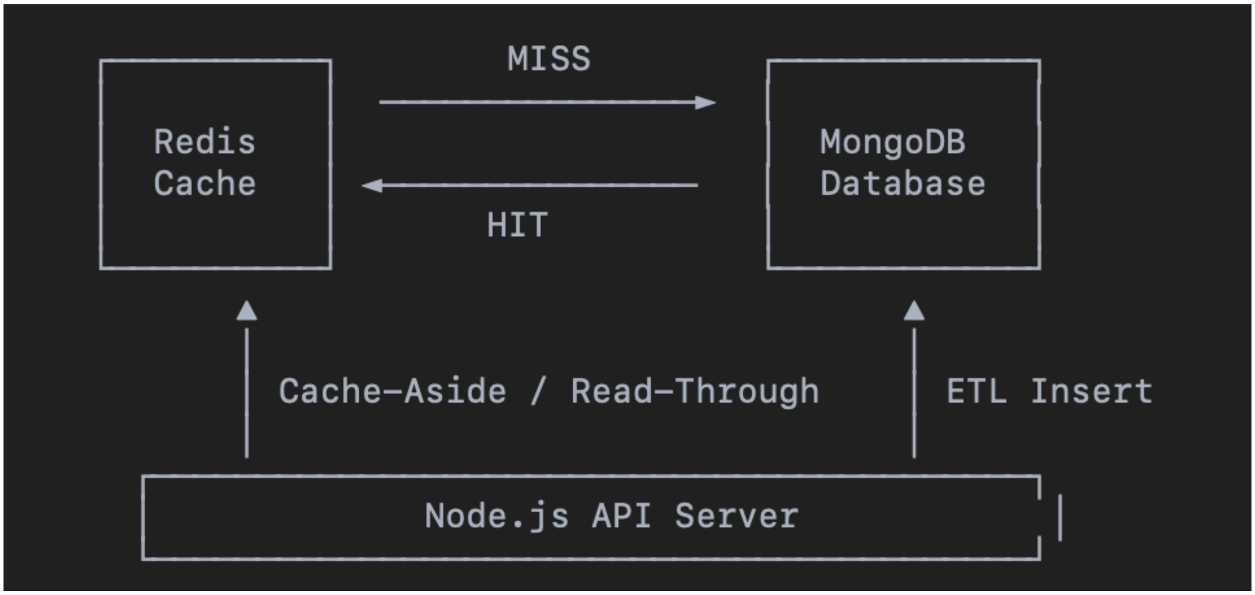
Title: *Lab 4: Caching Architecture & ETL Flow*



• Node.js API fetches all sensor readings
The Node.js server acts as the main gateway. Every request for a sensor value flows through it, and it decides whether to serve from Redis or fall back to MongoDB.

• Cache-Aside & Read-Through both check Redis first
Before touching MongoDB, both strategies try to pull the value from Redis to save time. Cache-Aside loads into Redis only after a miss; Read-Through automatically fills Redis whenever data is missing.

• MongoDB remains the system of record
Even though Redis speeds everything up, MongoDB still stores the actual data long-term. Redis only holds the “fast copies.”

• TTL ensures stale values get refreshed automatically
TTL-based caching forces entries to expire after a few seconds. Once expired, the next request triggers a clean reload from MongoDB.

• ETL pipeline stores raw data in MongoDB, Redis caches only on demand
Incoming sensor readings are written directly to MongoDB. Redis is not preloaded—values only appear in Redis when your app explicitly reads them and caches them afterward.

— Lessons Learned

• Built and tested multiple caching strategies in one pipeline
Implementing Cache-Aside, Read-Through, and TTL helped me understand how each pattern behaves differently and where each one is useful.

• Redis massively reduces response time
Before caching, MongoDB queries averaged around 55 ms. After caching, Redis hits dropped to around 1 ms. The performance difference was obvious and measurable.

• Learned how to structure clean Node.js cache helper functions
Breaking the logic into small, reusable functions (like readThrough() and cache setters) made the server easier to debug and extend.

• Understood how TTL affects real-time performance
Seeing requests slow down immediately after a TTL expiry (fresh reload) and then speed up again once Redis is populated helped reinforce how expiration-based caching works.

• Saw how important cache-DB consistency is
If the database changes but Redis doesn’t get updated, you serve stale data. Handling this consistently is a real design challenge.

• Learned core distributed system ideas: freshness, invalidation, and cache coherence
This project made these concepts real instead of theoretical—especially when I had to clear or reset Redis to simulate fresh reads.

What Worked Well

• Redis integration worked smoothly once the setup was stable
After correctly configuring the Redis client, the connection stayed solid and data retrieval was instant.

• Cache-Aside gave consistent performance boosts
The HIT vs MISS behavior was very clear. Once cached, the API felt significantly faster.

• TTL caching behaved exactly as expected
Entries expired right on schedule, and after expiry the system correctly reloaded and recached the data.

• MongoDB queries were stable and predictable
The database always returned clean, structured documents, making the pipeline reliable.

Challenges

• Route conflicts originally caused "Cannot GET /cache-aside/… " errors
This happened because multiple versions of the same route were defined in the server file, causing Express to break. Fixing it required cleaning duplicates.

• Multiple Node.js servers were running, causing port 3000 to be locked
Before restarting, I had to manually kill the process using lsof and kill -9, otherwise nothing worked.

• Redis had to be flushed repeatedly to test proper cache behavior
To simulate fresh misses, I had to manually clear Redis keys every time using FLUSHALL or DEL. Forgetting this produced confusing results.

• TTL testing required precise timing
If I didn't wait the correct number of seconds, I would get a Redis HIT instead of an expiry reload, which made testing slightly annoying.