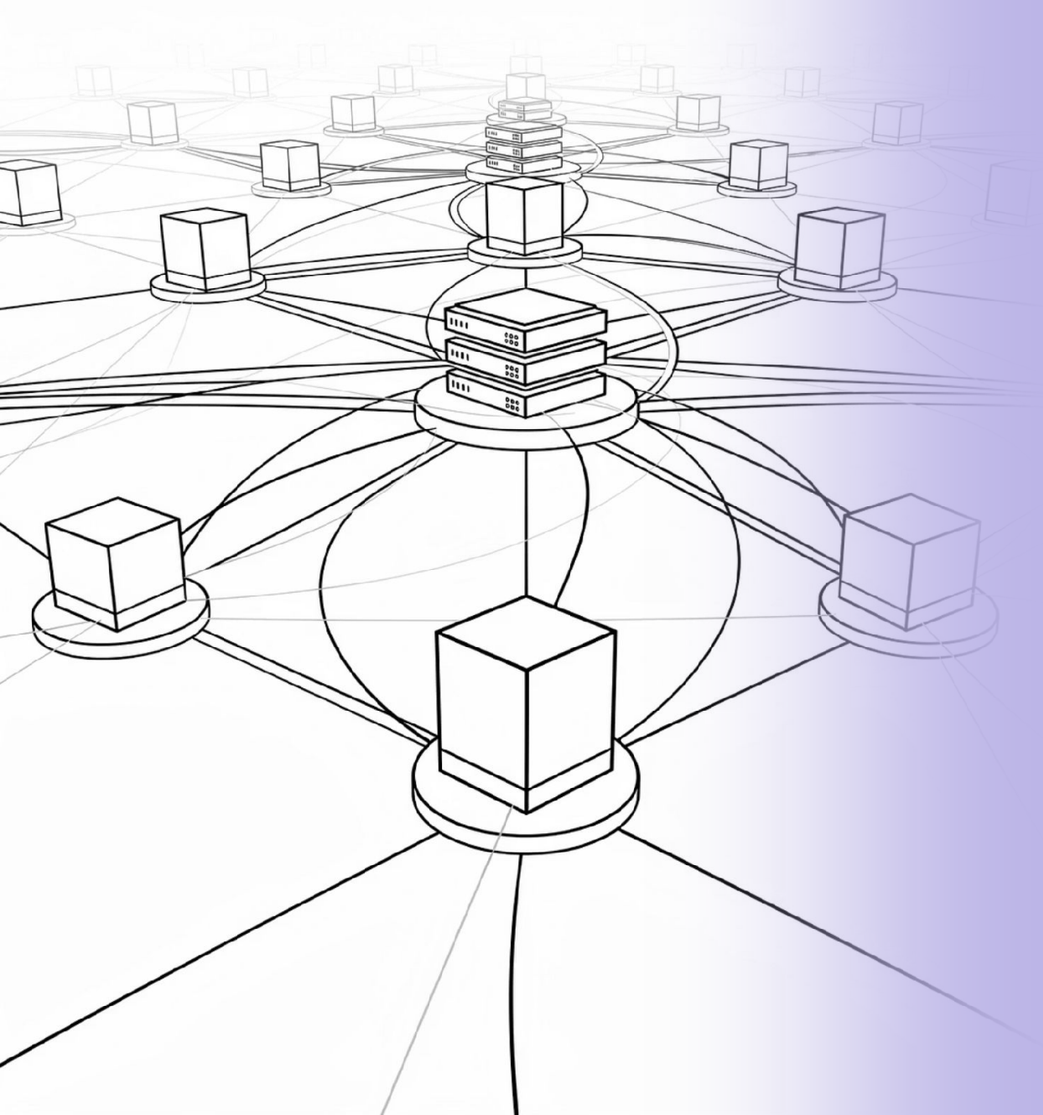


Two Hard Things: Naming Things & Cache Invalidation

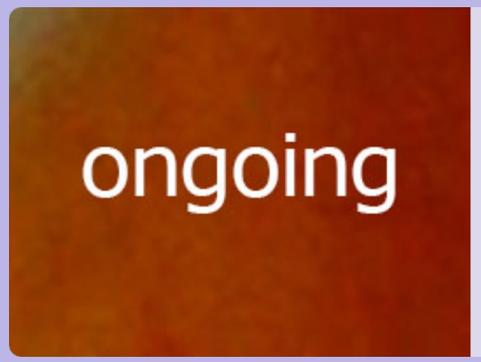
Valkey Edition

AlmaLinux Day: Germany

Roberto Luna-Rojas, Senior Developer Advocate, Valkey OSS @ AWS

"There are only two hard things in Computer Science: cache invalidation and naming things."

— Phil Karlton, Netscape, ~1995



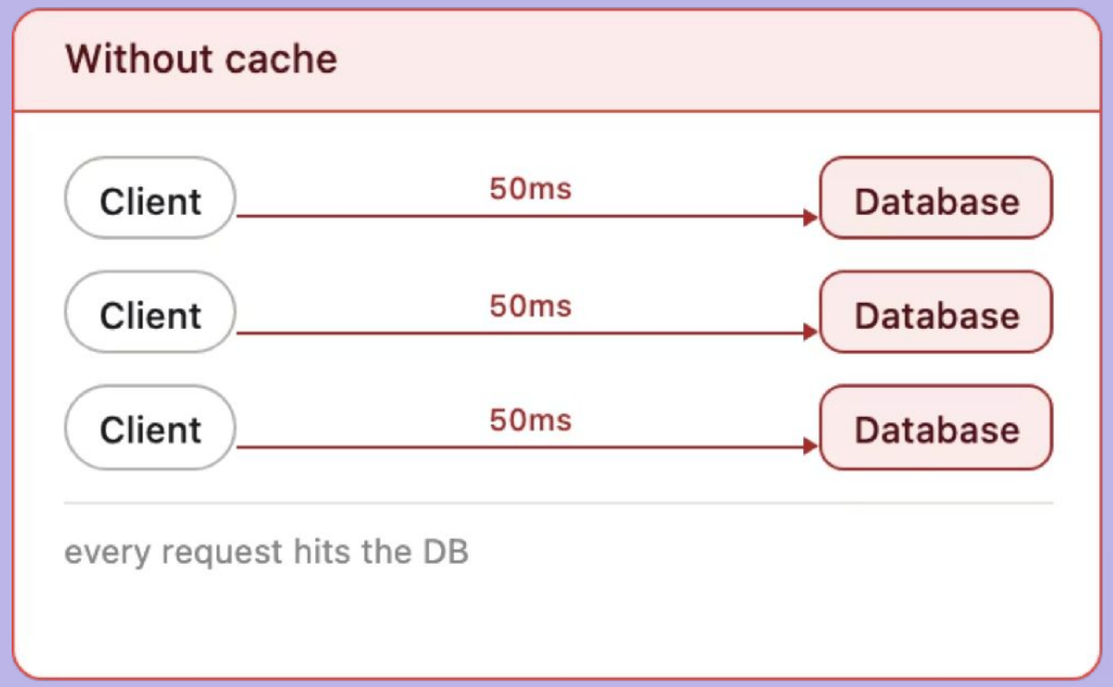
ongoing by Tim Bray

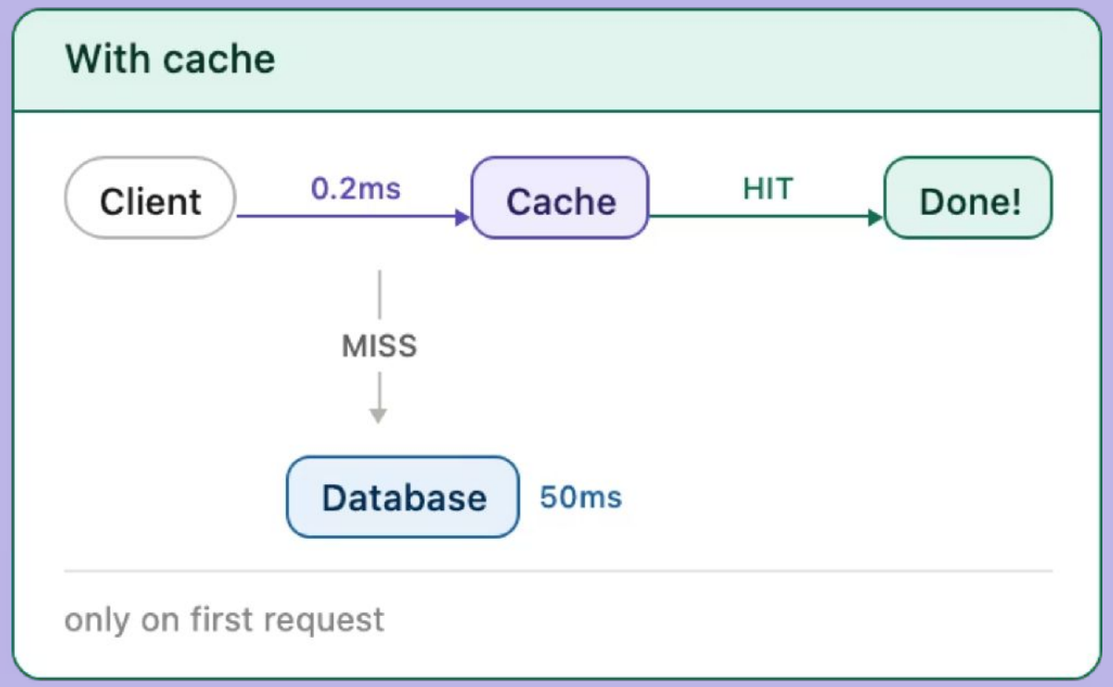
UCI

What happened was, the new cat went in for that little operation to ensure that he will be the Last of

What Is Caching?

Caching stores a copy of frequently accessed data in a fast layer so you do not have to fetch it from the slow source every time.



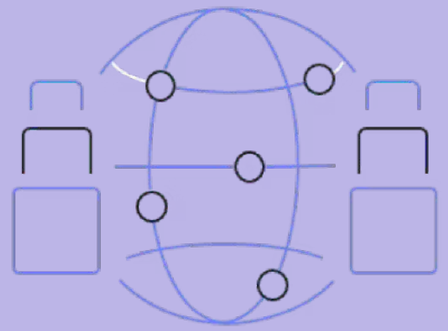


	Without Cache	With Cache
Response time	50 ms every time	0.2 ms on hit
DB load	Every request	Only on miss
Scalability	Limited by DB	Limited by memory

Introducing the Valkey community

Fully compatible
with Redis OSS 7.2

Vendor Neutral
BSD-3 Licensed



Built by contributors in the
open source community

Valkey Timeline

2024/03/20
Redis adopts dual licensing

2024/03/28
Linux Foundation launches Valkey

2024/04/16
Valkey 7.2.5 RC announced

2024/04/30
Fedora evaluates Valkey replacement

2024/09/15
Valkey 8.0.0 GA

2024/09/16
AlmaLinux at OpenSource Summit EU

2025/03/31
Valkey 8.1.0 GA

2025/10/21
Valkey 9.0.0 GA

> Valkey is integral to our mirror list, it's the brain for our geographic distribution backend; responses are within 3.5ms. Without this caching, requests to our mirror list would take about 100x longer, literally. Resource consumption would also be way higher.

— Jonathan Wright, AlmaLinux OS & Fedora EPEL

You Walk Out With:

1

Printable Key Naming Taxonomy

A printable key naming taxonomy you can adopt Monday morning

2

Decision Tree

A decision tree for choosing the right invalidation strategy

3

Real Benchmark Numbers

Real benchmark numbers, not theoretical ones

The Hidden Cost of Naive Caching

30–60% of cache entries are NEVER read a 2nd time before expiry
(Pelikan/Twitter research)

Poorly named keys cause ~15–25% duplicate storage
(same data, different key permutations)

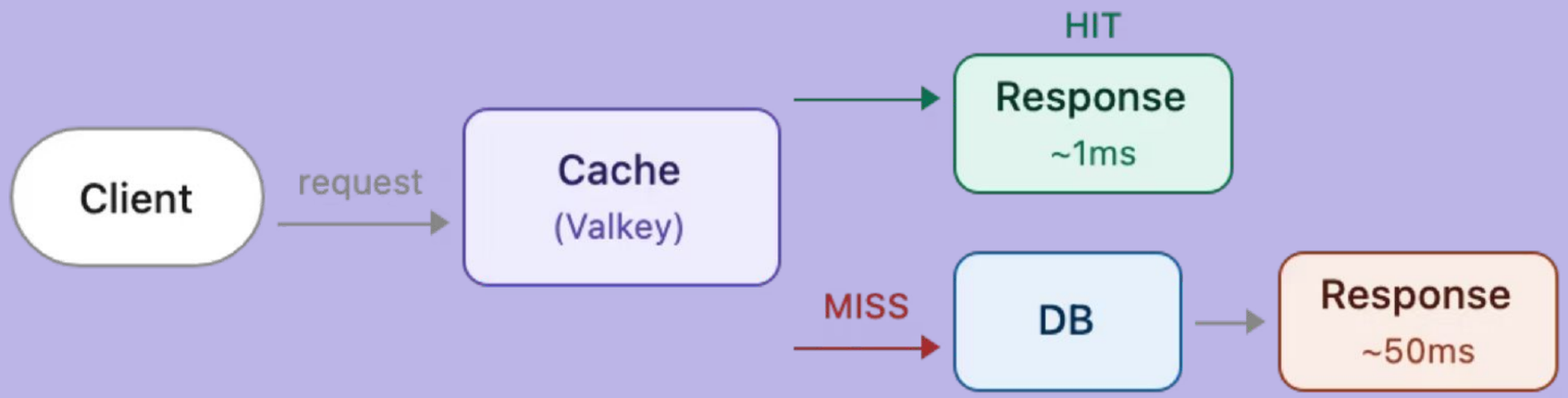
"Thundering herd" on expiry can 10x DB load in < 500ms

Stale cache serving = silent data corruption
(avg detection time: 47 min)

Wasted memory ≈ wasted energy ≈ wasted $

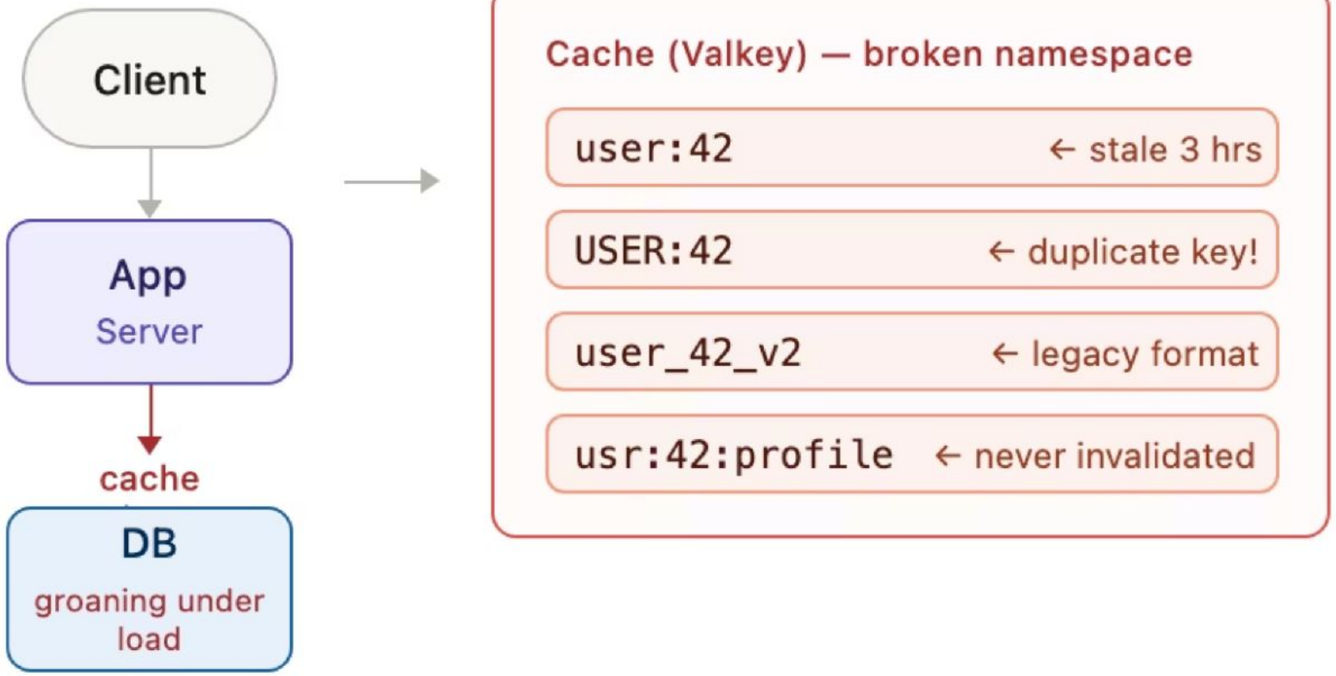
What We Think vs. What We Actually Have

What we think:



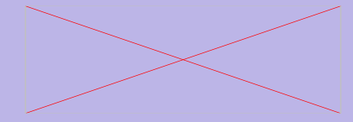
What We Think vs. What We Actually Have 2

What we actually have:



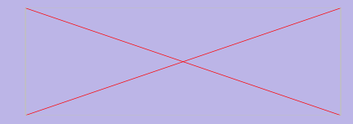
The Five Caching Mistakes

#	Mistake	Consequence
1	No key naming contract	Duplicate storage, collisions



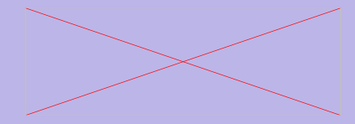
The Five Caching Mistakes

#	Mistake	Consequence
1	No key naming contract	Duplicate storage, collisions
2	TTL as only strategy	Stale reads, thundering herd



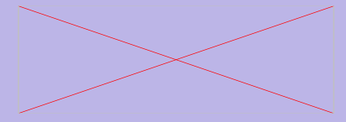
The Five Caching Mistakes

#	Mistake	Consequence
1	No key naming contract	Duplicate storage, collisions
2	TTL as only strategy	Stale reads, thundering herd
3	Cache-aside everywhere	Race conditions on write



The Five Caching Mistakes

#	Mistake	Consequence
1	No key naming contract	Duplicate storage, collisions
2	TTL as only strategy	Stale reads, thundering herd
3	Cache-aside everywhere	Race conditions on write
4	No observability	Silent memory bloat

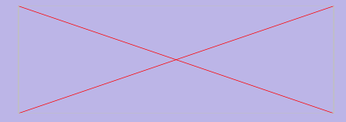


The Five Caching Mistakes

#	Mistake	Consequence
1	No key naming contract	Duplicate storage, collisions
2	TTL as only strategy	Stale reads, thundering herd
3	Cache-aside everywhere	Race conditions on write
4	No observability	Silent memory bloat
5	No invalidation graph	Cascading stale data

These five mistakes compound. Mistake #1 makes Mistake #5 impossible to fix.

Naming and invalidation are not separate problems - they are the same problem viewed from two angles.



Key Taxonomy

The Grammar:

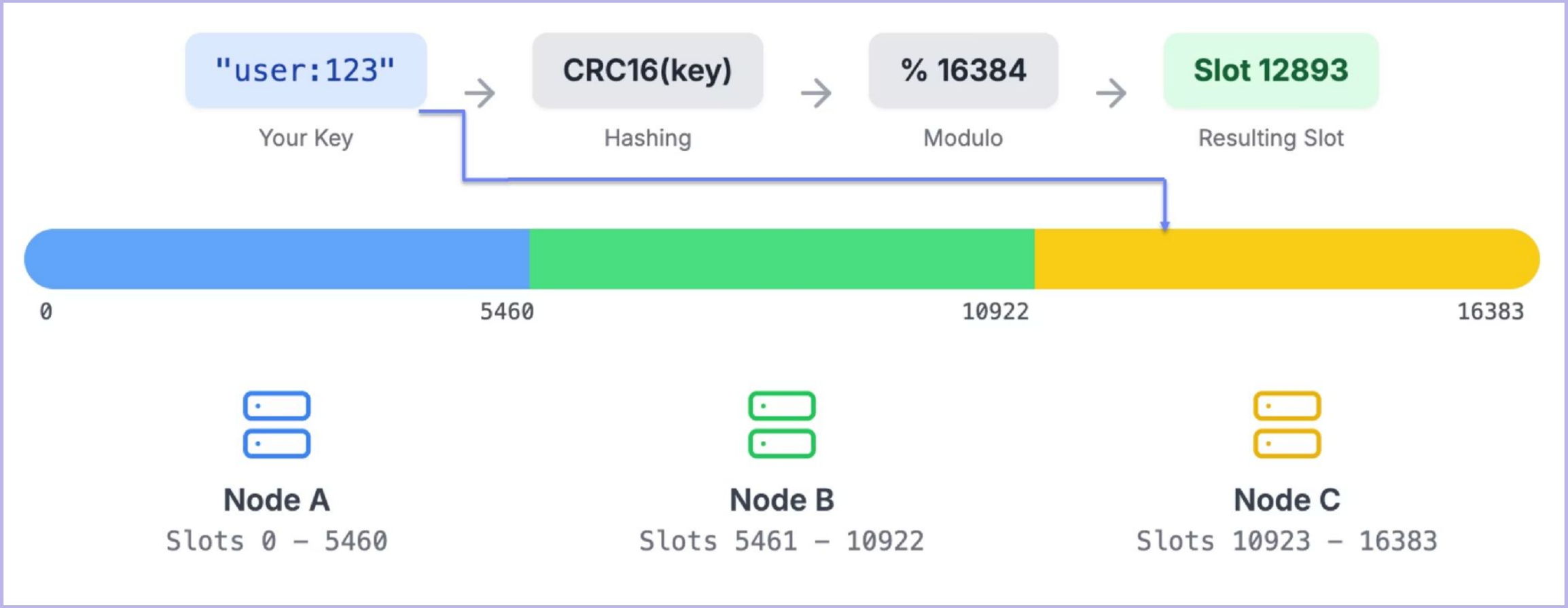
```
<namespace>:<entity>:<identifier>[:<qualifier>][:<version>]
```

Key Naming Rules (Non-Negotiable)

Rule	Rationale
Lowercase only	Case bugs are invisible until prod
Colons as separators	Cluster hash tags {} friendly, SCAN MATCH friendly
No spaces, no special chars except `:` and `-`	RESP3 parsing edge cases
Namespace must be first	Enables `SCAN 0 MATCH inventory:*` without full keyspace scan
Max 200 bytes	Large keys bloat the key table
Version your keys	You version your REST APIs. Version your cache keys.

Keys are an API contract. You wouldn't ship a REST API without a schema.
Don't ship cache keys without one either.

How Valkey Cluster Works: The Slots



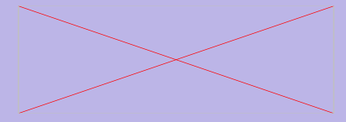
Cluster Hash Tags & Key Builder {}

Hash slot awareness, group by identifier:

```
# These two keys land on the SAME hash slot => same shard
catalog:product:{sku-82917}:details:v2
catalog:product:{sku-82917}:pricing:v2
# Enables atomic MULTI/EXEC or Lua across related keys
```

Avoid having all your keys in a single shard, group by entity:

```
# These three keys land on the SAME hash slot as well as all other products
catalog:{product}:sku-82917:details:v2
catalog:{product}:sku-93028:pricing:v2
catalog:{product}:sku-14139:pricing:v1
```



Key Builder Utility (Python)

```python
class ValkeyKeyBuilder:
    """Deterministic, collision-free key generation."""
    def __init__(self, namespace: str, version: int = 1):
        self.namespace = namespace
        self.version = f"v{version}"

    def build(self, entity: str, identifier: str | list[str],
              qualifier: str | None = None, hash_tag: bool = False) ->
str:    if isinstance(identifier, list):
            identifier = ":".join(sorted(identifier))
        ident = f"{{{identifier}}}" if hash_tag else identifier
        parts = [self.namespace, entity, ident]
        if qualifier:
            parts.append(qualifier)
        parts.append(self.version)
        return ":".join(parts)

keys = ValkeyKeyBuilder("catalog", version=2)
keys.build("product", "sku-82917", "details")
```

Key Builder Utility (Python): Usage

```python
# Usage
keys = ValkeyKeyBuilder("catalog", version=2)

keys.build("product", "sku-82917", "details")
# => "catalog:product:sku-82917:details:v2"

keys.build("product", "sku-82917", "details",
           hash_tag=True)
# => "catalog:product:{sku-82917}:details:v2"

keys.build("compare", ["sku-100", "sku-42"], "diff")
# => "catalog:compare:sku-100:sku-42:diff:v2"   (sorted!)
```

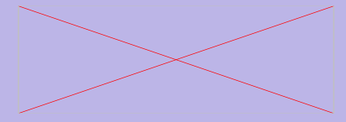
Anti-Pattern Gallery

Anti-Pattern	Example	Why It Hurts
Environment in key	`cache:prod:user:42`	Same code, different keys per env. Use separate Valkey instances instead.
Serialized query as key	`SELECT * FROM p WHERE color=red AND size=L`	Non-deterministic order: WHERE a=1 AND b=2 ≠ WHERE b=2 AND a=1. Hash the canonical form.
Timestamp in key	`user:42:1719283200`	Un-scannable, un-invalidatable. Use EXPIREAT on a stable key.
Caller-generated keys (no shared library)	Every team invents its own format.	Enforce via shared key builder SDK.
Mega-key (entire object graph as JSON blob)	1 field change invalidates 200KB blob.	Use Valkey Hash per entity.

Hash vs. String: The Mega-Key Problem

Strategy A: String
(Mega-Key)

```
SET catalog:product:sku-82917:v2
'{"name":"Widget","price":29.99,"stock":142,...}'

Price changes?
GET, deserialize, mutate, serialize, SET entire blob
Race condition window: ~2ms
```



Hash vs. String: The Mega-Key Problem

Strategy A: String (Mega-Key)

```
SET catalog:product:sku-82917:v2
'{"name":"Widget","price":29.99,"stock":142,..
.}'

Price changes?
GET, deserialize, mutate, serialize, SET
entire blob
Race condition window: ~2ms
```

Strategy B: Valkey Hash

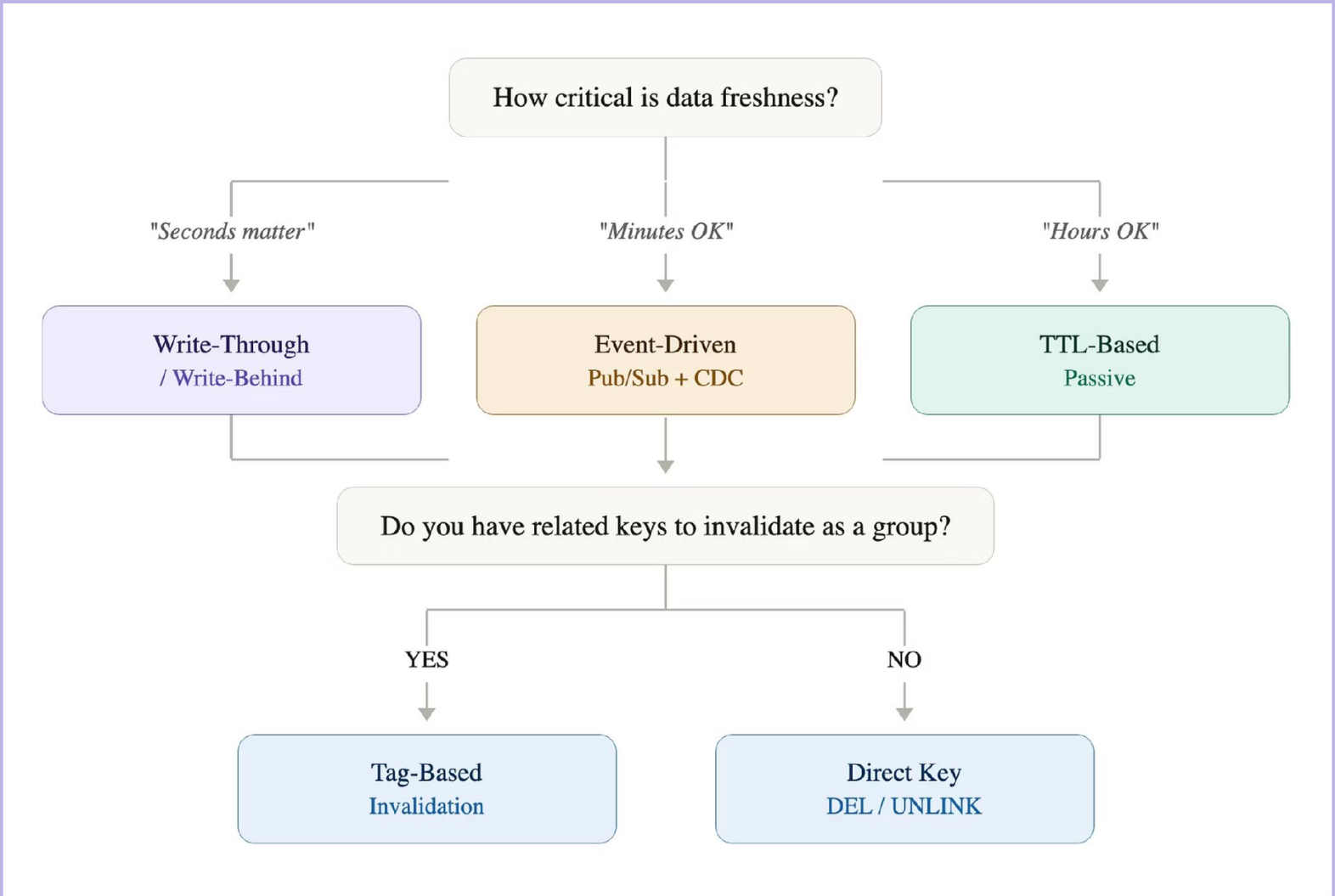
```
HSET catalog:product:{sku-82917}:v2
  name "Widget" price "29.99" stock "142"

Price changes?
HSET catalog:product:{sku-82917}:v2 price
"31.99"
Atomic, single field, ~0.1ms
```

Operation	String/JSON	Hash HSET	Improvement
Single field update	0.48 ms	0.07 ms	6.8× faster
Network bytes (RTT)	2,140 B	78 B	27.4× smaller
Invalidation blast	Entire obj	None	Eliminated
Race condition risk	High (RMW)	None	Eliminated

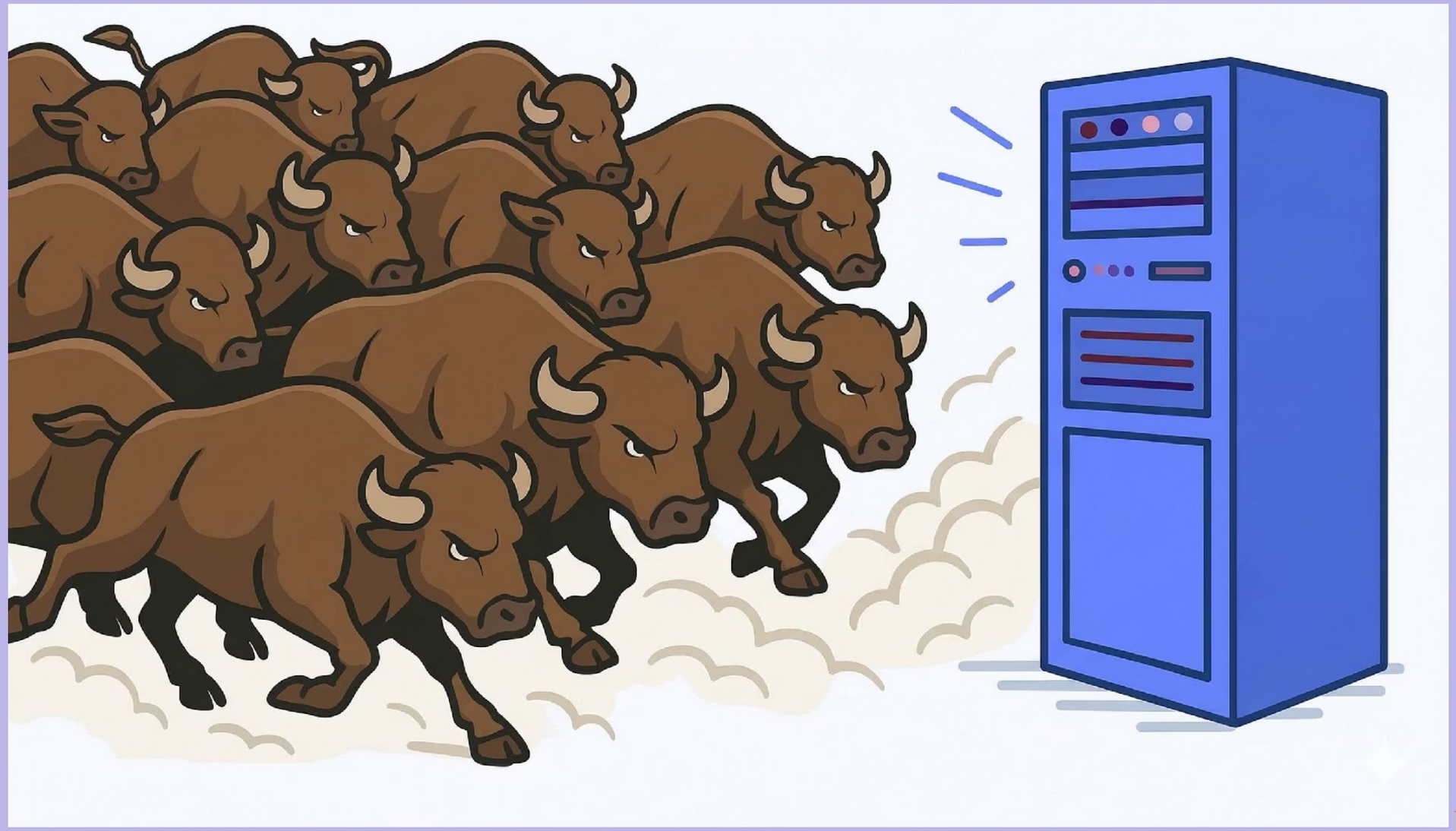
The Four Strategies at a Glance

Strategy	Stale data OK?	Write complexity	Read complexity	Valkey Primitives
TTL (passive expiry)	Yes, bounded	Low	Low	`EX, PX, EXPIREAT`
Event-driven (pub/sub, CDC)	No / near-zero	Medium	Low	`PUBLISH, Keyspace Notif.`
Tag-based	No, but grouped	Medium	Low	`SADD, SMEMBERS, UNLINK`
Write-through	Never	High	Lowest	`FUNCTION, MULTI/EXEC`
Write-behind	Eventual	High	Lowest	`FUNCTION, MULTI/EXEC`

The Decision Tree



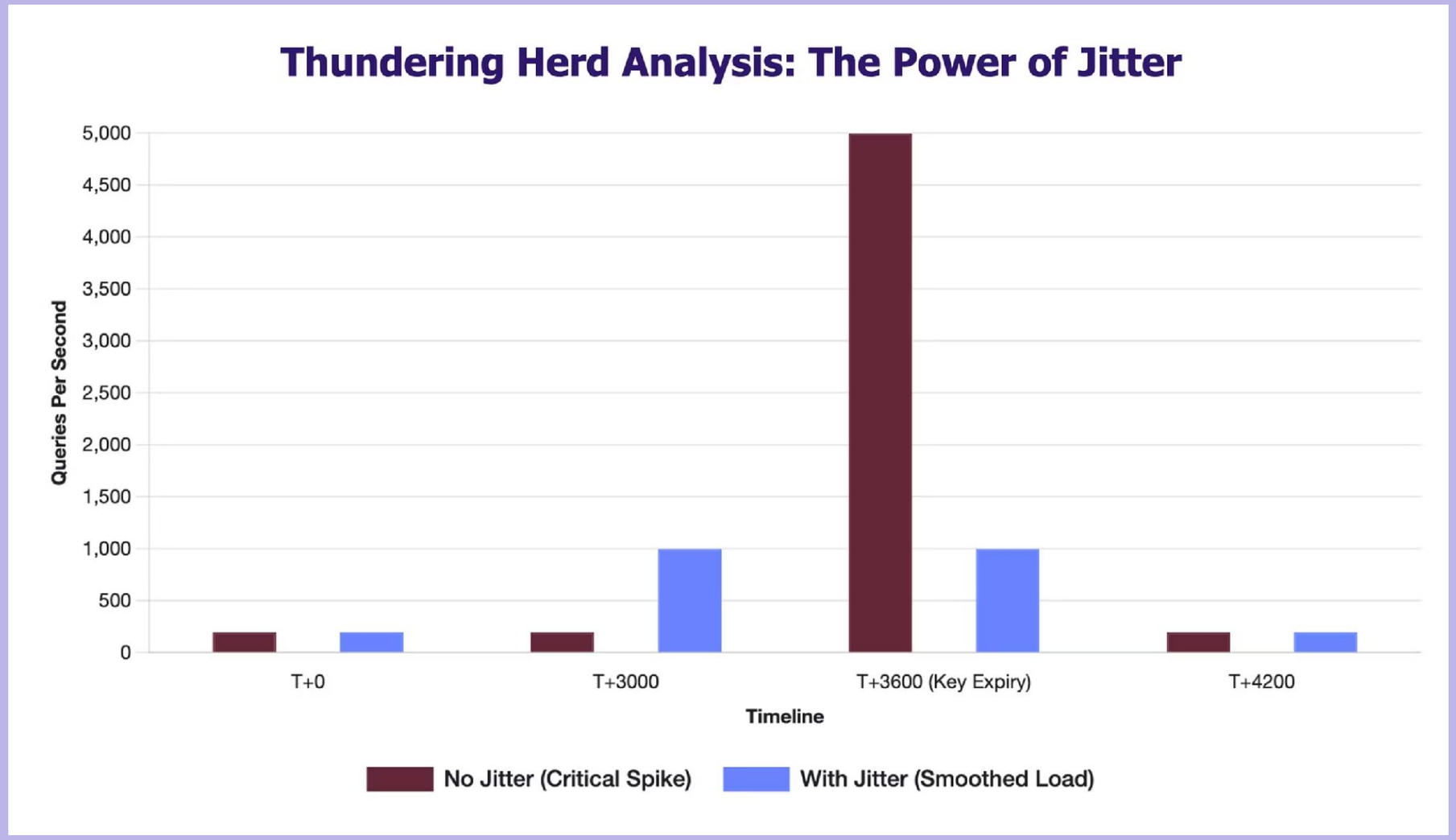
The Thundering Herd Problem



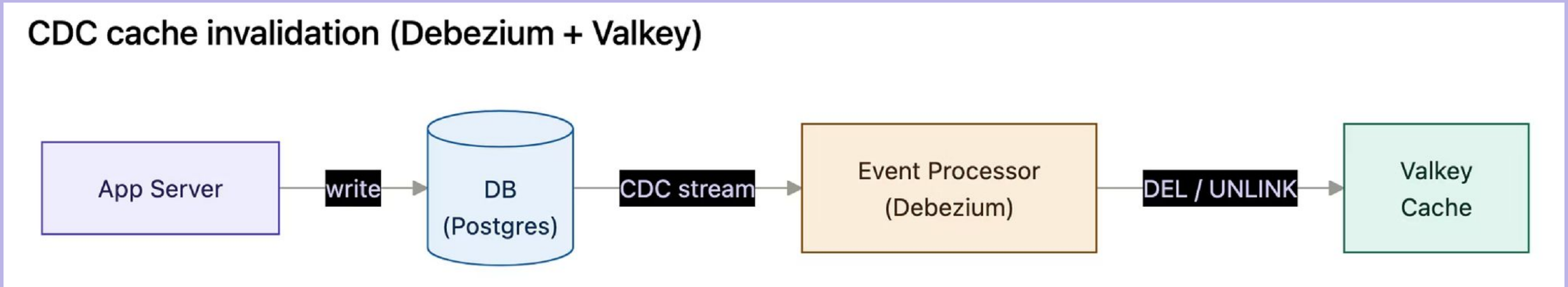
Strategy 1: TTL + Jitter + XFetch

```python
import valkey.asyncio as valkey
import random, time, math

client = valkey.Valkey(host="my-cluster.valkey.example.com", port=6379)

# Jittered TTL (prevent thundering herd)
BASE_TTL = 3600  # 1 hour
JITTER   = 600   # ±10 minutes
ttl = BASE_TTL + random.randint(-JITTER, JITTER)
await client.set(key, payload, ex=ttl)

# Probabilistic Early Expiration (XFetch algorithm)
async def xfetch(key: str, ttl: int, beta: float = 1.0, recompute_fn=None):
    """Optimal probabilistic cache stampede prevention."""
    raw = await client.get(key)
    if raw:
        value, delta, expiry = decode_with_metadata(raw)
        now = time.time()
        gap = expiry - now
        if gap > 0:
            rand = -delta * beta * math.log(random.random())
            if rand < gap:
                return value  # Still fresh enough
    # Recompute
    start = time.time()
    value = await recompute_fn()
    delta = time.time() - start
    expiry = time.time() + ttl
    await client.set(key, encode_with_metadata(value, delta, expiry),
ex=ttl)
    return value
```

Thundering Herd: Jitter vs. No Jitter



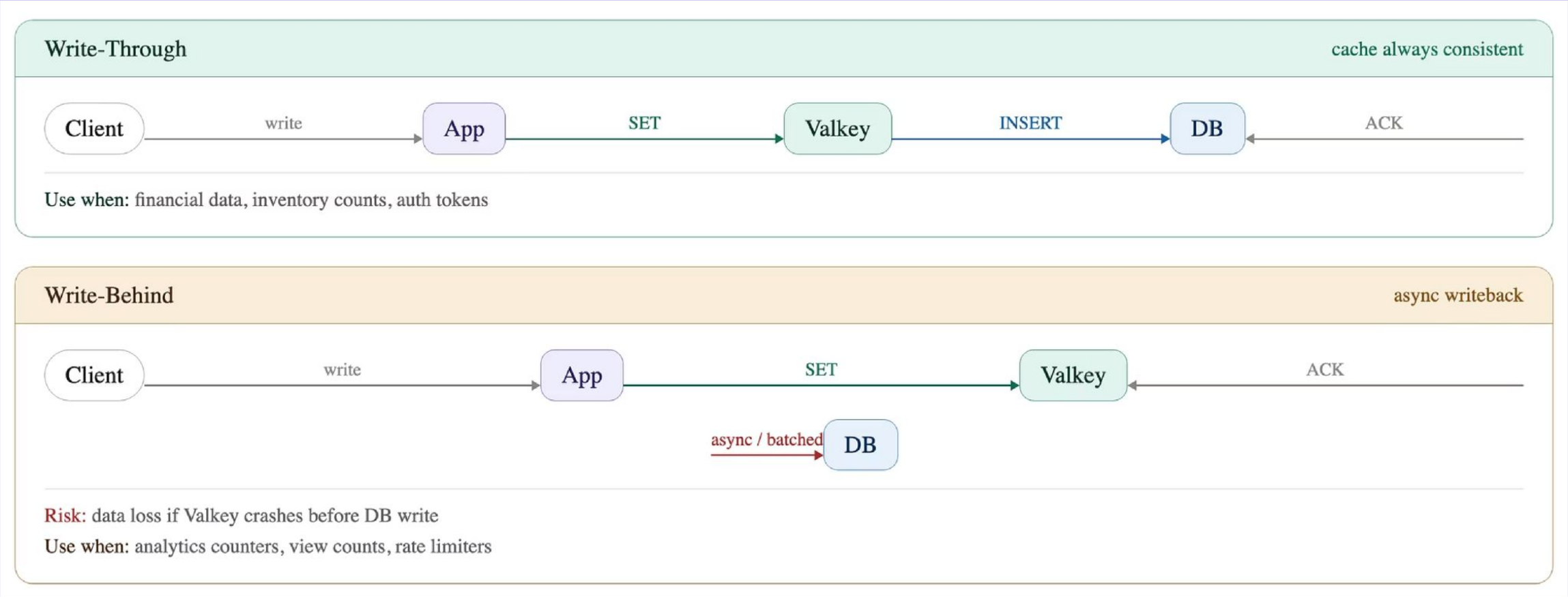
Strategy 2: Event-Driven Invalidation



Strategy 2: Event-Driven Invalidation (cont)

```python
# Publisher side (write path)
async def update_product(product_id: str, changes: dict):
    await db.execute(
        "UPDATE products SET price=$1, updated_at=NOW() WHERE id=$2",
        changes["price"], product_id)
    event = json.dumps({"action": "invalidate", "entity": "product",
                        "id": product_id, "fields": list(changes.keys())})
    await valkey_client.publish("cache:invalidation:catalog", event)

# Subscriber side (invalidation worker)
async def invalidation_listener():
    pubsub = valkey_client.pubsub()
    await pubsub.subscribe("cache:invalidation:catalog")
    async for message in pubsub.listen():
        if message["type"] != "message": continue
        event = json.loads(message["data"])
        key_builder = ValkeyKeyBuilder("catalog", version=2)
        primary_key = key_builder.build(event["entity"], event["id"],
"details")await valkey_client.unlink(primary_key)
```

Strategy 3: Write-Through / Write-Behind

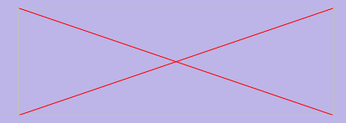


Strategy 4: Tag-Based Invalidation

Problem: A product appears in 5+ cached views. How do you find and invalidate all of them?

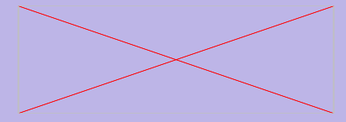
Solution: Inverted tag index using Valkey Sets

```python
class TaggedCache:
    """Tag-based cache invalidation using Valkey Sets."""
    def __init__(self, client): self.client = client

    async def set_with_tags(self, key: str, value: str, tags: list[str], ttl: int =
3600):  pipe = self.client.pipeline(transaction=True)
        pipe.set(key, value, ex=ttl)
        for tag in tags:
            tag_key = f"tag:{tag}"
            pipe.sadd(tag_key, key)
            pipe.expire(tag_key, ttl + 300)  # tag outlives members
        await pipe.execute()

    async def invalidate_by_tag(self, tag: str) -> int:
        tag_key = f"tag:{tag}"
        members = await self.client.smembers(tag_key)
        if not members: return 0
        pipe = self.client.pipeline(transaction=False)
        for key in members:
            pipe.unlink(key)       # non-blocking delete
        pipe.unlink(tag_key)        # clean up the tag itself
        await pipe.execute()
        return len(members)

# When product sku-82917 changes:
deleted = await cache.invalidate_by_tag("product:sku-82917")
# => Atomically removes ALL cached views containing this product
```

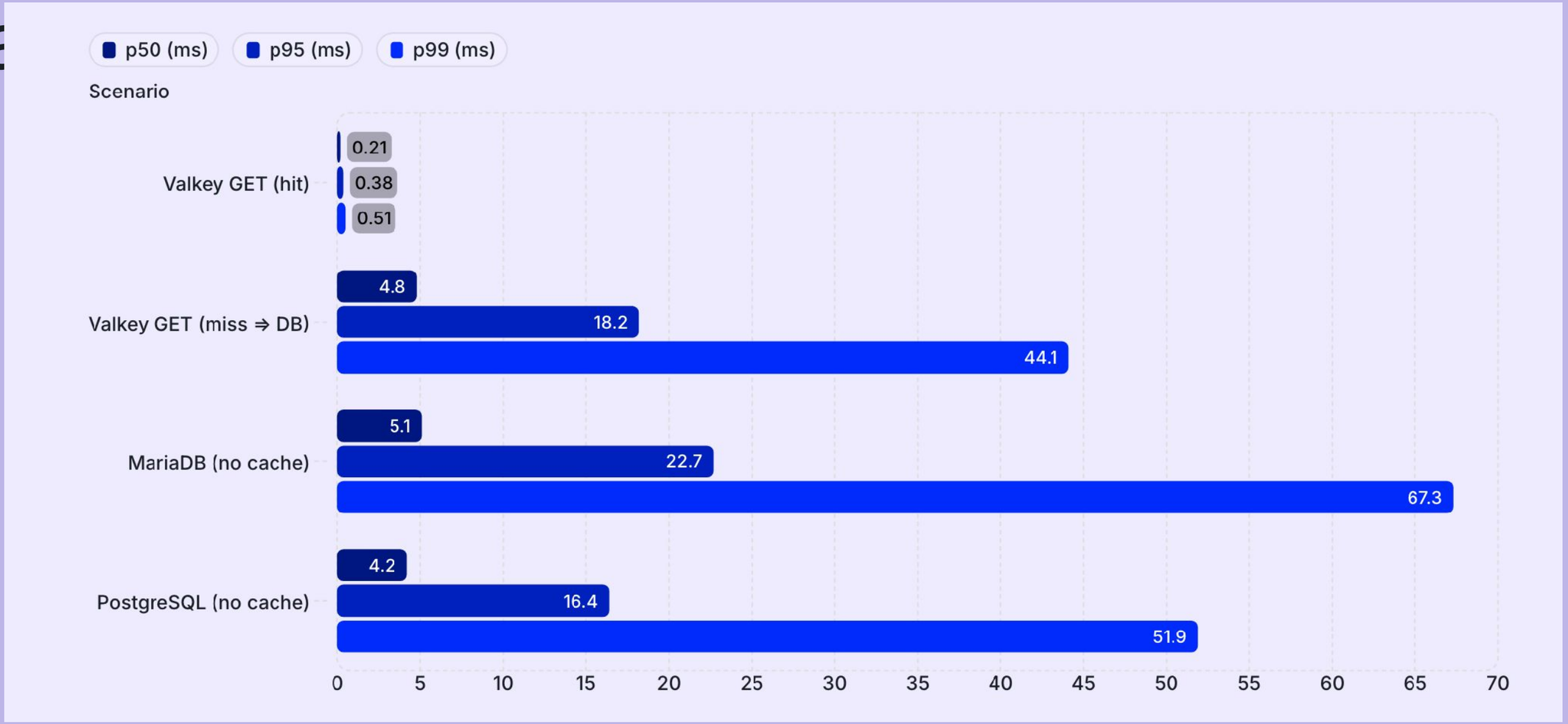


Benchmark Environment

Parameter	Value
Valkey Version	9.0 (OSS, single shard + cluster)
Instance	Amazon EC2 c7g.xlarge (ARM/Graviton3)
Client	valkey-benchmark + custom Python client
Network	Same AZ, enhanced networking
Data Profile	1M keys, avg value size 1.2KB
Workload	80% read / 20% write (YCSB Workload A)
Concurrency	100 connections, pipelining=10



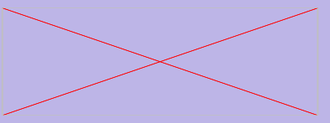
Cache Hit vs. Miss La



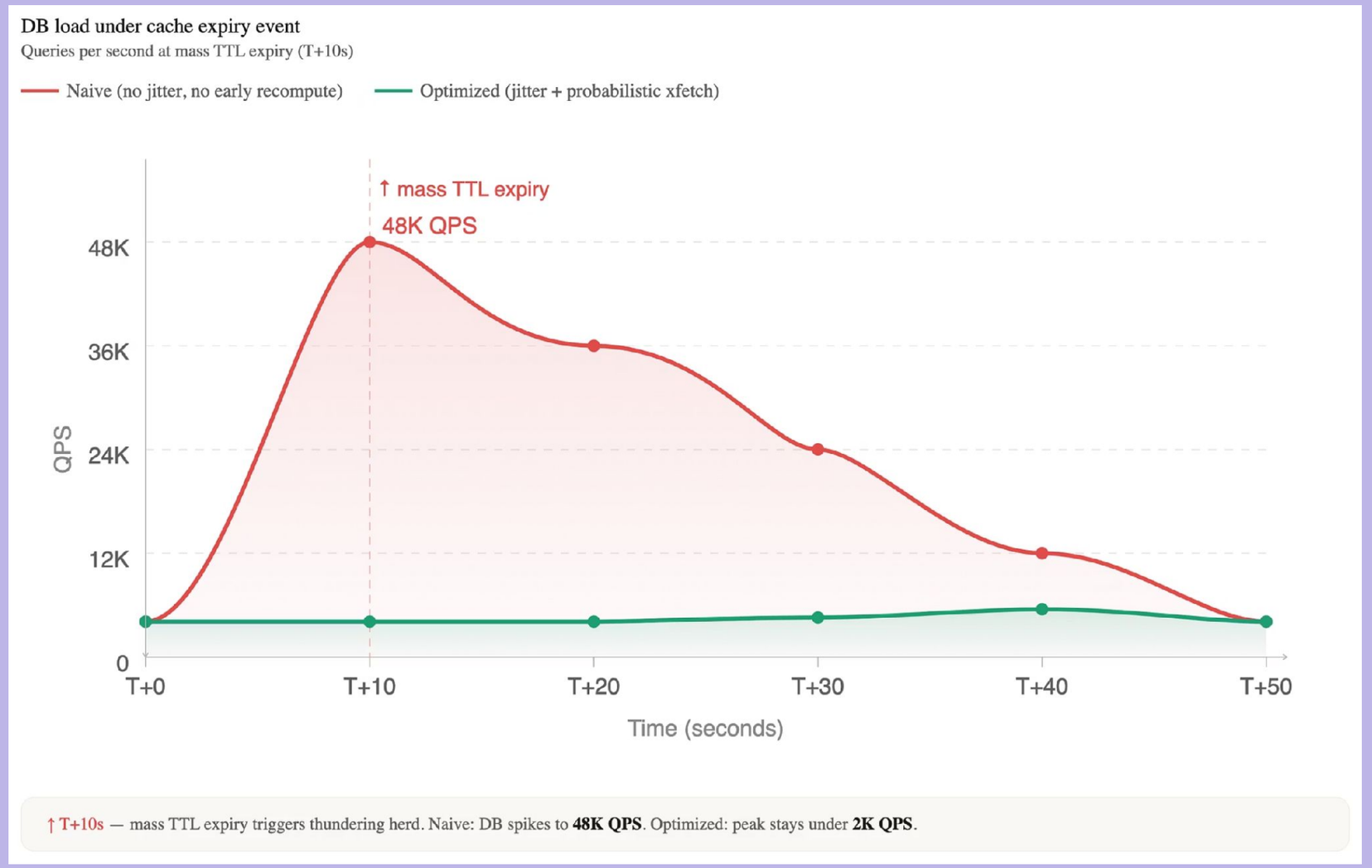
RPS: Valkey GET (hit) 412,000 | Valkey GET (miss ⇒ DB) 21,000 | MariaDB (no cache) 18,500 | PostgreSQL (no cache) 23,100

Naive vs. Optimized

Metric	🔴 Naive Cache	🟢 Optimized (Valkey)
Effective Hit Rate	62%	97.3%
Stale Read Rate	~18%	< 0.1%
DB Queries/sec (steady)	12,400	1,850
DB Queries/sec (burst)	48,000 (herd)	2,100 (jitter+xfetch)
Avg Latency (p50)	4.2 ms	0.3 ms
Tail Latency (p99)	89 ms	1.8 ms
Memory Efficiency	2.1 GB (dupes)	0.9 GB
Network Bytes/sec	380 MB/s	41 MB/s



DB Load Under Cache Expiry Event



The Complete Decision Matrix

Dimension	TTL + Jitter	Event-Driven	Write-Through	Tag-Based
Freshness SLA	Minutes	Seconds	Immediate	Seconds
Complexity	Low	Medium	Medium-High	Medium
Infra Overhead	None	CDC/Queue	Lua/Function	Tag Sets
Consistency	Eventual	Eventual+	Strong*	Eventual+
Failure Mode	Stale	Delayed invalidation	Write fails	Orphan tags
Memory Overhead	Baseline	Baseline	Baseline	+5–15% tags
Best For	Static content	Entity CRUD	Financial, inventory	Aggregations, search

Combine with ⇒ Always (floor) + TTL safety net + TTL safety net + Event trigger

* Strong consistency in write-through requires synchronous DB commit before ACK.

Three Takeaways: What to Do Monday Morning

1 Adopt a Key Naming Taxonomy

```
<namespace>:<entity>:<id>[:<qualifier>][:<version>]
```

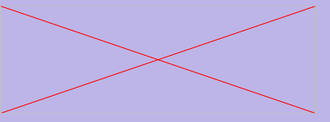
Ship a shared key builder library. Enforce in code review.

2 Match Invalidation Strategy to Data Sensitivity

Use the decision tree. TTL+jitter is the floor, not the ceiling. Add event-driven or tag-based where the data demands it.

3 Observe Your Cache Like You Observe Your Database

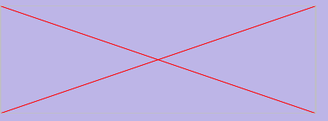
Hit rate, eviction rate, memory fragmentation, stale read rate. If you can't measure it, you can't fix it. Below 50% hit rate ⇒ you don't have a cache, you have an expensive network hop.



Resources & Q&A

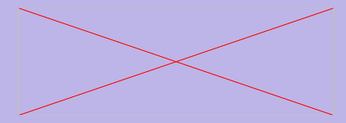
Resource	URL
Valkey official docs	https://valkey.io/docs/
Valkey GitHub	https://github.com/valkey-io/valkey
Keyspace notifications	https://valkey.io/topics/notifications/
Pub/Sub reference	https://valkey.io/commands/subscribe/
Linux Foundation Valkey	https://www.linuxfoundation.org/press/linux-foundation-launches-open-source-valkey-community

GitHub **@rlunar** - feedback welcome



Appendix A: Reproducible Benchmark Commands

```
# Environment Setup
# Valkey 9.0 on Amazon Linux 2023, c7g.xlarge (4 vCPU, 8 GiB)
docker run -d --name valkey -p 6379:6379 valkey/valkey:9-alpine \
 valkey-server --maxmemory 4gb --maxmemory-policy allkeys-lfu \
 --activedefrag yes --lazyfree-lazy-eviction yes --lazyfree-lazy-expire
yes
# Populate 1M keys
valkey-benchmark -h 127.0.0.1 -p 6379 \
 -t set -n 1000000 -r 1000000 -d 1200 -P 32 --threads 4

# Workload A (80/20 read/write)
valkey-benchmark -h 127.0.0.1 -p 6379 \
 -n 5000000 -r 1000000 -c 100 -P 10 --threads 4 \
 -- HSET myhash:__rand_int__ field1 __rand_str_1200__

# Latency histogram
valkey-cli --latency-history -i 5
valkey-cli --intrinsic-latency 10

# Memory analysis
valkey-cli MEMORY DOCTOR
valkey-cli INFO memory
valkey-cli --memkeys --memkeys-samples 100
```



Roberto Luna-Rojas, Sr Developer Advocate for Valkey at AWS OSS

Find me at /in/robertolunarojas/, @RobertLunaRojas and https://github.com/rlunar/

Thank You

Questions? Let's talk caching.