# Movie Recommendation System - Technical Documentation

## 1) Problem Statement

Build a simple movie recommendation system that:

* Stores movie metadata and user ratings
* Lets users rate movies and view top movies
* Produces recommendations based on other users with similar tastes
* Exposes a minimal UI and REST API

Constraints:

* Use NoSQL databases suited to each concern
* Keep architecture simple, modular, and locally runnable

## 2) System Architecture / Design

Components:

* API Service (FastAPI): `src/app/main.py`

- Endpoints for browsing, rating, top list, recommendations, and health

* UI (Streamlit): `src/streamlit\_app.py`

- Minimal front-end that calls the API

* Database Services (modular): `src/app/services/`

- `mongodb.py` for documents (movies, ratings)

- `redis\_store.py` for top list ranking

- `neo4j\_graph.py` for user–movie graph and recommendations

* Seeder Script: `src/init\_db.py`

- Inserts sample data into MongoDB, Redis, and Neo4j

Data flow (rate action):

1. User submits a rating → API stores it in MongoDB

2. API bumps the movie’s score in Redis sorted set

3. API adds/ensures a `(:User)-[:LIKES]->(:Movie)` relationship in Neo4j

4. Recommendations query the graph to find suitable movies

## 3) Role of Each Database (Why three?)

* MongoDB (Document Store)

- Purpose: Primary system of record for movies and user ratings

- Strengths: Flexible schema, easy to query movie metadata and store ratings as simple documents

- Collections:

- `movies(\_id, title, genre, year, director)`

- `ratings(user\_id, movie\_id, rating, rated\_time)`

* Redis (In-Memory Store)

- Purpose: Real-time “Top Movies” leaderboard using a sorted set

- Strengths: Fast increments, sorted reads by score, ideal for live counters and rankings

- Keys:

- `top\_movies` (ZSET: member=title, score=accumulated rating score)

* Neo4j (Graph Database)

- Purpose: Collaborative-filtering via user–movie graph

- Strengths: Graph traversal and relationship queries are natural and efficient

- Model:

- Nodes: `User(user\_id)`, `Movie(movie\_id, title)`

- Relationship: `(User)-[:LIKES]->(Movie)`

## 4) Python Integration Details

Modules:

* MongoDB: `src/app/services/mongodb.py`

- Exposes `get\_movie\_by\_id`, `search\_movies\_by\_title`, `insert\_rating`

- Holds `movies\_col`, used by API to fetch movie titles

* Redis: `src/app/services/redis\_store.py`

- Exposes `increment\_movie\_score`, `get\_top\_movies`, `health`

- Maintains `top\_movies` ZSET

* Neo4j: `src/app/services/neo4j\_graph.py`

- Exposes `create\_like\_edge`, `recommend\_for\_user`, `health`

- Uses Cypher to compute recommendations

API Endpoints: `src/app/main.py`

* `GET /movies/search?q=<name>&limit=<n>`: Name-based search (MongoDB)
* `GET /movie/{movie\_id}`: Fetch single movie (MongoDB)
* `POST /rate { user\_id, movie\_id, rating }`: Rate, bump Redis, add LIKES edge (MongoDB/Redis/Neo4j)
* `GET /recommend/{user\_id}`: Collaborative recommendations (Neo4j)
* `GET /top\_movies?limit=n`: Top N movies (Redis)
* `GET /health`: Connectivity snapshot (All)

Seeding Script: `src/init\_db.py`

* Seeds MongoDB with 15 movies and ratings
* Seeds Redis with initial scores
* Seeds Neo4j with users, movies, and LIKES relationships

Run commands:

* Seed: `python src/init\_db.py`
* API: `python -m src.app.main`
* UI: `streamlit run src/streamlit\_app.py`

Dependencies (requirements.txt): FastAPI, Uvicorn, Pydantic, PyMongo, Redis, Neo4j, Streamlit, Requests

## 5) Conclusion

This design purposefully splits responsibilities:

* MongoDB for durable content and ratings
* Redis for fast ranking reads/updates
* Neo4j for graph-based recommendations

The system is intentionally small yet realistic, showing how polyglot persistence can deliver the right tool for the job while remaining easy to run locally and extend. The modular Python services enable clean substitutions or scaling of any single component.