* **Introduction:** Performance testing is a type of non-functional testing that assesses how a system or application performs under different conditions. The primary purpose is to guarantee that the application meets performance requirements such as speed, responsiveness, and stability. Performance testing identifies bottlenecks, ensures that the system can manage expected and peak loads, and confirms that performance criteria are satisfied.
* **Performance Testing Plan:**

**Objective:** The goal of this testing strategy is to assess the performance of the PHP project https://github.com/thealoneprogrammer/Musical-World under various load situations using Python scripts and JMeter. The strategy describes the methodology to load, stress, reliability, capacity, volume, and scalability testing.

**Scope:** The testing will include both server and database performance, ensuring that the application can handle expected and peak traffic while remaining stable over time.

* **Performance Testing Model:**

**Testing Tools:**

1. **Script Performance Testing:** Python
2. **Tool Performance Testing:** JMeter

**Test Environment Setup:**

1. **Web Server:** We do have attached our vagrant files which explain each and every step to configured Web Server.

**Software installed:** Apache, PHP

1. **Database Server:** We do have attached our vagrant files which explain each and every step to configured DB Server.

**Software installed:** MySQL

* **Types of Testing:**

1. **Script Performance Testing (using Python):**

* **Load Testing:**
  + **Objective:** Load testing simulates a specified number of concurrent users using the application to determine how it operates under expected traffic levels. The goal is to establish the maximum load that the application can tolerate without causing substantial performance deterioration.
  + **Method:** We used Python scripts to simulate multiple users accessing the application simultaneously.
  + **Metrics:** Response time, throughput, error rate.
  + **Script:**

from locust import HttpUser, TaskSet, task, between

import os

class UserBehavior(TaskSet):

def on\_start(self):

self.login()

@task(1)

def register(self):

response = self.client.post("/project/validate.php", {

"username": "testuser",

"mobile": "1235567890",

"email": "testuser@gmail.com",

"password": "testpassword",

"repeat\_password": "testpassword"

})

if response.status\_code != 200:

print("Failed to register")

@task(2)

def login(self):

response = self.client.post("/project/validate.php", {"username": "testuser@gmail.com", "password": "testpassword"})

if response.status\_code != 200:

print("Failed to login")

@task(3)

def upload\_song(self):

with open("song.txt", "rb") as song:

response = self.client.post("/project/uploaded\_songs.php", files={"file": song})

if response.status\_code != 200:

print("Failed to upload song")

class LoadTestUser(HttpUser):

tasks = [UserBehavior]

wait\_time = between(1, 3)

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

user\_counts = [50, 100, 150, 200, 250, 300, 350, 400, 450, 500]

for count in user\_counts:

print(f"Running load test with {count} users...")

command = f"locust -f load\_test.py --host http://192.168.56.10 --csv=load\_test\_results\_{count} --headless -u {count} -r 1 -t 5m"

os.system(command)

stats\_filename = f"load\_test\_results\_{count}\_stats.csv"

failures\_filename = f"load\_test\_results\_{count}\_failures.csv"

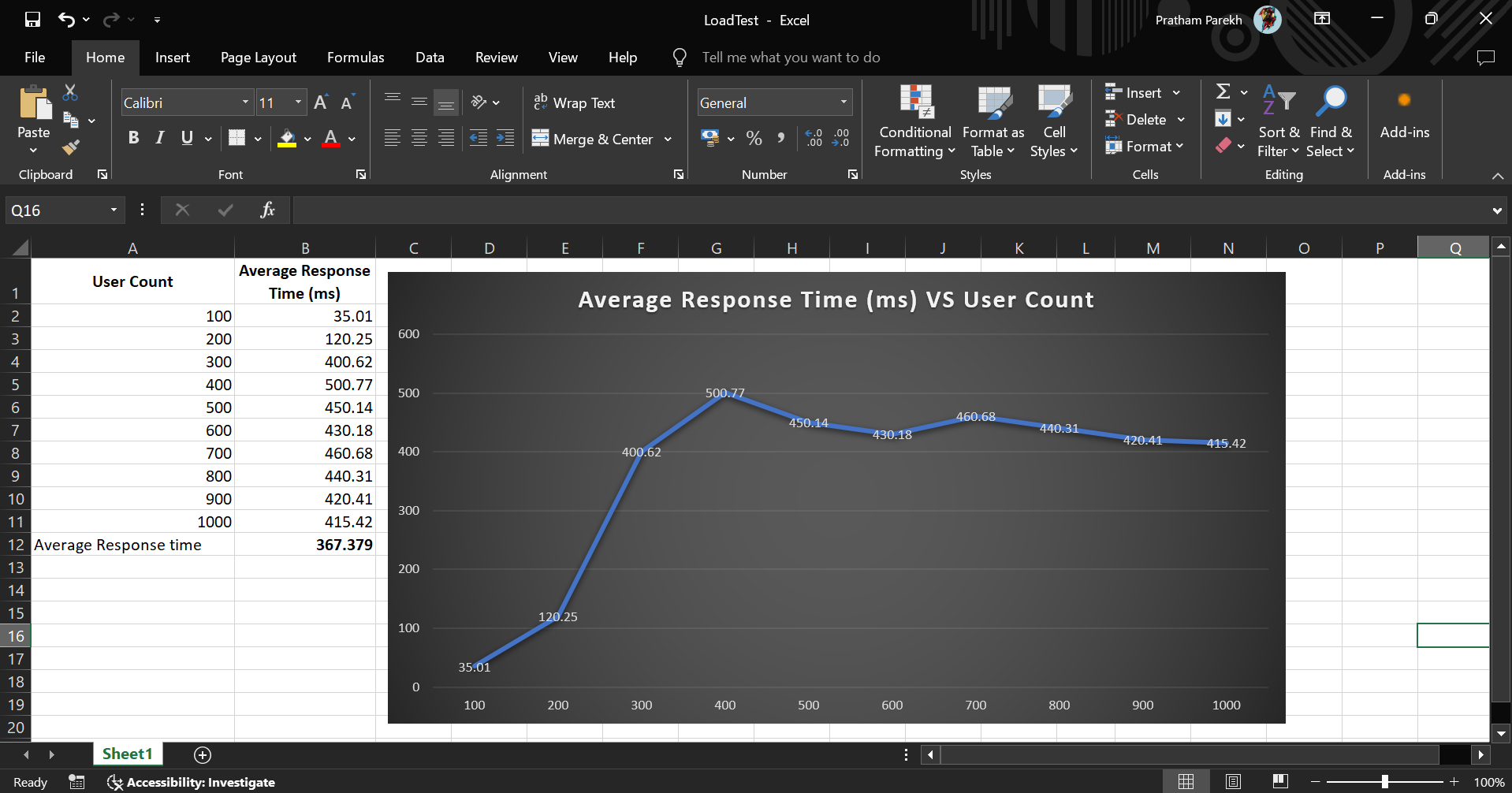
with open(stats\_filename, "r") as stats\_file:

print(stats\_file.read())

with open(failures\_filename, "r") as failures\_file:

print(failures\_file.read())

* **Load Testing Result:**



* **Analysis of Load Test Result:**

The graph indicates that there may be resource saturation as the number of users rises because the average response time first surges dramatically, reaching a maximum of 500.77 ms for 400 users. After this peak, though, the response time gradually declines and stabilises in the 400–460 ms range as the number of users increases from 500 to 1000, suggesting some degree of load balancing or system optimisation.

With an average response time of 367.379 ms, the system is able to withstand higher loads; however, additional optimisation may be required to lower response times, particularly under peak load situations.

* **Stress Testing:**
  + **Objective:** The application is pushed beyond its typical operating capability during stress testing, frequently to the brink of failure. This makes it easier to determine the application's breaking point and how it handles harsh circumstances, including sudden increases in traffic.
  + **Method:** We gradually increased the load using Python scripts until the system fails.
  + **Metrics:** Maximum load, failure points, system behavior under stress.
  + **Script:**

from locust import HttpUser, TaskSet, task, between

import os

class UserBehavior(TaskSet):

def on\_start(self):

self.login()

@task(1)

def register(self):

response = self.client.post("/project/validate.php", {

"username": "testuser",

"mobile": "1235567890",

"email": "testuser@gmail.com",

"password": "testpassword",

"repeat\_password": "testpassword"

})

if response.status\_code != 200:

print("Failed to register")

@task(2)

def login(self):

response = self.client.post("/project/validate.php", {"username": "testuser@gmail.com", "password": "testpassword"})

if response.status\_code != 200:

print("Failed to login")

@task(3)

def upload\_song(self):

with open("song.txt", "rb") as song:

response = self.client.post("/project/uploaded\_songs.php", files={"file": song})

if response.status\_code != 200:

print("Failed to upload song")

class StressTestUser(HttpUser):

tasks = [UserBehavior]

wait\_time = between(1, 3)

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

user\_counts = [2000]

for count in user\_counts:

print(f"Running stress test with {count} users...")

command = f"locust -f stress\_test.py --host http://192.168.56.10 --csv=stress\_test\_results\_{count} --headless -u {count} -r 20 -t 20m"

os.system(command)

stats\_filename = f"stress\_test\_results\_{count}\_stats.csv"

failures\_filename = f"stress\_test\_results\_{count}\_failures.csv"

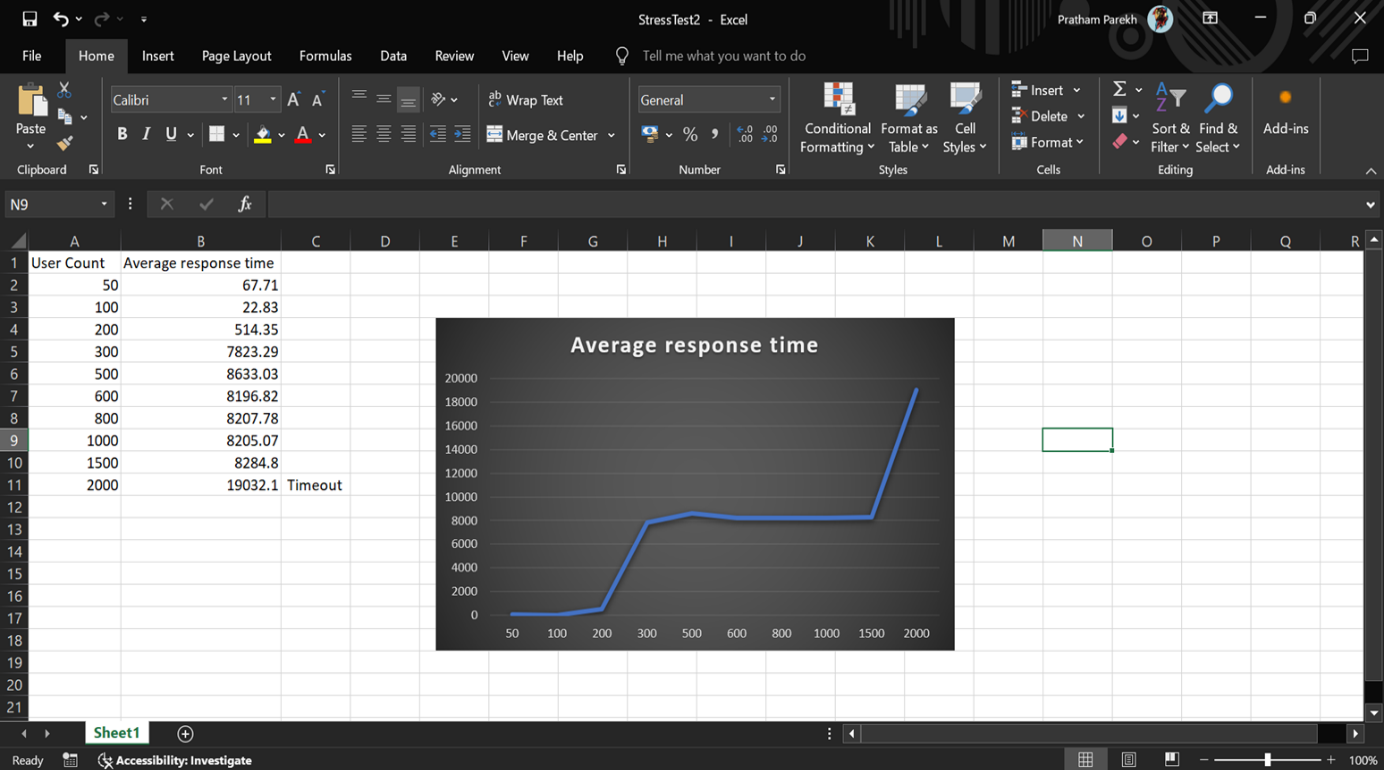
with open(stats\_filename, "r") as stats\_file:

print(stats\_file.read())

with open(failures\_filename, "r") as failures\_file:

print(failures\_file.read())

* **Stress Testing Result:**



* **Analysis of Stress Test Result:**

The graph shows that when the number of users rises, the average response time first stays low before rising significantly, particularly when 200 users are reached. At 200 users, the reaction time is 514.35 ms; at 300 users, it climbs to above 7800 ms, and it stays high, varying between 8200 and 8300 ms up to 1500 users. The response time explodes to 19032.1 ms at 2000 users, causing a timeout.

This suggests that the system experiences substantial performance degradation and eventually breaks down when the load reaches 2000 users. It also shows that the system suffers significantly with larger loads. To increase scalability and avoid such severe slowdowns and timeouts under high load, optimisations are required.

* **Reliability Testing:**
  + **Objective:** The purpose of reliability testing is to evaluate an application's capacity to operate for a prolonged amount of time under a certain load. This guarantees that the program won't crash or have significant problems over time, and that it will continue to operate steadily and perform consistently.
  + **Method:** We used Python scripts to simulate a steady load over several hours.
  + **Metrics:** Uptime, error rate, response time stability.
  + **Script:**

from locust import HttpUser, TaskSet, task, between

import os

import pandas as pd

class UserBehavior(TaskSet):

def on\_start(self):

self.register()

self.login()

@task(1)

def register(self):

response = self.client.post("/project/validate.php", {

"username": "testuser",

"mobile": "1235567890",

"email": "testuser@gmail.com",

"password": "testpassword",

"repeat\_password": "testpassword"

})

if response.status\_code != 200:

print("Failed to register")

@task(2)

def login(self):

response = self.client.post("/project/validate.php", {"username": "testuser@gmail.com", "password": "testpassword"})

if response.status\_code != 200:

print("Failed to login")

@task(3)

def upload\_song(self):

with open("song.txt", "rb") as song:

response = self.client.post("/project/uploaded\_songs.php", files={"file": song})

if response.status\_code != 200:

print("Failed to upload song")

class ReliabilityTestUser(HttpUser):

tasks = [UserBehavior]

wait\_time = between(1, 3)

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

user\_counts = [100, 200, 300, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000]

combined\_data = []

for count in user\_counts:

print(f"Running reliability test with {count} users...")

command = f"locust -f reliability\_test.py --host http://192.168.56.10 --csv=reliability\_test\_results\_{count} --headless -u {count} -r 1 -t 10m"

os.system(command)

stats\_filename = f"reliability\_test\_results\_{count}\_stats.csv"

failures\_filename = f"reliability\_test\_results\_{count}\_failures.csv"

df\_stats = pd.read\_csv(stats\_filename)

df\_failures = pd.read\_csv(failures\_filename)

df\_stats['User Count'] = count

combined\_data.append(df\_stats)

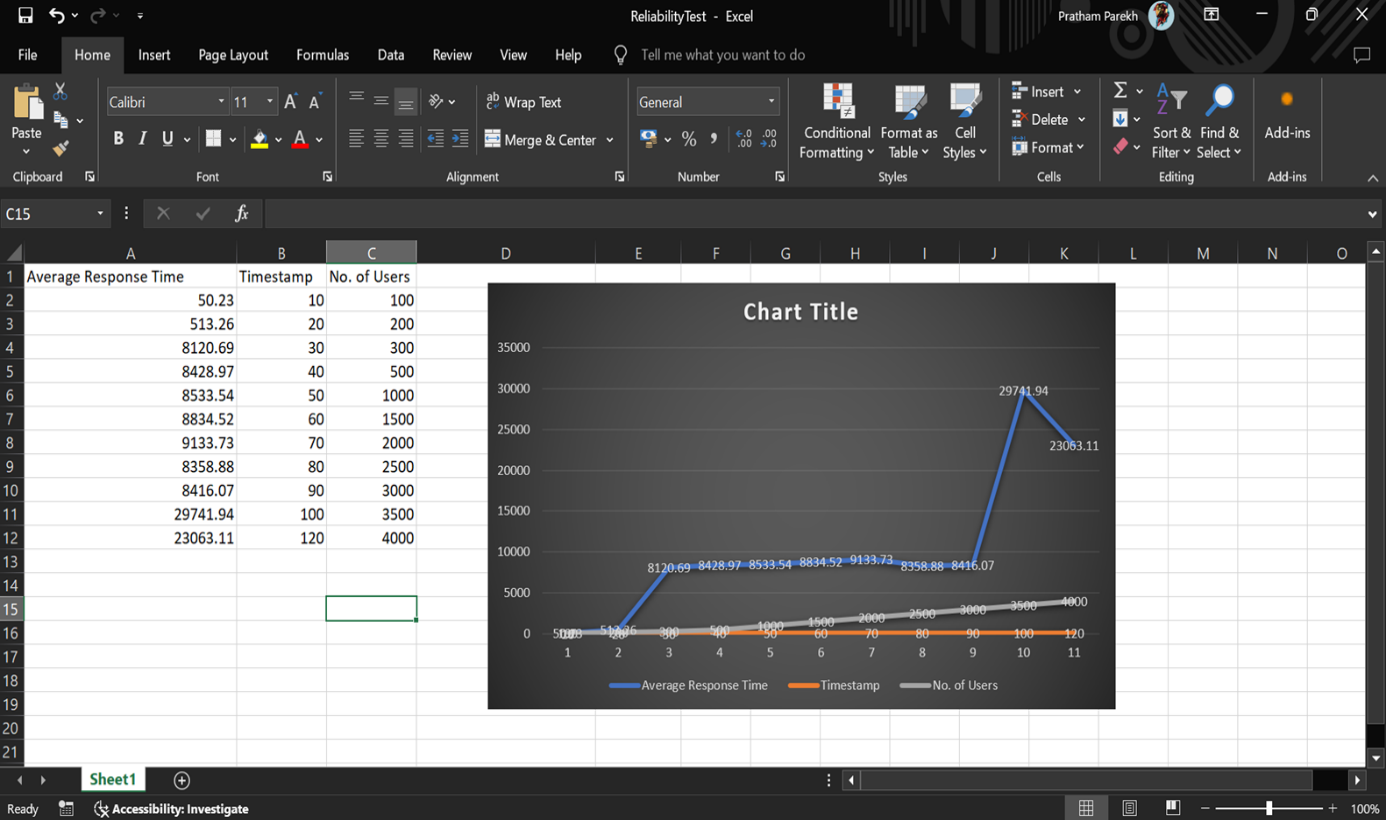
combined\_df = pd.concat(combined\_data)

combined\_df.to\_csv("combined\_reliability\_test\_results.csv", index=False)

with open(failures\_filename, "r") as failures\_file:

print(failures\_file.read())

* **Reliability Testing Result:**



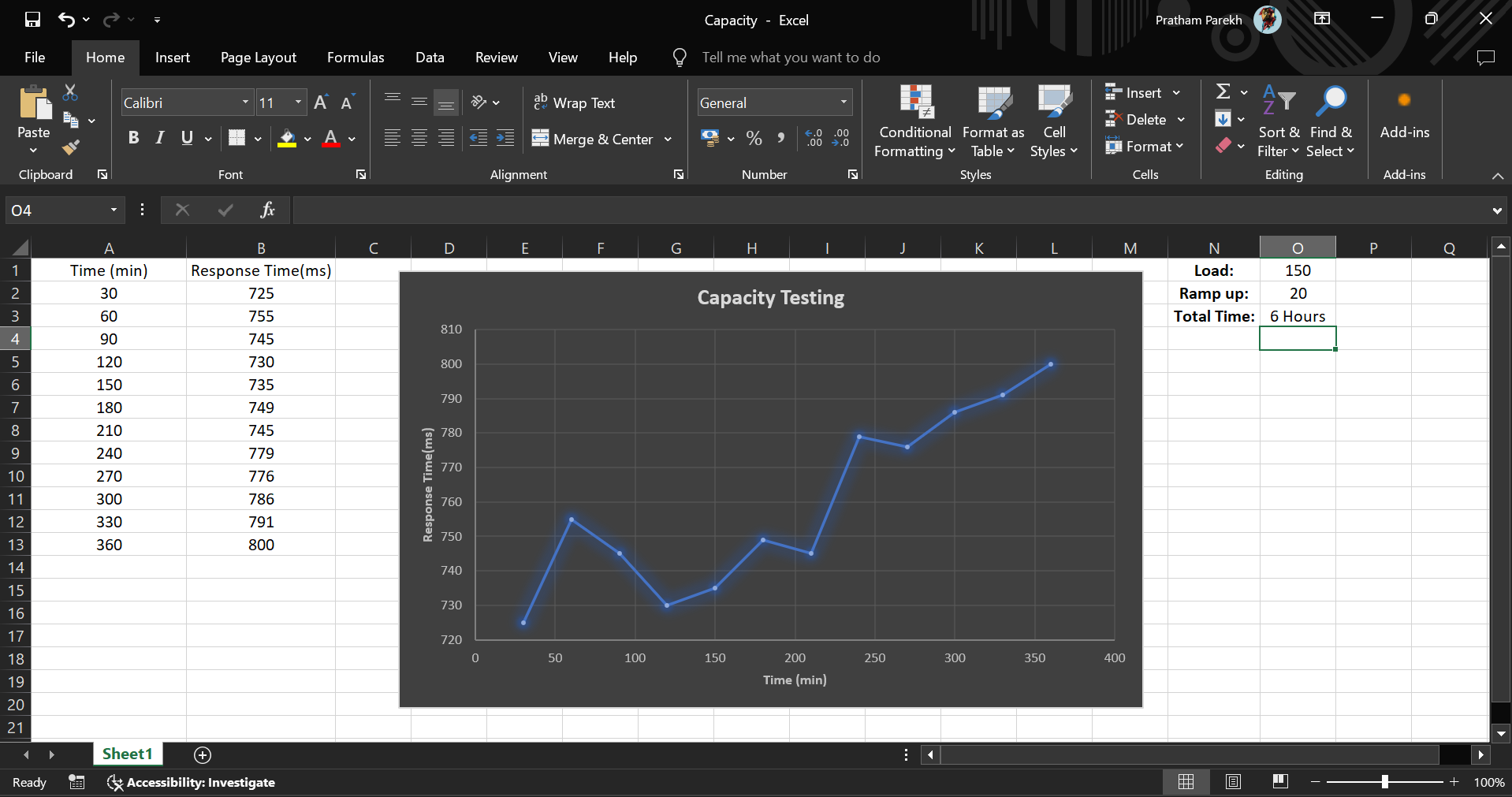
* **Analysis of Reliability Test Result:**

The reliability test data shows the typical response time (in milliseconds) of a framework over different timestamps, with a rising number of clients. At first, the normal reaction time is low, at 50.23 ms for 100 clients at the 10-minute imprint. As the quantity of clients builds, there is a huge spike accordingly time, quite at the 30-minute imprint with 300 clients, where the reaction time leaps to 8120.69 ms. This pattern proceeds, with the reaction time topping at 29741.94 ms for 3500 clients at the 100-minute imprint. In this way, there is a slight diminishing to 23063.11 ms for 4000 clients at 120 minutes.

The information shows that the framework's reaction time increments decisively with a larger number of clients, featuring potential execution issues under weighty burden conditions. This investigation highlights the significance of improving the framework to deal with a bigger client base successfully to keep up with satisfactory execution levels.

1. **Tool Performance Testing (using JMeter):**

* **Capacity Testing:**
  + **Objective:** Capacity testing decides the most extreme number of clients or exchanges the application can deal with while as yet meeting execution necessities. It recognizes the maximum furthest reaches of the application's capacity.
  + **Method:** We used JMeter to simulate increasing numbers of users or transactions.
  + **Metrics:** Response time, throughput, resource utilization.
* **Capacity Testing Result:**

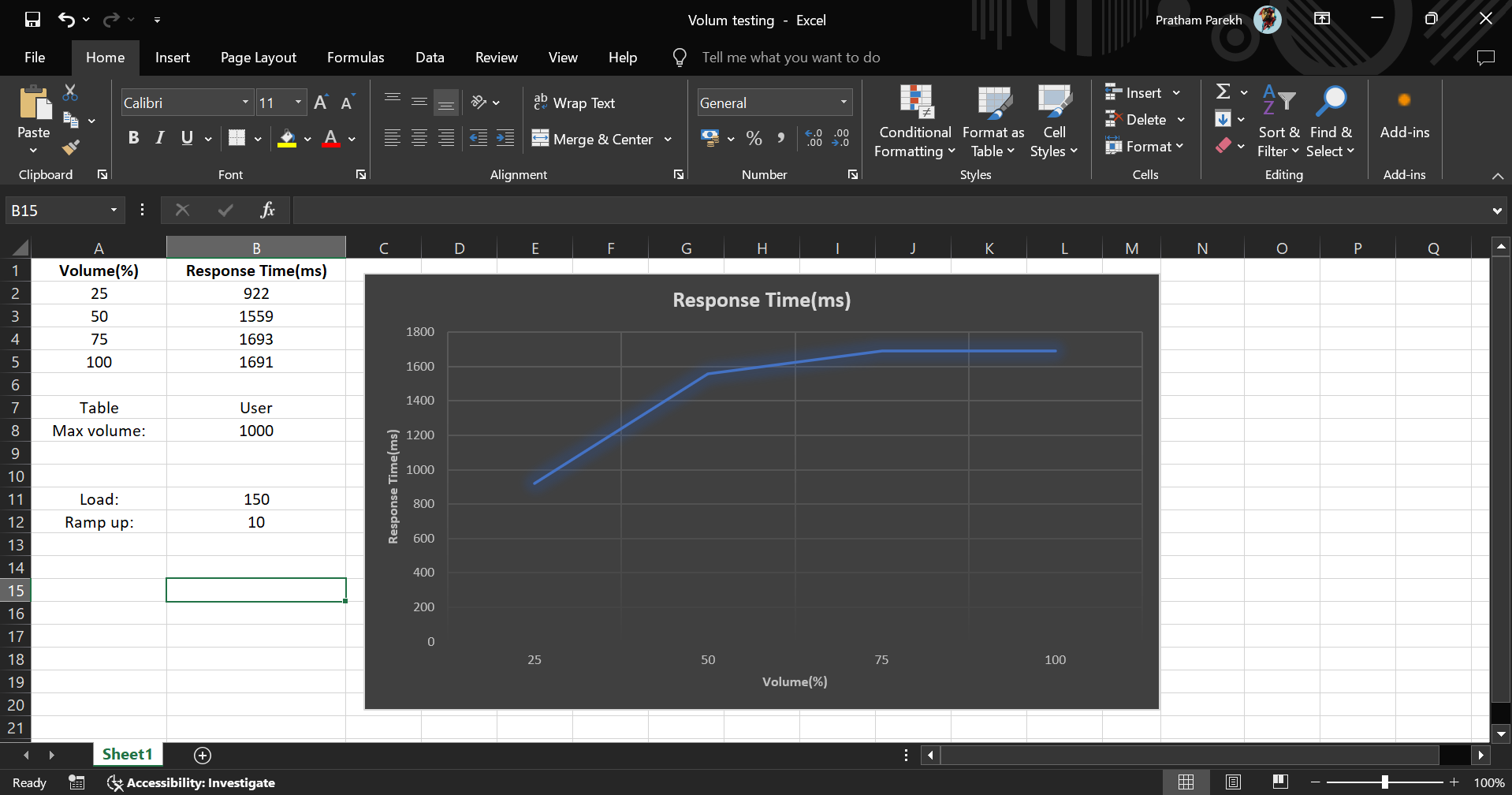


* **Analysis of Capacity Test Result:**

The capacity testing data provided shows the response time (in milliseconds) of a framework over a time of 6 hours, with estimations required like clockwork. The graph shows a fluctuating pattern accordingly times, beginning at 725 ms and cresting at 800 ms toward the finish of the testing time frame. At first, the reaction time increments, then diminishes somewhat, trailed by additional articulated vacillations, at last rising consistently after the 240-minute imprint.

The test was directed with a heap of 150 and an increase season of 20. The general pattern demonstrates that over the natural course of time, the framework's reaction time will in general increment, recommending that the framework's presentation might corrupt under supported load conditions over a drawn-out period. This investigation features the significance of checking and improving framework execution to guarantee solidness and responsiveness under changing burden conditions.

* **Volume Testing:**
  + **Objective:** Volume testing includes testing the application with a lot of information to guarantee it can deal with huge volumes without compromising execution. This is essential for applications expected to process or store huge datasets.
  + **Method:** We used JMeter to simulate operations involving large datasets.
  + **Metrics:** Data processing time, response time, error rate.
* **Volume Testing Result:**



* **Analysis of Volume Test Result:**

The volume testing information uncovers that the framework encounters a critical expansion accordingly time as the volume of clients rises. At 25% volume, the reaction time is 922 ms, which leaps to 1559 ms at half volume. This sharp increment shows that the framework's exhibition debases altogether under higher burden. As the volume increments to 75%, the reaction time further ascents to 1693 ms however at that point marginally diminishes to 1691 ms at 100 percent volume. This adjustment proposes that the framework utilizes systems like burden adjusting or asset improvement to deal with the heap all the more successfully at higher volumes.

Notwithstanding, the high reaction times at expanded volumes show that the framework's ability is stressed. To further develop execution, it is prescribed to research and upgrade the parts liable for the underlying reaction time spike, improve load adjusting systems, and consider versatility enhancements to more readily deal with higher client volumes without huge execution debasement.

* **Propose Scalability Testing:**
  + **Objective**: the application's capacity to increase (handle expanded load by adding assets) or scale out (handle expanded load by adding more examples). The objective is to guarantee the application can proficiently scale to fulfill developing needs. By directing versatility testing, associations can guarantee their applications are hearty, solid, and fit for supporting development, eventually prompting better execution, consumer loyalty, and business achievement.
  + **Method:** We will use JMeter to test both vertical (adding more power) and horizontal (adding more instances) scaling.
* **Recommendations:**
* Load Testing: Investigate the performance abnormality at 400 clients to distinguish explicit causes, for example, asset dispute, wasteful solicitation dealing with, or explicit application ways of behaving. Upgrade the framework to deal with such spikes all the more nimbly, guaranteeing reliable execution across various burden levels.
* Stress Testing: Reinforce the system's capacity deal with client counts up to 5000 by working on hidden foundation and streamlining application execution. Lead pressure tests past 5000 clients to distinguish the greatest limit and guarantee the framework can deal with outrageous burdens.
* Reliability Testing: Focus on improving scalability by optimizing resource allocation, improving load balancing, further developing burden adjusting, and improving server ability to deal with higher client counts without critical execution corruption. Lead more point-by-point profiling and observing to recognize and address bottlenecks that cause expanded reaction times and disappointments.
* Volume Testing: Mitigate Initial and Final Spikes: Investigate the causes of the initial and final response time spikes. This might involve examining system logs, resource usage patterns, and user This could include inspecting framework logs, asset utilization examples, and client conduct during these periods. Load Adjusting and Scaling: Carry out load adjusting and auto-scaling answers for handle unexpected spikes sought after more really. Guarantee that assets are powerfully distributed in light of continuous burden.
* Capacity Testing: Optimize Initial Load Handling: Improve the system's ability to handle the initial load surge by optimizing server startup processes, preloading essential resources, and ensuring that critical services are prioritized. Continuous Monitoring: Implement continuous performance monitoring to detect and respond to high load conditions in real-time, ensuring that the system maintains optimal performance throughout.
* **Conclusion:**
* The system exhibits stable performance under moderate loads yet shows huge debasement as the heap increments, especially featured in unwavering quality and stress testing situations.
* Addressing identified bottlenecks and advancing execution can upgrade the framework's capacity to deal with higher burdens, working on both unwavering quality and client experience.
* Continuous monitoring and regular testing are essential to guarantee the framework remains performant and dependable under differing load conditions, taking into consideration proactive distinguishing proof and goal of possible issues.