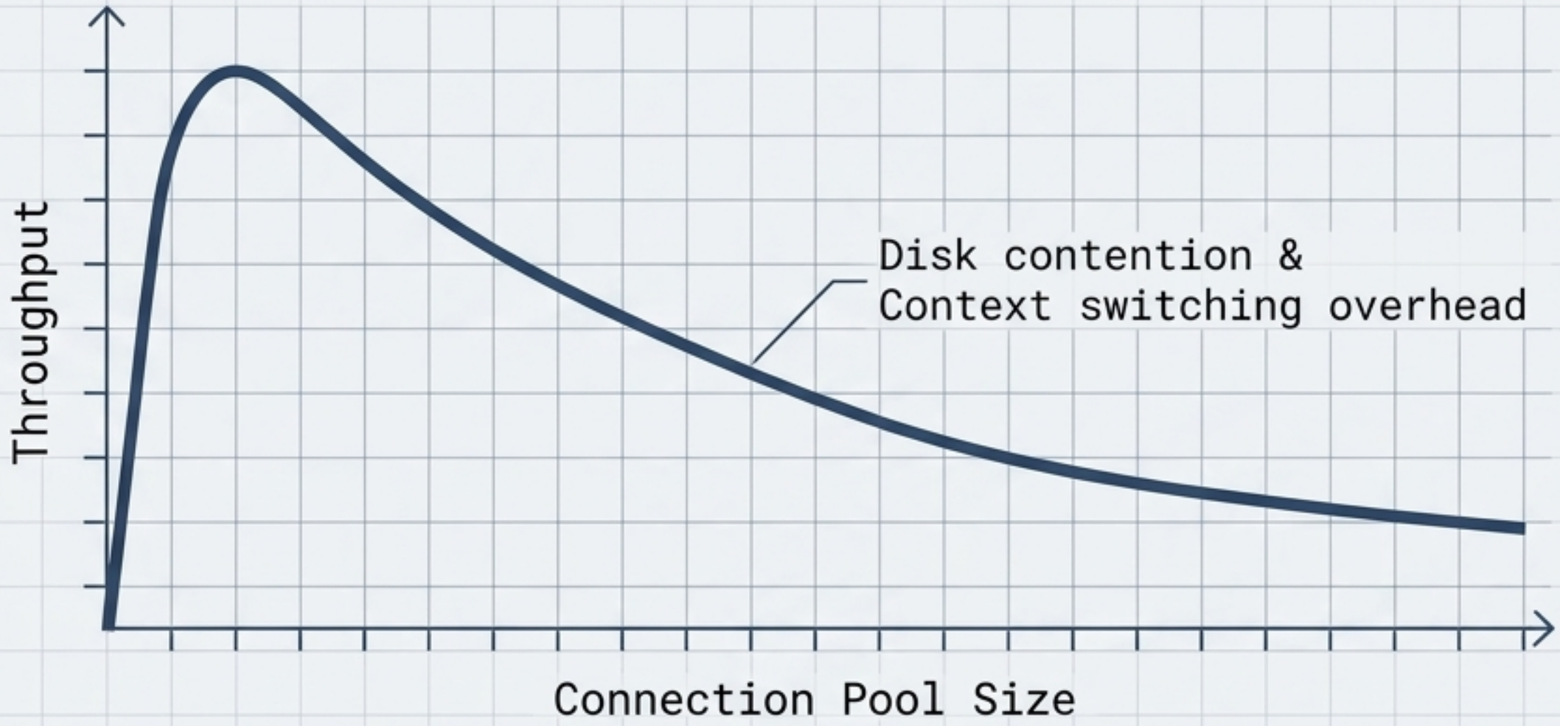
# Shielding the Database via Multiplexing




Transaction-mode multiplexing detaches the logical client connection from the physical DB connection, enabling massive application scale-out without connection exhaustion.

Gemini Notebook

# The Counter-Intuitive Math of Pool Sizing

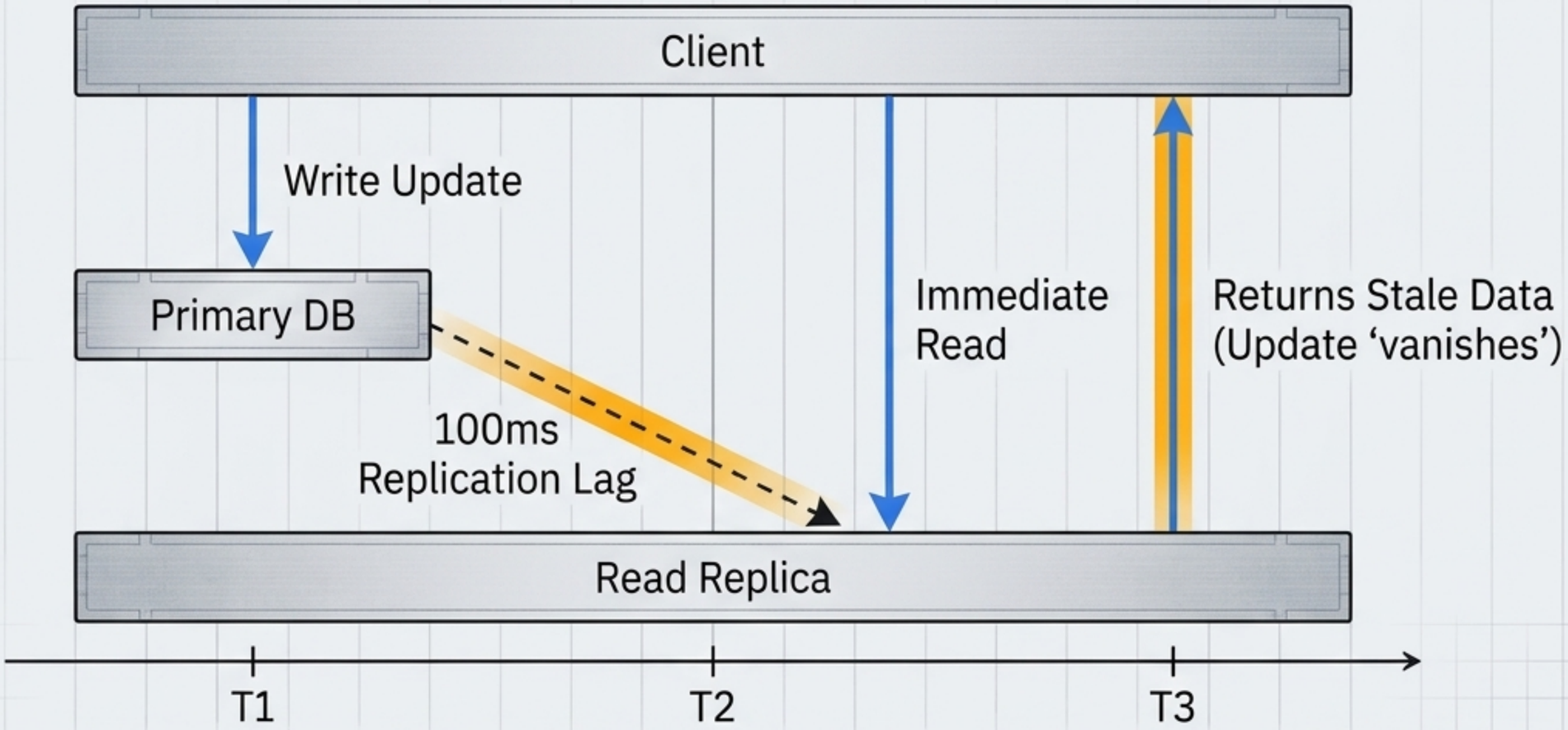



Optimal Pool Size ≈ (Core Count * 2) + Effective Spindle Count

Optimal pool sizes track physical CPU cores, not concurrent web traffic. A 4-core database often performs best with a pool of under 20 connections.

Gemini Notebook

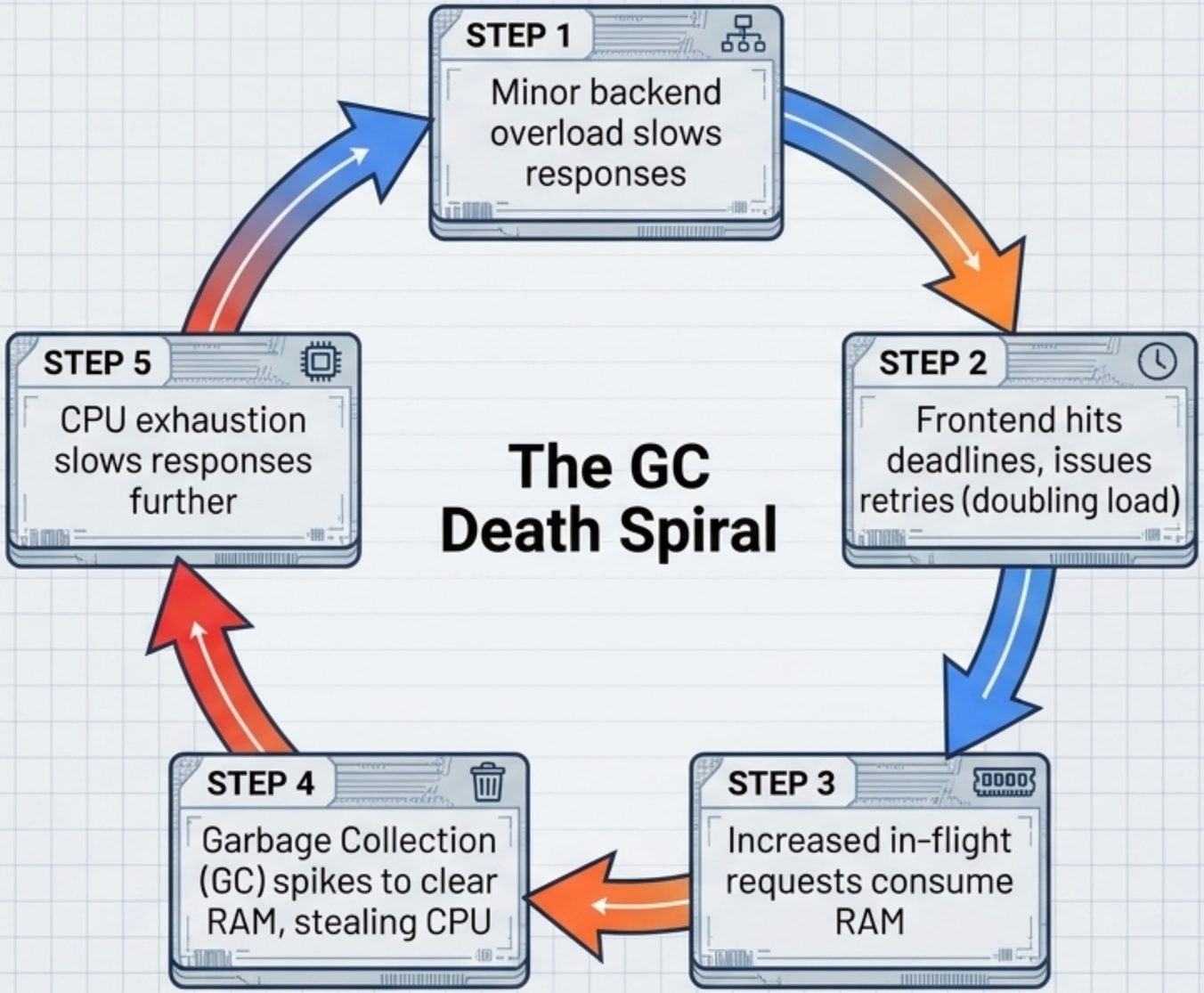
# Replication Lag & Causal Anomalies




**Causal Routing:** Route reads to the Primary for X seconds after a write by the same user, or use session tokens to verify replication log sequence numbers (LSN).

Gemini Notebook

# The Downward Spiral of Retries



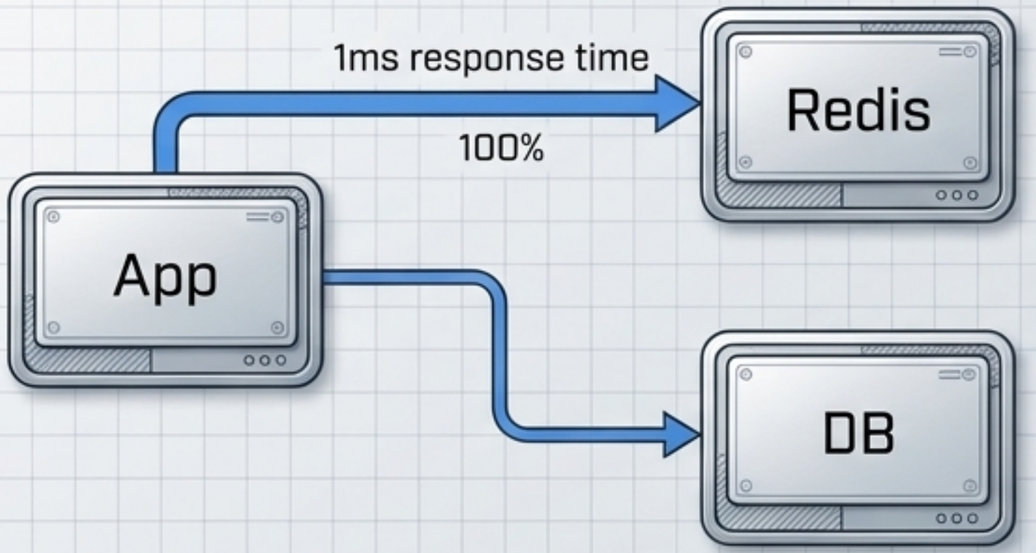

## Google SRE Principles

1. Always use Randomized Exponential Backoff.
2. Implement strict Server-Wide Retry Budgets (e.g., max 60 retries/min).
3. Use deadline propagation down the stack.

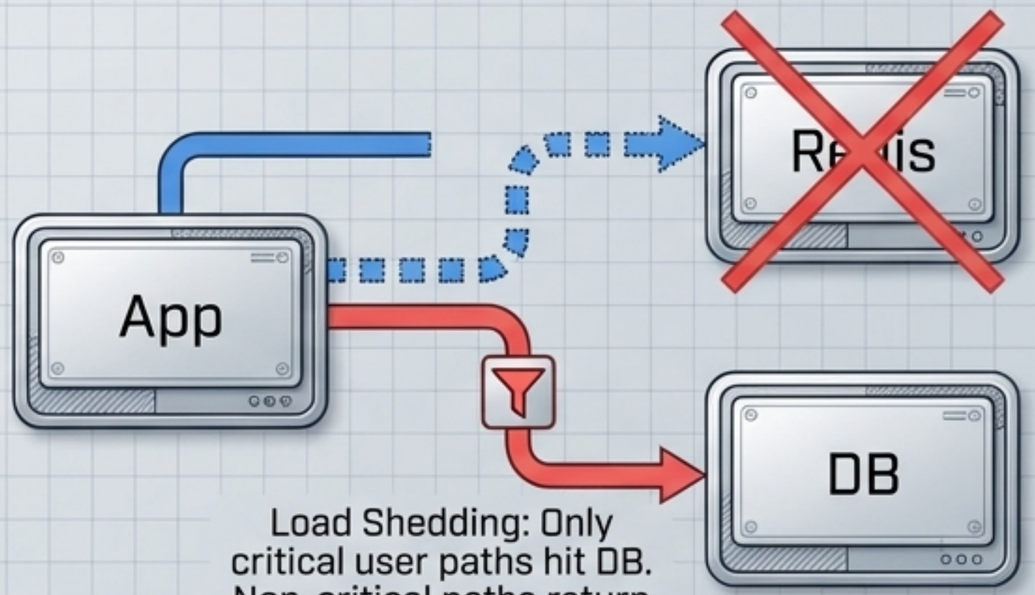
Gemini Notebook

# Designing for Degradation: Caches as Soft Dependencies

## State 1: Normal Operation




## State 2: Cache Outage / Degradation




Load Shedding: Only critical user paths hit DB. Non-critical paths return HTTP 503 or degraded UI.

‘If at first you don’t succeed, back off exponentially.’

A resilient architecture treats caches as performance accelerators, not hard availability dependencies.
If the cache dies, the system must gracefully degrade, not violently collapse.

Gemini Notebook