Distributed Image Processing System using Cloud Computing

(Phase 4)

Submitted to:

Prof. Ayman M. Bahaa-Eldin

Eng. Mostafa Ashraf

Submitted by:

Group 1

Mark Bassem	**21P0363**
Mina George	**20P4198**
Mina Morgan	**20P1943**
Rafik Tamer	**20P1046**

Table of Contents

List of Figures

The code will be available at:
https://github.com/MinaMorgan/Distributed-Image-Processing-System-using-Cloud-Computing.git

The Demo video available at: https://youtu.be/OEbBF2-DzoU

or at: drive

Introduction

In the 4^{th} and the final phase of this project, we finally take our **Distributed Image Processing System using Cloud Computing** project to the final level.

Overall, in this project, we were able to develop a Distributed Image Processing System using Cloud Computing technologies. Over the implementation of the four phases, we were able to take advantage of the Amazon's AWS infrastructure to establish a cloud environment and deploy Ubuntu-based VMs where throughout the phases we were able to optimize our VM(s).

We were able to integrate several VMs into a unified private network using AWS's VPC service. Phase by phase, we improved our system by integrating a fault tolerance mechanism which is based on the "ping-pong" concept. We were also able to improve our scalability features to ensure our system is dynamic and can handle any increase in the workload.

By phase 4, we can say that we successfully designed, implemented, and tested a distributed image processing system that effectively distributes tasks among multiple virtual machines, ensuring efficient usage of resources and parallel processing capabilities. We have enhanced the scalability of our system by implementing mechanisms to easily add more virtual machines, so that we have a dynamic system that can adapt to any increase in the workload.

In this phase, we improved one of the advanced image processing operations: the QR code reader. Using the pyzbar library, we enhanced our system to detect and decode QR codes embedded within images. Upon detection, the system extracts the decoded string from the QR code and returns it to the user, enriching the functionality of our distributed image processing system.

Furthermore, by this phase we were able to conduct rigorous testing and validation procedures to evaluate the reliability, stability, and efficiency of our distributed image processing system under diverse usage scenarios and workload conditions, ensuring its robustness and performance.

We even developed a user-friendly GUI allowing users to upload images and track processing progress. This monitoring functionality can be used to identify if any fault occurred during the process.

Detailed project description

In our project we were able to establish a cloud environment using Amazon's AWS, which gave us the possibility to set up several VMs/EC2 instances Ubuntu-based. We successfully optimized VM utilization by assigning distinct roles to each VM. Specifically, we implemented one VM to serve as the "Load Balancer", where this VM is tasked to receive user requests (comprising image data and operations) as an asynchronous HTTP request (to increase the performance) and forward it the dispatcher, the dispatcher divide the received image into several parts based on certain features such as image size, following that, the dispatcher distribute the parts to the other working VMs. The request are made asynchronous to ensure enhanced performance of the system.

Each working VM receives the request and uses MPI-based features, to distribute the workload across multiple threads within the working VM itself. After the request is handled, the working VM will reply with the processed part to the dispatcher and the results are combined and provided back to the user.

To ensure that there is synchronization within the distribution of work among the VMs, we learned about AWS's VPC service and integrated all VMs into the same VPC, enabling them to communicate within a unified private network. We then set up SSH (Secure Shell) by generating SSH key pairs to allow authentication between the VMs and configured the SSH daemon to listen on a specific port and allowed inbound connections.

We also created a security group to control the traffic for the VMs. In the security group we allowed specific protocols, ports, and IP ranges for the VM’s network traffic. Extra configurations were done as we implemented firewall rules to filter network traffic by configuring firewall rules to allow communication between VMs within the same VPC.

Furthermore, we implemented our fault tolerance mechanism, where we make sure there are several working VMs and several backup VMs. This is to ensure our system will continue working in case of failure.

To do so, we need to continuously monitor our VMs status. This was achieved by implementing an asynchronous "Ping-pong" concept, where the Load Balancer periodically sends a "ping" to the working VMs. A VM responds with "pong" to indicate that it’s still working. If a VM fails to respond, the Load Balancer can quickly reroute tasks to the backup VMs, ensuring uninterrupted processing and minimizing downtime. This process is performed asynchronously to enhance performance by allowing tasks to run concurrently without waiting for each other to finish. Asynchronous execution maximizes resource utilization and reduces overall processing time, leading to improved system efficiency.

We were also able to apply and enhance the scalability in our system, as all VMs used were initiated such that once the server (that receives the request) is run on them, these VMs will immediately listen for requests from the dispatcher and execute any task assigned to them, so that we have a dynamic system that can adapt to any increase in the workload.

To ensure the reliability and stability of our system, we conducted comprehensive testing and validation procedures to evaluate its performance, efficiency, and robustness. We rigorously tested various scenarios, including intentionally inducing faults, to assess how our system adapts and tolerates such situations. Additionally, we tested under different workload conditions to ensure consistent performance and robustness.

Finally, we developed a user-friendly GUI to ensure that users can effortlessly upload images and track processing progress. This monitoring functionality enables users to easily identify any faults that may occur during the process.

Beneficiaries of the project

In the final phase of the project, we find ourselves flattered with the many new concepts and features we learned in distributed systems and cloud computing. Initially, we gained insights into setting up the cloud environment and configuring Virtual Machines (EC2 instances) while exploring AWS and its user-centric features.

We extended our understanding of VMs by connecting multiple EC2 instances within the same VPC, assigning a specific VM as a load balancer, and enabling it to interact with other VMs.

Furthermore, we learned more about parallelism, especially the understanding and usage of the Message Passing Interface (MPI) and its role in distributing the workload, which even helped us more in the usage and understanding of threads.

Moreover, we explored fault tolerance mechanisms, by learning how to detect VM failures through an asynchronous "ping-pong" concept, ensuring the resilience of our system, the enhancing of our performance.

We also gained valuable insights into scaling, as we configured our VMs such that once the server is run on them, these VMs are prepared to listen for requests from the dispatcher and promptly execute assigned tasks upon receiving them.

In terms of code, we honed our skills in adapting basic and advanced image processing operations using the OpenCV library, discovering many image manipulation techniques.

Therefore, we can say that from our project's thorough exploration, we adapted a solid understanding of distributed computing and cloud technologies.

Detailed analysis

In the detailed analysis section, we decided to cover 4 main types of analysis which are:

1. **Technical Analysis**:
    - In our project we decided to choose AWS as our cloud service provider, to take advantage of it services such as EC2 for virtual machines, and VPC for networking. Which proved to be highly effective in meeting the project requirements. AWS services provided scalability and reliability, enabling easy deployment and management of virtual machines.
    - We chose python as programming language due to its extensive library support and ease of development and to use the OpenCV library which is essential for image processing tasks.
2. **Performance Analysis**:
    - We continuously improved our system's performance by refining and improving our implementation. By the end we reached in the implementation of an asynchronous system, ensuring better performance.
    - To ensure continuous improvement in performance, we conducted comparisons based on benchmarks such as the time taken to handle multiple requests (images) and the system's ability to handle simultaneous requests from multiple users.
    - By these comparisons we were able to identify areas for enhancement and led to iterative improvements in system responsiveness and efficiency.
3. **Cost Analysis**:
    - We analyzed costs linked to AWS services, focusing on expenses like EC2 instances and VPC features.
    - We controlled our resource usage to mitigate expenses. We did this by ensuring efficient allocation of resources and cost-effectiveness. This proactive approach helped us manage our budget effectively and minimize unnecessary expenses.
4. **Security Analysis**:
    - We implemented security measures such as SSH configuration, security groups, and firewalls to protect against unauthorized access and data breaches.
    - We made sure to follow security standards to keep our computing environment safe.

Task breakdown structure

1. **Project Planning**:
    - We started by defining the project objectives, scope, and deliverables.
    - Based on that we created the project plans, timelines, and workload distribution.
2. **Requirement Analysis**:
    - We then gathered and analyzed the user requirements and system specifications based on the project specification provided to us.
    - We decided what are the functional and non-functional requirements, and prioritized these requirements based on the timeline and submission of the phases.
3. **Design and Architecture**:
    - We designed the expected system architecture, by identifying the needed components, and modules.
    - We created detailed design documents, including diagrams such system architecture and component diagram.
4. **Implementation**:
    - We established a cloud environment using Amazon's AWS, deploying multiple VMs/EC2 instances Ubuntu-based.
    - We optimized VM usage by assigning distinct roles to each VM, including implementing a "Load Balancer" and worker VMs.
    - We configured AWS VPC service to integrate all VMs into the same private network, ensuring communication within the system.
    - We set up SSH for secure authentication between VMs, along with security groups and firewall rules to control traffic.
    - We Implemented a fault tolerance mechanism to ensure system continuity in case of failures, utilizing asynchronous "Ping-pong" concept for continuous monitoring.
    - We applied scalability features to dynamically adapt to workload changes, allowing VMs to listen for requests and execute tasks promptly.
    - We conducted comprehensive testing and validation procedures to evaluate system performance, efficiency, and robustness under various scenarios.

- We developed a user-friendly GUI enabling users to upload images, track processing progress, and identify faults during the process.

5. **Testing**:
    - We conducted systematic testing procedures to evaluate system performance and reliability.
    - We induced intentional faults by closing VMs to test fault tolerance mechanisms.
    - We verified the effectiveness of the asynchronous "ping-pong" mechanism in detecting and rerouting tasks during VM failures.
    - We evaluated the scalability of our system by simulating increased workloads to assess system adaptability and efficiency.
6. **Deployment and Maintenance**:
    - We established the Cloud Environment with AWS and deploying multiple VMs/EC2 instances.
    - We Integrated all VMs into the same VPC to enable communication between the VMs.
    - We set up and configured SSH, security groups, and firewall to allow secure communication between VMs.
    - We initialized the VM by the execution of the VM configuration script
    - We assigned each VM a distinct role for its specific function within the distributed system architecture.
7. **Documentation and Reporting**:
    - We created the project documentation, including system manuals, user guides, and technical documentation.

Questions

1. What were the results of our system testing?

The results of our system testing indicated that our distributed image processing system performed well, particularly in terms of fault tolerance. Where we intentionally induced faults by closing VMs, and our system demonstrated its ability to detect these failures through the asynchronous "ping-pong" mechanism and quickly reroute tasks to backup VMs. Overall, our tests showed that our system could maintain uninterrupted processing even in the face of VM failures, highlighting its robustness and reliability.

Additionally, in terms of scaling our distributed system showed positive results whilst testing. We have created a dynamic environment to adapt to the increase in workload, allowing an efficient handling of any additional tasks (if needed). This scalability feature ensures that our system can effectively manage varying levels of demand without affecting the performance or reliability of our distribution system.

For more detailed information and comprehensive results regarding our system testing, please refer to the dedicated section on Testing Scenarios and results later in the document.

2. What information is included in our system documentation?

Here is the important information we included in our system documentation:

1. **Cloud Environment Setup**:
    - A brief description of how we established a cloud environment using Amazon's AWS and deployment of multiple VMs/EC2 instances.
2. **VMs communication and networking**:
    - A description of how we integrated all VMs into the same VPC to enable communication within a unified private network.
    - The configuration of SSH for secure authentication between VMs.
    - The creation of security groups and implementation of firewall rules to control traffic and allow communication between VMs.
3. **VMs distinct roles**:
    - An explanation of how we assigned distinct roles to each VM, including the implementation of a "Load Balancer" VM and worker VM.

4. **Dispatcher Functionality**:
    - Details on how the dispatcher divides received images into several parts based on specific features such as image size and how the dispatcher distributes image parts to other working VMs for processing.
5. **Fault Tolerance Mechanism**:
    - Description of the fault tolerance mechanism, including the use of the "Ping-pong" concept for continuous monitoring of VMs.
6. **Scalability Features**:
    - The implementation of scalability features allowing VMs to dynamically adapt to workload changes and how VMs immediately listen for requests from the dispatcher and execute assigned tasks upon receiving them.
7. **Testing and Validation Procedures**:
    - The type of testing and validation procedures were conducted to evaluate system performance, efficiency, robustness, and the testing scenarios.
8. **User Interface Development**:
    - The development of a user-friendly GUI enabling users to upload images and track processing progress.
9. **Time Plan**:
    - Inclusion of a time plan detailing project tasks and their timelines, which is represented with a Gantt chart.
10. **System Architecture**:
    - The system architecture, illustrating how components interact and function together.
11. **End-User Guide**:
    - Instructions for end-users on how to effectively use the system, including installation of required libraries and packages.
12. **Outputs**:
    - Presentation of some outputs generated by the system, demonstrating its functionality and performance.

3. How did we deploy our system to the cloud?

1. **Establishing the Cloud Environment with AWS**:
    - We used Amazon's AWS to set up the cloud infrastructure.
    - We deployed multiple VMs/EC2 instances, all running Ubuntu OS.
2. **Integration into VPC**:
    - We used AWS's VPC service to integrate all VMs into the same VPC.
    - This enabled communication between the VMs within a unified private network, ensuring synchronization in the distribution of work among the VMs.
3. **SSH Configuration**:
    - We then set up SSH (Secure Shell) for secure remote access to the VMs.
    - We generated SSH key pairs for authentication between the VMs.
    - We configured the SSH daemon to listen on a specific port and allowed inbound connections.
4. **Security Groups Configuration**:
    - We created a security group to define inbound and outbound traffic rules for the VMs.
    - We specified allowed protocols, ports, and IP ranges for inbound and outbound traffic.
    - We associated security groups with the VM instances to enforce the defined rules.
5. **Firewall Configuration**:
    - We implemented firewall rules to filter network traffic based on predefined criteria.
    - We configured firewall rules to allow communication between VMs within the same VPC.
    - We restricted access to specific ports and services based on security requirements.
6. **Executing the VM Utilization Script**:
    - We initiated the execution of the VM configuration script as detailed in the Developer Guide within this report.
    - This step was crucial to verify that all VMs were initialized and configured correctly, including the installation of required libraries and packages

essential for their intended functions within the distributed image processing system.

7. **VM roles division**:

    - Each VM was assigned a distinct role for its specific function within the distributed system architecture.
    - For the "Dispatcher" VM, we added specialized files and code, such as **DispatcherV3.py**, to facilitate the management and routing of tasks among the working VMs.
    - Meanwhile, the worker VMs were equipped with dedicated files including **WorkerV3.py**, **ImageProcessingFunctions.py**, and **serverV2.py**. These files were essential for implementing the Image processing distributed system's structure and enabling seamless communication between virtual machines.

Time plan

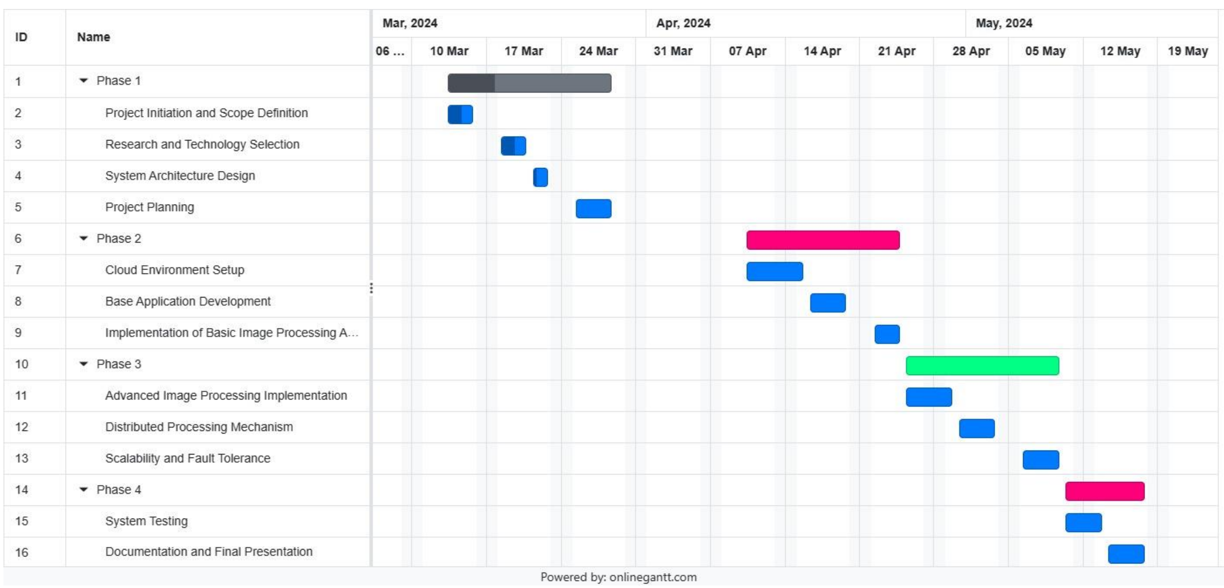

Figure 1: Time plan/ Gantt chart

Phase 1: Project Planning and Design

Objective: Establish a clear understanding of the project scope, design the system architecture, and create a detailed project execution plan.

Duration: 2 weeks

Delivery Date: 28 March 2024

Task 1: Project Initiation and Scope Definition

Responsibility: Entire Team

Timeline: Week 1, Days 1-2

Deliverables: Project scope, objectives, and requirements.

Task 2: Research and Technology Selection

Responsibility: Members 1, 2

Timeline: Week 1, Days 5-7

Deliverables: Technologies selected with justification (Python, AWS, MPI).

Task 3: System Architecture Design

Responsibility: Members 2, 4

Timeline: Week 2, Days 1-2

Deliverables: Detailed system architecture diagram and component design.

Task 4: Project Planning

Responsibility: Members 1, 3

Timeline: Week 2, Days 3-6

Deliverables: Detailed project plan with responsibilities and timelines for each phase.

Phase 2: Development and Environment Setup

Objective: Implement the system components, set up the cloud environment, and integrate the image processing algorithms.

Duration: 3 weeks

Delivery Date: 25 April 2024

Task 1: Cloud Environment Setup

Responsibility: Entire Team

Timeline: Week 4

Deliverables: Configured AWS environment (EC2).

Task 2: Base Application Development

Responsibility: Members 1, 3

Timeline: Week 5

Deliverables: Core application and Worker thread.

Task 3: Implementation of Basic Image Processing Algorithms

Responsibility: Members 2, 4

Timeline: Week 5

Deliverables: Image processing operations (edge detection, color manipulation).

Phase 3: Enhancement and Integration (2 weeks)

Objective: Expand the system's capabilities by adding advanced image processing features, integrating distributed processing mechanisms, and ensuring the system can scale and recover from failures efficiently.

Duration: 2 weeks

Delivery Date: 9 May 2024

Task 1: Advanced Image Processing Implementation

Responsibility: Members 1, 3

Timeline: Week 6

Deliverables: Advanced image processing operations integrated and tested within the system.

Task 2: Distributed Processing Mechanism

Responsibility: Entire Team

Timeline: Week 6, 7

Deliverables: Functional distributed processing using MPI, with tasks efficiently distributed across VMs.

Task 3: Scalability and Fault Tolerance

Responsibility: Members 2, 4

Timeline: Week 7

Deliverables: Auto-scaling setup for VMs and fault tolerance mechanisms implemented, ensuring system flexibility.

Phase 4: Testing, Optimization, and Documentation (1 weeks)

Objective: Ensure the system's functionality and performance through testing, optimize for better performance, and document the system.

Duration: 1 week

Delivery Date: 16 May 2024

Task 1: System Testing

Responsibility: Member 1,3

Timeline: Week 8

Deliverables: Test cases, bug reports, and system validation results.

Task 2: Documentation and Final Presentation

Responsibility: Members 2, 4

Timeline: Week 8

Deliverables: Project report, system documentation, and final presentation.

System architecture and design

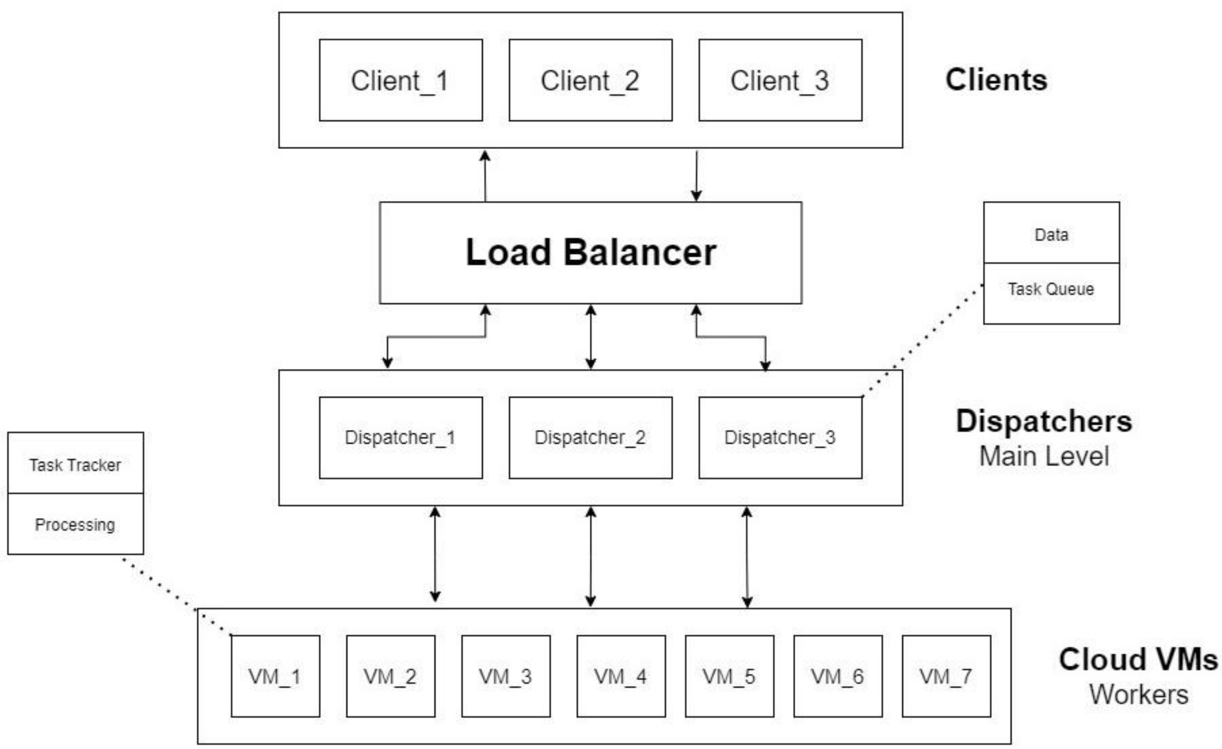

Figure 2: System architecture

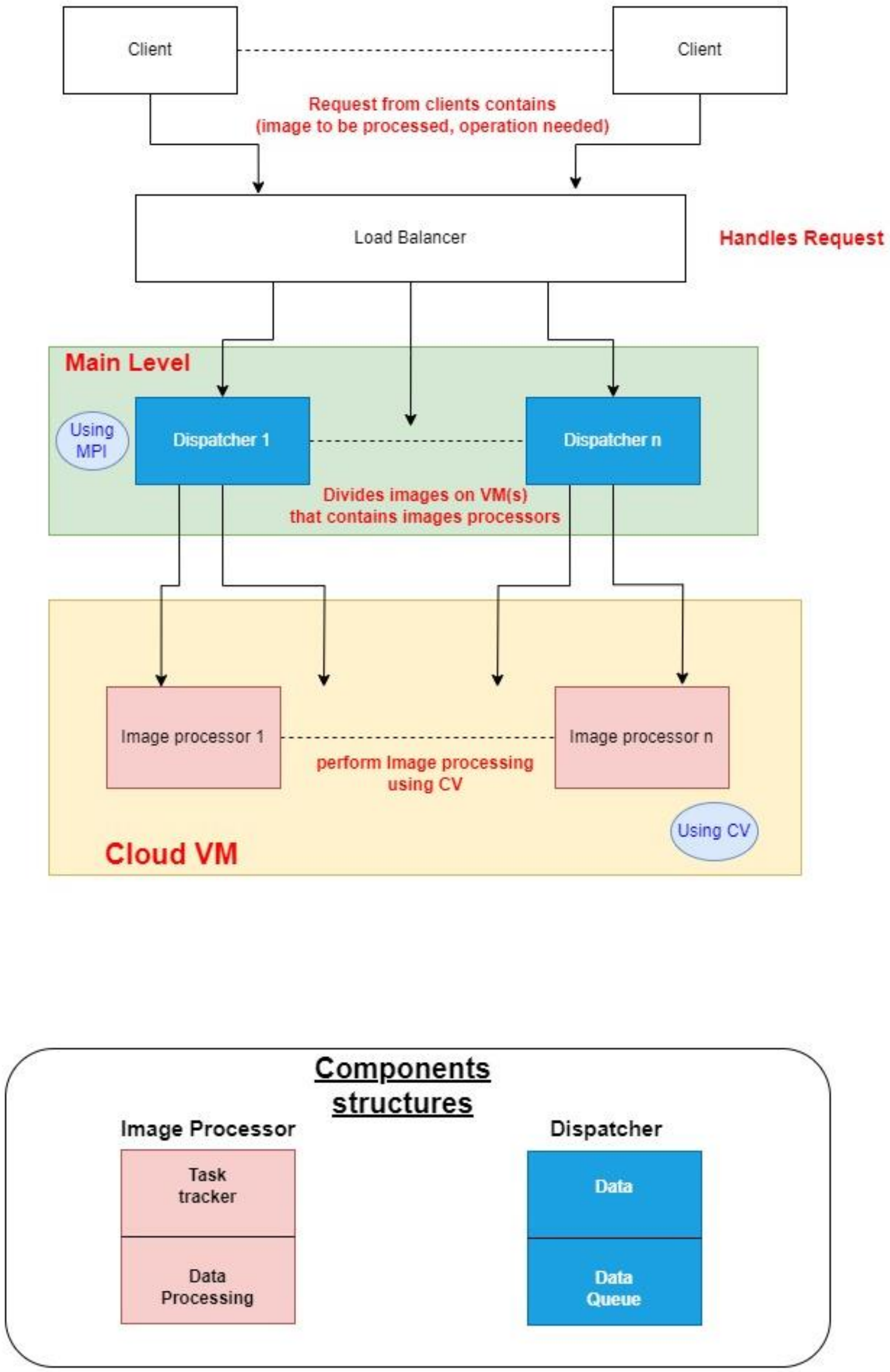

Figure 3: Detailed System architecture

Testing Scenarios and results

In our testing phase, we made sure to maintain the accuracy and integrity of the data obtained. We conducted thorough checks to verify that the images processed by our system were free from any corruption.

As mentioned earlier, the results of our system testing indicated that our distributed image processing system performed well. We conducted two primary types of testing:

1) **Fault tolerance:** To verify the fault tolerance of our system, we deliberately induced faults by closing VMs. This allowed us to observe how our system would respond to such failures. Leveraging our asynchronous fault-tolerance mechanism, known as the "ping-pong" system, tasks were swiftly rerouted to backup VMs. This ensured uninterrupted processing, as the system seamlessly resumed operations from the point of failure, without the need to restart procedures. This indicated that our system could maintain uninterrupted processing even in the face of VM failures, highlighting its robustness and reliability.

2) **Increase in the workload and scalability:** The other type of testing we focus on is how our system handles an increase in workload. To do so we created different scenarios:

1. For the first scenario:
    - We have one user send 5 images to be processed on our system
    - **Average image size:** 60KB
    - **Total data sent**: 5 images * 60 KB/image = 300 KB
    - **Processing time**: 10 seconds
    - **Average processing time per image**: 10 seconds / 5 images = 2 seconds/image
    - **Average data send per processing time**: 300 KB / 10 seconds = 30 KB/s
2. For the second scenario:
    - We have three users sending 5 images/each to be processed on our system
    - **Average image size:** 60KB
    - **Total data sent**: 15 pictures * 60 KB/image = 900 KB
    - **Processing time**: 50 seconds
    - **Average processing time per image**: 50 seconds / 15 images ≈ 3.33 seconds/image
    - **Average data send per processing time**: 900 KB / 15 seconds = 18 KB/s

3. For the third scenario:

    - We have several users sending 20 images total to be processed on our system
    - **Average image size:** 70KB
    - **Total data sent**: 20 images * 70 KB/image = 1050 KB
    - **Processing time**: 73 seconds
    - **Average processing time per image**: 73 seconds / 20 images ≈ 3.65 seconds/image
    - **Average data send per processing time**: 1050 KB / 73 seconds ≈ 13.39 KB/s

4. For the fourth scenario:

    - We have several users sending 25 total images to be processed on our system
    - **Average image size:** 80KB
    - **Total data sent**: 25 pictures * 80 KB/image = 1200 KB
    - **Processing time**: 94 seconds
    - **Average processing time per image**: 94 seconds / 25 images ≈ 3.76 seconds/image
    - **Average data send per processing time**: 1200 KB / 94 seconds ≈ 12.76 KB/s

Table 1: Scenario Comparison Table

	Average image size (KB)	Total Images	Total data sent (KB)	Processing time (s)	Avg processing time per image (s/image)	Avg data sent per processing time (KB/s)
Scenario 1	60	5	300	10	2	30
Scenario 2	60	15	900	50	3.333333333	18
Scenario 3	70	20	1050	73	3.65	14.38356164
Scenario 4	80	25	1200	94	3.76	12.76595745

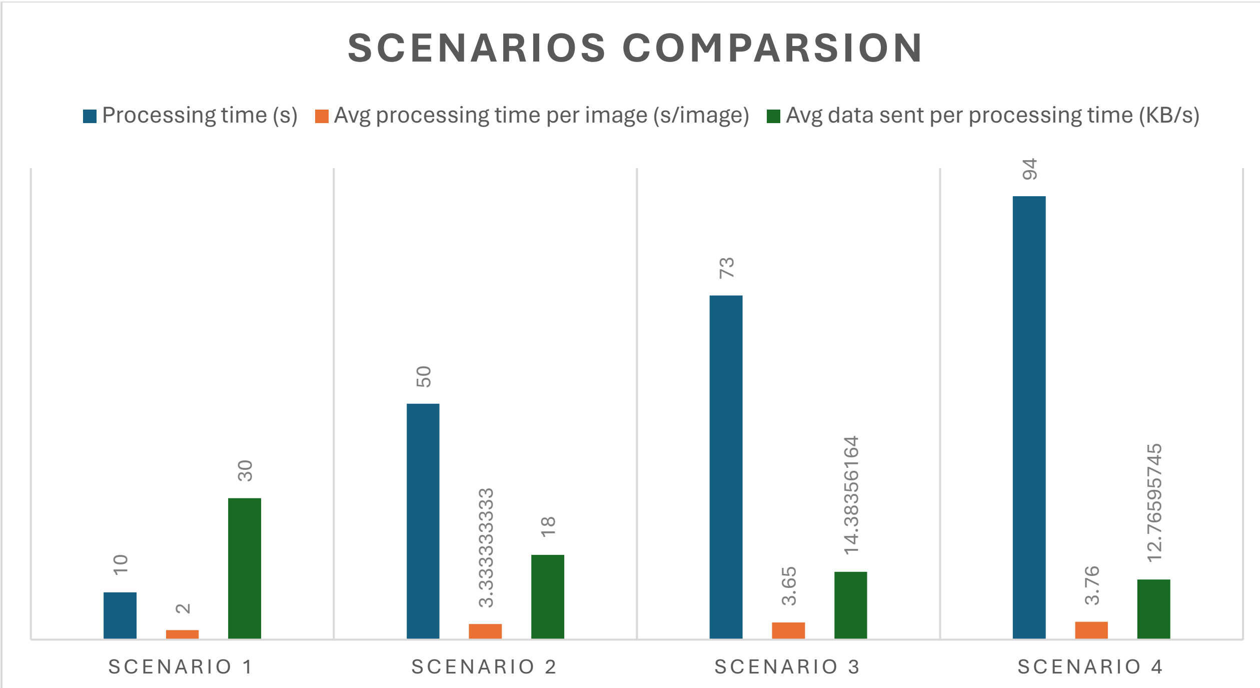

Figure 4: Scenarios Comparison Chart 1

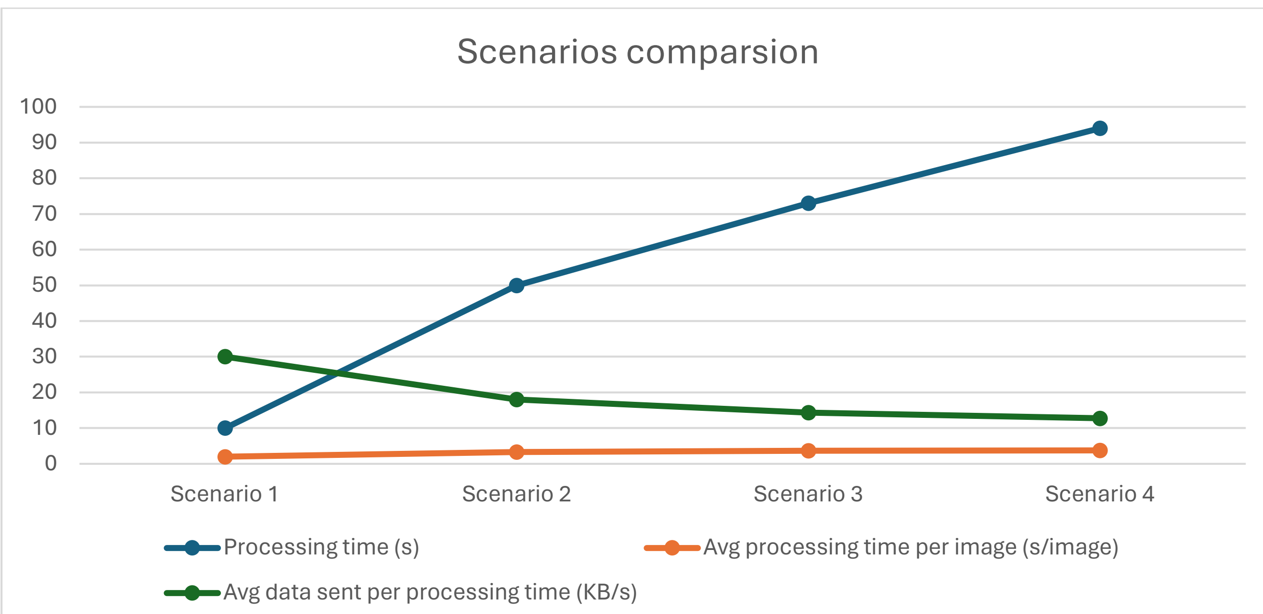

Figure 5: Scenario Comparison Chart 2

Now, let's analyze the scenarios presented:

- **Increase in Data Volume:** With an increase in the number of requests, there's a corresponding increase in the total amount of data to be processed. This means the system must handle larger data volumes as workload increases.

- **Processing Time Efficiency:** Despite the rise in data volume, the processing time may not scale proportionally due to concurrent processing. This indicates that the system can efficiently manage multiple requests simultaneously without a significant linear increase in processing time.
- **Impact on Processing Time per Image:** While the overall processing time may remain reasonable, the processing time per image might slightly increase with a larger number of requests. This is because more resources are allocated to handle the increased workload, resulting in a marginally longer processing time per image.
- **Effect of Image Size:** Smaller images are processed faster compared to larger ones, which is expected. However, the system aims to ensure equitable delivery of images to users, maintaining reasonable processing times regardless of image size.

In summary, despite variations in the number and size of images processed, the system strives to maintain efficient processing times and consistent delivery of images to users. The concurrent processing capability allows the system to handle increased workloads without significantly impacting performance, ensuring a seamless user experience.

This indicates our system allows an efficient handling of any additional tasks. This scalability feature ensures that our system can effectively manage varying levels of demand without affecting the performance or reliability of our distribution system.

Conclusion

By the final phase of our Distributed Image Processing System using Cloud Computing project, we have achieved significant milestones in the understanding and implementation of distributed computing, cloud technology, and image processing.

Using Amazon's AWS infrastructure allowed us to learn and implement new concepts and services of distributed computing and cloud technology. Throughout the four development phases, we improved and polished our system. This process helped us to understand more about cloud computing and optimize our system for efficiency and scalability.

We even learned to implement other concept aside cloud technology such as the integration of fault tolerance mechanisms and scalability features, which resulted in each phase bringing us closer to our goal of creating an efficient, reliable, and dynamic system.

Our rigorous testing and validation procedures validated the robustness, stability, and efficiency of our system under certain scenarios and workload conditions. Some scenarios included intentional fault induction and scalability testing, we demonstrated the system's ability to adapt and perform consistently, regardless of challenges or increased demands.

Furthermore, the development of a user-friendly GUI made it easier for users to effortlessly upload images, track their processing progress, and identify any faults that may occur during the process. This monitoring functionality enhances user experience and system transparency, ensuring seamless interaction and troubleshooting capabilities.

In conclusion, our Distributed Image Processing System reflects our hard work and learning during this project. It combines new technologies, smart solutions, and careful work. As we move forward, we take with us the lessons and successes from this project, setting the stage for more exploration and innovation in distributed computing and cloud technologies.

Output

Example of Image Processing Features

Figure 6: Input image 1

Figure 7: Output Image 1 (After performing Inversion)

Figure 8: Input image 2 (QR code)

Figure 9: Output image 2 (QR Code)

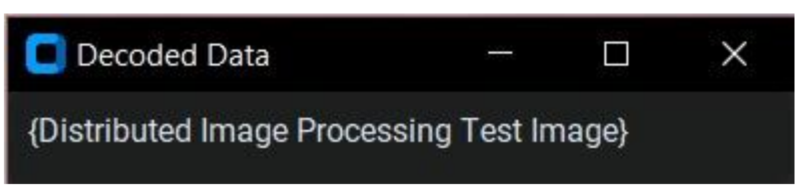

Figure 10:Output image 2 (Decoded String)

Images processed after distributing them

Figure 11: Input image 3

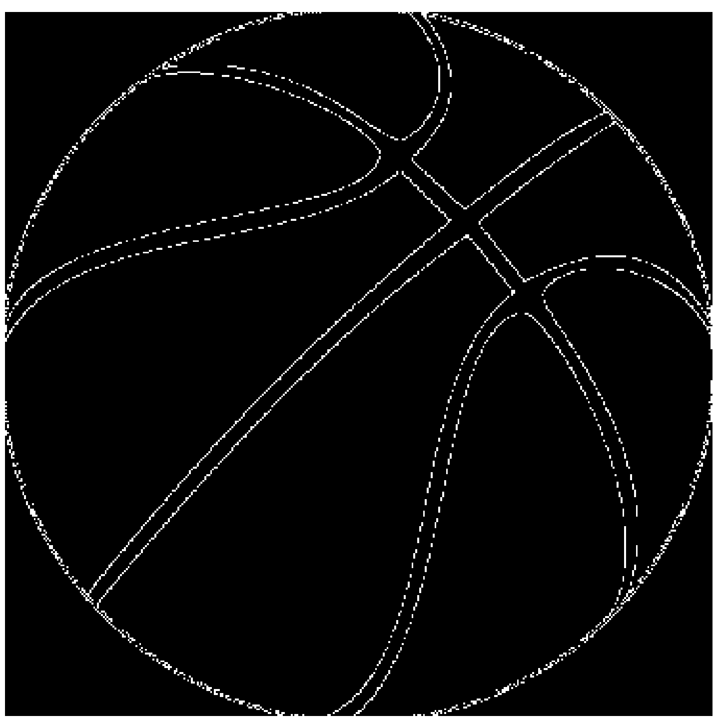

Figure 12:Output image 3 (after image distribution & edge detection)

Client GUI

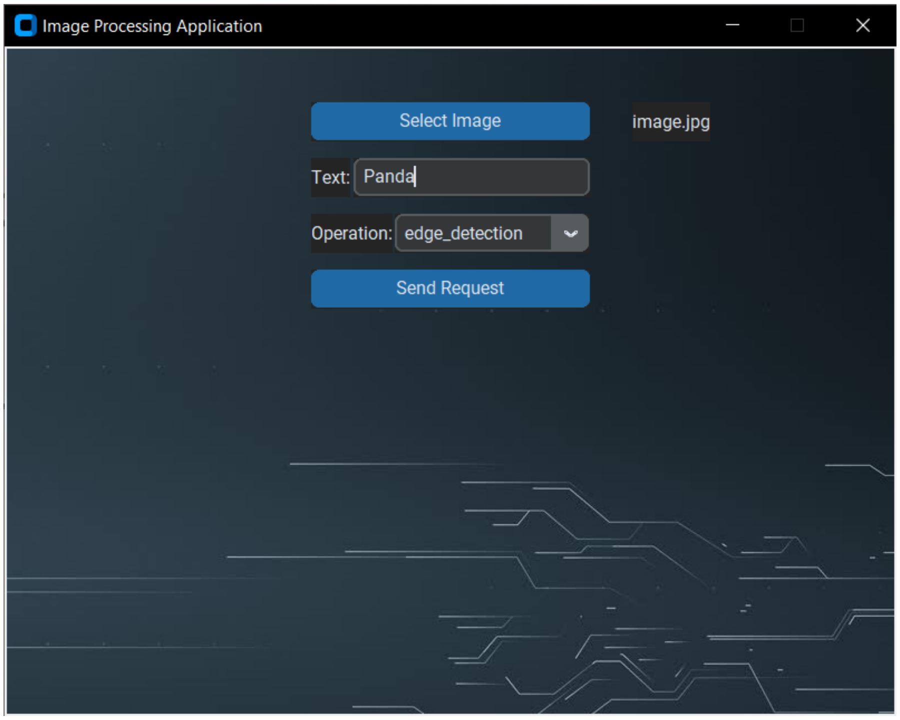

Figure 13:Client GUI

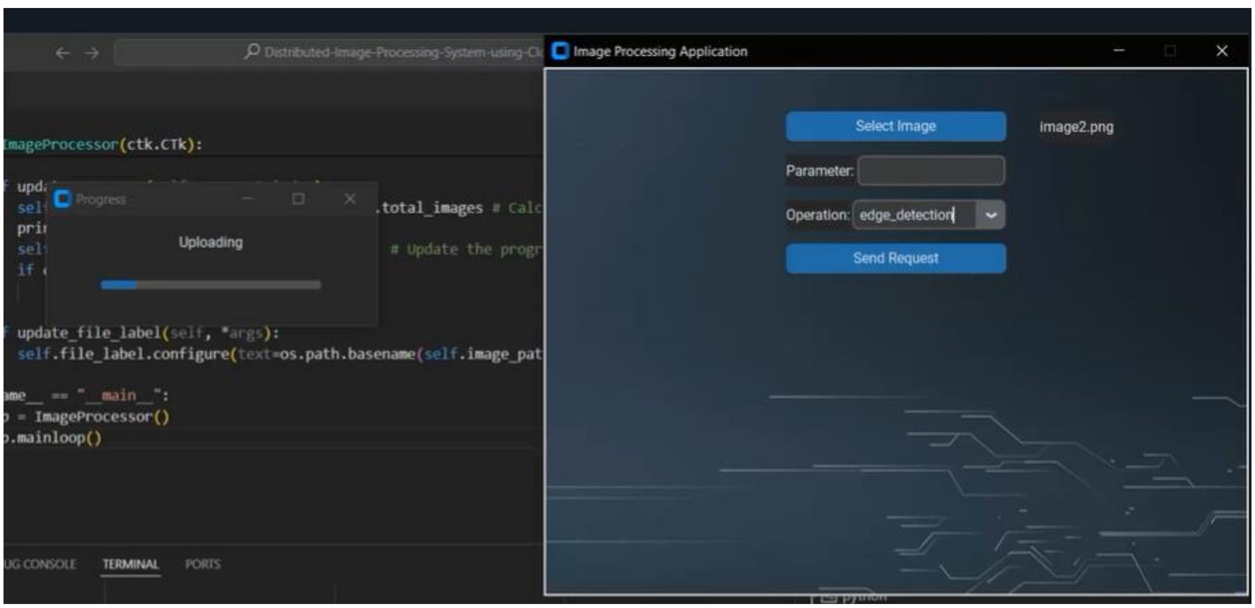

Figure 14: Client GUI with process tracking

End-user Guide

1. **Download and Setup**

    - Download all files provided in the GitHub repository.
    - Install any necessary libraries required for running the system.

```
git clone https://github.com/MinaMorgan/Distributed-Image-Processing-System-using-Cloud-Computing

cd Distributed-Image-Processing-System-using-Cloud-Computing

pip install requests customtkinter Pillow numpy
```

2. **Run the Application**

    - Run the **Send.py** script to start the application.

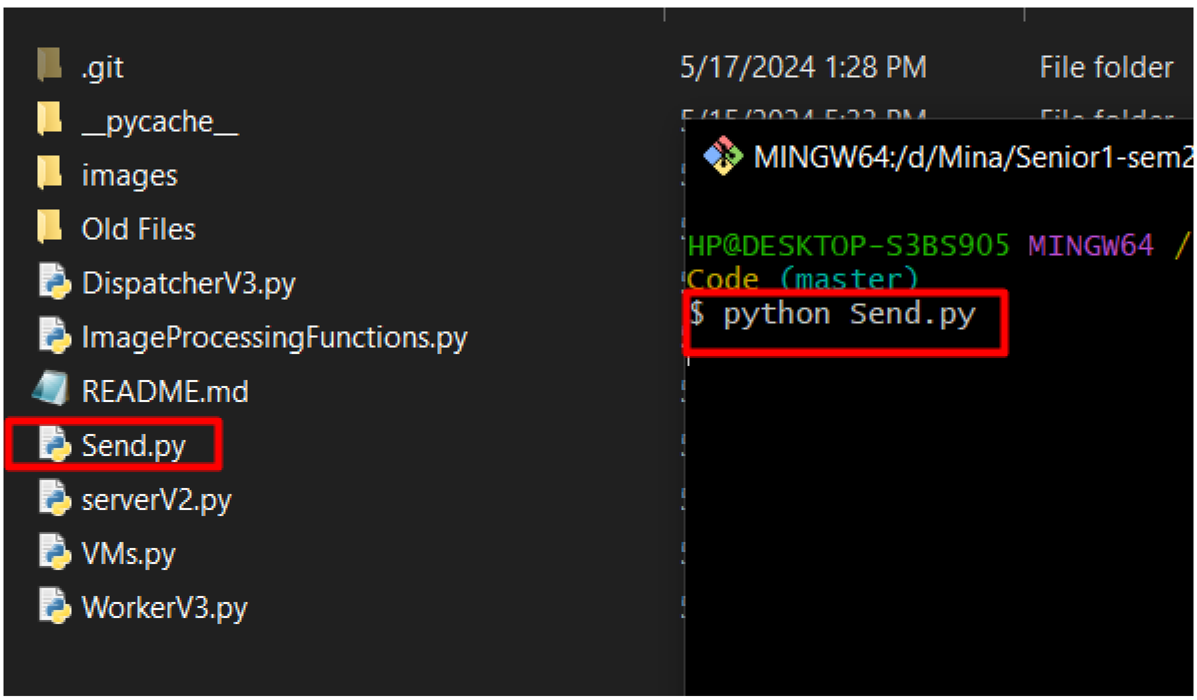

Figure 15: Running Send.py

3. **Upload Images**

- Select the image(s) that you want to process.

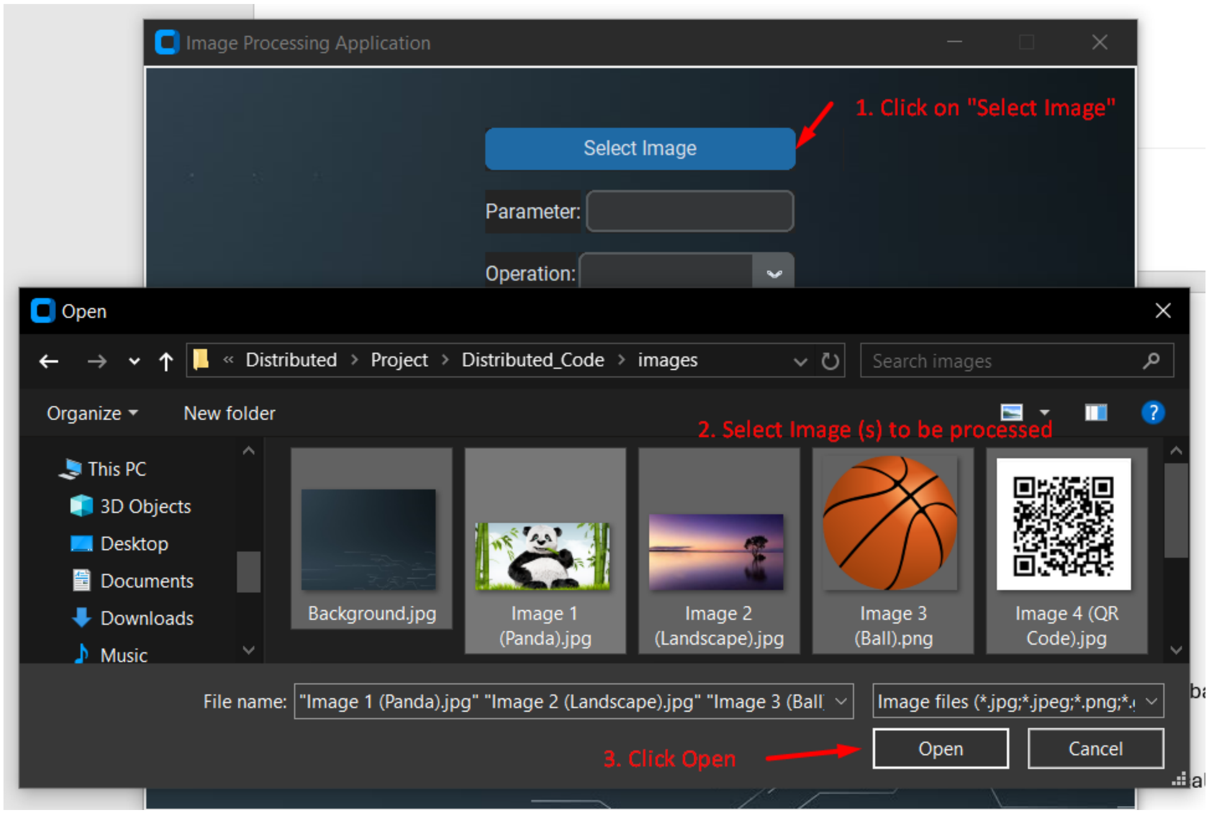

Figure 16: uploading image(s)

4. **Select Operations**

- Choose the desired image processing operation(s) from the available options.
- For some operations, parameter is used to specify the mask size (if needed).

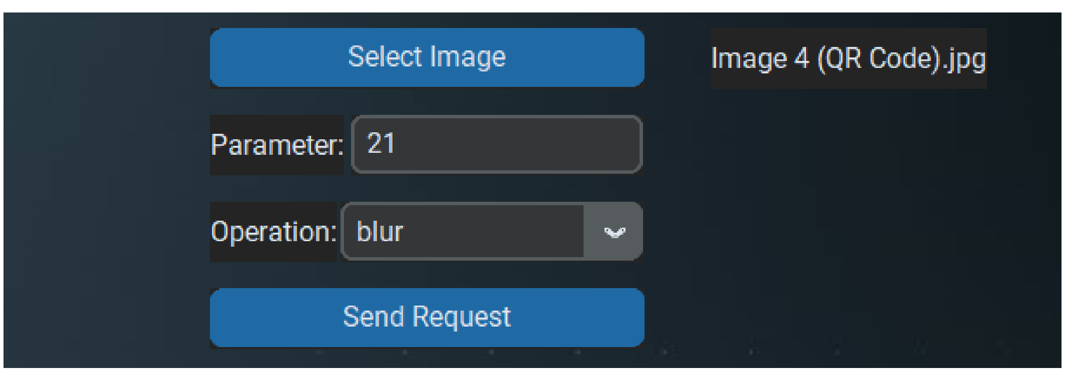

Figure 17: Choosing parameter and Operation

5. **Send Request**

- Click on the "Send Request" button to initiate the processing.

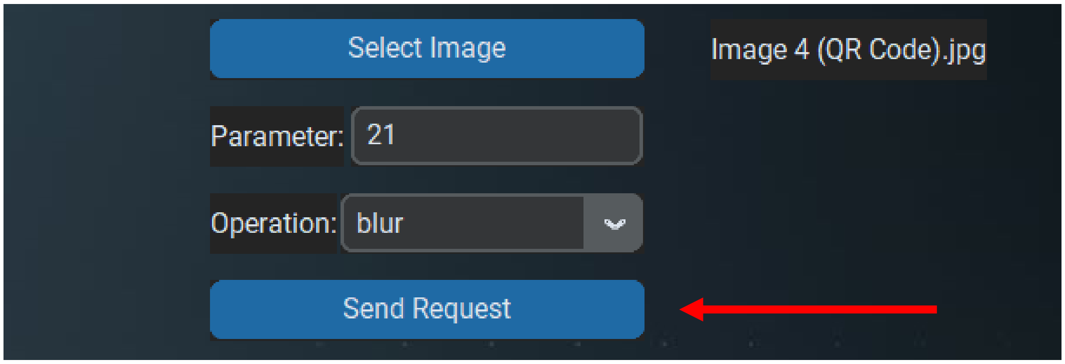

Figure 18: Send request

6. **Track Processing Progress**

- Monitor the progress of image processing using the provided progress bar.

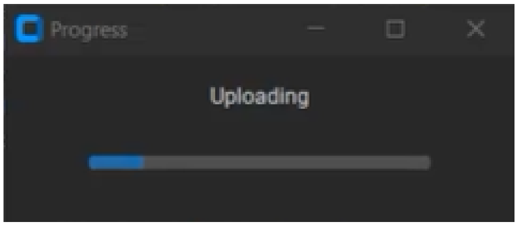

Figure 19: uploading

7. **Download Processed Images**

- Once processing is complete, the processed images will be automatically downloaded to your device.

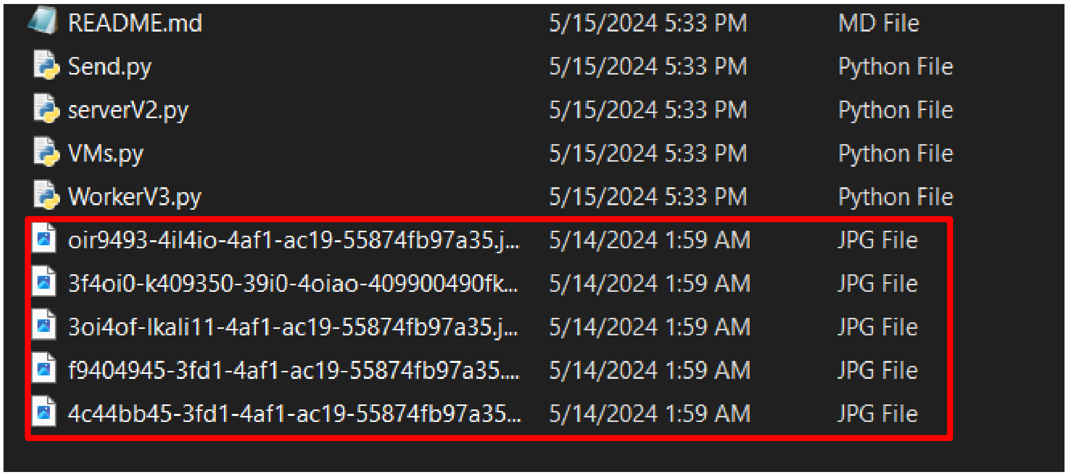

Figure 20: Downloaded image

Developer Installation Guide

1. **Create AWS EC2 Instances:**

Set up EC2 instances on AWS, including one Load Balancer and at least two worker VMs.

2. **Ensure Same VPC Configuration:**

Ensure that all instances are configured within the same Virtual Private Cloud (VPC) to enable communication.

3. **Configure Security Groups:**

Configure the security groups associated with the instances to allow inbound and outbound connections as per project requirements.

4. **Start VMs:**

Start all the VMs to initiate the setup process.

5. **Run Setup Script:**

Execute the following setup script on all VMs to prepare the environment:

```
sudo apt update
sudo apt install software-properties-common -y
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.10 python3.10-venv python3.10-dev
python3 --version

ls -la /usr/bin/python3
sudo rm /usr/bin/python3
sudo ln -s python3.10 /usr/bin/python3
python3 --version

sudo apt update
python3 -m venv env
source env/bin/activate
sudo apt install python3-pip
sudo apt install python3-mpi4py
sudo apt install python3-opencv
sudo apt install python3-numpy
pip install numpy
pip install fastapi

sudo apt update
```

```
sudo apt install libzbar0
pip install pyzbar

ssh-keygen -t rsa
cd /home/ubuntu/.ssh/
cat ~/.ssh/id_rsa.pub

nano authorized_keys
```

6. **Create Hostfile:**

Create a file named **my_hostfile.txt** on all VMs with the following content:

```
Localhost slots=2
```

7. **Copy Files to Load Balancer VM:**

Copy the files **DispatcherV3.py** and **VMs.py** to the Load Balancer VM.

8. **Edit IP Addresses in VMs.py:**

Update the IP addresses in **VMs.py** with the private IPs of the worker VMs.

9. **Run Load Balancer:**

Execute the following command on the Load Balancer VM to run the Dispatcher:

```
mpirun --hostfile my_hostfile.txt -np 2 python3 DispatcherV2.py
```

10. **Copy Files to Worker VMs:**

Copy the files **serverV2.py**, **WorkerV3.py**, and **ImageProcessingFunctions.py** to the worker VMs.

11. **Run Worker VMs:**

Execute the following command on each worker VM to run the server:

```
python3 serverV2.py
```

Code

Client Side (send.py)

```python
import requests
import customtkinter as ctk
from tkinter import Canvas, filedialog, messagebox
from PIL import Image, ImageTk
import os
import numpy as np
import time
import uuid
import threading


ctk.set_appearance_mode("Dark")
ctk.set_default_color_theme("blue")


def show_decoded_data(title, message):
    # Create a top-level window
    window = ctk.CTkToplevel()
    window.title(title)

    # Create a text widget to display the message
    text_widget = ctk.CTkTextbox(window, wrap='word', height=10, width=40)
    text_widget.insert('1.0', message)
    text_widget.pack(expand=True, fill='both')

    # Resize the window to fit the content
    window.update_idletasks()  # Update the window to calculate its size
    width = window.winfo_reqwidth() + 20  # Add some padding
    height = window.winfo_reqheight() + 20  # Add some padding
    window.geometry(f"{width}x{height}")


# def plot_image(data):
#     image_array = np.array(data)
#     plt.figure()
#     plt.imshow(image_array, cmap='gray', interpolation='nearest')
#     plt.title("Result")
#     plt.axis('off')
#     plt.show()
def save_image(data, filename_prefix="image"):
    timestamp = int(time.time())
    filename = f"{filename_prefix}_{timestamp}.jpg"
```

```python
    image_array = np.array(data, dtype=np.uint8)
    image = Image.fromarray(image_array)
    image.save(filename)
    return filename

def open_image(filename):
    # Open the image using the default image viewer
    image = Image.open(filename)
    image.show()

def plot_image(data):
    # Generate a unique filename using UUID
    unique_filename = str(uuid.uuid4()) + ".jpg"

    # Check if the filename already exists
    while os.path.exists(unique_filename):
        unique_filename = str(uuid.uuid4()) + ".jpg"

    # Save the image with the unique filename
    filename = save_image(data, unique_filename)
    #open_image(filename)


class ImageProcessor(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.title("Image Processing Application")
        self.geometry("640x480")
        self.resizable(False, False)

        self.load_background()

        self.create_widgets()

    def load_background(self):
        # Load the background image
        image_path = "images/Background.jpg"
        original_image = Image.open(image_path)
        self.background_image = ImageTk.PhotoImage(original_image)

        self.canvas = Canvas(self, width=640, height=480)
        self.canvas.pack(fill="both", expand=True)
        self.canvas.create_image(0, 0, image=self.background_image,
anchor="nw")
```

```
    def create_widgets(self):
        # Button for selecting an image
        self.select_image_button = ctk.CTkButton(self, text="Select Image", 
width=200, command=self.select_image)
        self.select_image_button.place(x=220, y=40)

        # Label for the text entry
        self.text_label = ctk.CTkLabel(self, text="Parameter:")
        self.text_label.place(x=220, y=80)

        # Text entry
        self.text_entry = ctk.CTkEntry(self, width=135)
        self.text_entry.place(x=285, y=80)

        # Label for the text entry
        self.text_label = ctk.CTkLabel(self, text="Operation:")
        self.text_label.place(x=220, y=120)


        # Combobox for selecting operations
        self.operation_combobox = ctk.CTkComboBox(self, width=140, 
values=["edge_detection", "color_inversion", "grayscale", "threshold", "blur", 
"dilate", "erode", "resize", "equalize_histogram", "find_contours", 
"read_qr_code"])
        self.operation_combobox.set("")  # Set the empty string as the default 
selected value
        self.operation_combobox.place(x=280, y=120)

        # Button for sending requests
        self.send_request_button = ctk.CTkButton(self, text="Send Request", 
width=200, command=self.send_request)
        self.send_request_button.place(x=220, y=160)

        # File label to show the name of the selected file
        self.image_path = ctk.StringVar()
        self.image_path.trace_add("write", self.update_file_label)
        self.file_label = ctk.CTkLabel(self, text="")
        self.file_label.place(x=450, y=40)

    def select_image(self):
        file_types = [
            ('Image files', '*.jpg *.jpeg *.png *.gif *.bmp'),
            ('JPEG', '*.jpg;*.jpeg'),
            ('PNG', '*.png'),
            ('GIF', '*.gif'),
```

```
            ('BMP', '*.bmp'),
        ]

        filenames = filedialog.askopenfilenames(filetypes=file_types)

        if filenames:
            print("Selected files:", filenames)
            # Convert tuple of filenames to a single string separated by
semicolons
            self.image_path.set(";".join(filenames))
            self.total_images = len(filenames)  # Set the total number of
images selected
            self.show_progress_bar()
        else:
            self.image_path.set("")



    def send_request(self):
        def request_thread():
            url = 'http://127.0.0.1:5000/process'  # PUT PUBLIC IP <-----------
--------

            image_paths = self.image_path.get().split(";")
            self.label.configure(text="Processing")
            for idx, image_path in enumerate(image_paths):
                try:
                    with open(image_path, 'rb') as image_file:
                        files = {'images': image_file}
                        data = {
                            'text': self.text_entry.get(),
                            'operation': self.operation_combobox.get()
                        }
                        response = requests.post(url, files=files, data=data)
                        if response.status_code == 200:
                            self.update_progress(idx+1)
                            json_response = response.json()
                        if 'results' in json_response:
                            results = json_response['results']
                            for result in results:
                                if data['operation'] == 'read_qr_code':
                                    plot_image(result[0])
                                    decoded_data = result[1]
                                    show_decoded_data("Decoded Data",
decoded_data)
```

```
                                else:
                                    plot_image(result)
                        else:
                            messagebox.showerror("Error", "No 'results' found
in the response.")
                except Exception as e:
                    messagebox.showerror("Error", f"Failed to send request:
{e}")
        # Start the request in a new thread
        thread = threading.Thread(target=request_thread)
        thread.start()

    def show_progress_bar(self):
        # Create a new top-level window
        self.progress_window = ctk.CTkToplevel()
        self.progress_window.title("Progress")
        self.progress_window.geometry("300x100")  # Adjust size as needed

        # Add a label to display the message above the progress bar
        self.label = ctk.CTkLabel(self.progress_window, text="Ready to
Process")
        self.label.pack(pady=10)  # Add some padding vertically

        # Progress Bar
        self.progress_bar = ctk.CTkProgressBar(self.progress_window, width=200)
        self.progress_bar.pack(pady=10)  # Add some padding vertically

        # Initially set progress bar to 0
        self.progress_bar.set(0)

    def update_progress(self, current_index):
        self.progress_value = current_index / self.total_images # Calculate
progress as a percentage
        self.progress_bar.set(self.progress_value)  # Update the progress bar
        if current_index >= self.total_images:
            self.label.configure(text="Task Done")

    def update_file_label(self, *args):
        self.file_label.configure(text=os.path.basename(self.image_path.get()))

if __name__ == "__main__":
    app = ImageProcessor()
    app.mainloop()
```

Load Balancer and Dispatcher (DispatcherV3.py)

```
from fastapi import FastAPI, UploadFile, Form
from typing import List
from mpi4py import MPI
import numpy as np
import threading
import asyncio
import uvicorn
import httpx
import queue
import time
import json
import cv2
import VMs

app = FastAPI()
task_queue = queue.Queue()
ping_queue = queue.Queue()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
vm_status = {vm_ip: 0 for vm_ip in VMs.ip}

def check_online():
    online_vms = [vm_ip for vm_ip, last_ping_time in vm_status.items() if 
last_ping_time != 0]
    online_vms.sort(key=lambda vm_ip: vm_status[vm_ip], reverse=True)  # Sort 
VMs by last ping time in descending order
    online_vms = online_vms[:2]  # Select the first two online VMs
    print("Online VMS: ", online_vms)
    return online_vms

class WorkerThread(threading.Thread):
    def __init__(self, task_queue):
        threading.Thread.__init__(self)
        self.task_queue = task_queue

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                break
            image_path, text, operation = task
            result = asyncio.run(self.process_image(image_path, text, 
operation))  # Run process_image asynchronously
            self.send_result(result)
```

```
class WorkerThread(threading.Thread):
    def __init__(self, task_queue):
        threading.Thread.__init__(self)
        self.task_queue = task_queue

    def run(self):
        while True:
            task = self.task_queue.get()
            if task is None:
                break
            image_path, text, operation = task
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)
            result = loop.run_until_complete(self.process_image(image_path,
text, operation))  # Run process_image asynchronously
            self.send_result(result)

    async def process_image(self, image_path, text, operation):
        img = cv2.imread(image_path, cv2.IMREAD_COLOR)

        # Get image dimensions
        height, width, _ = img.shape

        # Split the image into two parts (vertical split)
        img1 = img[:, :width//2]
        img2 = img[:, width//2:]
        base_url = "http://{}:5000/receive_result"

        # Check the status of VMs and select the first two online VMs
        online_vms = check_online()

        # Store responses for the first two online VMs
        responses = []
        one_operation_response = None
        temp_img = None
        failed_ip = None

        # Create a list to hold tasks for async post_data calls
        tasks = []

        for i, vm_ip in enumerate(online_vms):
            if len(online_vms) == 1:
                img1 = img
            img_part = img1 if i == 0 else img2  # Assign image part to VM
```

```
            try:
                if operation in ['read_qr_code', 'resize',
'equalize_histogram']:
                    if i == 0:
                        one_operation_response = await
post_data(base_url.format(vm_ip), img, text, operation)  # Await post_data
asynchronously
                else:
                    task = post_data(base_url.format(vm_ip), img_part, text,
operation)  # Create post_data task
                    response = await task  # Await the task individually
                    responses.append(response)  # Append response to list
            except Exception as e:
                temp_img = img_part
                failed_ip = vm_ip
                print("There is a VM Failed:", e)

        # Check if both responses are valid
        if one_operation_response is None:
            if not (all(response and response.status_code == 200 for response
in responses) and (len(responses) == len(online_vms))):
                print("Failed to receive valid responses from one or both
servers.")
                online_vms = check_online()
                # Reassign failed chunk to another VM
                for new_vm_ip in online_vms:
                    if new_vm_ip != failed_ip:
                        print("Reassigned the failed chunk to another VM:",
new_vm_ip)
                        new_response = await
post_data(base_url.format(new_vm_ip), temp_img, text, operation)  # Await
post_data asynchronously
                        if new_response and new_response.status_code == 200:
                            responses.append(new_response)

        # Concatenate the results vertically using numpy
        if operation in ['read_qr_code', 'resize', 'equalize_histogram']:
            new_response = httpx.Response(httpx.codes.OK)
            new_response._content =
json.dumps(one_operation_response.json()).encode('utf-8')
        else:
            json_responses = [response.json() for response in responses]
            concatenated_result = np.concatenate(json_responses, axis=1)
            concatenated_result_list = concatenated_result.tolist()
```

```
            new_response = httpx.Response(httpx.codes.OK)
            new_response._content =
json.dumps(concatenated_result_list).encode('utf-8')

        print("Finished and returning to User")

        return new_response




    def send_result(self, result):
        comm.send(result, dest=0)

async def post_data(url, image, text, operation):
    async with httpx.AsyncClient() as client:
        response = await client.post(url, json={"image": image.tolist(),
"text": text, "operation": operation})
        if response.status_code == 200:
            return response
        else:
            return None

@app.post('/process')
async def process(images: List[UploadFile] = Form(...), text: str = Form(None),
operation: str = Form(...)):
    #print("Received images:", images)   # Print received images for debugging
    #print("Received text:", text)       # Print received text for debugging
    #print("Received operation:", operation)   # Print received operation for
debugging

    if rank == 0:
        image_paths = []
        for image in images:
            image_path = f'images/{image.filename}'
            image_paths.append(image_path)
            with open(image_path, 'wb') as f:
                f.write(await image.read())
            task_queue.put((image_path, text, operation))

        results = []
        for _ in range(len(image_paths)):
            result = comm.recv(source=MPI.ANY_SOURCE)
            results.append(result.json())
```

```python
        return {'results': results}
    else:
        return {'message': 'This endpoint is for master node only.'}

@app.post('/ping')
async def ping(vm_id: dict):
    vm_ip = vm_id.get("vm_id")
    vm_status[vm_ip] = time.time()  # Update timestamp for the VM
    return {'response': 'pong'}

def handle_ping_from_vms():
    uvicorn.run(app, host='0.0.0.0', port=5000)

if __name__ == '__main__':
    worker_threads = []
    if rank == 0:
        threading.Thread(target=handle_ping_from_vms).start()
    for i in range(MPI.COMM_WORLD.Get_size() - 1):
        worker_thread = WorkerThread(task_queue)
        worker_thread.start()
        worker_threads.append(worker_thread)

    # Wait for all worker threads to finish
    for worker_thread in worker_threads:
        worker_thread.join()
```

VMs Receiver (serverV2.py)

```
import numpy as np
import subprocess
from fastapi import FastAPI, HTTPException
import tempfile
import httpx
import asyncio
import signal
import threading
import uvicorn
import sys

app = FastAPI()

# Global flag to indicate whether the program should continue running
running = True

# Function to handle SIGINT signal (Ctrl+C)
def signal_handler(sig, frame):
    global running
    print("Exiting...")
    running = False
    sys.exit(0)

async def send_ping_to_dispatcher(vm_id: str):
    dispatcher_url = "http://172.31.9.48:5000/ping" #<----------------
Dispatcher IP
    payload = {'vm_id': vm_id}
    try:
        async with httpx.AsyncClient() as client:
            response = await client.post(dispatcher_url, json=payload)
            if response.status_code == 200:
                pass
            else:
                pass
    except Exception:
        print(f"Error sending ping")


async def run_ping_thread():
    while running:
        print("Sending ping...")
        await send_ping_to_dispatcher("172.31.46.231")#<----------------
Private IP
        await asyncio.sleep(5)
```

```
@app.post('/receive_result')
async def receive_result_handler(image: dict):
    received_image = np.array(image.get("image"), dtype=np.uint8)
    operation = image.get("operation")
    text = image.get("text")
    if text is not None:
        if text.isdigit():
            pass
        else:
            text = None
    try:
        print("Before executing MPI process")
        with tempfile.NamedTemporaryFile(delete=False, suffix='.npy') as
temp_file:
            np.save(temp_file, received_image)
            temp_file_path = temp_file.name
        if text is None:
            subprocess.run(['mpirun', '--hostfile', 'my_hostfile.txt', '-np',
'2', 'python3', 'WorkerV3.py', temp_file_path, operation])
        else:
            subprocess.run(['mpirun', '--hostfile', 'my_hostfile.txt', '-np',
'2', 'python3', 'WorkerV3.py', temp_file_path, operation, text])
        print("MPI process completed")
    except Exception as e:
        print("Error executing MPI process:", e)
    finally:
        print("MPI Finish")
        try:
            processed_result = np.load('processed_result.npy')
            processed_result_list = processed_result.tolist()
            if operation == "read_qr_code":
                with open("decoded_data.txt", "r") as file:
                    decoded_data = [line.strip() for line in file.readlines()]
                return processed_result_list, decoded_data
            return processed_result_list
        except Exception as e:
            print("Error loading processed result:", e)
            return []

def run_worker():
    uvicorn.run(app, host='0.0.0.0', port=5000)

if __name__ == "__main__":
    # Register signal handler for SIGINT (CTRL+C)
```

```
    signal.signal(signal.SIGINT, signal_handler)
    ping_thread = threading.Thread(target=asyncio.run,
args=(run_ping_thread(),))
    ping_thread.start()
    run_worker()
    ping_thread.join()
```

VMs Processing (WorkerV3.py)

```
from queue import Queue
import sys
import numpy as np
from mpi4py import MPI
import ImageProcessingFunctions


# Initialization MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
result_queue = Queue()

def process_image(img, operation, text):
    if operation == 'read_qr_code':
        if rank ==0:
            processed_chunk = ImageProcessingFunctions.read_qr_code(img)
            return processed_chunk
        else:
            return
    elif operation == 'resize':
        if rank ==0:
            if text is None:
                processed_chunk = ImageProcessingFunctions.resize(img)
            else:
                processed_chunk =
ImageProcessingFunctions.resize(img,int(text),int(text))
            return processed_chunk
        else:
            return
    elif operation == 'equalize_histogram':
        if rank ==0:
            processed_chunk = ImageProcessingFunctions.equalize_histogram(img)
            return processed_chunk
        else:
            return
    height, width = img.shape[:2]
    # Broadcast image dimensions to all MPI Threads
    height, width = comm.bcast((height, width), root=0)
    # Define overlap size
    overlap_size = 0  # Adjust if needed (Make it zero for now !! :Rafik)
    # Calculate chunk height for each process including overlap
    chunk_height = (height + (size - 1) * overlap_size) // size
    # Calculate start and end rows for each process
```

```
    start_row = rank * (chunk_height - overlap_size)
    end_row = min((rank + 1) * chunk_height, height)
    # Adjust start and end rows for overlapping chunks
    if rank != 0:
        start_row += overlap_size
    if rank != size - 1:
        end_row += overlap_size
    # Ensure start_row and end_row are within image boundaries
    start_row = max(start_row, 0)
    end_row = min(end_row, height)
    # Calculate chunk height after adjusting for overlap
    chunk_height = end_row - start_row
    # Create the chunk with the correct dimensions
    chunk = np.zeros((chunk_height, width, 3), dtype=np.uint8)
    # Copy the corresponding part of the image to the chunk
    if chunk_height > 0:
        chunk[:chunk_height, :, :] = img[start_row:end_row, :, :]
    # Processing with overlap
    if operation == 'edge_detection':
        processed_chunk = ImageProcessingFunctions.edge_detection(chunk)
    elif operation == 'color_inversion':
        processed_chunk = ImageProcessingFunctions.color_inversion(chunk)
    elif operation == 'grayscale':
        processed_chunk = ImageProcessingFunctions.grayscale(chunk)
    elif operation == 'threshold':
        processed_chunk = ImageProcessingFunctions.threshold(chunk)
    elif operation == 'blur':
        if text is None:
            processed_chunk = ImageProcessingFunctions.blur(chunk,text)
        else:
            processed_chunk = ImageProcessingFunctions.blur(chunk,int(text))
    elif operation == 'dilate':
        if text is None:
            processed_chunk = ImageProcessingFunctions.dilate(chunk,text)
        else:
            processed_chunk = ImageProcessingFunctions.dilate(chunk,int(text))
    elif operation == 'erode':
        if text is None:
            processed_chunk = ImageProcessingFunctions.erode(chunk,text)
        else:
            processed_chunk = ImageProcessingFunctions.erode(chunk,int(text))
    elif operation == 'find_contours':
        processed_chunk = ImageProcessingFunctions.find_contours(chunk)
    else:
        processed_chunk = None  # Operation not supported
```

```
    # Send chunks to rank 0
    gathered_chunks = comm.gather(processed_chunk, root=0)

    # Stitching the image
    if rank == 0:
        # Concatenate gathered chunks into a single list
        concatenated_chunks = []
        for i, proc_chunk in enumerate(gathered_chunks):
            if i == 0:
                concatenated_chunks.append(proc_chunk)
            else:
                concatenated_chunks.append(proc_chunk[overlap_size:])

        # Concatenate gathered chunks into processed image
        processed_image = np.vstack(concatenated_chunks) # this is vertical
stack
        return processed_image


if __name__ == "__main__":
    # Extract image and operation from command-line arguments
    received_image = np.load(sys.argv[1])
    #print(received_image)
    operation = sys.argv[2]
    if len(sys.argv) == 4:
        text = sys.argv[3]
    else:
        text = None
    # Perform image processing using MPI
    print(f"Processing in workerV3 and my rank: {rank}")
    processed_result = process_image(received_image, operation,text)
    # Save the processed result into a file
    if rank ==0:
        if operation == "read_qr_code":
                    # Save the decoded data to a text file
            with open("decoded_data.txt", "w") as file:
                for data in processed_result[1]:
                    file.write(data + "\n")
            np.save('processed_result.npy', processed_result[0])
        else:
            np.save('processed_result.npy', processed_result)
```

Implemented Processing Functions (ImageProcessingFunctions.py)

```
import cv2
import numpy as np
from pyzbar.pyzbar import decode

def edge_detection(image):
    return cv2.Canny(image, 100, 200)

def color_inversion(image):
    return cv2.bitwise_not(image)

def grayscale(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

def threshold(image):
    threshold_value=128
    max_value=255
    threshold_type=cv2.THRESH_BINARY
    image = grayscale(image)
    _, binary_image = cv2.threshold(image, threshold_value, max_value,
threshold_type)
    return binary_image

def blur(image, kernel_size=None):
    if kernel_size is None:
        kernel_size = (5, 5)
    elif isinstance(kernel_size, int):
        if kernel_size %2 == 0:
            kernel_size = kernel_size-1
        kernel_size = (kernel_size, kernel_size)
    return cv2.GaussianBlur(image, kernel_size, 0)

def dilate(image, kernel_size=None):
    if kernel_size is None:
        kernel_size = (5, 5)
    elif isinstance(kernel_size, int):
        if kernel_size %2 == 0:
            kernel_size = kernel_size-1
        kernel_size = (kernel_size, kernel_size)
    kernel = np.ones(kernel_size, np.uint8)
    iterations = 2
    return cv2.dilate(image, kernel, iterations=iterations)

def erode(image, kernel_size=None):
    if kernel_size is None:
```

```
        kernel_size = (5, 5)
    elif isinstance(kernel_size, int):
        if kernel_size %2 == 0:
            kernel_size = kernel_size-1
        kernel_size = (kernel_size, kernel_size)
    kernel = np.ones(kernel_size, np.uint8)
    iterations = 2
    return cv2.erode(image, kernel, iterations=iterations)

def find_contours(image):
    # Convert the image to grayscale if it's not already
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) if len(image.shape) == 3
else image

    edges = cv2.Canny(gray, 30, 100)

    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    contour_image = image.copy()

    cv2.drawContours(contour_image, contours, -1, (0, 255, 0), 1)

    return contour_image
##########################################################
def resize(image, width=None, height=None):
    if width is None and height is None:
        return image
    elif width is None:
        r = height / image.shape[0]
        dim = (int(image.shape[1] * r), height)
    elif height is None:
        r = width / image.shape[1]
        dim = (width, int(image.shape[0] * r))
    else:
        dim = (width, height)
    return cv2.resize(image, dim, interpolation=cv2.INTER_AREA)

def equalize_histogram(image):
    if len(image.shape) == 3:
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray_image = image
    img_equalized = cv2.equalizeHist(gray_image)
    return img_equalized
```

```
def read_qr_code(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    qr_codes = decode(gray)
    decoded_data = []

    for qr_code in qr_codes:
        x, y, w, h = qr_code.rect
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
        data = qr_code.data.decode("utf-8")
        decoded_data.append(data)

        # Calculate the font scale based on the length of the decoded data
        font_scale = min(1, 1000 / len(data))  # Adjust 1000 as needed based on
your image size

        # Calculate the thickness of the text
        thickness = max(1, int(font_scale))

        # Get the size of the text
        (text_width, text_height), _ = cv2.getTextSize(data,
cv2.FONT_HERSHEY_SIMPLEX, font_scale, thickness)

        # Calculate the position to place the text
        text_x = max(x, 0)
        text_y = y + h + text_height + 10  # Adjust 10 as needed

        # Ensure text stays within image boundaries
        if text_x + text_width > image.shape[1]:
            text_x = image.shape[1] - text_width - 10  # Adjust 10 as needed

        # Draw the text on the image
        cv2.putText(image, data, (text_x, text_y), cv2.FONT_HERSHEY_SIMPLEX,
font_scale, (0, 255, 0), thickness)

    return image, decoded_data


#############################################
def PrintTest(l):
    l = "Hello"
    return l
def Test(image):
    print("HEEEEEERE")
```

```
    return image
```