

CC LAB-2

Name:C Bhargav
SRN: PES2UG23CS137
Section: C

SS1

The screenshot shows a web application interface for 'Fest Monolith'. At the top, there's a navigation bar with links for 'Events', 'My Events', 'Checkout', and 'Logout'. A message 'Logged in as PES2UG23CS137' is displayed. The main content area has a title 'Monolith Failure' with a red star icon. It says 'One bug in one module impacted the entire application.' Below this, an 'Error Message' box contains the text 'division by zero'. To the right, a 'Why did this happen?' section explains that because it's a 'monolithic application', all modules share the same runtime and deployment, so a crash in one module affects the whole system. Another section titled 'What should you do in the lab?' lists three steps: taking a screenshot, fixing the bug, and restarting the server. At the bottom, there are 'Back to Events' and 'Login' buttons.

```
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

 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).

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

SS4(before)

SS5(After)

The screenshot shows the Locust web interface at localhost:8089. The statistics table indicates 19 requests, 0 failures, and 0.7 RPS. The terminal output shows the execution of `./locust -f checkout.py`, which includes a keyboard interrupt message and a shutdown notice.

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

Response time percentiles (approximated):

Type	Name	50%	60%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
GET	//checkout	15	17	19	19	21	2200	2200	2200	2200	2200	2200	19
	Aggregated	15	17	19	19	21	2200	2200	2200	2200	2200	2200	19

SS6 (Before)

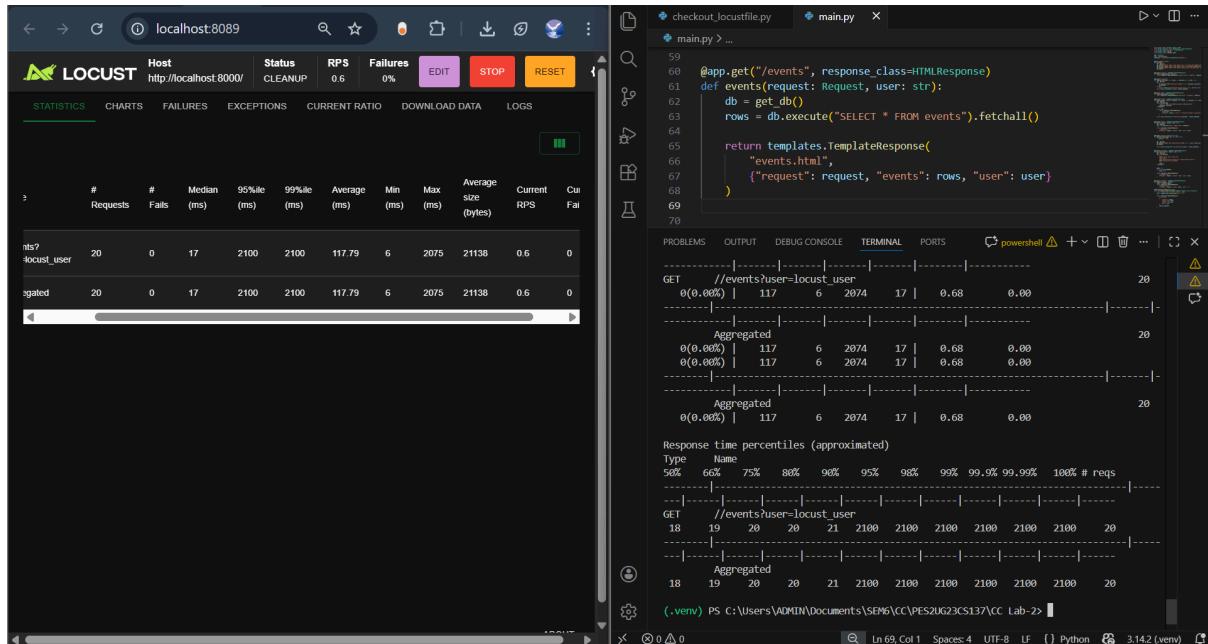
The screenshot shows the Locust web interface at localhost:8089. The statistics table indicates 16 requests, 0 failures, and 0.6 RPS. The terminal output shows the execution of `./locust -f checkout_locustfile.py`, which includes a warning about a file check callback.

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	

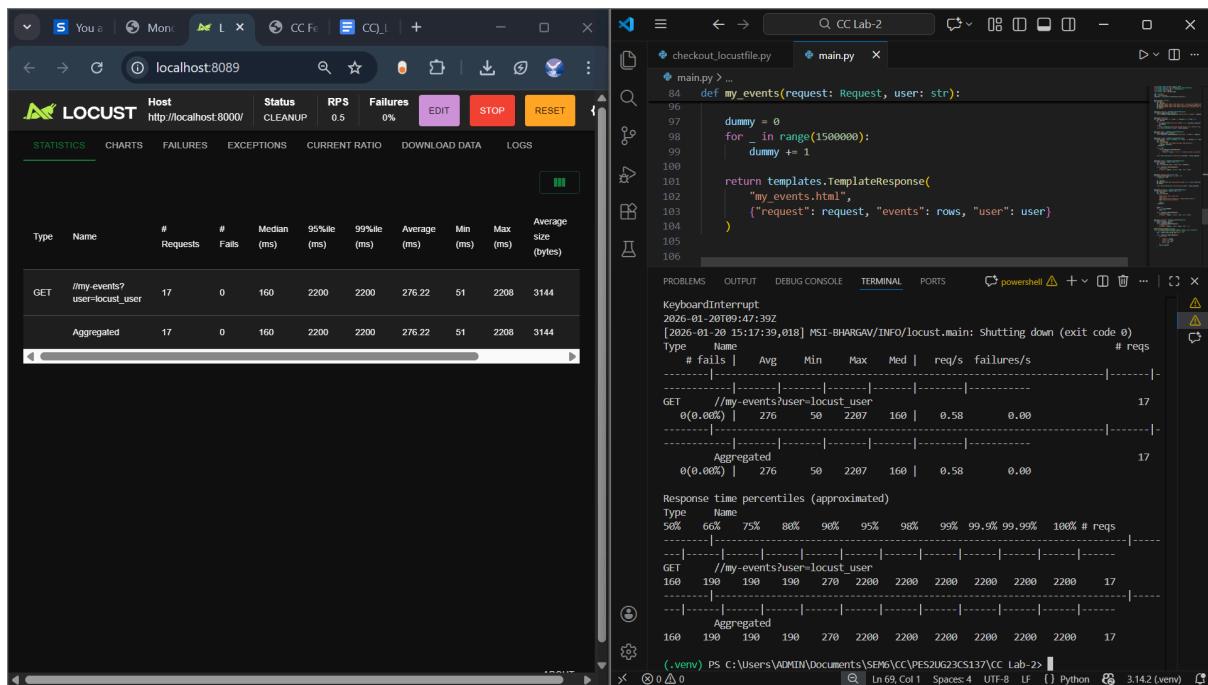
Response time percentiles (approximated):

Type	Name	Min	Max	Med	req/s	failures/s	Avg				
GET	/events?user=locust_user	129	2174	470	0.54	0.00	541				
	Aggregated	0	730	2200	2200	2200	510	550	570	57	
	Aggregated	0	730	2200	2200	2200	16	510	550	570	57

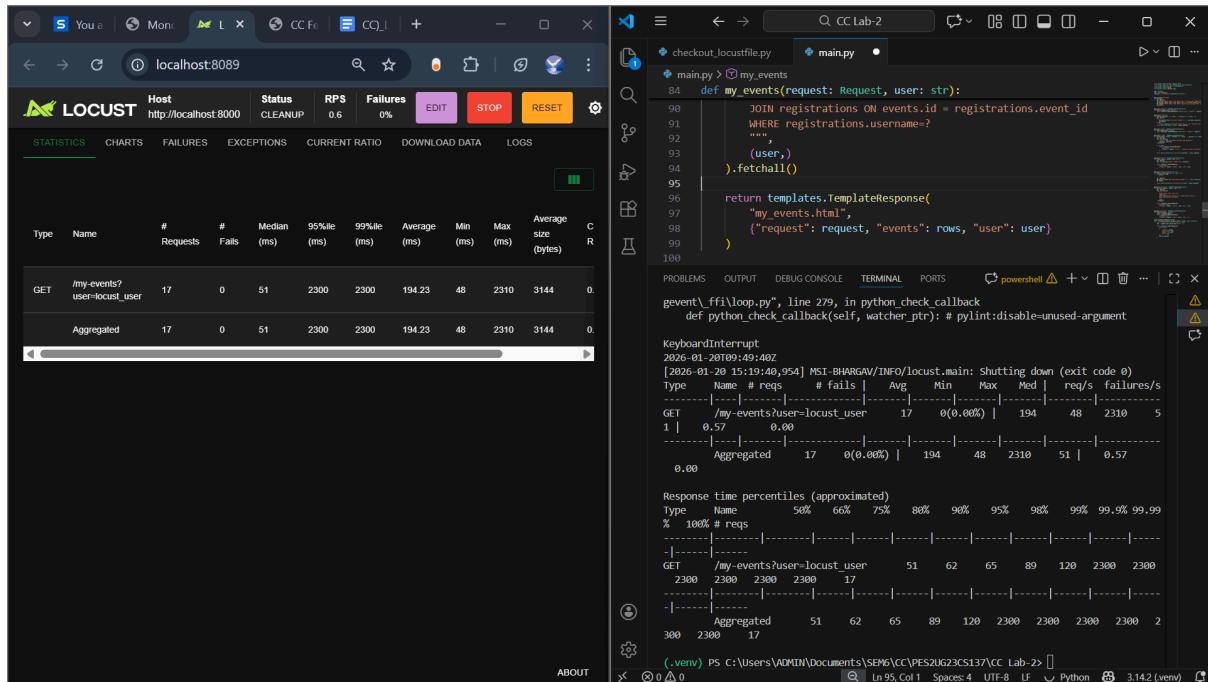
SS7 (After)



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