

CC LAB 2

NAME : CHARAN L

SRN : PES1UG23CS159

SECTION : C

GITHUB LINK : [GitHub - Charan-1810/PES1UG23CS159_CCLAB-2](#)

SCREENSHOTS:

SS1

The screenshot shows a web application interface for event registration. At the top, there's a header bar with the logo 'Fest Monolith' (FastAPI · SQLite · Locust), user information ('Logged in as PES1UG23CS159'), and navigation links for 'Events', 'My Events', 'Checkout', and 'Logout'. A 'View My Events →' button is also present. Below the header, the main content area is titled 'Events' with a sub-instruction 'Welcome PES1UG23CS159. Register for events below.' There are nine event cards arranged in a 3x3 grid:

- Event ID: 1** (₹ 500) **Hackathon**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 2** (₹ 300) **Dance**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 3** (₹ 500) **Hackathon**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 4** (₹ 300) **Dance Battle**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 5** (₹ 400) **AI Workshop**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 6** (₹ 200) **Photography Walk**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 7** (₹ 350) **Gaming Tournament**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 8** (₹ 250) **Music Night**: Includes certificate • instant registration • limited seats. **Register**
- Event ID: 9** (₹ 150) **Treasure Hunt**: Includes certificate • instant registration • limited seats. **Register**

SS2:



💥 Monolith Failure

One bug in one module impacted the **entire application**.

HTTP 500

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

SS3:



💳 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

SS4:

The image shows two side-by-side screenshots. On the left is the Locust web interface, displaying performance statistics for a 'checkout' endpoint. The table includes columns for Type, Name, # Requests, # Fails, Median (ms), 95%ile (ms), 99%ile (ms), and Average (ms). The right screenshot shows a portion of a Python file named `__init__.py` in a code editor, containing logic for a 'checkout' function. Below the editor is the VS Code terminal, showing detailed performance metrics for the 'checkout' endpoint, including response time percentiles and request rates.

Locust Performance Statistics:

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)
GET	/checkout	20	0	7	49	49	9.85
Aggregated		20	0	7	49	49	9.85

VS Code Terminal Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... > < V TERMINAL
> Type      Name    # reqs     # fails | Avg   Min   Max   Me
> d | req/s   failures/s
> -----+-----+-----+-----+-----+-----+-----+-----+-----+
> GET      /checkout  20        0(0.00%) | 9     2     48
>          7 | 0.67    0.00
> -----
> Aggregated 20        0(0.00%) | 9     2     48
>          7 | 0.67    0.00

Response time percentiles (approximated)
Type      Name    50%   66%   75%   80%   90%   95%   98%
99%      99.9% 99.99% 100% # reqs
-----+-----+-----+-----+-----+-----+-----+-----+
GET      /checkout 7       9       10      13     14     49
```

SS5:

SS6:

The screenshot shows a dual-monitor setup. The left monitor displays the Locust web interface, which includes a navigation bar with 'localhost' and 'Locust' tabs, and a main area with tabs for 'STATISTICS', 'CHARTS', 'FAILURES', 'EXCEPTIONS', and 'CURRENT RATIO'. Below these tabs is a table showing performance metrics for a single request type ('GET') and an aggregated row. The right monitor displays a dark-themed code editor in VS Code, showing Python code for a Locust user script named 'events_locustfile.py'. The code defines a class 'EventsUser' that inherits from 'HttpUser' and contains a task 'view_events' that makes a GET request to '/events?user=locust_user'. The bottom part of the VS Code window shows the 'TERMINAL' tab, which displays the output of the Locust command, including summary statistics and response time percentiles.

Locust

localhost Locust

STATISTICS CHARTS FAILURES EXCEPTIONS CURRENT RATIO >

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)
GET	/events?user=locust_user	19	0	160	180	180	151.67
Aggregated							
		19	0	160	180	180	151.67

`__init__.py events_locustfile.py`

```
CC_Lab2 > locust > events_locustfile.py > ...
1   from locust import HttpUser, task, between
2
3   class EventsUser(HttpUser):
4       wait_time = between(1, 2)
5
6       @task
7       def view_events(self):
8           self.client.get("/events?user=locust_user")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ...

> < zsh - CC_Lab2 + ...

```
hutting down (exit code 0)
Type      Name # reqs    # fails | Avg     Min     Max     Me
d |      req/s failures/s
-----|-----|-----|-----|-----|-----|-----|-----|-----|
GET      /events?user=locust_user      19      0(0.00%) | 151
125      177      160 | 0.64      0.00
-----|-----|-----|-----|-----|-----|-----|-----|
Aggregated          19      0(0.00%) | 151      125      177
160      0.64      0.00
-----|-----|-----|-----|-----|-----|-----|-----|
Response time percentiles (approximated)
Type      Name 50% 66% 75% 80% 90% 95% 98%
99%      99.9 99.99 100% # reqs
-----|-----|-----|-----|-----|-----|-----|-----|

```

SS7:

The screenshot shows two windows side-by-side. On the left is the Locust web interface, displaying performance statistics for a test. On the right is a PyCharm code editor showing Python code for a Locust user task.

Locust Performance Statistics:

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)
GET	/events?user=locust_user	12	0	160	170	170	158.03
GET	/health	1	1	3.58	4	4	3.58
Aggregated		13	1	160	170	170	146.15

PyCharm Code Editor (events_locustfile.py):

```
CC_Lab2 > locust > events_locustfile.py > ...
3     class EventsUser(HttpUser):
16         def view_events(self):
22             if response.status_code != 200:
23                 response.failure("Failed to fetch events")
25             # Secondary task
26             @task(1)
27             def health_check(self):
28                 self.client.get("/health")
31
```

PyCharm Terminal:

```
> zsh - CC_Lab2
4 4 4 4 4 1
--|-----|-----|-----|-----|-----|-----|-----|-----|
--|-----|-----|-----|-----|-----|-----|-----|-----|
170 170 170 170 170 170 170 170
Error report
# occurrences      Error
-----
1          GET /health: HTTPError('404 Client Error: Not
Found for url: /health')
-----
```

ss8:

The screenshot shows two monitors. The left monitor displays the Locust web interface with a green header 'LOCUST'. It has tabs for STATISTICS, CHARTS, FAILURES, EXCEPTIONS, and CURRENT RATIO. Under the STATISTICS tab, there is a table with columns: Type, Name, # Requests, # Fails, Median (ms), 95%ile (ms), 99%ile (ms), and Average (ms). The table shows one row for a GET request to '/my-events?user=locust_user' with 20 requests, 0 fails, median 86 ms, 95%ile 92 ms, 99%ile 92 ms, and average 84.79 ms. Below this is an 'Aggregated' row with the same values. The right monitor shows a code editor with Python files: __init__.py, events_locustfile.py, and main.py. The events_locustfile.py file contains code for a 'EventsUser' class with a 'view_events' task. The main.py file contains an exception handler for global exceptions. Below the code editors is a terminal window showing Locust test results. The terminal output includes a summary table, response time percentiles, and a detailed table for each request type.

SS9:

This screenshot shows the same dual-monitor setup as the previous one. The left monitor displays the Locust web interface with a green header 'LOCUST'. The right monitor shows the code editor and terminal window. The code editor now shows a different version of main.py with a global exception handler that returns an error template. The terminal window shows the results of a new test with 22 requests, 0 fails, median 7 ms, 95%ile 12 ms, 99%ile 53 ms, and average 9.35 ms. The results table is identical to the one in the first screenshot.

QUESTIONS:

1.What was the bottleneck?

The bottleneck was **inefficient data processing and repeated looping over event records**, which caused unnecessary CPU usage and increased response time during load testing.

2.What change did you make?

I optimized the logic by:

- Removing unnecessary loops
- Using direct aggregation instead of incremental counting
- Reducing redundant operations while fetching and processing event data

3.Why did the performance improve?

Performance improved because:

- Fewer iterations were executed
- CPU workload was reduced
- The response was generated using optimized logic

This decreased average response time and improved overall throughput

4.Explanation about optimization

Optimization was done by removing inefficient loops and redundant computations, and by simplifying data processing logic. These changes reduced CPU usage and execution time per request. As a result, average response time decreased while handling the same number of requests, demonstrating improved performance of the monolithic application.