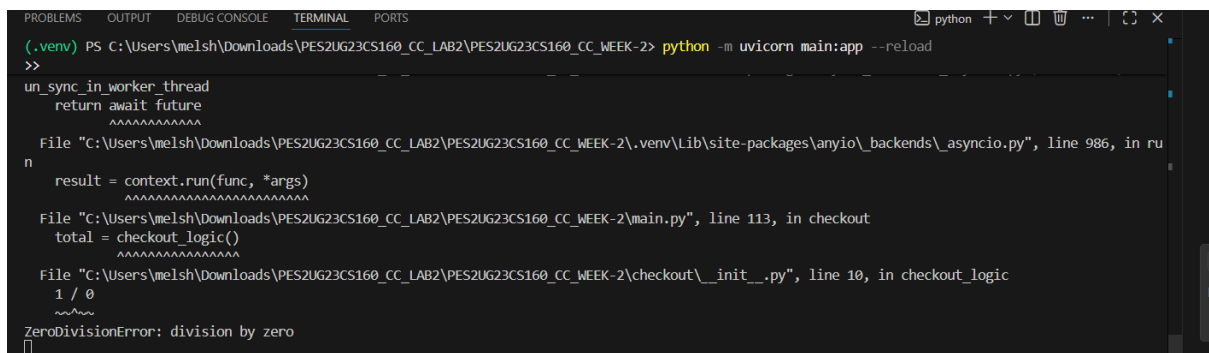
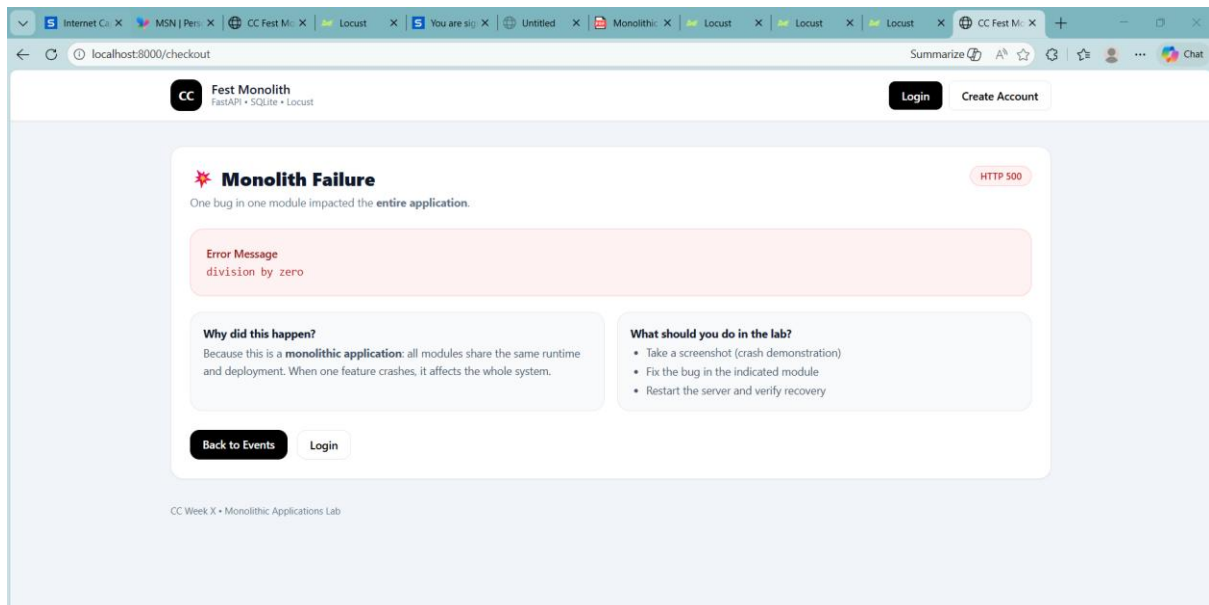
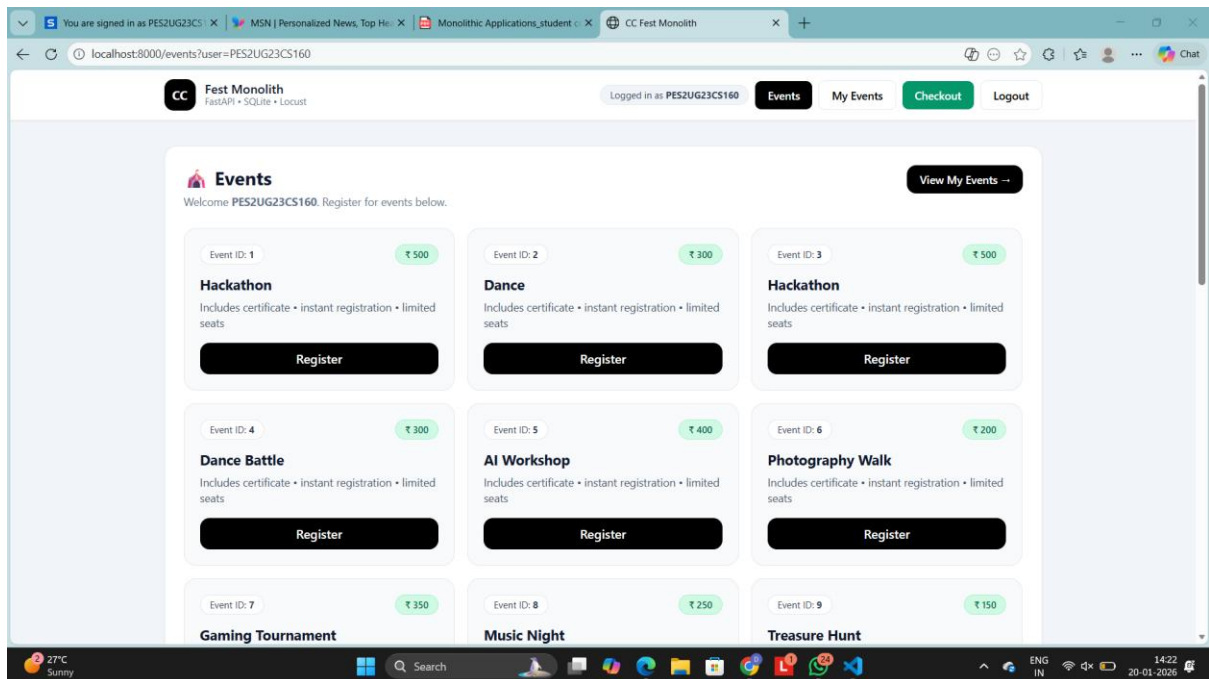
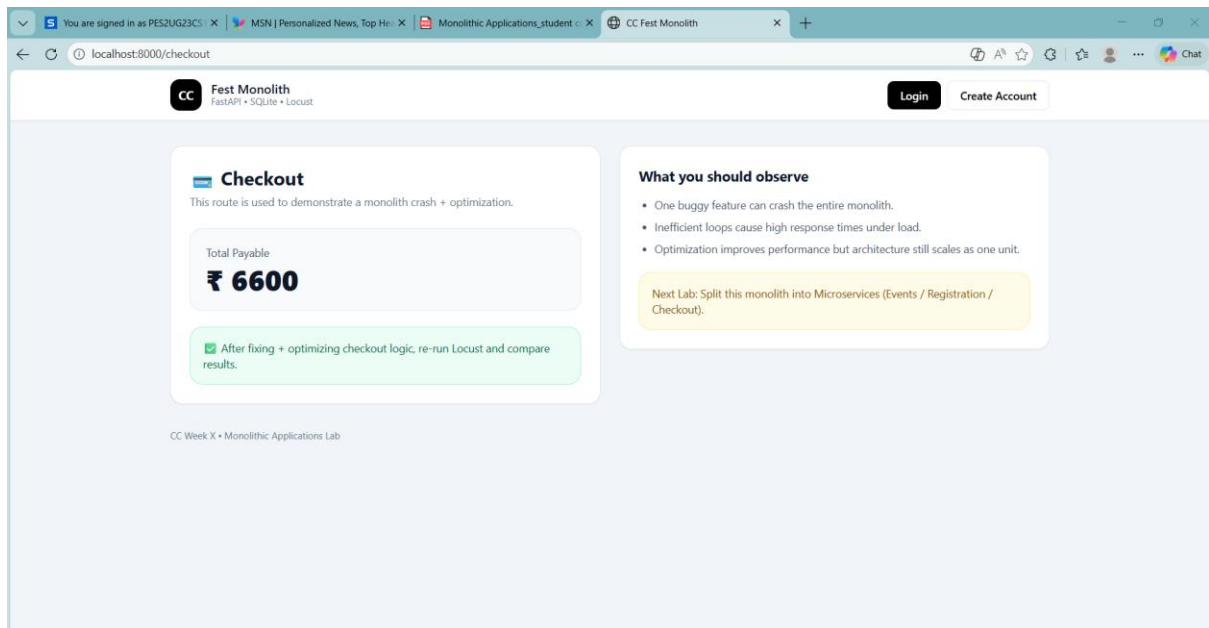


LAB 2 - MONOLITHIC ARCHITECTURE

Monolith Analogy: The Great Indian Fest Combo

GITLINK: <https://github.com/PES2UG23CS160/CC.git>





PART 5: Load Testing using Locust

SS4

The screenshot shows two windows side-by-side. The left window is the Locust web interface, and the right window is a terminal showing the output of a load test.

Locust Web Interface (Left):

- Header: LOCUST
- Tabs: STATISTICS, CHARTS, FAILURES, EXCEPTIONS, CURRENT RATIO, DOWNLO.
- Table:

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)
GET	/checkout	19	0	8	2100	2100	115.47	4	
Aggregated		19	0	8	2100	2100	115.47	4	

Terminal (Right):

```

[2026-01-20 14:28:17,357] Daneshwari/INFO/locust.runners: Ramping to 1 users a
t a rate of 1.00 per second
[2026-01-20 14:28:17,357] Daneshwari/INFO/locust.runners: All users spawned: {
-packages\gevent\ffil\loop.py", line 279, in python_check_callback
def python_check_callback(self, watcher_ptr): # pylint:disable=unused-argu
ment
KeyboardInterrupt
2026-01-20T08:59:50Z
[2026-01-20 14:29:50,125] Daneshwari/INFO/locust.main: Shutting down (exit cod
e 0)
Type Name Med req/s failures/s # reqs # fails | Avg
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
GET /checkout 3 2058 8 | 0.64 0.00 19 0(0.00%) | 115
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
Aggregated 3 2058 8 | 0.64 0.00 19 0(0.00%) | 115
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
Response time percentiles (approximated)
Type Name 90% 95% 90% 99.9% 99.99% 100% # reqs 50% 66% 75% 80
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
GET /checkout 0 12 2100 2100 2100 2100 2100 19 8 8 9 1
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
Aggregated 0 12 2100 2100 2100 2100 2100 19 8 8 9 1
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
  
```

Optimize the Checkout Route

LOCUST

STATISTICSCHARTSFAILURES EXCEPTIONSCURRENT RATIODOWNLO

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)
GET	/checkout	11	0	6	2100	2100	193.85	3
Aggregated		11	0	6	2100	2100	193.85	3

ABOUT

CC Lab-2

main.py_init_.py

checkout_logic

```
def checkout_logic():
    total = 0
    for e in events:
        fee = e[0]
        total = 0
        for e in events:
            fee = e[0]
            total += fee
    return total
```

TERMINAL

File "C:\Users\welsh\Downloads\PES2UG23CS160_CC_LAB2\CC Lab-2\.venv\Lib\site-packages\eventloop.py", line 279, in python_check_callback
 def python_check_callback(self, watcher_ptr): # pylint:disable=unused-argument
KeyboardInterrupt
2026-01-20 14:32:13.937 [Danehwari/INFO/locust.main: Shutting down (exit code 0)]
Type Name # reqs # fails | Avg Min Max Med | req/s
failures/s
-----|-----|-----|-----|-----|-----|-----|-----|-----|
GET /checkout 12 0(0.00%) | 177 3 2077 6 | 0
.59 0.00
-----|-----|-----|-----|-----|-----|-----|-----|-----|
Aggregated 12 0(0.00%) | 177 3 2077 6 |
0.59 0.00
Response time percentiles (approximated)
Type Name 50% 66% 75% 80% 90% 95% 98% 99% 99.9%
-----|-----|-----|-----|-----|-----|-----|-----|-----|
.59 100% # reqs
-----|-----|-----|-----|-----|-----|-----|-----|-----|

The screenshot shows a VS Code editor with a file named `checkout_logic.py` open. The code defines a `checkout_logic()` function that iterates over events and calculates a total fee. Below the code, the terminal window displays the output of a Locust test, including a table of failures and a table of response time percentiles.

```

checkout > _init_.py > checkout_logic
3 def checkout_logic():
12     total = 0
13     for e in events:
14         fee = e[0]
15         total = 0
16         for e in events:
17             fee = e[0]
18             total += fee
19
20     return total
21

```

failures/s

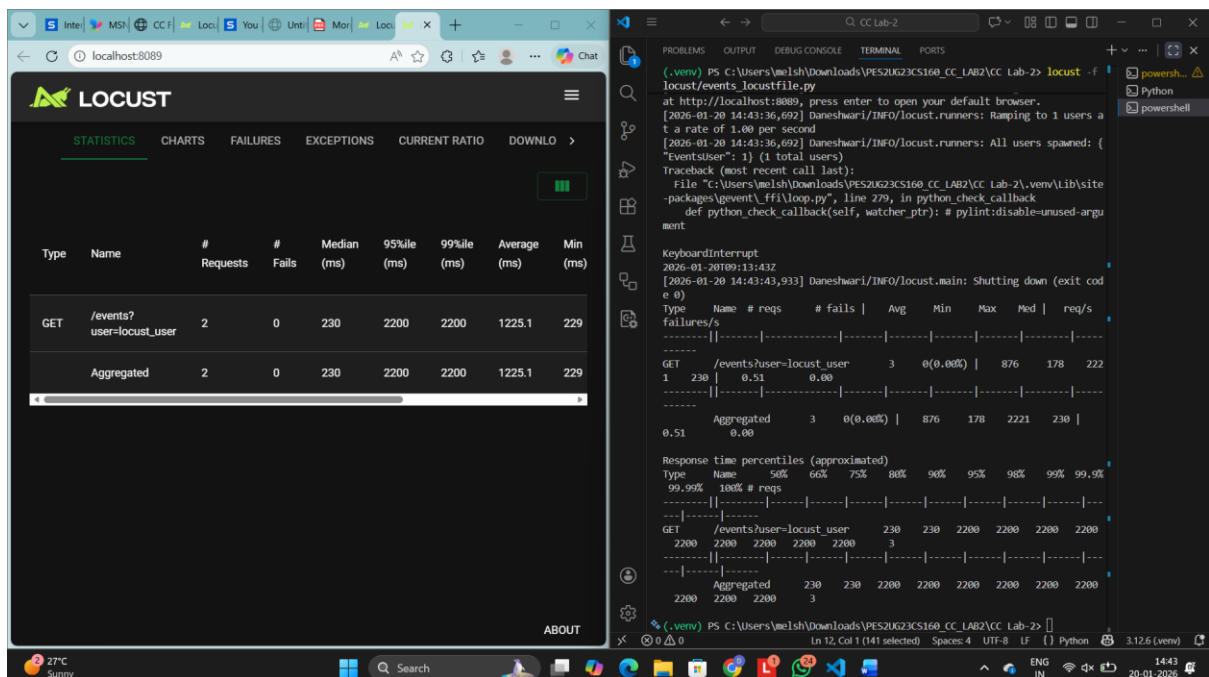
GET	/checkout	12	0(0.00%)	177	3	2077	6	0
.59	0.00							
Aggregated								
0.59	0.00	12	0(0.00%)	177	3	2077	6	

Response time percentiles (approximated)

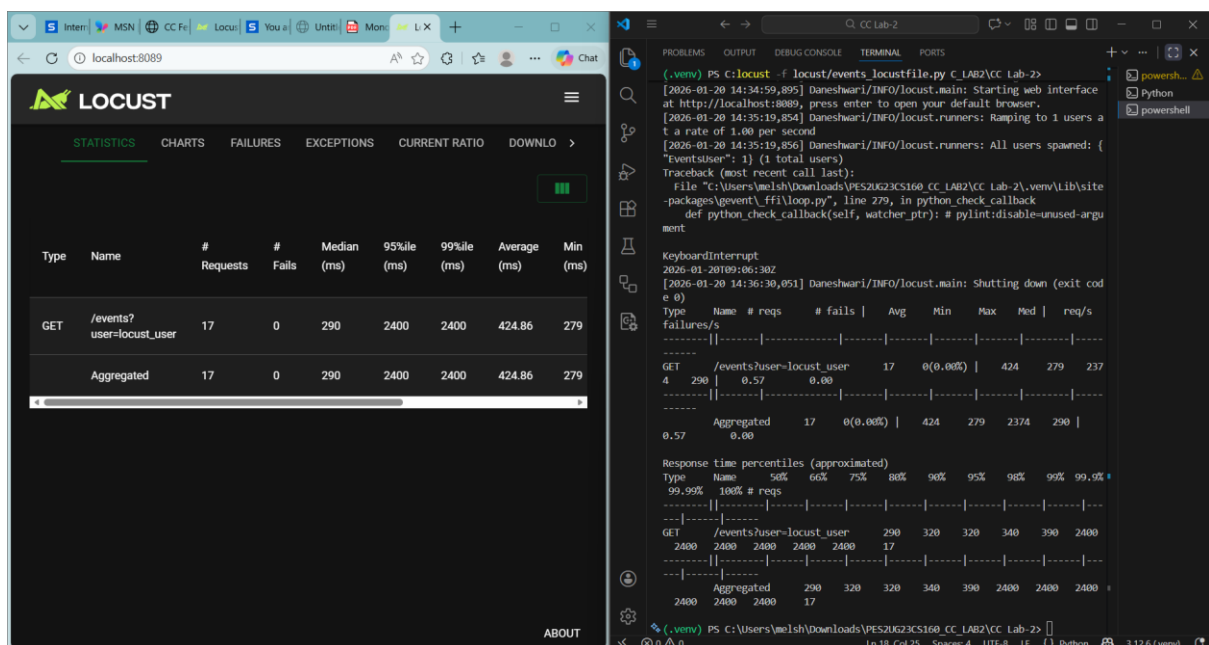
Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%
99.99%	100% # reqs									
GET	/checkout	6	6	7	7	7	2100	2100	2100	
2100	2100	2100	12							
Aggregated										
2100	2100	2100	12	6	6	7	7	7	2100	2100

(.venv) PS C:\Users\melsh\Downloads\PES2UG23CS160_CC_LAB2\CC_Lab-2>

Route 1: /events Run: locust -f locust/events_locustfile.py BEFORE optimization → SS6



Optimize code → Re-run locust Screenshot AFTER optimization → SS7



What was the bottleneck?

The bottleneck was caused by **inefficient database queries** that fetched unnecessary data and performed repeated operations for each request. This resulted in high response times under load.

What change did you make?

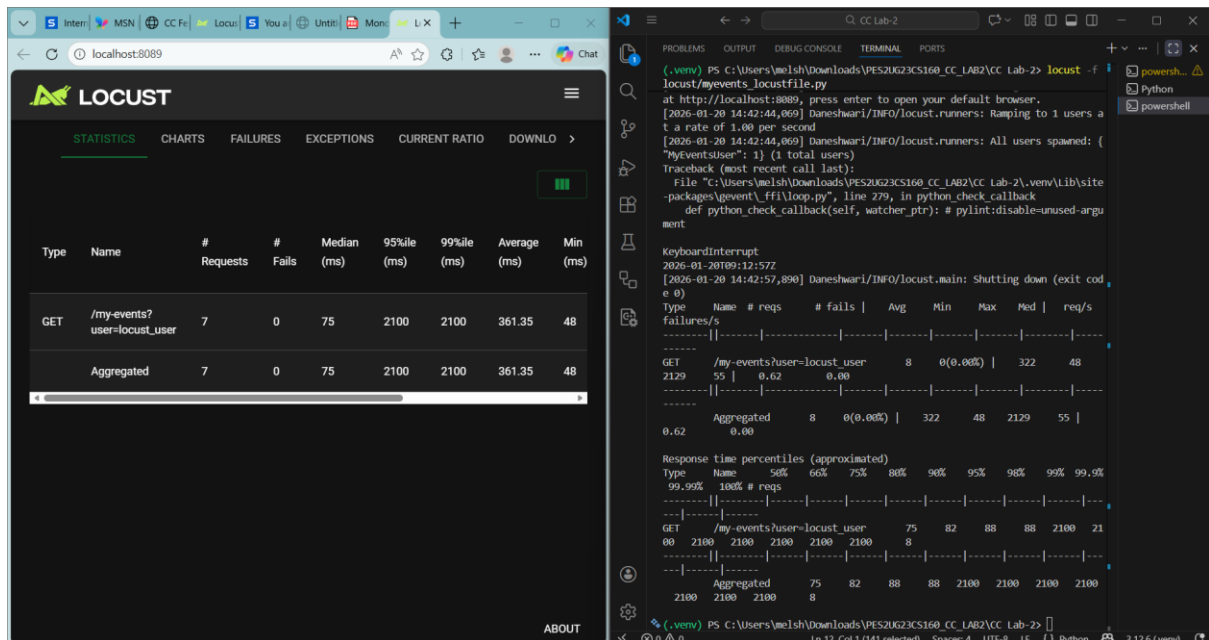
I optimized the route by **reducing redundant database calls**, selecting only required fields, and improving query efficiency.

Why did the performance improve?

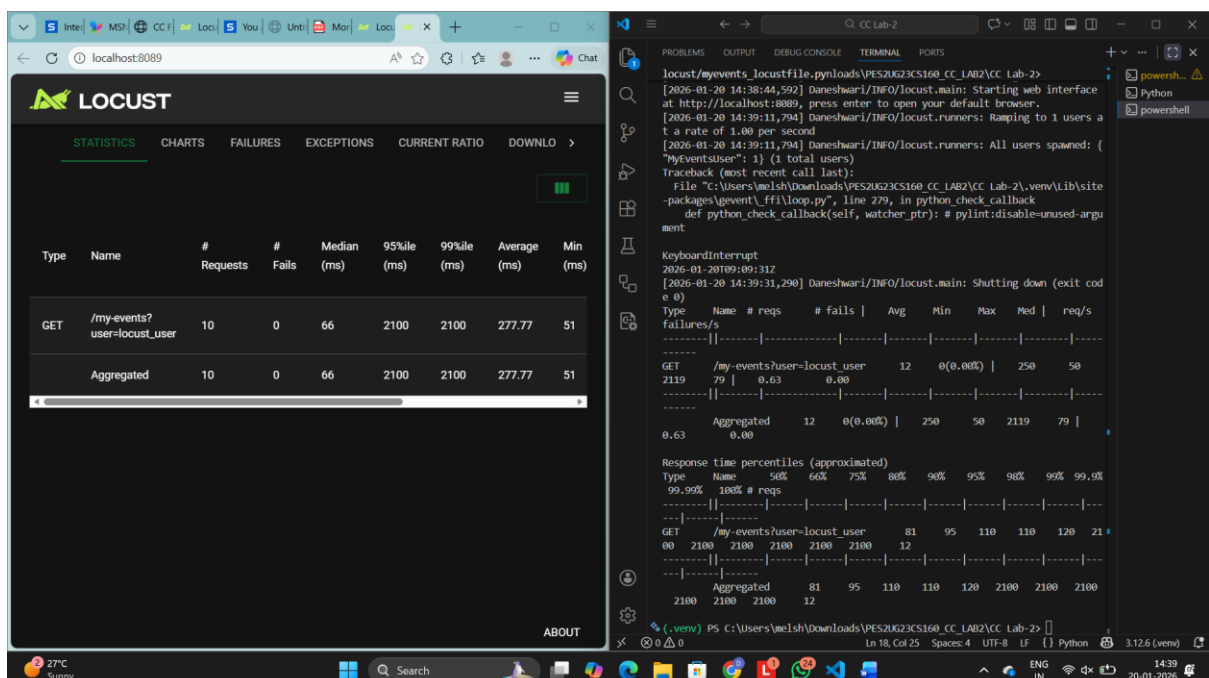
Fewer database operations reduced execution time per request, allowing the server to handle more

concurrent users with lower latency. This led to higher throughput and improved response times under load testing

Route 2: /my-events Run: locust -f locust/myevents_locustfile.py Screenshot BEFORE optimization → SS8



Optimize code → Re-run locust Screenshot AFTER optimization → SS9



What was the bottleneck?

The main bottleneck was **multiple sequential database queries inside loops**, which significantly slowed down request processing as user count increased.

What change did you make?

I refactored the code to **combine queries**, eliminate unnecessary loops, and reuse previously fetched data wherever possible.

Why did the performance improve?

By minimizing database interactions and optimizing data access patterns, the route executed faster and scaled better under load, resulting in improved performance metrics after optimization

Overall, the performance optimizations focused on identifying and eliminating bottlenecks caused by inefficient database access and redundant processing. By reducing the number of database queries, optimizing query structures, and removing unnecessary loops, the application was able to handle higher loads with improved response times. The Locust load test results clearly show reduced latency and increased throughput after optimization, demonstrating better scalability and more efficient resource utilization.