

# IMDb WebApp Performance Analysis

Software Performance and Scalability Project

Filippo Zane (880119)

Maksim Kovalkov (888980)

May 2025

## Contents

<b>1</b>	<b>Project architecture</b>	<b>4</b>
<b>2</b>	<b>Scalability test in closed loop</b>	<b>5</b>
2.1	Setup of the load test . . . . .	5
2.2	First configuration . . . . .	6
2.2.1	Experimental performances evaluation . . . . .	6
2.2.2	JMT results . . . . .	8
2.3	Enhanced configuration . . . . .	9
2.3.1	Experimental performances evaluation . . . . .	10
2.3.2	JMT results . . . . .	11
<b>3</b>	<b>Future improvements</b>	<b>13</b>
<b>4</b>	<b>Conclusion</b>	<b>14</b>

## Introduction

The objective of this project is to design, implement, and evaluate a web application that serves movie-related data derived from the publicly available IMDb dataset. This work is part of the course [CM0481] *Software Performance and Scalability* and focuses specifically on applying closed-loop load testing methodologies and quantitative performance analysis techniques.

The system is built using a microservice architecture composed of three key components:

1. A backend REST API developed using FastAPI
2. A PostgreSQL relational database
3. A Redis-based caching layer

All components are containerized using Docker and orchestrated with Docker Compose.

Following an initial performance evaluation, a caching mechanism was introduced to improve response time and reduce database load. The Redis cache is pre-populated with the 5,000 most popular movies, allowing for rapid retrieval of frequently accessed entries.

The IMDb dataset is automatically downloaded and ingested at runtime, triggered by a Bash script executed during container startup.

Once running, the system exposes HTTP endpoints that allow users to query for movie titles. These endpoints were tested under various load conditions to evaluate the system's behavior.

The complete source code is available in the GitHub repository:  
<https://github.com/zaneef/imdb-scalability-analysis>

# 1 Project architecture

The application is structured with three main microservices:

- **FastAPI web server**  
A RESTful API implemented in Python using the FastAPI framework. It is responsible for handling incoming HTTP requests, parsing query parameters, querying the database or cache, and returning structured JSON responses.
- **PostgreSQL database**  
Stores the full IMDb movie dataset. The main table, `movies`, includes metadata such as title, year, rating, runtime, directors, and actors. Data is loaded once at container startup and remains read-only during runtime.
- **Redis caching system**  
Stores precomputed responses for the top 5,000 most-voted films, allowing fast retrieval for popular search queries. The cache is populated during startup.


All components are defined in a single `docker-compose.yml` file. The system can be launched with a single command:

```
docker compose up --build
```

This configuration sets up three services:

- **db**: initializes the PostgreSQL container (`imdb-postgres`) and expose port 5432
- **redis**: initializes the Redis container and exposes port 6379
- **webapp**: builds the FastAPI container, installs dependencies, loads the environment file, and executes the `startup.sh` script

The application can be accessed at `http://localhost:8000`, where users can search for movies by typing words or titles into the search bar.

 **IMDb Movie Search**

**The Matrix Reloaded (2003)**

★ 7.2 (651,927 votes)

**Directors:** Lana Wachowski, Lilly Wachowski

**Actors:** Laurence Fishburne, Keanu Reeves, Ray Anthony, Steve Bastoni, Christine Anu, Helmut Bakaitis, Andy Arness, Hugo Weaving, Alima Ashton-Sheibu, Carrie-Anne Moss

**The Matrix Revolutions (2003)**

★ 6.7 (560,753 votes)

**Directors:** Lana Wachowski, Lilly Wachowski

**Actors:** Tanveer K. Atwal, Kate Beahan, Laurence Fishburne, Francine Bell, Keanu Reeves, Monica Bellucci, Helmut Bakaitis, Hugo Weaving, Mary Alice, Carrie-Anne Moss

**Matrix (2020)**

★ 5.8 (82 votes)

**Directors:** Kim Harrington

**Actors:** Chris Harvey

**The Living Matrix (2009)**

★ 6.6 (194 votes)

**Directors:** Greg Becker

## 2 Scalability test in closed loop

This section describes the scalability testing methodology and the performance results obtained.

### 2.1 Setup of the load test

To evaluate how the application behaves under increasing user load, we conducted a series of closed-loop load tests using *Apache JMeter*.

In a closed-loop load test, the number of simulated users is fixed. Each user waits for a response before sending the next request.

The testbed was configured as follows:

- **CSV Data Set Config**  
Specifies a CSV file from which data will be fetched and included in the request.
- **Thread Group**  
Specifies a pool of a certain number of users concurrently sending requests. Test were repeated multiple times with different amounts of users to observe the behavior of the system under different loads. We run tests of 300s (5 minutes), and we allocate 33s for the ramp-up period.
- **HTTP request sampler**  
Sends **GET** requests to the `/movies/` endpoint, with `?title=` as query parameter. The list of queries is taken from a CSV file generated offline using a probability distribution proportional to the number of votes of each movie. This allows us to simulate realistic search behavior.
- **Constant Timer**  
Introduced to simulate a think time of **1.5 seconds**, between user actions to mimic realistic usage behavior.
- **Result listener**  
Included to collect and visualize performance metrics.

The `project.jmx` project file used for the tests is available in the repository under the path `jmeter/project.jmx`.

Based on the number of customers, ramp up was configured on JMeter to allow the system not to be overloaded at the same time and thus to perform a test of increasing load as evenly as possible. The ramp up was configured as described in Table 1.

Number of customers	Ramp up (s)
1	1
10	10
20	20
50	25
100	33
200	60
500	100
1000	150

Table 1: Ramp up based on the number of customers in the load performance

Furthermore, is important to denote that all load tests were conducted for 300 seconds.

## 2.2 First configuration

In the first configuration, the FastAPI application directly queries the PostgreSQL database. Every client request results in a new SQL query. No caching is employed, meaning the full load is absorbed by the web application and database.

The relevant function that handles the search request is defined in `app/crud.py` file, and it's developed as follows:

```
def get_movie_by_title(db: Session, title: str):
    cache_key = title.lower()

    movies = db.query(models.Movie).filter(
        models.Movie.title.ilike(f"%{ title }%")
    ).all()

    if movies:
        schema_movies = []
        for m in movies:
            if isinstance(m.directors, str):
                m.directors = [d.strip() for d in m.directors.split(",")]
            if isinstance(m.actors, str):
                m.actors = [a.strip() for a in m.actors.split(",")]

            schema_movies.append(MovieSchema.from_orm(m))

        return [m.model_dump() for m in schema_movies]
    return None
```

An important note should be made about the line of code responsible for retrieving the results obtained from the db:

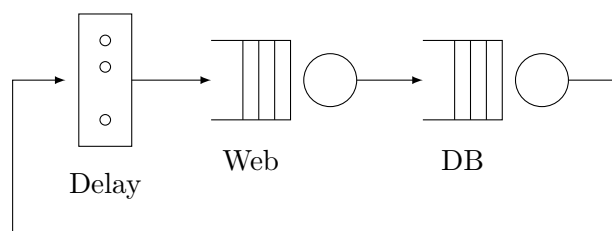
```
movies = db.query(models.Movie).filter(
    models.Movie.title.ilike(f"%{ title }%")
).all()
```

Since the ILIKE "%keyword%" pattern is used, PostgreSQL performs a full sequential scan in the entire table, which is computationally expensive.

### 2.2.1 Experimental performances evaluation

We modeled the system as a closed queueing network with three stations:

- The **Delay** represents the delay station
- The **Web** queue represents the web application
- **DB** queue represents the PostgreSQL database



#	Tot	Err (%)	Avg (ms)	Min (ms)	Max (ms)	Med (ms)	90 (ms)	95 (ms)	99 (ms)	Through (req/s)
1	202	0.000	178	106	6217	139	173	182	573	0.68
10	1942	0.000	217	104	7477	188	287	356	596	6.51
20	3512	0.000	349	111	7540	268	648	822	1220	11.75
50	4116	0.000	2200	138	9435	2144	2869	3023	3399	13.68
100	787	45.997	40130	0	90192	57483	89453	89839	90160	2.22
200	-	-	-	-	-	-	-	-	-	-
500	-	-	-	-	-	-	-	-	-	-
1000	-	-	-	-	-	-	-	-	-	-

Table 2: First configuration performances

The results of the load test can be visualized in Table 2. At 100 users, the system becomes saturated, resulting in drastic drop in throughput, long response times and high error rate. Tests with 200+ users were considered unnecessary due to complete saturation.

**Web application response time.** To isolate the response time of the web application alone, we created a minimal endpoint:

```
@app.get("/ping")
def pong(request: Request):
    return {"response": "pong"}
```

Using JMeter with a single user, we measured the average response time as **4 ms = 0.004 ms**.

**PostgreSQL response time.** To isolate the PostgreSQL component, we executed the following command:

```
docker exec -it imdb-postgres psql -d imdb -U postgres -c "EXPLAIN (ANALYZE,
    ↪ BUFFERS) SELECT * FROM movies WHERE title ILIKE '%matrix%';"

[... snip ...]
Execution Time: 112.477 ms
(12 rows)
```

This query was executed 20 times. The measured *Execution Time* values were averaged, resulting in: **101.63 ms = 0.1016 s**.

**Service rates.** From the response times above, we calculated the service rate of each component:

- $\mu_{\text{Web}} = \frac{1}{0.004 \text{ s}} = 250 \text{ req/s}$
- $\mu_{\text{DB}} = \frac{1}{0.1016 \text{ s}} = 9.84 \text{ req/s}$

**Relative visit ratios.** To compute the relative visit ratios  $\bar{V}_i$ , we first solve the traffic equation, by calculating the value of  $e_i$  at each station using the formula

$$e_i = \sum_{j=1}^k e_j p_{ji} \quad (1)$$

Therefore, assuming that Delay = 1, Web = 2 and DB = 3 and by calculating all components by solving (1), we get

$$\begin{cases} e_1 = e_3 \\ e_2 = e_1 \\ e_3 = e_2 \end{cases}$$

Since this is a closed system, an infinite number of solutions can be found. We need to fix  $e_1 = 1$ . Solving the system, we obtain:

$$e_1 = e_2 = e_3$$

In a closed queueing network, the relative visit ratio to a station is equal to its ,  $e_i = \bar{V}_i$  picked a reference station. Having chosen  $Q_1$  (the delay station), we obtain

- $\bar{V}_1 = 1$
- $\bar{V}_2 = e_2 = 1$
- $\bar{V}_3 = e_3 = 1$

This simply means that, for each visit a job makes to the reference station, on average it performs a single visit to the webapp and a single visit to the DB, which is intuitively correct.

**Service demands.** With these information, we can compute the service demand of each component by solving the equation

$$\bar{D}_i = \bar{V}_i \cdot \frac{1}{\mu_i} \quad (2)$$

Using the previously calculated service rates, we obtain:

- $\bar{D}_{\text{Web}} = \bar{V}_{\text{Web}} \cdot \frac{1}{\mu_{\text{Web}}} = 1 \cdot \frac{1}{250 \text{ req/s}} = 0.004 \text{ s}$
- $\bar{D}_{\text{DB}} = \bar{V}_{\text{DB}} \cdot \frac{1}{\mu_{\text{DB}}} = 1 \cdot \frac{1}{5.74 \text{ req/s}} = 0.1016 \text{ s}$

We can now conclude by saying that a job, for each visit it does at the delay station, it demands roughly **0.004 seconds** from the webapp and **0.1016 seconds** from the DB.

From these results, it can be asserted that the bottleneck of the system is definitely the DB, since it has higher service demand, and every request spends most of its time inside of it.

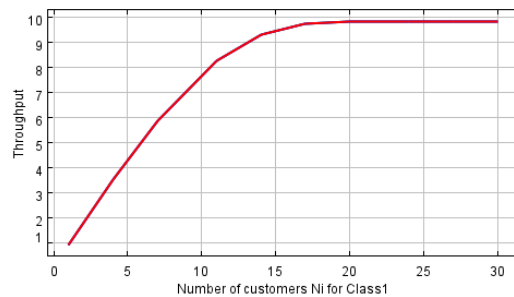
### 2.2.2 JMT results

All previous results was used to plot the utilization and throughput of the queueing system in *Java Modelling Tools (JMT)*. The model used to perform the calculations can be found under the path `jmt/project_raw.jmva`. Figures 1a and 1b present the simulated throughput and utilization obtained through JMT for the first configuration.

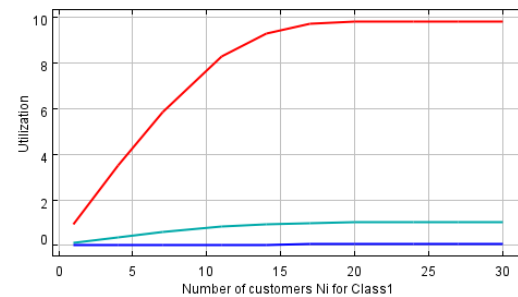
**Throughput.** The simulated throughput initially grows linearly with the number of users, which is expected in closed systems with a delay station. However, the throughput saturates around 17 users, which indicates the maximum processing capacity of the system has been reached.

**Utilization.** The utilization graph confirms this diagnosis: the database - represented by the light blue line - quickly reaches full utilization (close to 1.0), while the web application remains underutilized (the blue line). This further reinforces that the database is the bottleneck.





(a) Throughput of first configuration



(b) Utilization of first configuration

## 2.3 Enhanced configuration

In this model, a caching system was introduced in Redis, also containerized, against the pressure observed on the database. Query responses are stored in Redis with a key associated with the title being searched and a TTL of six hours. In addition, a cache *warm-up* is performed at system startup by loading into memory the responses related to the 5,000 movies with the highest number of ratings. In case of a cache hit, the query is served entirely by Redis without involving the database. Furthermore, the query returns only the first 100 results that contain the searched word within the title, further reducing the load on the database when reading the record.

The function accountable for each previously described operation can be found in `app/crud.py` and it's defined as:

```
def cache_get_movie_by_title(db: Session, title: str):
    cache_key = f"movie:{title.lower()}"
    cached = redis_client.get(cache_key)

    if cached:
        return json.loads(cached)

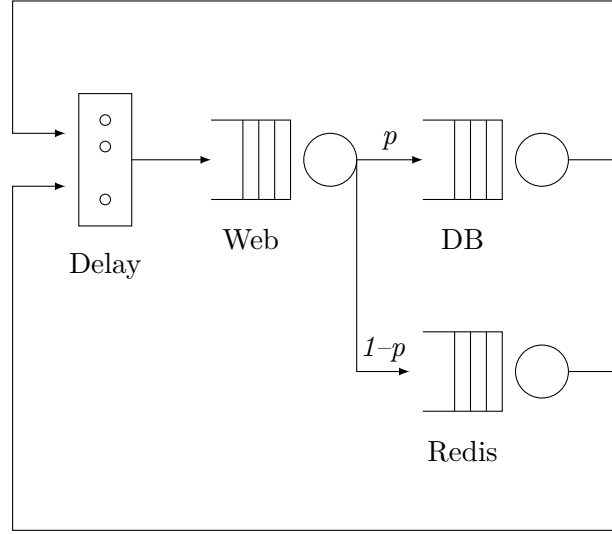
    movies = db.query(models.Movie).filter(
        models.Movie.title.ilike(f"%{title}%")
    ).limit(100).all()

    if movies:
        schema_movies = []
        for m in movies:
            if isinstance(m.directors, str):
                m.directors = [d.strip() for d in m.directors.split(",")]
            if isinstance(m.actors, str):
                m.actors = [a.strip() for a in m.actors.split(",")]
            schema_movies.append(MovieSchema.from_orm(m))

        redis_client.set(
            cache_key,
            json.dumps([m.model_dump() for m in schema_movies]),
            ex=21600
        )
        return [m.model_dump() for m in schema_movies]
    return None
```

### 2.3.1 Experimental performances evaluation

The queueing network now includes a Redis station, modifying the model as:



We let  $p$  describe the probability of a cache miss (i.e., the database is accessed), and  $1 - p$  the probability of a cache hit. Given the pre-warming strategy adopted, we can estimate:

$$p = 0.5$$

We tested the system using JMeter, with the same ramp up described in Table 1, and obtained the results described in Table 3.

#	Tot	Err (%)	Avg (ms)	Min (ms)	Max (ms)	Med (ms)	90 (ms)	95 (ms)	99 (ms)	Through (req/s)
1	224	0.000	36	2	201	9	159	175	186	0.75
10	2197	0.091	40	2	319	6	158	176	209	7.35
20	4323	0.046	41	1	508	5	161	185	234	14.47
50	10574	0.039	56	1	1681	4	205	284	639	35.41
100	17501	0.046	316	1	6823	4	1562	2677	3858	58.60
200	-	-	-	-	-	-	-	-	-	-
500	-	-	-	-	-	-	-	-	-	-
1000	-	-	-	-	-	-	-	-	-	-

Table 3: Enhanced configuration performances

We can confirm that all performance has benefited from the change, making the system perform better and with 4.5 times higher throughput than the first system. In addition, the load test with 100 users was passed successfully and with excellent results, finding a computational limit at 200 users.

**Redis response time.** Through a simple Python script, we were able to estimate the response time of Redis by simulating multiple requests with cache hits and calculating the average cache response time. The sum of all times, divided by the number of requests made, gave us the average response time of the cache: **0.5 ms = 0.0005 s**.

**Service rates.** Combining this result along with the ones already computer, we retrieved the service rate of each component in the system:

- $\mu_{\text{Web}} = \frac{1}{0.004 \text{ s}} = 250 \text{ req/s}$
- $\mu_{\text{DB}} = \frac{1}{0.1016 \text{ s}} = 9.84 \text{ req/s}$
- $\mu_{\text{Redis}} = \frac{1}{0.0005} = 2000 \text{ req/s}$

**Relative visit rations.** As described in the previous section, we need to solve the global balance equation to obtain the relative visit ratio of each station. Let  $Q_1 = \text{Delay}$ ,  $Q_2 = \text{Web}$ ,  $Q_3 = \text{DB}$  and  $Q_4 = \text{Redis}$ , then

$$\begin{cases} e_1 = e_3 + e_4 \\ e_2 = e_1 \\ e_3 = p \cdot e_2 \\ e_4 = (1 - p) \cdot e_2 \end{cases}$$

Once again, since we are dealing with a closed system, infinite amount of solutions could be found. We have to pick  $Q_1$  as the reference station, fix  $e_1 = 1$  and solve the entire system. Also, with  $p = 0.5$ , we obtain:

$$\begin{cases} e_1 = 1 \\ e_2 = 1 \\ e_3 = 0.5 \\ e_4 = 0.5 \end{cases}$$

As before, since we are dealing with a closed system, the relative visit ratio  $\bar{V}_i$  for each station  $i$ , is equal to  $e_i$ , so:

- $\bar{V}_1 = 1$
- $\bar{V}_2 = 1$
- $\bar{V}_3 = 0.5$
- $\bar{V}_4 = 0.5$

To calculate the service demand, we have to solve (2) for this queueing system. So,

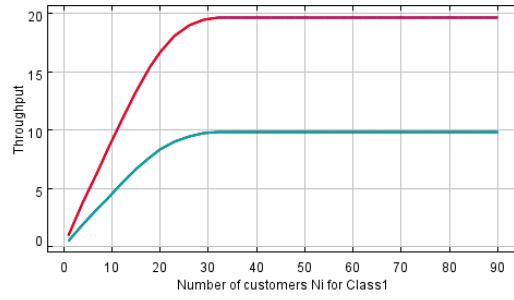
- $\bar{D}_{\text{Web}} = \bar{V}_{\text{Web}} \cdot \frac{1}{\mu_{\text{Web}}} = 1 \cdot \frac{1}{250 \text{ req/s}} = 0.004 \text{ s}$
- $\bar{D}_{\text{DB}} = \bar{V}_{\text{DB}} \cdot \frac{1}{\mu_{\text{DB}}} = 0.5 \cdot \frac{1}{9.84 \text{ req/s}} = 0.0508 \text{ s}$
- $\bar{D}_{\text{Redis}} = \bar{V}_{\text{Redis}} \cdot \frac{1}{\mu_{\text{Redis}}} = 0.5 \cdot \frac{1}{2000 \text{ req/s}} = 0.00025 \text{ s}$

The DB service demand dropped by 50%, due to the lower visit ratio but it's still the one with the highest service demand, resulting in it still being the bottleneck of the system.

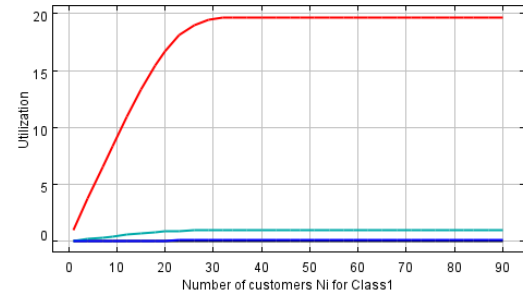
### 2.3.2 JMT results

All previous results were used in JMT to plot the system behaviour. Figure 2a shows the throughput of the system as the number of customers increase, while Figure 2b shows the utilization under the same circumstances.

**Throughput.** The throughput curve demonstrates a much more gradual saturation, compared to the first one. Under high loads, the system now sustains over 55 requests per second, confirming the significant benefits of Redis caching.



(a) Throughput graph of configuration 2



(b) Utilization graph of configuration 2

**Utilization.** The Redis station (black line) remains underutilized due to its extremely fast response time and high capacity. The DB station (light blue line) still approaches saturation, but much more slowly than in the uncached scenario, confirming that Redis reduces DB pressure effectively.

### 3 Future improvements

While the current system achieves substantial performance improvements through caching, several enhancements could further increase scalability, reduce latency, and improve overall performances of the system.

One of the main issues lies in the SQL query pattern used:

```
WHERE title ILIKE '%...%'
```

The use of `'%...%'` wildcard makes PostgreSQL to perform a full table scan. This significantly increases I/O latency, especially as the dataset grows.

A possible solution is to introduce pagination to limit the number of returned records (partially implemented but still being too high), or to restrict the search pattern to only match full movie titles instead of partial substrings.

## 4 Conclusion

This project analyzed the performance and scalability of a containerized web application built on FastAPI and PostgreSQL. Using closed-loop load testing and queueing network models, we identified the PostgreSQL database as the **primary system bottleneck**, with the WebApp layer showing negligible service time in comparison.

To alleviate this bottleneck, we introduced a Redis caching system that stores responses for popular movies. This reduced database pressure significantly, increased throughput, and decreased average response times.

Both experimental results (via JMeter) and analytical results (via JMT) confirmed that:

- The uncached system saturates quickly
- The cached system sustains up to 58 requests per second
- The database remains the dominant factor in the system limits