Distributed Image Processing Phases 1,2,3,4

Presented to Prof. Ayman Bahaa & Eng. Mustafa Ashraf

Prepared by:

Matthew Sherif 20P6785

Mina Shawket 20P9476

Osama Khaled 20P5486

Sarah Sherif 20P2202

Contents

PHASE 1

Introduction

Image processing applications have become increasingly popular lately. Whether in personal usage such as editing personal photos for social media, or in industrial usage, where machine vision algorithms are used all over manufacturing pipelines. Our project here focuses on the former. We design a **distributed backend** with a **web application frontend**. The user of the application can upload a photo and select the operation he/she wants to apply. The photo will be sent to our distributed backend and the processed image will be **transparently** sent back to the user as if all operations and computations were performed locally on the user's device.

We offer a variety of operations, ranging from highly technical operations such as edge detection and color inversion, to operations that are used by a non-technical user such as background removal.

We also aim to provide a real time overview of the process, where the user can visually see where the image currently is in the distributed system. This would help in understanding the distribution of the work in the cloud.

Requirements

Functional requirements

- The user should be able to request any of the image processing features offered.
- Allow the upload and processing of multiple images.
- Allow the tracking of the status of the processing.

Nonfunctional requirements

- The fact that the system is distributed over the cloud must be hidden, thus achieving **distribution transparency**.
- The Graphical User Interface should be simple, intuitive, and easy to use.
- The system should be easily **scalable**.
- The system should have a reasonable response time, no more than **10 seconds**.

Image processing use cases

In this section, we define the use cases we are aiming for in this project. Our goal is to offer a range of computer vision services for a variety of use cases. We split the services offered into two groups, **technical use cases** and **general-purpose use cases**.

Technical use cases

These are the use cases most likely accessed by a computer vision technical user. Whether the user is curious on how canny edge detection affects the image or the user wants to quickly invert an image through a web interface, we have these use cases covered.

The main intention of these use cases is to allow the user to see the results of different algorithms, without the need to run them locally. We also allow the user to pass the parameters to explore the effects of tweaking the parameters.

Next are the use cases that we aim to implement in the initial phase of the project

Edge Detection

This feature extracts the edges in the image using Canny's algorithm.

Corner Detection

Detect all interest points of an image. For this feature, there are many popular algorithms, such as Harris' algorithm and Shi-Tomasi.

Image filtering

A variety of filters, such as gaussian blur, median filtering.

Fourier Transform

This feature allows the user to see the image in the frequency domain by taking the Fourier transform of the image. Interesting patterns can be found in this domain such as sinusoidal waves.

General-Purpose use cases

These use cases are intended for personal usage.

Contrast enhancement

This feature enhances the contrast of the image using histogram equalization algorithm.

Background removal

Remove the background from an image.

Text detection

Extract the text from an image using deep learning.

Approach

Software architecture

We will create a restful API using flask. This API exposes the backend of our application. We will also have a simple web frontend that sends the user requests to the backend and displays the result.

Here is the logical architecture of the system software components.

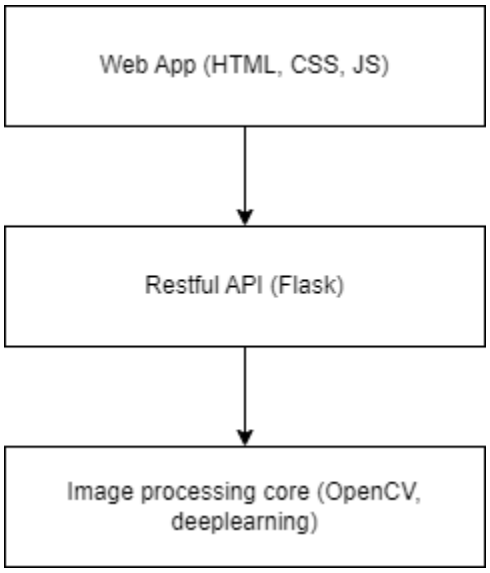


Figure 1: Software Architecture

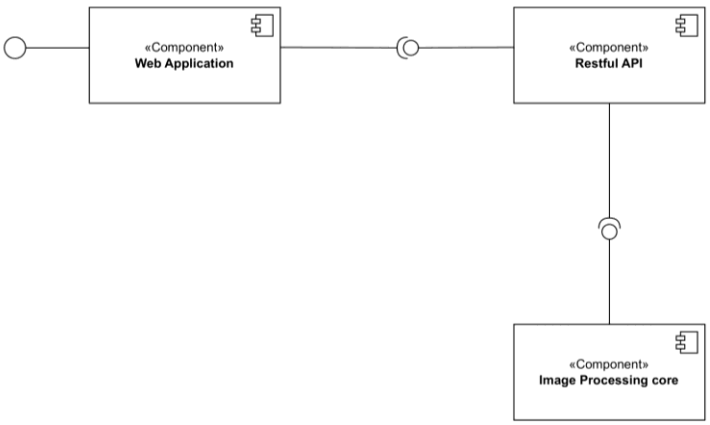


Figure 2: Component Diagram

Deployment architecture

We will use AWS as our cloud provider. Here is a diagram of the initial deployment plan.

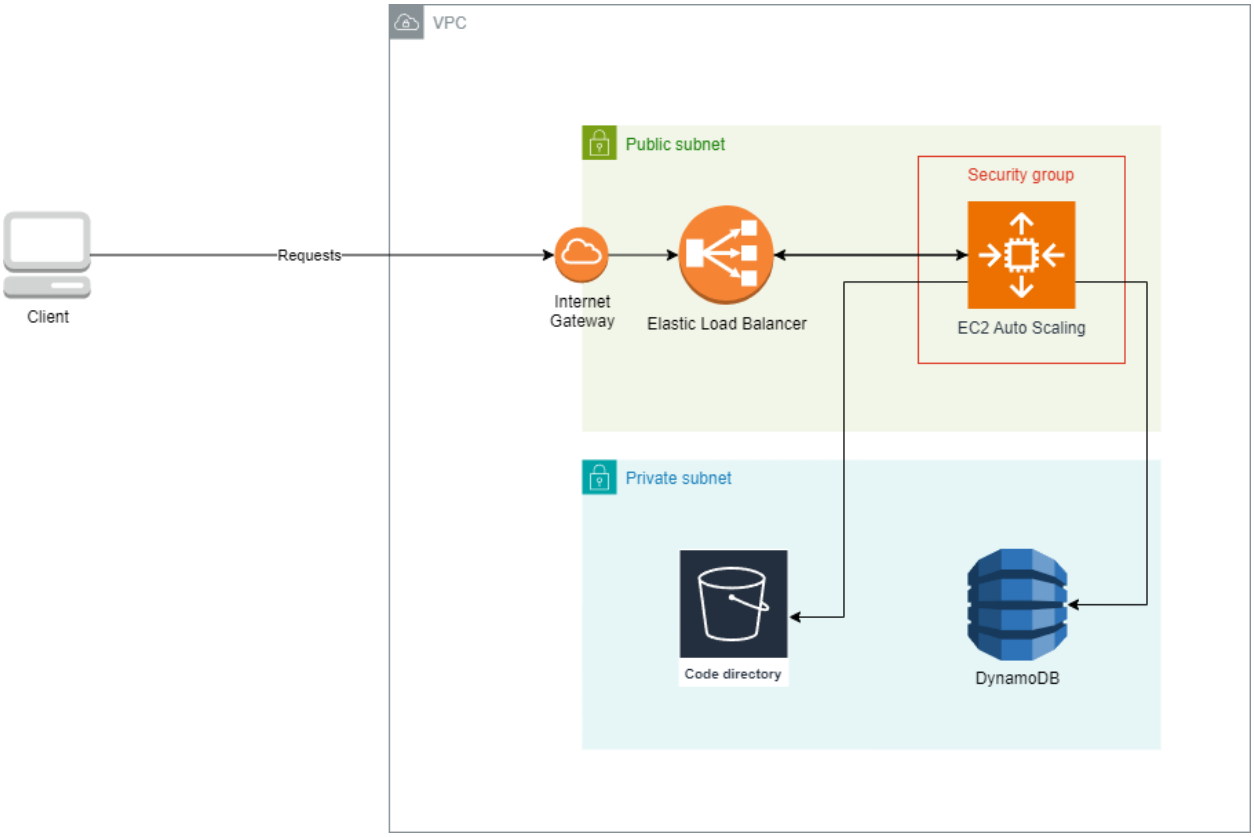


Figure 3: Deployment Diagram

Our server machines will run on Amazon **Elastic Compute Cloud (EC2)** instances. These machines will contain the Restful API that handles the image processing and will also contain the Apache web server to serve the frontend content. We will configure the machines to **automatically scale out** by adding them to Amazon auto scale. This means that if the CPU utilization of the machines exceeds a certain threshold, new EC2 instances will be automatically added. The EC2 instances will run behind an Amazon **Elastic Load Balancer (ELB)** which evenly distributes the work among the running instances. We also use Amazon **Simple Storage Service (S3)** to store our code. This allows new machines to quickly download the code on boot and run without any manual configuration. We also use Amazon **DynamoDB** which is a serverless NoSQL database to store processed images.

Sequence Diagram

The sequence diagram illustrates how messages are being exchanged between the system's components and the order in which they are received.

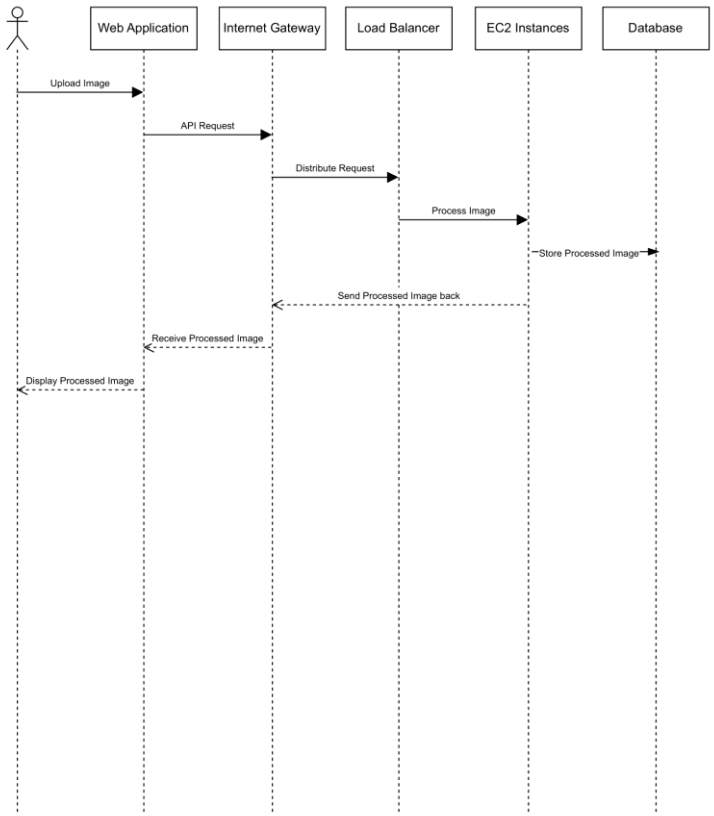


Figure 4: Sequence Diagram

Gantt Chart

This is an overview of the expected time plan of the project. The plan starts on Wednesday 17th of April.

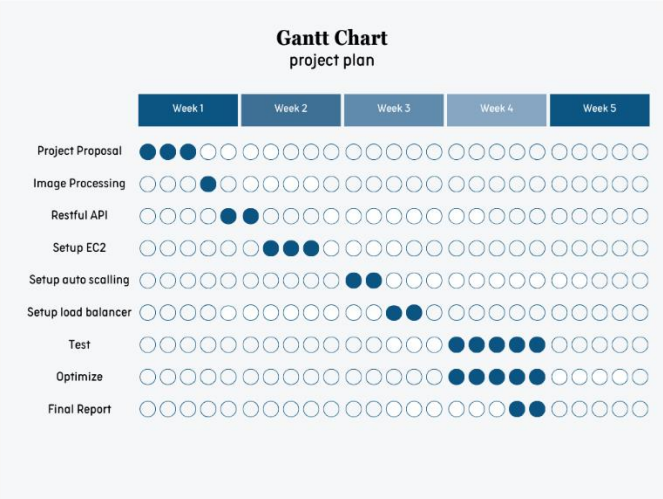


Figure 5: Gantt Chart

PHASE 2

Introduction

In this phase, our objective is to have a working system deployed on the cloud. The focus of this stage is to make sure that the components are working correctly without much emphasis on the distribution and parallelism of the process. In other words, we build a centralized system in this phase that we will distribute in the next phase.

Objective

These are our objectives of this phase:

- Create the image processing core code.
- Create the frontend of the application.
- Create the restful API to be able to access the image processing operations.
- Setup our cloud environment.
- Deploy our centralized system.

Image Processing Core

This python file contains all the necessary code to do the image processing operations. We mainly use OpenCV for image processing, but we also use deep learning in some of our operations. We have applied multiple processing operations on the image requested by the user whereas the user uploads an image and select an image processing operation, which we will discuss next precisely, the image with its requested operation is sent to an EC2 instance (which is an AWS server and has the code for the image processing techniques which will be applied to the image).

Image Processing Techniques:

Edge Detection:

Here we have our image as an argument, and we apply canny edge detection algorithm to our image because it is known for its accuracy and noise removal.

The arguments passed to canny algorithm are threshold1 and threshold2 which specify the lower and upper thresholds used for detecting the edge, aperatureSize

Which is the size of the sobel kernel, and the Boolean parameter L2gradient that specifies the gradient calculation method and set to false because we want to use L1 norm.

```
def edge_detection(image, threshold1=100, threshold2=200):
    """Return the edge map of the image."""
    result = cv2.Canny(image=image, threshold1=threshold1,
                       threshold2=threshold2, apertureSize=3, L2gradient=False)
    return result
```

Figure 6 shows edge detection operation

Corner Detection:

Here we apply corner detection to the image by using cornerHarris corner detection Algorithm. We first convert our requested image from BGR to gray, apply cornerHarris algorithm for getting the corners then dilate the image to highlight the corners.

```
def corner_detection(image):
    """Return the image with the corners/interest points highlighted."""
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Detect corners using Harris Corner Detection
    dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

    # Threshold to highlight corners
    dst = cv2.dilate(dst, None)
    image[dst > 0.01 * dst.max()] = [0, 0, 255]  # Highlight corners in red
```

Figure 7 shows corner detection operation

Gaussian Blur

Here we apply gaussian blurring algorithm to blur an image based on the kernel size that is applied to the image. This kernel passes through the pixels of an image and applies the gaussian filter to the pixels of the image.

```
def gaussian_blur(image, kernel=5):
    """Return the blurred image."""
    result = cv2.GaussianBlur(image, (kernel, kernel), 0)
    return result
```

Figure 8 shows gaussian blur detection

Median Blur

This method applies median blur filter to an image. Median filter is basically used when we want to remove noise, especially salt and pepper noise, and apply median blurring to the image.

```
def median_blur(image, kernel=5):
    """Return the blurred image"""
    result = cv2.medianBlur(image, ksize=kernel)
    return result
```

Figure 9 shows median blur detection

Fourier Transformation

Here we apply Fourier transform to an image to return the image's representation in the frequency domain.

```
def fourier_transform(image):
    """Return the centered fourier transform of the image."""
    if len(image.shape) == 3:
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray_image = image

    # Get the optimal size for the Fourier transform
    rows, cols = gray_image.shape
    m = cv2.getOptimalDFTSize(rows)
    n = cv2.getOptimalDFTSize(cols)

    # Create a padded image for optimal Fourier transform
    padded = cv2.copyMakeBorder(
        gray_image, 0, m - rows, 0, n - cols, cv2.BORDER_CONSTANT, value=0)

    # Perform the Discrete Fourier Transform (DFT)
    dft = cv2.dft(np.float32(padded), flags=cv2.DFT_COMPLEX_OUTPUT)

    # Shift the zero frequency component to the center
    dft_shift = np.fft.fftshift(dft)

    # Compute the magnitude and take the logarithm for better visualization
    magnitude_spectrum = cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1])
    magnitude_spectrum = np.log(
        magnitude_spectrum + 1)  # Add 1 to avoid log(0)

    return magnitude_spectrum
```

Figure 10 shows Fourier transform operation

Contrast Enhancement

This method is used to enhance the contrast of an image by using histogram equalization.

```
def contrast_enhancement(image):
    """Return the image with enhanced contrast using histogram equalization."""
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Apply histogram equalization
    equalized_image = cv2.equalizeHist(gray_image)
    # Original grayscale image
    return equalized_image
```

Figure 11 shows contrast enhancement operation

Background Removal

This method uses deep learning techniques (Convolution neural networks) to identify the foreground objects and chroma keying to remove the backgrounds based on specific color. All these techniques are embedded within the remove method which is implemented within rembg Library.

```
from rembg import remove
```

Figure 12 shows library for background removal

```
def background_removal(image):
    """Return the image without its background."""
    result = remove(image)
    return result
```

Figure 13 shows background removal operation

Invert Image

This operation inverts the colors of an image using bitwise_not operation.

```
def invert_image(image):
    """Return the inverted image."""
    inverted_image = cv2.bitwise_not(image)

    return inverted_image
```

Figure 14 shows invert color operation

Text Detection

We also applied text detection algorithms using easyocr which worked on our local machines but the machine learning model and computations could not fit on the EC2 instance's resources so we disable this feature for now.

```
import easyocr
```

Figure 15 shows easyocr library

```
def ocr(file):
    """Return a string with the text in the image."""
    image_bytes = file.read()
    nparr = np.frombuffer(image_bytes, np.uint8)
    image = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
    reader = easyocr.Reader(['en'])
    result = reader.readtext(image, paragraph="False")
    return result[-1][-1]
```

Figure 16 shows text detection operation

Frontend

This is the front-end view of our web application before uploading any image to be processed

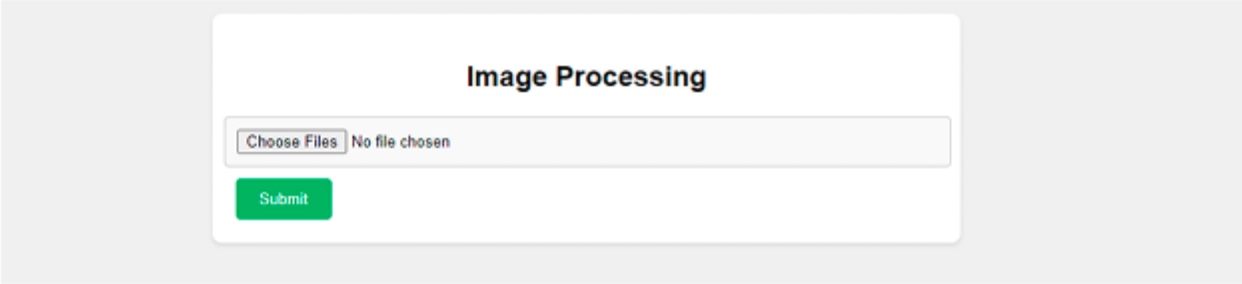


Figure 17 shows webpage view

Images uploaded by user are displayed on the webpage

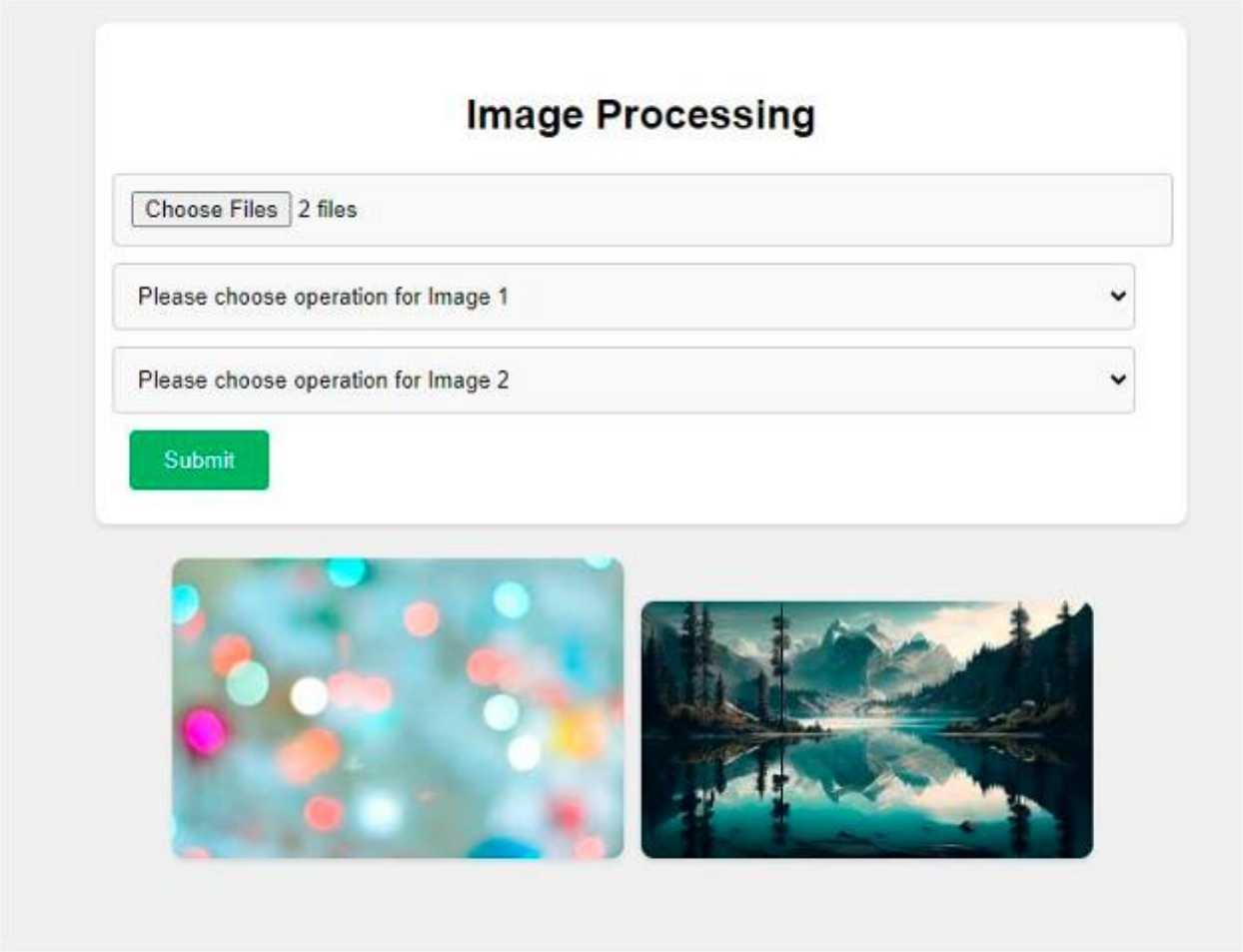


Figure 18 shows images uploaded by user

User chooses what operation to be performed on each image

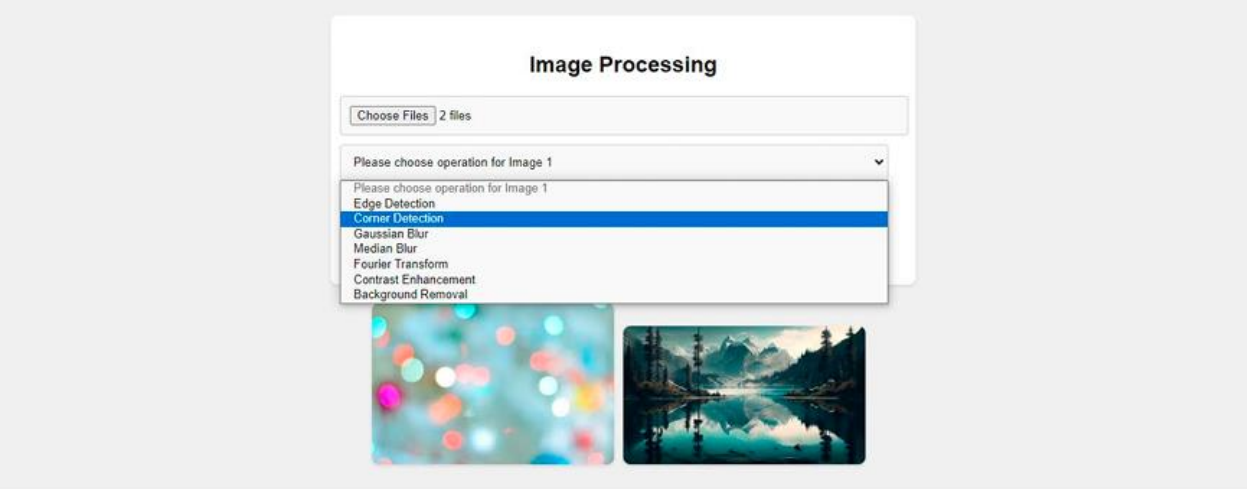


Figure 19 shows operations user chooses from

JavaScript

Mapping each operation name (that is viewed to the user) with its corresponding name on the backend distributed system.

```
operation_name_map = {
  "Edge Detection": "edge_detection",
  "Corner Detection": "corner_detection",
  "Gaussian Blur": "gaussian_blur",
  "Median Blur": "median_blur",
  "Fourier Transform": "fourier_transform",
  "Contrast Enhancement": "contrast_enhancement",
  "Background Removal": "background_removal",
}
```

Figure 20 shows operation name map

Each time user uploads image/images a dropdown box appears for each image with a placeholder “please choose operation for image + image number”

```
document.querySelector('input[type="file"]').addEventListener('change', function (event) {
  let imageFiltersDiv = document.getElementById('imageFilters');
  imageFiltersDiv.innerHTML = ''; // Clear previous inputs

  let files = event.target.files;
  for (let i = 0; i < files.length; i++) {
    let select = document.createElement('select');
    select.name = 'imageFilters[]';
    select.style.display = 'block';
    let options = ['Edge Detection', 'Corner Detection', 'Gaussian Blur', 'Median Blur', 'Fourier Transform', 'Contrast Enhancement',
    for (let j = 0; j < options.length; j++) {
      let option = document.createElement('option');
      option.value = options[j];
      option.text = options[j];
      select.appendChild(option);
    }
    let placeholderOption = document.createElement('option');
    placeholderOption.value = '';
    placeholderOption.text = 'Please choose operation for Image ' + (i + 1);
    placeholderOption.disabled = true;
    placeholderOption.selected = true;
    select.insertBefore(placeholderOption, select.firstChild);
    imageFiltersDiv.appendChild(select);
  }
```

Figure 21 shows how a user chooses an operation

Each uploaded image by the user is displayed on the webpage

```
  let uploadedImagesDiv = document.getElementById('uploadedImages');
  uploadedImagesDiv.innerHTML = '';
  for (let i = 0; i < files.length; i++) {
    let image = document.createElement('img');
    image.classList.add('uploaded-image');
    image.src = URL.createObjectURL(files[i]);
    uploadedImagesDiv.appendChild(image);
  }
});
```

Figure 22 shows how image appears to the user

To deploy distributed property, we needed to create a separate request and response for each image. So, we firstly had to prevent default submission of form

```
document.getElementById('uploadForm').addEventListener('submit', function (event) {
  event.preventDefault(); // Prevent default form submission
```

Figure 23 preventing default submission

We made 2 lists, one that holds images, and the other one holds operations to be performed on each corresponding image

```
let files = document.querySelector('input[type="file"]').files;
let selects = document.querySelectorAll('select[name="imageFilters[]"]');
```

Figure 24 list for images and list for operations

We created a loop that iterates files.length iterations (number of images uploaded by the user) and for each image uploaded a form data variable is defined and image and operation appended to it.

Then a request is sent to a URL, for now this URL is the local host. However, in future phases this URL is going to be the load balancer, so that each image is sent with a separate request to the load balancer which would distribute requests on several EC2 instances

```
for (let i = 0; i < files.length; i++) {
  let file = files[i];
  let operation = selects[i].value;

  let formData = new FormData();
  formData.append('image', file);
  formData.append('operation', operation_name_map[operation]);

  console.log(operation)

  // Send request for each image-operation pair
  let xhr = new XMLHttpRequest();
  xhr.open('POST', 'http://192.168.1.3:5000/process-image', true);
  xhr.responseType = 'blob'; // Set response type to Blob
```

Figure 25 each request is sent independently

This function is called when a response is received from a request (each image is retrieved in a separate response). It first checks if status is 200 (indicating a successful request) Then it checks if (xhr.response). This condition checks if there's a response from the server. If so, image ,retrieved from server is downloaded with the name “image + image number”

```
        xhr.onload = function () {
          console.log(xhr);
          if (xhr.status === 200) {
            if (xhr.response) {
              image_url = URL.createObjectURL(xhr.response);

              // Create a new anchor elemen
              let a = document.createElement('a');
              a.href = image_url;
              a.download = `image ${i + 1}`;

              // Append the anchor to the body
              document.body.appendChild(a);

              // Programmatically trigger a click event on the anchor to start the download
              a.click();

              URL.revokeObjectURL(image_url);
            }
          } else {
            console.error('Error: ' + xhr.statusText);
          }
        };
        xhr.send(formData);
      }
```

Figure 26 download image processed if response received successfully

Restful API

For our API, we use flask to provide a restful interface for the frontend to send requests to. This is our very simple API code.

```python
from flask import Flask, request, make_response
from image_processing import process_image
from flask_cors import CORS

app = Flask(__name__)
CORS(app)


@app.post("/process-image")
def image_processing_endpoint():
    """Take the image and the operation.

    Returns:
        PNG Image: Processed image.
    """
    image = request.files["image"]
    operation = request.form["operation"]
    result = process_image(image, operation)
    response = make_response(result)
    response.headers["Content-Type"] = "image/png"
    return response


# @app.post("/ocr")
# def ocr_endpoint():
#     """Take the image and apply ocr.

#     Returns:
#         PNG Image: Processed image.
#     """
#     image = request.files["image"]
#     result = ocr(image)
#     response = make_response(result)
#     return response
```

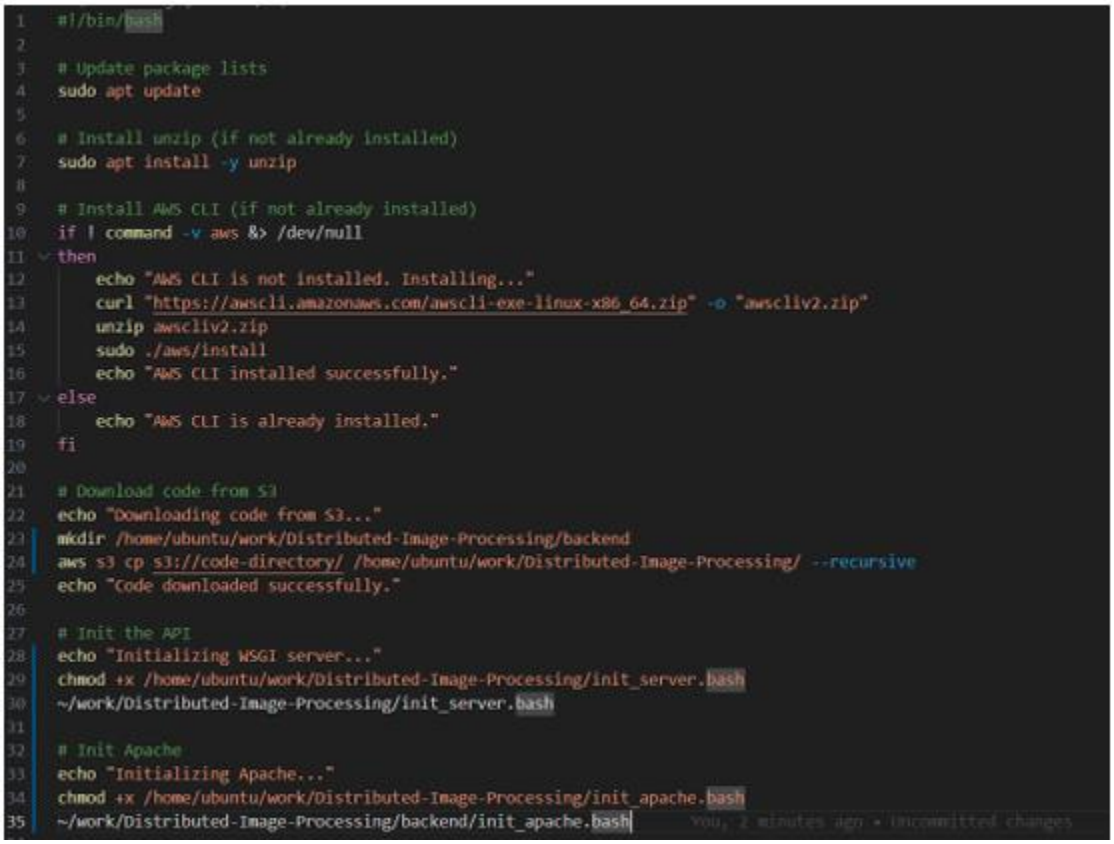
Figure 27 shows flask app

We have two endpoints. /process-image handles all the operations and sends back the processed image. /ocr sends back the scanned text. /ocr is currently disabled due to the EC2 machines limitations.

Cloud Setup

In this stage we just use one EC2 instance to handle all incoming traffic. The EC2 instance contains the WSGI server responsible for handling the rest API requests, and the Apache server for serving the frontend.

Here is the initialization script of the EC2 instance.



```bash
#!/bin/bash

# Update package lists
sudo apt update

# Install unzip (if not already installed)
sudo apt install -y unzip

# Install AWS CLI (if not already installed)
if ! command -v aws &> /dev/null
then
    echo "AWS CLI is not installed. Installing..."
    curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
    unzip awscliv2.zip
    sudo ./aws/install
    echo "AWS CLI installed successfully."
else
    echo "AWS CLI is already installed."
fi

# Download code from S3
echo "Downloading code from S3..."
mkdir /home/ubuntu/work/Distributed-Image-Processing/backend
aws s3 cp s3://code-directory/ /home/ubuntu/work/Distributed-Image-Processing/ --recursive
echo "Code downloaded successfully."

# Init the API
echo "Initializing WSGI server..."
chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_server.bash
~/work/Distributed-Image-Processing/init_server.bash

# Init Apache
echo "Initializing Apache..."
chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_apache.bash
~/work/Distributed-Image-Processing/backend/init_apache.bash
```

Figure 28 shows initialization script of the EC2 instance

Here is the initialization script for the WSGI server.

```
# !/bin/bash        You, 2 hours ago • modified indexe.html and allo

# Install required packages
sudo apt install -y python3-pip
sudo apt install -y python3-venv
sudo apt-get install ffmpeg libsm6 libxext6  -y

# Go to backend directory and create virtual environment
cd ~/work/Distributed-Image-Processing/backend
python3 -m venv venv

# Activate virtual environment
source venv/bin/activate

# Install dependencies
pip3 install -r requirements.txt
pip3 install gunicorn

# Start the server
gunicorn -w 4 -b 0.0.0.0 app:app
```

Figure 29 shows initialization script of the WSGI server

Here is the initialization script for the Apache web server.

```
# !/bin/bash

# Install Apache web server
sudo apt install apache2 -y

# Move the index.html file to the web server root directory
sudo cp ~/work/Distributed-Image-Processing/index.html /var/www/html
```

Figure 30 shows initialization script of the Apache web server

We host our machines and storage in eu-north-1 region.

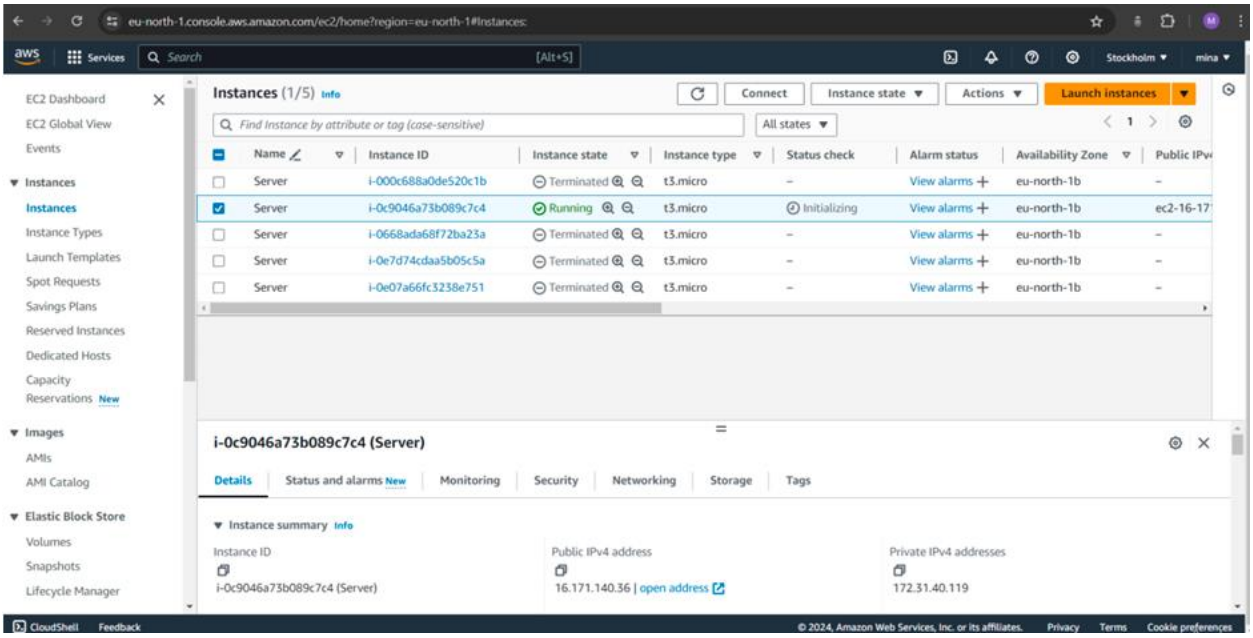


Figure 31 shows EC2 instances

We configure the machine to have read permissions from the S3 to download the code, and we allow inbound traffic to port 8000 for the WSGI server in addition to port 80 for Apache. With this setup, we now have a web server, and a Restful API.

PHASE 3

Introduction

In this phase, we focus on the work distribution aspect of the project. In phase two, we had a fully functioning application with a frontend and backend. The application was running on a single EC2 instance that handled all the traffic. The problem was that should this machine fail, the entire application would fail. In this phase, we horizontally scale the system to accommodate heavier traffic.

In our approach we did the following steps:

- Advanced Image processing operations
- Creating an image for the EC2 instances
- Setting up the load balancer
- Setting up the auto scaling
- Modifying the middleware

Advanced image processing operations

In the previous phase, we had already implemented all image processing operations (basic and advanced). Here is a demo of all the operations we implemented. Notice that the requests are sent to the public IP address of the EC2 instance.

Gaussian blur

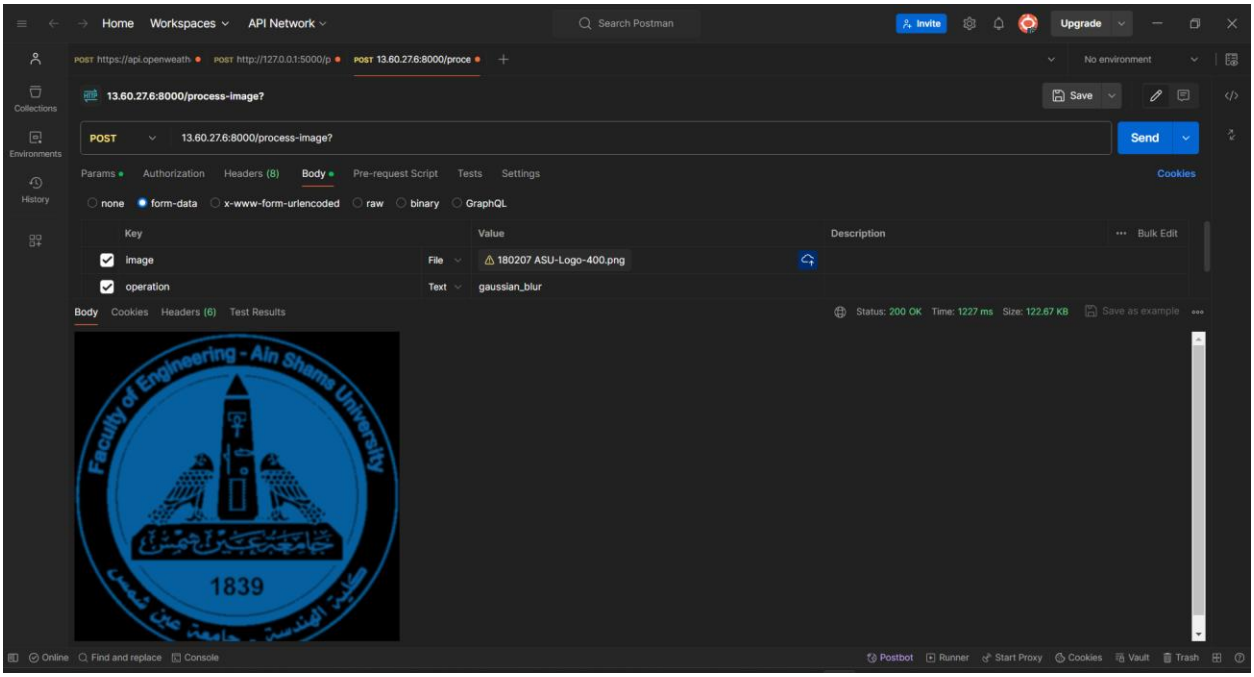


Figure 32 Testing Gaussian blur from postman

Median blur

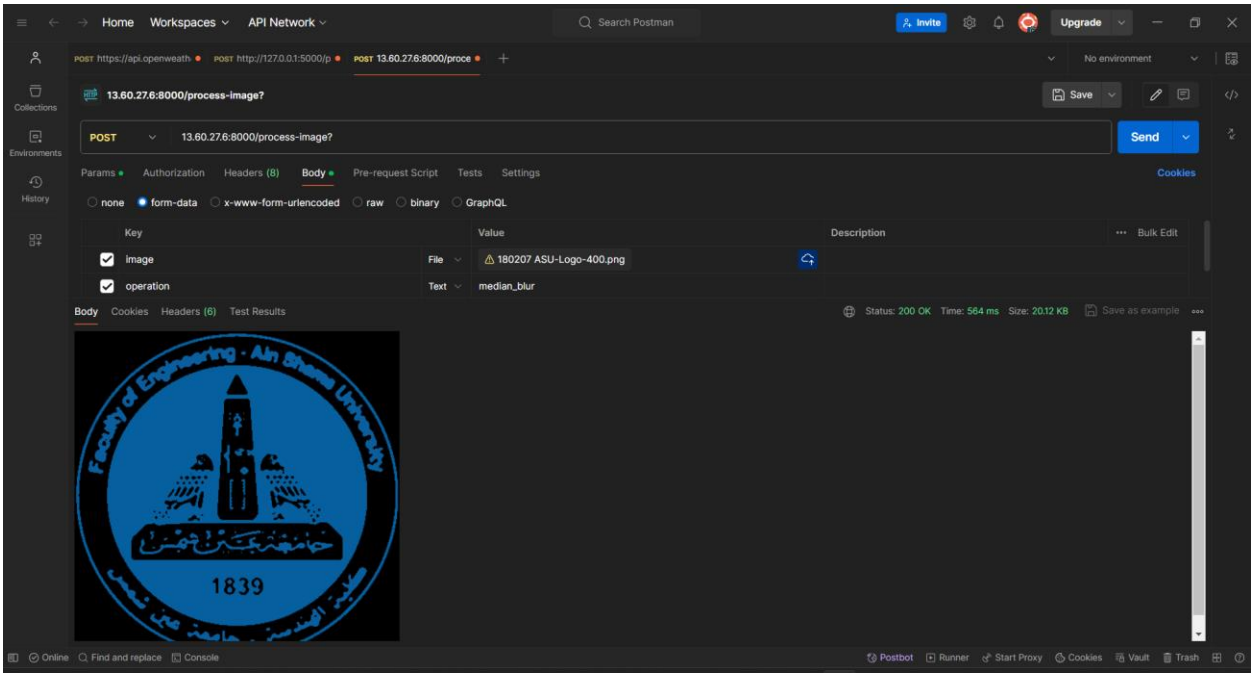


Figure 33 Testing median blur from postman

Fourier transform

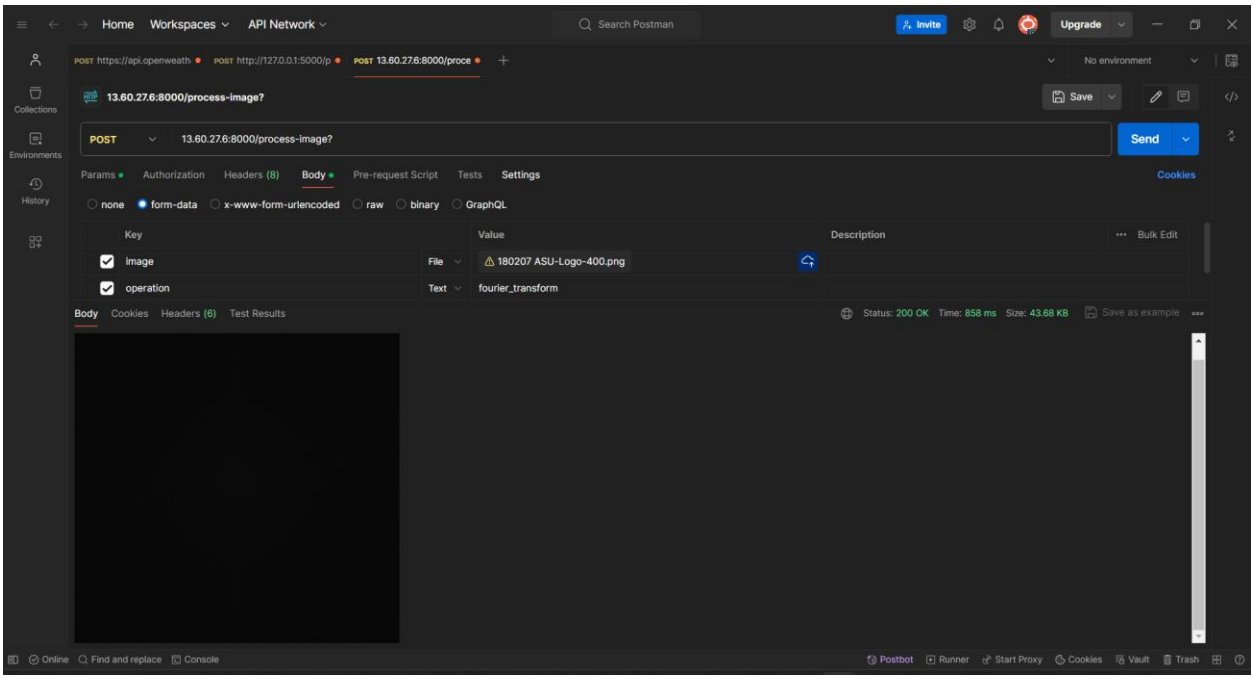


Figure 34 Testing Fourier transform from postman

Contrast enhancement

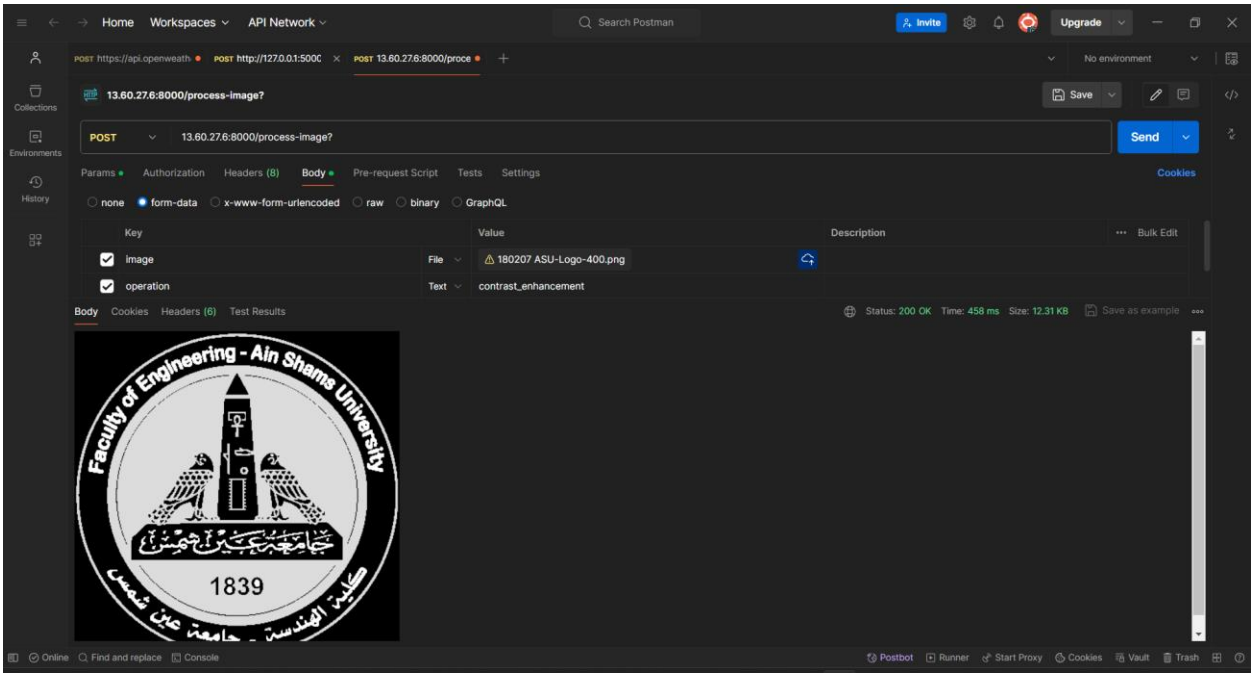


Figure 35 Testing contrast enhancement from postman

Corner detection

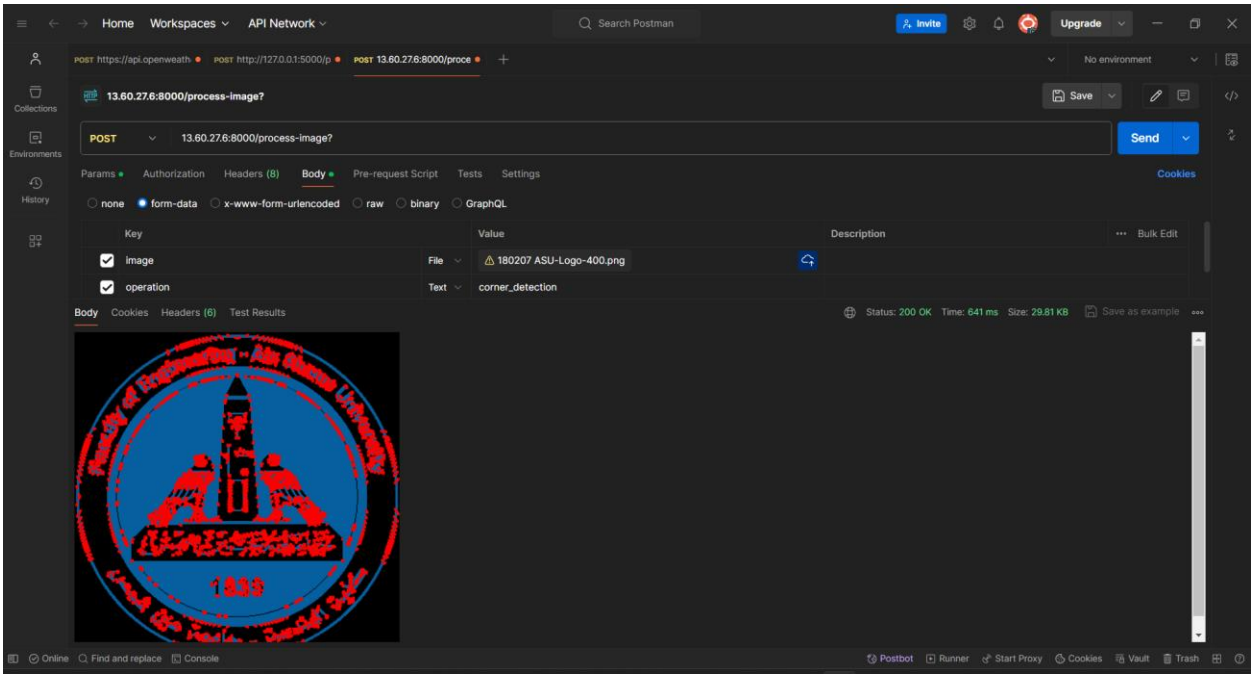


Figure 36: Testing corner detection from postman

Edge detection

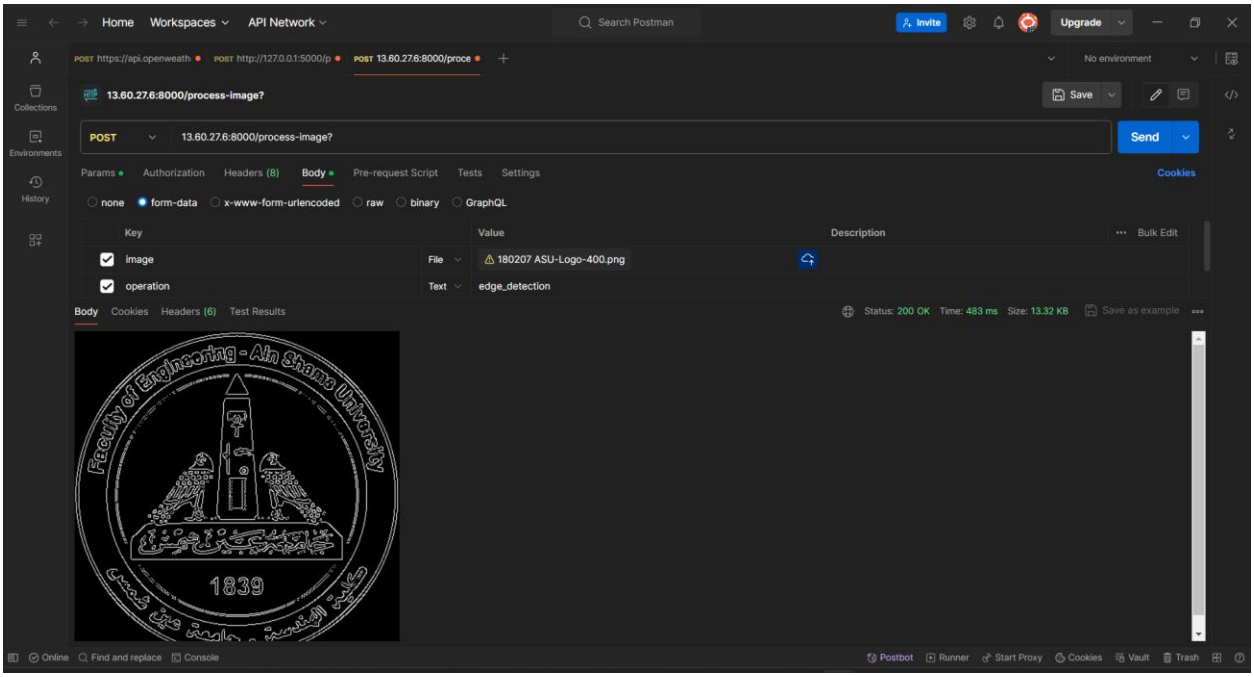


Figure 37Testing edge detection from postman

Color inversion

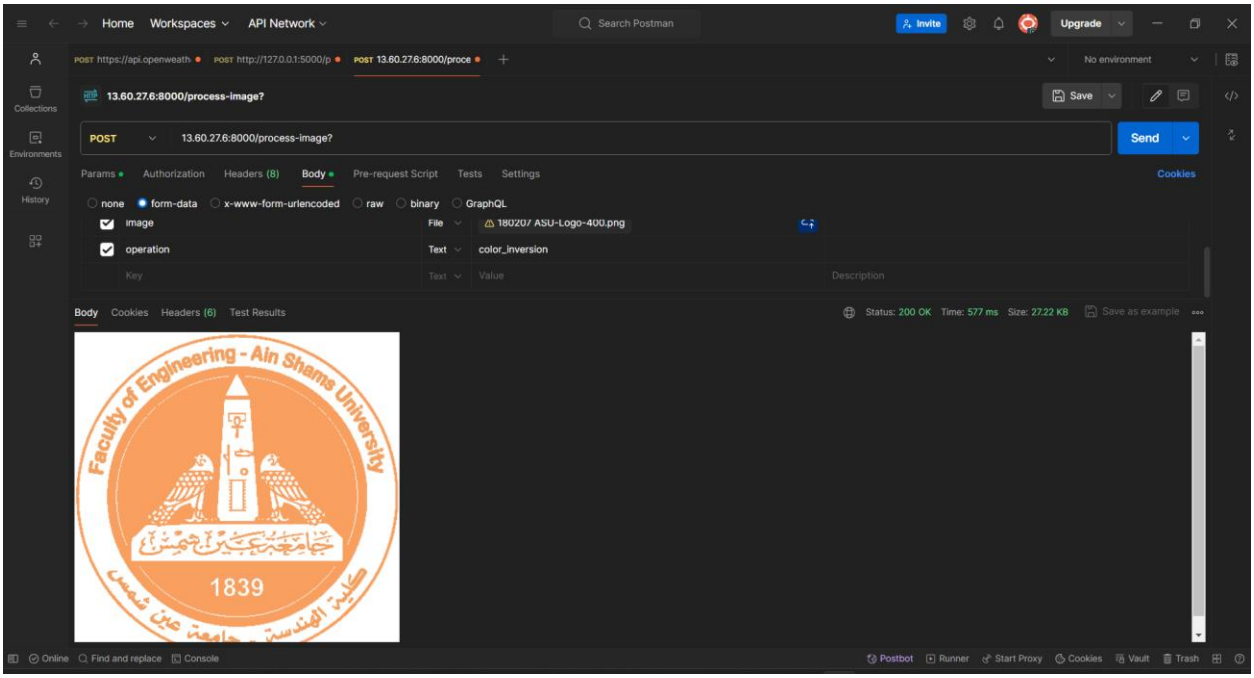


Figure 38 Testing color inversion from postman

Background removal

This operation uses a deep learning model to remove the background. The deep learning model is too large to fit in the instance memory (1GB). This causes the server to fail with out of memory error. We will use this operation to test the how the system can recover from failures.

Creating an image for the EC2 instances

To allow quick and efficient auto scaling, we needed to have an image to launch the EC2 instances from. In the previous phase, we utilized the user data feature in AWS which allowed us to provide a bash script to be run automatically when the instance is booted. Here is the script we used in the previous phase:

```
#!/bin/bash

# Update package lists
sudo apt update

# Install unzip (if not already installed)
sudo apt install -y unzip

# Install AWS CLI (if not already installed)
if ! command -v aws &> /dev/null
then
    echo "AWS CLI is not installed. Installing..."
    curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
    unzip awscliv2.zip
    sudo ./aws/install
    echo "AWS CLI installed successfully."
else
    echo "AWS CLI is already installed."
fi

# Download code from S3
echo "Downloading code from S3..."
mkdir /home/ubuntu/work/Distributed-Image-Processing/backend
aws s3 cp s3://code-directory/ /home/ubuntu/work/Distributed-Image-Processing/ --recursive
echo "Code downloaded successfully."

# Init Apache
echo "Initializing Apache..."
chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_apache.bash
/home/ubuntu/work/Distributed-Image-Processing/init_apache.bash

# Init the API
echo "Initializing WSGI server..."
chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_server.bash
/home/ubuntu/work/Distributed-Image-Processing/init_server.bash
```

Figure 39: old user data script

The problem we faced with this approach is that everytime a new instance is booted, all the libraries and packages needed to be downloaded. This consumed a lot of the data transfer quota in AWS free tier.

In this phase we use a much more effient approach. We created a new custom Amazon Machine Image (AMI) with our code and packages installed. To do this we used the create image feature in AWS, which takes a copy of the storage of an existing machine and use it to create the new image. Now any new machine will be launched from this image and is ready to serve our application.

Setting the Gunicorn daemon

Our EC2 instances have 2 purposes, serve the frontend and serve the application image processing. The frontend is served by the Apache web server which automatically runs in the background as a service daemon. We wanted to do the same for the Gunicorn server (application server). This would allow the Gunicorn server to automatically start on instance boot and reboot.

To do this we created a new file in `/etc/systemd/system/` directory called `app.service`. Here is the content of the file:

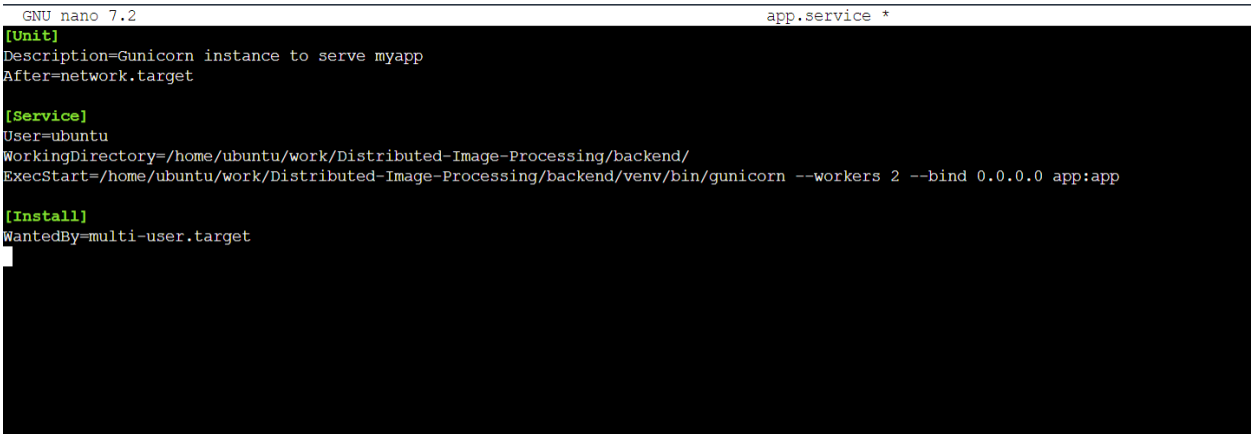


Figure 40 app.service file to setup Gunicorn as a service daemon

Now both the web and application servers will run as daemon service tasks.

This is the setup we used for the Image.

Setting up the load balancer

In this phase we added a load balancer to our application. We used Amazon Elastic Load Balancer (ELB).

We launch the load balancer in three availability zones in the region to provide high availability and fault tolerance. Should a load balancer in one availability zone fail, the other replicas will do its work.

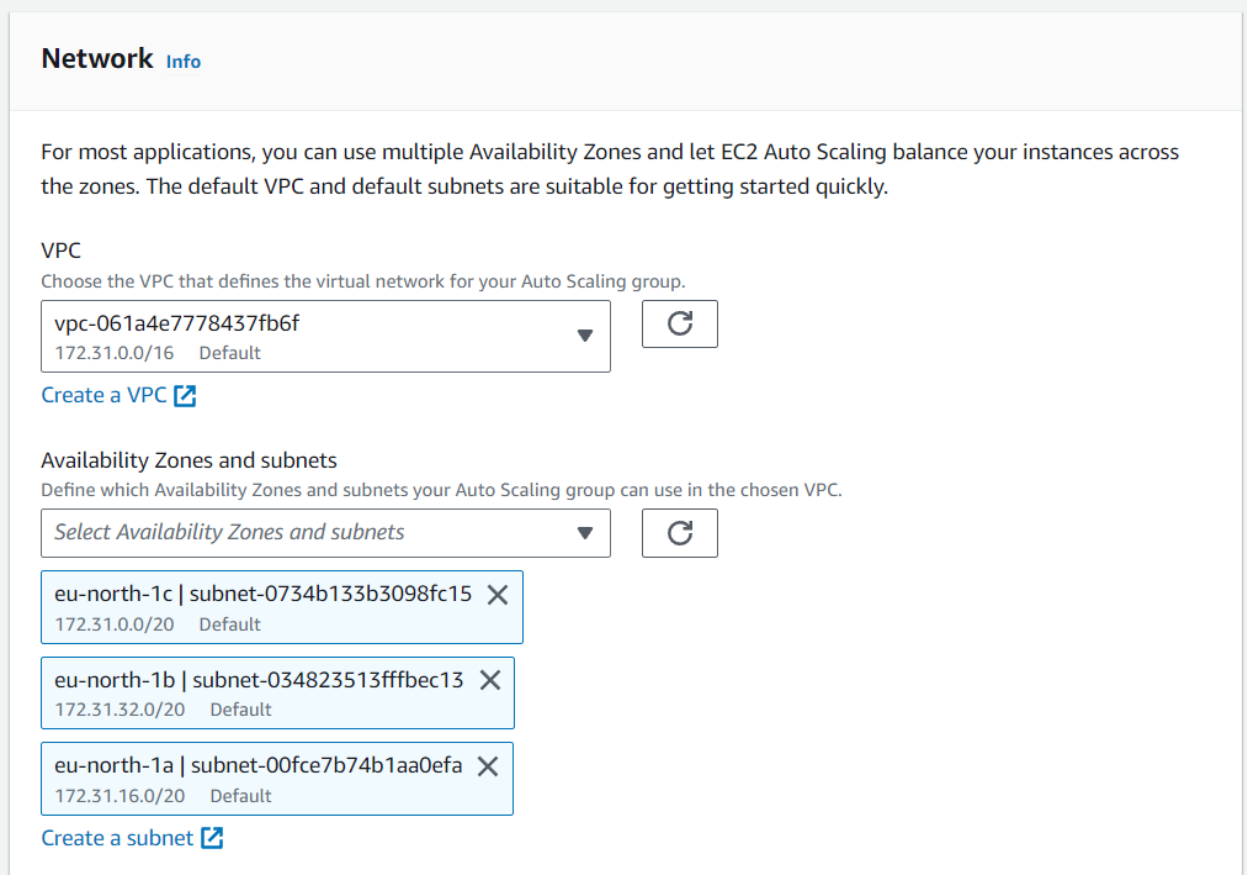


Figure 41 load balancer availability zones

We configure our load balancer for the two types of traffic our servers receive. The web traffic requesting the frontend of the application, and the application traffic requesting the image processing services. The web traffic is forwarded to port 80 and the application traffic is forwarded to port 8000.

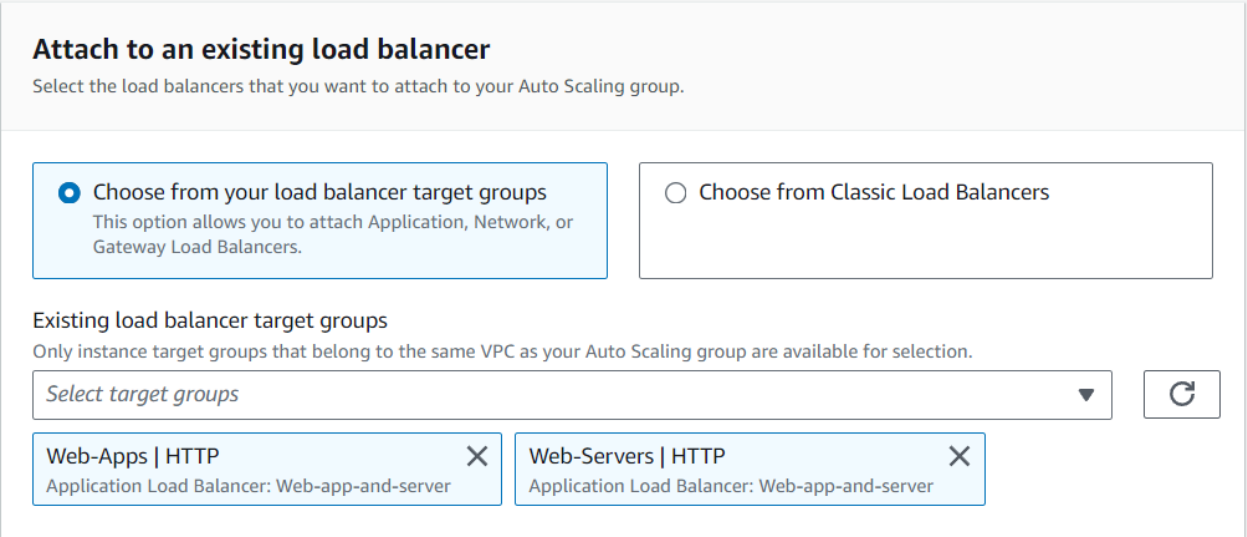


Figure 42: load balancer target groups

Setting up the auto scaling

After setting up the load balancer, we wanted to have a dynamic number of instances running behind it. Instead of having a fixed number of instances running, causing us to pay for the machine in low traffic time and overloading the machines on heavy traffic times, we set up an auto scaling group.

We set the minimum capacity to 1 instance to have at least one machine running all the time, the desired capacity is set to 3 to provide high availability, and the maximum capacity is set to 10.

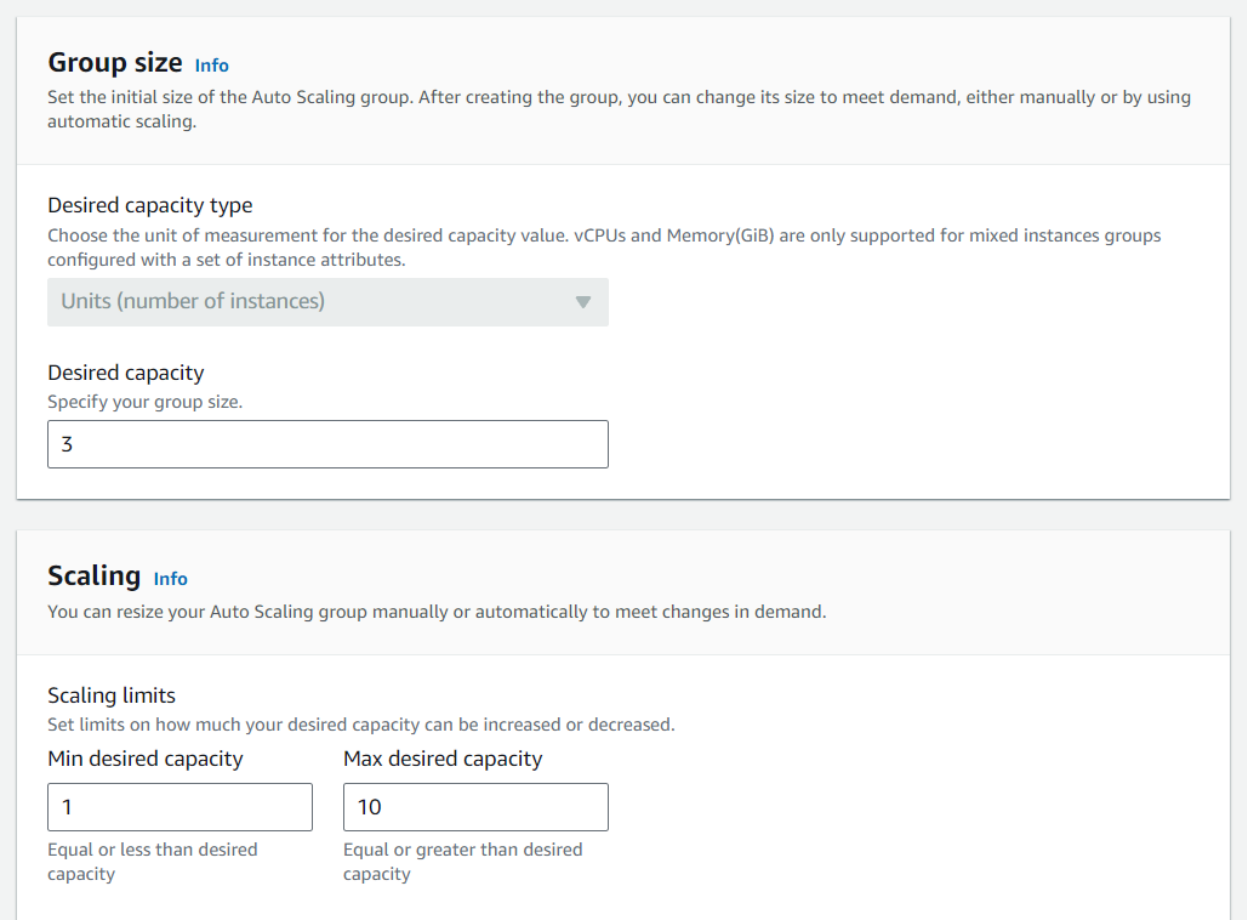


Figure 43 auto scaling capacity configurations

We set up the auto scaling criteria according to the CPU utilization of the machine. We set the target CPU utilization of the machines to be 70% to ensure that the machines are efficiently utilized. If the average utilization exceeds 70% a new instance will be launched automatically from the image we created.

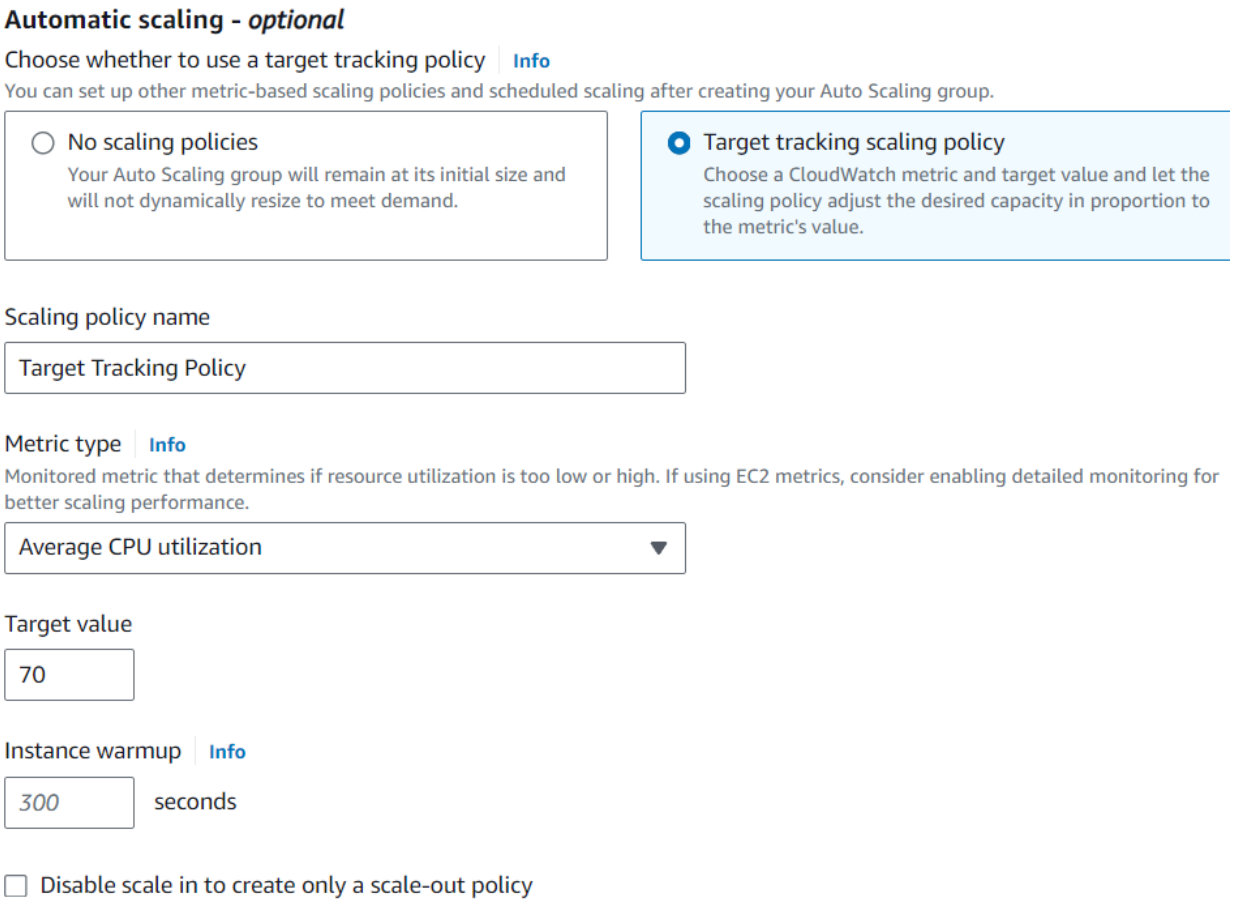


Figure 44 scaling criteria configuration

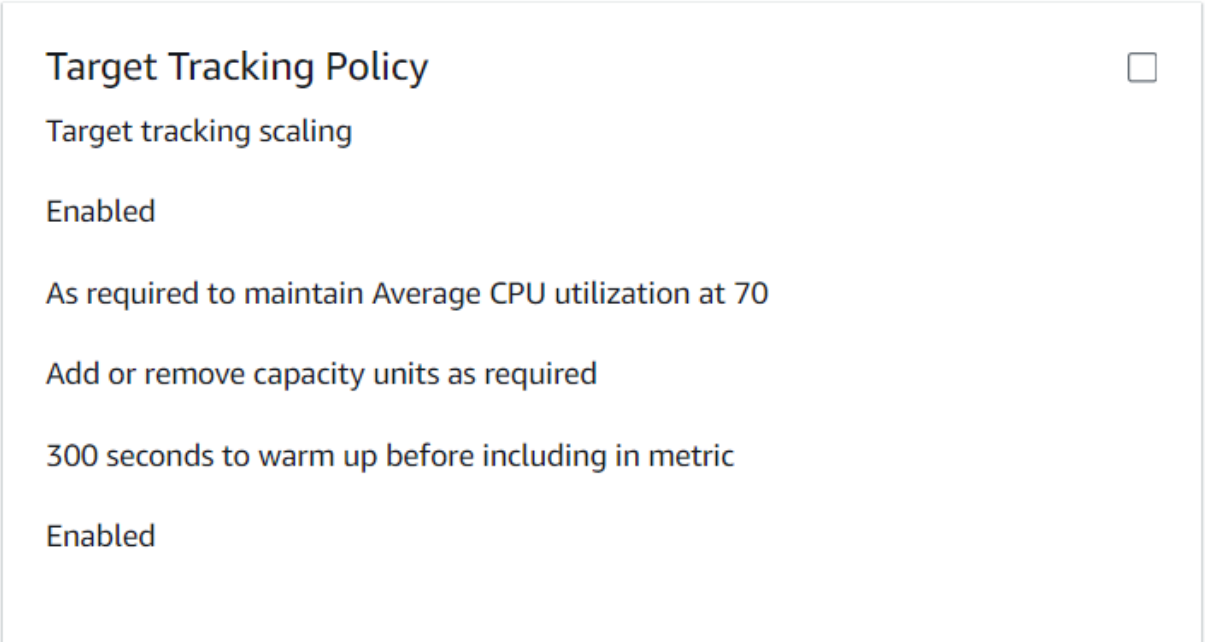


Figure 45 auto scaling summary

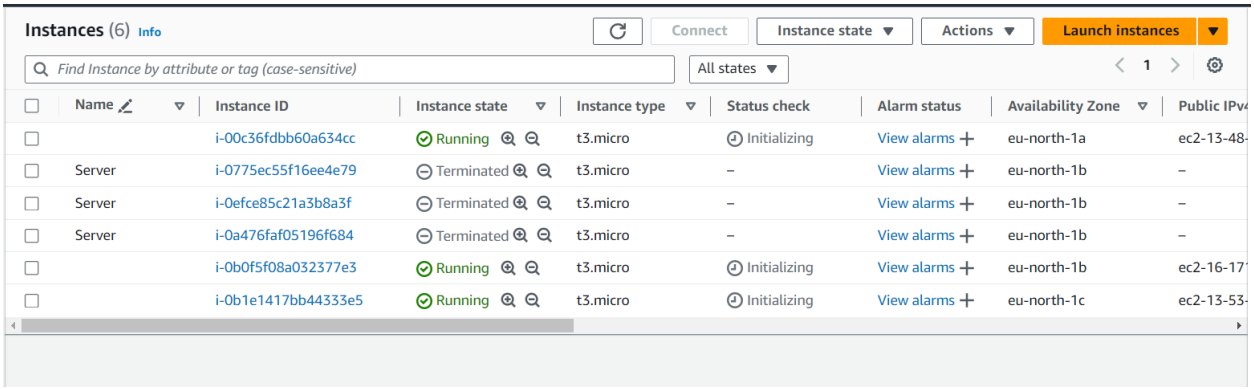


Figure 46 three auto scaling instances running

Modifying the middleware

We also implemented fault tolerance from our client-side middleware. Each request contains one image and its associated operations. When multiple images are requested, multiple requests are sent to the load balancer and the load balancer distributes these requests among the running EC2 instances. This means that the user can submit two images and they may not be processed by the same worker instance. Additionally, when a request fails the middleware transparently retries sending this request 3 times before displaying an error message to the user. Here is the implementation of the middleware:

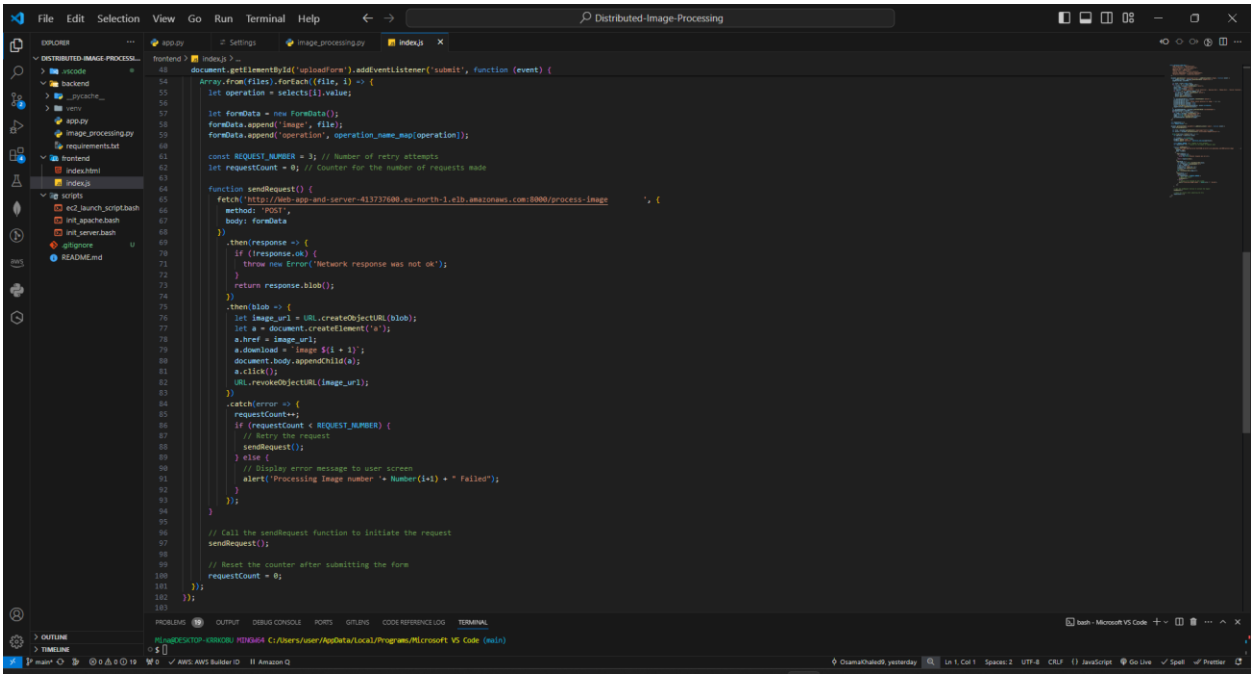


Figure 47 client side middleware, request retrying code snippet

Demo

Notice the URL in the search bar is the URL of the load balancer.

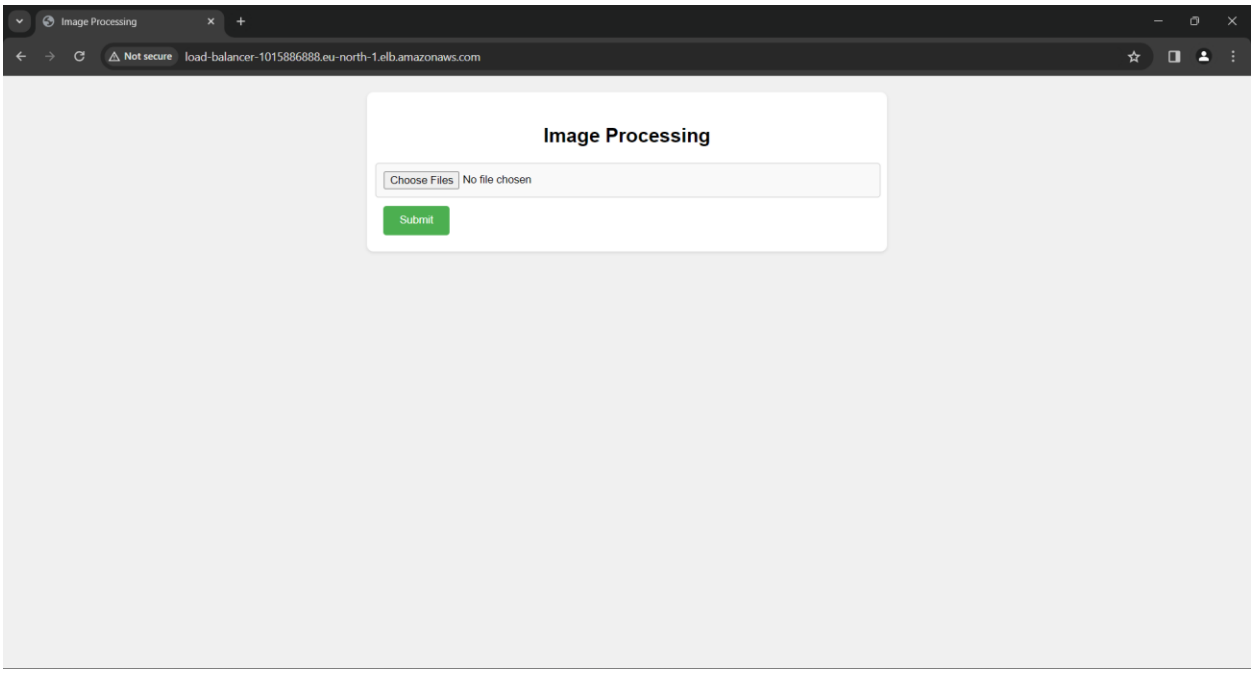


Figure 48 demo, home screen with the load balancer URL

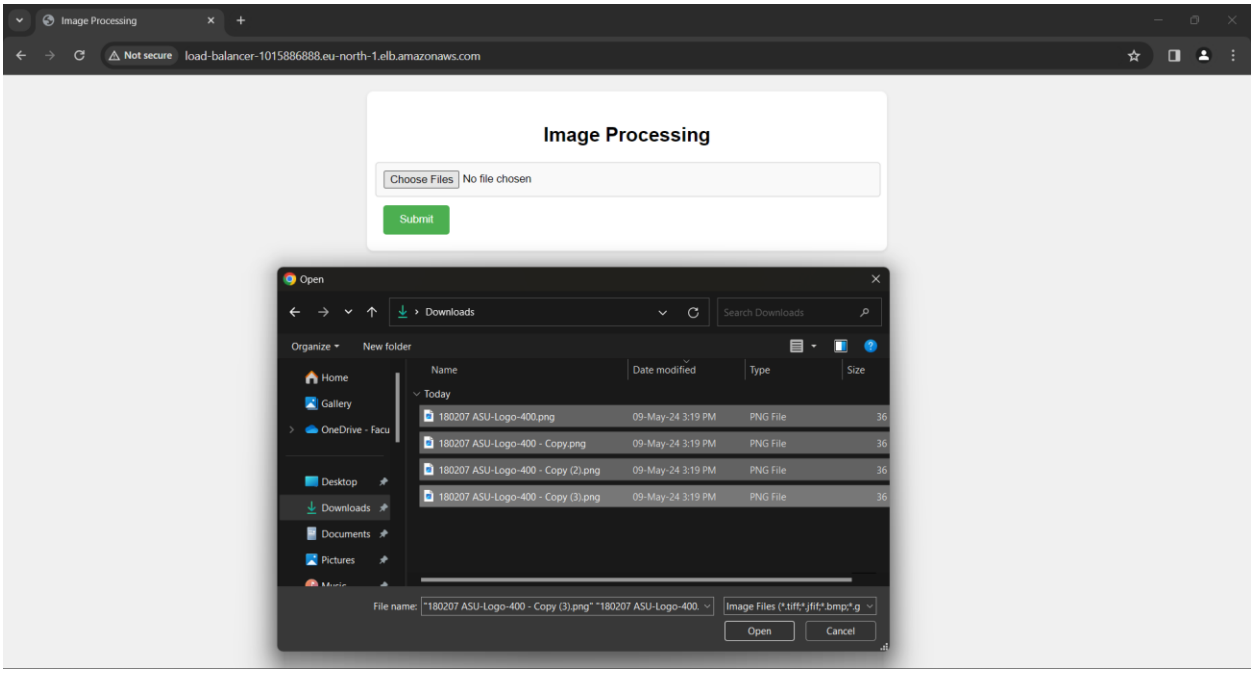


Figure 49: demo, uploading multiple images

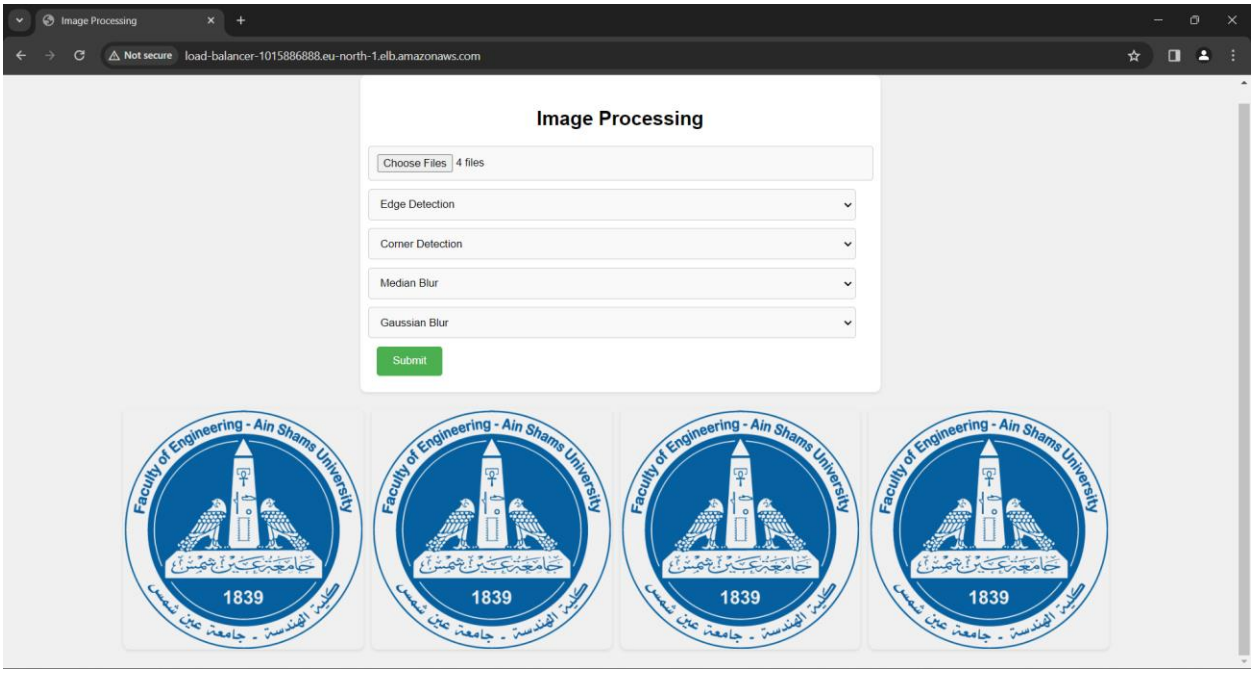


Figure 50: demo, different operation for each image

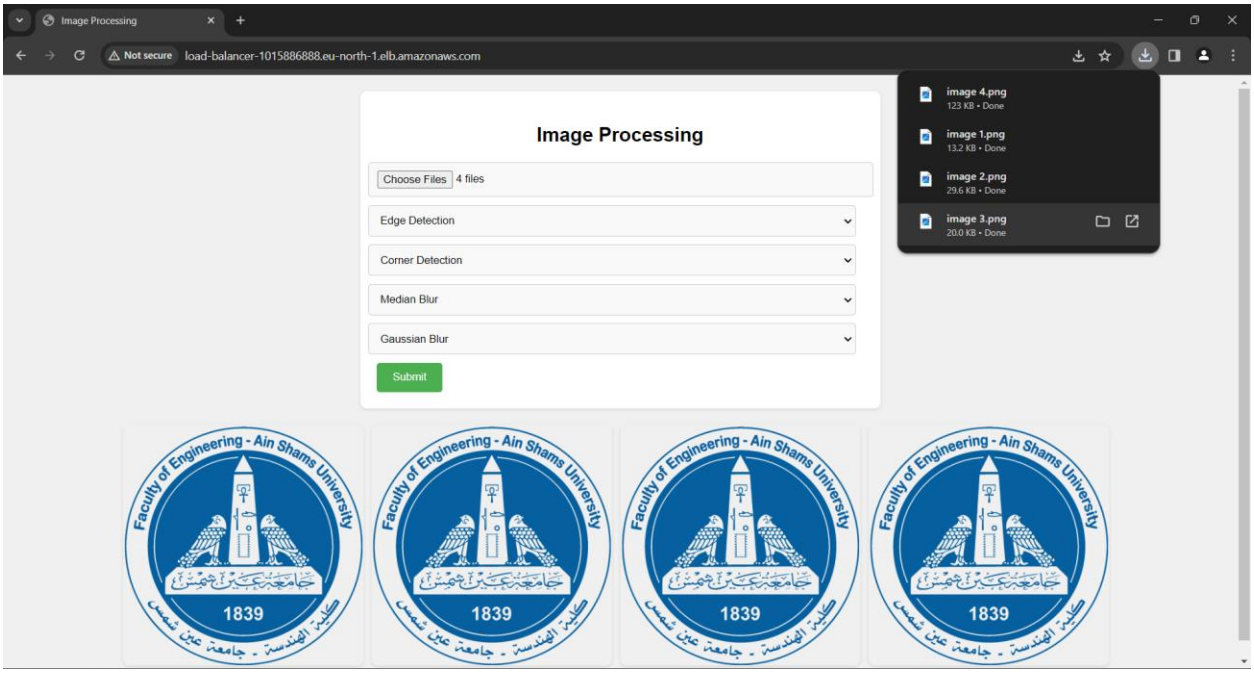


Figure 51: demo, image processing results downloaded

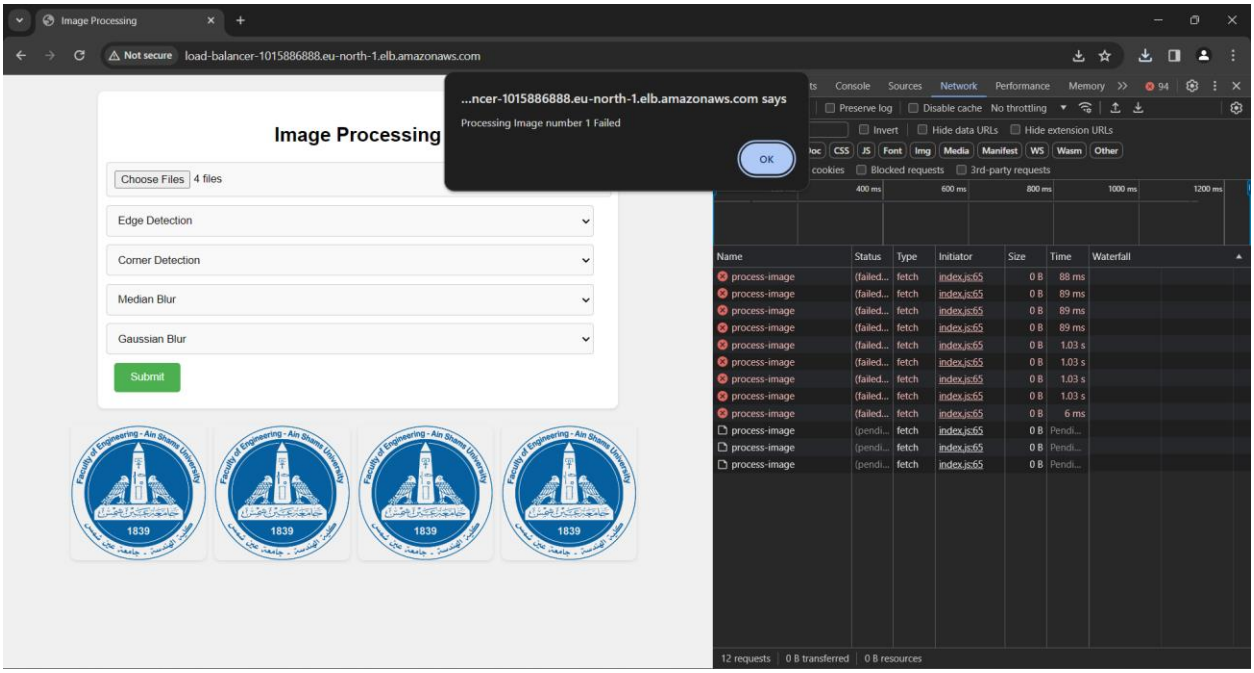


Figure 52 demo, fault tolerance mechanism

In the last screenshot we demonstrate the request retrying mechanism. We set the application to send requests to the wrong URL to simulate a failed request. As can be seen the network tab in the browser developer tool, the application sent 3 requests for each image hoping that the request would be forwarded to a running machine.

PHASE 4

Additional functionalities

We added the functionality for the user to apply all the operations on the image. If the user selects this option, the client-side middleware creates a request for each operation to distribute the operations among the running instances.

```
        Array.from(files).forEach((file, i) => {
            let operation = selects[i].value;

            if (operation === "All Operations") {
                Object.keys(operation_name_map).forEach((key) => {
                    if (key !== "Background Removal") {
                        sendRequest(file, operation_name_map[key], i);
                    }
                });
            } else {
                sendRequest(file, operation_name_map[operation], i);
            }
        });
});
```

There is a tradeoff here between the data transfer and performance. Making a request for each image means that the image is sent 8 times at the cost of distributing the work. We exclude the background removal operation from this feature.

API Testing

To simulate real world Traffic on our application we used API testing through postman.

Fixed load testing

We started by simulating a sustained load on 20 virtual users for five minutes through post man. Note that in the five minutes of the simulations, 1,167 requests were sent which realistically is much more than 20 users. Check the full reports at the end of the document

Test setup

Virtual users	Start time	Load profile
20 VU	May 17, 14:17:45 (GMT+2)	Fixed
Duration	**End time**	**Environment**
5 minutes	May 17, 14:22:54 (GMT+2)	-

1. Summary

Total requests sent	Throughput	Average response time	Error rate
1,167	3.78 requests/second	4,049 ms	1.71 %

1.1 Response time

Response time trends during the test duration.

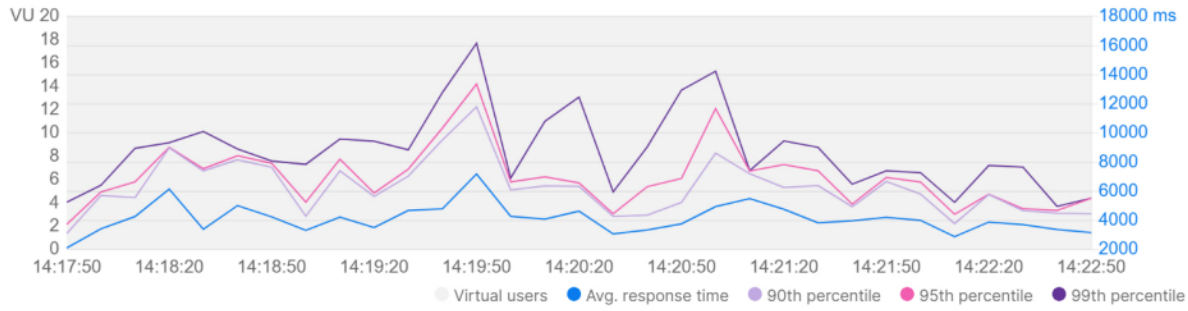


Figure 53: Fixed load with 20 virutal users results

We did the same with 50 virtual users for 10 minuts to further challenge the system

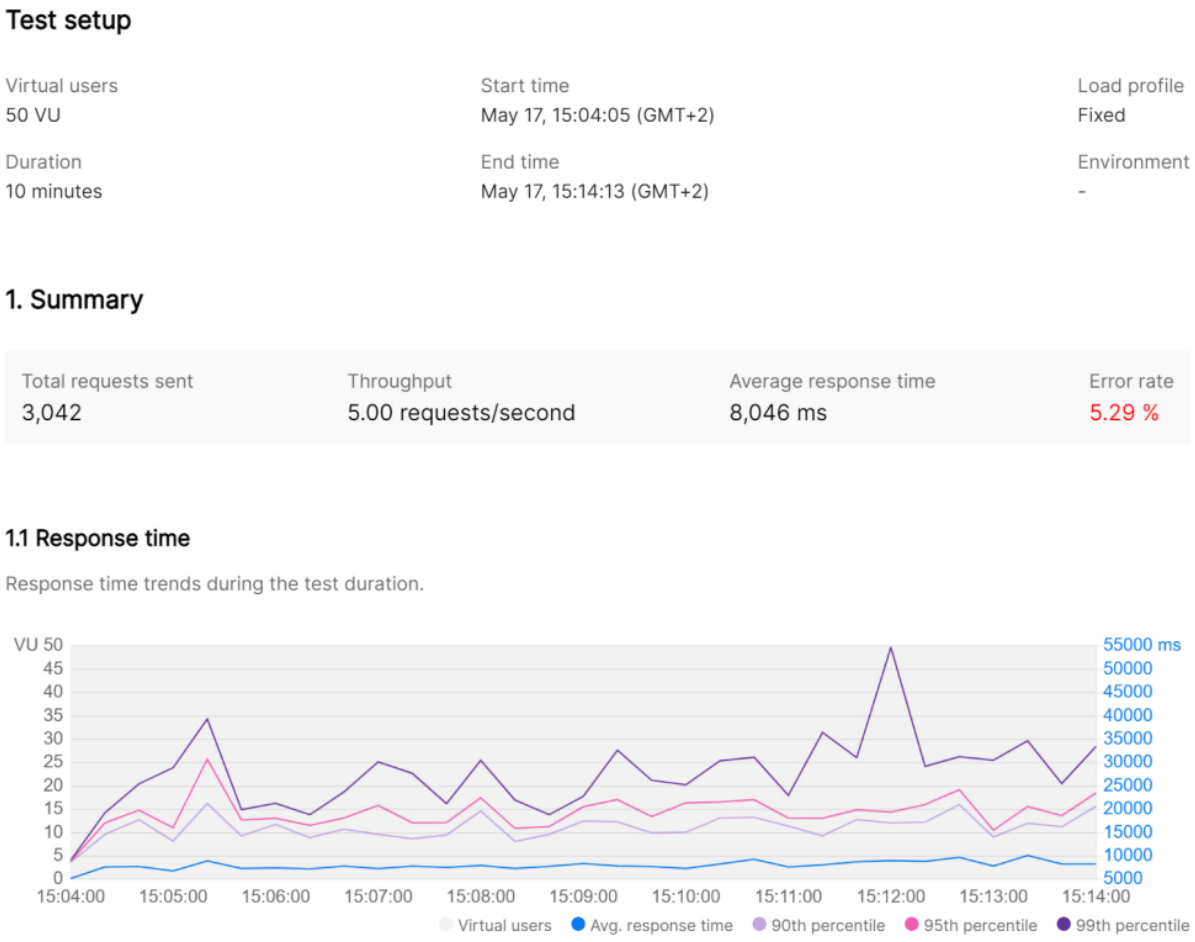


Figure 54: Fixed load with 50 virtual users results

Observe that the average response time is less than 10s which we specified in the non-functional requirements section. Also note that the 5.29% percent error rate can be neglected because the user retries 3 times if the request fails. The resulting error rate for an image of a client is $(0.0529)^3$.

Ramp-up load testing

We also applied ramp-up load testing where the number of virtual users starts at 10 and goes up to 100 in 10 minutes.

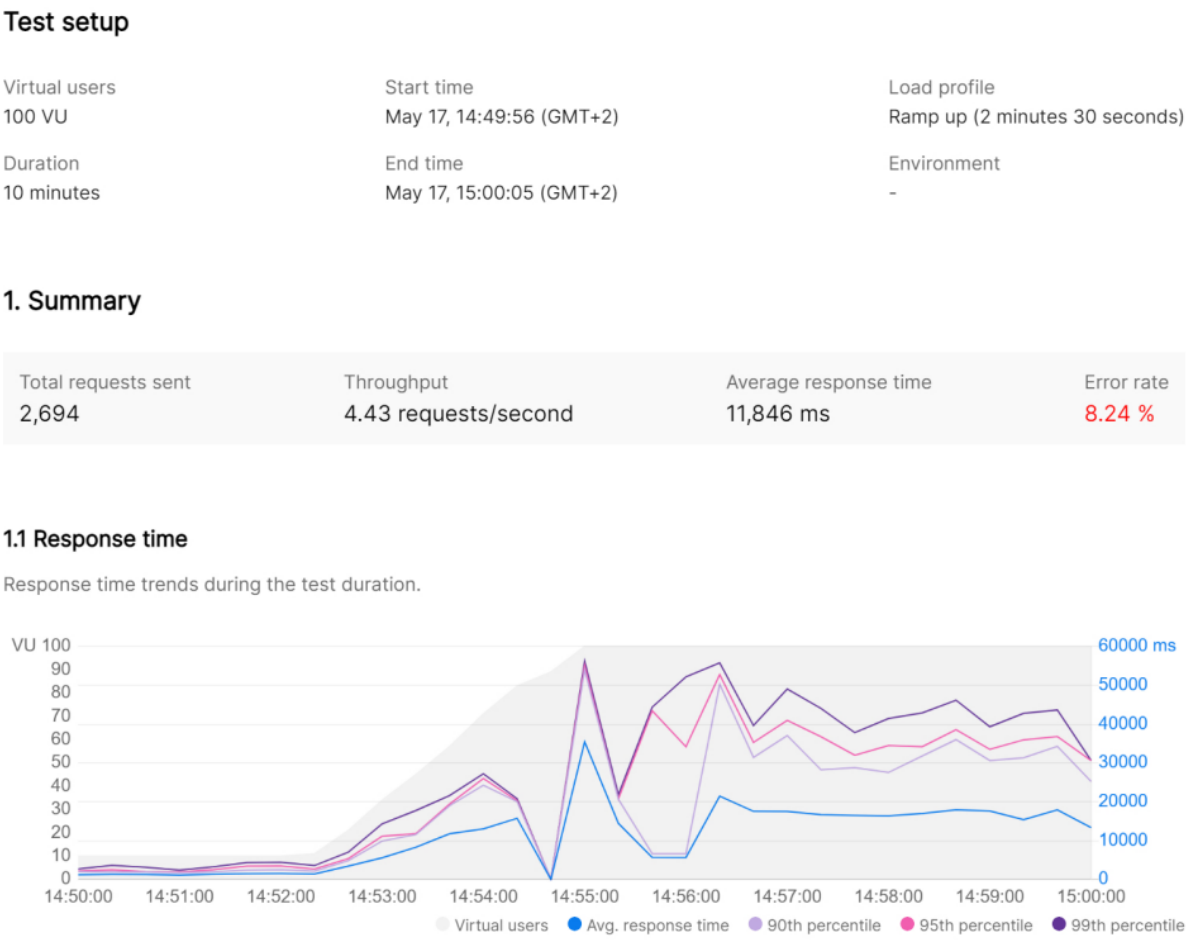


Figure 55: Ramp up testing results

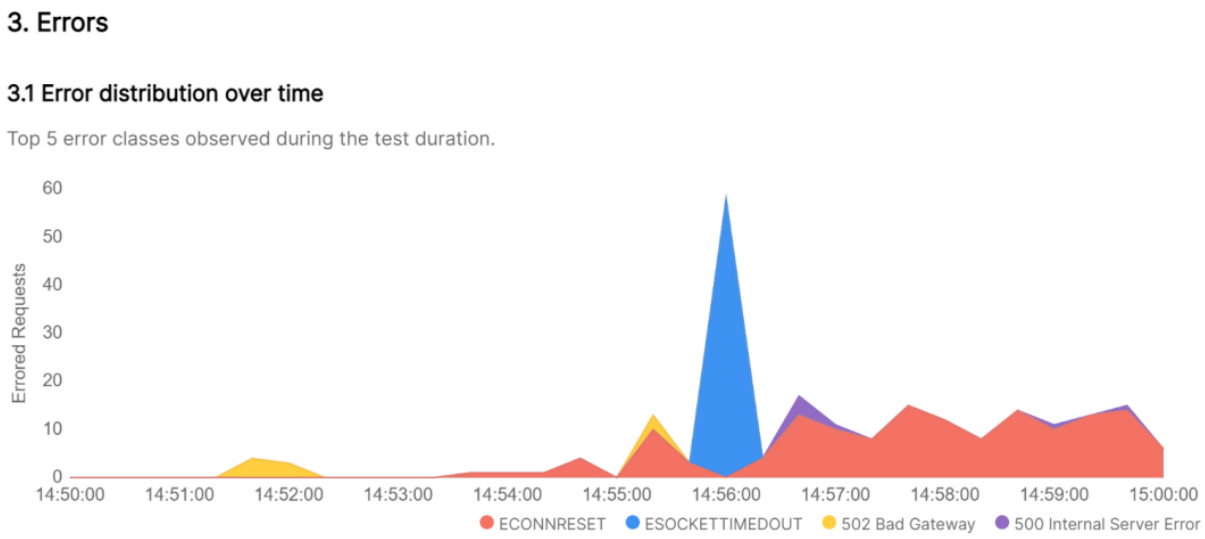


Figure 56 Ramp up testing error distribution

Observe that there is a spike in the errors afterwards, the number of errors goes down again. At that point the number of EC2 instances running was not sufficient. Afterwards, more instances were launched restoring the performance of the system. The system scaled out from 3 instances to 25 instances.

From these results we conclude that the system has the capacity to serve **5 requests/second** and still stay within the accepted response time. This capacity can easily be increased by increasing the maximum capacity of the auto scaling group. We set the maximum capacity to 30 to limit the incurred charges.

GUI Testing

In this testing phase we simulate multiple users using the application through web interface. This phase was relatively peaceful because we could not open as much tabs as we would have liked to simulate due to the limitations of the device. We used Selenium Testing, which basically applies automatic testing to our frontend. We have used unit test library to apply our testing. We used chrome driver, we created an image folder path that contains images to be uploaded to the webpage and tested.
Image files is a list that contains image files in the specified folder.

Also in the image below num_tabs specifies number of tabs to be opened and tested

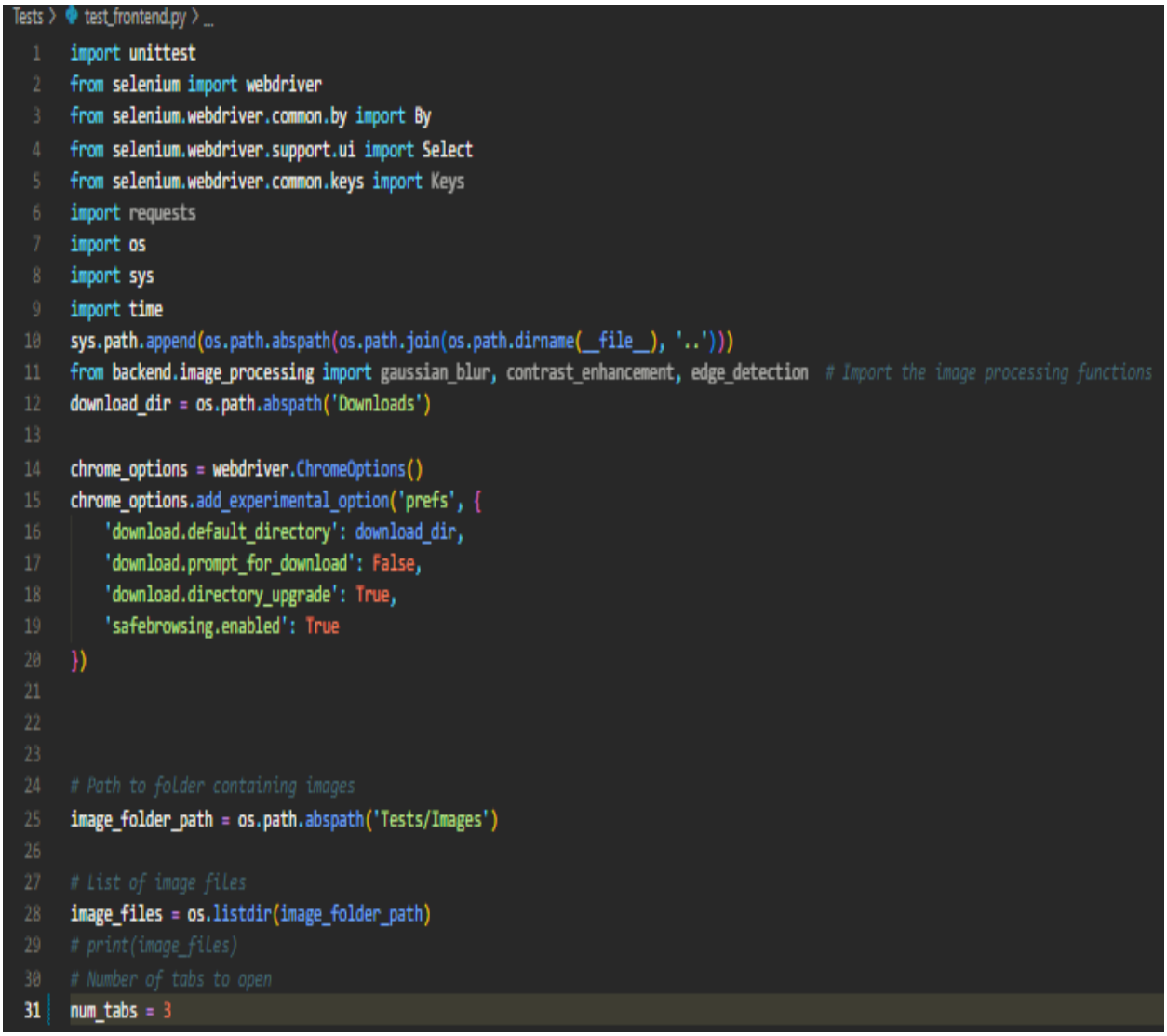


Figure 57 Shows library imports

The figure belows shows the function that uploads images to the webpage.

First we had to find the element of type file (where images are uploaded) and then we had to find the select element to choose the operation to be made on the image.
We found the select element using xpath. By using the imported select library from selenium. We can choose any operation which would be applied automatically to our image

```
from selenium.webdriver.support.ui import Select
```

Figure 58 shows select library

Before selenium clicks the submit button we had to wait for 5 seconds to make sure that instructions made are completely processed.

Now we want to test that images are downloaded in the specified path.But we had to make a delay to make sure that the Assert instruction is not made before the image gets downlaoded. So we made a while loop that iterates until maximum delay is reached (60 seconds) or “image 1” is downloaded in the specified path

Then Assertion is made to check if image exists (has been downloaded) in the specified path

```python
class TestImageUpload(unittest.TestCase):

    def upload_image(self, driver, image_path):
        file_input = driver.find_element(By.XPATH, "//input[@type='file']")
        file_input.send_keys(image_path)
        select_element = driver.find_element(
            By.XPATH, "//div[@id='imageFilters']//select")
        select = Select(select_element)
        select.select_by_visible_text('Edge Detection')
        # print("Hello")
        # Wait for the option to be selected (add a delay if necessary)
        driver.implicitly_wait(5)

    # Find the submit button element
        submit_button = driver.find_element(
            By.XPATH, "//button[@type='submit']")
        # Click the submit button
        submit_button.click()

        # Wait for the image to be downloaded
        max_wait_time = 60  # Maximum wait time in seconds
        polling_interval = 1  # Polling interval in seconds
        image_filename = "image 1.png"
        image_path = "./Tests/Images/image1.jpg"
        wait_time = 0

        while not os.path.exists(image_path):
            time.sleep(polling_interval)
            wait_time += polling_interval

            if wait_time >= max_wait_time:
                self.fail(f"Image download took too long. Expected file '{
                          image_filename}' was not found.")

        # Assertion: Check if the file exists
        self.assertTrue(os.path.exists(image_path),
                        f"Processed image not found: {image_path}")


if __name__ == "__main__":
    unittest.main()
```

Figure 59 shows function of uploading an image

The figure below shows Test class of our project

Chrome driver is used (as discussed before). Multiple tabs are opened and then each tab is navigated to the file path (form page) then upload image function is called (discussed in the previous page)

```
31  class TestImageUpload(unittest.TestCase):
33      def setUp(self):
34          self.driver = webdriver.Chrome()
35
36      def tearDown(self):
37          self.driver.quit()
38
39      def test_upload_images(self):
40          driver = self.driver
41
42          # Open multiple tabs
43          for i in range(num_tabs):
44              if i > 0:
45                  driver.execute_script("window.open('');")
46                  driver.switch_to.window(driver.window_handles[i])
47                  # print("hi")
48
49              # Navigate to the form page
50              driver.get('file://' + os.path.abspath('./frontend/index.html'))
51
52              # Upload images
53              for image in image_files:
54                  self.upload_image(driver, os.path.join(
55                      image_folder_path, image))
```

Figure 60 shows test class

The figure shows a successful test that run successfully in 12 seconds.

Response from each submission is 200 (indicating a successful operation)

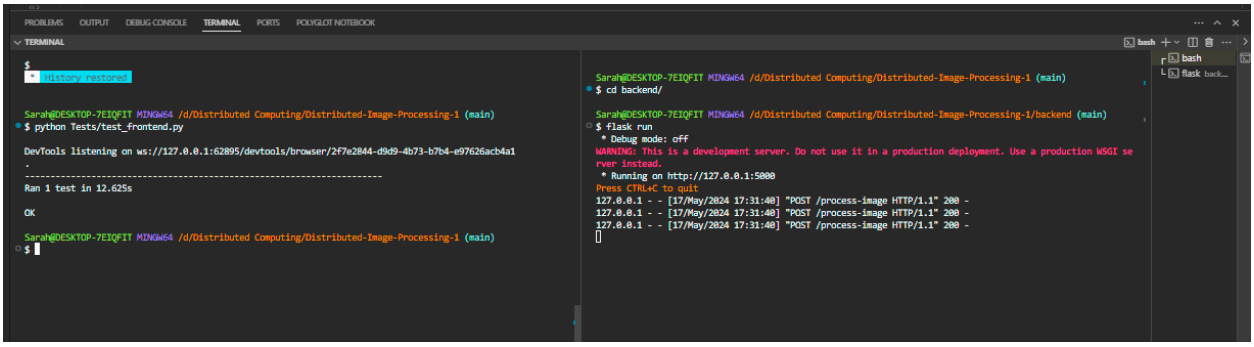


Figure 61 shows successful test

Demo

https://drive.google.com/file/d/15-1gsSUfRz_P8EkznLFlMcaZfxNMfjrw/view?usp=sharing

GitHub Repo

Distributed-Image-Processing

Answering all the phases questions

Phase 1

Q: What is the main objective of our project?

A: To design a **distributed backend** with a **web application frontend**. The user of the application can upload a photo and select the operation he/she wants to apply. The photo will be sent to our distributed backend and the processed image will be **transparently** sent back to the user as if all operations and computations were performed locally on the user's device.

Q: Which technologies have we decided to use and why?

A: For the image processing tasks, we use OpenCV for its large suite of image processing operations. We use HTML, CSS, and JS for the web interface. For the web server, we use python Flask for its simplicity. We chose AWS as our cloud provider.

Q: What are the key components of our system architecture?

A: Our logical architecture is a simple client server model where the client sends his image processing requests through a web interface to the distributed backend servers. Our cloud architecture consists of an autoscaling group of EC2 instances behind a load balancer. All the incoming traffic is sent to the load balancer which distributes the traffic among the servers.

Phase 2

Q: How does our worker thread process tasks?

A: We use a Gunicorn server in the backend which is a multithreaded server. We launch Gunicorn with two worker processes each of which is multithreaded and can handle multiple requests. We only launch two processes because the EC2 instance only has one

CPU. When we tried to launch more, the CPU could not handle the work. Whenever a request is received, a new thread is forked to serve the request.

Q: What basic image processing operations have we implemented?

A: We implemented Gaussian blur, median blur, and color inversion, all of which were implemented using OpenCV.

Q: How have we set up our cloud environment?

A: We have set up a single EC2 instance, which contains both the web server (Apache) and the application server (WSGI). On boot, the EC2 instance downloads the code from an S3 bucket and runs it, effectively running the server

Phase 3

Q: What advanced image processing operations have we implemented?

A: Please refer to the start of the document. We have implemented Fourier transform, edge detection, corner detection, and background removal. All the implementations and the results are documented at the start of this document.

Q: How does our system handle distributed processing?

A: We have multiple EC2 instances running, and the images processing can be done across these multiple instances. The work is evenly distributed across all the instances through the load balancer.

Q: How have we implemented scalability and fault tolerance?

A: Our setup implements scalability and fault tolerance in the following manner. The number of running EC2 instances scales up automatically if there is heavy traffic and scales down if the traffic is not heavy. Paired with the load balancer, an arbitrary number of users can access the system and the system will scale appropriately.

If one of the instances fails, it will automatically be rebooted, and the load balancer will route the requests to healthy running instances. Additionally, the middleware would resend the requests if a faulty response is received or if no response at all was received thus fault tolerance is implemented.

Phase 4

Q: What were the results of our system testing?

Our system can handle 5 requests/second and this number can be easily scaled by scaling the maximum number of allowed EC2 instances running. In the tests we performed 25 instances were running.

Q: What information is included in our system documentation?
A: All our system functionalities, deployment steps, and test results are documented in this report.

Q: How did we deploy our system to the cloud?

The system was deployed to AWS using the following services:

- Amazon Elastic Cloud Compute (EC2)
- Amazon Elastic Load Balancer (ELB)
- Amazon Machine Image (AMI)
- Amazon Simple Storage Service (S3)
- Amazon Auto Scaling
- Amazon Virtual Private Cloud (VPC)

Team Roles

Cloud setup: All team members

Image processing: Sarah Sherif

Flask Server: Mina Shawket

GUI: Matthew Sherif

Middleware: Matthew Sherif and Osama Khaled

API Testing: Mina Shawket and Ossama Khaled

GUI Testing: Sarah Sherif and Matthew Sherif

Documentation: All team members

API Testing Reports

Performance test report - May 17, 2024 (#9)

Open in Postman

Postman collection: New Collection

Report exported on: May 17, 2024, 14:23:20 (GMT+2)

Test setup

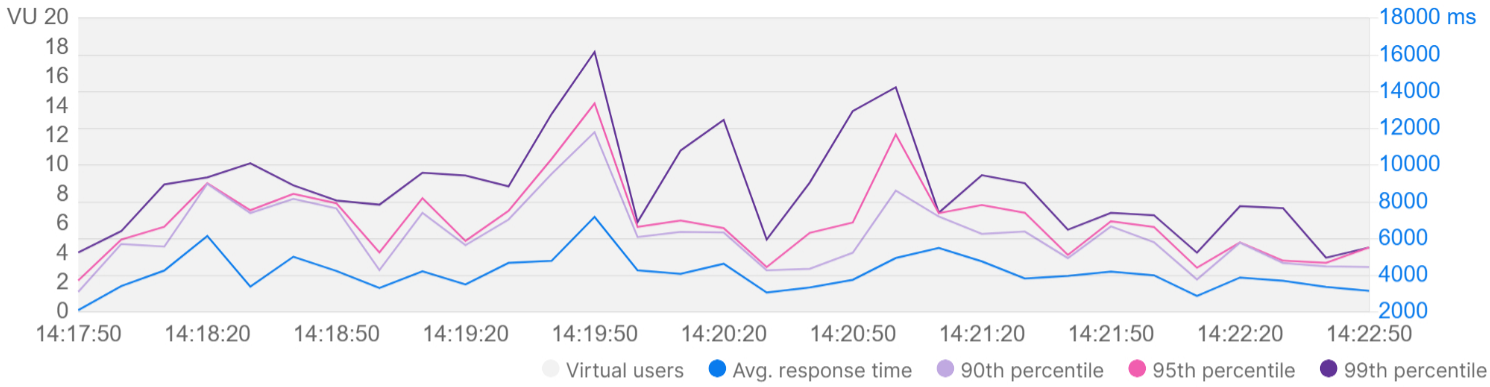
Virtual users	Start time	Load profile
20 VU	May 17, 14:17:45 (GMT+2)	Fixed
Duration	**End time**	**Environment**
5 minutes	May 17, 14:22:54 (GMT+2)	-

1. Summary

Total requests sent	Throughput	Average response time	Error rate
1,167	3.78 requests/second	4,049 ms	1.71 %

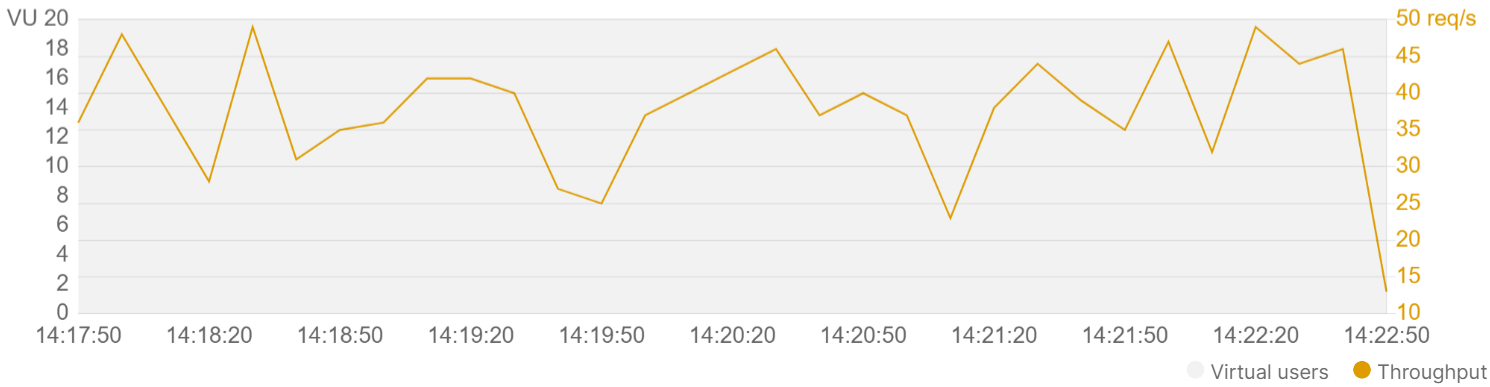
1.1 Response time

Response time trends during the test duration.



1.2 Throughput

Rate of requests sent per second during the test duration.



1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	4,049	6,433	7,777	10,476	93	16,169

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	20	502 Bad Gateway (20)	-	0

2. Metrics for each request

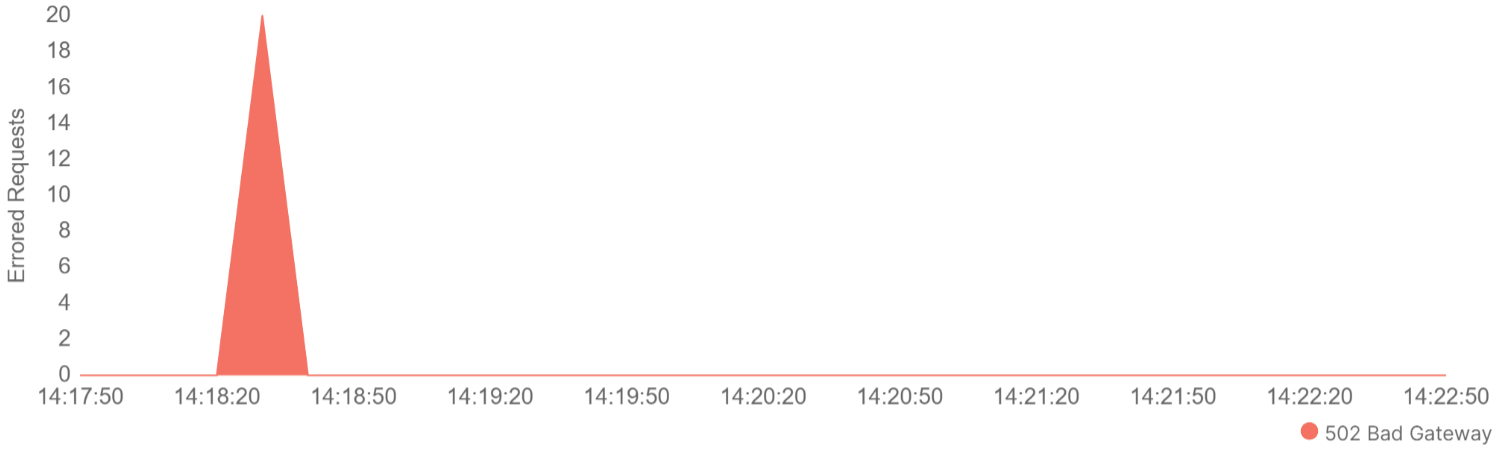
The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	1,167	3.78	93	4,049	6,433	16,169	1.71

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.



3.2 Error distribution for requests

Errored requests grouped by error class, along with the error count for each class.

Error class	Total counts
502 Bad Gateway	20
POST New Request	20

Testing API performance on Postman

Postman enables you to simulate user traffic and observe how your API behaves under load. It also helps you identify any issues or bottlenecks that affect performance.

Learn more about testing API performance.

Performance test report - May 17, 2024 (#14)

Open in Postman

Postman collection: New Collection

Report exported on: May 17, 2024, 15:15:16 (GMT+2)

Test setup

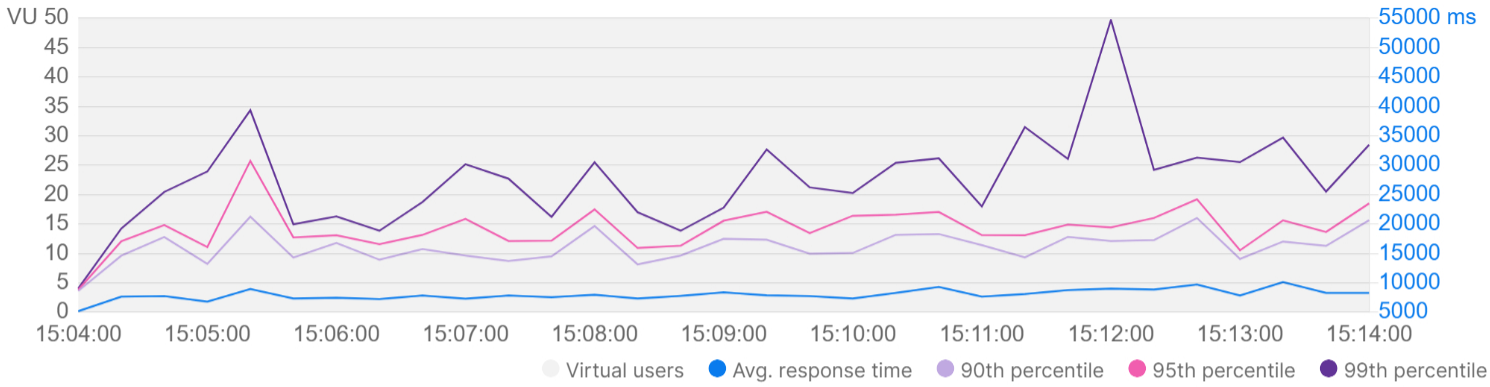
Virtual users	Start time	Load profile
50 VU	May 17, 15:04:05 (GMT+2)	Fixed
Duration	**End time**	**Environment**
10 minutes	May 17, 15:14:13 (GMT+2)	-

1. Summary

Total requests sent	Throughput	Average response time	Error rate
3,042	5.00 requests/second	8,046 ms	5.29 %

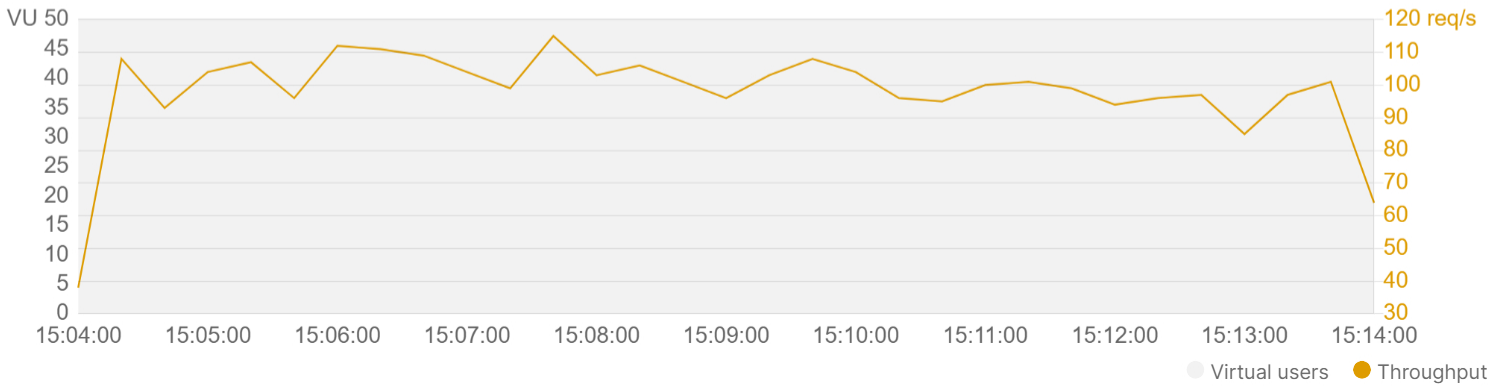
1.1 Response time

Response time trends during the test duration.



1.2 Throughput

Rate of requests sent per second during the test duration.



1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	8,046	15,925	19,439	28,875	124	54,715

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	161	ECONNRESET (130)	502 Bad Gateway (19)	2

2. Metrics for each request

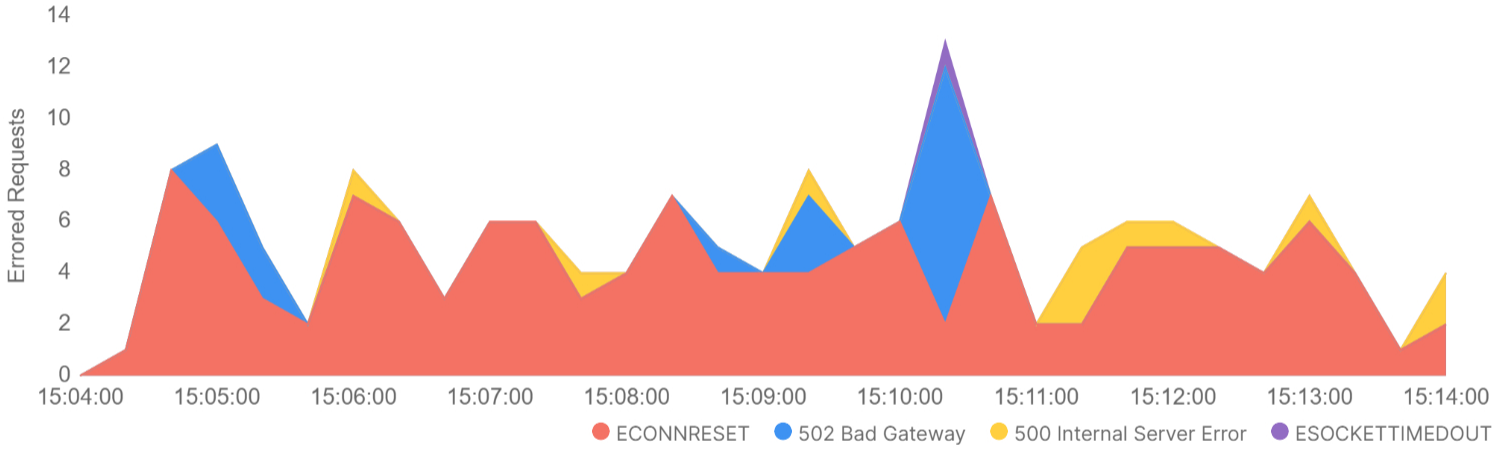
The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	3,042	5.00	124	8,046	15,925	54,715	5.29

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.



3.2 Error distribution for requests

Errored requests grouped by error class, along with the error count for each class.

Error class	Total counts
ECONNRESET	130
POST New Request	130
502 Bad Gateway	19
POST New Request	19
500 Internal Server Error	11
POST New Request	11
ESOCKETTIMEDOUT	1
POST New Request	1

Testing API performance on Postman

Postman enables you to simulate user traffic and observe how your API behaves under load. It also helps you identify any issues or bottlenecks that affect performance.

Learn more about testing API performance.

Performance test report - May 17, 2024 (#13)

Open in Postman

Postman collection: New Collection

Report exported on: May 17, 2024, 15:00:31 (GMT+2)

Test setup

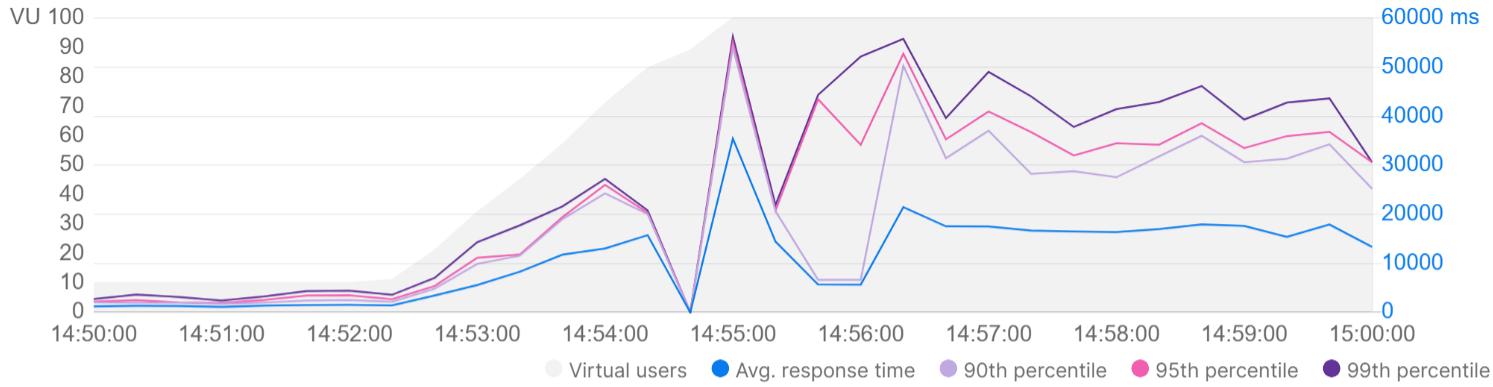
Virtual users	Start time	Load profile
100 VU	May 17, 14:49:56 (GMT+2)	Ramp up (2 minutes 30 seconds)
Duration	**End time**	**Environment**
10 minutes	May 17, 15:00:05 (GMT+2)	-

1. Summary

Total requests sent	Throughput	Average response time	Error rate
2,694	4.43 requests/second	11,846 ms	8.24 %

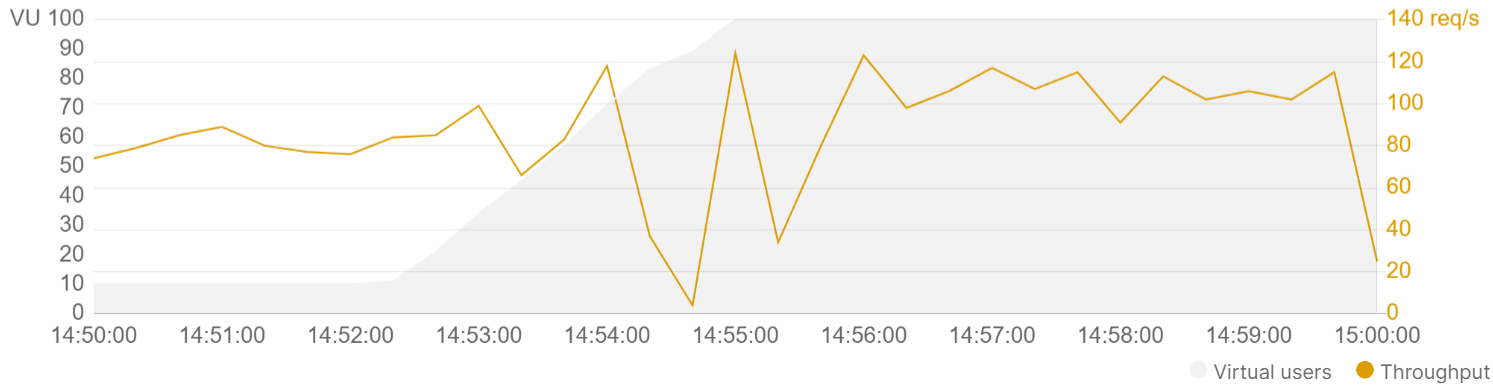
1.1 Response time

Response time trends during the test duration.



1.2 Throughput

Rate of requests sent per second during the test duration.



1.3 Requests with slowest response times

Top 5 slowest requests based on their average response times.

Request	Resp. time (Avg ms)	90th (ms)	95th (ms)	99th (ms)	Min (ms)	Max (ms)
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	11,846	29,966	38,605	52,576	81	57,142

1.4 Requests with most errors

Top 5 requests with the most errors, along with the most frequently occurring errors for each request.

Request	Total error count	Error 1	Error 2	Other errors
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	222	502 Bad Gateway (10)	ECONNRESET (147)	2

2. Metrics for each request

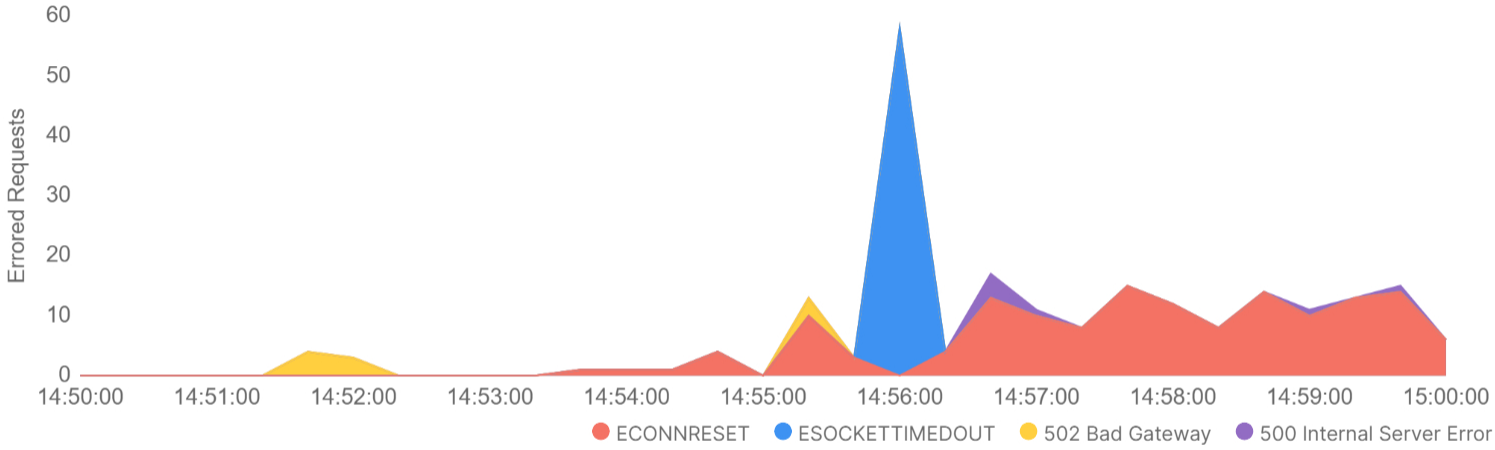
The requests are shown in the order they were sent by virtual users.

Request	Total requests	Requests/s	Min (ms)	Avg (ms)	90th (ms)	Max (ms)	Error %
POST New Request application-and-web-server-LB-570742999.eu-north-1.elb.amazonaws.com:8000/process-image	2,694	4.43	81	11,846	29,966	57,142	8.24

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.



3.2 Error distribution for requests

Errored requests grouped by error class, along with the error count for each class.

Error class	Total counts
ECONNRESET	147
POST New Request	147
ESOCKETTIMEDOUT	58
POST New Request	58
502 Bad Gateway	10
POST New Request	10
500 Internal Server Error	7
POST New Request	7

Testing API performance on Postman

Postman enables you to simulate user traffic and observe how your API behaves under load. It also helps you identify any issues or bottlenecks that affect performance.

Learn more about testing API performance.