**Report: Performance and Scalability of IMDb Movie Search Application**

**1. Introduction**

This report presents the design, implementation, and performance analysis of a web application developed as part of the course CM0481 - Software Performance and Scalability. The project replicates a miniature version of the IMDb platform by enabling movie search and displaying associated metadata. The system is evaluated for scalability, performance bottlenecks, and improved using modeling tools.

The objective is to:

* Create a functional web interface to query movie data
* Generate and simulate real-world load patterns
* Measure scalability using load testing
* Analyze bottlenecks and propose architectural improvements
* Validate optimization using queueing network simulation (JMT)

**2. Step 1: Web Application Design and Implementation**

**2.1 Overview**

The system is built using FastAPI for the backend and HTML/CSS (Jinja2 templates) for the frontend. It provides search capabilities for movie titles and displays detailed metadata.

**2.2 Data Pipeline**

IMDb's dataset is used, downloaded from <https://www.imdb.com/interfaces/>.  
The following .tsv files were used:

* title.basics.tsv
* title.ratings.tsv
* title.principals.tsv
* name.basics.tsv

The relevant data is preprocessed using Python scripts and stored into a local SQLite database (movies.db). Key relationships (e.g., titles to actors, directors, writers) were built and stored for efficient querying.

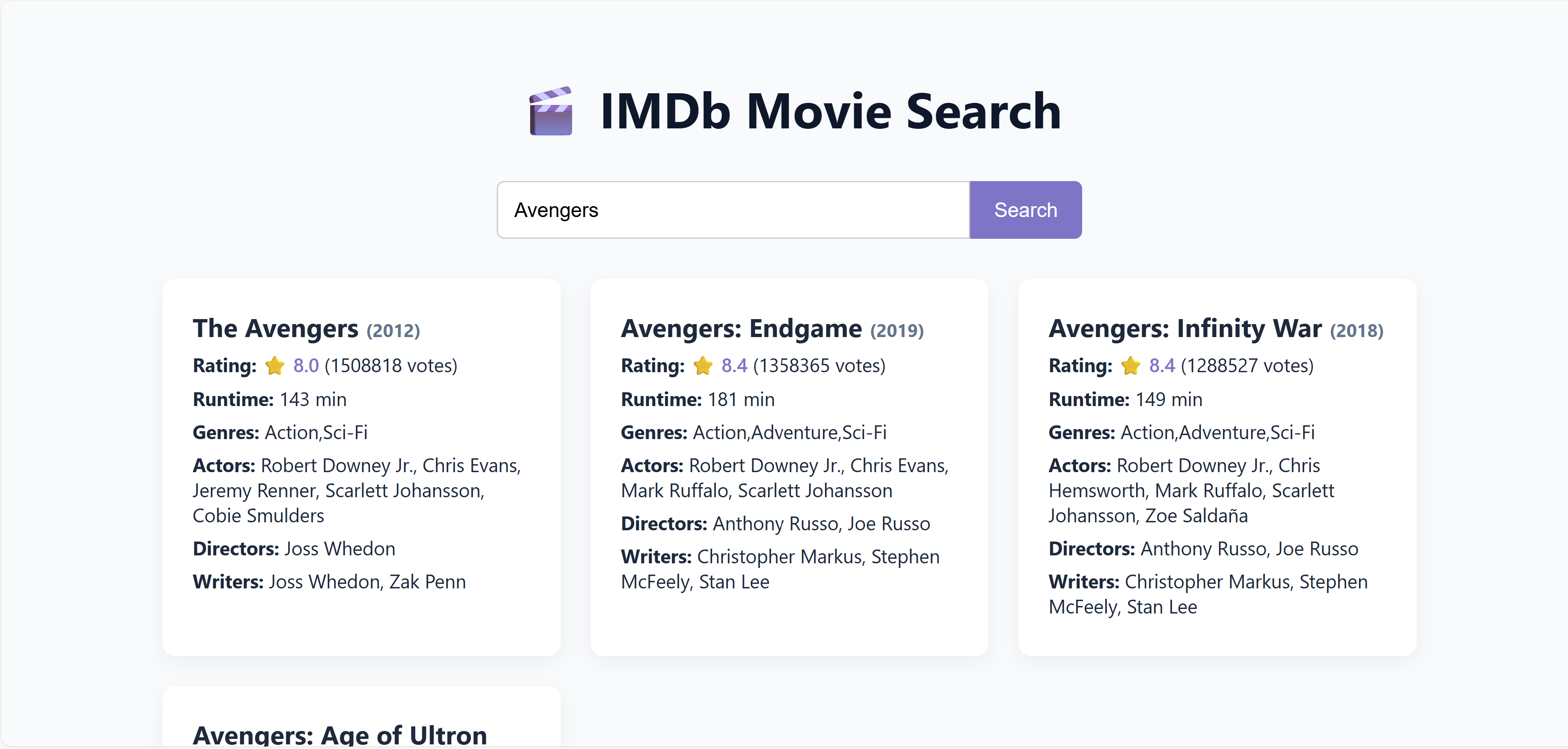
**2.3 Features**

* Search by movie title (partial matching)
* Display:
  + Title and release year
  + Average rating and number of votes
  + Runtime
  + Genres
  + Main actors
  + Directors
  + Writers

**2.4 Technologies**

* Backend: FastAPI + SQLite (synchronous access)
* Frontend: HTML/CSS with Jinja templating
* Data: IMDb dataset parsed via pandas

**2.5 Screenshot**

**

**3. Step 2: Query Set Generation**

To simulate real user behavior, a query set of 10,000 movie titles was generated.

**3.1 Sampling Rule**

We assume the likelihood of a movie being searched is proportional to the number of ratings it has received. Therefore, titles with more ratings are more likely to appear in the query set.

**3.2 Methodology**

* Load movie list with their corresponding numVotes
* Normalize numVotes to create a probability distribution
* Use numpy.random.choice with p=probabilities to sample 10,000 titles with replacement
* Output saved in query\_set.txt

**3.3 Validation**

* Duplicates observed, as expected under weighted sampling
* Query set distribution matches the popularity skew in the dataset

**4. Step 3: Load Testing and Performance Analysis**

The application was tested using Locust, a modern load testing framework. Five load scenarios were executed to assess how the system handles increasing traffic.

**4.1 Test Plan**

| **Test** | **Users** | **Spawn Rate** | **Duration** | **Description** |
| --- | --- | --- | --- | --- |
| 1 | 10 | 2 | 2 mins | Baseline |
| 2 | 25 | 5 | 3 mins | Moderate load |
| 3 | 50 | 10 | 4 mins | Increased concurrency |
| 4 | 75 | 15 | 5 mins | High stress level |
| 5 | 100 | 20 | 6 mins | Near max capacity |

**4.2 Observed Metrics Summary**

| **Test** | **Users** | **Requests/sec** | **Avg Resp Time (ms)** | **Median (ms)** | **95%ile (ms)** | **99%ile (ms)** | **Max Resp Time (ms)** | **Failures** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 10 | 6.6 | 14.95 | 12 | 28 | 62 | 62 | 0 |
| 2 | 25 | 16.2 | 16.72 | 13 | 37 | 76 | 76 | 0 |
| 3 | 50 | 31.8 | 18.86 | 14 | 43 | 130 | 130 | 0 |
| 4 | 75 | 49.0 | 23.14 | 16 | 58 | 150 | 150 | 0 |
| 5 | 100 | 65.1 | 25.73 | 17 | 74 | 150 | 150 | 0 |

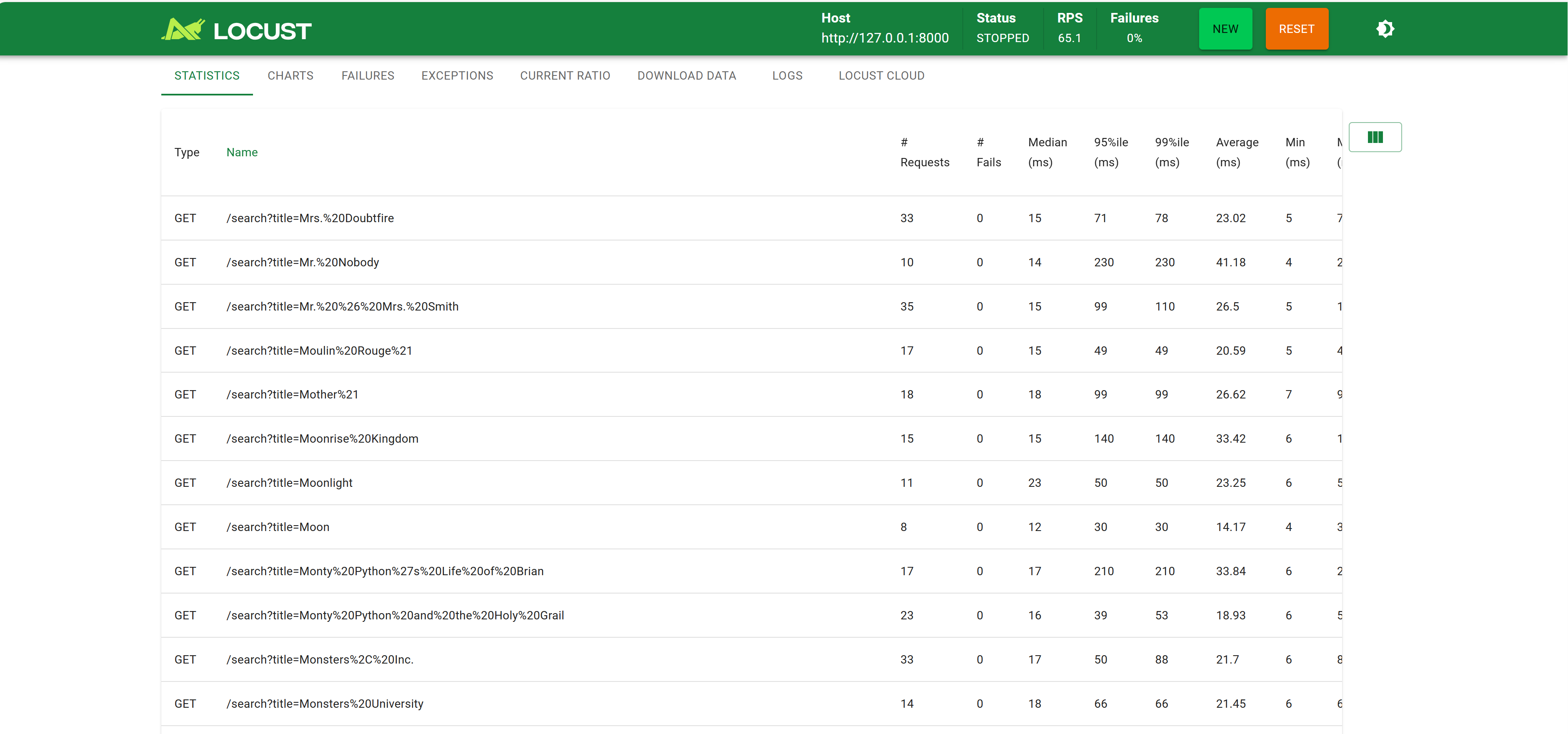
**4.3 Key Observations**

* Scalability: Linear increase in throughput (requests/sec) with added users
* Stability: 0% failure rate across all tests
* Emerging Saturation: Response time started to rise more sharply from Test 4 onward
* Tail Latency: 99%ile latency increased significantly in Test 3 and plateaued at 150 ms in Tests 4 and 5, indicating pressure on the backend under high load
* Efficiency: SQLite's read-only design helped under moderate loads, but signs of bottlenecks appeared

**4.4 Bottleneck Analysis**

| **Area** | **Issue Description** |
| --- | --- |
| SQLite | Non-concurrent; serialized access can limit scaling |
| Sync DB Access | FastAPI using blocking sqlite3 module stalls event loop |
| Single Worker | Uvicorn runs on one worker by default |
| No Caching | Repeated queries re-hit DB each time |
| No Horizontal Scale | No load balancing or multi-instance deployment |

**Note:** Step 4 (Optimized Design + JMT simulation results) will be presented in the next section after final simulation testing. The current results serve as the performance baseline for architectural comparison.

**

