



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	7
Introduction	8
Requirements	8
Functional requirements	8
Nonfunctional requirements	8
Image processing use cases	9
Technical use cases.....	9
Edge Detection	9
Corner Detection	9
Image filtering.....	9
Fourier Transform.....	9
General-Purpose use cases.....	10
Contrast enhancement.....	10
Background removal	10
Text detection	10
Approach	11
Software architecture	11
Deployment architecture	12
Sequence Diagram	13
Gantt Chart	14
PHASE 2	15
Introduction	16
Objective.....	16
Image Processing Core	17
Image Processing Techniques:	17
Edge Detection:.....	17
Corner Detection:.....	18
Gaussian Blur.....	18

Median Blur.....	19
Fourier Transformation.....	19
Contrast Enhancement.....	20
Background Removal	20
Invert Image	20
Text Detection	21
Frontend	22
JavaScript	24
Restful API.....	28
Cloud Setup	29
PHASE 3.....	32
Introduction	33
Advanced image processing operations.....	34
Gaussian blur.....	34
Median blur.....	35
Fourier transform.....	35
Contrast enhancement.....	36
Corner detection	36
Edge detection	37
Color inversion	37
Background removal.....	38
Creating an image for the EC2 instances.....	38
Setting the Gunicorn daemon.....	39
Setting up the load balancer	40
Setting up the auto scaling.....	42
Modifying the middleware.....	44
Demo.....	46
PHASE 4.....	49
Additional functionalities.....	50

API Testing.....	51
Fixed load testing.....	51
Ramp-up load testing	53
GUI Testing	55
Demo.....	59
GitHub Repo.....	59
Answering all the phases questions.....	59
Phase 1.....	59
Phase 2.....	59
Phase 3.....	61
Phase 4.....	61
Team Roles.....	62
API Testing Reports	63

Figure 1: Software Architecture	11
Figure 2: Component Diagram.....	11
Figure 3: Deployment Diagram	12
Figure 4: Sequence Diagram	13
Figure 5: Gantt Chart	14
Figure 6 shows edge detection operation	17
Figure 7 shows corner detection operation.....	18
Figure 8 shows gaussian blur detection	18
Figure 9 shows median blur detection	19
Figure 10 shows Fourier transform operation.....	19
Figure 11 shows contrast enhancement operation.....	20
Figure 12 shows library for background removal	20
Figure 13 shows background removal operation	20
Figure 14 shows invert color operation.....	20
Figure 15 shows easyocr library.....	21
Figure 16 shows text detection operation	21
Figure 17 shows webpage view.....	22
Figure 18 shows images uploaded by user	23

Figure 19 shows operations user chooses from	24
Figure 20 shows operation name map	24
Figure 21 shows how a user chooses an operation.....	25
Figure 22 shows how image appears to the user	25
Figure 23 preventing default submission	25
Figure 24 list for images and list for operations	26
Figure 25 each request is sent independently.....	26
Figure 26 download image processed if response received successfully	27
Figure 27 shows flask app	28
Figure 28 shows initialization script of the EC2 instance.....	29
Figure 29 shows initialization script of the WSGI server	30
Figure 30 shows initialization script of the Apache web server	30
Figure 31 shows EC2 instances	31
Figure 32 Testing Gaussian blur from postman	34
Figure 33 Testing median blur from postman	35
Figure 34 Testing Fourier transform from postman	35
Figure 35 Testing contrast enhancement from postman	36
Figure 36: Testing corner detection from postman.....	36
Figure 37Testing edge detection from postman.....	37
Figure 38 Testing color inversion from postman	37
Figure 39: old user data script	38
Figure 40 app.service file to setup Gunicorn as a service daemon.....	39
Figure 41 load balancer availability zones	40
Figure 42: load balancer target groups	41
Figure 43 auto scaling capacity configurations	42
Figure 44 scaling criteria configuration.....	43
Figure 45 auto scaling summary	44
Figure 46 three auto scaling instances running	44
Figure 47 client side middleware, request retrying code snippet	45
Figure 48 demo, home screen with the load balancer URL.....	46
Figure 49: demo, uploading multiple images	46
Figure 50: demo, different operation for each image	47
Figure 51: demo, image processing results downloaded	47
Figure 52 demo, fault tolerance mechanism.....	48
Figure 53: Fixed load with 20 virutal users results	51
Figure 54: Fixed load with 50 virtual users results	52
Figure 55: Ramp up testing results.....	53
Figure 56 Ramp up testing error distribution	54

Figure 57 Shows library imports.....	56
Figure 58 shows select library	56
Figure 59 shows function of uploading an image.....	57
Figure 60 shows test class	58
Figure 61 shows successful test.....	58

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.

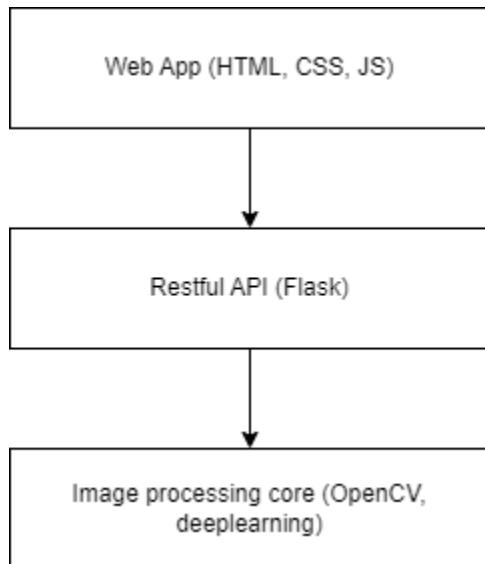


Figure 1: Software Architecture

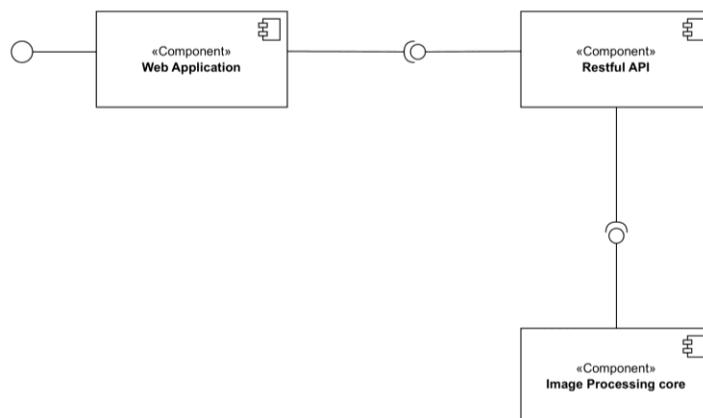


Figure 2: Component Diagram

Deployment architecture

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

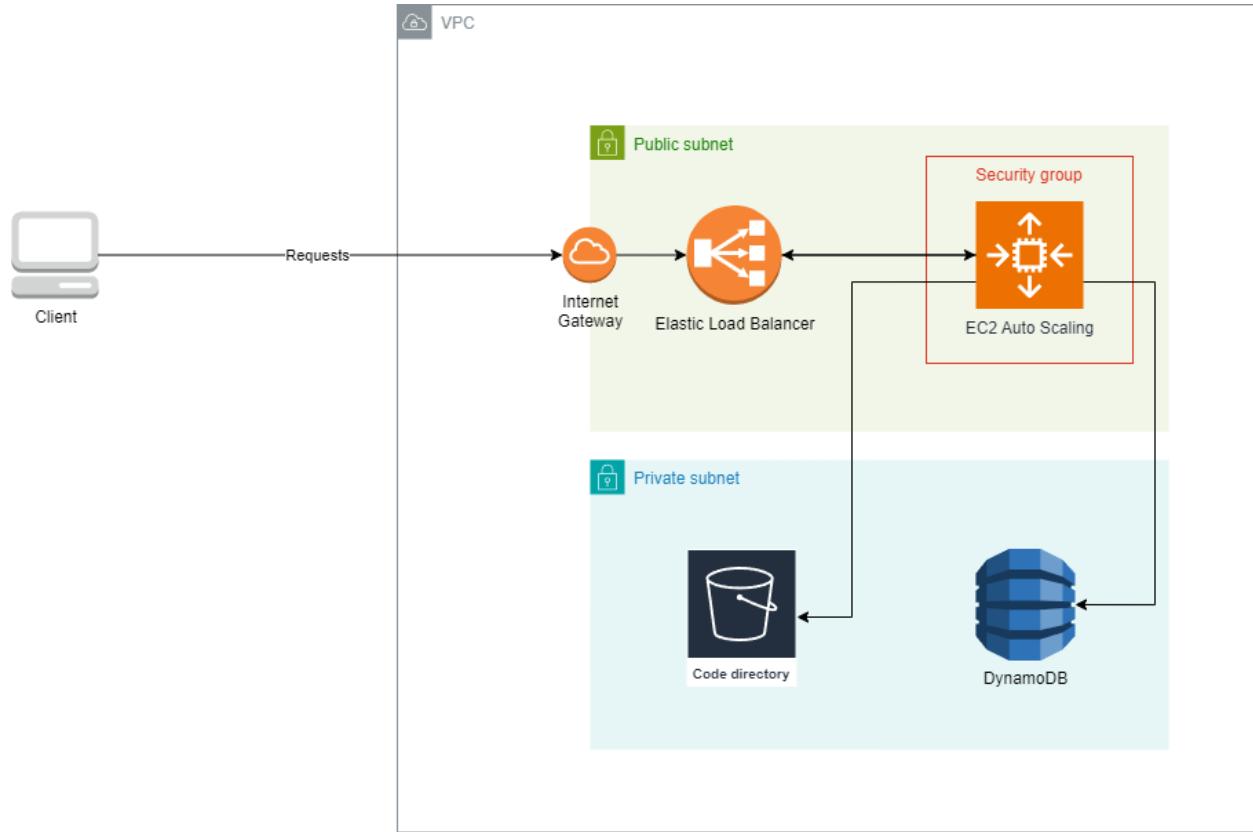


Figure 3: Deployment Diagram

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.

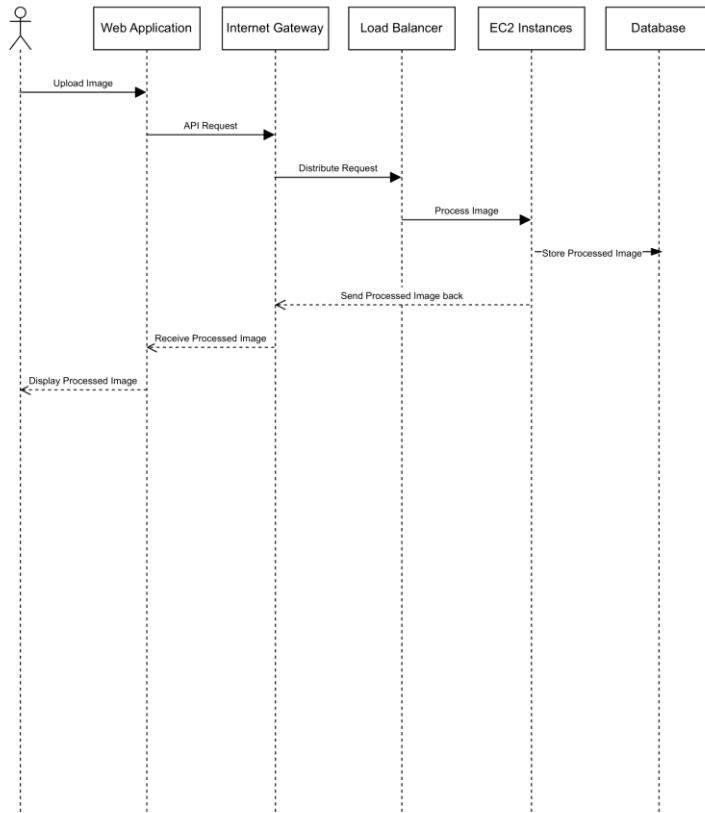


Figure 4: Sequence Diagram

Gantt Chart

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

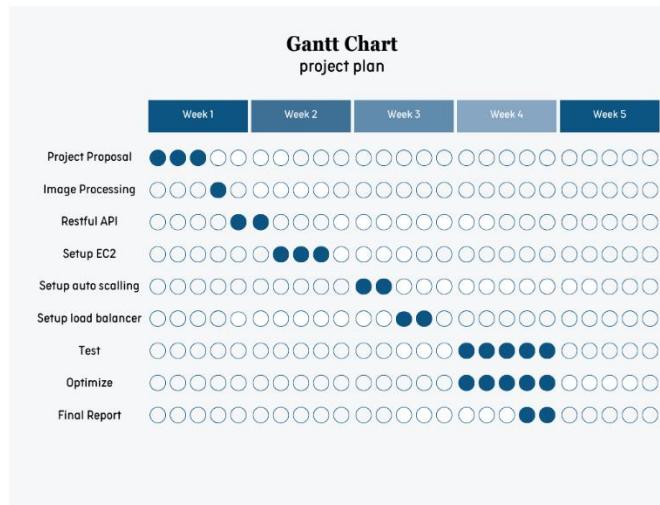


Figure 5: Gantt Chart

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, apertureSize

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

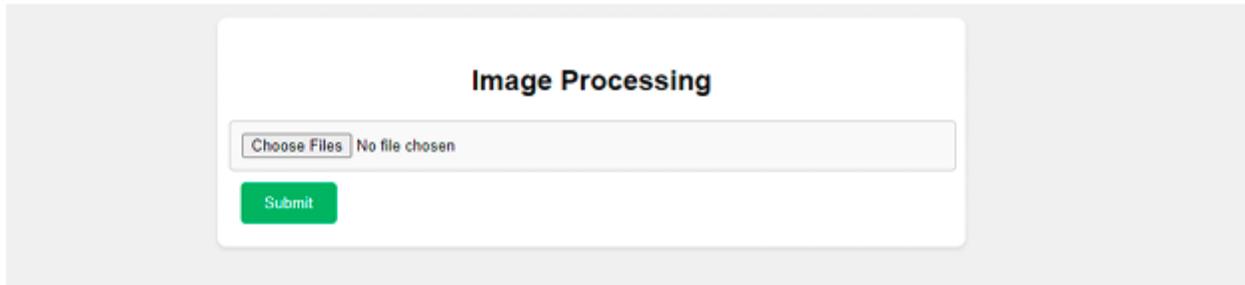


Figure 17 shows webpage view

Images uploaded by user are displayed on the webpage

Image Processing

Choose Files 2 files

Please choose operation for Image 1

Please choose operation for Image 2

Submit



Figure 18 shows images uploaded by user

User chooses what operation to be performed on each image

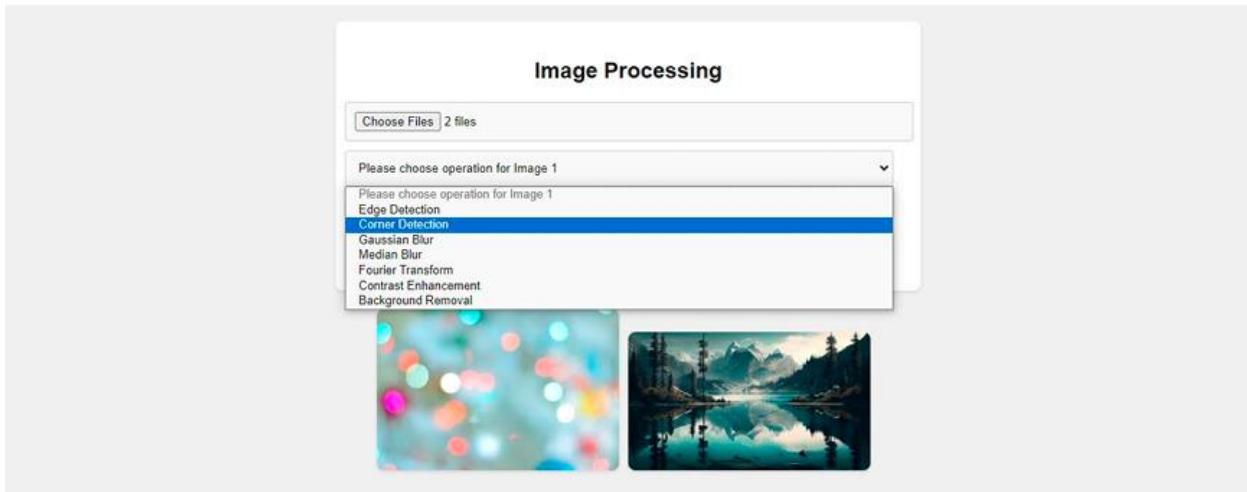


Figure 19 shows operations user chooses from

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 element
            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.

```
1  from flask import Flask, request, make_response
2  from image_processing import process_image
3  from flask_cors import CORS
4
5  app = Flask(__name__)
6  CORS(app)
7
8
9  @app.post("/process-image")
10 def image_processing_endpoint():
11     """Take the image and the operation.
12
13     Returns:
14         PNG Image: Processed image.
15     """
16
17     image = request.files["image"]
18     operation = request.form["operation"]
19     result = process_image(image, operation)
20     response = make_response(result)
21     response.headers["Content-Type"] = "image/png"
22     return response
23
24
25 # @app.post("/ocr")
26 # def ocr_endpoint():
27 #     """Take the image and apply ocr.
28
29 #     Returns:
30 #         PNG Image: Processed image.
31 #
32 #         image = request.files["image"]
33 #         result = ocr(image)
34 #         response = make_response(result)
35 #         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.

```
1  #!/bin/bash
2
3  # Update package lists
4  sudo apt update
5
6  # Install unzip (if not already installed)
7  sudo apt install -y unzip
8
9  # Install AWS CLI (if not already installed)
10 if ! command -v aws &> /dev/null
11 then
12     echo "AWS CLI is not installed. Installing..."
13     curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
14     unzip awscliv2.zip
15     sudo ./aws/install
16     echo "AWS CLI installed successfully."
17 else
18     echo "AWS CLI is already installed."
19 fi
20
21 # Download code from S3
22 echo "Downloading code from s3..."
23 mkdir /home/ubuntu/work/Distributed-Image-Processing/backend
24 aws s3 cp s3://code-directory/ /home/ubuntu/work/Distributed-Image-Processing/ --recursive
25 echo "Code downloaded successfully."
26
27 # Init the API
28 echo "Initializing WSGI server..."
29 chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_server.bash
30 ~/work/Distributed-Image-Processing/init_server.bash
31
32 # Init Apache
33 echo "Initializing Apache..."
34 chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_apache.bash
35 ~/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.

```
1 #!/bin/bash      You, 2 hours ago • modified indexe.html and allow
2
3 # Install required packages
4 sudo apt install -y python3-pip
5 sudo apt install -y python3-venv
6 sudo apt-get install ffmpeg libsm6 libxext6 -y
7
8 # Go to backend directory and create virtual environment
9 cd ~/work/Distributed-Image-Processing/backend
10 python3 -m venv venv
11
12 # Activate virtual environment
13 source venv/bin/activate
14
15 # Install dependencies
16 pip3 install -r requirements.txt
17 pip3 install gunicorn
18
19 # Start the server
20 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.

```
You, 54 minutes ago | Author (100)
1 #!/bin/bash
2
3 # Install Apache web server
4 sudo apt install apache2 -y
5
6 # Move the index.html file to the web server root directory
7 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.

The screenshot shows the AWS EC2 Instances page. The left sidebar navigation includes EC2 Dashboard, EC2 Global View, Events, Instances (selected), Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity, and Reservations. The main content area displays a table of instances:

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 address
Server	i-000c688a0de520c1b	Terminated	t3.micro	-	View alarms	eu-north-1b	-
Server	i-0c9046a73b089c7c4	Running	t3.micro	Initializing	View alarms	eu-north-1b	ec2-16-17
Server	i-0668ada68f72ba23a	Terminated	t3.micro	-	View alarms	eu-north-1b	-
Server	i-0e7d74cdcaa5b05c5a	Terminated	t3.micro	-	View alarms	eu-north-1b	-
Server	i-0e07a66fc3238e751	Terminated	t3.micro	-	View alarms	eu-north-1b	-

Below the table, a detailed view is shown for the instance i-0c9046a73b089c7c4 (Server). The details tab is selected, showing the following information:

Instance ID	Public IPv4 address	Private IPv4 addresses
i-0c9046a73b089c7c4 (Server)	16.171.140.36 open address	172.31.40.119

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

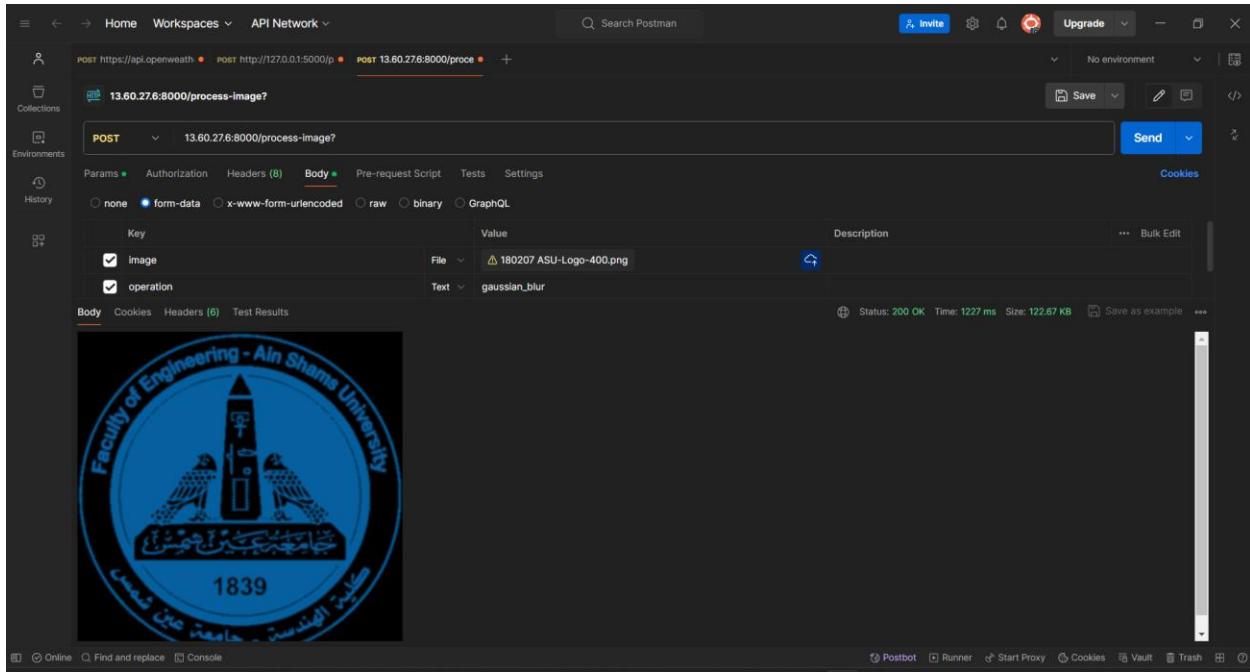


Figure 32 Testing Gaussian blur from postman

Median blur

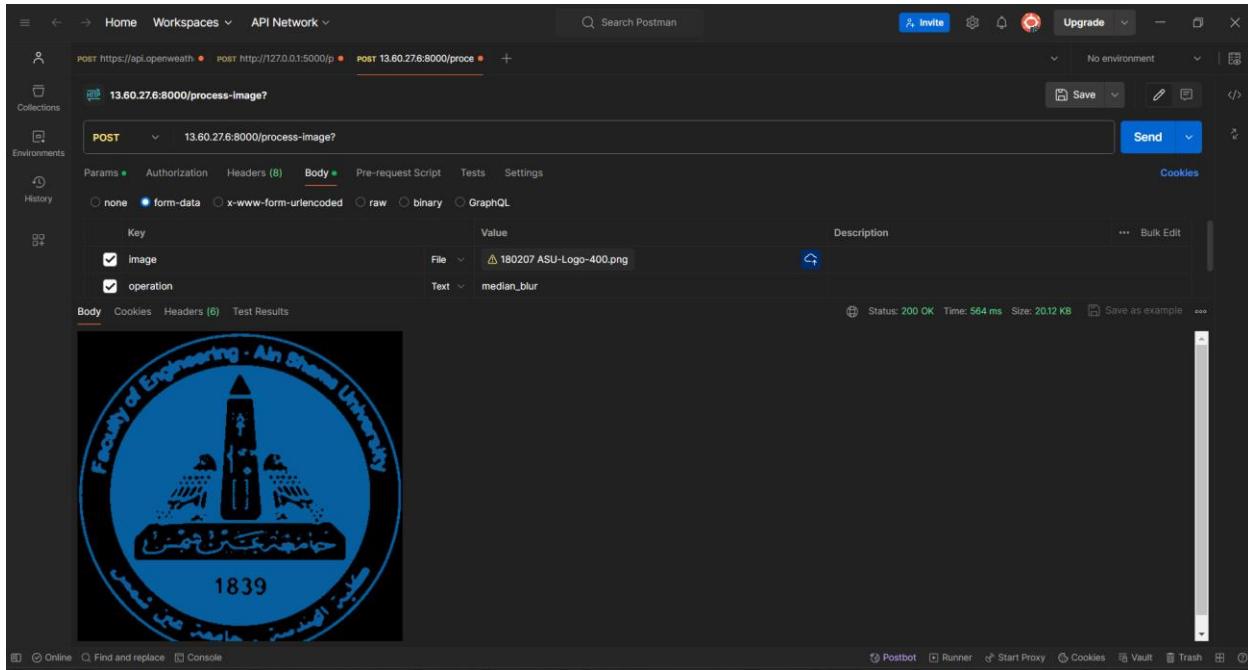


Figure 33 Testing median blur from postman

Fourier transform

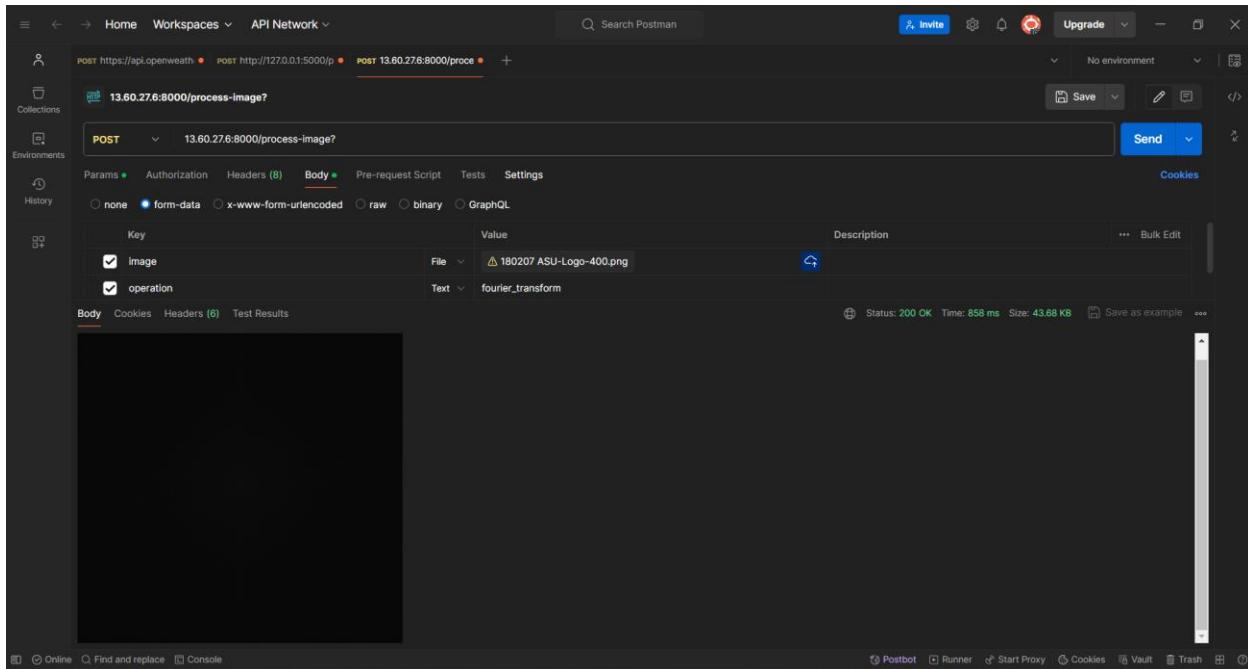


Figure 34 Testing Fourier transform from postman

Contrast enhancement

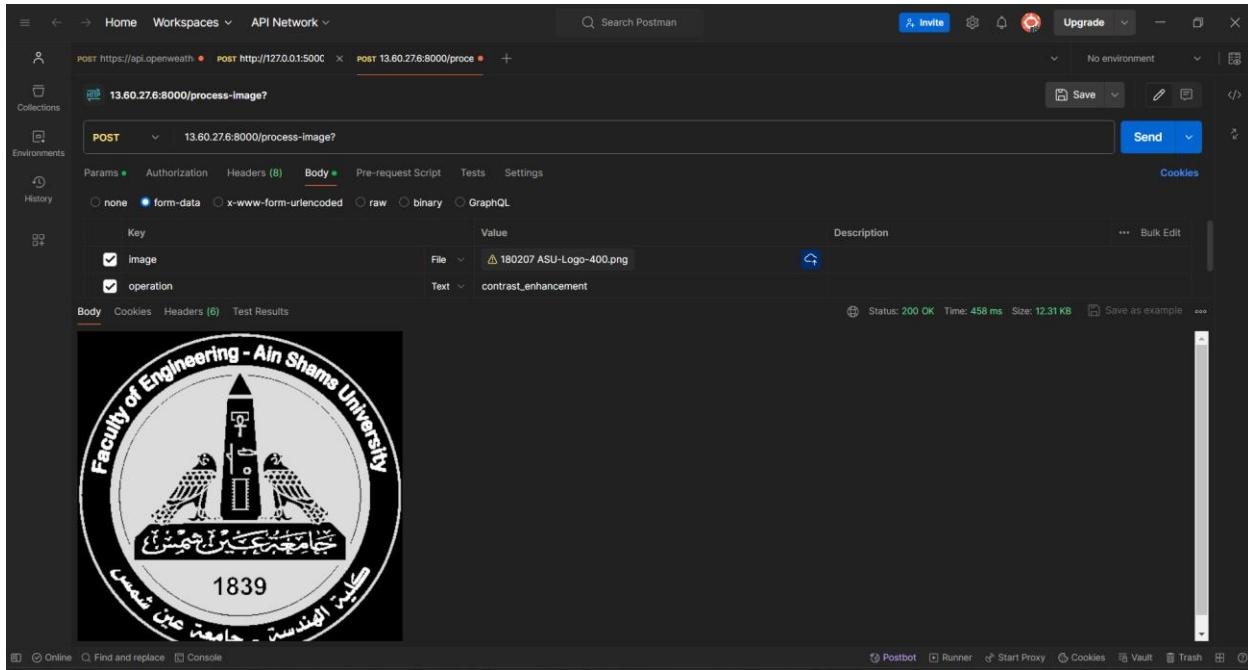


Figure 35 Testing contrast enhancement from postman

Corner detection

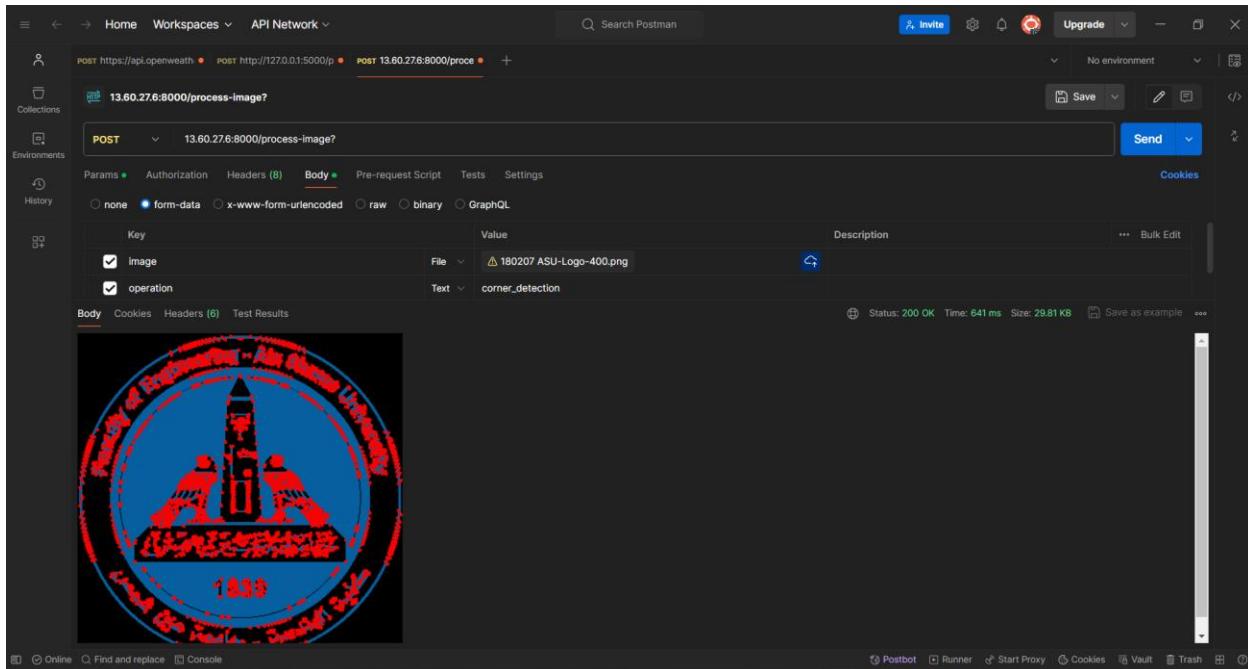


Figure 36: Testing corner detection from postman

Edge detection

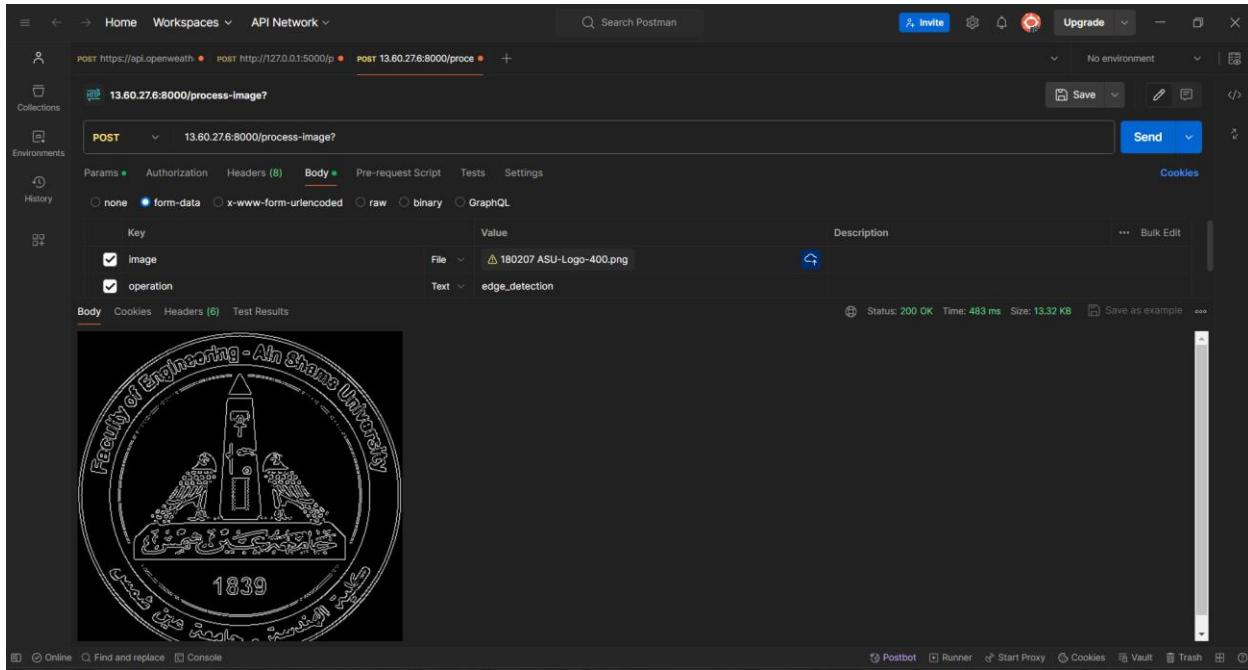


Figure 37 Testing edge detection from postman

Color inversion

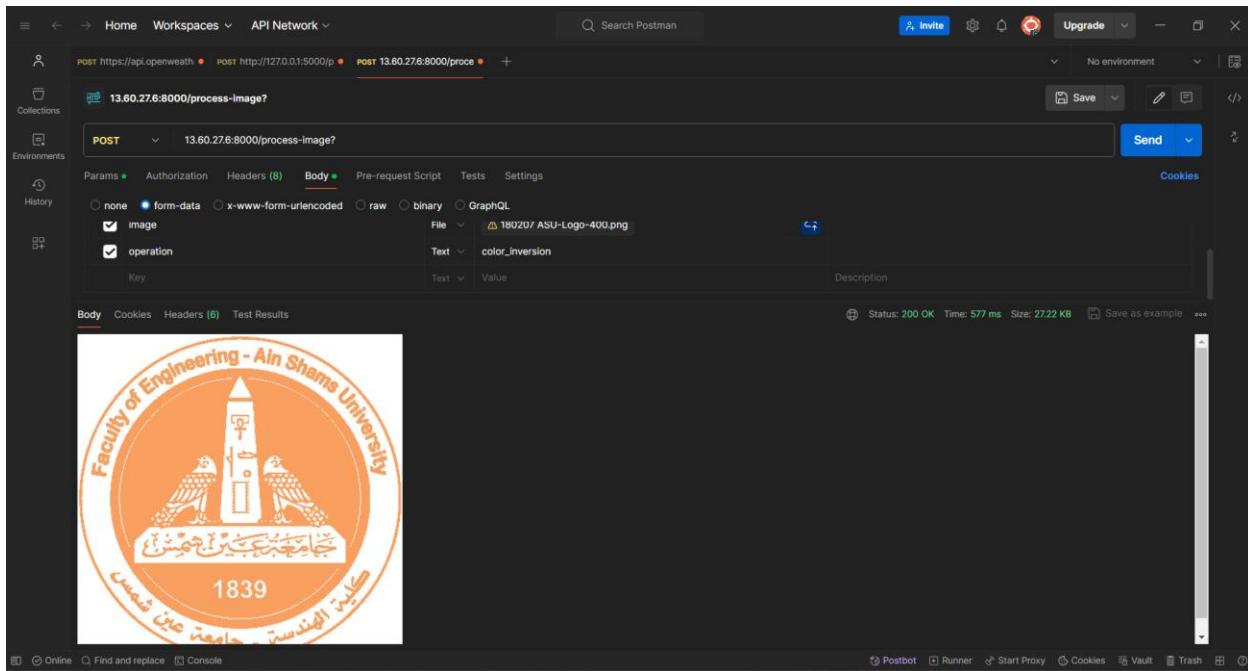


Figure 38 Testing color inversion from postman

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:

```
1  #!/bin/bash
2
3  # Update package lists
4  sudo apt update
5
6  # Install unzip (if not already installed)
7  sudo apt install -y unzip
8
9  # Install AWS CLI (if not already installed)
10 if ! command -v aws >& /dev/null
11 then
12     echo "AWS CLI is not installed. Installing..."
13     curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
14     unzip awscliv2.zip
15     sudo ./aws/install
16     echo "AWS CLI installed successfully."
17 else
18     echo "AWS CLI is already installed."
19 fi
20
21 # Download code from S3
22 echo "Downloading code from S3..."
23 mkdir /home/ubuntu/work/Distributed-Image-Processing/backend
24 aws s3 cp s3://code-directory/ /home/ubuntu/work/Distributed-Image-Processing/ --recursive
25 echo "Code downloaded successfully."
26
27 # Init Apache
28 echo "Initializing Apache..."
29 chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_apache.bash
30 /home/ubuntu/work/Distributed-Image-Processing/init_apache.bash
31
32 # Init the API
33 echo "Initializing WSGI server..."
34 chmod +x /home/ubuntu/work/Distributed-Image-Processing/init_server.bash
35 /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:

```
GNU nano 7.2                                         app.service *
```

```
[Unit]
Description=Gunicorn instance to serve myapp
After=network.target

[Service]
User=ubuntu
WorkingDirectory=/home/ubuntu/work/Distributed-Image-Processing/backend/
ExecStart=/home/ubuntu/work/Distributed-Image-Processing/backend/venv/bin/gunicorn --workers 2 --bind 0.0.0.0 app:app

[Install]
WantedBy=multi-user.target
```

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.

The screenshot shows the 'Network' configuration section of the AWS Auto Scaling VPC setup. It includes fields for selecting a VPC and defining availability zones and subnets.

Network Info

For most applications, you can use multiple Availability Zones and let EC2 Auto Scaling balance your instances across the zones. The default VPC and default subnets are suitable for getting started quickly.

VPC
Choose the VPC that defines the virtual network for your Auto Scaling group.

vpc-061a4e7778437fb6f
172.31.0.0/16 Default C

[Create a VPC](#) C

Availability Zones and subnets
Define which Availability Zones and subnets your Auto Scaling group can use in the chosen VPC.

Select Availability Zones and subnets ▼ C

eu-north-1c | subnet-0734b133b3098fc15 X
172.31.0.0/20 Default

eu-north-1b | subnet-034823513fffbec13 X
172.31.32.0/20 Default

eu-north-1a | subnet-00fce7b74b1aa0efa X
172.31.16.0/20 Default

[Create a subnet](#) C

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.

Attach to an existing load balancer

Select the load balancers that you want to attach to your Auto Scaling group.

Choose from your load balancer target groups

This option allows you to attach Application, Network, or Gateway Load Balancers.

Choose from Classic Load Balancers

Existing load balancer target groups

Only instance target groups that belong to the same VPC as your Auto Scaling group are available for selection.

Select target groups



Web-Apps | HTTP



Application Load Balancer: Web-app-and-server

Web-Servers | HTTP



Application Load Balancer: Web-app-and-server

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.

The screenshot shows the configuration interface for an Auto Scaling group. It is divided into two main sections: 'Group size' and 'Scaling'.

Group size (Info): Set the initial size of the Auto Scaling group. After creating the group, you can change its size to meet demand, either manually or by using automatic scaling.

Desired capacity type: Choose the unit of measurement for the desired capacity value. vCPUs and Memory(GiB) are only supported for mixed instances groups configured with a set of instance attributes.

Units (number of instances): A dropdown menu currently set to 'Units (number of instances)'.

Desired capacity: Specify your group size.

Specify your group size.: An input field containing the value '3'.

Scaling (Info): You can resize your Auto Scaling group manually or automatically to meet changes in demand.

Scaling limits: Set limits on how much your desired capacity can be increased or decreased.

Min desired capacity: An input field containing the value '1'.

Max desired capacity: An input field containing the value '10'.

Equal or less than desired capacity: A note indicating the scaling limit for the minimum capacity.

Equal or greater than desired capacity: A note indicating the scaling limit for the maximum capacity.

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.

Automatic scaling - optional

Choose whether to use a target tracking policy | [Info](#)

You can set up other metric-based scaling policies and scheduled scaling after creating your Auto Scaling group.

No scaling policies

Your Auto Scaling group will remain at its initial size and will not dynamically resize to meet demand.

Target tracking scaling policy

Choose a CloudWatch metric and target value and let the scaling policy adjust the desired capacity in proportion to the metric's value.

Scaling policy name

Target Tracking Policy

Metric type | [Info](#)

Monitored metric that determines if resource utilization is too low or high. If using EC2 metrics, consider enabling detailed monitoring for better scaling performance.

Average CPU utilization ▾

Target value

70

Instance warmup | [Info](#)

300 seconds

Disable scale in to create only a scale-out policy

Figure 44 scaling criteria configuration

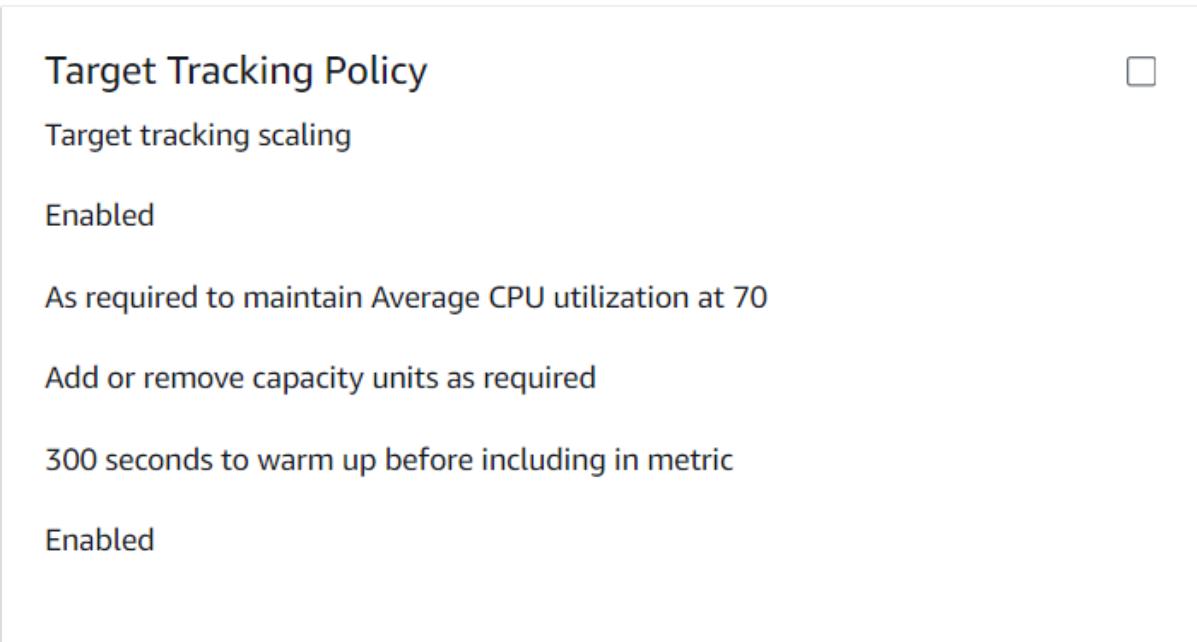


Figure 45 auto scaling summary

Instances (6) Info							
	Name Edit	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
<input type="checkbox"/>		i-00c36fdb60a634cc	Running View details Logs	t3.micro	Initializing	View alarms +	eu-north-1a
<input type="checkbox"/>	Server	i-0775ec55f16ee4e79	Terminated View details Logs	t3.micro	-	View alarms +	eu-north-1b
<input type="checkbox"/>	Server	i-0efce85c21a3b8a3f	Terminated View details Logs	t3.micro	-	View alarms +	eu-north-1b
<input type="checkbox"/>	Server	i-0a476faf05196f684	Terminated View details Logs	t3.micro	-	View alarms +	eu-north-1b
<input type="checkbox"/>		i-0bf5f08a032377e3	Running View details Logs	t3.micro	Initializing	View alarms +	eu-north-1b
<input type="checkbox"/>		i-0b1e1417bb44333e5	Running View details Logs	t3.micro	Initializing	View alarms +	eu-north-1c

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:

The screenshot shows the Microsoft VS Code interface with the following details:

- File Explorer:** Shows the project structure for "DISTRIBUTED-IMAGE-PROCESSING". The "frontend" folder contains "index.html" and "index.js".
- Code Editor:** Displays the content of "index.js". The code handles file uploads via a form, creates a FormData object, and sends a POST request to an endpoint at "http://Web-app-and-server-413737800.eu-north-1.elb.amazonaws.com:8080/process-image". It includes logic for retrying requests if they fail.
- Terminal:** Shows the command "aws s3 cp index.html s3://amazon-q-test" being run.
- Status Bar:** Shows the path "H:\DESKTOP-ERK90\MDK6464 C:\Users\user\ApplData\Local\Programs\Microsoft VS Code (main)" and the status "bash - Microsoft VS Code".

Figure 47 client side middleware, request retrying code snippet

Demo

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

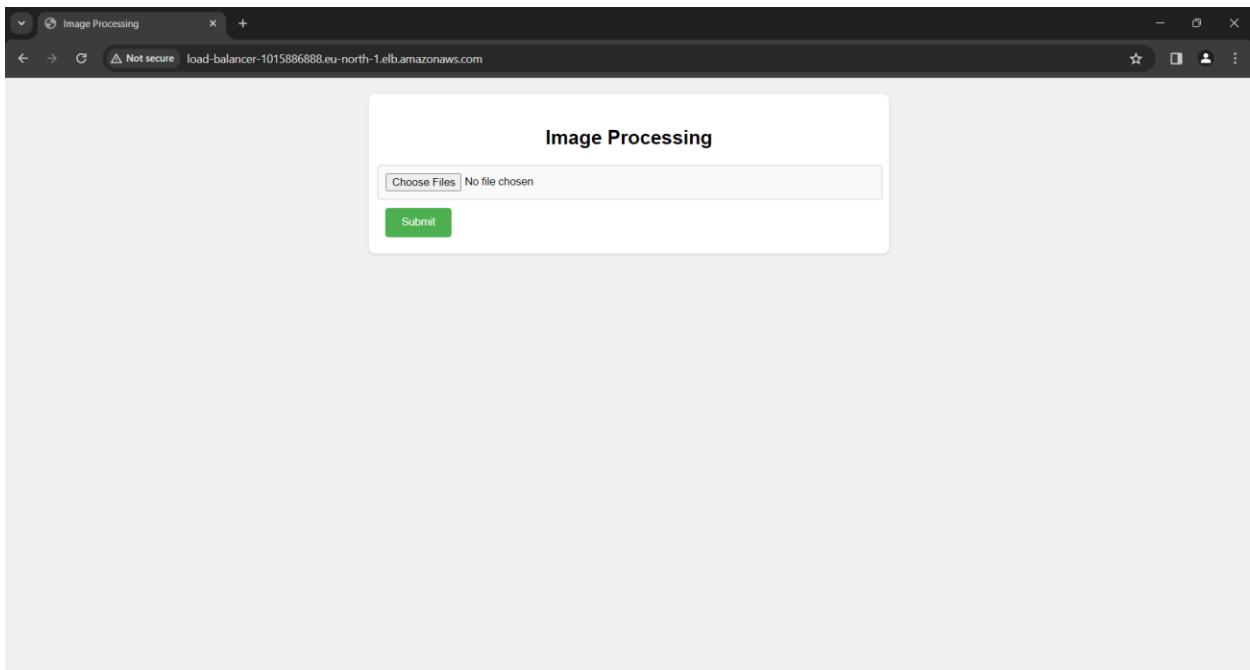


Figure 48 demo, home screen with the load balancer URL

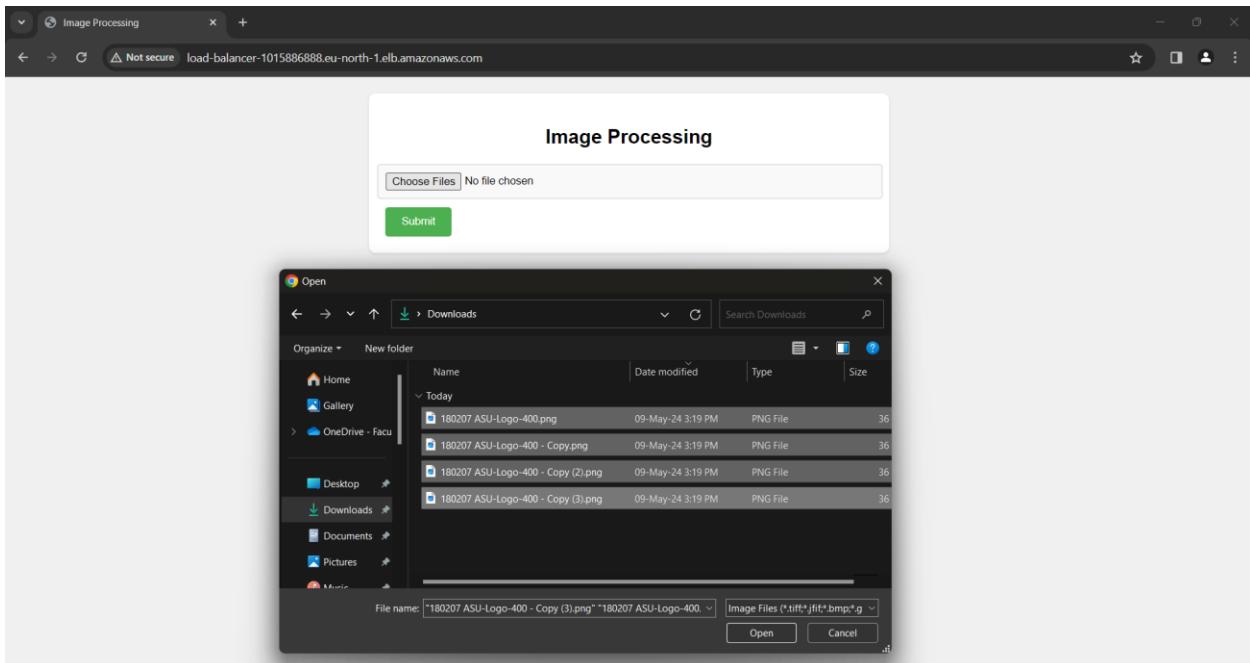


Figure 49: demo, uploading multiple images

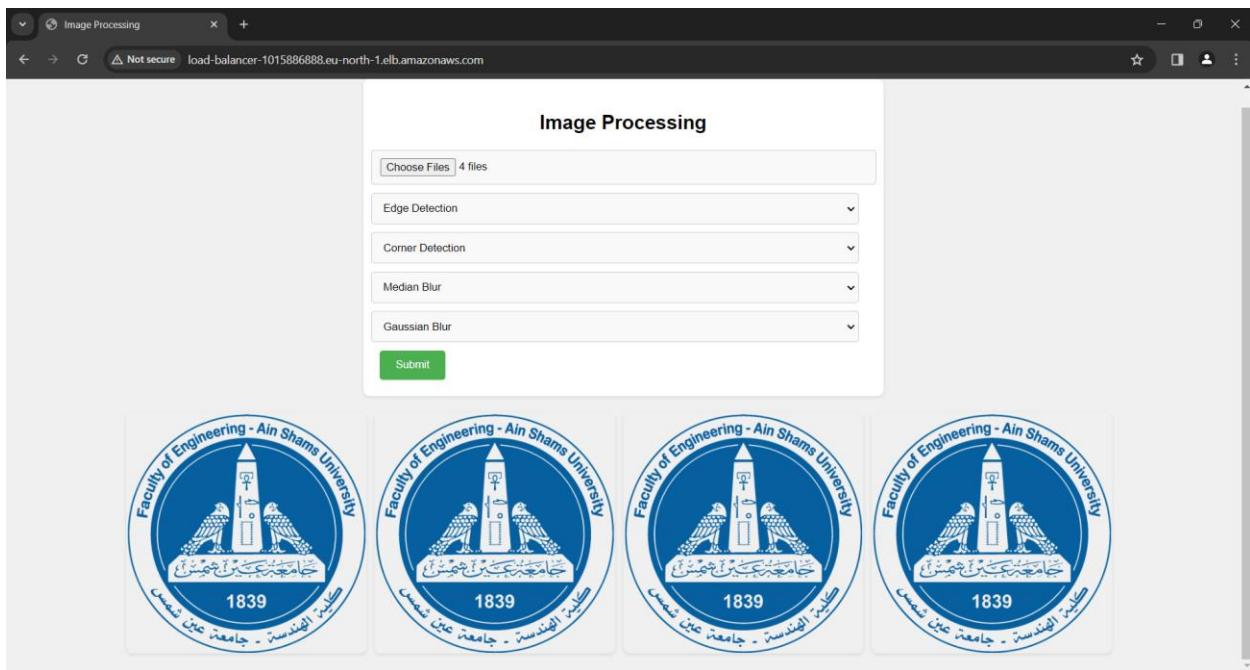


Figure 50: demo, different operation for each image

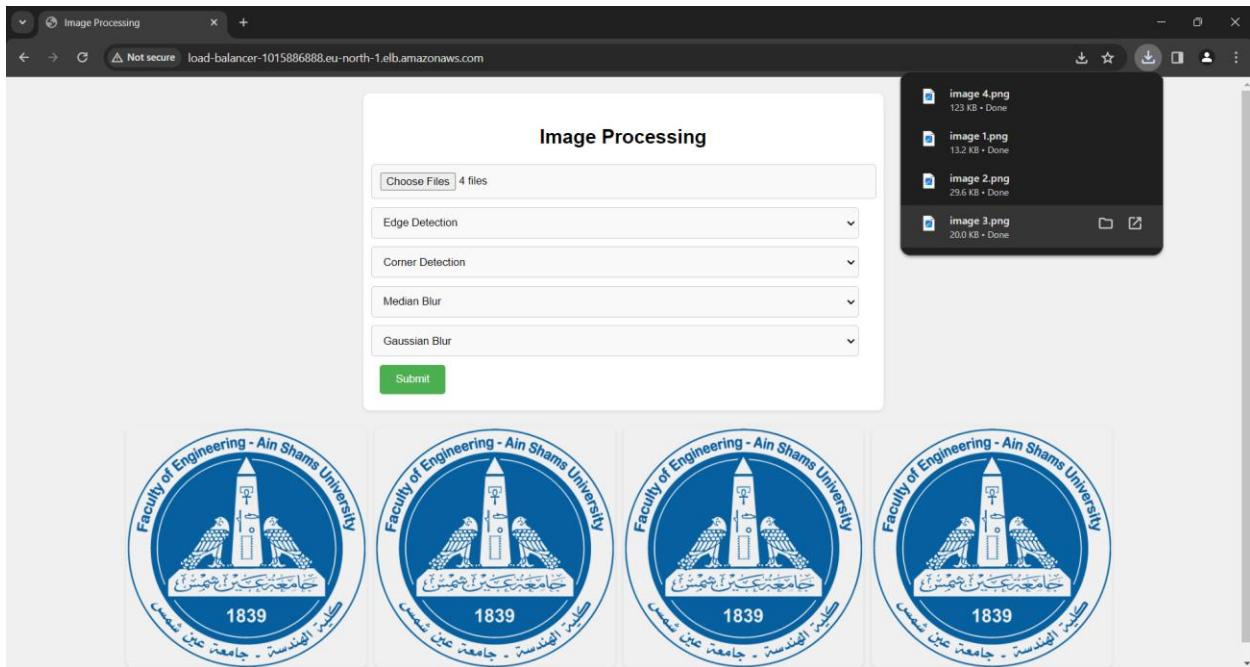


Figure 51: demo, image processing results downloaded

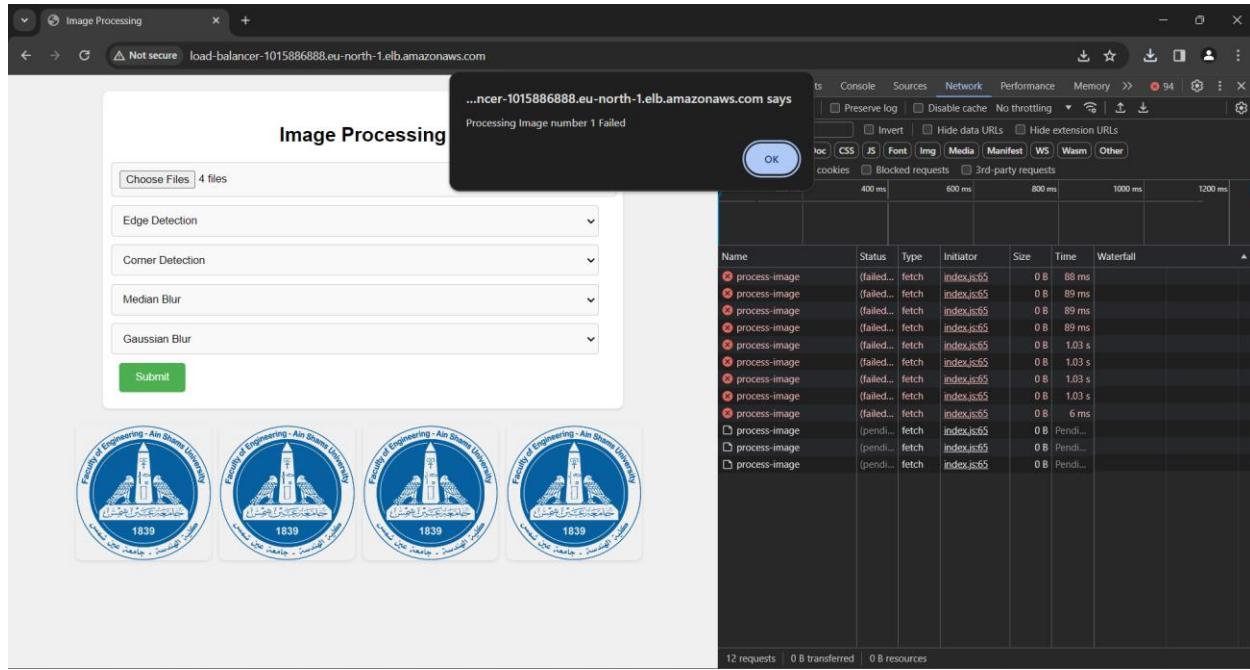


Figure 52 demo, fault tolerance mechanism

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.

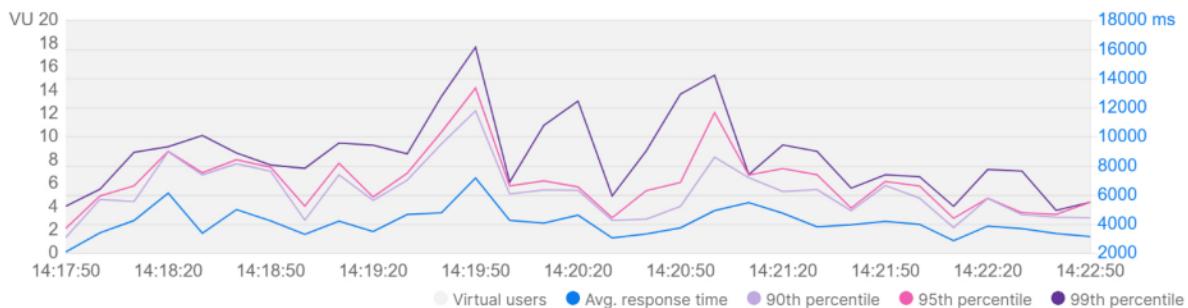


Figure 53: Fixed load with 20 virtual users results

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

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.

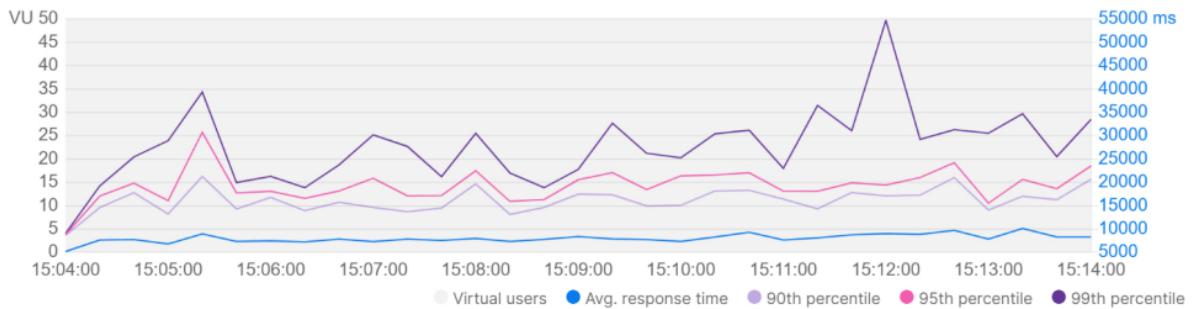


Figure 54: Fixed load with 50 virtual users results

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.

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.

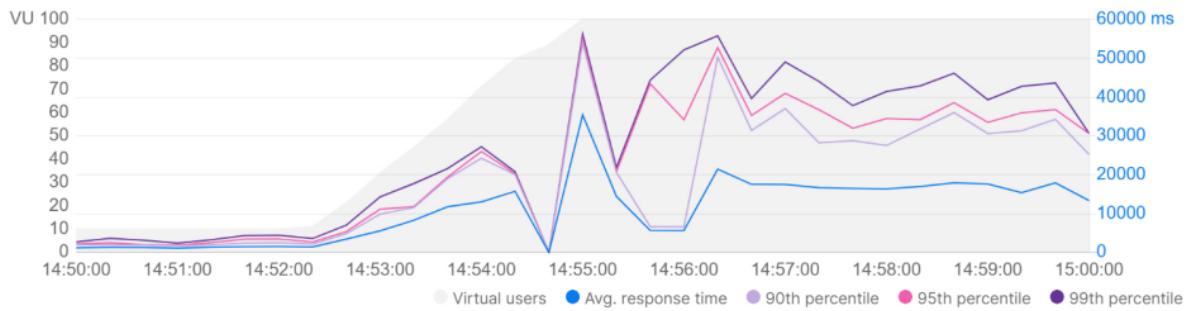


Figure 55: Ramp up testing results

3. Errors

3.1 Error distribution over time

Top 5 error classes observed during the test duration.

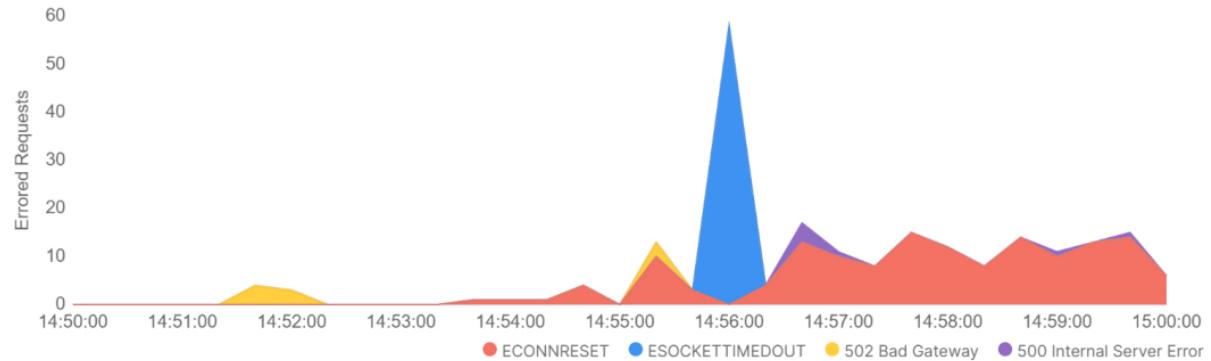


Figure 56 Ramp up testing error distribution

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

```

Tests > ⚡ test_frontend.py > ...
1 import unittest
2 from selenium import webdriver
3 from selenium.webdriver.common.by import By
4 from selenium.webdriver.support.ui import Select
5 from selenium.webdriver.common.keys import Keys
6 import requests
7 import os
8 import sys
9 import time
10 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
11 from backend.image_processing import gaussian_blur, contrast_enhancement, edge_detection # Import the image processing functions
12 download_dir = os.path.abspath('Downloads')
13
14 chrome_options = webdriver.ChromeOptions()
15 chrome_options.add_experimental_option('prefs', {
16     'download.default_directory': download_dir,
17     'download.prompt_for_download': False,
18     'download.directory_upgrade': True,
19     'safebrowsing.enabled': True
20 })
21
22
23
24 # Path to folder containing images
25 image_folder_path = os.path.abspath('Tests/Images')
26
27 # List of image files
28 image_files = os.listdir(image_folder_path)
29 # print(image_files)
30 # Number of tabs to open
31 num_tabs = 3

```

Figure 57 Shows library imports

The figure below 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

```
1  class TestImageUpload(unittest.TestCase):
2
3
4
5
6
7
8     def upload_image(self, driver, image_path):
9         file_input = driver.find_element(By.XPATH, "//input[@type='file']")
10        file_input.send_keys(image_path)
11        select_element = driver.find_element(
12            By.XPATH, "//div[@id='imageFilters']/select")
13        select = Select(select_element)
14        select.select_by_visible_text('Edge Detection')
15        # print("Hello")
16        # Wait for the option to be selected (add a delay if necessary)
17        driver.implicitly_wait(5)
18
19
20    # Find the submit button element
21    submit_button = driver.find_element(
22        By.XPATH, "//button[@type='submit']")
23    # Click the submit button
24    submit_button.click()
25
26    # Wait for the image to be downloaded
27    max_wait_time = 60 # Maximum wait time in seconds
28    polling_interval = 1 # Polling interval in seconds
29    image_filename = "image 1.png"
30    image_path = "./Tests/Images/image1.jpg"
31    wait_time = 0
32
33    while not os.path.exists(image_path):
34        time.sleep(polling_interval)
35        wait_time += polling_interval
36
37        if wait_time >= max_wait_time:
38            self.fail(f"Image download took too long. Expected file '{image_filename}' was not found.")
39
40    # Assertion: Check if the file exists
41    self.assertTrue(os.path.exists(image_path),
42                    f"Processed image not found: {image_path}")
43
44
45    if __name__ == "__main__":
46        unittest.main()
```

Figure 59 shows function of uploading an image

The figure below shows Test class of our project

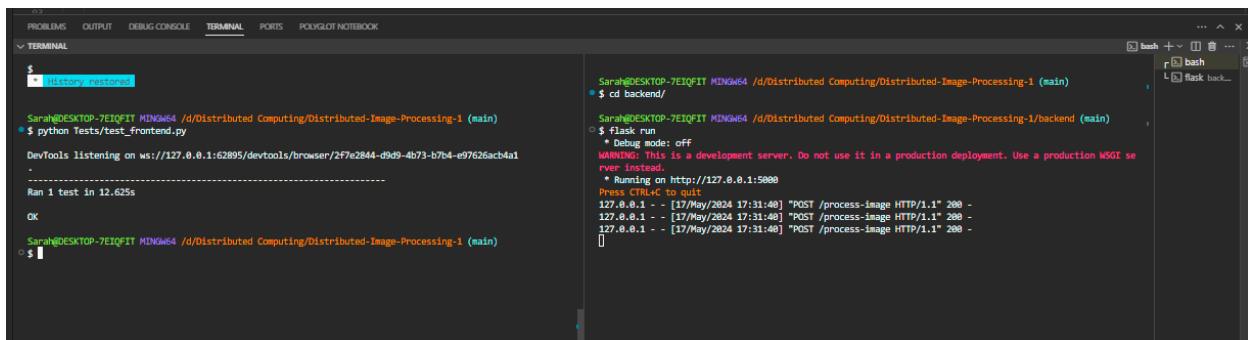
Chrome driver is used (as discussed before). Multiple tabs are opened and then each tab is navigated to the file path (form page) then upload image function is called (discussed in the previous page)

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

Figure 60 shows test class

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

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



A screenshot of a terminal window titled "TERMINAL". The window has two tabs: "bash" and "flask backend". The "bash" tab contains the following command and output:

```
Sarah@DESKTOP-7EIQFIT MINGW64 /d/Distributed Computing/Distributed-Image-Processing-1 (main)
$ history restored
Sarah@DESKTOP-7EIQFIT MINGW64 /d/Distributed Computing/Distributed-Image-Processing-1 (main)
$ python Tests/test_frontend.py
DevTools listening on ws://127.0.0.1:62895/devtools/browser/2F7e2844-d9d9-4b73-b7b4-e97626acb4a1
.
.
.
Ran 1 test in 12.625s
OK
Sarah@DESKTOP-7EIQFIT MINGW64 /d/Distributed Computing/Distributed-Image-Processing-1 (main)
$
```

The "flask backend" tab shows the server is running on port 5000, and it is receiving POST requests for "/process-image".

Figure 61 shows successful test

Demo

https://drive.google.com/file/d/15-1gsSUfRz_P8EkznLFIMcaZfxNMfjrw/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 Osama 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

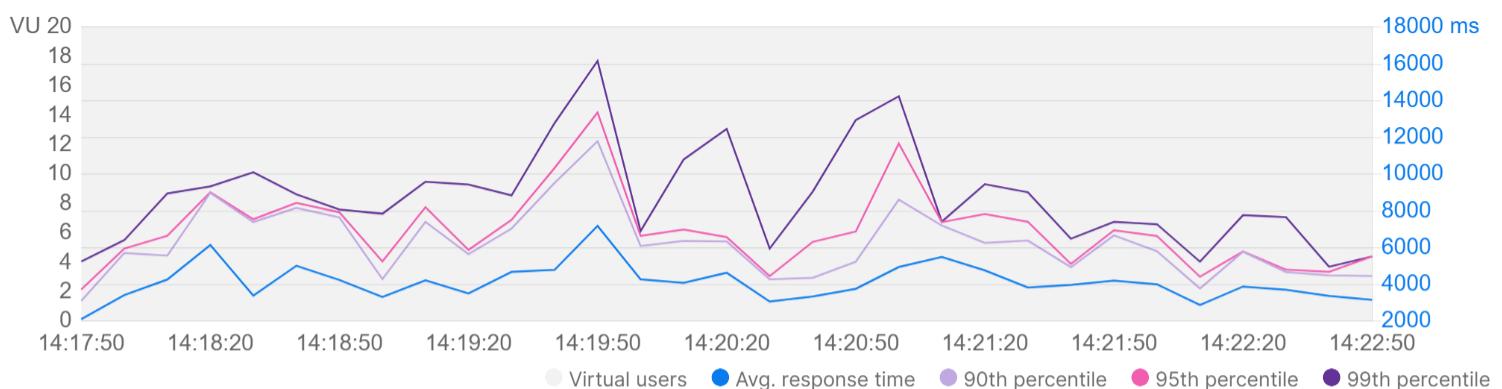
Virtual users	Start time	Load profile
20 VU	May 17, 14:17:45 (GMT+2)	Fixed
Duration	End time	Environment
5 minutes	May 17, 14:22:54 (GMT+2)	-

1. Summary

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

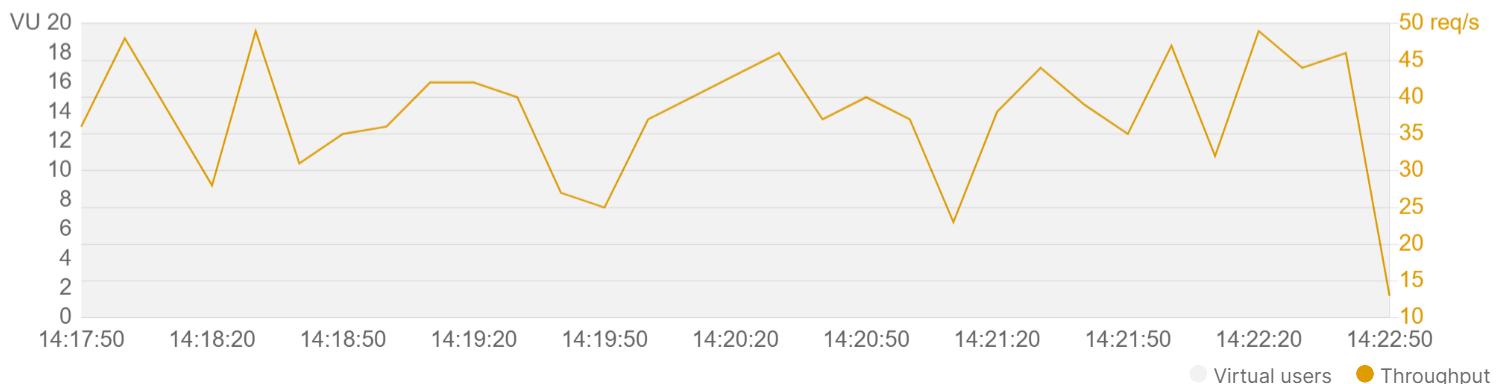
1.1 Response time

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

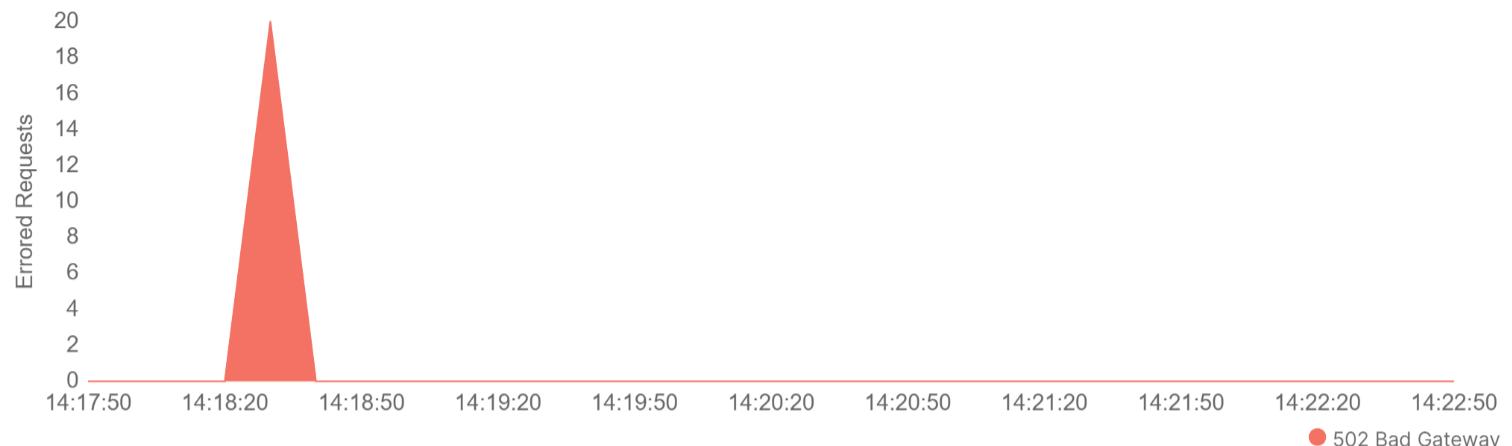
The requests are shown in the order they were sent by virtual users.

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
<code>POST</code> 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

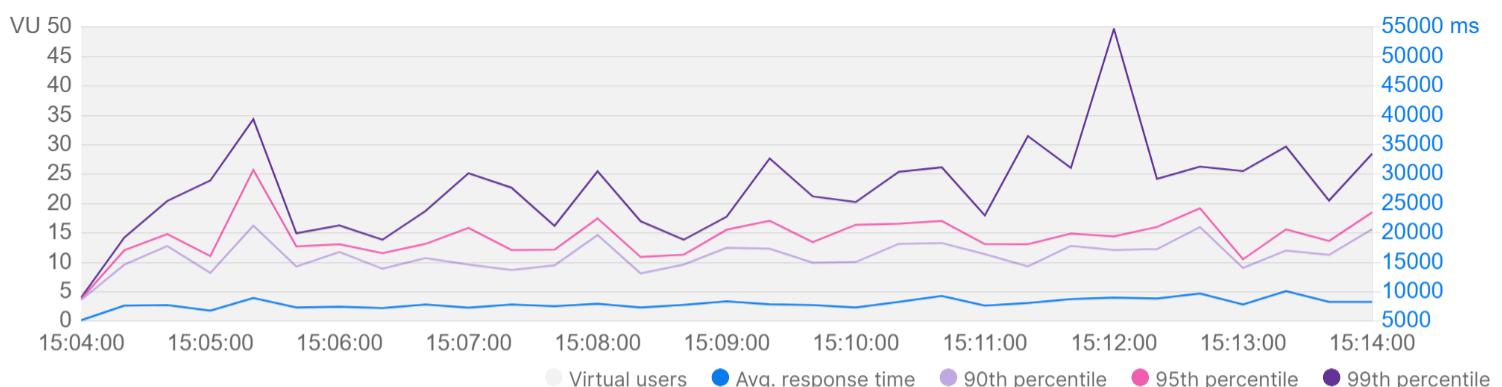
Virtual users	Start time	Load profile
50 VU	May 17, 15:04:05 (GMT+2)	Fixed
Duration	End time	Environment
10 minutes	May 17, 15:14:13 (GMT+2)	-

1. Summary

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

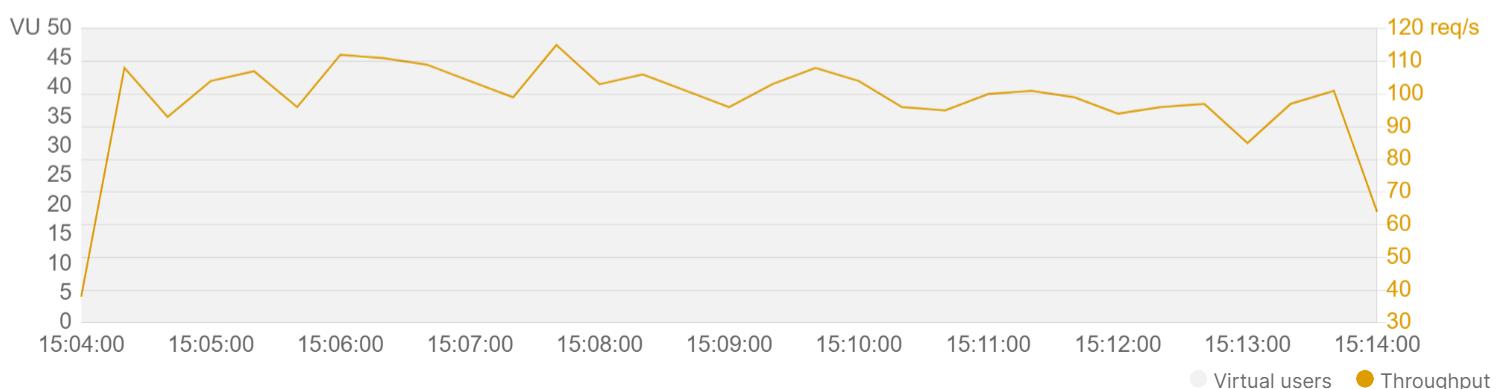
1.1 Response time

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

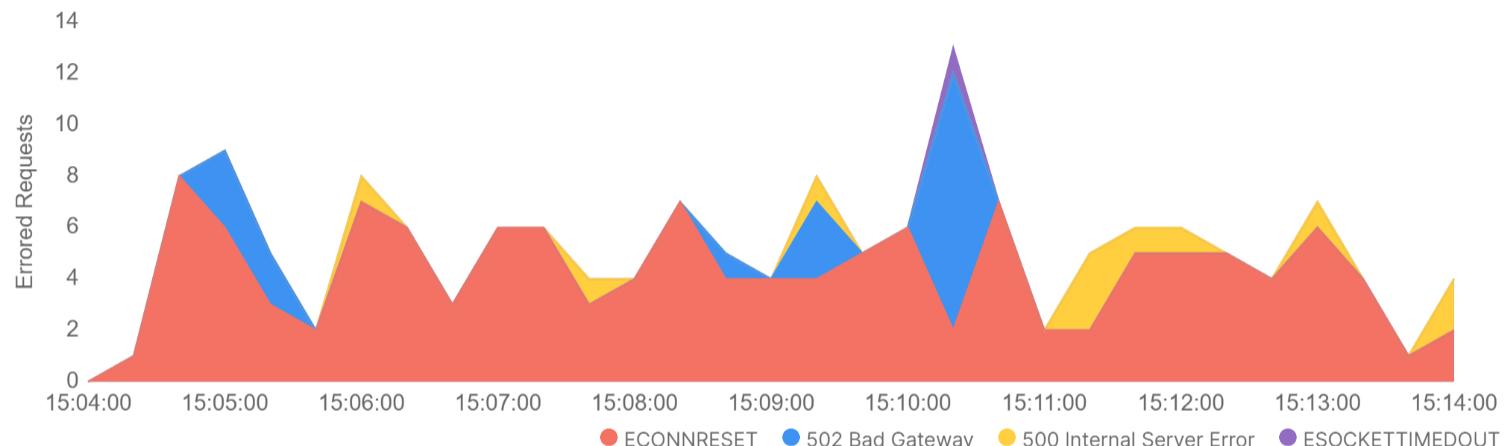
The requests are shown in the order they were sent by virtual users.

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
<code>POST</code> New Request	130
502 Bad Gateway	19
<code>POST</code> New Request	19
500 Internal Server Error	11
<code>POST</code> New Request	11
ESOCKETTIMEDOUT	1
<code>POST</code> 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

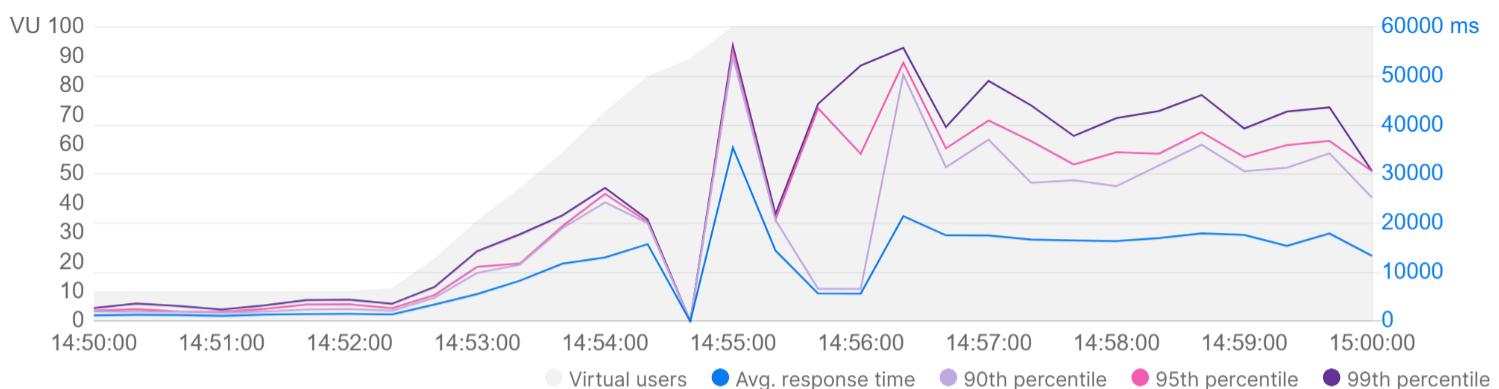
Virtual users	Start time	Load profile
100 VU	May 17, 14:49:56 (GMT+2)	Ramp up (2 minutes 30 seconds)
Duration	End time	Environment
10 minutes	May 17, 15:00:05 (GMT+2)	-

1. Summary

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

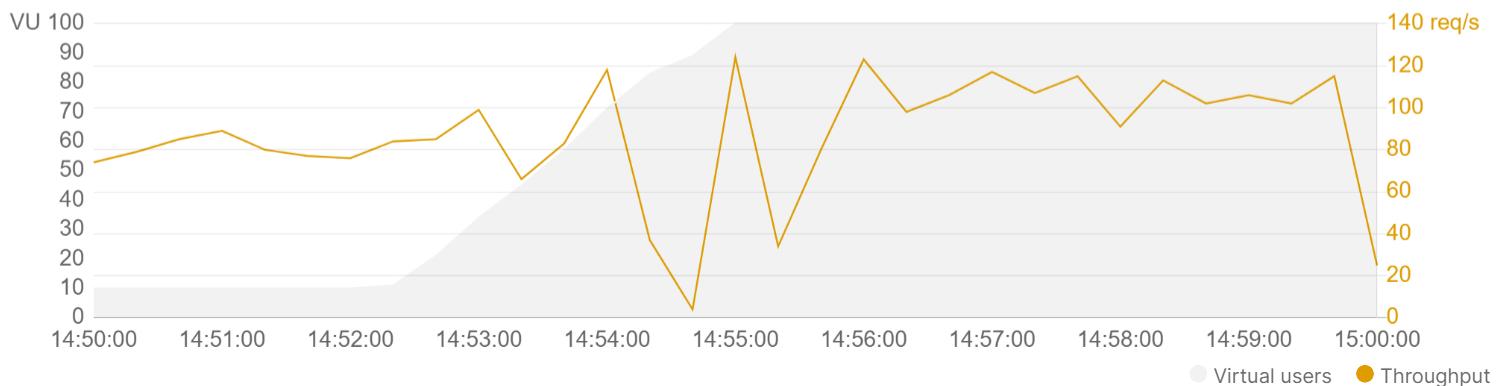
1.1 Response time

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

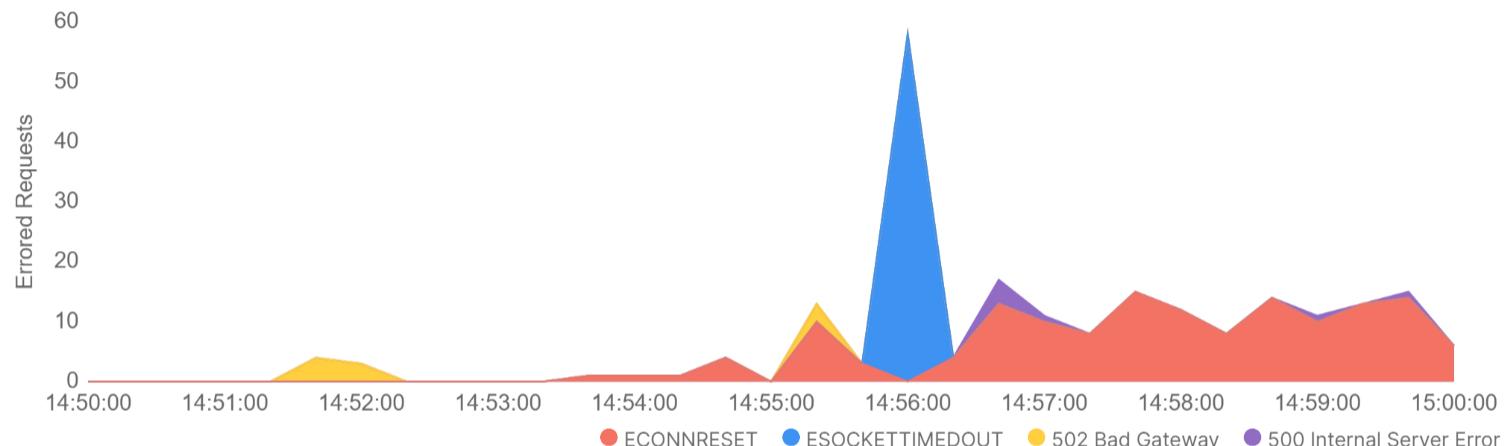
The requests are shown in the order they were sent by virtual users.

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](#).