LOG8415E – Advanced Concepts of Cloud Computing

Assignment #1

Load Balancer From Scratch

Alexis Desforges - 2146454
Samy Labassi - 1953898
Louis Lalonde - 2335476
Loic Nguemegne Temena - 2180845

Submitted to

Vahid Majdinasab

Fall 2025 – 23 september 2025

Contents

Chapter 1

Experiment architecture

In figure 1.1, we depict the architecture used for our AWS load balancing benchmarking experiment.

1.1 Architecture Overview

Incoming requests are received by a custom load balancer running on an EC2 instance [3], which acts as a reverse proxy. For each request, the load balancer selects the EC2 instance with the lowest latency within the target cluster and forwards the traffic to it. Requests are only routed to one cluster at a time, based on the chosen endpoint.

1.1.1 Target Groups and EC2 Instances

Target Group 1 contains several EC2 instances of type `t2.micro` running a FastAPI service [5], intended for lighter workloads in the benchmark. Target Group 2 contains several EC2 instances of type `t2.large` running the same FastAPI service, enabling side-by-side comparison of performance across instance sizes.

1.1.2 Monitoring and Metrics

Both the custom load balancer and the EC2 instances publish metrics to Amazon CloudWatch [2], where dashboards and alarms track request counts, latency, errors, healthy host counts, and resource utilization. CloudWatch reports custom load balancer metrics at 1-minute intervals while traffic is flowing [1], enabling verification of system behavior during each benchmark run.

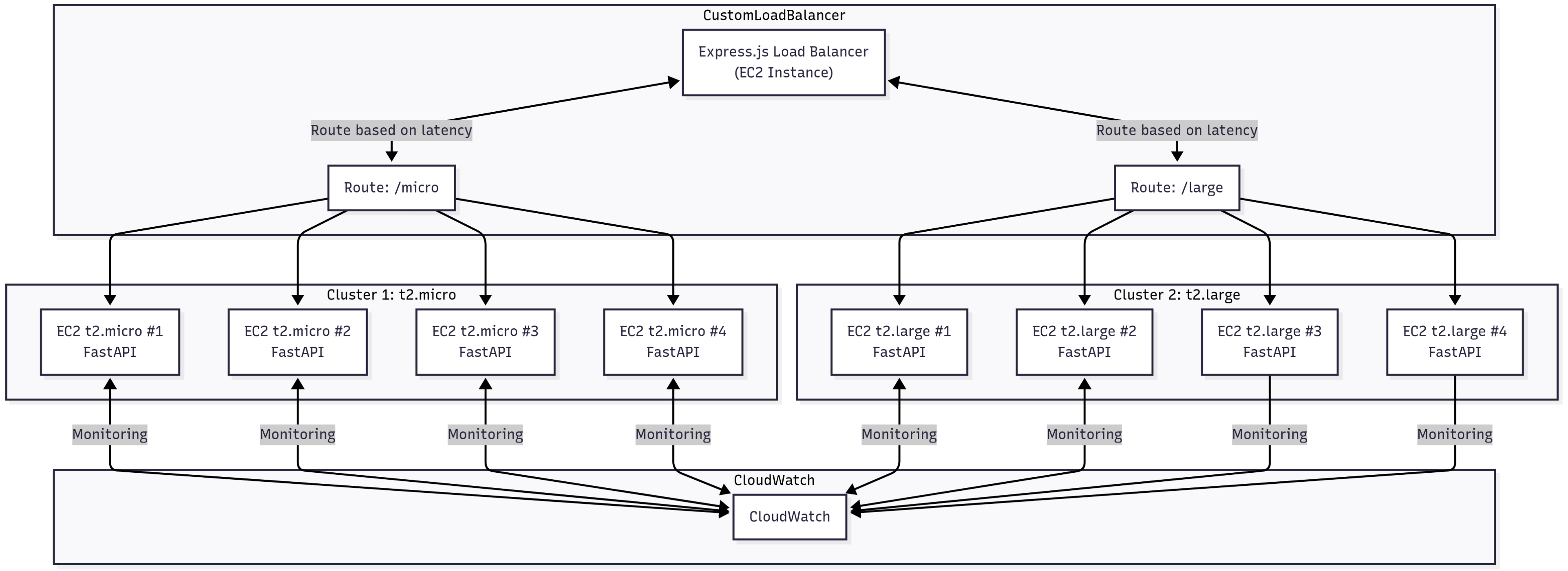

Figure 1.1: AWS Load balancing benchmark architecture.

Chapter 2

EC2 Clusters Setup

2.1 Automated Deployment with Deployment Script

The following procedure is implemented in `TP1.py` to automate the setup of two EC2 clusters and configure an EC2 instance which act as a custom load balancer with path-based routing:

1. **Add Inbound Rule for Security Group:** (`create_my_ip_inbound_sg_rule(sg_id)`)
   - Adds an inbound rule to the project security group allowing all traffic from the caller's current public IP address.
   - When AWS creates a default VPC for an account in a region, it also creates a default security group for that VPC.
2. **Create EC2 Instances:** (`create_t2_micro_instances(sg_id)`, `create_t2_large_instances(sg_id)`)
   - Launches 4 `t2.micro` instances for Cluster 1, each running `main_cluster1.py`.
   - Launches 4 `t2.large` instances for Cluster 2, each running `main_cluster2.py`.
   - Each instance is provisioned in a specific subnet, automatically created with the account's VPC, and tagged for identification.
   - A startup script is injected to automatically launch the FastAPI app on boot.
3. **Create Custom Load Balancer:** (`create_load_balancer()`)
   - Deploys and creates the custom load balancer in an EC2 instance attached to the specified security group. The load balancer listens on port 8000 with a default fixed 404 response.
4. **Wait for Instances:** (`wait_for_instances()`)
   - The script waits until all EC2 instances are in the `running` state and retrieves their public DNS names.
5. **Initialize Load Balancer Instance:** (`init_load_balancer(public_dns)`)
   - Copies the load balancer source files to the EC2 instance using `scp`.
   - Connects to the instance via SSH, sets execute permissions on the bootstrap script, and runs it to install dependencies and launch the load balancer application.
6. **Initialize Cluster Instances:** (`init_cluster(instances, cluster_name)`)
   - For each EC2 instance in a cluster, securely copies the FastAPI application files and bootstrap script to the instance.

- Connects via SSH, sets execute permissions, and runs the bootstrap script to install dependencies and launch the FastAPI web API.
- The deployed FastAPI application exposes endpoints for health checks and request handling, logging each request with the instance and cluster information.

Chapter 3

FastAPI Deployment Procedure

Each EC2 virtual machine instance in our deployment is provisioned using an official Amazon Linux AMI. After instance creation, our deployment script (`TP1.py`) copies the FastAPI application code and a bootstrap script to each instance. The bootstrap script is then executed remotely via SSH to automate the setup and launch of the FastAPI application using Uvicorn as the ASGI server [4].

3.1 Automated Deployment with Bootstrap Script

The deployment process for each EC2 instance is as follows:

1. **Copy Application Files:** The script copies the `app/` directory (containing the FastAPI code and `bootstrap.sh`) to the EC2 instance using `scp`.
2. **Install Dependencies:** The bootstrap script installs Python 3 and pip, then installs FastAPI and Uvicorn using pip.
3. **Set Environment Variables:** The script sets the `INSTANCE_ID` and `CLUSTER_NAME` environment variables for the running process.
4. **Launch FastAPI Application:** The bootstrap script starts the FastAPI application with Uvicorn as the `ec2-user`, binding to all interfaces on port 8000. Logs are redirected to `uvicorn.log`.

The relevant part of the bootstrap script is:

```
#!/bin/bash

sudo yum update -y
sudo yum install python3 python3-pip -y

pip3 install fastapi uvicorn

cd /home/ec2-user/app

export INSTANCE_ID=$1
export CLUSTER_NAME=$2

sudo -E -u ec2-user INSTANCE_ID="$1" CLUSTER_NAME="$2" \
  /home/ec2-user/.local/bin/uvicorn main_cluster:app \
  --host 0.0.0.0 --port 8000 > uvicorn.log 2>&1 &
```

Chapter 4

Custom Load Balancer Setup and Implementation

4.1 Overview

To provide fine-grained control over request routing and to benchmark custom load balancing strategies, we implemented a custom load balancer as a Node.js application with Express.js running on a dedicated EC2 instance. This load balancer dynamically discovers backend FastAPI instances in each cluster and forwards incoming requests to the optimal instance based on real-time health and response time checks.

4.2 Deployment Procedure

The deployment of the custom load balancer is automated in the `TP1.py` script via the `init_load_balancer` function. The steps are as follows:

1. **File Transfer:** The load balancer source code (Node.js project) is securely copied to the EC2 instance using `scp`.
2. **Remote Setup:** The script connects to the instance via SSH and executes a bootstrap script to:
    - Update system packages.
    - Install Node.js (using NVM).
    - Install project dependencies with `npm install`.
    - Launch the load balancer as a background process.

4.3 Bootstrap Script

A shell script (`bootstrap.sh`) is executed on the load balancer instance to automate environment setup and application launch:

4.4 Load Balancer Application Code

The core of the custom load balancer is implemented in `index.js`. Its main features are:

- **Dynamic Discovery:** Uses AWS SDK to list all running EC2 instances tagged for each cluster.

- **Health and Latency Checks:** Periodically probes each backend instance to determine the healthiest and fastest target.
- **Request Forwarding:** Forwards incoming HTTP requests to the best backend instance for each cluster.
- **Express Server:** Exposes endpoints (`/cluster1`, `/cluster2`) for routing client requests.

Chapter 5

Benchmark Results

In this section, we analyze the outcomes of our load balancing benchmark tests performed on the FastAPI application deployed across different EC2 cluster configurations. The primary objective of these benchmarks is to assess how the application responds to varying levels of simulated user load.

5.1 Benchmark Methodology

To test the response times and throughput of our FastAPI application, we utilized the `benchmark_cluster.py` script. This script simulates concurrent user requests to the application endpoints hosted on two distinct clusters: one consisting of `t2.micro` instances and the other of `t2.large` instances. Each cluster was accessed via the Application Load Balancer (ALB) using path-based routing. The benchmarking process involved the following steps:

1. **Load Simulation:** The script generated a specified number of concurrent requests (one thousand) to the application endpoints, simulating real-world user traffic.
2. **Data Collection:** For each request, the script recorded the response time and status code, allowing us to evaluate both performance and reliability. We can also monitor the CPU usage of each instance during the benchmark using CloudWatch metrics.
3. **Analysis:** The collected data was analyzed to determine average response times, error rates, and throughput for each cluster configuration.
4. **Comparison:** Finally, we compared the performance metrics between the two clusters to understand the impact of instance type on application responsiveness under load.
5. **Results Logging:** The results of the benchmarks were logged into separate files for each cluster configuration, enabling detailed post-benchmark analysis.

5.2 Results Overview

The benchmarking results clearly show that the `t2.large` cluster (0.8s/1000 requests) outperformed the `t2.micro` cluster (0.9s/1000 requests) in both response time and throughput. All requests to both clusters received HTTP 200 responses, confirming the reliability and correctness of the deployment. However, the `t2.large` instances were able to process requests more quickly and efficiently, maintaining lower latency even as the simulated load increased.

This performance difference highlights the significant impact that EC2 instance type has on application responsiveness and scalability. While both clusters were stable under load, the larger instance type consistently delivered better results, making it more suitable for high-demand scenarios.

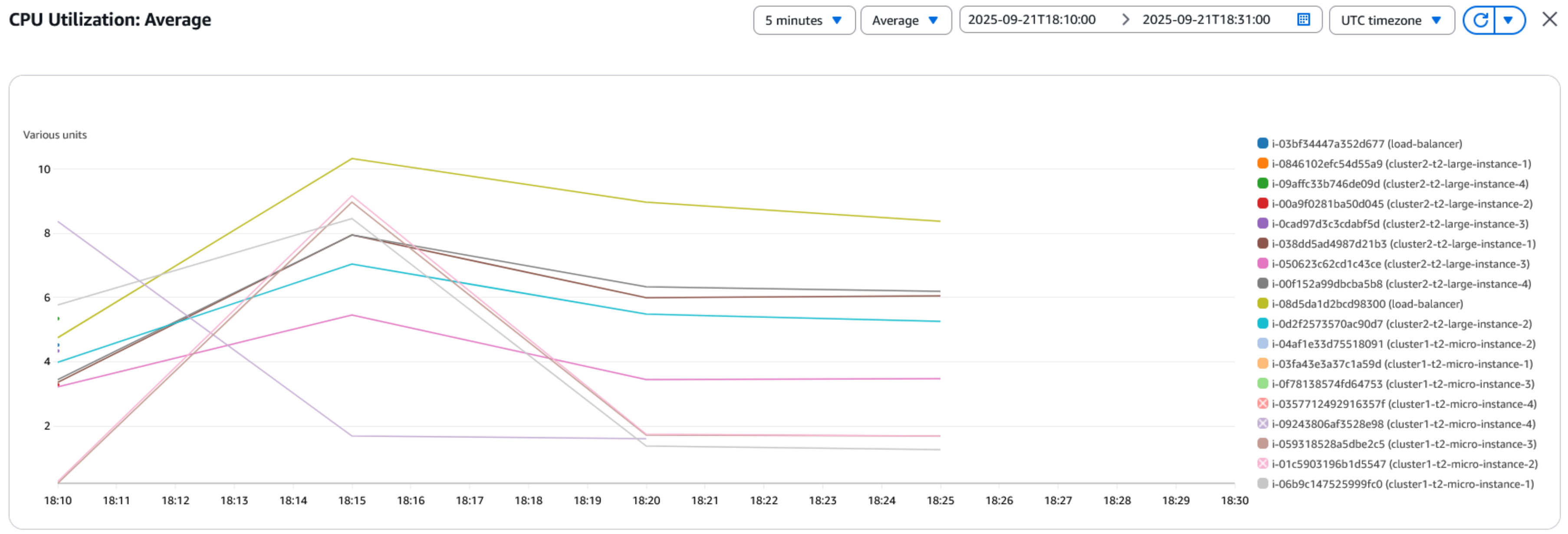

Figure 5.1: CPU usage benchmark results for the load balancer, the t2.micro instances and t2.large instances.

The CPU usage remained below 10 percent for all instances, we can notice that the load balancer had the highest CPU usage, which is expected since it handles all incoming requests and distributes them to the backend instances (figure 5.1). The t2.large instances had a slightly higher CPU usage compared to the t2.micro instances, which is also expected due to their higher processing capabilities.

Chapter 6

Instructions to Run the Code

Instructions to run the project code are provided in the project `README.md` file. Please refer to the `README.md` for detailed setup and execution steps.

Bibliography

[1] CloudWatch metrics for your Application Load Balancer - Elastic Load Balancing.

[2] Amazon Web Services. Amazon cloudwatch. `https://aws.amazon.com/cloudwatch/`, 2024. Accessed: 2025-09-12.

[3] Amazon Web Services. Amazon ec2 (elastic compute cloud). `https://aws.amazon.com/ec2/`, 2024. Accessed: 2025-09-12.

[4] Tom Christie and contributors. Uvicorn: The lightning-fast asgi server. `https://www.uvicorn.org/`, 2024. Accessed: 2025-09-12.

[5] Sebastián Ramírez. Fastapi: The modern, fast (high-performance), web framework for building apis with python 3.6+ based on standard python type hints. *GitHub*, 2021.