Text

Description automatically generated

**AIN SHAMS UNIVERSITY**

**FACULTY OF ENGINEERING**

**AIN SHAMS UNIVERSITY**

**FACULTY OF ENGINEERING**

**CREDIT HOURS ENG. PROGRAM**

**Computer engineering and software systems**

A picture containing logo

Description automatically generated

**Distributed Computing: CSE 354**

**Project**

**Submitted to:**

**Prof. Ayman M. Bahaa-Eldin**

**Submitted by:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| **Mohamed Mostafa ElMaghraby** | **20p7732** |
| **Anass Zikry** | **20P6382** |
| **Tarek Khaled Ezzat** | **20P1087** |
| **Mohamed Hesham El Said** | **20P7579** |

Contents

[Project Plan 3](#_Toc166860593)

[Phase 1: Planning and Design (week 1) 3](#_Toc166860594)

[Phase 2: Development (week 2) 3](#_Toc166860595)

[1- Setting up Cloud Infrastructure 3](#_Toc166860596)

[2 - Implementing UI 3](#_Toc166860597)

[3 - Implementing Image Processing Layer 3](#_Toc166860598)

[Phase 3: Distribution infrastructure (week 3) 3](#_Toc166860599)

[Phase 4: Testing and Deployment (week 4) 3](#_Toc166860600)

[Introduction 4](#_Toc166860601)

[Project Scope: 4](#_Toc166860602)

[Objectives: 4](#_Toc166860603)

[Requirements: 4](#_Toc166860604)

[Functional Requirements: 4](#_Toc166860605)

[Non-Functional Requirements: 5](#_Toc166860606)

[System Architecture 5](#_Toc166860607)

[1- User Interface (UI): 5](#_Toc166860608)

[2- Application Layer: 5](#_Toc166860609)

[3- Image Processing Layer: 5](#_Toc166860610)

[4- Cloud Infrastructure: 5](#_Toc166860611)

[5- Fault Tolerance and Monitoring: 6](#_Toc166860612)

[User Stories 8](#_Toc166860613)

[UML Diagrams: 8](#_Toc166860614)

[Sequence Diagram: 8](#_Toc166860615)

[Class Diagram: 9](#_Toc166860616)

[Component Diagram: 9](#_Toc166860617)

[Network Diagram: 10](#_Toc166860618)

[Beneficiaries of the project 10](#_Toc166860619)

[Photographers and Graphic Designers 10](#_Toc166860620)

[E-commerce Platforms 10](#_Toc166860621)

[Social Media Platforms 10](#_Toc166860622)

[Artificial Intelligence and Machine Learning Researchers 10](#_Toc166860623)

[Testing scenarios and results 11](#_Toc166860624)

[Time Test 11](#_Toc166860625)

[Scenario 11](#_Toc166860626)

[Result 11](#_Toc166860627)

[Scalability Test 11](#_Toc166860628)

[Scenario 11](#_Toc166860629)

[Result 11](#_Toc166860630)

[Resource Utilization Test 11](#_Toc166860631)

[Scenario 11](#_Toc166860632)

[Result 11](#_Toc166860633)

[User Interface Test 11](#_Toc166860634)

[Scenario 11](#_Toc166860635)

[Result 11](#_Toc166860636)

[Monitoring 11](#_Toc166860637)

[Logs: 12](#_Toc166860638)

[Merging: 12](#_Toc166860639)

[Utilized function: 12](#_Toc166860640)

[Utilized bash script: 12](#_Toc166860641)

[Merge function: 13](#_Toc166860642)

[End-user guide 14](#_Toc166860643)

[Azure Structure 18](#_Toc166860644)

[Diagram 18](#_Toc166860645)

[Virtual Machines 18](#_Toc166860646)

[Distributed Processing Implementation 19](#_Toc166860647)

[System Architecture 19](#_Toc166860648)

[Fault Tolerance 20](#_Toc166860649)

[Scenario 20](#_Toc166860650)

[Result 20](#_Toc166860651)

[System Scaling 20](#_Toc166860652)

[Conclusion 20](#_Toc166860653)

[Table of figures 22](#_Toc166860654)

# Project Plan

## Phase 1: Planning and Design (week 1)

1. Discuss project scope, objectives, and requirements.
2. Assign team roles and responsibilities.
3. Review and finalize functional requirements.
4. Identify any additional non-functional requirements.
5. Design the high-level system architecture.
6. Determine the technology stack and tools to be used.
7. Document user stories and acceptance criteria.
8. Create diagrams (e.g., UML diagrams) to illustrate system interactions.

## Phase 2: Development (week 2)

### Setting up Cloud Infrastructure

* Provision cloud resources (VM instances, object storage).
* Set up monitoring and logging tools.

### 2 - Implementing UI

* Set up a framework for the web-based UI.
* Implement functionality for uploading images and selecting processing operations.
* Integrate UI with backend APIs for task submission and monitoring.

### 3 - Implementing Image Processing Layer

* Integrate OpenCV library for image processing algorithms.
* Implement parallel processing using OpenCL or MPI.
* Test and optimize image processing algorithms for performance.

## Phase 3: Distribution infrastructure (week 3)

* Implement task scheduler for distributing processing tasks.
* Develop load balancer for evenly distributing workload across VMs and making the system scalable.
* Implement error handling and retry mechanisms for fault tolerance.

## Phase 4: Testing and Deployment (week 4)

* Conduct thorough testing to ensure the system works as expected.
* Deploy the system to the cloud and ensure it is operational.
* Prepare end-user manual. (Readme file on GitHub)

# Introduction

By leveraging networked computing paradigms, the proposed idea introduces a novel method of image processing. Using distributed systems to their full potential, this project aims to transform image processing, improving scalability, efficiency, and reliability. The project intends to speed image processing operations while supporting fluctuating workloads and resource demands by smoothly spreading processing activities over a network of nodes. This program offers new opportunities for picture analysis, enhancement, and modification in addition to a substantial processing time reduction. By means of rigorous design, implementation, and testing, the project aims to provide a strong and flexible framework that can handle the many requirements and difficulties linked with contemporary image processing applications.

# Project Scope:

The scope of the project encompasses the design, implementation, and testing of a distributed image processing system using cloud computing technologies. The system will allow users to upload images for various processing operations, such as filtering, edge detection, and color manipulation. It will distribute processing tasks across multiple virtual machines in the cloud to achieve scalability and improve performance. The system will also be designed to be fault-tolerant, capable of detecting and recovering from failures to ensure uninterrupted processing.

# Objectives:

* Develop a distributed image processing system that leverages cloud computing technologies for scalability and performance.
* Implement various image processing algorithms, including filtering, edge detection, and color manipulation, to provide a diverse range of processing options.
* Enable users to upload images, select processing operations, monitor task progress, and download processed images.
* Ensure the system is scalable, allowing for the addition of more virtual machines to handle increased workload.
* Design fault tolerance mechanisms to detect and recover from failures, ensuring uninterrupted processing even in the event of VM failures.

# Requirements:

## Functional Requirements:

* Users should be able to upload images for processing.
* The system should support various image processing operations, including filtering, edge detection, and color manipulation.
* Users should be able to select the desired processing operation for each uploaded image.
* The system should distribute image processing tasks across multiple virtual machines in the cloud.
* Progress monitoring functionality should be provided to allow users to track the status of their processing tasks.
* Processed images should be available for download once processing is complete.

## Non-Functional Requirements:

* The system should process image tasks within a reasonable timeframe.
* The system should be scalable.
* The system should be capable of recovering from failures (e.g., VM crashes, network outages) automatically without human intervention.
* Code should be well-documented and adhere to coding best practices to ease maintenance and troubleshooting efforts.
* The user interface should be intuitive and easy to navigate, with clear instructions provided for each functionality.

# System Architecture

## User Interface (UI):

* Flask (Python web frameworks): For building a web-based UI where users can upload images, select processing operations, monitor task progress, and download processed images.

## Application Layer:

* Python: As the primary language for implementing the application logic.
* Task Scheduler: Orchestrates the distribution of image processing tasks among virtual machines (VMs) in the cloud.
* Load Balancer: Distributes incoming user requests across multiple VMs to ensure even workload distribution.

## Image Processing Layer:

* OpenCV: A popular library for image processing in Python, providing functions for various operations such as filtering, edge detection, and color manipulation.
* OpenCL: For parallel processing of image data across multiple VMs. OpenCL is suitable for heterogeneous computing environments.

## Cloud Infrastructure:

* Amazon Web Services (AWS), Google Cloud Platform (GCP), or Microsoft Azure: For provisioning virtual machines and managing the cloud infrastructure.
* Virtual Machines (VMs): Multiple VM instances will be created to distribute image processing tasks. VM instances should be scalable based on workload demands.
* Object Storage: Store uploaded images and processed images. AWS S3, Google Cloud Storage, or Azure Blob Storage can be used for this purpose.

## Fault Tolerance and Monitoring:

* Kubernetes: For container orchestration and managing VMs as containers. Kubernetes provides features like automatic scaling and self-healing, improving fault tolerance.
* Prometheus and Grafana: For monitoring the health and performance of VMs, containers, and application components.

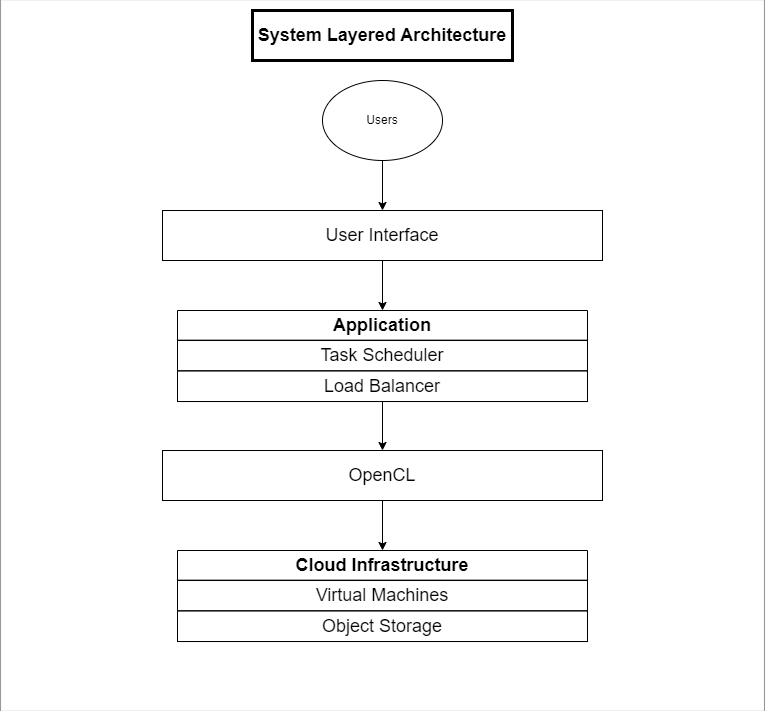


Figure 1 System layered architecture

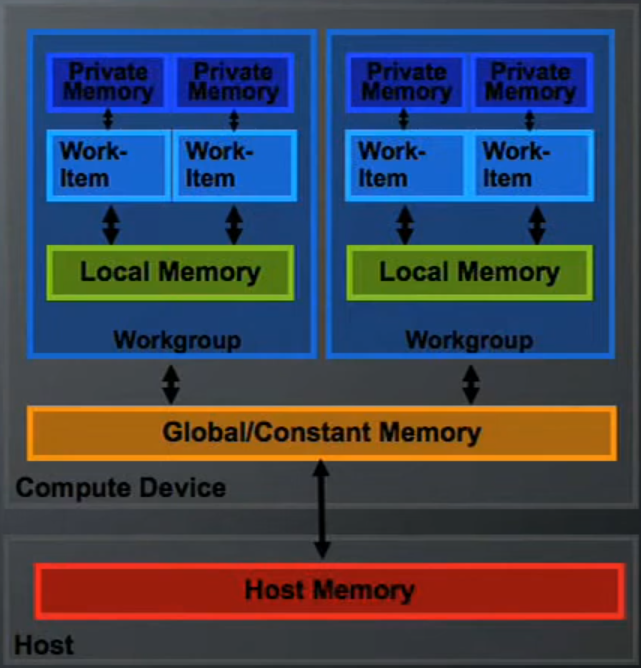


Figure 2 OpenCL architecture

# User Stories

* As a user, I want to upload an image to the system for processing.
* As a user, I want to select the type of image processing operation to be performed.
* As a user, I want to download the processed image once the operation is complete.
* As a user, I want to monitor the progress of the image processing task.
* As a team, we need to implement an efficient architecture for distributing image processing tasks across multiple virtual machines.
* As a team, we need to ensure the system’s scalability and fault tolerance.
* As a team, we need to consider image size, processing complexity, and available resources when splitting tasks into smaller units that can be distributed among available resources.
* As a team, we need to implement functionalities to monitor and track progress of tasks and inform the user of any change.
* As a team, we need to design and implement an easy to use and simple user interface to facilitate the user’s interaction with the app.
* As a team, we need to handle task failures through retrying, error logging, and automated recovery of the system.

# UML Diagrams:

## Sequence Diagram:

A close-up of a document

Description automatically generated

Figure 3 Sequence Diagram

## Class Diagram:

A diagram of a company

Description automatically generated

Figure 4 Class Diagram

## Component Diagram:

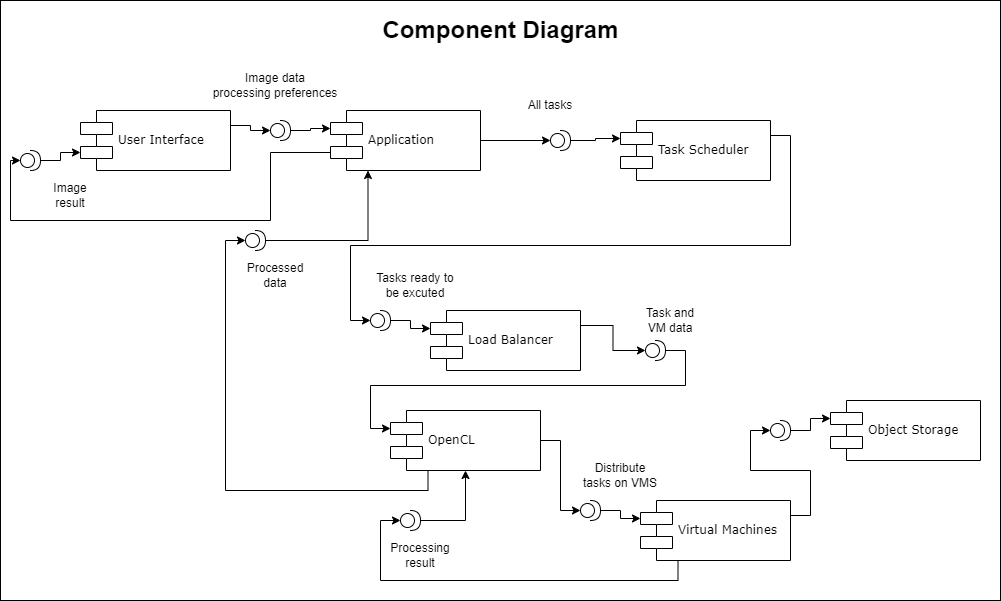


Figure 5 Component Diagram

## Network Diagram:

A computer network diagram with many servers

Description automatically generated with medium confidence

Figure 6 Network Diagram

# Beneficiaries of the project

## Photographers and Graphic Designers

They can edit and improve large quantities of photos for business or personal projects more rapidly thanks to quicker picture processing time.

## E-commerce Platforms

The technology enables online merchants to effectively resize and optimize product photos for their websites, enhancing user experience and perhaps boosting revenue.

## Social Media Platforms

By adding filters, effects, or face recognition capabilities to user-uploaded photographs, social media platforms such as Instagram, Facebook, and Snapchat may leverage this technology to improve user experience and boost engagement.

## Artificial Intelligence and Machine Learning Researchers

By using the system to preprocess and enhance image datasets for training and testing their algorithms, researchers in the domains of computer vision, image processing, and machine learning may accelerate the creation of new technologies and applications.

# Testing scenarios and results

## Time Test

### Scenario

Record the amount of time it takes the system to process a batch of images in different sizes and formats.

### Result

In comparison to sequential processing, the system should efficiently divide the processing duties among multiple nodes, resulting in quicker processing times.

## Scalability Test

### Scenario

Monitor the system's performance under various workload scenarios as you progressively increase the number of concurrent image processing processes.

### Result

In order to manage increasing workloads, the system should be able to dynamically expand resources (such as computational nodes) without going beyond resource constraints and while keeping processing speeds consistent.

## Resource Utilization Test

### Scenario

During the processing of a task monitor the resource utilization across individual nodes and across the system.

### Result

The system should efficiently utilize resources and resource usage should scale with workload.

## User Interface Test

### Scenario

Interact with the user interface to upload images and choose the processing operation.

### Result

The user interface should be responsive and easy to use.

# Monitoring

For monitoring the library logging was used to provide useful, sorted, and time-stamped monitoring logs specifically indicative parts of the code to determine what worked and what didn’t; in addition, it also indicates which cluster handled the request.

## Logs:

1. Ui log: this log was used to provide logs about the request sent by the UI. It’s cleared each time the application is used to avoid redundancy.

A screen shot of a computer

Description automatically generated



1. App.log: this log is on the VMs master1 and master2 and contains a detailed log of steps that indicate if data was sent successfully and to which master node and if any errors/exceptions triggered.

## Merging:

To merge both logs we utilized Azure storage accounts, that we can upload app.log to the storage account each time it’s updated, and then, download it on the host device, so that it can be merged with the ui.log file.

A screenshot of a computer

Description automatically generated

### Utilized function:

A computer screen shot of text

Description automatically generated

### Utilized bash script:

A black screen with red and purple text

Description automatically generated

### Merge function:

A screen shot of a computer program

Description automatically generatedResult:

A screenshot of a computer

Description automatically generated

# End-user guide

The application can be started by running the ui.py file and accessing the website <http://127.0.0.1:5000/> from any web browser.

This will be the initial screen:

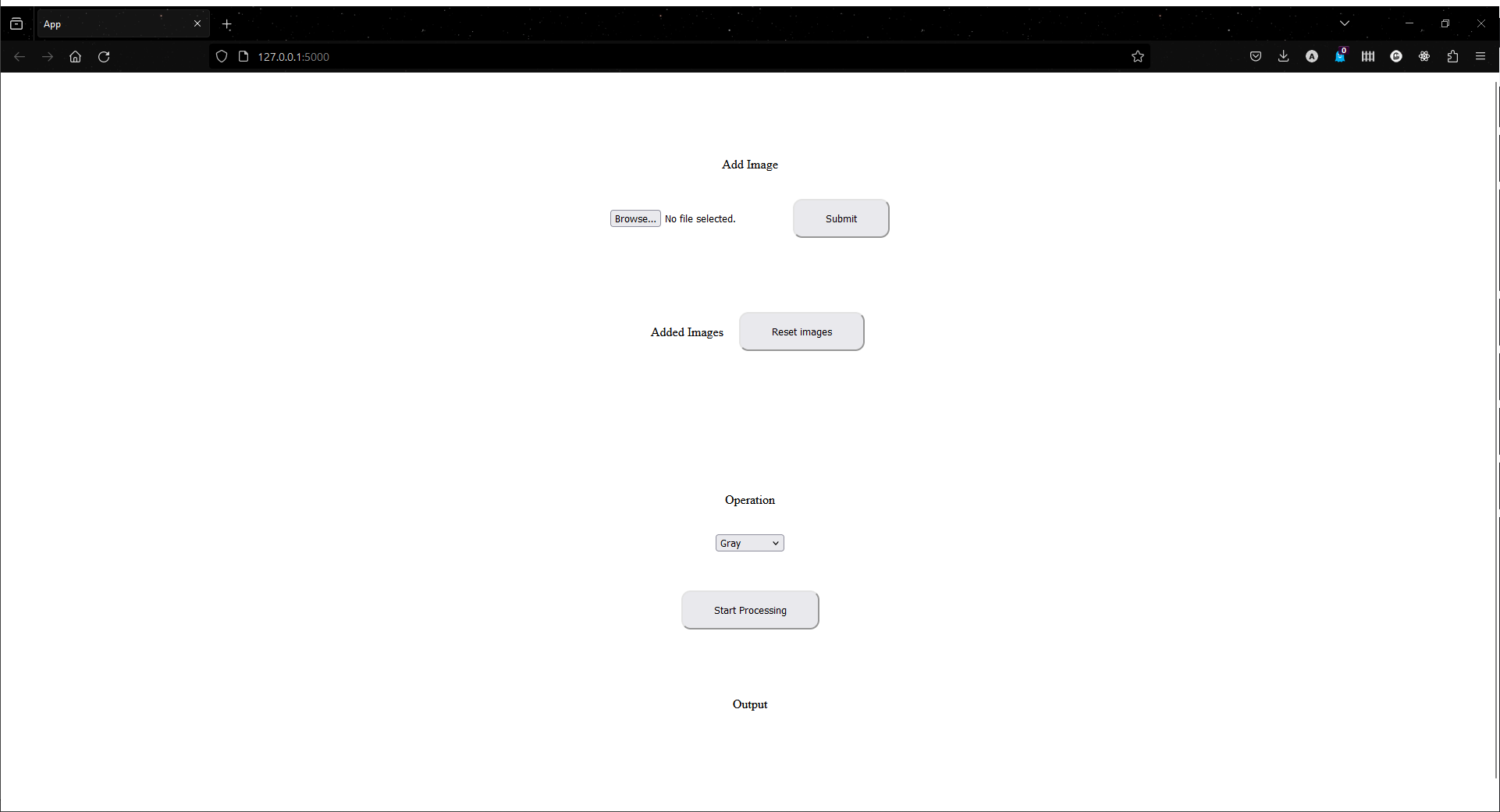


Figure 7 GUI start screen

Click on browse to select an image to be uploaded then click on “Submit” button to upload it.

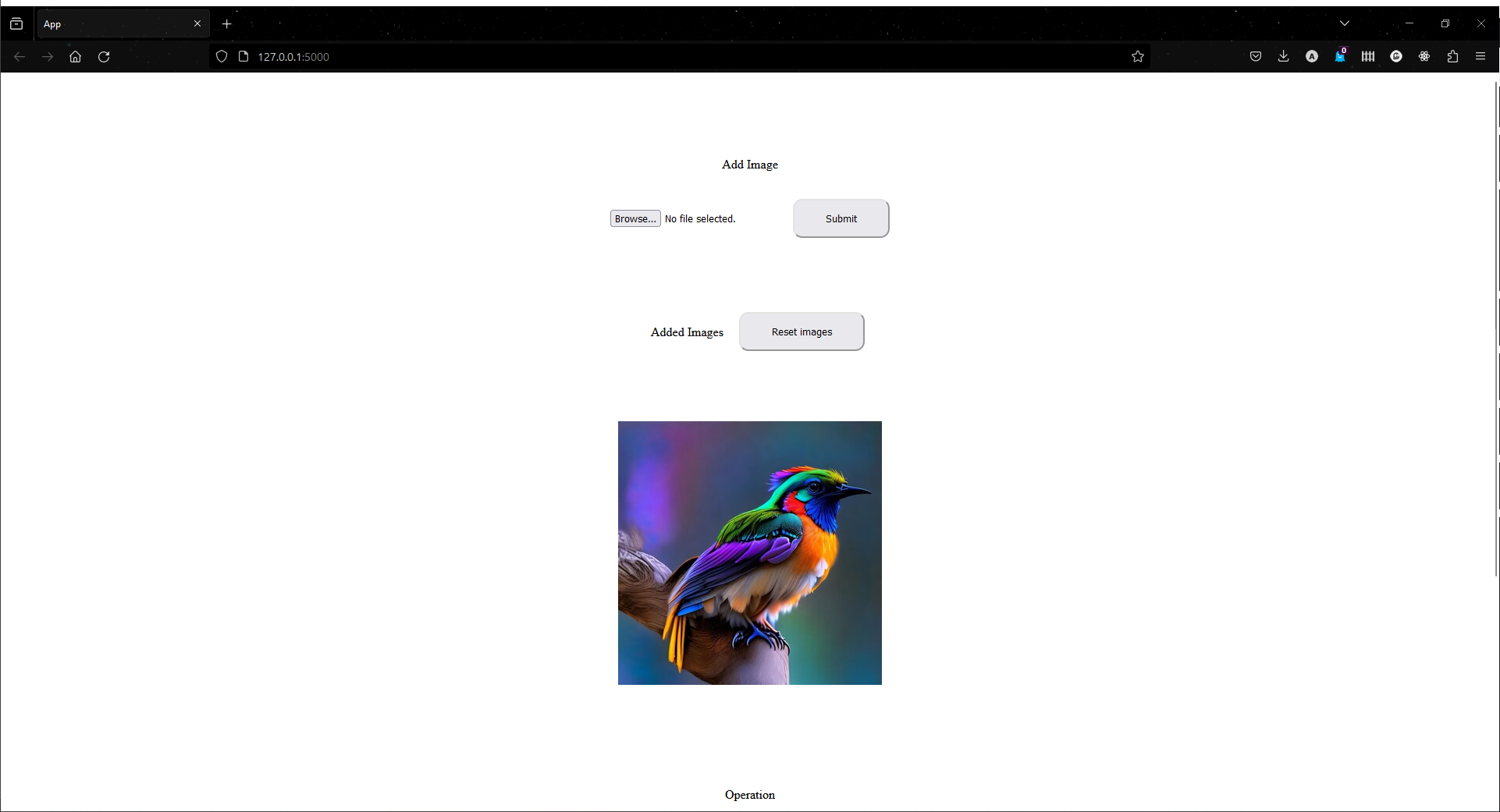


Figure 8 GUI upload image

If you want to reset the uploaded images you can do that by clicking on the “Reset Images” button.

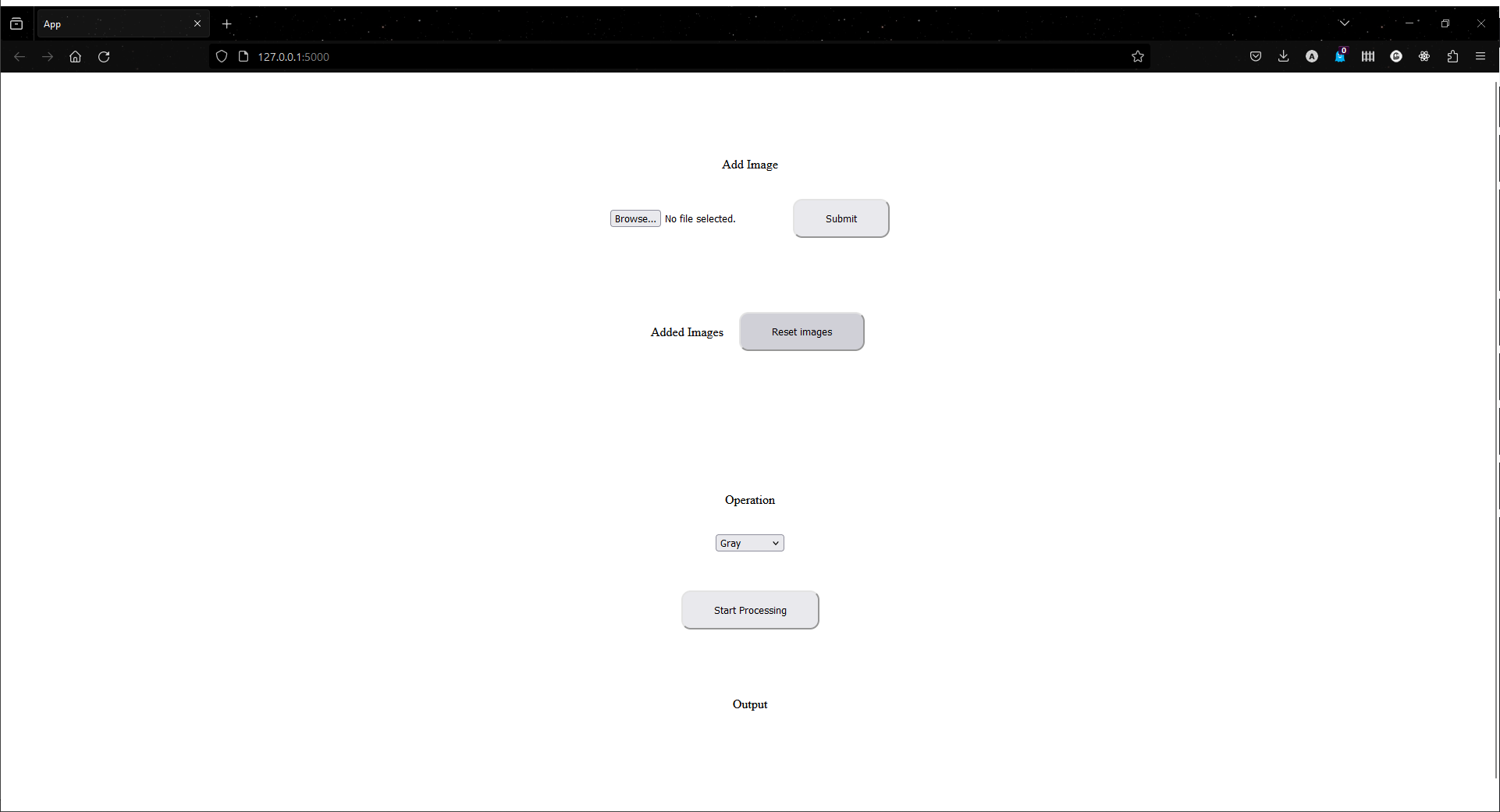


Figure 9 GUI reset images

After uploading an image, you can select the desired operation from the dropdown menu then click on “Start Processing” button to process the image and display the output.

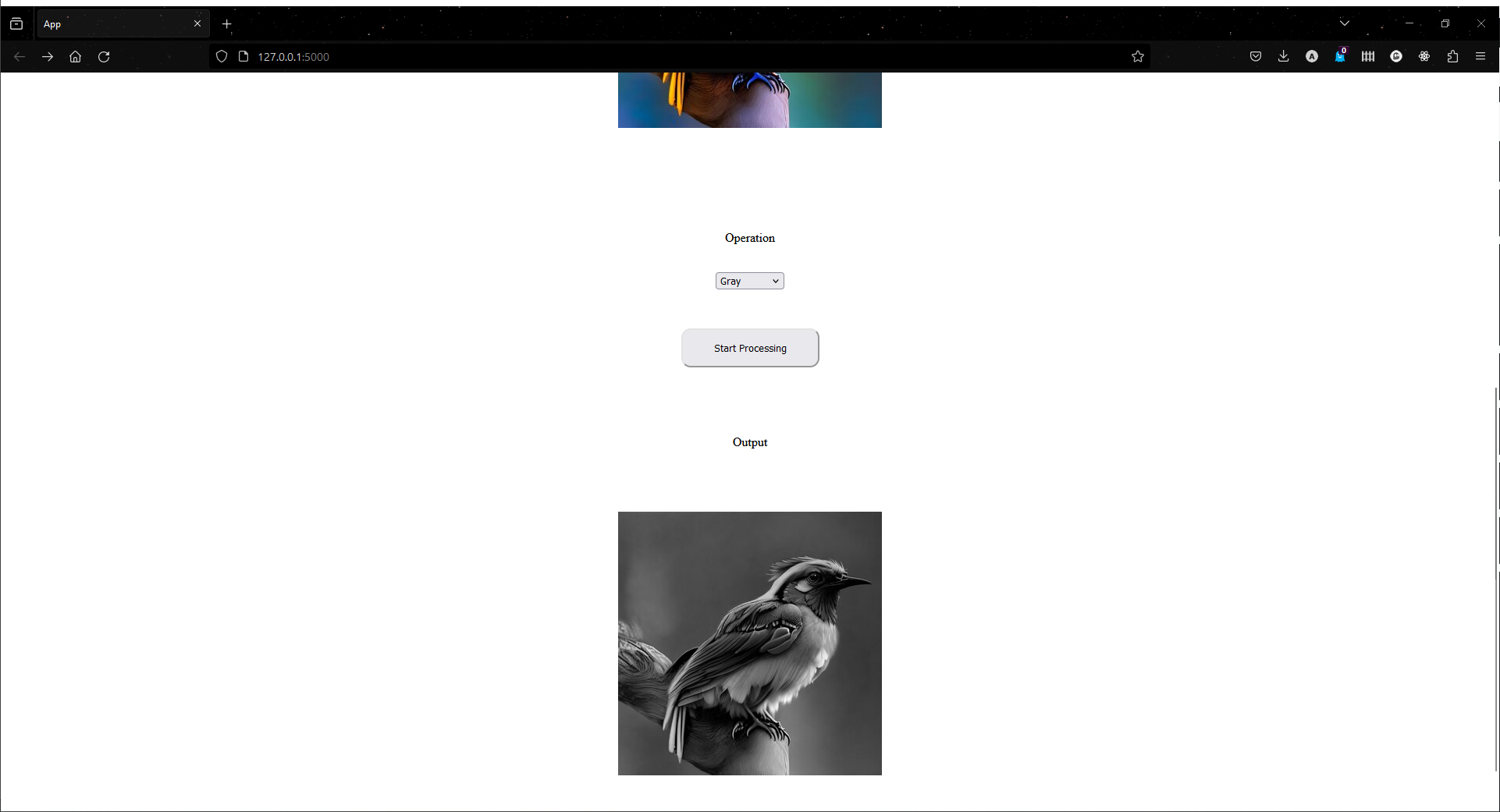


Figure 10 GUI output image 1

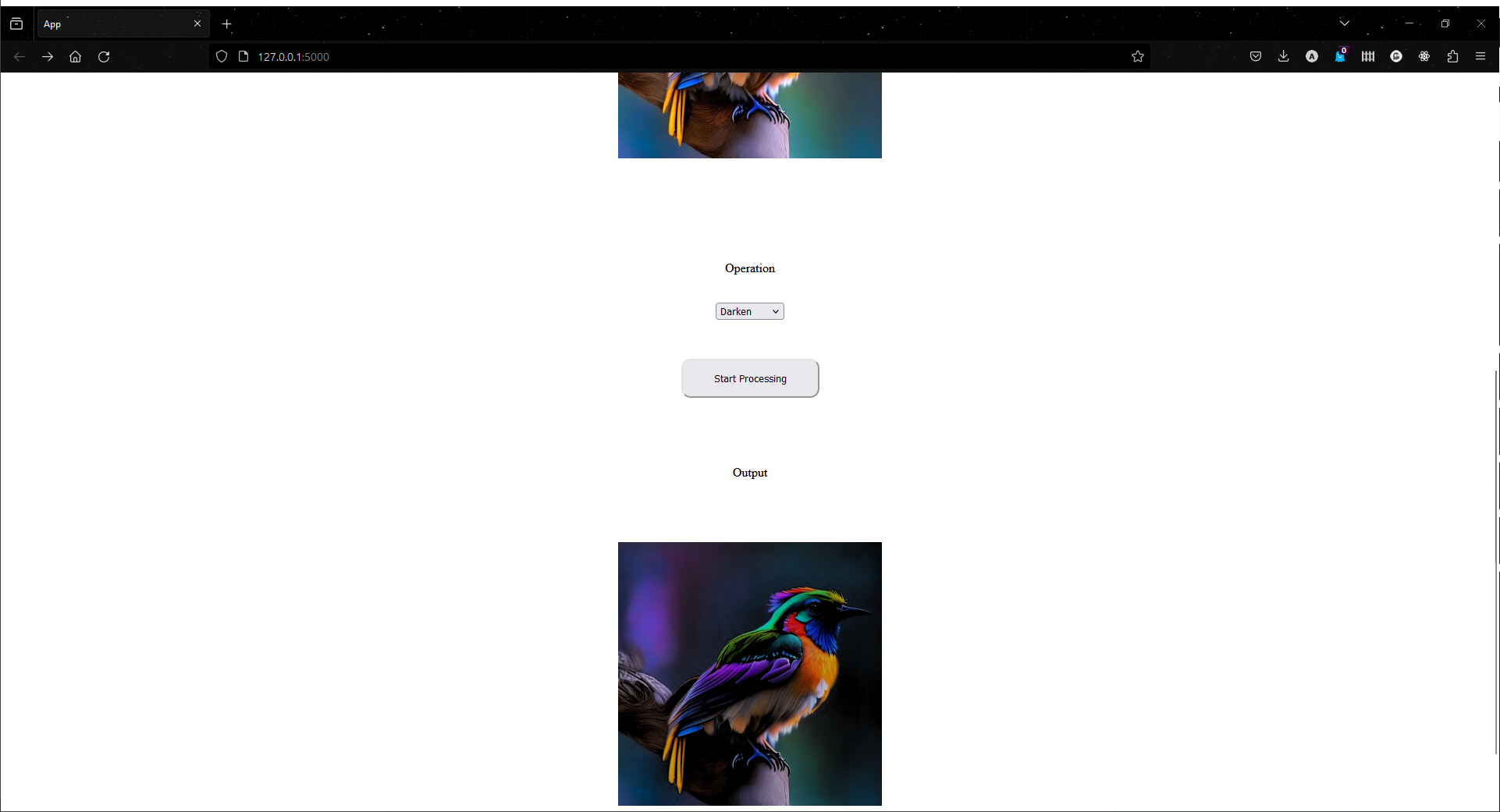


Figure 11 GUI output image 2

# Azure Structure

## Diagram

A diagram of a software

Description automatically generated

Figure 12 Azure Diagram

## Virtual Machines

A screenshot of a computer

Description automatically generated

Figure 13 Azure Virtual Machines

* Virtual Machines have been provided such that the load balancer will determine a master node to be sent the request from the web app, where the task scheduler will divide the processing of an image into three parts in 3 different VMs and a object storage to store the images.

# Distributed Processing Implementation

## System Architecture

The architecture of this distributed system comprises two clusters, each consisting of four virtual machines (VMs) interconnected within the same virtual network. In each cluster, one VM serves as the master node, responsible for delegating tasks to the remaining three VMs.

The clusters are geographically dispersed, with one located in the UK and the other in France. Within each region, a regional load balancer manages the distribution of incoming traffic across the VMs included in its backend pool. The backend pool is a designated set of VMs that are eligible to receive traffic from the load balancer.

A global load balancer, situated in West Europe, oversees the distribution of workload between the regional load balancers, ensuring efficient traffic routing across clusters. Each load balancer is assigned a unique public IP address.

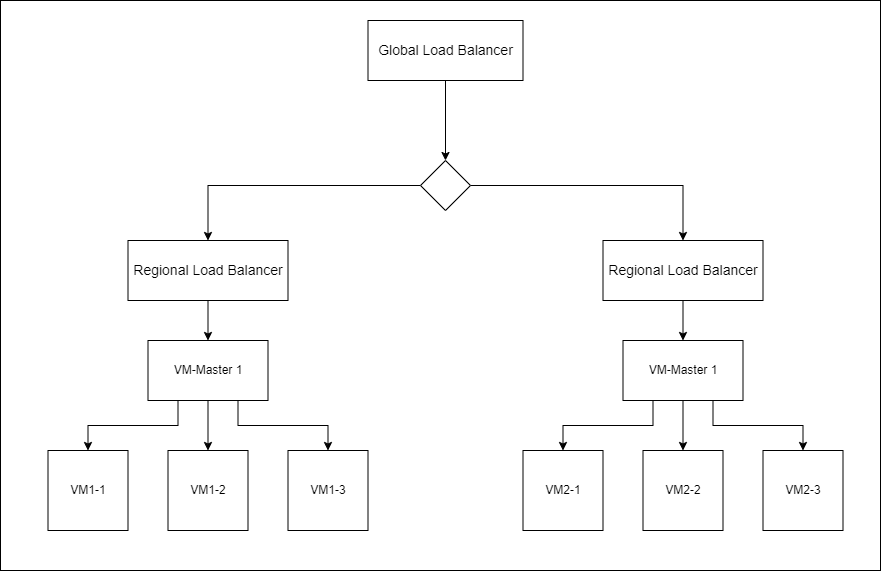


Figure 14 System final architecture

The communication between the virtual machines within this distributed system is facilitated through socket connections. This method of communication extends not only between the VMs themselves but also includes interactions between the end-users and the master VM, which are conducted via a graphical user interface (GUI). The GUI enables users to seamlessly send and receive data to and from the master VM.

To optimize the handling of images, threads are employed. This multithreading approach ensures that the images are processed efficiently across the virtual machines, enabling parallel processing and enhancing the overall performance and responsiveness of the system. By leveraging threads, the system can manage multiple tasks concurrently, thereby improving the speed and reliability of image processing operations within the distributed environment.

## VM health checking

The load balancer in this distributed system is equipped with a sophisticated health probe feature designed to continuously monitor the status of each virtual machine within its backend pool. This health probe is configured to initiate a socket connection with each VM at regular intervals of 20 seconds.

When the health probe attempts to establish a connection, if the VM successfully accepts the connection and responds appropriately, the VM is deemed healthy and capable of handling incoming traffic. Conversely, if the VM fails to respond to the health probe's connection attempt within the specified timeframe, the load balancer will classify that VM as unhealthy. As a result, the load balancer will automatically exclude the unresponsive VM from the traffic distribution process, ensuring that only healthy VMs receive and process incoming requests. This mechanism helps maintain optimal performance and reliability of the system by dynamically managing the availability of resources.

# Fault Tolerance

## Scenario

As elaborated in the previous point, our architecture consists of two clusters or two master nodes with three worker nodes each. If one Master node is offline for any reason, the load balancer will direct the incoming requests to the other master node.

### Result

The system should be able to detect and continue functioning normally if one of the master nodes fails. Utilizing other working master nodes and redistributing the tasks to the remaining resources.

# System Scaling

The system is designed with flexibility in mind, allowing for multiple methods of scaling to enhance performance under varying loads. One approach to scaling involves increasing the number of virtual machines within the same virtual network. By incorporating these additional VMs into the backend pool of the regional load balancer, the system can more effectively manage and distribute the increased workload, thereby boosting overall performance.

Furthermore, the system's architecture supports geographic scalability. This can be achieved by establishing a new cluster of virtual machines in a different geographical location, complete with its own regional load balancer. Once this new load balancer is added to the global load balancer's pool, it can participate in the efficient distribution of traffic alongside the existing regional load balancers. This geographic expansion not only enhances the system's ability to handle higher loads but also improves resilience and availability by distributing the workload across multiple regions.

# Conclusion

In conclusion, the project redefines efficiency, scalability, and dependability by utilizing distributed computing, which is a major advancement in the field of image processing. By adopting cutting-edge technologies and approaches, we have established the foundation for an adaptable platform that can perform a wide range of image processing jobs with unmatched speed and agility. We have proven the viability as well as effectiveness of our strategy via thorough testing and improvement, opening the door for broad adoption and influence across several domains and sectors.

# Demo Video Link

<https://drive.google.com/file/d/1X6Si7Uq1k-ItPayo5FYgOcJ6FYymN0a9/view?usp=sharing>

# GitHub Link

<https://github.com/MHZDN/Distributed-Image-Processing-System-using-Cloud-Computing>

# Table of figures

[Figure 1 System layered architecture 6](#_Toc166958195)

[Figure 2 OpenCL architecture 7](#_Toc166958196)

[Figure 3 Sequence Diagram 8](#_Toc166958197)

[Figure 4 Class Diagram 9](#_Toc166958198)

[Figure 5 Component Diagram 9](#_Toc166958199)

[Figure 6 Network Diagram 10](#_Toc166958200)

[Figure 7 GUI start screen 15](#_Toc166958201)

[Figure 8 GUI upload image 15](#_Toc166958202)

[Figure 9 GUI reset images 16](#_Toc166958203)

[Figure 10 GUI output image 1 16](#_Toc166958204)

[Figure 11 GUI output image 2 17](#_Toc166958205)

[Figure 12 Azure Diagram 18](#_Toc166958206)

[Figure 13 Azure Virtual Machines 18](#_Toc166958207)

[Figure 14 System final architecture 19](#_Toc166958208)