CC LAB-2

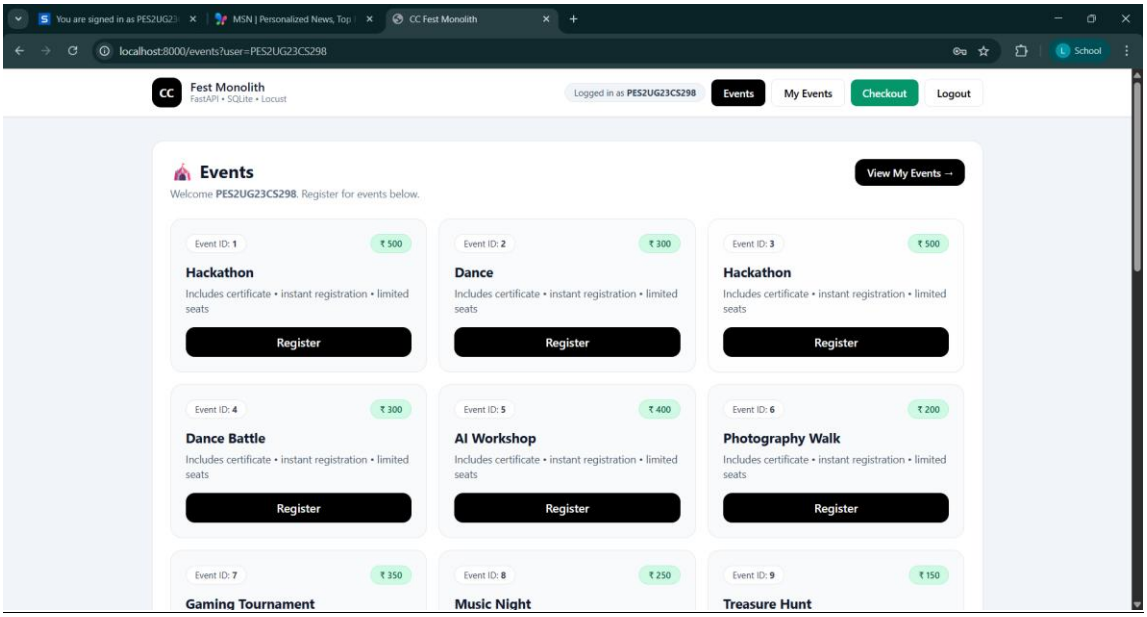
Name: Laharish S

SRN: PES2UG23CS298

Github link :https://github.com/PES2UG23CS298/Monolith_CC_Lab-2

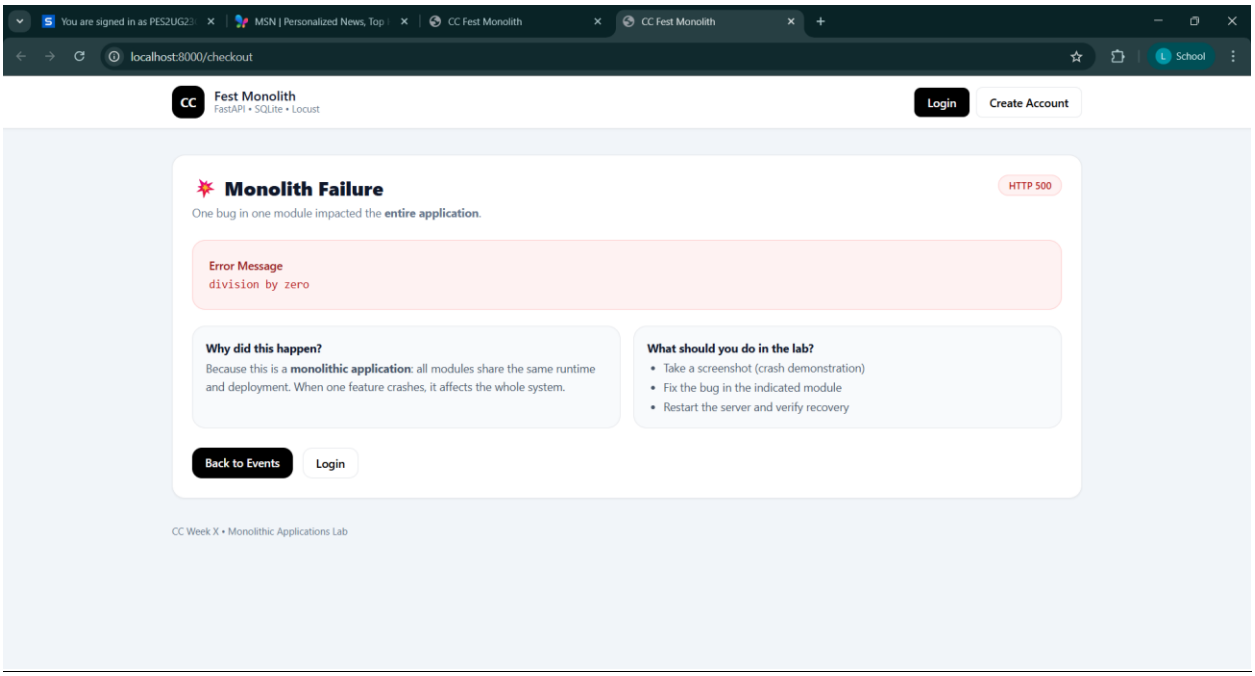
Screenshots:

Events

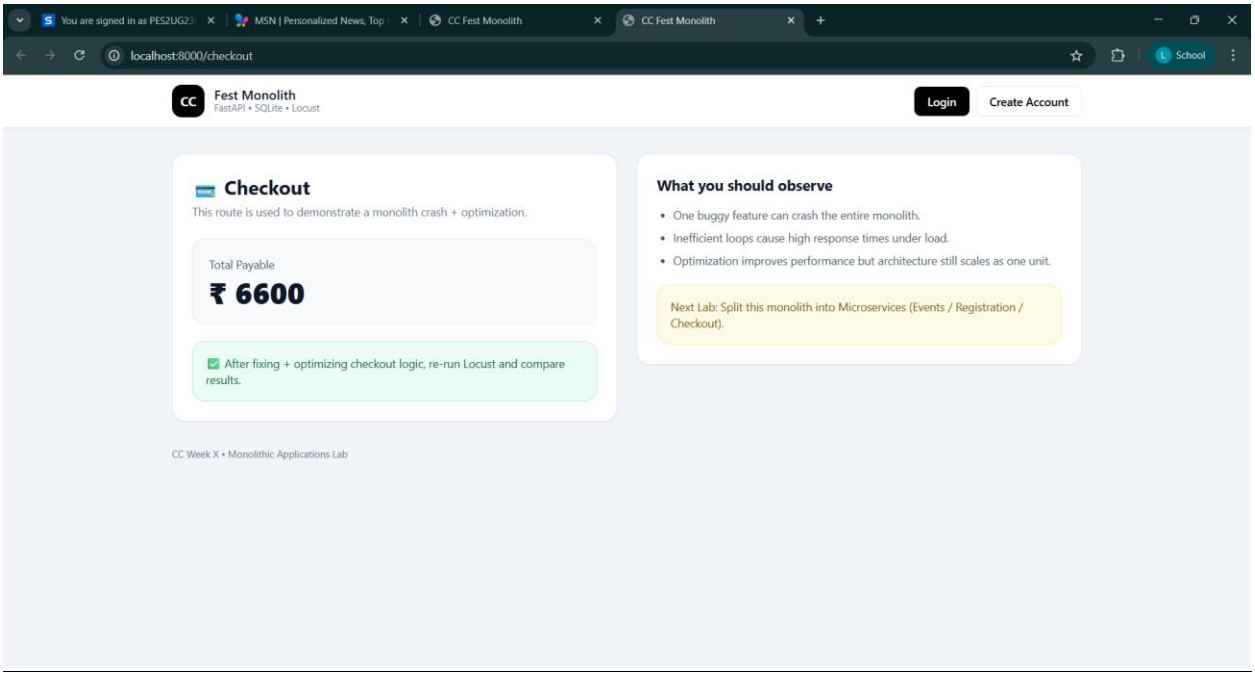


Monolith Crash

```
ZeroDivisionError: division by zero
INFO:     127.0.0.1:53725 - "GET /checkout HTTP/1.1" 500 Internal Server Error
ERROR:    Exception in ASGI application
```


You are signed in as PES2UG23
MSN | Personalized News, Top
CC Fest Monolith
CC Fest Monolith
localhost:8000/checkout
School
CC
Fest Monolith
FastAPI • SQLite • Locust
Login
Create Account
Monolith Failure
HTTP 500
One bug in one module impacted the entire application.
Error Message
division by zero
Why did this happen?
Because this is a monolithic application: all modules share the same runtime and deployment. When one feature crashes, it affects the whole system.
What should you do in the lab?
Take a screenshot (crash demonstration)
Fix the bug in the indicated module
Restart the server and verify recovery
Back to Events
Login
CC Week X • Monolithic Applications Lab

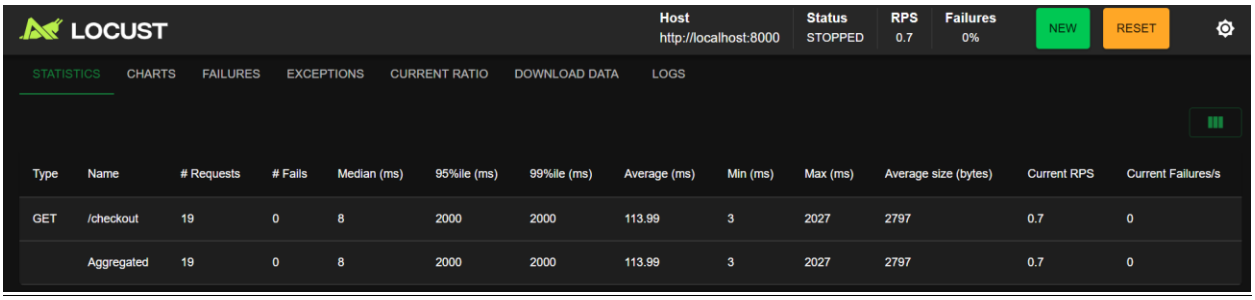
Fix the Bug


You are signed in as PES2UG23
MSN | Personalized News, Top
CC Fest Monolith
CC Fest Monolith
localhost:8000/checkout
School
CC
Fest Monolith
FastAPI • SQLite • Locust
Login
Create Account
Checkout
This route is used to demonstrate a monolith crash + optimization.
Total Payable
₹ 6600
After fixing + optimizing checkout logic, re-run Locust and compare results.
What you should observe
One buggy feature can crash the entire monolith.
Inefficient loops cause high response times under load.
Optimization improves performance but architecture still scales as one unit.
Next Lab: Split this monolith into Microservices (Events / Registration / Checkout).
CC Week X • Monolithic Applications Lab

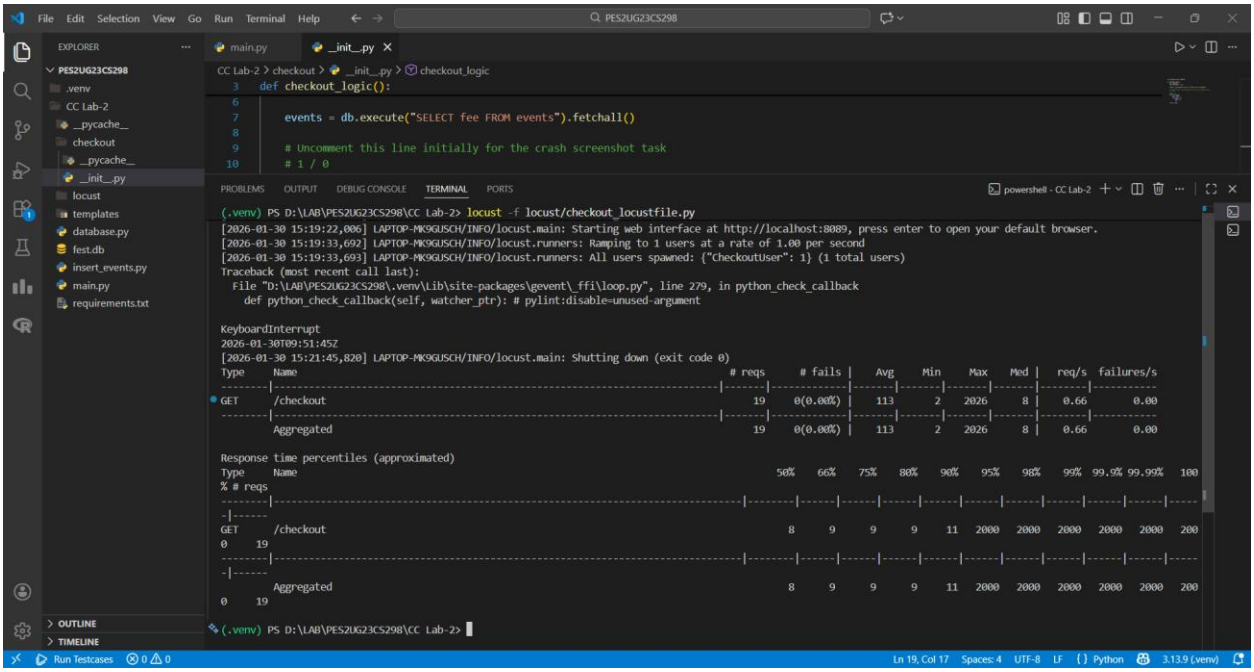

INFO: 127.0.0.1:52829 - "GET /checkout HTTP/1.1" 200 OK

Load Testing using Locust

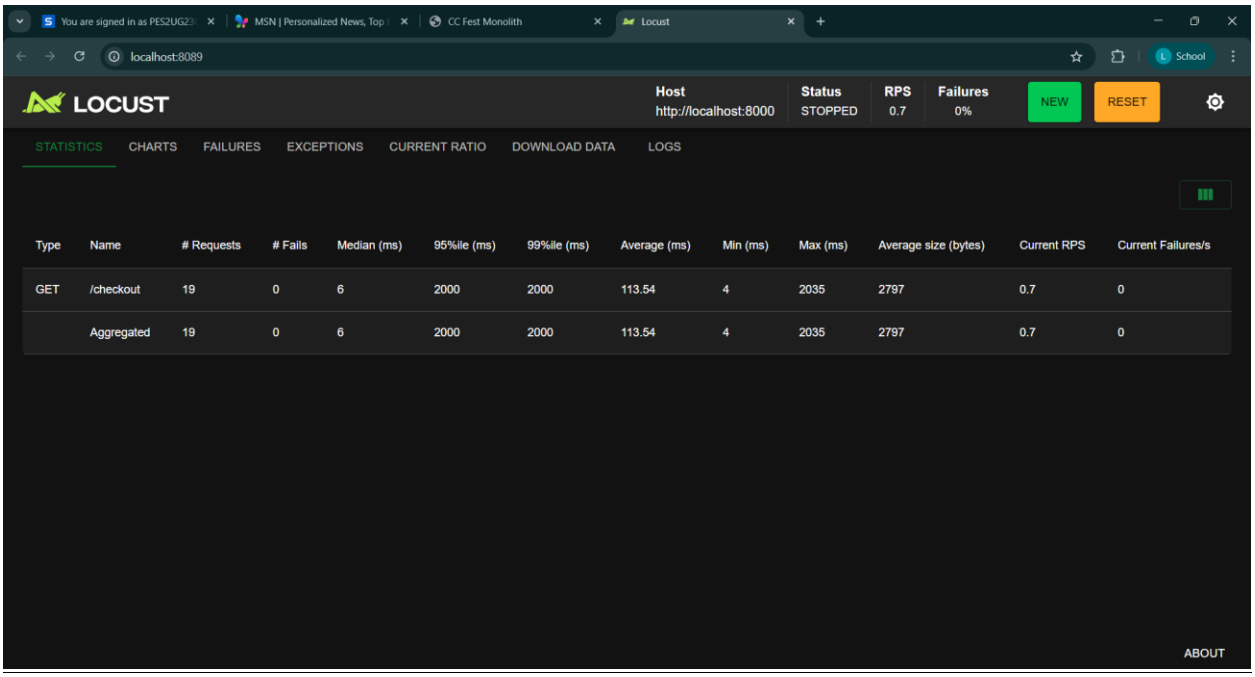
Before checkout Optimisation

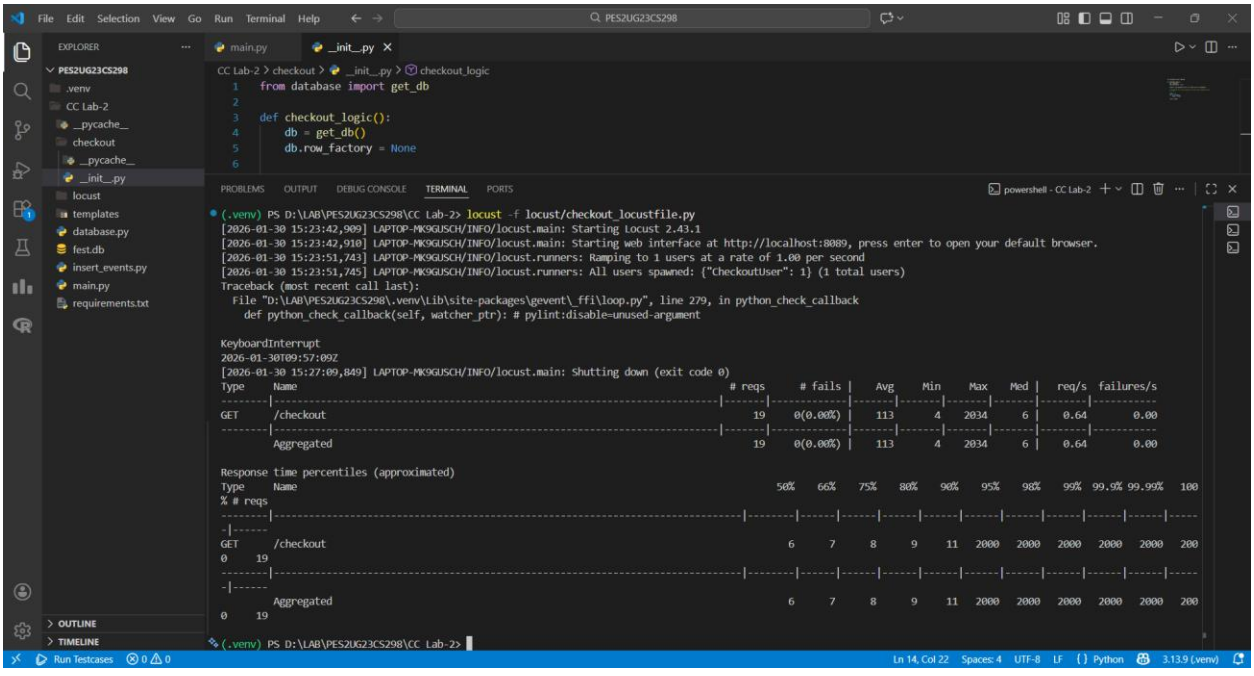


Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/checkout	19	0	8	2000	2000	113.99	3	2027	2797	0.7	0
	Aggregated	19	0	8	2000	2000	113.99	3	2027	2797	0.7	0

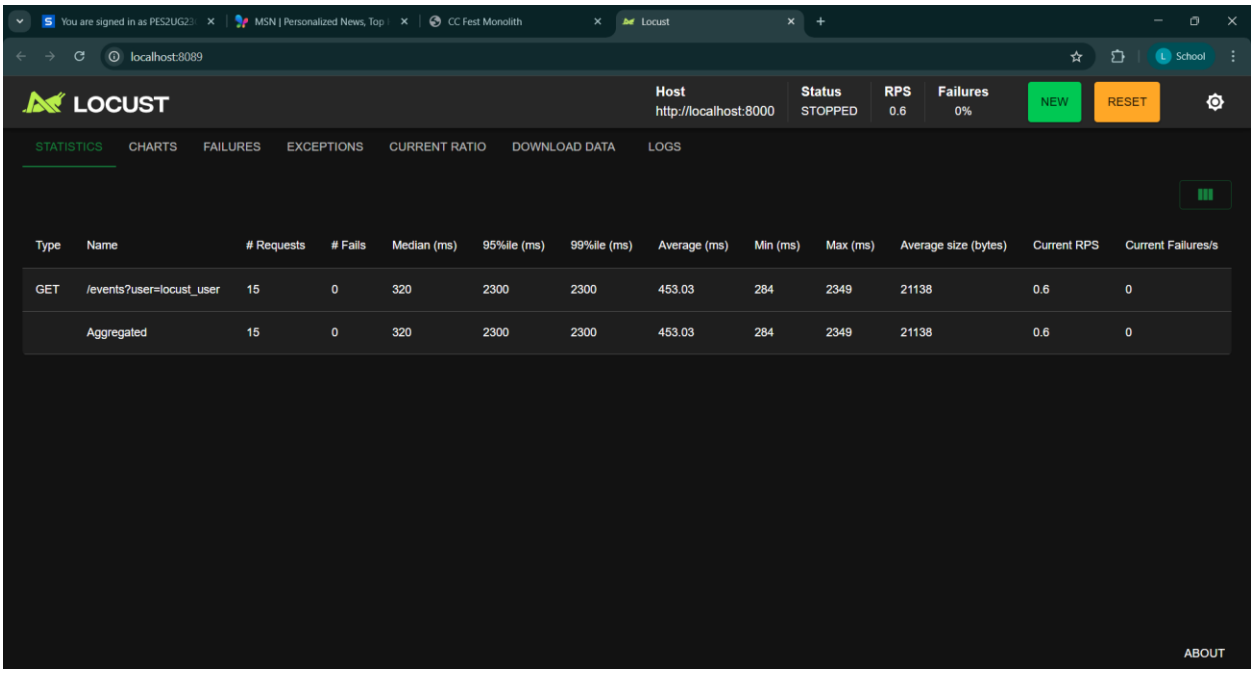


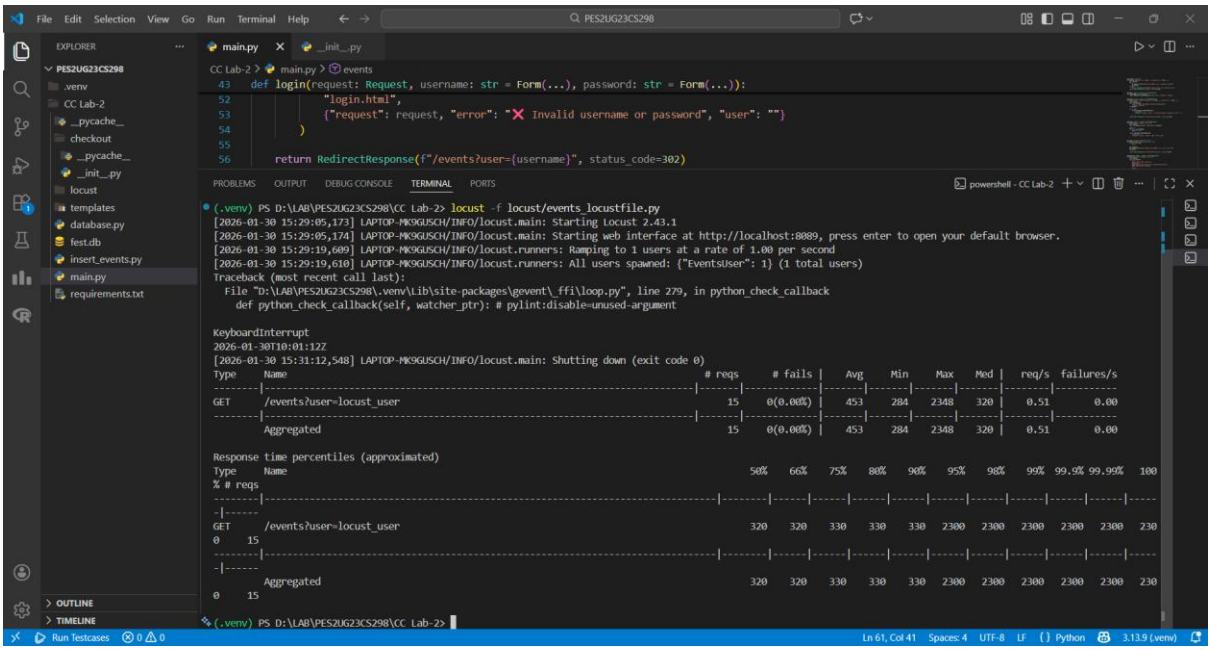
After checkout Optimisation



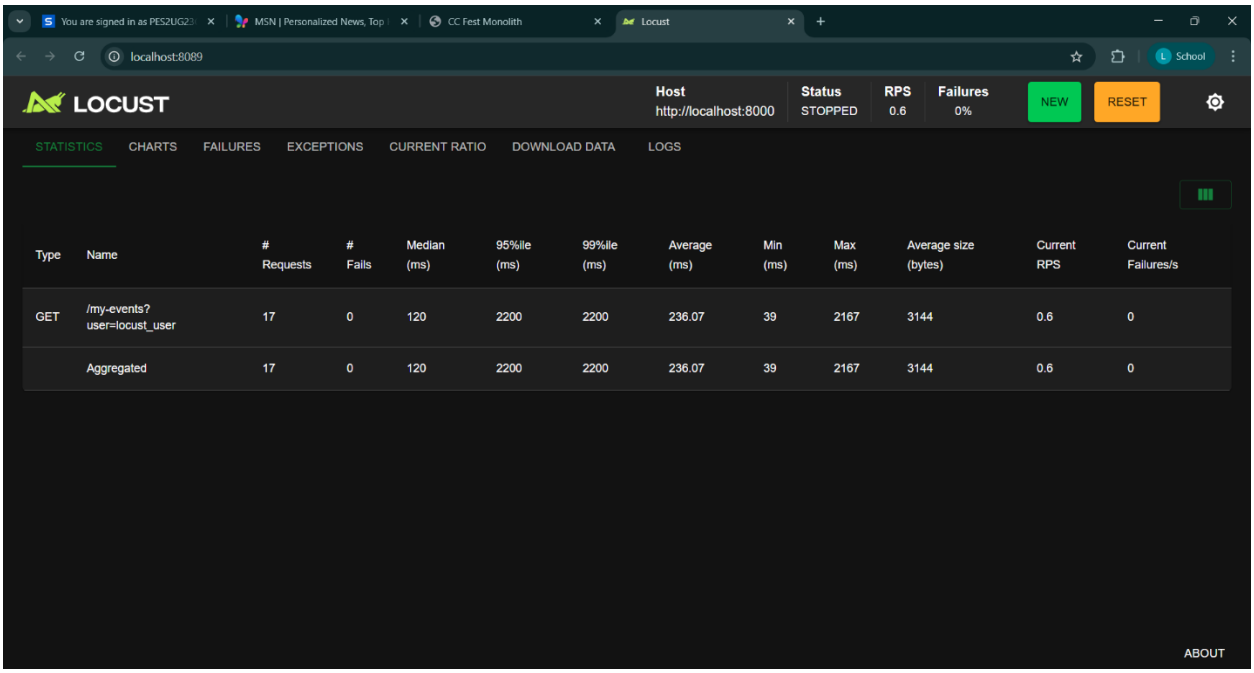


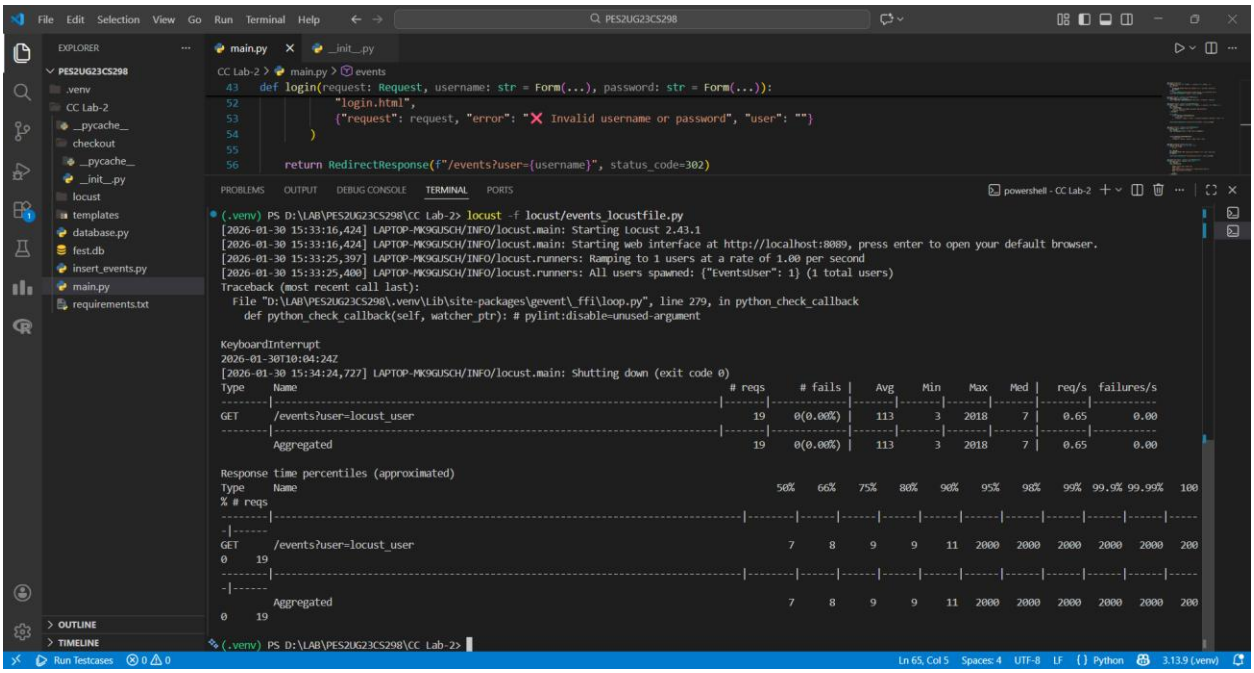
Before events route optimisation



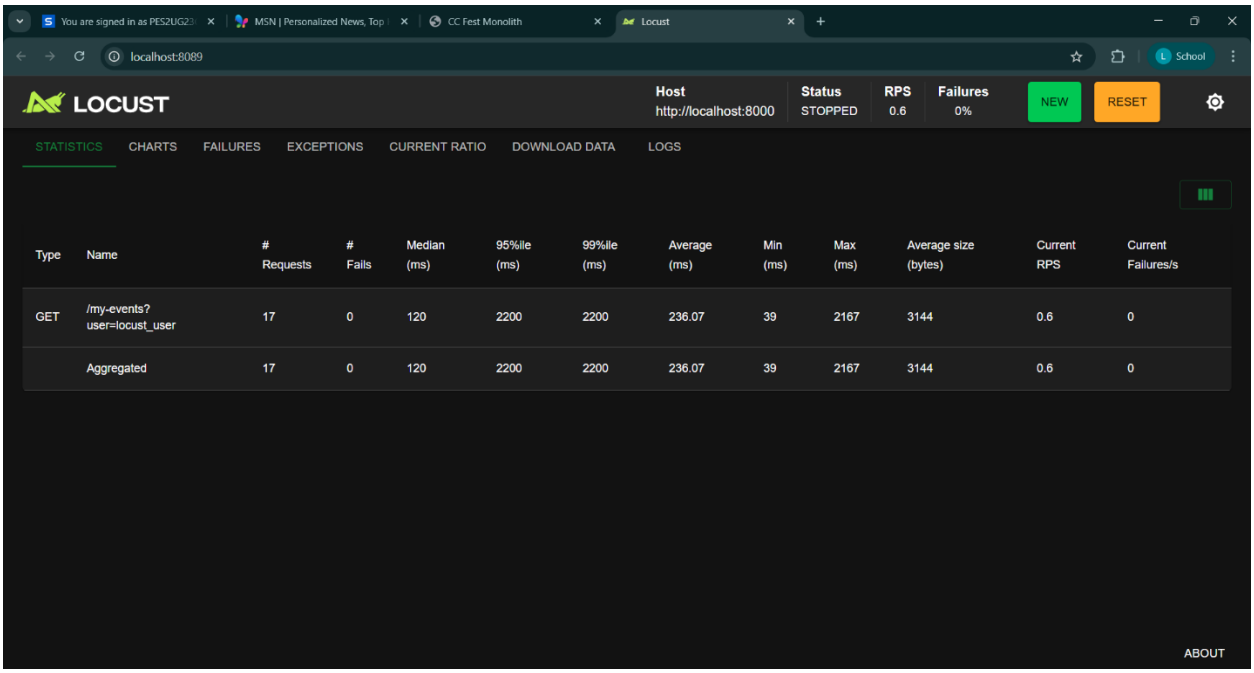


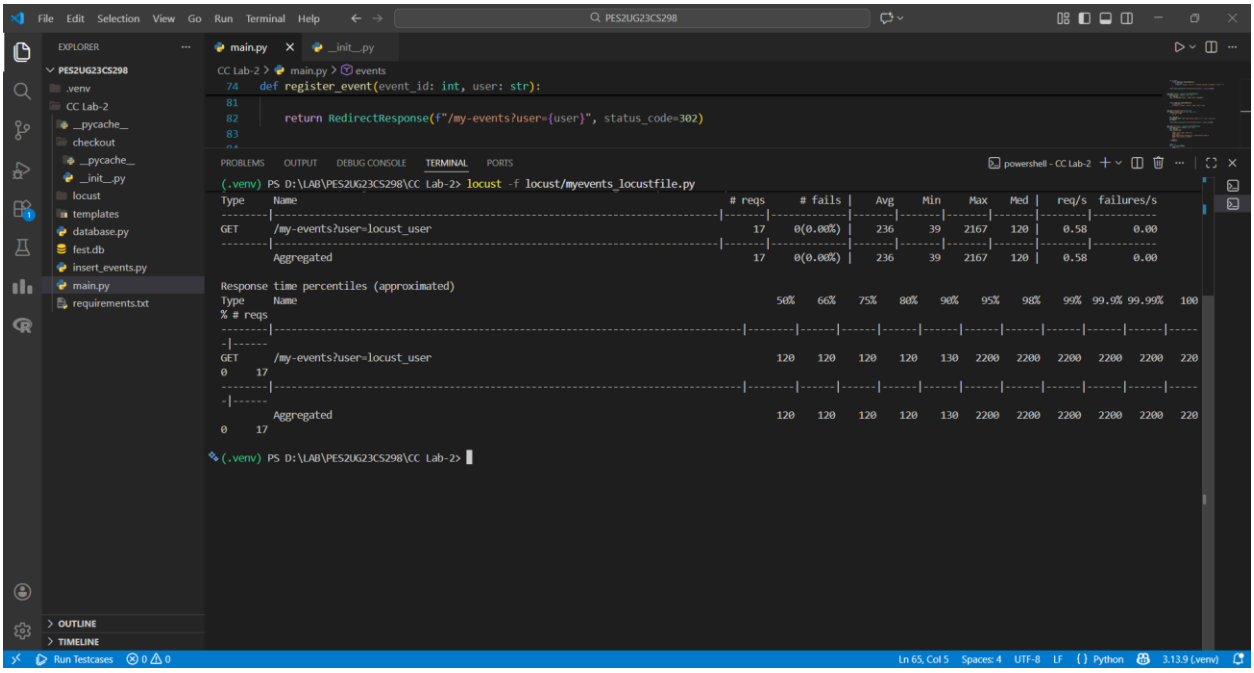
After events route optimisation



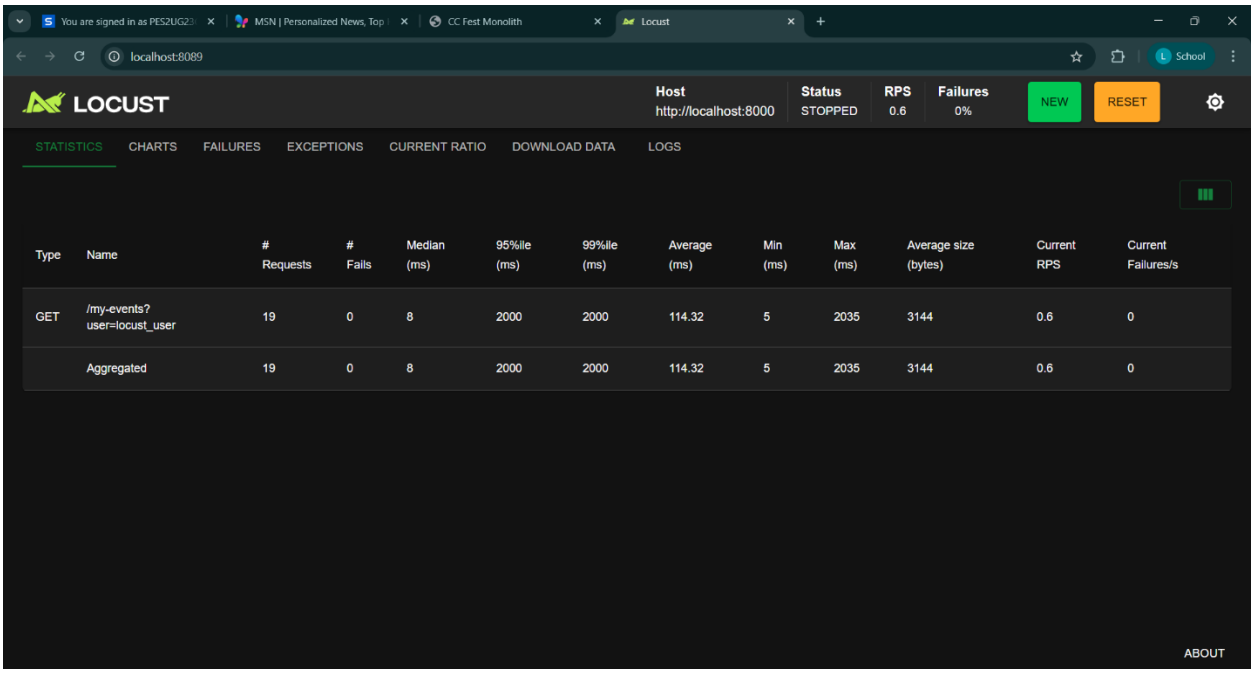


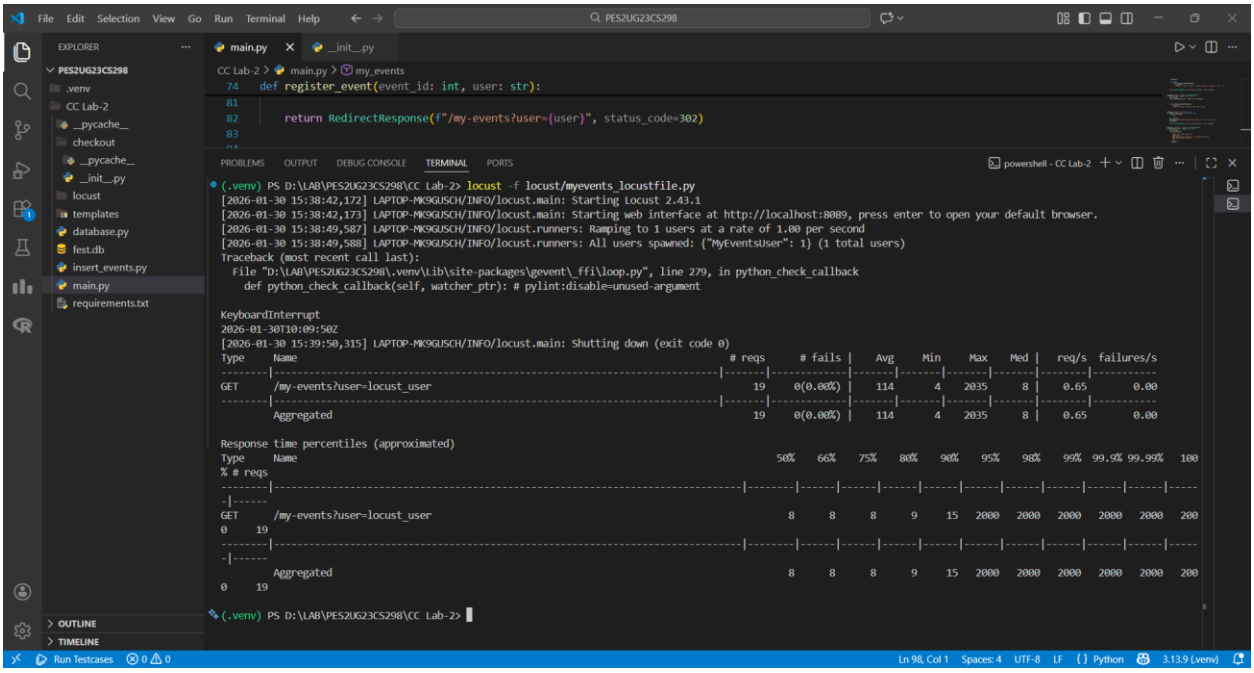
Before my-events route optimisation





After my-events route optimisation





Checkout Load Test (Before Optimization)

Observation:

- Average response time is relatively high
- Requests per second (RPS) are low
- Performance degrades due to inefficient computation

Reason:
The checkout logic uses an inefficient loop-based calculation.

Checkout Load Test (After Optimization)

What changed?
The inefficient loop was replaced with a direct aggregation over event prices.

Result:

- Average response time decreased
- RPS remained approximately the same

Conclusion:
The route became faster due to reduced computational overhead, **even though system capacity stayed constant.**

Route: /events

Bottleneck Identified:
The /events route contained a dummy loop that performed unnecessary iterations without contributing to the response. This caused extra CPU usage and increased response time under load.

Optimization Performed:
The dummy loop was **removed completely**, allowing the route to return event data directly without redundant processing.

Why Performance Improved:
By eliminating unnecessary computation, the CPU workload per request was reduced. This resulted in **lower response time** and improved performance during load testing.

Route: /my-events

Bottleneck Identified:
Similar to the /events route, /my-events contained a dummy loop that executed pointlessly for every request, increasing execution time.

Optimization Performed:
The dummy loop was **removed**, and only the required logic for fetching user-specific events was retained.

Why Performance Improved:
Removing the redundant loop reduced processing overhead, making request handling faster and improving response times under concurrent load.

Final Conclusion

- In both routes, performance improved because unnecessary dummy loops were removed, reducing computational overhead and making the monolithic application more efficient under load.
- Monolithic applications are easy to build and deploy
- They suffer from poor fault isolation
- Performance optimization helps, but scalability and reliability remain limited
- Microservices solve these issues by isolating failures and scaling independently