A close up of a black background

Description automatically generatedA blue circle with two birds and a tower

Description automatically generated

# CSE345 Distributed Computing Major Task

(Distributed Image Processing System using Cloud Computing)

A close up of a black background

Description automatically generated

**Submitted by:**

**Ahmed Hesham 19P8052**

**Mohamed ElSayed 1901757**

**Jumana Yasser 20P8421**

**Seif ElDin Amr 20P2006**

Table of Contents

[CSE345 Distributed Computing Major Task 1](#_Toc166937539)

[Table of Figures 3](#_Toc166937540)

[1 INTRODUCTION 3](#_Toc166937541)

[2 PHASE 1 5](#_Toc166937542)

[2.1 Main Objectives: 5](#_Toc166937543)

[2.2 Main Technologies: 6](#_Toc166937544)

[2.3 Key Components and System Architecture: 7](#_Toc166937545)

[2.4 Architecture Diagram: 7](#_Toc166937546)

[2.5 Sequence Diagram: 8](#_Toc166937547)

[2.6 Component Diagram: 8](#_Toc166937548)

[3 PHASE 2 9](#_Toc166937549)

[3.1 How does our worker thread process tasks? 9](#_Toc166937550)

[3.2 Basic image processing operations: 10](#_Toc166937551)

[3.3 How have we set up our cloud environment? 12](#_Toc166937552)

[3.4 How does our system handle distributed processing? 13](#_Toc166937553)

[3.4.1 Mater-VM: 13](#_Toc166937554)

[3.4.2 Worker-VMs 13](#_Toc166937555)

[3.4.3 Code: 14](#_Toc166937556)

[4 PHASE 3 (new approaches) 16](#_Toc166937557)

[4.1 New System Architecture and Main Process 16](#_Toc166937558)

[4.1.1 System architecture 16](#_Toc166937559)

[4.1.2 Main process 17](#_Toc166937560)

[4.2 Advanced Image Processing Operations 18](#_Toc166937561)

[4.3 How have we implemented scalability and fault tolerance? 20](#_Toc166937562)

[4.3.1 Scalability Techniques (demo video) 20](#_Toc166937563)

[4.3.2 Fault Tolerance (demo video) 20](#_Toc166937564)

[5 PHASE 4 21](#_Toc166937565)

[5.1 What are the Outcomes of Our System Testing? 21](#_Toc166937566)

[5.1.1 Overview of Testing Process 21](#_Toc166937567)

[5.1.2 Functionality Testing (demo video) 21](#_Toc166937568)

[5.1.3 Scalability Testing (demo video) 21](#_Toc166937569)

[5.1.4 Fault Tolerance Testing (demo video) 21](#_Toc166937570)

[5.1.5 Summary of Results 22](#_Toc166937571)

[5.2 What Information is Included in Our Documentation? 22](#_Toc166937572)

[5.3 How Did we Deploy Our System to the Cloud? 23](#_Toc166937573)

[5.3.1 Overview 23](#_Toc166937574)

[5.3.2 Setup Azure Resource: 23](#_Toc166937575)

[5.3.3 VM Configuration: 23](#_Toc166937576)

[5.3.4 Environment Configuration: 23](#_Toc166937577)

[5.3.5 Code Deployment: 23](#_Toc166937578)

[5.3.6 Starting the Application 23](#_Toc166937579)

[5.3.7 Monitoring and Scaling: 24](#_Toc166937580)

[5.3.8 Tools and Services Used 24](#_Toc166937581)

[6 CONCLUTION 25](#_Toc166937582)

[7 IMPORTANT LINKS 26](#_Toc166937583)

[7.1 Drive Link 26](#_Toc166937584)

[7.2 GitHub Link 26](#_Toc166937585)

# Table of Figures

[Figure 1: System Architecture 7](#_Toc166918625)

[Figure 2: Sequence Diagram 8](#_Toc166918626)

[Figure 3:Component Diagram 8](#_Toc166918627)

[Figure 4: Azure VMs 12](#_Toc166918628)

[Figure 5: New System Architecture 16](#_Toc166918629)

[Figure 6: Sequence Diagram 17](#_Toc166918630)

# INTRODUCTION

In the present era, the demands for image processing systems are still rising to perform it at high speed and efficiency due to various applications in the domain of medical imaging, computer vision, and entertainment. Most of the conventional image processing methods on a single machine simply do not work well for large-scale workloads; results include processing times going through the roof and unresponsive machines.

By using the power and scalability offered by cloud computing, the proposed system will process the image data on multiple virtual machines in an involved architecture to distribute processing tasks among many VMs. One can see that an improvement in processing speed and scalability can be gotten, which allows the system to easily process large datasets and even complex processing operations.

The system will be Python-driven with the utilization of the OpenCV application for the image processing operations. In the process, one will see that I will use the Message Passing Interface (MPI) to organize the distributed process. Cloud computing platforms, such as Amazon Web Services, Microsoft Azure, or Google Cloud Platform, will be used to provision the virtual machines and orchestrate the distributed environment.

This paper will depict how the design, implementation, and testing of the distributed image processing system would be structured. It will outline key features such as fault tolerance, scalability, and advanced image processing operations.

The goal is to implement a flexible, robust distributed system that can work with any image processing task that needs to be performed as part of the workflow with scalability and reliability. This system is going to be implemented to test the benefits of the cloud-based distributed computing for image processing applications and give a platform for further developments.

# PHASE 1

## Main Objectives:

Our main objective is to design, implement and test a distributed image processing system using cloud computing technologies in the main aim of your project. The system is a distributed processing system that would empower the processing of image data efficiently through leveraging distributed computing across several virtual machines in the cloud. Implementing a distributed system, the project attempts to acquire:

**1. Scalability**: The ability to scale the system by adding more virtual machines with increasing workload and hence the maintenance of processing speed and efficiency.

**2. Fault Tolerance**: A system ensuring reliability by ensuring that tasks are re-distributed in the event of failures so that the system works even with failures in nodes.

**3. Efficiency**: The application of parallel processing techniques for the processing of image-processing tasks in quick and efficient ways to ensure low time for complex operations to be executed.

**4. User Interface**: A user-friendly interface by which users can submit images, select the processing operations, download the processed images, and monitor the tasks' progress.

In the achievement of these aims, the project intends to test and prove the effectiveness of cloud-based distributed computing for image processing and, consequently, create a basis for further research in this area.

## Main Technologies:

We have chosen **Python** as the main language for its simplicity, the rich number of libraries that support applications, and the vast community support. Python is also flexible in its development and easily integrated with other technologies, making it perfect for building distributed systems. We use **OpenCV**, which is a widely used library that provides an extensive set of tools for operations like filtering, edge detection, and color manipulation. Together with Python and OpenCV, we achieve the capability of developing simple to complex image processing algorithms easily.

For the distributed environment, we have opted for **Microsoft Azure**, which is thought of as the best in the cloud-computing platform in terms of scalability and flexibility with strong infrastructure. Azure's VMs serve as resources, which our distributed computing setup requires and needs to run on multiple instances to perform processing in parallel. Azure ensures a rapid scalability that can meet the demands of fluctuating workloads, ensuring that our system is always responsive. Besides that, Azure provides built-in services that monitor and manage the virtual machines to give us fault-tolerant integration. By combining these, we will build a strong and scalable distributed image processing system.

## Key Components and System Architecture:

An architecture for distributed image processing consists of a master node and multiple worker nodes, with the control of their management being performed through virtual machines on the cloud platform Azure. The master node coordinates the processes and essentially is the human interface to a cloud-based service. It receives image processing requests from end-users and interleaves them amongst the available worker nodes, then receives results.

## Architecture Diagram:

Worker nodes function on separate virtual machines in the cloud and perform image processing by using the OpenCV framework. Worker nodes receive tasks from the master node, which assigns the processing algorithms to be used, and send the results back to the master node when they are ready. This is done using the Message Passing Interface, which maintains sufficient coordination and effectively distributes tasks among the nodes. This architecture allows the scaling of processing power through the creation of additional worker nodes when the need is detected and can be fault tolerant due to processing loads being spread out across node failure.

A diagram of a cloud

Description automatically generated

Figure 1: System Architecture

## A diagram of a process Description automatically generatedSequence Diagram:

Figure 2: Sequence Diagram

## Component Diagram:

A diagram of a computer process

Description automatically generated

Figure 3:Component Diagram

# PHASE 2

## How does our worker thread process tasks?

A task-based model of execution would be used for the worker thread, pulling tasks from a shared task queue and executing them in turn. Assuming that tasks are tuples, containing all data and instructions needed to perform an operation on that task, when a task is pulled from the queue, the worker thread would call the appropriate processing function, passing the input data as a parameter. The processing function would be whatever operation the worker thread performs, such as image processing, data analysis, or any other computation needed to fulfill the task.

After the processing function is completed, the worker thread sends the result back to the master node via some means of communication, like message passing or shared memory. If the value picked up is a special value, usually referred to as a poison pill, the worker thread recognizes it as a call to quit and intelligently quits the loop. The worker thread can also be designed to handle errors that might occur when tasks are processed, by catching and reporting on such exceptions. This then allows for proper distribution of work in multiple worker threads, enabling parallel processing and proper use of available computing resources.

class WorkerThread(threading.Thread):

def \_\_init\_\_(self, task\_queue):

super().\_\_init\_\_()

self.task\_queue = task\_queue

self.comm = MPI.COMM\_WORLD

self.rank = self.comm.Get\_rank()

def run(self):

while True:

task = self.task\_queue.get()

if task is None:

self.task\_queue.put(None) # Put the poison pill back for other workers

break

try:

image, operation = task

result = self.process\_image(image, operation)

self.send\_result(result)

except Exception as e:

print(f"Error occurred in worker {self.rank}: {e}")

def send\_result(self, result):

self.comm.send(result, dest=0)

## Basic image processing operations:

import tkinter as tk # For the file dialog

from tkinter import filedialog

import cv2 # For image processing

import os # To check if file exists

import numpy as np

# Function to select an image file

def select\_image():

# Create a Tkinter root window but keep it hidden

root = tk.Tk()

root.withdraw() # Hide the root window

# Open a file dialog to select an image

file\_path = filedialog.askopenfilename(

title="Select an Image File",

filetypes=[("JPEG files", " \*.jpeg;\*.jpg"), ("PNG files", "\*.png"), ("All files", "\*.\*")]

)

return file\_path # Return the selected file path

# Function to process an image

def process\_image(image\_path, operation):

# Load the image

img = cv2.imread(image\_path, cv2.IMREAD\_COLOR)

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

gray = cv2.cvtColor(img, cv2.COLOR\_RGB2GRAY)

# Flip the image Horizontally

gray = cv2.flip(img, 1)

threshold = 0

# Create a mask for gray pixels

mask = gray <= threshold

# Set gray pixels to white

gray[mask] = 255

kernel = np.ones((5,5), np.uint8)

dilated\_image = cv2.dilate(gray, kernel, iterations=3)

eroded\_image = cv2.erode(dilated\_image, kernel, iterations=3)

result = cv2.convertScaleAbs(eroded\_image)

if img is None:

raise ValueError(f"Failed to load image: {image\_path}")

return result # Return the processed image

# Main code

def main():

file\_path = select\_image() # Get the selected image path

if not file\_path or not os.path.exists(file\_path):

print("No valid file selected")

return

# Specify the operation to perform

operation = 'edge\_detection' # Change to 'color\_inversion' or other operation

try:

processed\_image = process\_image(file\_path, operation) # Process the image

# Display the processed image

cv2.imshow("Processed Image", processed\_image) # Display the processed image

cv2.waitKey(0) # Wait for a key press to close the window

cv2.destroyAllWindows() # Close all OpenCV windows

except ValueError as e:

print(f"Error: {e}") # Handle exceptions appropriately

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

main()

## How have we set up our cloud environment?

A screenshot of a computer

Description automatically generated

**Figure 4: Azure VMs**

* **Virtual Machines on Azure**: Three virtual machines have been set up on the Azure platform. The operating system for the VMs is set to Ubuntu 22.04 LTS.
* **Master Node:** One of the three VMs will be the master node, which will manage and coordinate the distributed computing tasks.
* **Worker Nodes:** The remaining two VMs will act as worker nodes, which will take the tasks from the master node, process them, and send back the results to the master node.
* **Azure Virtual Network:** The three VMs are connected by an Azure Virtual Network. This ensures that they can communicate and exchange data securely in the same virtual network environment.
* **Shared Storage:** A shared storage solution is in place, probably an Azure Storage service, such as Azure Files or Azure Blob Storage. The shared storage will be used for storing and retrieving the images which need to be processed by the distributed system.
* **Network Security:** Suitable network security measures like NSGs or Azure Firewall might have been implemented to control the inbound and outbound traffic and make the virtual network environment secure.

## How does our system handle distributed processing?

### Mater-VM:

We are using MPI to handle the master and worker nodes.

The communicator MPI.COMM\_WORLD binds all nodes so that rank 0, the master node, is responsible for distributing tasks with a queue.Queue, which dispatches the starting tasks to the worker nodes and collects the results. Once tasks are finished, the master now distributes the next one until the queue is empty, sending the signal None for the workers to terminate.

The rank 1 and onwards worker nodes continuously receive tasks from the master node using comm.recv(). If the worker receives None, then it gracefully exits the program. Otherwise, it proceeds to process the image using process\_image and sends the result to the master, including the rank, task information, and status. The master node initializes with a task queue, assigns tasks to workers, and re-assigns tasks as results come in, with the workers completing tasks and awaiting further instruction or the termination signal.

### Worker-VMs

In our setup a multiple worker threads will execute tasks composed of images concurrently. we use threading and MPI for message passing to handle the distribution of tasks and the collection of results.

The class WorkerThread is derived from threading.Thread , it handles the processing logic by fetching tasks from a shared instance of queue.Queue. Each of the threads will process tasks composed of the filename of the image and the operation to be carried out with methods such as process\_image, handling edge detection or color inversion via OpenCV; it will then send back results to the master node using MPI's send().

The global communicator MPI.COMM\_WORLD deals with the communication between all processes. comm.Get\_rank() identifies each process, while comm.send() allows workers to send back results to the master node (rank 0). The main setup first initializes a thread-safe task\_queue. The number of worker threads equals the total MPI processes minus one. The workers then get initiated, started, and stored in a list. When done, the master places None tasks in the queue to stop the workers.

### Code:

from mpi4py import MPI

import queue

# Initialize the MPI communicator

comm = MPI.COMM\_WORLD

rank = comm.Get\_rank()

size = comm.Get\_size()

# Master node (rank 0)

if rank == 0:

# List of images and their corresponding operations

tasks = [

("image1.jpg", "edge\_detection"),

("image2.jpg", "color\_inversion"),

]

for task in tasks:

task\_queue.put(task)

# Track the number of results to collect

num\_tasks = len(tasks)

results = []

# Send initial tasks to each worker node

for worker\_rank in range(1, size):

if not task\_queue.empty():

task = task\_queue.get()

comm.send(task, dest=worker\_rank)

else:

break

# Collect results and distribute remaining tasks

while num\_tasks > 0:

# Receive results from any worker

result = comm.recv(source=MPI.ANY\_SOURCE)

results.append(result)

num\_tasks -= 1

if not task\_queue.empty():

task = task\_queue.get()

comm.send(task, dest=result['worker\_rank'])

else:

comm.send(None, dest=result['worker\_rank'])

for result in results:

print(f"Worker {result['worker\_rank']} processed {result['task']} with status: {result['status']}")

# Worker nodes (rank 1 and higher) would use the worker class already defined

else:

while True:

task = comm.recv(source=0)

if task is None:

break

# Unpack the received task

image, operation = task

# Process the task using the worker's methods

result\_status = "success"

try:

result\_image = process\_image(image, operation)

except Exception as e:

result\_status = f"error: {str(e)}"

# Send the result back to the master

comm.send({

'worker\_rank': rank,

'task': (image, operation),

'status': result\_status

}, dest=0)

import threading

import queue

import cv2 # OpenCV for image processing

from mpi4py import MPI # MPI for distributed computing

class WorkerThread(threading.Thread):

def \_\_init\_\_(self, task\_queue):

threading.Thread.\_\_init\_\_(self)

self.task\_queue = task\_queue

self.comm = MPI.COMM\_WORLD

self.rank = self.comm.Get\_rank()

self.running = True # Add a running flag

def run(self):

while self.running:

try:

task = self.task\_queue.get(timeout=3) # Timeout for clean shutdown

except queue.Empty:

continue

if task is None:

self.running = False

break

image, operation = task

result = self.process\_image(image, operation)

if result is not None:

self.send\_result(result)

def process\_image(self, image, operation):

# Load the image safely

img = cv2.imread(image, cv2.IMREAD\_COLOR)

if img is None:

print(f"Error: Unable to load image '{image}'")

return None

# Perform the specified operation

if operation == 'edge\_detection':

result = cv2.Canny(img, 100, 200)

elif operation == 'color\_inversion':

result = cv2.bitwise\_not(img)

else:

print(f"Warning: Operation '{operation}' is not recognized")

return None

# Add more operations as needed...

return result

def send\_result(self, result):

# Send the result to the master node

self.comm.send(result, dest=0)

# Create a queue for tasks

task\_queue = queue.Queue()

# Create worker threads

num\_workers = MPI.COMM\_WORLD.Get\_size() - 1

workers = []

for i in range(num\_workers):

worker = WorkerThread(task\_queue)

worker.start()

workers.append(worker)

# Example of shutting down workers

# Add `None` tasks to queue to signal workers to stop

for \_ in range(num\_workers):

task\_queue.put(None)

# Wait for all workers to finish

for worker in workers:

worker.join()

# PHASE 3 (new approaches)

We switched from using MPI to using TCP communication as MPI had issues in exciting the command that distributes tasks from the master to workers. Also, we added some new architecture components which changed the sequence of execution and the system architecture.

## New System Architecture and Main Process

### System architecture

* **Master Node**: Master node managing’s tasks and worker nodes using Azure Queue Storage in task distribution and Azure Blob Storage in image and result storage. A Flask server provides a web interface for users to upload images, select processing operations, and view results.
* **Worker Nodes:** Connects to the master node to get the tasks, process the images using OpenCV, and upload results back to Azure Blob Storage. They keep attempting to reconnect upon losing the connection to the master node.
* A screenshot of a computer

  Description automatically generated**Flask Application:** This allows the user to upload images and specify the desired processing operation, adding tasks into Azure Queue Storage for the workers to retrieve and show the status of the workers and processed results.

Figure 5: New System Architecture

### Main process

* **File Upload and Task Creation**: Users upload images from the Flask web interface, specify an image processing operation, save images locally, then upload them to Azure Blob Storage. A task message is created and added to Azure Queue Storage.
* **Task Distribution:** Worker nodes fetch tasks from Azure Queue Storage, fetch the corresponding images from Azure Blob Storage, and undertake the specified image processing operation.
* **Image Processing:** Using OpenCV, workers process the images, save the processed images locally, then upload them back to Azure Blob Storage and delete the local copy for space-saving. The Master node send the result URL back to the master node, updating the status and results in the Flask application.
* A screenshot of a computer

  Description automatically generated**Viewing the result**: Users can view the status of their tasks and download the processed images from the Flask web interface.

Figure 6: Sequence Diagram

## Advanced Image Processing Operations

* **Watershed Segmentation**
* **Canny Edge Detection**
* **Feature Matching**
* **Face Detection**

def watershed\_segmentation(image\_path):

logging.info(f"Starting watershed segmentation on {image\_path}")

image = cv2.imread(image\_path)

if image is None:

logging.error(f"Failed to load image at {image\_path}")

return None

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

\_, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

kernel = np.ones((3, 3), np.uint8)

opening = cv2.morphologyEx(thresh, cv2.MORPH\_OPEN, kernel, iterations=2)

sure\_bg = cv2.dilate(opening, kernel, iterations=3)

dist\_transform = cv2.distanceTransform(opening, cv2.DIST\_L2, 5)

\_, sure\_fg = cv2.threshold(dist\_transform, 0.7 \* dist\_transform.max(), 255, 0)

sure\_fg = np.uint8(sure\_fg)

unknown = cv2.subtract(sure\_bg, sure\_fg)

\_, markers = cv2.connectedComponents(sure\_fg)

markers = markers + 1

markers[unknown == 255] = 0

markers = cv2.watershed(image, markers)

image[markers == -1] = [255, 0, 0]

logging.info(f"Completed watershed segmentation on {image\_path}")

return save\_image(image, "watershed\_segmented")

def canny\_edge\_detector(image\_path):

logging.info(f"Starting Canny edge detection on {image\_path}")

image = cv2.imread(image\_path, 0)

if image is None:

logging.error(f"Failed to load image at {image\_path}")

return None

edges = cv2.Canny(image, 100, 200)

logging.info(f"Completed Canny edge detection on {image\_path}")

return save\_image(edges, "canny\_edges")

def feature\_matching(image\_path1, image\_path2):

logging.info(f"Starting feature matching between {image\_path1} and {image\_path2}")

img1 = cv2.imread(image\_path1, cv2.IMREAD\_GRAYSCALE)

img2 = cv2.imread(image\_path2, cv2.IMREAD\_GRAYSCALE)

if img1 is None or img2 is None:

logging.error(f"Failed to load images at {image\_path1} or {image\_path2}")

return None

orb = cv2.ORB\_create()

kp1, des1 = orb.detectAndCompute(img1, None)

kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv2.BFMatcher(cv2.NORM\_HAMMING, crossCheck=True)

matches = bf.match(des1, des2)

matches = sorted(matches, key=lambda x: x.distance)

img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=2)

logging.info(f"Completed feature matching between {image\_path1} and {image\_path2}")

return save\_image(img3, "feature\_matches")

def face\_detection(image\_path):

logging.info(f"Starting face detection on {image\_path}")

img = cv2.imread(image\_path)

if img is None:

logging.error(f"Failed to load image at {image\_path}")

return None

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

face\_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')

faces = face\_cascade.detectMultiScale(gray, 1.3, 5)

for (x, y, w, h) in faces:

cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)

logging.info(f"Completed face detection on {image\_path}")

return save\_image(img, "detected\_faces")

## How have we implemented scalability and fault tolerance?

### Scalability Techniques ([demo video](https://drive.google.com/file/d/1Z0sOgKXO6-3eGV0KORXSo_lsLqLVQrDC/view?usp=share_link))

* **Cloud-based storage and queuing**: Azure Blob Storage allows scaling with the amount of data, and Azure Queue Storage handles the messages of tasks to be processed. It allows the distribution of tasks dynamically.
* **Distributed Processing**: Many worker nodes can be deployed that will handle the tasks in parallel and increase the capability of the system to process many images at the same time.
* **Elasticity:** The number of worker nodes can be adjusted according to the load, and hence the system can handle loads of varying magnitudes without overprovisioning resources.

### Fault Tolerance ([demo video](https://drive.google.com/file/d/1Z0sOgKXO6-3eGV0KORXSo_lsLqLVQrDC/view?usp=share_link))

* **Resilience:** The worker nodes will keep trying to reconnect with the master node if the connection is lost, and this ensures that processing is not interrupted in the case of transient network failures. If any worker node fails in the middle of processing a task, the task will stay in the queue to be processed by another worker.
* **Task reassignment**: The worker nodes fetch the tasks from Azure Queue Storage, and in case of failure of any worker node to process the task or disconnection, the task remains in the queue for fetching by another worker for reprocessing.
* **Logging and Monitoring:** This system logs critical events and errors that are useful for health monitoring of nodes, hence the entire system, to quickly identify and solve issues. Status updates from the worker nodes are sent back to the master node and can be monitored through the Flask web interface to keep tabs on the system state.
* **Redundancy**: In this case, by making use of Azure services, one obtains inherent redundancy and high availability that are resistant to data loss and outages.

# PHASE 4

## What are the Outcomes of Our System Testing?

### Overview of Testing Process

The system underwent full-fledged testing to make sure that the system meets the project requirements and performs well under several conditions. The testing included functionality, performance, scalability, and fault tolerance.

### Functionality Testing ([demo video](https://drive.google.com/file/d/1Z0sOgKXO6-3eGV0KORXSo_lsLqLVQrDC/view?usp=share_link))

* **File Upload:** Tested file upload with different formats and sizes to ensure that the system can handle any kind of input correctly.
* **Task Processing**: All defined image processing operations, such as feature matching and Canny edge detection, have been tested for their accuracy.
* **Result Retrieval**: Confirmed that the processed images are uploaded properly to Azure Blob Storage, and the URLs are returned to the user properly.

### Scalability Testing ([demo video](https://drive.google.com/file/d/1Z0sOgKXO6-3eGV0KORXSo_lsLqLVQrDC/view?usp=share_link))

* **Worker Scaling**: Dynamically added and removed worker nodes to test the ability of the system to scale up and down based on workload.
* **Queue Management**: Tested whether the Azure Queue Storage handles task messages efficiently, even when there are many tasks.

### Fault Tolerance Testing ([demo video](https://drive.google.com/file/d/1Z0sOgKXO6-3eGV0KORXSo_lsLqLVQrDC/view?usp=share_link))

* **Worker Failure Simulation**: Simulated the failure of worker nodes and tested the reassigned and reprocessed tasks from the failed node by other available nodes.
* **Network Issues**: Tested the resilience of the system to temporary network disruptions by disconnecting and reconnecting worker nodes during task processing.

### Summary of Results

* + **Successful Upload**: All the tested image files were successfully uploaded, with appropriate error handling for unsupported formats or sizes.
  + **Accuracy in Processing**: All image processing operations were executed with accuracy. Processed images were compared to the expected outcome.
  + **Efficiency in Task Handling**: The system handled multiple concurrent tasks in an efficient manner with acceptable response times.
  + **Scalable Performance**: The system demonstrated the ability to scale dynamically by handling an increased amount of workload by adding worker nodes.
  + **Robust Fault Tolerance:** Tasks were reassigned from failed workers and maintained overall functionality during network issues.

## What Information is Included in Our Documentation?

* **System Architecture**
  + Overview of Components: Details about the master node, worker nodes, Flask application, and Azure integration
  + Data Flow: Explanation and diagrams of how data flows through the application—from file upload to retrieving the results.
* **Setup and Configuration**
  + Environment Setup: A setup guide on the development environment for all required software and dependencies
  + Configuration Files: Sample configuration files for environment variables and settings, such as Azure connection strings
* **Deployment Guide**
  + Cloud Deployment: A step-by-step guide to deploying the system to the cloud: setup of Azure resources and configuration of VM instances.
  + Local Testing: Instructions on how to test the application locally before deployment: how to start a server and the worker nodes.

## How Did we Deploy Our System to the Cloud?

### Overview

The system was deployed to the cloud using Azure Virtual Machines and Azure services, which include Blob Storage and Queue Storage. The deployment entailed the setup of the necessary infrastructure, the configuration of the environment, and the launching of the application's components.

Deployment Steps

### Setup Azure Resource:

* + **Blob Storage:** set up Azure Blob Storage containers to store images and results.
  + **Queue Storage:** set up Azure Queue Storage to handle task messages from the master to worker nodes.

### VM Configuration:

* + **Master VM:** we sat up the environment to run the master node on a VM for task distribution and status updates.
  + **Worker VMs**: we sat the environment on multiple VMs to run worker nodes for the processing of tasks.

### Environment Configuration:

* **Environment Variables**: set up environment variables on each VM for Azure connection strings and container names.
* **Dependency Installation:** installed necessary software and libraries, for example, Python, OpenCV, Azure SDK on every VM.

### Code Deployment:

* + **Master Node:** deployed the script of the master node and Flask application to the master VM.
  + **Worker Nodes**: deployed the script of the worker node to each worker VM.
  + **Configuration Files:** uploaded the necessary configuration files to each of the VMs.

### Starting the Application

* + **Flask Server:** run the server to start Flask on the master VM to provide the web interface and handle API requests.
  + **Worker Processes:** run worker processes on each worker VM to start connecting to the master node and begin processing tasks.

### Monitoring and Scaling:

* + **Log Monitoring:** monitor logs during initial deployment to identify any errors or issues.
  + **Dynamic Scaling:** scaled up the number of workers VMs based on the workload to keep the performance optimal.

### Tools and Services Used

* + **Azure Virtual Machines:** to run master and worker nodes.
  + **Azure Blob Storage:** to store images and processed images.
  + **Azure Queue Storage:** to manage messages for tasks.
  + **Flask:** to provide the web interface and API for the system
  + **OpenCV**: to perform image processing tasks.

# CONCLUTION

The project successfully designed, implemented, and tested a distributed image processing system using cloud computing technologies. By leveraging the capabilities of Microsoft Azure for virtual machines, storage, and queue management, we have created a robust and scalable platform capable of handling high volumes of image processing tasks efficiently. The system demonstrated its ability to scale dynamically, handling increased workloads by adding worker nodes. This elasticity ensures that the system can maintain performance and efficiency under varying loads. The system's architecture incorporated fault tolerance mechanisms, ensuring that tasks are reassigned and processed even in the event of worker node failures or network disruptions. This resilience is critical for maintaining continuous operation and reliability. Implementing advanced image processing operations, including watershed segmentation, Canny edge detection, feature matching, and face detection, showcased the system's versatility and capability to handle complex tasks.

The use of Azure Queue Storage for task management and Azure Blob Storage for image storage allowed efficient distribution and processing of tasks across multiple worker nodes. This setup ensured that the system could process multiple images in parallel, significantly reducing processing times. The Flask web application provided a user-friendly interface for uploading images, selecting processing operations, and monitoring task status. This interface makes the system accessible and easy to use for end-users.

# IMPORTANT LINKS

* ***Note: we worked on this project as a team via meetings at the same time in parallel since only one hade the VMs on his device***

## Drive Link

<https://drive.google.com/drive/folders/17sp8Fd2TqU86mgmK_w6eRflxrvl0bLuh?usp=share_link>

## GitHub Link

<https://github.com/ahmed-hesham1221/Image-Processing-on-CLoud->