SAPIENZA
UNIVERSITÀ DI ROMA

Data Science Master's Degree

Scalability and performance of an online shop

CLOUD COMPUTING

Students:
Laura Concari, 1890490
Francesco Sbordone, 1969869

Academic Year 2023/2024

Contents

1 Introduction

In this project, we developed a web application using Django to simulate a simple online store where users can browse and order products. The application is containerized using Docker, which helps to package the application and its dependencies, making it easy to deploy. We also utilized various AWS services to improve the system's scalability, reliability, and efficiency.

2 Implementation

We built a Django-based web application that acts as an online shop. To ensure that the application can be easily deployed and run on different systems, we created a Docker file:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim
# Set the working directory in the container
WORKDIR /app
# Install necessary system dependencies
RUN apt-get update && apt-get install -y gcc libpq-dev
build-essential
# Copy the current directory contents into the container /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Ensure psycopg2-binary is installed
RUN pip install psycopg2-binary
RUN pip install boto3
# Collect static files
RUN python manage.py collectstatic --noinput
# Make port 80 available to the world outside this container
EXPOSE 80
# Set the DJANGO_SETTINGS_MODULE environment variable
ENV DJANGO_SETTINGS_MODULE=online_shop.settings
# Run the application with Gunicorn
CMD ["gunicorn", "--workers", "3", "--bind", "0.0.0.0:80", "online_shop.wsgi:application"]
```

The Python code and Docker file were uploaded to an EC2 instance (t2.medium type) with an Elastic IP assigned for easy access.

To start the Docker container and run the server on port 80, we used the following commands:

```
docker build -t my-django-app .
docker run -d -p 80:80 my-django-app
```

We also set up an RDS (Relational Database Service) database, which is the best option for Django due to its compatibility with PostgreSQL. This database stores all the information related to orders, products, categories, and users.

To access the database:

```
psql -h database-1.cdljbhlm6hi7.us-east-1.rds.amazonaws.com -U postgres -d database-1
```

When a user places an order, an email is sent to them, and the order details are updated in the database.

A manager can add products directly through the interface at

`<ip-server>/accounts/login/manager`.

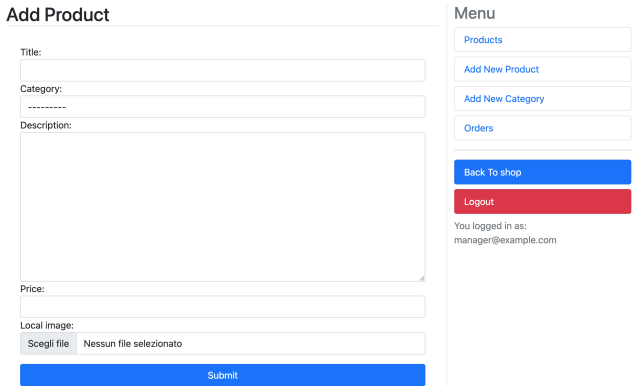


Figure 1: *Manager Interface Screenshot*

In this interface, the manager can select an image from their local storage. When the image is uploaded, it is stored in an S3 bucket, which triggers a Lambda function that resizes the image. Another Lambda function, triggered by an SQS queue, updates the database with the URL of the resized image.

2.1 Implementation of AWS Lambda

To make our Django web application more scalable and efficient, especially when handling images uploaded to an AWS S3 bucket, we used AWS Lambda and Amazon SQS, which are services that allow us to perform tasks automatically and manage them better.

2.1.1 Implementation process

Here's how the system works:

1. **Resizing images automatically**: when an image is uploaded to the S3 bucket, an AWS Lambda function automatically resizes the image to a smaller size (268x200 pixels) while keeping the same proportions. The resized image is then saved in a specific folder within the same S3 bucket. A message is sent to an Amazon SQS queue.
2. **Updating the database**: the SQS queue triggers another Lambda function that updates the PostgreSQL database. This function replaces the original image URL with the URL of the resized image in the `image` field of the `shop_product` table. The web application automatically displays the resized image instead of the original one.

2.1.2 Why we used an SQS queue

Using an SQS queue between the two Lambda functions offers several advantages:

- **Task separation**: improves workload management by isolating image resizing from database updates.
- **Traffic spike handling**: SQS acts as a buffer, allowing continued image resizing during database delays.
- **Reliability**: messages in the SQS queue are retained until processed, ensuring database updates.
- **Monitoring**: is easier, allowing quick issue detection by tracking messages in the queue.
- **Flexibility**: SQS enables adding new functions without altering existing ones.

3 Design

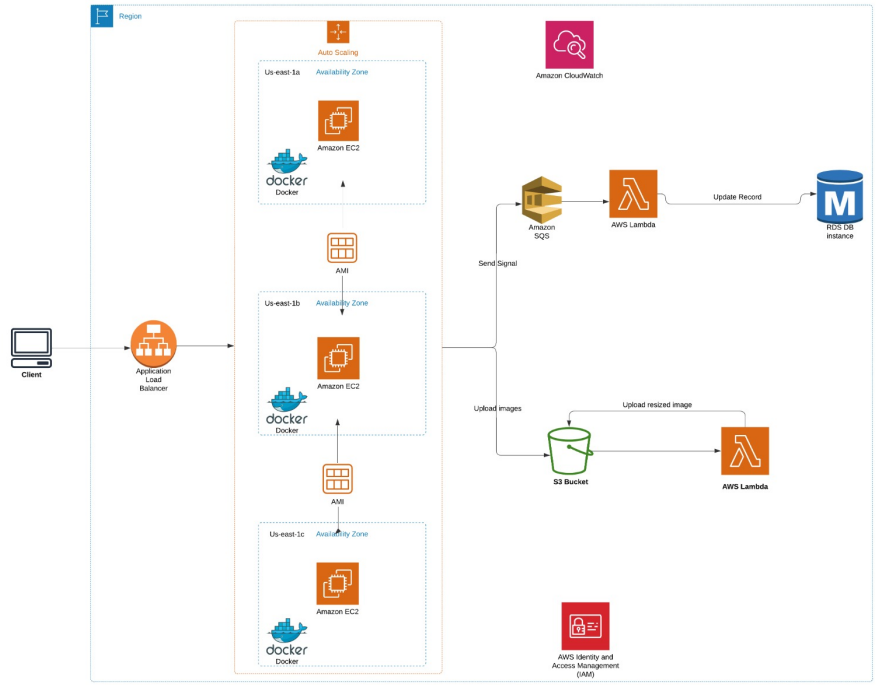


Figure 2: *System architecture using AWS services for auto-scaling, load balancing, and image processing.*

In our architecture, we use an **Application Load Balancer (ALB)** to manage and direct incoming traffic. The DNS address for the ALB is:

http://alb-clc-972897949.us-east-1.elb.amazonaws.com/

We have three **Availability Zones** in our deployment:

- **us-east-1a**
- **us-east-1b**
- **us-east-1c**

These Availability Zones help ensure redundancy and load distribution across multiple zones. The **Auto Scaling Group** is linked to the ALB to automatically start or stop EC2 instances based on current traffic and resource needs.

3.1 Scaling Policies

The Auto Scaling group uses two main scaling policies:

1. **Alarm-based scaling**: This policy is triggered when the CPU usage of an instance exceeds a specific threshold, indicating high usage, and automatically adjusts the instance capacity accordingly.

2. **Target Tracking Scaling**: This policy aims to maintain a target CPU utilization across the instances in the Auto Scaling group. When the CPU utilization exceeds or falls below the target, the group adjusts the number of running instances to balance the load.

Each EC2 instance launched in the group uses a **Custom AMI** with the a script that automates the installation and configuration of a Docker application from a GitHub repository. First, system packages are updated, and Docker and Git are installed. Then, the Docker service is started and enabled to run automatically at system startup. The specified GitHub repository is cloned into the `/home/ubuntu/shop` directory. Once the cloning is completed, a Docker image is built using the `Dockerfile` from the repository, and it is assigned the name "shop-app." Finally, a container based on this image is run, exposing port 80 to allow HTTP access to the application.

This is crucial because it ensures that the instances passing the **Health Checks** are ready for production and are configured to run the required services.

3.2 S3 Bucket Policy

We have added a policy to our **S3 bucket** named `bucket-clc`, which grants public read access to all objects. This means anyone can view and download the files stored in the bucket without requiring authentication.

The policy applied to the bucket is as follows:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::bucket-clc/*"
        }
    ]
}
```

3.3 IAM Role

We use the **LabRole** (the default role for AWS Academy learner labs) to manage permissions and allow services like EC2 and Lambda to interact with other AWS resources such as S3 and RDS.

3.4 VPC and Security Group

All our components (EC2 instances, S3 bucket, RDS, etc.) are within the same **VPC (Virtual Private Cloud)**. This ensures that all the services can communicate securely within the cloud environment.

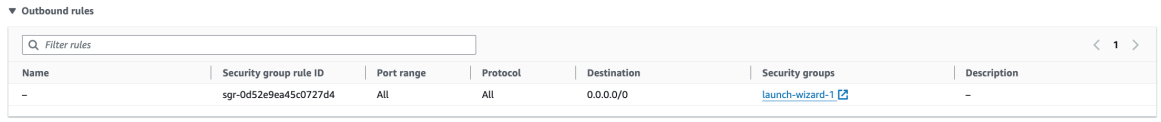
Each service uses the same **Security Group**, with rules that allow HTTP and HTTPS traffic. Here are the details of the security group configuration:

3.4.1 Inbound rules

Filter rules

Name	Security group rule ID	Port range	Protocol	Source	Security groups	Description
–	sgr-008f761f686252daf	22	TCP	0.0.0.0/0	launch-wizard-1	–
–	sgr-028e12f087752103b	80	TCP	0.0.0.0/0	launch-wizard-1	HTTP access for web traffic.
–	sgr-061959f3aae2b3354	443	TCP	0.0.0.0/0	launch-wizard-1	HTTPS access for secure web traffic.
–	sgr-0b8cd08ba7284120d	8000	TCP	0.0.0.0/0	launch-wizard-1	–

3.4.2 Outbound rules



Outbound rules

Filter rules

Name	Security group rule ID	Port range	Protocol	Destination	Security groups	Description
–	sgr-0d52e9ea45c0727d4	All	All	0.0.0.0/0	launch-wizard-1	–

These rules are necessary for allowing inbound HTTP/HTTPS traffic to the instances and enabling the instances to communicate with other services as needed.

3.5 Monitoring

We created **dashboards** to monitor various metrics using CloudWatch. These dashboards track the health and performance of the instances, traffic, and CPU utilization, and provide insights into system behavior. The **CloudWatch Agent** is installed on each EC2 instance to push metrics to CloudWatch for monitoring and alerting.

4 Testing

In this chapter, we explain the different tests we used to check how well our Django - based online shopping application performs. The main goal of these tests was to make sure the application can handle various situations, including normal user traffic, high stress, and long-term use.

We performed several types of tests, each designed to assess specific aspects of the system:

- **Load Testing** was done to see how the application performs under normal conditions when a typical number of users are using it. We looked at how quickly the application responded, how much data it could handle, and how well it used resources like memory and CPU.
- **Stress Testing** involved pushing the system to its limits, far beyond normal use. This helped us find out how the application behaves under extreme pressure and whether it can handle unexpected surges in traffic without breaking.
- **Endurance Testing** was used to see how the application performs over a long period. By keeping it under constant use for several hours, we checked for issues like memory leaks or slowing down, which might not appear in shorter tests.
- **Auto-Scaling Verification** was included to test if our auto-scaling setup works as expected. We set rules to automatically add more server instances when the CPU

usage gets too high. This showed that our system can adjust itself to handle more users and keep working smoothly, even if some servers fail.

The results from these tests give us a clear picture of how well the system can handle real-world scenarios. In the following sections, we will go into more detail about each test, how we conducted them, and what we learned.

4.1 Load Test

The load test was configured with the following parameters:

- Number of Threads (Users): 30
- Ramp-up Period: 20 seconds
- Duration: 20 minutes

This test simulates 30 users accessing the application, with all users starting their activity within the first 20 seconds and continuing to interact with the application for 20 minutes.

The metrics tracked during the test include CPU utilization, memory utilization, network traffic, and database activity.

First of all let's analyze figure 3

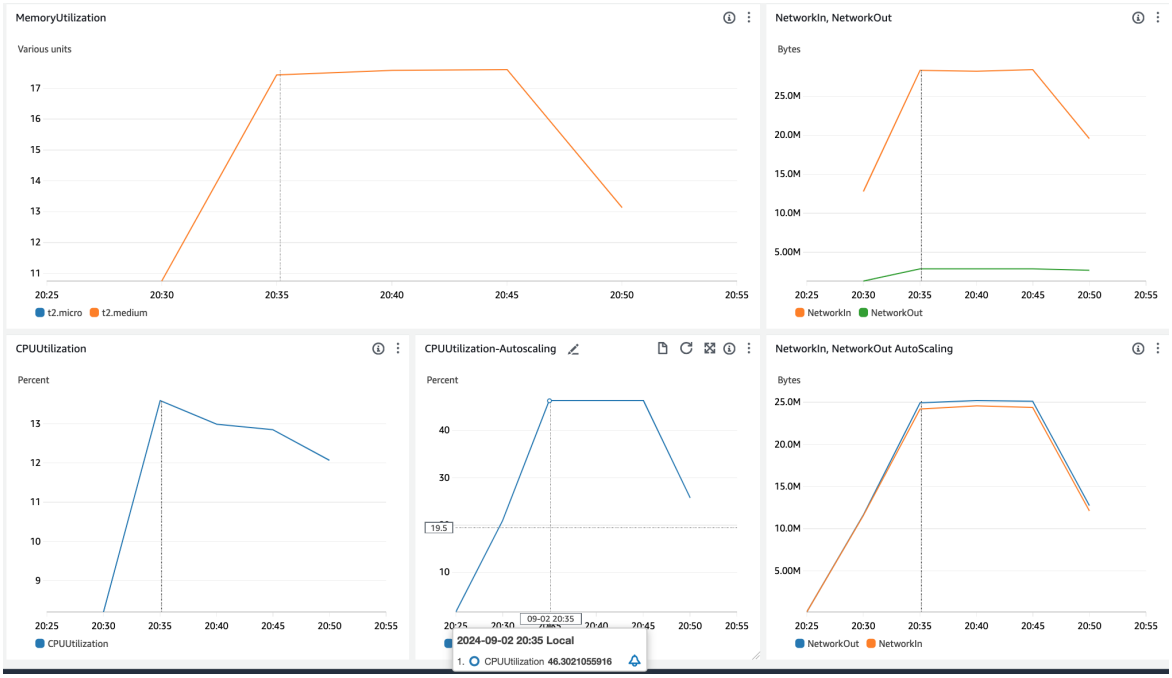


Figure 3: *metrics collected during a load test on the cloud infrastructure side*

1. **CPU Utilization**: the CPU usage on the auto-scaling instance peaks at around 46% during the test. This indicates that the system was actively processing the simulated user requests. The graph shows that after the initial ramp-up, the CPU utilization remains relatively steady, which suggests that the system can handle the load without major performance degradation.

2. **Memory Utilization**: the memory usage increased significantly, especially on the t2.medium instance, which reflects the demands of handling multiple concurrent users. The memory utilization plateau suggests that the system reached a stable state in terms of memory consumption under the given load.

3. **Network Traffic (NetworkIn and NetworkOut)**: the network traffic graphs show spikes in data transfer, particularly outbound traffic (NetworkOut), which correlates with the system responding to user requests. The steady state of the network traffic after the initial spike indicates that the application maintains a consistent level of data processing and communication.

4. **Auto-Scaling Activity**: the auto-scaling mechanism was triggered as CPU utilization approached 46%. However, given the short duration of the test and the quick stabilization of CPU usage, the system likely didn't need to scale out to additional instances. This shows that the auto-scaling settings are properly configured to respond to increased load but are also conservative, preventing unnecessary scaling actions.

Next we focus on the Application Load Balancer (ALB) [Figure 4]. Here's what the graphs depict:

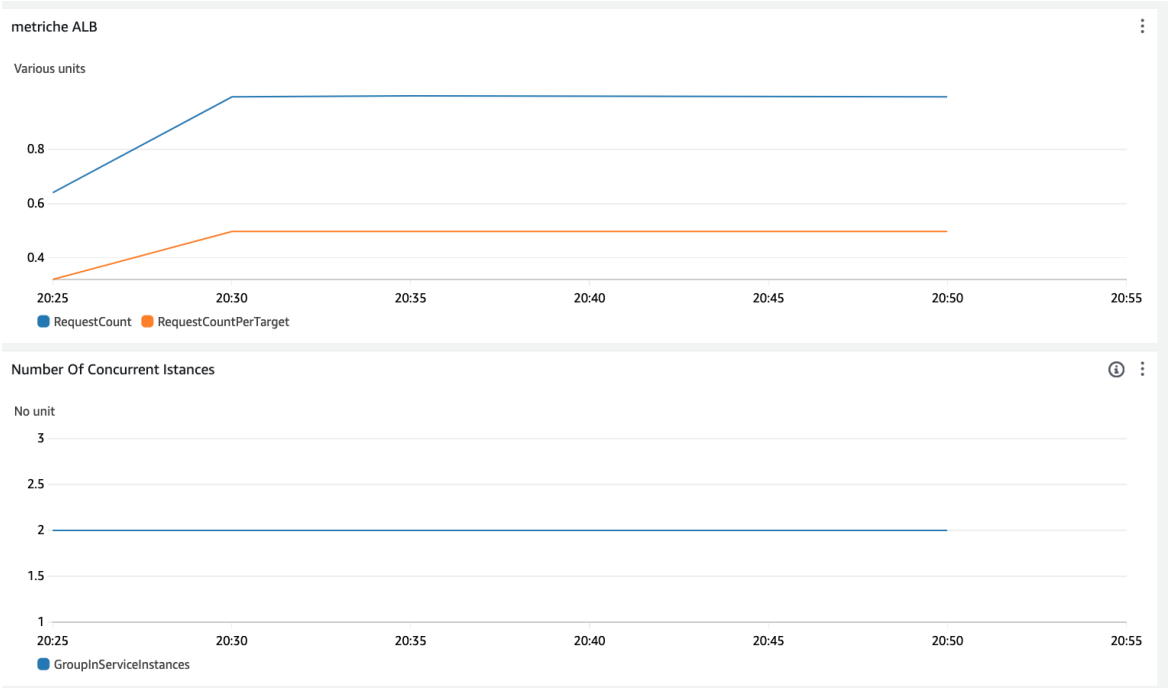


Figure 4: *metrics related to the ALB during the load test*

- **Request Count (top graph)**: the blue line represents the total number of requests received by the ALB over time. The orange line represents the request count per target (i.e., per instance behind the load balancer). Both metrics show an increasing trend until they stabilize, indicating that the load on the ALB increased during the test and then reached a steady state.

- **Number of Concurrent Instances (bottom graph)**: this graph shows the number

of instances that were actively serving traffic behind the ALB during the test. The number remains constant at 2 throughout the test, indicating that no additional instances were added or removed during this specific period

The last image [Figure 5] shows the behavior of the Amazon RDS (Relational Database Service) instance under load

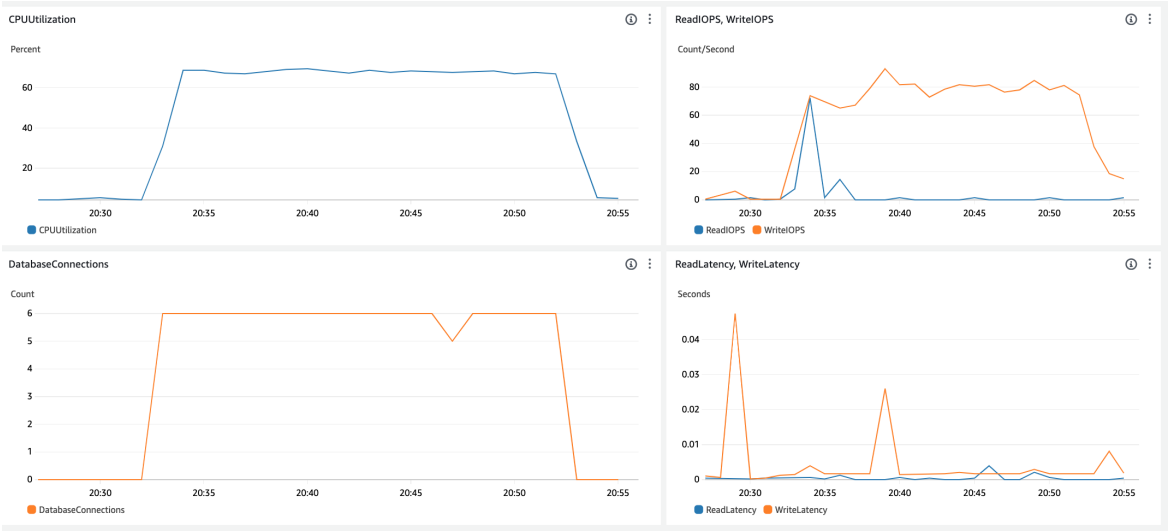


Figure 5: *behavior of the Amazon RDS instance under load*

4.2 Stress Test

Test setting:

- Number of Threads (Users): 150
- Ramp-up Period: 10 minutes
- Duration: 1 hour and 30 minutes

This stress test effectively pushed the system to its limits, particularly in terms of CPU and memory utilization. The database and network metrics indicate that the system was handling a significant load, with substantial write operations and corresponding latency increases. The ALB metrics suggest that the load distribution was managed effectively, without the need for additional instances, which could indicate a well-sized infrastructure for the given load or a potential area to explore further with more aggressive scaling policies.

This detailed analysis highlights the system's strengths and potential bottlenecks, providing valuable insights into its performance under high stress.

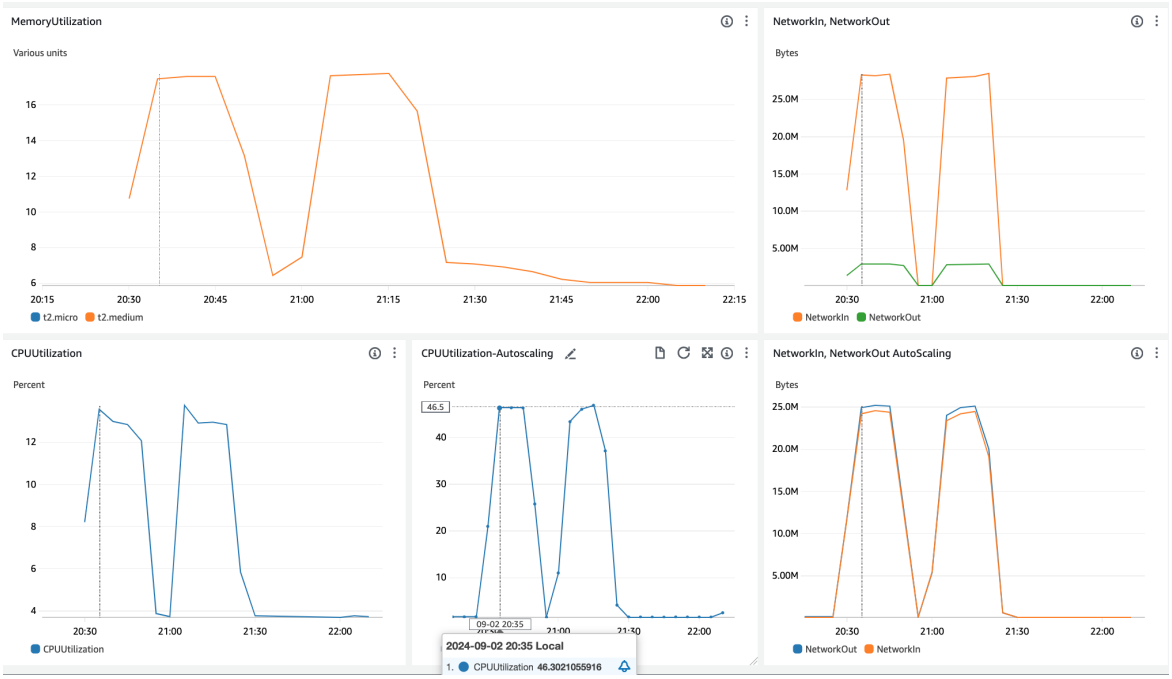


Figure 6: *metrics collected during a stress test on the cloud infrastructure side*

- **CPU Utilization**
  - Interpretation: during peak periods, the system is subjected to significant stress, but it seems to recover quickly once the load decreases. This indicates that the system can handle temporary load spikes without becoming permanently overwhelmed.
- **Memory Utilization**
  - Observation: memory utilization shows a sharp increase at the beginning of the stress test, indicating that the system is allocating more resources to handle the incoming load. The memory usage stabilizes during the peak stress period before gradually decreasing as the load diminishes.
  - Interpretation: the system's memory usage behavior is typical of a stress test, where more memory is required to handle the increased load. The stabilization of memory usage indicates that the system can allocate and manage memory effectively under stress.
- **Network In/Out**
  - Observation: both Network In and Network Out metrics show significant activity during the stress period, with Network Out being more prominent. This indicates that the system is sending more data out than it is receiving.
  - Interpretation: the higher Network Out suggests that the system is primarily dealing with operations that generate outbound data, such as sending responses to requests or writing data to an external storage or database.

Figure 7: *metrics related to the ALB during the stress test*

- **Request Count (Blue Line)**: this shows the total number of requests received by the ALB over time. The graph indicates that the request count increased significantly during the stress periods, peaking at around 0.9 requests per second.
- **Request Count Per Target (Orange Line)**: this metric indicates the average number of requests handled by each target (backend server) behind the ALB. It also shows a similar trend, with an increase during the peak periods and a decrease as the stress test intensity reduced.

The consistent rise and fall in request counts suggest that the ALB successfully managed the incoming load by distributing the requests across the available instances. The aligned patterns between the total request count and the per-target request count indicate that the load was evenly spread across the backend instances.

- **Number of Concurrent Instances**: this metric shows the number of instances that were in service and handling requests during the test. The graph indicates a constant number of 2 instances throughout the test, with no scaling up or down.

Despite the increased load, the number of concurrent instances remained constant. This could indicate that the system was able to handle the stress with the existing instances without triggering the autoscaling mechanism.

Figure 8: *metrics collected during a stress test on the Amazon RDS instance*

- **Read and Write Operations (IOPS)**:
  - Observation: write IOPS (Input/Output Operations Per Second) shows substantial activity, especially during the peak stress periods, reaching as high as 80 operations per second. The Read IOPS, however, remains low, suggesting that the test is more write-intensive.
  - Interpretation: the high write IOPS indicates that the system is performing a significant number of write operations, likely due to the nature of the transactions being processed during the test. The lower Read IOPS suggests that the test does not require as many read operations, or the system's read performance is less stressed compared to write operations.
- **Latency (Read/Write)**:
  - Observation: write latency spikes noticeably during the stress period, with peaks up to 0.04 seconds. The read latency remains relatively low throughout the test.
  - Interpretation: the increase in write latency is a clear sign of the system struggling to keep up with the write operations, which is expected under stress conditions. However, the fact that read latency remains low suggests that the read operations are not as impacted, possibly due to lower read demand or more efficient read handling

4.3 Endurance Test

This is the heavier test done:

- Thread(users): 500
- Ramp-up: 45 minuti
- Duration: 3 ore

This endurance test demonstrates that the system can handle a prolonged high load of 500 users over three hours, with effective use of auto-scaling to manage resources dynamically. The CPU and memory metrics indicate that the system operates near its capacity during peak periods but recovers as the load decreases. The RDS metrics show that the database is handling a significant amount of write operations, with some latency increases under stress. The ALB effectively distributes requests across instances, ensuring even load balancing. This test validates the system's ability to maintain performance over extended periods under heavy load, though it highlights areas such as write latency that may need further optimization for even better performance under sustained stress.

- **CPU Utilization (RDS and CloudSide):**
  - **RDS CPU Utilization:**[10] initially, the CPU usage is steady around 60-80%, indicating that the database is handling a significant number of requests. The CPU usage spikes and then stabilizes, which suggests that the system is managing to handle the load over the long duration but is operating near its capacity during the peak periods. Towards the end of the test, there's a noticeable drop in CPU utilization, indicating that the load on the database has reduced, likely as users begin to finish their sessions or due to a decrease in active transactions.
  - **CloudSide CPU Utilization:**[9] the CloudSide CPU utilization shows a similar trend with an increase during the ramp-up phase and peaks during the test, followed by stabilization. The CPU usage graph shows that the system is actively scaling to handle the load and then reduces as the load decreases. The AutoScaling graph highlights that the CPU utilization was kept in check by dynamically adding or removing instances to handle the incoming requests, showing that the auto-scaling mechanism is functioning as intended.
- **Database Connections and IOPS:**
  - **Database Connections:**[10] the database connections increase rapidly during the initial phase of the test, peaking as expected with the rise in user activity. The connections are stable throughout most of the test, with some fluctuations, indicating that the database is actively managing a large number of concurrent connections. This stability suggests that the connection pooling or management strategies are effective. As the test progresses, there's a gradual decline in active connections, corresponding with the decrease in CPU utilization, signifying the end of active user sessions.
  - **IOPS (Read/Write Operations per Second):**[10] the write IOPS dominate the metrics, indicating that the system is write-heavy during the test. This is consistent with a scenario where users are actively generating data, such as making transactions or updates. The read IOPS are significantly lower, which might suggest that the workload is not heavily read-dependent, or that caching mechanisms are effectively reducing the read load on the database. There are periods where write IOPS spike, which could correlate with specific high-intensity

operations such as batch updates or transaction processing.

Figure 9: *metrics collected during a endurance test on the cloud infrastructure side*

- **Memory Utilization:**[9] The memory utilization on the t2.medium instances increases rapidly during the initial load and remains high throughout the test. This indicates that the application is memory-intensive, and the instances are under significant pressure to handle the active sessions. The drop in memory usage at the end suggests that as user sessions end, memory is freed up, and the system can recover.
- **Network In/Out:**[9] the network in/out graphs show a significant amount of data being processed, with spikes corresponding to periods of high activity. The system is handling a large amount of traffic, which is consistent with the high number of active users. The auto-scaling network metrics also show peaks and troughs, indicating the adjustment of resources to manage the traffic effectively.

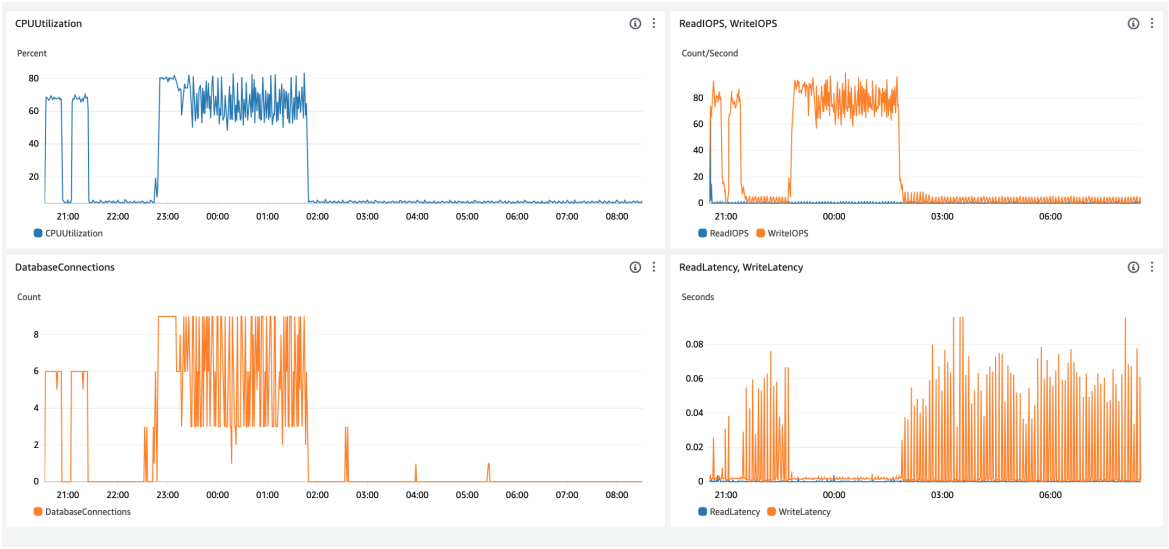


Figure 10: *metrics collected during a endurance test on the Amazon RDS instance*

- **Latency (Read and Write Latency):**
  - **Write Latency:**[10] the write latency increases as the load ramps up, indicating that the system is experiencing delays in processing write operations. This could be a sign of resource contention or the database reaching its limits in handling concurrent writes. The spikes in latency suggest moments of significant stress on the database, where it struggles to maintain performance under the sustained load. Towards the end, the latency decreases, which aligns with the reduction in IOPS and overall load, indicating that the system recovers as the load diminishes.
  - **Read Latency:**[10] the read latency remains relatively low throughout the test, with minor increases. This suggests that read operations are less impacted, potentially due to effective indexing, caching, or optimized read paths in the database.

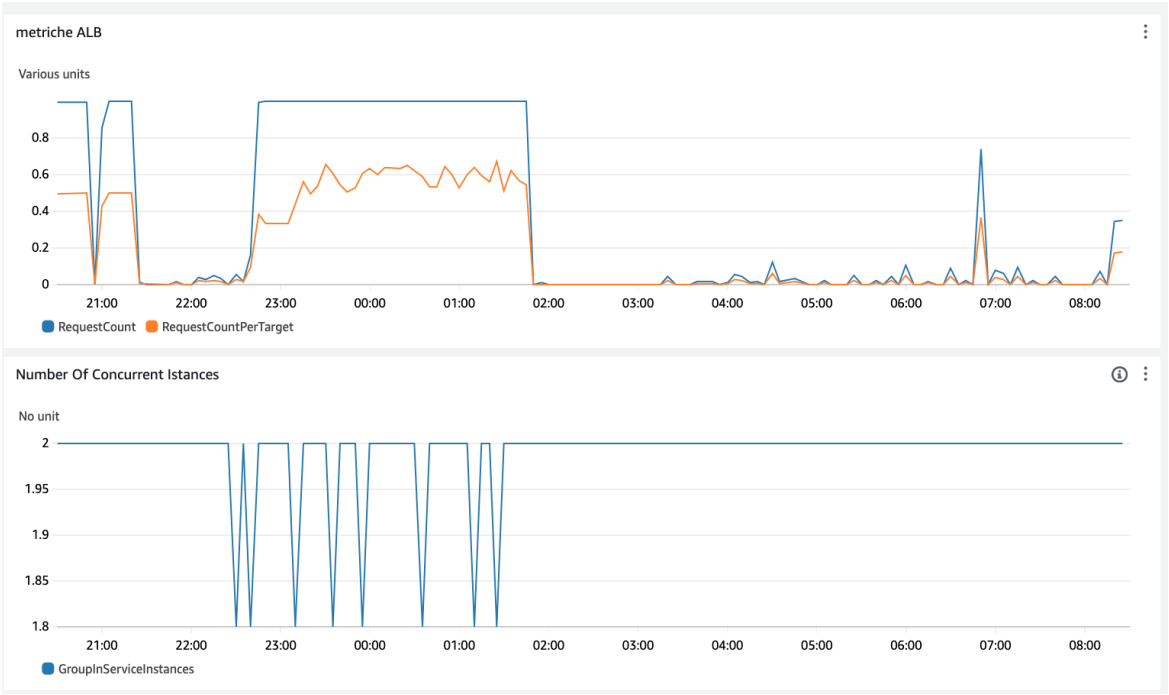


Figure 11: *metrics related to the ALB during the endurance test*

- **ALB Metrics (Request Count and Concurrent Instances):**[11]
  - **Request Count:** the request count steadily increases during the ramp-up and remains high throughout the test, showing that the ALB is efficiently distributing the load among the available instances. The request count per target also increases, which is a direct indicator of the load each instance is handling. The balancing mechanism ensures that requests are distributed evenly, which is crucial for maintaining performance.
  - **Concurrent Instances:** the number of concurrent instances fluctuates slightly, suggesting that the auto-scaling mechanism is adjusting the number of active instances in response to the load. This dynamic scaling is essential to ensure that the system can handle high loads without overprovisioning resources.