Architecture & ETL Data Flow

1. Request & Cache Check

The Node.js application first checks Redis for the requested data (e.g., sensor readings). This in-memory check is extremely fast.

2. Cache Miss & DB Query

If the data is not in Redis (a 'cache miss'), the application performs a query against the persistent MongoDB (AgriDB) to retrieve the source of truth.

3. Cache Write & Response

The data from MongoDB is then written *back* into Redis for future requests before being returned to the user. This makes subsequent requests for the same data instant.

Key Lessons Learned

- **Drastic Performance Gains:** Redis (in-memory) is orders of magnitude faster for read operations than a disk-based database. Caching is essential for high-traffic applications.
- **Reduced Database Load:** Caching protects the primary MongoDB database from being overwhelmed by repetitive read requests, allowing it to focus on writes and complex queries.
- **The Challenge of Staleness:** The biggest "lesson" is cache invalidation. You must have a strategy to update or delete cached data in Redis when the original data in MongoDB changes.

Challenges & Retrospective

What Worked Well

The Node.js drivers for both **`redis`** and **`mongodb`** are mature and easy to use with `async/await`. Initial setup and connection (`redis-cli ping`, Compass connection) are surprisingly straightforward. Seeding the database with 2,000 records using a script was also very efficient.

What Did Not Work Well (Challenges)

Managing two data stores increases complexity. The main challenge is **cache invalidation**. If a record is updated in MongoDB, how does Redis know? This can easily lead to stale data being served to the user if a proper update/delete strategy isn't implemented.