# EEX5362 Performance Modelling

# Mini Project

# Auto-Scaling Cloud System for Load Balancing

| | | |
|---|---|---|
| Name | : | P.D.K. Sulakshana |
| Student ID | : | S22010213 |
| Registration No | : | 322513797 |

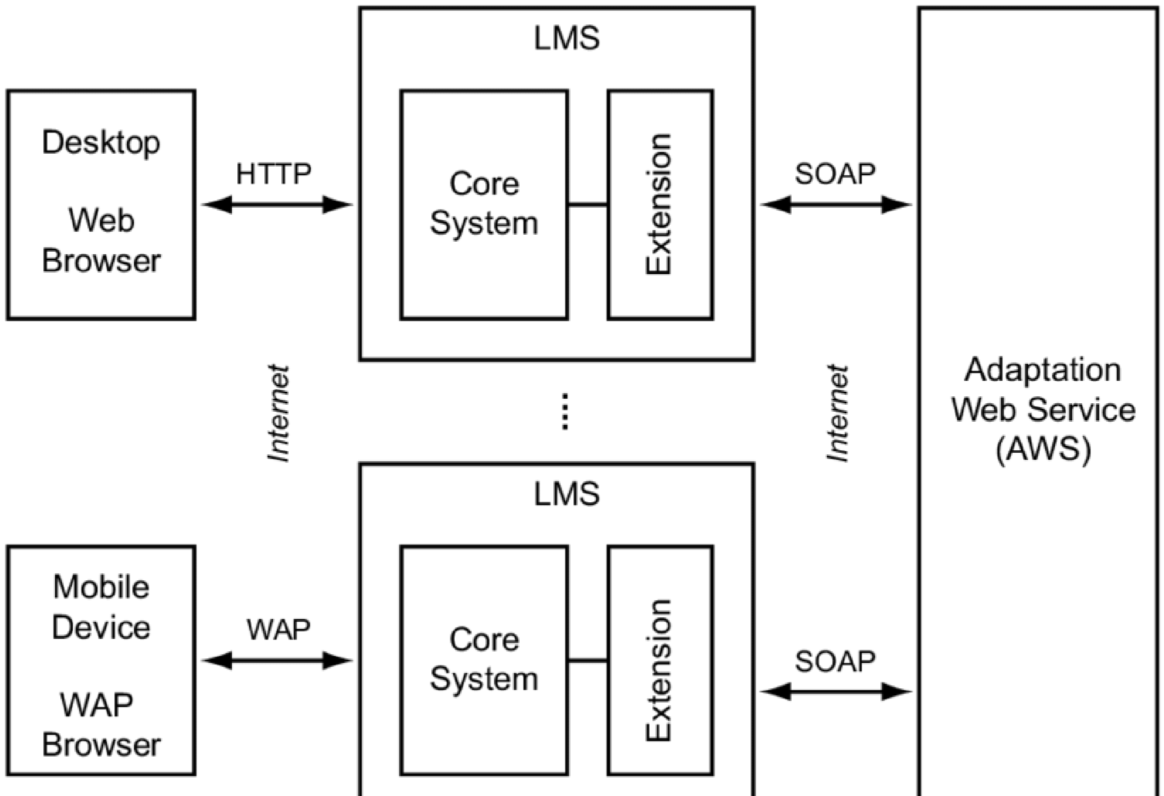
## Table of Contents

## 01. Introduction

I have chosen an auto-scaling cloud system that manages user traffic for a web application, specifically modeling a scenario like an Online Learning Platform (LMS). In this environment, performance is the top priority to ensure stability. The server is designed to operate continuously and handle many concurrent users making real-time requests.

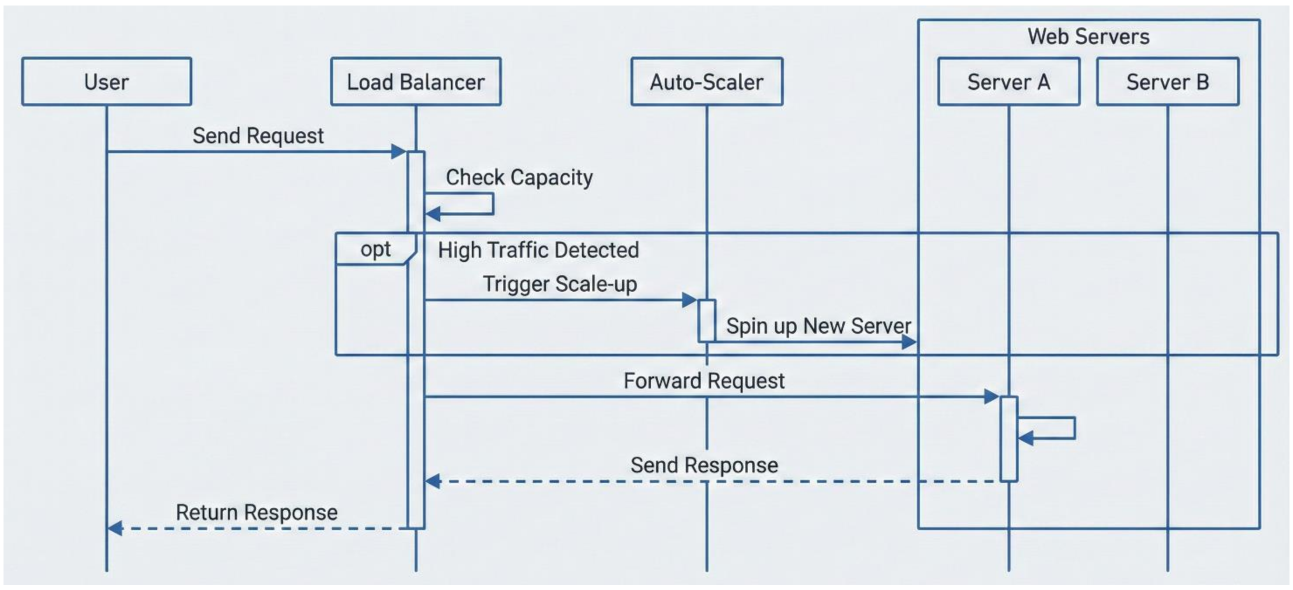
I plan to evaluate how well this system performs under different loads like when thousands of students try to submit an assignment at once. I will specifically focus on optimizing response times and resource usage to identify potential bottlenecks and ensure a smooth user experience.

The system receives various types of requests. Some are lightweight, such as loading a dashboard or viewing a page, while others are resource heavy, like uploading assignments or searching through course materials. To handle this, the system uses cloud resources that can automatically expand or shrink based on demand. A typical request can range from a few kilobytes for a simple view to several megabytes for file transfers.



## 1.1 Sequence of the System

The workflow begins when a user interacts with the application. A load balancer receives the request first and directs it to a server that has available capacity. If the incoming traffic gets too high, the auto-scaler activates to spin up new servers instantly. Once the request is processed, the response is sent back to the user. This entire cycle is continuously monitored to ensure the system scales up or down at the right time.



## 1.2 Why Performance is Crucial in This System

- Speed is everything. Users expect the system to react instantly. If it performs poorly, requests get stuck in line, causing long waits. This makes the system feel unreliable and frustrates the people trying to use it.
- Handling one user is easy, but handling thousands at once like students rushing to submit an assignment is hard. Without effective auto-scaling, this sudden pressure overloads the servers, causing the site to crash right when people need it most.
- If we run too many servers when things are quiet, we waste cost. If we run too few during busy times, the site slows down or shows errors. Getting this balance right is vital to keep the system smooth for users and affordable to run.

## 02. Performance Goals

My performance goals for this system are based on queuing theory. Basically, I want to ensure the system stays stable and fast, even when a rush of users logs in at the same time.

- Average Response Time (W): My primary goal is to minimize the average time a user must wait for a page to load. If the system is too slow, users get frustrated, so speed is key.
- Arrival Rates (λ): I want to maximize the system's throughput, which means ensuring it can process a high volume of incoming requests without dropping any connections or crashing.
- Utilization (ρ): I aim to keep server utilization between 80-90%. Running at 100% is risky because it leads to overloads, leaving a safety buffer helps prevent the system from freezing up.
- Queue Length (L): The system needs to detect when the queue of waiting requests is getting too long and automatically add more servers to clear the backlog. This prevents the infinite wait scenarios that happen when a system gets overwhelmed.

## 03. Modeling Approach and Assumptions

### 3.1 Modeling Approach

For this analysis, I chose to mix queuing theory with discrete-event simulation (DES). This setup works well for dealing with complicated systems that do not stay steady over time. On the analytical side, I treat the system like an M/G/c queue.

- M (Arrivals): This represents the user requests coming in. I assumed they follow a random (Poisson) pattern based on the timestamps in my data.
- G (Service Times): This stands for General distribution. I used the cpu_demand data here because real tasks vary wildly in how long they take to process.
- c (Servers): This is the number of active servers. I configured the system to start with a baseline (1 to 5 servers) and scale up to 10 as needed.

Formulas like Little's Law ($L = \lambda W$) are great for calculating averages, but they miss the specific moments when a system gets overwhelmed. By using a simulation (SimPy), I can test what-if scenarios to see how the system reacts to sudden pressure in real-time.

### 3.2 Assumptions

- I calculated the time gaps between requests from the dataset and treated them as a Poisson distribution.
- I used the absolute value of cpu_demand.
- If the average queue length exceeds 5 requests, add a server. If server utilization drops below 30%, remove a server to save resources.
- I assumed the queue is infinite. This means requests are never dropped or rejected, they just must wait in line if the servers are busy.
- The system respects the priority_level column high priority tasks cut to the front of the line.
- While the dataset covers a long period, I simulated a specific sample (the first 1000 rows) to keep the processing efficient.

I built this entire analysis using Python. I used pandas to clean the data, SimPy to run the simulation logic, and matplotlib to generate the performance graphs.

## 04. Data Description and Methodology

### 4.1 Data Description

For this study, I used a dataset containing 10,678 records and 10 columns. The data covers a significant timestamp from February 19, 2022, to December 22, 2023.

Key Data Fields:

- task_size: Measures the overall size of each task, in numeric form.
- cpu_demand: Shows the CPU resources needed for the task.
- memory_demand: Indicates the memory required.
- network_latency: Captures any network delays linked to the task.
- priority_level: Assigns a priority to the task for handling order.
- timestamp: Records when each task was logged.

### 4.2 Methodology

To turn this raw data into a working simulation, I followed a five steps process using Python:

- The dataset was loaded, sorted by timestamp, and transformed into positive, time-based values to obtain realistic interarrival and service times.
- The arrival rate (λ) was calculated from the mean interarrival time, while the service rate (μ) was derived from the mean service time.
- A discrete-event simulation was built using SimPy to model request arrivals, queue behavior, and server processing based on service demand.
- Three scenarios were evaluated, a fixed system with three servers, an auto-scaling configuration, and a high-load case where the arrival rate was doubled.
- Key metrics including response time (W), queue length (L), utilization (ρ), and throughput were collected and visualized for analysis.

## 05. Detailed Analysis and Findings

I evaluate how the cloud system performs under three distinct scenarios, a Fixed-Capacity setup, an Auto-Scaling setup, and a High-Load stress test where I doubled the traffic. I focused on the metrics that matter most to users, how long they wait (response time), how long the lines get (queue behavior), and how hard the servers are working (utilization).

1. Fixed Server Scenario (3 Servers)

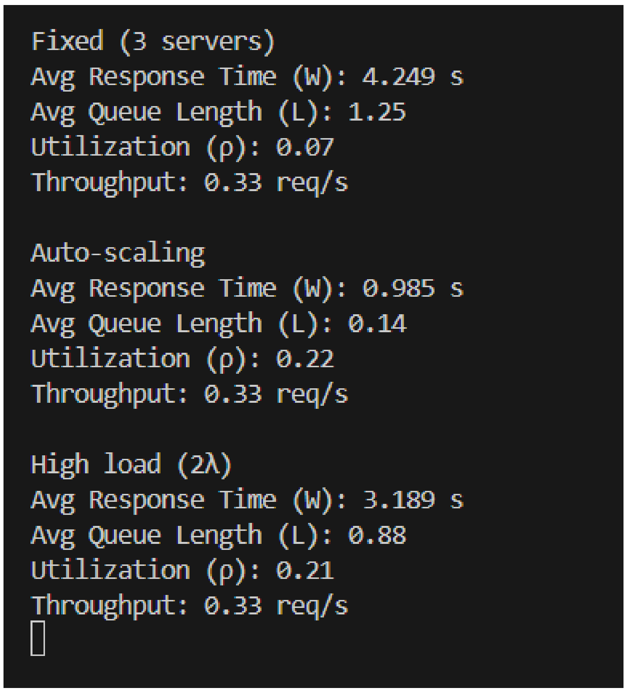
   The system used three servers that could not change. The average wait was 4.25 seconds. Some users waited much longer. Even though the servers were mostly idle, short traffic bursts caused lines to form immediately. The system could not handle spikes.

2. Auto-Scaling Scenario

   The system could add servers automatically. The average wait dropped to 0.99 seconds. Users rarely had to wait in line. Servers were used more efficiently without being overloaded. The system handled traffic smoothly.

3. High-Load Scenario

   I doubled the traffic. The average wait increased to 3.19 seconds, but this was still better than the fixed-server setup. A short line formed when the rush started, but the system added servers and caught up. It processed all requests without crashing.

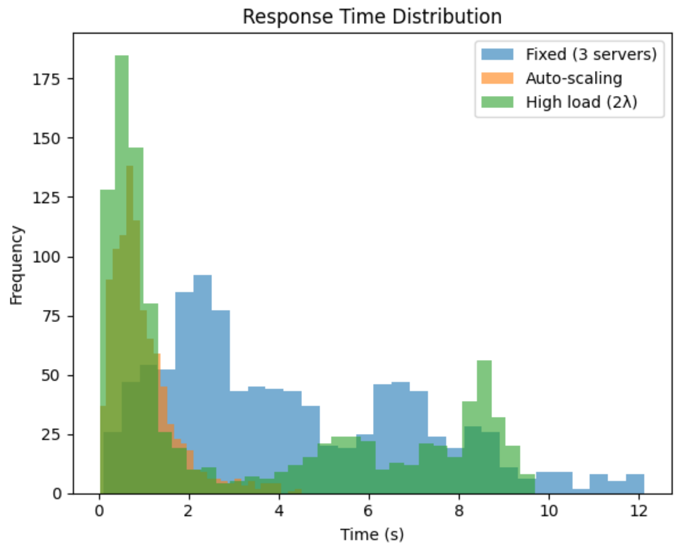


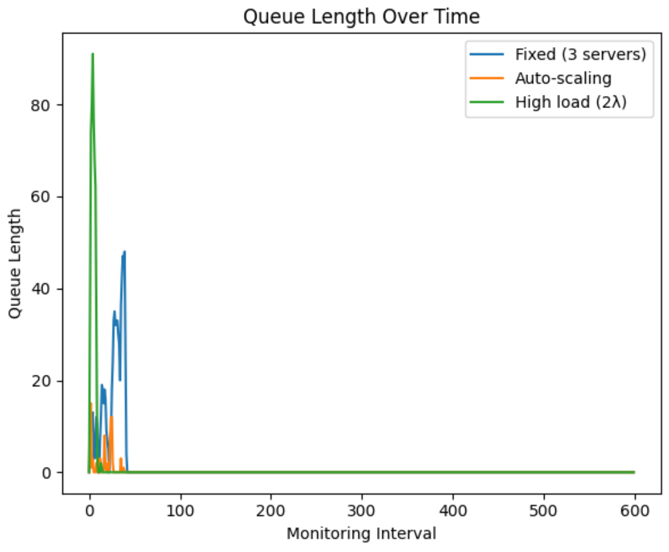
```
Fixed (3 servers)
Avg Response Time (W): 4.249 s
Avg Queue Length (L): 1.25
Utilization (ρ): 0.07
Throughput: 0.33 req/s

Auto-scaling
Avg Response Time (W): 0.985 s
Avg Queue Length (L): 0.14
Utilization (ρ): 0.22
Throughput: 0.33 req/s

High load (2λ)
Avg Response Time (W): 3.189 s
Avg Queue Length (L): 0.88
Utilization (ρ): 0.21
Throughput: 0.33 req/s
```

## 06. Visualizations





The charts provide a clear picture of the differences between the three scenarios,

**Response Time Distribution:**

- The Fixed-Server line has a long tail, meaning many users got stuck waiting for a long time.
- The Auto-Scaling line is tightly packed at the low end, showing that almost everyone got a fast response.
- The High-Load scenario spreads out a bit more, but it remains controlled compared to the fixed servers.

**Queue Length Over Time:**

- Fixed Servers show repeated spikes, indicating that backlogs kept forming repeatedly.
- Auto-Scaling stays flat near zero, meaning the line rarely built up.
- The High-Load case shows brief spikes this captures the specific moments where the queue grew just before the new servers came online to clear it.

## 07. Limitations and Future Extensions

### 7.1 Limitations

While this simulation offers valuable insights, it has key limitations.

- I used a Poisson to model user arrivals.
- I assumed the system never rejects a request. In real life, servers run out of memory or time out, leading to dropped connections.
- I treated adding a new server as instant. It takes time for a virtual machine to boot up.
- To keep the analysis manageable, I simulated a specific subset of the data.

### 7.2 Future Extensions

To make this simulation even more realistic, here is what I would improve next:

- Factor in the time servers take to boot up, rather than assuming they are ready instantly.
- Set a maximum queue size to see when requests get dropped or time out, which happens in real production systems.
- Use past data to predict spikes before they happen, rather than just reacting after the fact.
- Calculate the financial cost of running the extra servers to find the best balance between speed and budget.

## 08. References

[1] S. Garg, "Load Balancing Dataset," Kaggle, 2023. [Online]. Available:

https://www.kaggle.com/datasets/shantanugarg274/load-balancing-dataset

[2] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris, *Fundamentals of Queueing Theory*, 5th ed. Hoboken, NJ, USA: Wiley, 2018.

[3] SimPy Development Team, "SimPy Documentation," 2024. [Online]. Available:

https://simpy.readthedocs.io/en/latest/

## 09. Appendix

GitHub Repo: https://github.com/ksulakshana02/EEX5362-Mini-Project