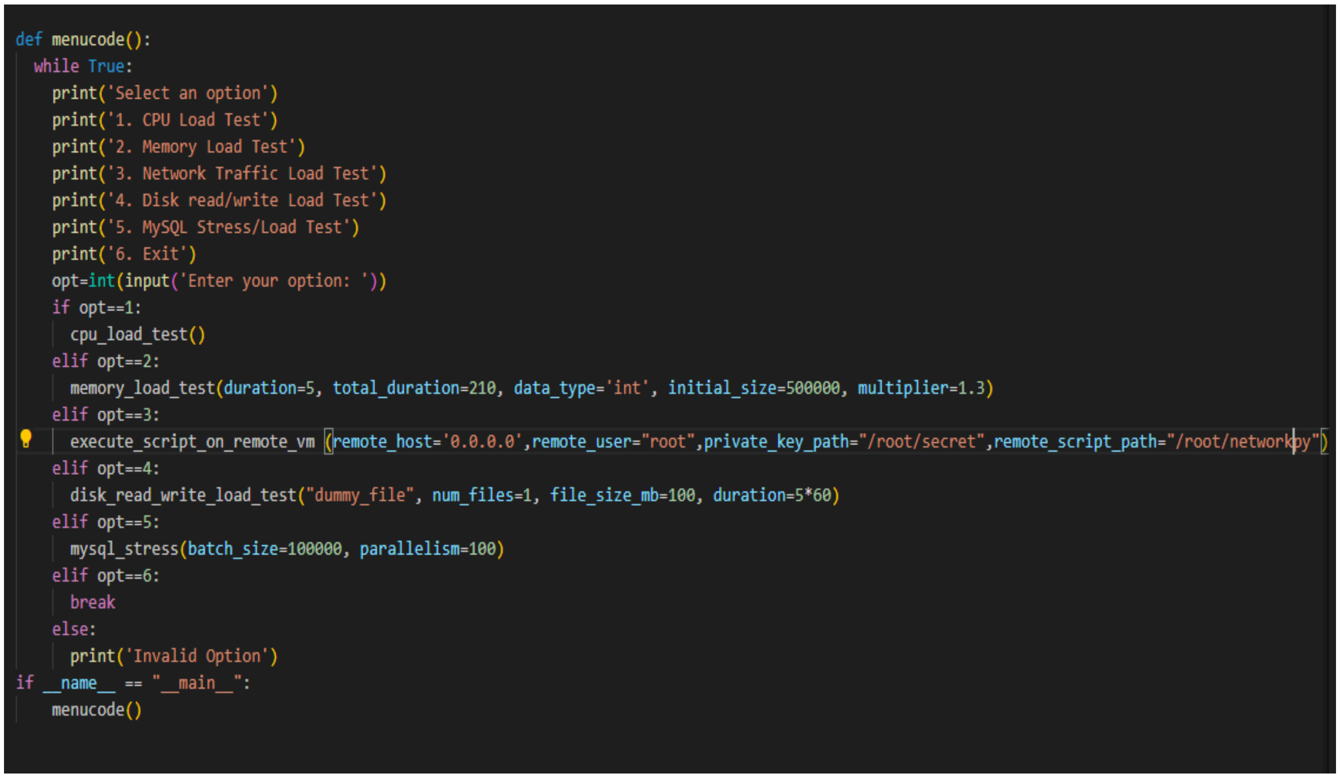
Team Member:

Vyshnavi Tallam
Rishi Devanpalli
Sushma Priyanka
Prasanna Kumar
Vaishnavi Uttarkar

Stress/Load Testing using Python

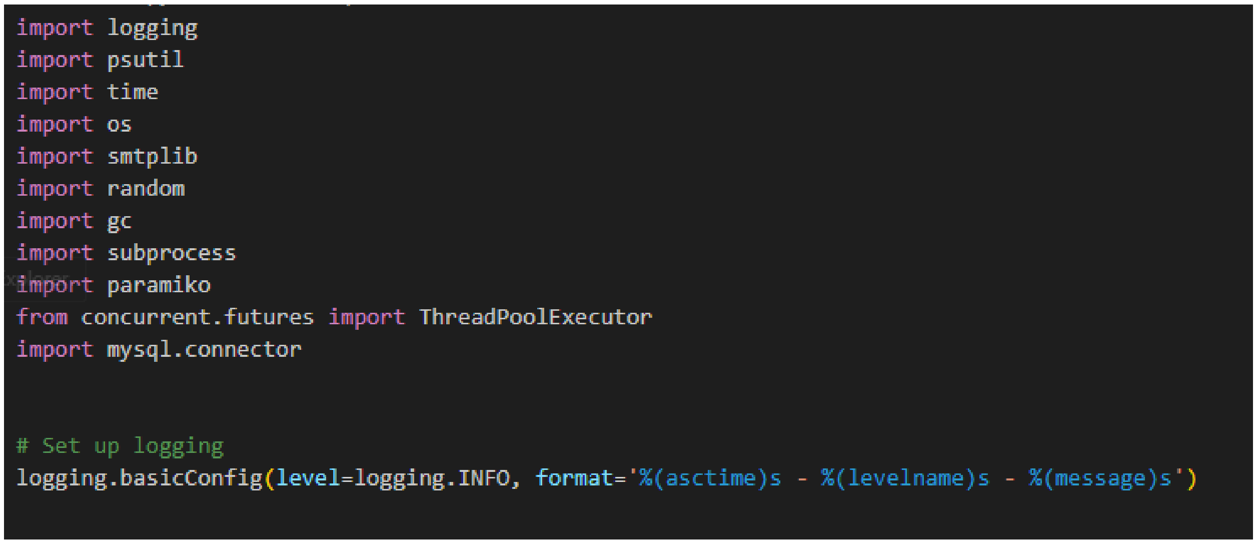
Code Logic:

- **Load Testing Menu:**



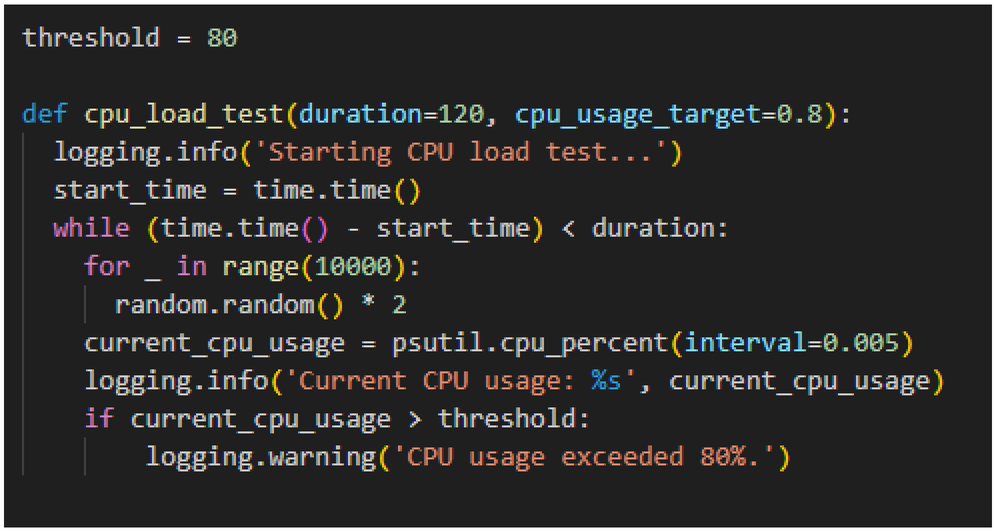
```python
def menucode():
  while True:
    print('Select an option')
    print('1. CPU Load Test')
    print('2. Memory Load Test')
    print('3. Network Traffic Load Test')
    print('4. Disk read/write Load Test')
    print('5. MySQL Stress/Load Test')
    print('6. Exit')
    opt=int(input('Enter your option: '))
    if opt==1:
      cpu_load_test()
    elif opt==2:
      memory_load_test(duration=5, total_duration=210, data_type='int', initial_size=500000, multiplier=1.3)
    elif opt==3:
      execute_script_on_remote_vm (remote_host='0.0.0.0',remote_user="root",private_key_path="/root/secret",remote_script_path="/root/networkpy")
    elif opt==4:
      disk_read_write_load_test("dummy_file", num_files=1, file_size_mb=100, duration=5*60)
    elif opt==5:
      mysql_stress(batch_size=100000, parallelism=100)
    elif opt==6:
      break
    else:
      print('Invalid Option')
if __name__ == "__main__":
    menucode()
```

- **Importing All the Libraries:**



```python
import logging
import psutil
import time
import os
import smtplib
import random
import gc
import subprocess
import paramiko
from concurrent.futures import ThreadPoolExecutor
import mysql.connector


# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
```

1.Cpu Load Testing (Vyshnavi Tallam):



```
threshold = 80

def cpu_load_test(duration=120, cpu_usage_target=0.8):
    logging.info('Starting CPU load test...')
    start_time = time.time()
    while (time.time() - start_time) < duration:
        for _ in range(10000):
            random.random() * 2
        current_cpu_usage = psutil.cpu_percent(interval=0.005)
        logging.info('Current CPU usage: %s', current_cpu_usage)
        if current_cpu_usage > threshold:
            logging.warning('CPU usage exceeded 80%.')
```

- Code Logic and Approach used:

Step 1:
def cpu_load_test(duration=120, cpu_usage_target=0.8):

Next, I created a function named cpu_load_test with two parameters:

- duration=120: The entire duration of the test, expressed in seconds. Here, it's set to run for 2 minutes.
- cpu_usage_target=0.8: The target CPU usage, which is set to 80%.

Step 2:
logging.info('Starting CPU load test...')

This line logs a message to inform that the CPU load test is starting.

Step 3:
start_time = time.time()

I stored the current time at the beginning of the test using time.time(). This will help in determining when to stop the test.

Step 4:
while (time.time() - start_time) < duration:

A while loop is started, and it will keep going as long as the total time is less than the duration, which is 120 seconds.

Step 5:
for _ in range(10000):
random.random() * 2

I created another for loop that runs 10,000 times. It generates random numbers using random and performs a simple multiplication. Although this operation seems trivial, it keeps the CPU busy and helps simulate load.

Step 6:
```
current_cpu_usage = psutil.cpu_percent(interval=0.005)
logging.info('Current CPU usage: %s', current_cpu_usage)
```

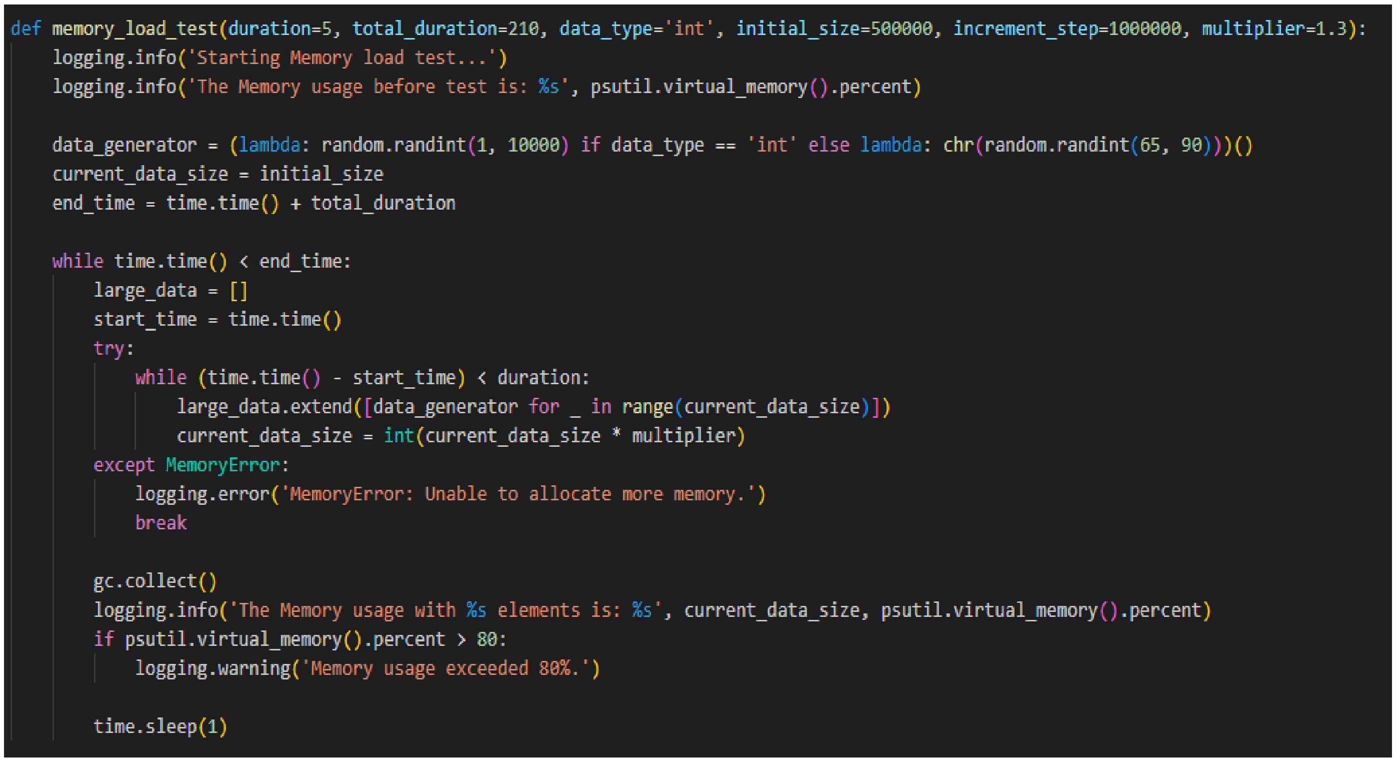
After generating load, the current CPU usage is checked using psutil.cpu_percent(interval=0.005), which measures the CPU usage over a very short interval of 0.005 seconds, which is equal to 5 milliseconds.

The usage percentage is then logged for monitoring.

Step 7:
```
if current_cpu_usage > threshold:
    logging.warning('CPU usage exceeded 80%.')
```

FInally, if and when the current CPU usage exceeds the defined threshold (80%), a warning message is logged to alert that the CPU usage has exceeded the threshold value.

2.Memory Load Testing (Rishi Devanpalli):



```python
def memory_load_test(duration=5, total_duration=210, data_type='int', initial_size=500000, increment_step=1000000, multiplier=1.3):
    logging.info('Starting Memory load test...')
    logging.info('The Memory usage before test is: %s', psutil.virtual_memory().percent)

    data_generator = (lambda: random.randint(1, 10000) if data_type == 'int' else lambda: chr(random.randint(65, 90)))()
    current_data_size = initial_size
    end_time = time.time() + total_duration

    while time.time() < end_time:
        large_data = []
        start_time = time.time()
        try:
            while (time.time() - start_time) < duration:
                large_data.extend([data_generator for _ in range(current_data_size)])
                current_data_size = int(current_data_size * multiplier)
        except MemoryError:
            logging.error('MemoryError: Unable to allocate more memory.')
            break

        gc.collect()
        logging.info('The Memory usage with %s elements is: %s', current_data_size, psutil.virtual_memory().percent)
        if psutil.virtual_memory().percent > 80:
            logging.warning('Memory usage exceeded 80%.')

        time.sleep(1)
```

-Code Logic & Approach used:

Step1:
def memory_load_test(duration=5, total_duration=180, data_type='int', initial_size=500000, increment_step=1000000, multiplier=1.3):

Firstly I defined a function ‘memory_load_test’ and gave parameters:
duration = 5s to run each memory cycle
total_duration = 120s for the code and run and break
Initial_size = The data size the program should initialize
Increment_step = It is amount by which the data size should increase in each iteration
multiplier = To increase the data size by a large amount for it to trigger the 80% threshold

Step2:
data_generator = (lambda: random.randint(1, 10000) if data_type == 'int' else lambda: chr(random.randint(65, 90)))()

I created a ‘data_generator’ function which generates random integers between 1 and 10000

Step3:
current_data_size = initial_size
end_time = time.time() + total_duration
while time.time() < end_time:

Then I set the initial_size and run a loop which will end once the run duration goes beyond 210s

Step4:

```
    large_data = []
    start_time = time.time()
```

Then I created an empty list 'large_data' to store the generated data and 'start_time' to record the start time of the current memory.

Step5:

```
    try:
      while (time.time() - start_time) < duration:
        large_data.extend([data_generator for _ in range(current_data_size)])  #
Generate and store more data
        current_data_size = int(current_data_size * multiplier)
    except MemoryError:
      logging.error('MemoryError: Unable to allocate more memory.')
      break
```

Here the new data generated by 'data_generator' will get appended continuously to the 'large_data' list.
Then the 'current_data_size' is increased by multiplying with the 'multiplier'.
And if the system cannot further allocate memory, it will log an error and break out of the loop.

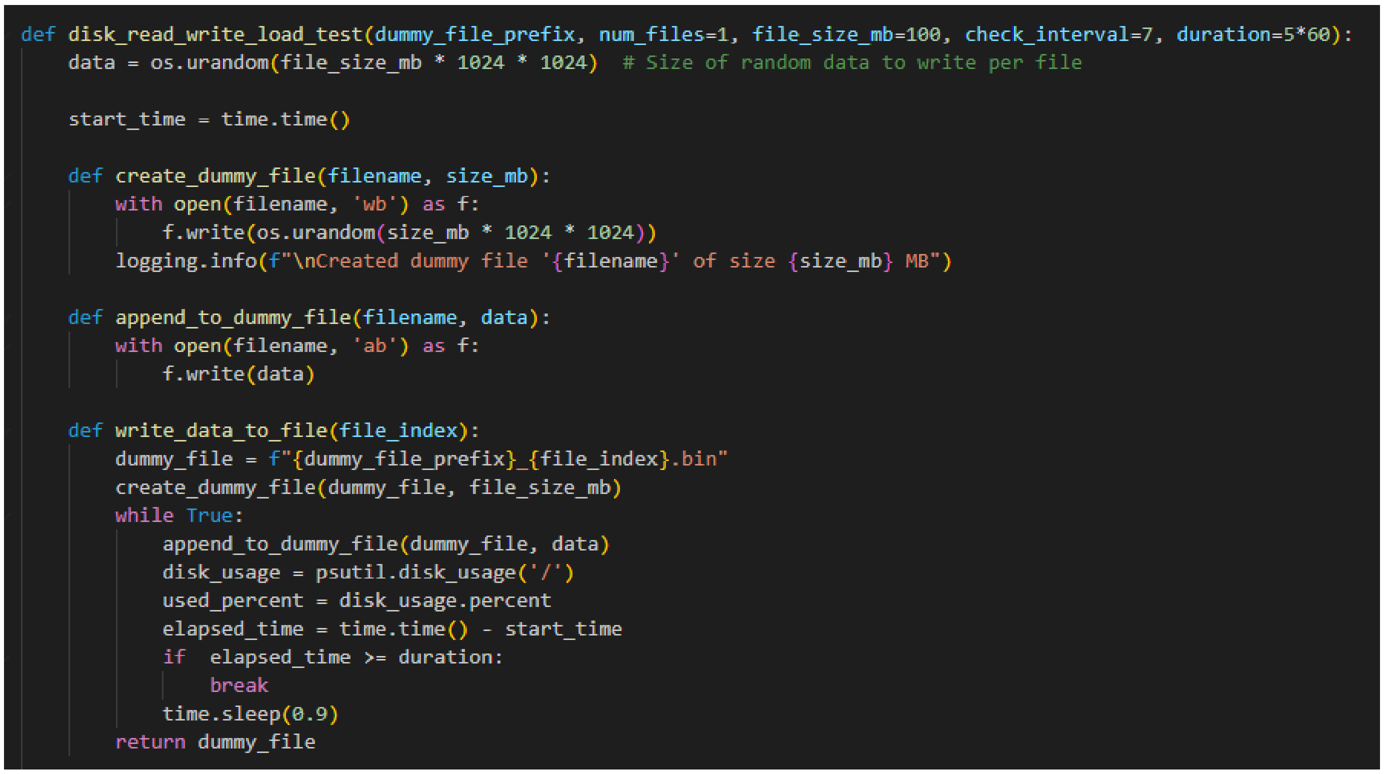
gc.collect() : it triggers garbage collection to free up memory that is no longer in use.

Step6:

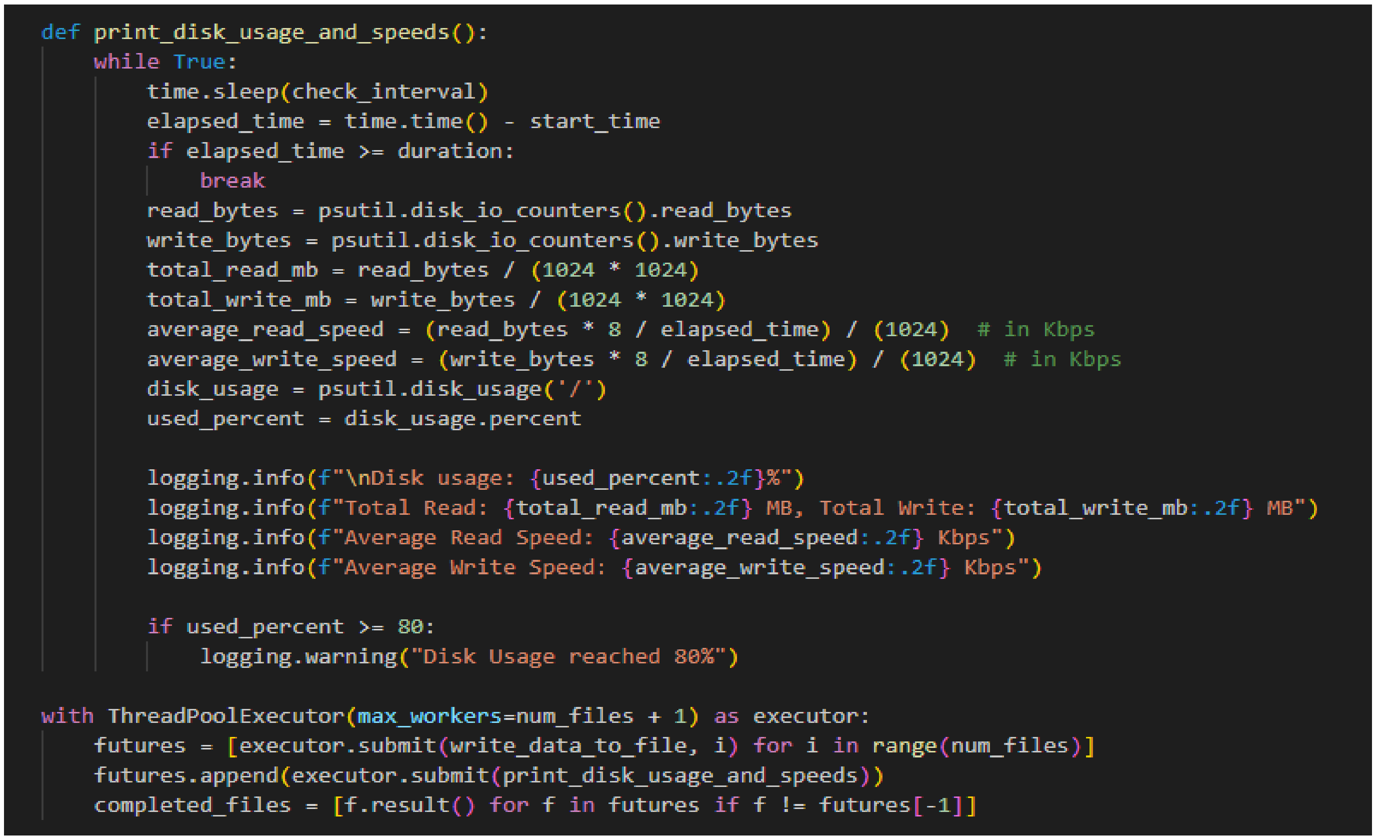
```
    if psutil.virtual_memory().percent > 80:
      logging.warning('Memory usage exceeded 80%.')
```

Finally, it checks if the memory usage exceeds the 80% threshold. If yes it will log a warning.

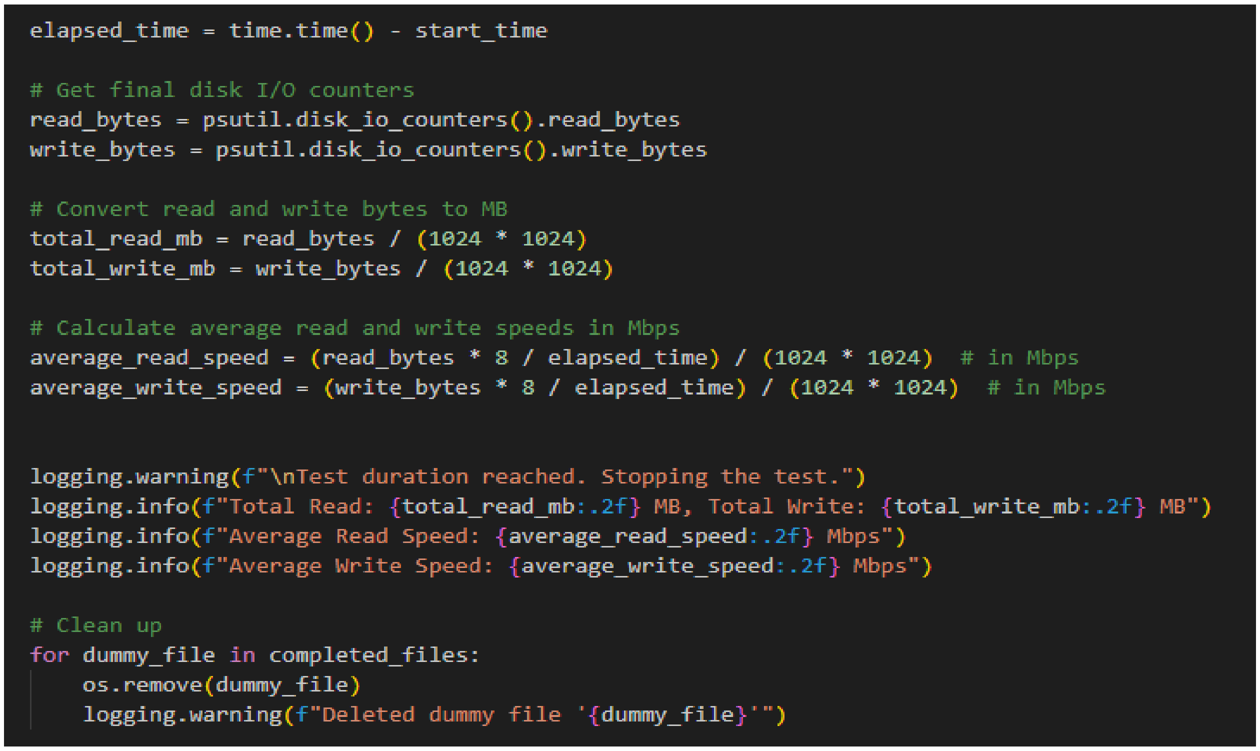
3.Disk Read/Write testing (Sushma Priyanka):



```python
def disk_read_write_load_test(dummy_file_prefix, num_files=1, file_size_mb=100, check_interval=7, duration=5*60):
    data = os.urandom(file_size_mb * 1024 * 1024)  # Size of random data to write per file

    start_time = time.time()

    def create_dummy_file(filename, size_mb):
        with open(filename, 'wb') as f:
            f.write(os.urandom(size_mb * 1024 * 1024))
        logging.info(f"\nCreated dummy file '{filename}' of size {size_mb} MB")

    def append_to_dummy_file(filename, data):
        with open(filename, 'ab') as f:
            f.write(data)

    def write_data_to_file(file_index):
        dummy_file = f"{dummy_file_prefix}_{file_index}.bin"
        create_dummy_file(dummy_file, file_size_mb)
        while True:
            append_to_dummy_file(dummy_file, data)
            disk_usage = psutil.disk_usage('/')
            used_percent = disk_usage.percent
            elapsed_time = time.time() - start_time
            if  elapsed_time >= duration:
                break
            time.sleep(0.9)
        return dummy_file
```



```python
    def print_disk_usage_and_speeds():
        while True:
            time.sleep(check_interval)
            elapsed_time = time.time() - start_time
            if elapsed_time >= duration:
                break
            read_bytes = psutil.disk_io_counters().read_bytes
            write_bytes = psutil.disk_io_counters().write_bytes
            total_read_mb = read_bytes / (1024 * 1024)
            total_write_mb = write_bytes / (1024 * 1024)
            average_read_speed = (read_bytes * 8 / elapsed_time) / (1024)  # in Kbps
            average_write_speed = (write_bytes * 8 / elapsed_time) / (1024)  # in Kbps
            disk_usage = psutil.disk_usage('/')
            used_percent = disk_usage.percent

            logging.info(f"\nDisk usage: {used_percent:.2f}%")
            logging.info(f"Total Read: {total_read_mb:.2f} MB, Total Write: {total_write_mb:.2f} MB")
            logging.info(f"Average Read Speed: {average_read_speed:.2f} Kbps")
            logging.info(f"Average Write Speed: {average_write_speed:.2f} Kbps")

            if used_percent >= 80:
                logging.warning("Disk Usage reached 80%")

    with ThreadPoolExecutor(max_workers=num_files + 1) as executor:
        futures = [executor.submit(write_data_to_file, i) for i in range(num_files)]
        futures.append(executor.submit(print_disk_usage_and_speeds))
        completed_files = [f.result() for f in futures if f != futures[-1]]
```



```python
elapsed_time = time.time() - start_time

# Get final disk I/O counters
read_bytes = psutil.disk_io_counters().read_bytes
write_bytes = psutil.disk_io_counters().write_bytes

# Convert read and write bytes to MB
total_read_mb = read_bytes / (1024 * 1024)
total_write_mb = write_bytes / (1024 * 1024)

# Calculate average read and write speeds in Mbps
average_read_speed = (read_bytes * 8 / elapsed_time) / (1024 * 1024)  # in Mbps
average_write_speed = (write_bytes * 8 / elapsed_time) / (1024 * 1024)  # in Mbps


logging.warning(f"\nTest duration reached. Stopping the test.")
logging.info(f"Total Read: {total_read_mb:.2f} MB, Total Write: {total_write_mb:.2f} MB")
logging.info(f"Average Read Speed: {average_read_speed:.2f} Mbps")
logging.info(f"Average Write Speed: {average_write_speed:.2f} Mbps")

# Clean up
for dummy_file in completed_files:
    os.remove(dummy_file)
    logging.warning(f"Deleted dummy file '{dummy_file}'")
```

-Code Logic & Approach used:

Step-1:
def disk_read_write_load_test(dummy_file_prefix, num_files=1, file_size_mb=100, check_interval=7, duration=5*60):

This function defines a disk read/write load test by creating a specified number of dummy files with a given size, checking performance at regular intervals, and running the test for a set duration in seconds.

Step-2 :
data = os.urandom(file_size_mb * 1024 * 1024) # Size of random data to write per file

This line generates random data with a size equal to file_size_mb megabytes to be written into each dummy file.

Step-3:
start_time = time.time()

This line records the start time of the test.

Step-4:
def create_dummy_file(filename, size_mb):

This is a helper function to create a dummy file. It writes random data of size size_mb to the specified filename . By using the logging function we are printing the line as a dummy file is created.

Step-5:
def append_to_dummy_file(filename, data):

This append_to_dummy_file function appends data to an existing dummy file**.**

Step-6:
def write_data_to_file(file_index):

The write_data_to_file function creates and continuously appends data to a dummy file named based on file_index, monitoring disk usage and elapsed time. It runs an infinite loop, writing data to the file and checking if the test duration has been exceeded, breaking the loop if so. It also introduces a 0.9-second delay between iterations and returns the filename of the dummy file after completion.

Step-7:
def print_disk_usage_and_speeds():

This function continuously monitors disk usage and read/write speeds, pausing for a specified interval between checks. It calculates total data read and written, average read/write speeds, and disk usage percentage, logging these values at each interval. If disk usage exceeds 80%, a warning is logged, and the loop breaks after the specified duration.

Step-8:
with ThreadPoolExecutor(max_workers=num_files + 1) as executor:

This block of code uses a ThreadPoolExecutor to execute the write_data_to_file function for each dummy file and the print_disk_usage_and_speeds function concurrently. The max_workers argument specifies the maximum number of worker threads to use.

Step-8:
elapsed_time = time.time() - start_time

This line calculates the elapsed time since the test began by subtracting start_time from the current time (time.time()).

Step-9:
read_bytes = psutil.disk_io_counters().read_bytes
write_bytes = psutil.disk_io_counters().write_bytes

These lines use psutil.disk_io_counters().read_bytes and write_bytes to retrieve the total bytes read from and written to the disk since system boot, which are used to calculate disk I/O speeds and monitor activity during the test.

Step-10:
total_read_mb = read_bytes / (1024 * 1024)
total_write_mb = write_bytes / (1024 * 1024)

These lines convert the total read and write bytes to megabytes.

Step-11:
average_read_speed = (read_bytes * 8 / elapsed_time) / (1024 * 1024) # in Mbps
average_write_speed = (write_bytes * 8 / elapsed_time) / (1024 * 1024) # in Mbps

These lines calculate the average read and write speeds in megabits per second (Mbps) by converting the total bytes read and written to bits, dividing by elapsed time to get bits per second, and then converting to megabits per second, providing a measure of disk performance during the test.

Step-12:

```
logging.warning(f"\nTest duration reached. Stopping the test.")
logging.info(f"Total Read: {total_read_mb:.2f} MB, Total Write: {total_write_mb:.2f} MB")
logging.info(f"Average Read Speed: {average_read_speed:.2f} Mbps")
logging.info(f"Average Write Speed: {average_write_speed:.2f} Mbps")
```

These logging statements record a warning that the test duration has ended, and provide informational logs detailing the total data read and written in megabytes, as well as the average read and write speeds in megabits per second.

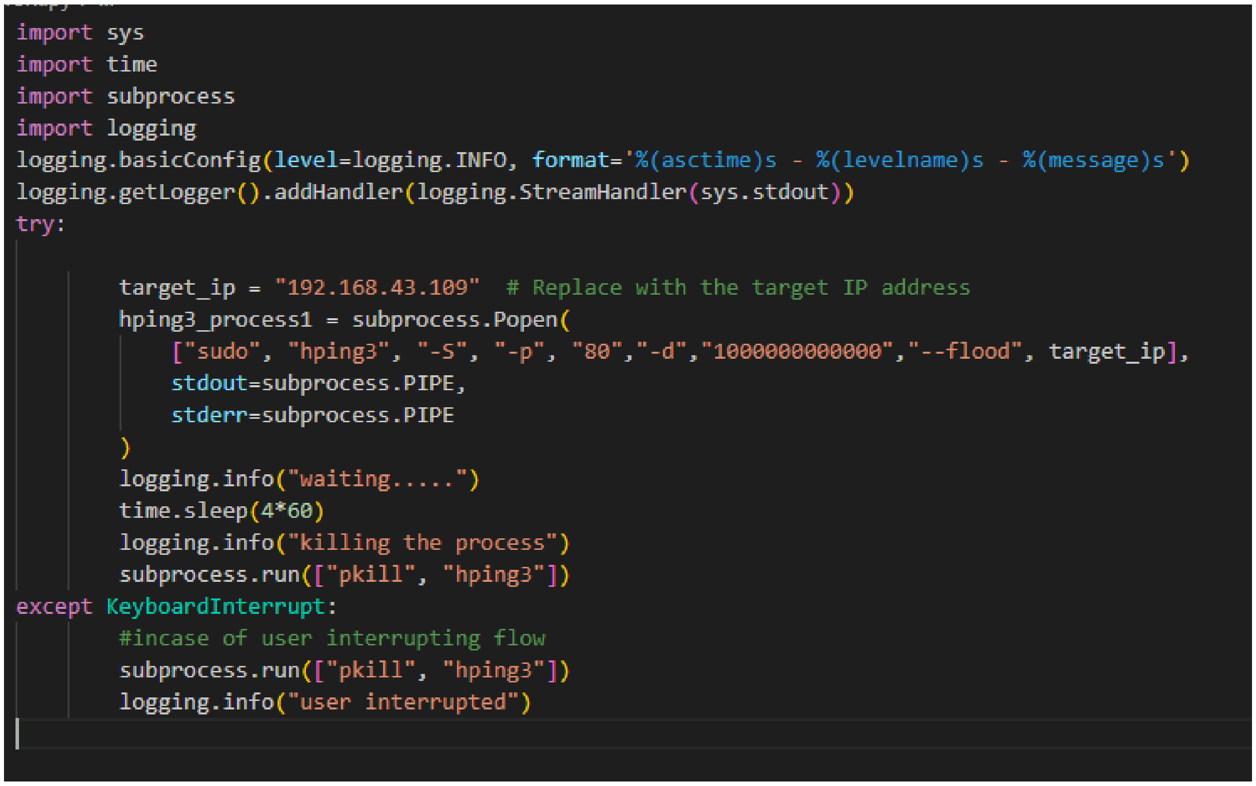
Step-13:

```
for dummy_file in completed_files:
```

This loop iterates over a list of completed files, deletes each one from the system, and logs a warning message indicating that the specific dummy file has been deleted.

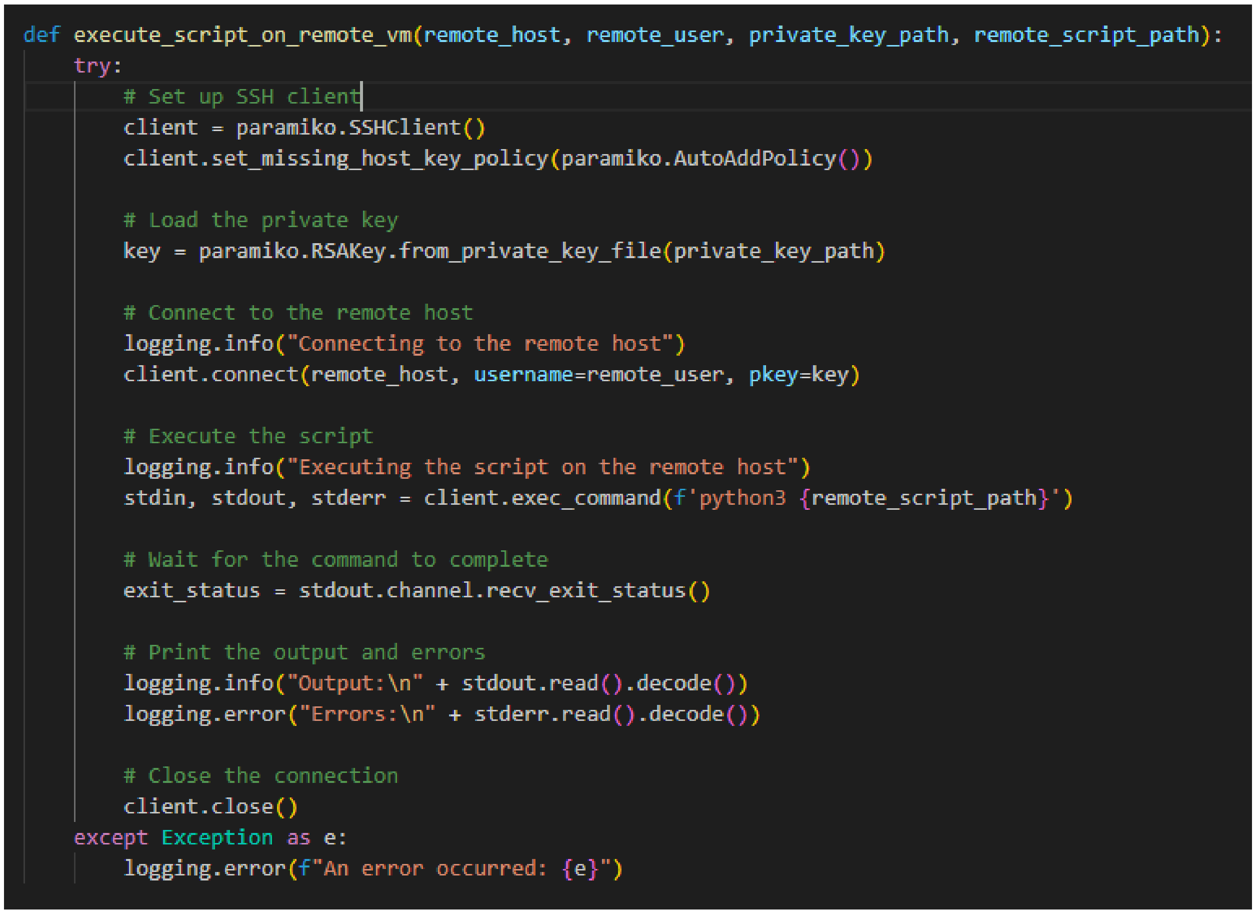
4.Network Traffic Testing (T R Prasannakumar Nayak):

Code executed from Grafana server



```
import sys
import time
import subprocess
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logging.getLogger().addHandler(logging.StreamHandler(sys.stdout))
try:

        target_ip = "192.168.43.109"  # Replace with the target IP address
        hping3_process1 = subprocess.Popen(
            ["sudo", "hping3", "-S", "-p", "80","-d","1000000000000","--flood", target_ip],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE
        )
        logging.info("waiting.....")
        time.sleep(4*60)
        logging.info("killing the process")
        subprocess.run(["pkill", "hping3"])
except KeyboardInterrupt:
        #incase of user interrupting flow
        subprocess.run(["pkill", "hping3"])
        logging.info("user interrupted")
```

Code executed from Node_exporter server:



```
def execute_script_on_remote_vm(remote_host, remote_user, private_key_path, remote_script_path):
    try:
        # Set up SSH client
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        # Load the private key
        key = paramiko.RSAKey.from_private_key_file(private_key_path)

        # Connect to the remote host
        logging.info("Connecting to the remote host")
        client.connect(remote_host, username=remote_user, pkey=key)

        # Execute the script
        logging.info("Executing the script on the remote host")
        stdin, stdout, stderr = client.exec_command(f'python3 {remote_script_path}')

        # Wait for the command to complete
        exit_status = stdout.channel.recv_exit_status()

        # Print the output and errors
        logging.info("Output:\n" + stdout.read().decode())
        logging.error("Errors:\n" + stderr.read().decode())

        # Close the connection
        client.close()
    except Exception as e:
        logging.error(f"An error occurred: {e}")
```

-Code Logic & Approach used:

Step 1: Import Required Libraries

- Imported required libraries like **time, subprocess and logging**

Step 2: Set Up Logging Configuration

- **logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')** Configure logging to capture messages at the INFO level or higher and format them with a timestamp, severity level, and message.

Step 3: Define Target IP

-
  ```
  try:
      target_ip = "192.168.72.234"  # Replace with the target IP address
  ```
  Set the IP address of the target. The **try** block begins here to catch any exceptions that might occur.

Step 4: Start hping3 Process

-
  ```
  hping3_process1 = subprocess.Popen(
      ["sudo", "hping3", "-S", "-p", "80", "-d", "1400", "--flood", target_ip],
      stdout=subprocess.PIPE,
      stderr=subprocess.PIPE
  )
  ```
  Execute the **hping3** command with specified options to start a SYN flood attack on the target IP. This command is run with superuser privileges. Output and error streams are captured.

Step 5: Wait for a Specified Time

-
  ```
  time.sleep(120)
  ```
  Pause the script execution for 120 seconds (2 minutes).

Step 6: Terminate hping3 Process

-
  ```
  subprocess.run(["pkill", "hping3"])
  ```
  Execute the **pkill** command to kill the **hping3** process.

Step 7: Handle KeyboardInterrupt Exception

-
  ```
  except KeyboardInterrupt:
      #incase of user interrupting flow
      subprocess.run(["pkill", "hping3"])
      logging.info("user interrupted")
  ```

If the script is interrupted by the user (e.g., by pressing Ctrl+C):
Terminate the **hping3** process to ensure it doesn't continue running.Log a message indicating that the user interrupted the script.

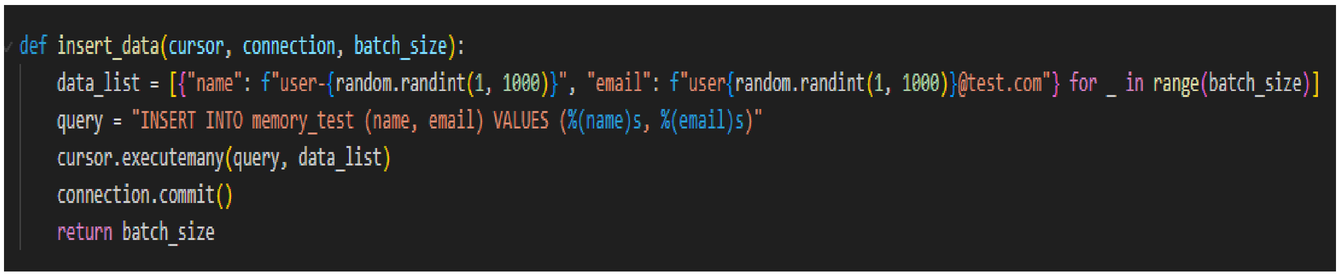
Code executed from Node_exporter server:

This code snippet defines a Python function called `execute_script_on_remote_vm` that allows you to remotely execute a Python script on another machine (virtual machine) over SSH.

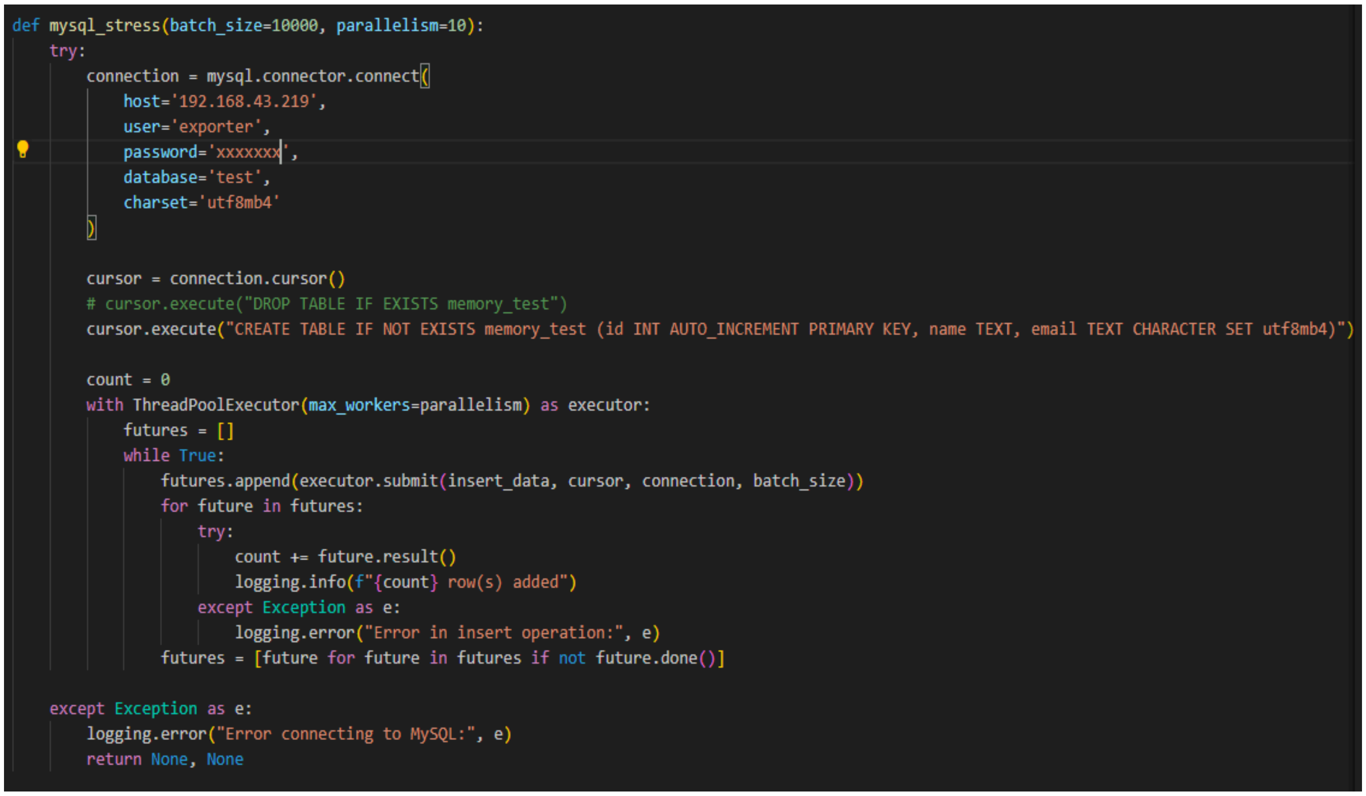
- **Setting Up SSH Connection:**
    - Imports the `paramiko` library which enables SSH connections in Python.
    - client = paramiko.SSHClient()
    - client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    - Creates an SSH client object.
    - Sets up automatic addition of new SSH host keys (if encountered for the first time).
- **Loading Private Key:**
    - Reads the provided private key file using the `paramiko.RSAKey` class. This key is used for authentication on the remote machine.
    - key = paramiko.RSAKey.from_private_key_file(private_key_path)
- **Connecting to Remote Host:**
    - Establishes an SSH connection to the target machine using the provided hostname, username, and the loaded private key.
    - client.connect(remote_host, username=remote_user, pkey=key)
- **Executing the Script:**
    - Constructs the command to execute the remote script using `f-strings`. The command likely involves running the script using `python3`.
    - Executes the command on the remote machine using `client.exec_command`.
    - Captures the standard output and error streams of the executed script.
    - stdin, stdout, stderr = client.exec_command(f'python3 {remote_script_path}')
- **Handling Output and Errors:**
    - Waits for the script to finish running.
    - Reads and decodes the standard output and error streams from the remote execution.
    - Logs the output and errors using logging library (assuming it's imported).
- **Closing Connection:**
    - Closes the SSH connection to the remote machine.
- **Exception Handling:**
    - Catches any exceptions that might occur during the process and logs the error message.

Overall, this function provides a way to automate script execution on remote machines securely using SSH and private key authentication.

5. MySQL Load Testing (Vaishnavi Uttarkar):



```python
def insert_data(cursor, connection, batch_size):
    data_list = [{"name": f"user-{random.randint(1, 1000)}", "email": f"user{random.randint(1, 1000)}@test.com"} for _ in range(batch_size)]
    query = "INSERT INTO memory_test (name, email) VALUES (%(name)s, %(email)s)"
    cursor.executemany(query, data_list)
    connection.commit()
    return batch_size
```



```python
def mysql_stress(batch_size=10000, parallelism=10):
    try:
        connection = mysql.connector.connect(
            host='192.168.43.219',
            user='exporter',
            password='xxxxxxx',
            database='test',
            charset='utf8mb4'
        )

        cursor = connection.cursor()
        # cursor.execute("DROP TABLE IF EXISTS memory_test")
        cursor.execute("CREATE TABLE IF NOT EXISTS memory_test (id INT AUTO_INCREMENT PRIMARY KEY, name TEXT, email TEXT CHARACTER SET utf8mb4)")

        count = 0
        with ThreadPoolExecutor(max_workers=parallelism) as executor:
            futures = []
            while True:
                futures.append(executor.submit(insert_data, cursor, connection, batch_size))
                for future in futures:
                    try:
                        count += future.result()
                        logging.info(f"{count} row(s) added")
                    except Exception as e:
                        logging.error("Error in insert operation:", e)
                futures = [future for future in futures if not future.done()]

    except Exception as e:
        logging.error("Error connecting to MySQL:", e)
        return None, None
```

- Code Logic & Approach used:

- **Code Logic:**

Step 1:

def insert_data(cursor, connection, batch_size):

This line defines a function named insert_data which takes three parameters:

- **cursor**: Database cursor for executing queries.
- **connection**: Database connection object for committing transactions.
- **batch_size**: Number of rows to insert in each batch.

Step 2:

data_list = [{"name": f"user-{random.randint(1, 1000)}", "email": f"user{random.randint(1, 1000)}@test.com"} for _ in range(batch_size)]

This line generates a list of dictionaries, each containing random `name` and `email` values. The size of the list is determined by the `batch_size` parameter.

Step 3:

```
query = "INSERT INTO memory_test (name, email) VALUES (%(name)s, %(email)s)"
cursor.executemany(query, data_list)
```

These lines prepare and execute a batch insert SQL query using the generated data.

Step 4:

```
connection.commit()
```

This line commits the transaction to the database.

Step 5:

```
return batch_size
```

This line returns the `batch_size` after successful insertion.

Step 6:

```
def mysql_stress(batch_size=10000, parallelism=10):
```

This line defines a function named `mysql_stress` which takes two parameters:

- **batch_size**: Number of rows to insert per batch (default is 10,000).
- **parallelism**: Number of concurrent insertion tasks (default is 10).

Step 7:

```
try:
    connection = mysql.connector.connect(
        host='192.168.43.219',
        user='exporter',
        password=Somu@123',
        database='test',
        charset='utf8mb4'
    )
```

These lines attempt to connect to a MySQL database using the provided credentials and connection details.

Step 8:

```
cursor = connection.cursor()
cursor.execute("CREATE TABLE IF NOT EXISTS memory_test (id INT AUTO_INCREMENT
PRIMARY KEY, name TEXT, email TEXT CHARACTER SET utf8mb4)")
```

These lines create a cursor and ensure the `memory_test` table exists, creating it if it does not exist.

Step 9:

```
count = 0
with ThreadPoolExecutor(max_workers=parallelism) as executor:
    futures = []
```

These lines initialize a row count (`count`) and create a `ThreadPoolExecutor` with a specified level of parallelism to manage concurrent data insert operations.

Step 10:

```
    while True:
        futures.append(executor.submit(insert_data, cursor, connection, batch_size))
        for future in futures:
            try:
                count += future.result()
                logging.info(f"{count} row(s) added")
            except Exception as e:
                logging.error("Error in insert operation:", e)
        futures = [future for future in futures if not future.done()]
```

These lines continuously submit `insert_data` tasks to the executor, monitor and log the number of rows added, handle exceptions for insertion operations, and filter completed futures from the list.

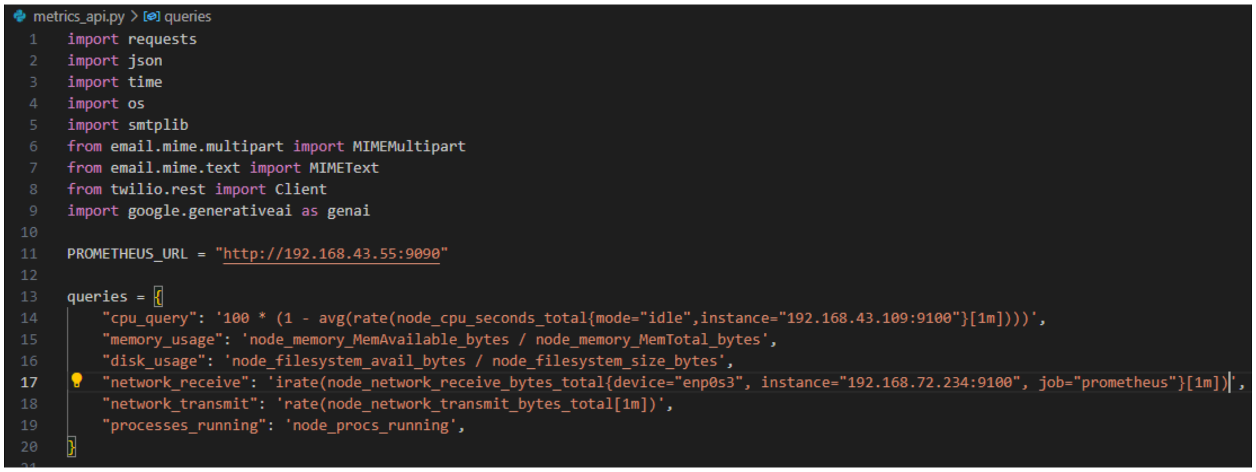
Step 11:

```
except Exception as e:
    logging.error("Error connecting to MySQL:", e)
    return None, None
```

These lines handle exceptions for the database connection, logging any errors encountered, and return `None` in case of an error.

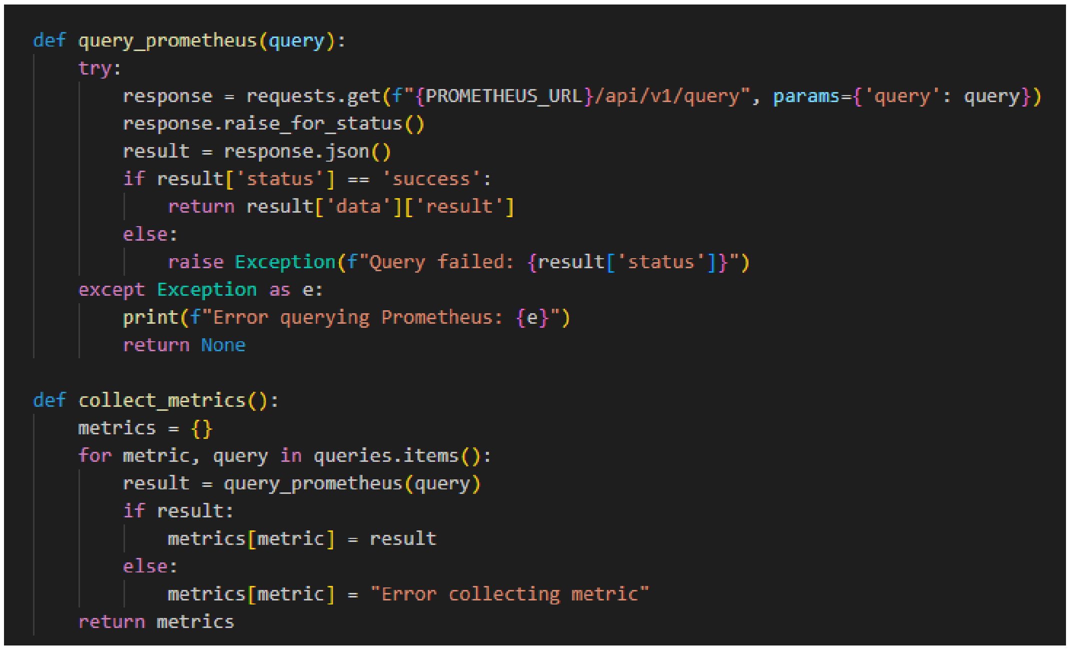
Insights with Notification

1. Collecting Metrics:

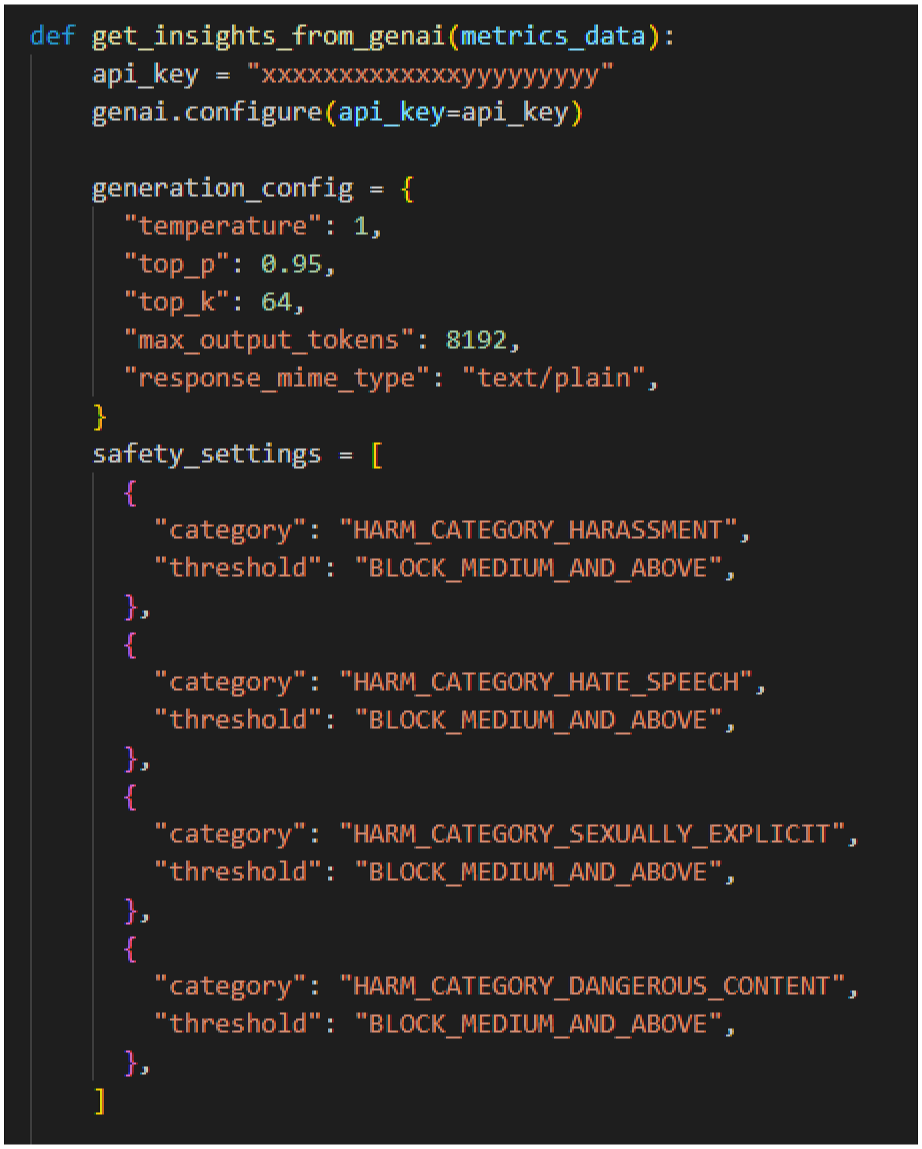


metrics_api.py > queries

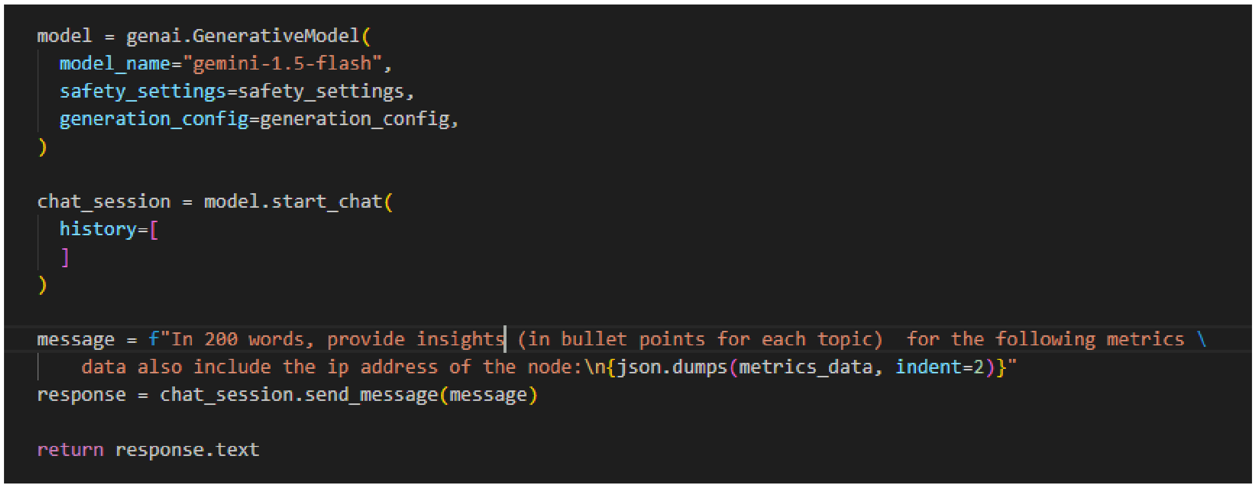
```python
import requests
import json
import time
import os
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from twilio.rest import Client
import google.generativeai as genai

PROMETHEUS_URL = "http://192.168.43.55:9090"

queries = {
    "cpu_query": '100 * (1 - avg(rate(node_cpu_seconds_total{mode="idle",instance="192.168.43.109:9100"}[1m])))',
    "memory_usage": 'node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes',
    "disk_usage": 'node_filesystem_avail_bytes / node_filesystem_size_bytes',
    "network_receive": 'irate(node_network_receive_bytes_total{device="enp0s3", instance="192.168.72.234:9100", job="prometheus"}[1m])',
    "network_transmit": 'rate(node_network_transmit_bytes_total[1m])',
    "processes_running": 'node_procs_running',
}
```



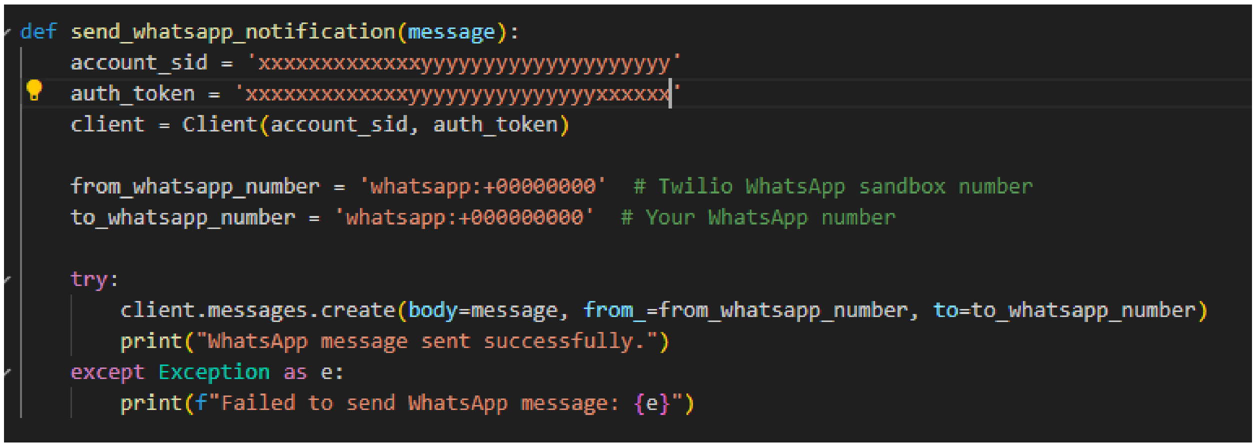
```python
def query_prometheus(query):
    try:
        response = requests.get(f"{PROMETHEUS_URL}/api/v1/query", params={'query': query})
        response.raise_for_status()
        result = response.json()
        if result['status'] == 'success':
            return result['data']['result']
        else:
            raise Exception(f"Query failed: {result['status']}")
    except Exception as e:
        print(f"Error querying Prometheus: {e}")
        return None

def collect_metrics():
    metrics = {}
    for metric, query in queries.items():
        result = query_prometheus(query)
        if result:
            metrics[metric] = result
        else:
            metrics[metric] = "Error collecting metric"
    return metrics
```



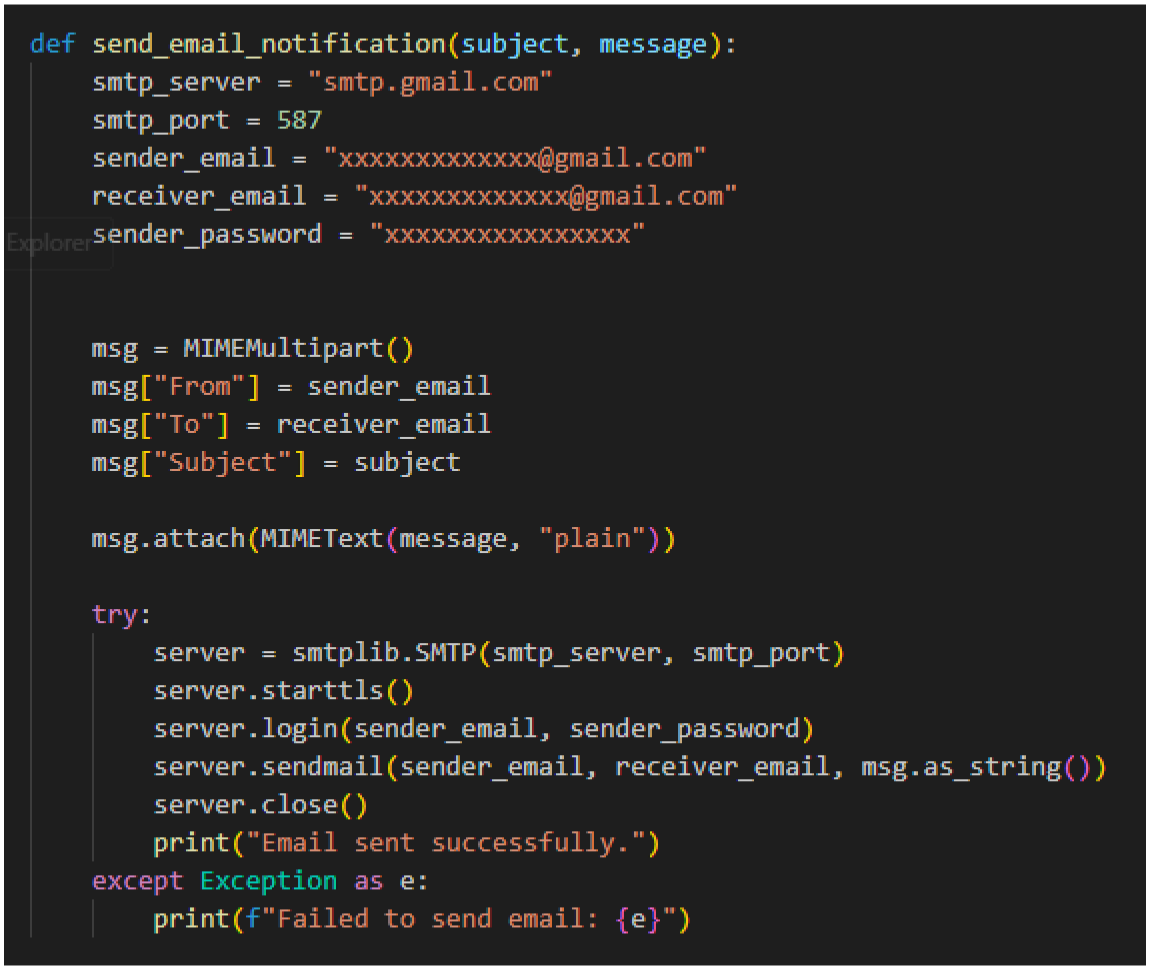
```
def get_insights_from_genai(metrics_data):
    api_key = "xxxxxxxxxxxxxyyyyyyyyy"
    genai.configure(api_key=api_key)

    generation_config = {
      "temperature": 1,
      "top_p": 0.95,
      "top_k": 64,
      "max_output_tokens": 8192,
      "response_mime_type": "text/plain",
    }
    safety_settings = [
      {
        "category": "HARM_CATEGORY_HARASSMENT",
        "threshold": "BLOCK_MEDIUM_AND_ABOVE",
      },
      {
        "category": "HARM_CATEGORY_HATE_SPEECH",
        "threshold": "BLOCK_MEDIUM_AND_ABOVE",
      },
      {
        "category": "HARM_CATEGORY_SEXUALLY_EXPLICIT",
        "threshold": "BLOCK_MEDIUM_AND_ABOVE",
      },
      {
        "category": "HARM_CATEGORY_DANGEROUS_CONTENT",
        "threshold": "BLOCK_MEDIUM_AND_ABOVE",
      },
    ]
```



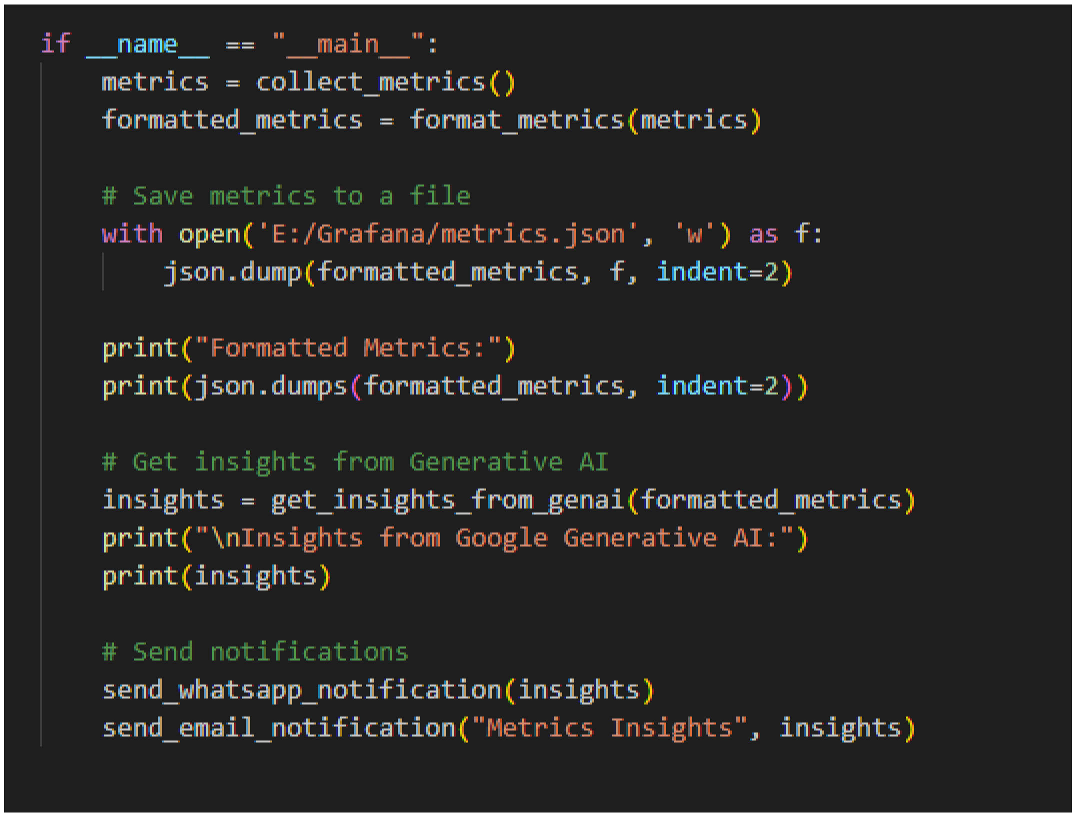
```
model = genai.GenerativeModel(
  model_name="gemini-1.5-flash",
  safety_settings=safety_settings,
  generation_config=generation_config,
)

chat_session = model.start_chat(
  history=[
  ]
)

message = f"In 200 words, provide insights (in bullet points for each topic)  for the following metrics \
    data also include the ip address of the node:\n{json.dumps(metrics_data, indent=2)}"
response = chat_session.send_message(message)

return response.text
```



```python
def send_whatsapp_notification(message):
    account_sid = 'xxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyy'
    auth_token = 'xxxxxxxxxxxxxyyyyyyyyyyyyyyyyxxxxxx'
    client = Client(account_sid, auth_token)

    from_whatsapp_number = 'whatsapp:+00000000'  # Twilio WhatsApp sandbox number
    to_whatsapp_number = 'whatsapp:+000000000'  # Your WhatsApp number

    try:
        client.messages.create(body=message, from_=from_whatsapp_number, to=to_whatsapp_number)
        print("WhatsApp message sent successfully.")
    except Exception as e:
        print(f"Failed to send WhatsApp message: {e}")
```



```python
def send_email_notification(subject, message):
    smtp_server = "smtp.gmail.com"
    smtp_port = 587
    sender_email = "xxxxxxxxxxxxx@gmail.com"
    receiver_email = "xxxxxxxxxxxxx@gmail.com"
    sender_password = "xxxxxxxxxxxxxxxx"


    msg = MIMEMultipart()
    msg["From"] = sender_email
    msg["To"] = receiver_email
    msg["Subject"] = subject

    msg.attach(MIMEText(message, "plain"))

    try:
        server = smtplib.SMTP(smtp_server, smtp_port)
        server.starttls()
        server.login(sender_email, sender_password)
        server.sendmail(sender_email, receiver_email, msg.as_string())
        server.close()
        print("Email sent successfully.")
    except Exception as e:
        print(f"Failed to send email: {e}")
```



```python
if __name__ == "__main__":
    metrics = collect_metrics()
    formatted_metrics = format_metrics(metrics)

    # Save metrics to a file
    with open('E:/Grafana/metrics.json', 'w') as f:
        json.dump(formatted_metrics, f, indent=2)

    print("Formatted Metrics:")
    print(json.dumps(formatted_metrics, indent=2))

    # Get insights from Generative AI
    insights = get_insights_from_genai(formatted_metrics)
    print("\nInsights from Google Generative AI:")
    print(insights)

    # Send notifications
    send_whatsapp_notification(insights)
    send_email_notification("Metrics Insights", insights)
```

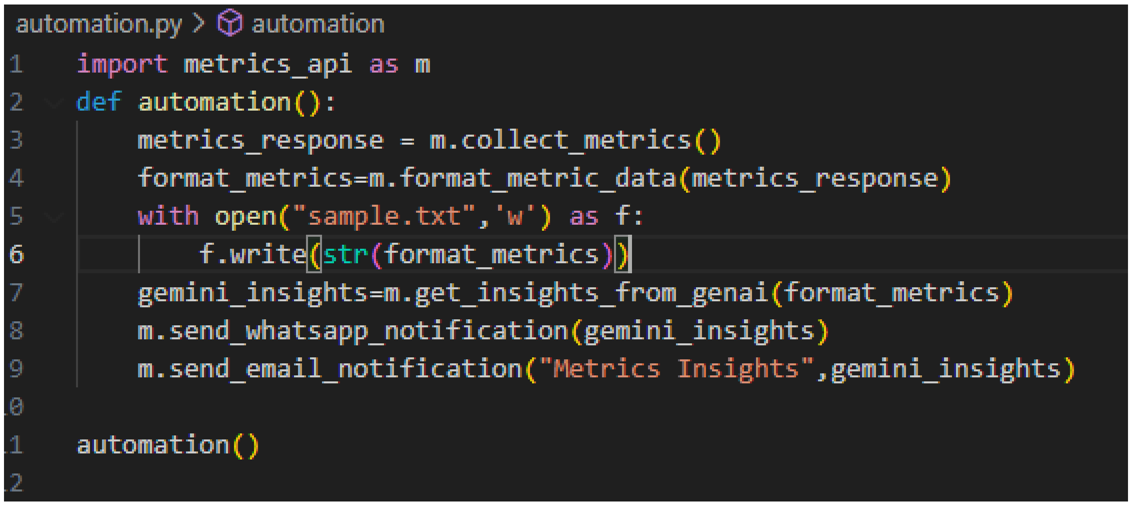
1. **Importing Required Libraries**:
   - `requests`: For making HTTP requests to Prometheus.
   - `json`: For handling JSON data.
   - `time`: For timestamping.
   - `os`: For file operations.
   - `smtplib`: For sending email.
   - `email.mime`: For creating email messages.
   - `twilio`: For sending WhatsApp messages.
   - `google.generativeai`: For accessing Google's Generative AI.
2. **Prometheus Configuration**:
   - `PROMETHEUS_URL`: URL of Prometheus server.
3. **Queries**:
   - Prometheus queries defined in `queries` dictionary for different metrics like CPU usage, memory usage, disk usage, etc.
4. **Functions**:
   - `query_prometheus(query)`: Function to query Prometheus server with provided query.
   - `collect_metrics()`: Collects metrics by querying Prometheus for each metric defined in `queries`.
   - `format_metric_data(result)`: Formats the metric data obtained from Prometheus.
   - `format_metrics(metrics)`: Formats all collected metrics.
   - `get_insights_from_genai(metrics_data)`: Generates insights using Google Generative AI based on the metrics data provided.

- `send_whatsapp_notification(message)`: Sends WhatsApp notification using Twilio.
- `send_email_notification(subject, message)`: Sends email notification.

5. **Main Execution**:
    - Collect metrics using `collect_metrics()`.
    - Format the collected metrics.
    - Save metrics to a JSON file.
    - Get insights using Generative AI.
    - Send WhatsApp and email notifications with the insights.
6. **Sending Notifications**:
    - WhatsApp notification is sent using Twilio's API.
    - Email notification is sent using SMTP server.
7. **Google Generative AI**:
    - It generates insights based on the provided metrics data.
    - The generated insights are limited to 100 words.
    - Safety settings are applied to filter out harmful content.

It's a comprehensive approach where Prometheus is used for collecting metrics, Generative AI is used for generating insights, and notifications are sent via WhatsApp and email.

automation.py



automation.py > automation

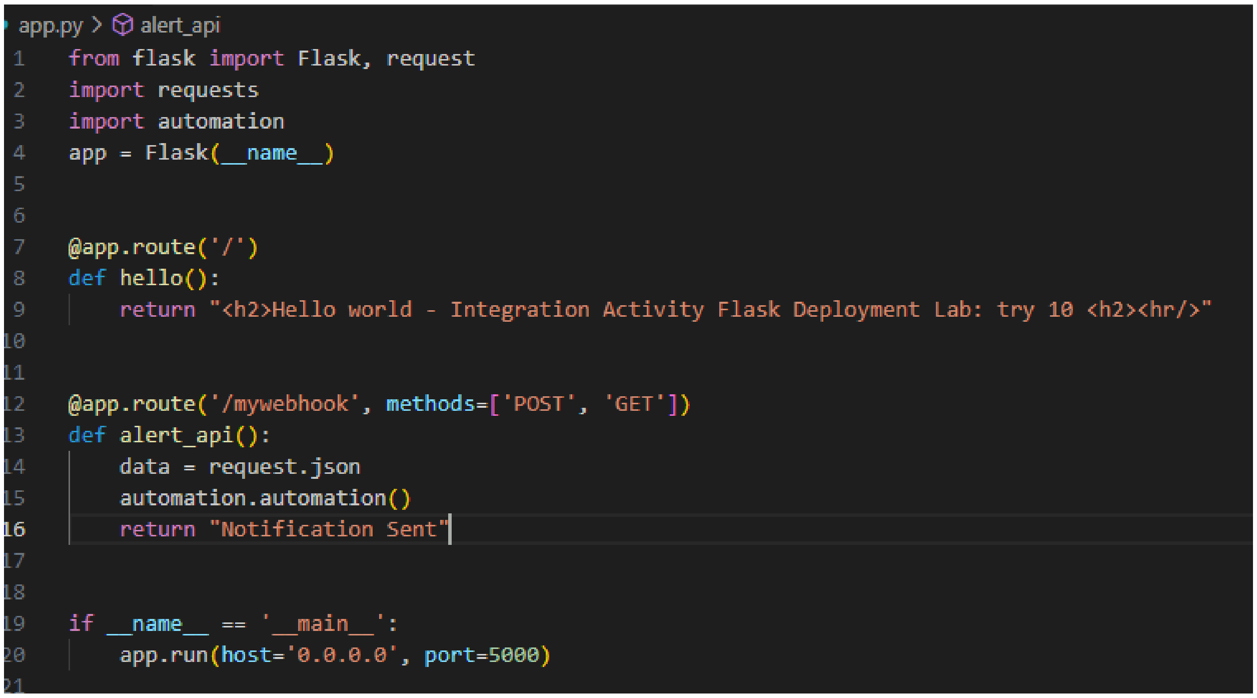
```python
import metrics_api as m
def automation():
    metrics_response = m.collect_metrics()
    format_metrics=m.format_metric_data(metrics_response)
    with open("sample.txt",'w') as f:
        f.write(str(format_metrics))
    gemini_insights=m.get_insights_from_genai(format_metrics)
    m.send_whatsapp_notification(gemini_insights)
    m.send_email_notification("Metrics Insights",gemini_insights)

automation()
```

1. **Metric Collection**:
    - `metrics_response = m.collect_metrics()`: Collects metrics using the `collect_metrics()` function from `metrics_api` module.
2. **Data Formatting**:
    - `format_metrics = m.format_metric_data(metrics_response)`: Formats the collected metrics data.
3. **Saving to File**:
    - Metrics data is saved to a file named "sample.txt".
4. **Insight Generation**:
    - `gemini_insights = m.get_insights_from_genai(format_metrics)`: Generates insights using Generative AI based on formatted metrics.

5. **Notification Sending**:
    - WhatsApp and email notifications are sent with the generated insights:
        - `m.send_whatsapp_notification(gemini_insights)`: Sends WhatsApp notification.
        - `m.send_email_notification("Metrics Insights", gemini_insights)`: Sends email notification.

This function encapsulates the entire process of collecting metrics, generating insights, and sending notifications in a concise manner.

app.py



```python
from flask import Flask, request
import requests
import automation
app = Flask(__name__)


@app.route('/')
def hello():
    return "<h2>Hello world - Integration Activity Flask Deployment Lab: try 10 <h2><hr/>"


@app.route('/mywebhook', methods=['POST', 'GET'])
def alert_api():
    data = request.json
    automation.automation()
    return "Notification Sent"


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

1. **Flask Routes**:
    - `'/'`: Displays a greeting message.
    - `'/mywebhook'`: Listens for webhook requests.
2. **'/mywebhook' Route**:
    - Receives POST requests.
    - Calls `automation.automation()` to perform metric collection and notification sending.
    - Responds with "Notification Sent".
3. **Main Execution**:
    - If the script is run directly, the Flask app runs on host '0.0.0.0' and port 5000.

This Flask app listens for webhook requests and triggers the automation process when a POST request is received on `/mywebhook`.