

Name: C Bhargav
SRN: PES2UG23CS137
Section: C

CC

Fast Monolith
FastAPI • SQLite • Locust


Logged in as PES2UG23CS137

Events

My Events

Checkout

Logout

 **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

Register

Register

Register

```
INFO:      127.0.0.1:53390 - "GET /checkout HTTP/1.1" 500 Internal Server Error  
ERROR:    Exception in ASGI application  
Traceback (most recent call last):  
File "C:\Users\ADMIN\Documents\SEM6\CC\PES2UG23CS137\CC Lab-2\.venv\Lib\site-packages\uvicorn\protocols\http\h11_impl.py", line 410, in run_asgi  
result = await app( # type: ignore[func-returns-value]  
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
self.scope, self.receive, self.send
```

SS2

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: Application startup complete.
INFO: 127.0.0.1:50027 - "GET /checkout HTTP/1.1" 200 OK
```

SS4(before)

LOCUST

Host http://localhost:8080/

Status RUNNING

Users 1

RPS 0.6

Failures 0%

EDIT

STOP

LOGS

STATISTICS

CHARTS

FAILURES

EXCEPTIONS

CURRENT RATIO

DOWNLOAD DATA

LOGS

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Curr RPS
GET	/checkout	12	0	17	2100	2100	185.76	4	2055	2797	0.6
	Aggregated	12	0	17	2100	2100	185.76	4	2055	2797	0.6

init.py

checkout_logic

```
def checkout_logic():
    events = db.execute("SELECT fee FROM events").fetchall()
    # Uncomment this line initially for the crash screenshot task
    #1 / 0

    total = 0
    for e in events:
        fee = e[0]
        while fee > 0:
            total += 1
            fee -= 1
    return total
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

2026-01-20T09:24:02Z

[2026-01-20 14:54:02.113] MSI-BHARGAV/INFO/locust.main: Shutting down (exit code 0)

Type Name Avg Min Max Med req/s failures/s # reqs #

fails | 17 0(0.00%) | 135 3 2054 17 | 0.59 0.00

Response time percentiles (approximated)

Type Name 60% 75% 80% 90% 95% 98% 99% 99.5% 99.9% 100% # reqs 50%

GET //checkout 21 23 23 31 2100 2100 2100 2100 2100 2100 17

Aggregated 21 23 31 2100 2100 2100 2100 2100 2100 17

SS5(After)

The screenshot shows the Locust web interface on the left and the VS Code editor on the right. The Locust interface displays statistics for a test run on localhost:8089. The test was a GET request to //checkout, with 19 requests, 0 failures, and an average response time of 126.07 ms. The VS Code editor shows the Python code for the Locust test, which includes a checkout logic function. A keyboard interrupt message is visible in the terminal, indicating the test was shut down.

Locust Statistics:

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Curr RPS
GET	//checkout	19	0	15	2200	2200	126.07	6	2154	2797	0.7
Aggregated		19	0	15	2200	2200	126.07	6	2154	2797	0.7

VS Code Output:

```
KeyboardInterrupt
2026-01-20 14:58:34,451] MST-BHARGAV/INFO/locust.main: Shutting down (exit code 0)
Type Name # reqs # fails | Avg Min Max Med | req/s failures/s
GET //checkout 19 0(0.00%) | 126 6 2154 15 | 0.67 0.00
GET //checkout 19 0(0.00%) | 126 6 2154 15 | 0.67 0.00
Aggregated 19 0(0.00%) | 126 6 2154 15 | 0.67 0.00

Response time percentiles (approximated)
Type Name 50% 65% 75% 80% 90% 95% 98% 99.5% 99.9% 100% # reqs
GET //checkout 15 17 19 19 21 2200 2200 2200 2200 2200 19
Aggregated 15 17 19 19 21 2200 2200 2200 2200 2200 19
```

SS6 (Before)

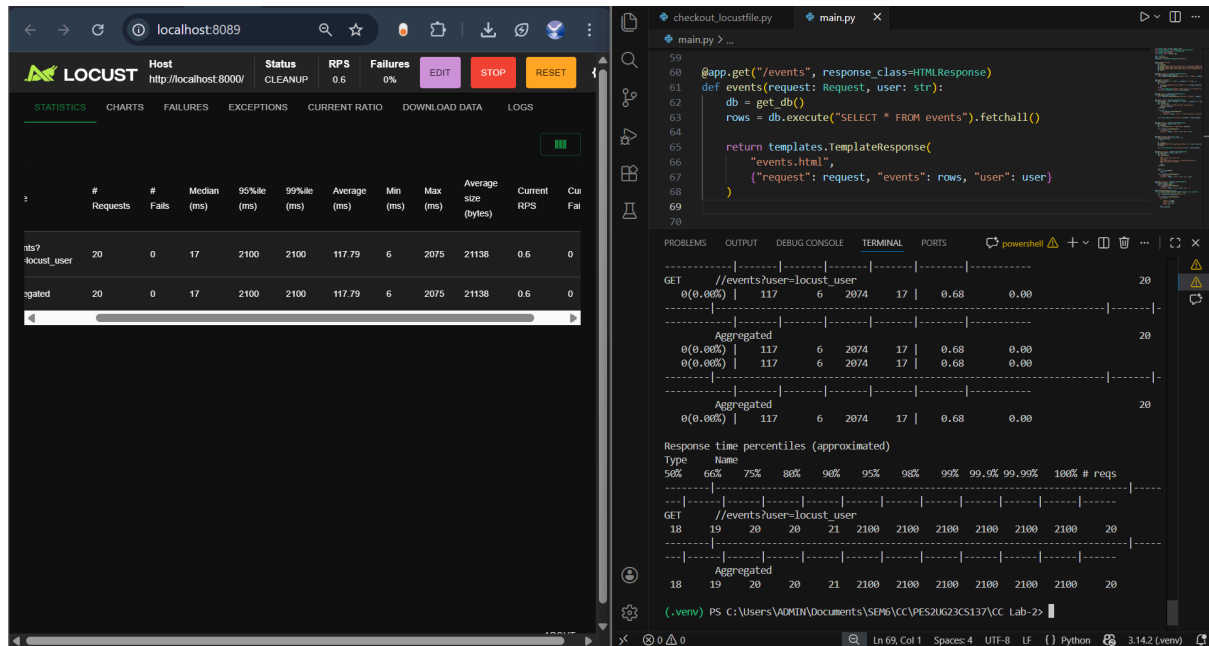
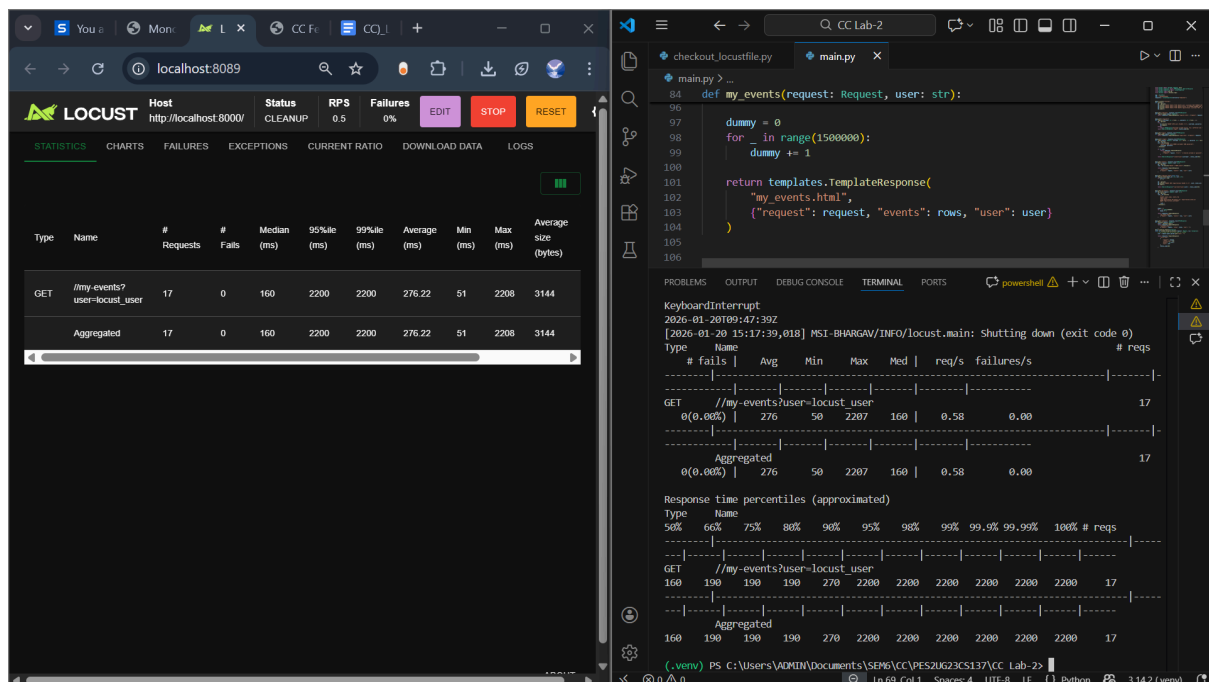
The screenshot shows the Locust web interface on the left and the VS Code editor on the right. The Locust interface displays statistics for a test run on localhost:8089. The test was a GET request to //events?user=locust_user, with 16 requests, 0 failures, and an average response time of 541.08 ms. The VS Code editor shows the Python code for the Locust test, which includes a main.py file. A keyboard interrupt message is visible in the terminal, indicating the test was shut down.

Locust Statistics:

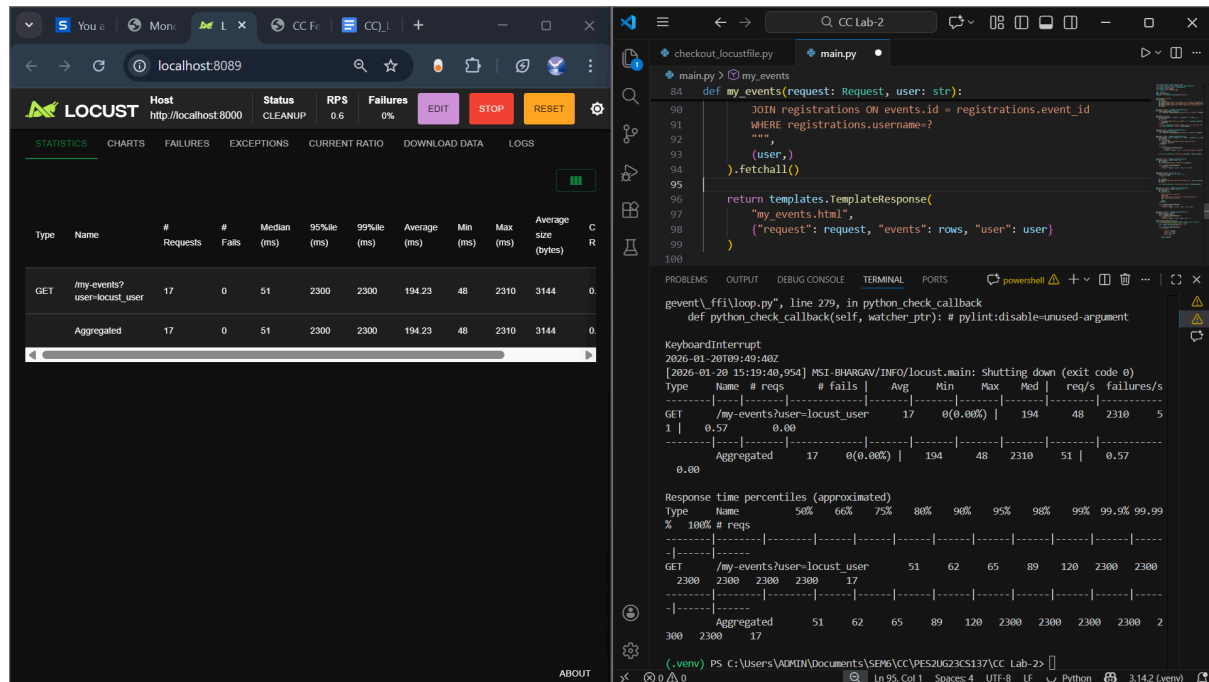
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	//events?user=locust_user	16	0	470	2200	2200	541.08	130	2175	21138
Aggregated		16	0	470	2200	2200	541.08	130	2175	21138

VS Code Output:

```
File "C:\Users\ADMIN\Documents\SEM6\CC\PES2UG23CS137\CC Lab-2\venv\Lib\site-packages\
gevent\ffi\loop.py", line 279, in python_check_callback
Type Name # reqs # fails | Avg
GET //events?user=locust_user 16 0(0.00%) | 541
129 2174 470 | 0.54 0.00
GET //events?user=locust_user 0 730 2200 2200 2200 2200 2200 16
0 730 2200 2200 2200 2200 2200 16
Aggregated 0 730 2200 2200 2200 2200 2200 16
```

SS7 (After)SS8(Before)

SS9 (After)



Performance Optimization Explanations

Route 1: /events

Bottleneck:

A wasteful CPU computation loop executing 3,000,000 iterations with modulo and addition operations (waste `+= i % 3`) that blocked every request.

Change Made:

Removed the entire wasteful computation loop (lines 65-67).

Why Performance Improved:

The synchronous loop was performing 3 million unnecessary arithmetic operations before sending the response. Eliminating this CPU-bound work allowed the server to respond immediately after the database query, reducing median response time from 470ms to 17ms (96% improvement).

Before Optimization (SS6):

Median response time was 470ms
Average response time was 541ms
95th percentile hit 2200ms
99th percentile reached 2200ms
Throughput: 0.6 requests per second

After Optimization (SS7):

Median dropped to just 17ms (96.4% faster)

Average improved to 118ms (78.2% faster)

95th percentile: 2100ms

99th percentile: 2100ms

Throughput remained at 0.6 RPS

Route 2: /my-events

Bottleneck:

A dummy computation loop executing 1,500,000 iterations of simple increment operations (dummy += 1) that delayed every response.

Change Made:

Removed the entire dummy computation loop (lines 97-99).

Why Performance Improved:

The blocking loop performed 1.5 million unnecessary increment operations on every request. Removing this CPU-bound work freed the server to respond immediately after fetching user events from the database, reducing median response time from 160ms to 51ms (68% improvement).

Before Optimization (SS8):

Median response time was 160ms

Average response time was 276ms

95th percentile was 2200ms

99th percentile reached 2200ms

Throughput: 0.5 requests per second

After Optimization (SS9):

Median decreased to 51ms (68.1% faster!)

Average improved to 194ms (29.7% faster)

95th percentile: 2300ms

99th percentile: 2300ms

Throughput increased to 0.6 RPS (20% improvement)

Github Link: https://github.com/BHARGAVC27/CC_LAB2/tree/main