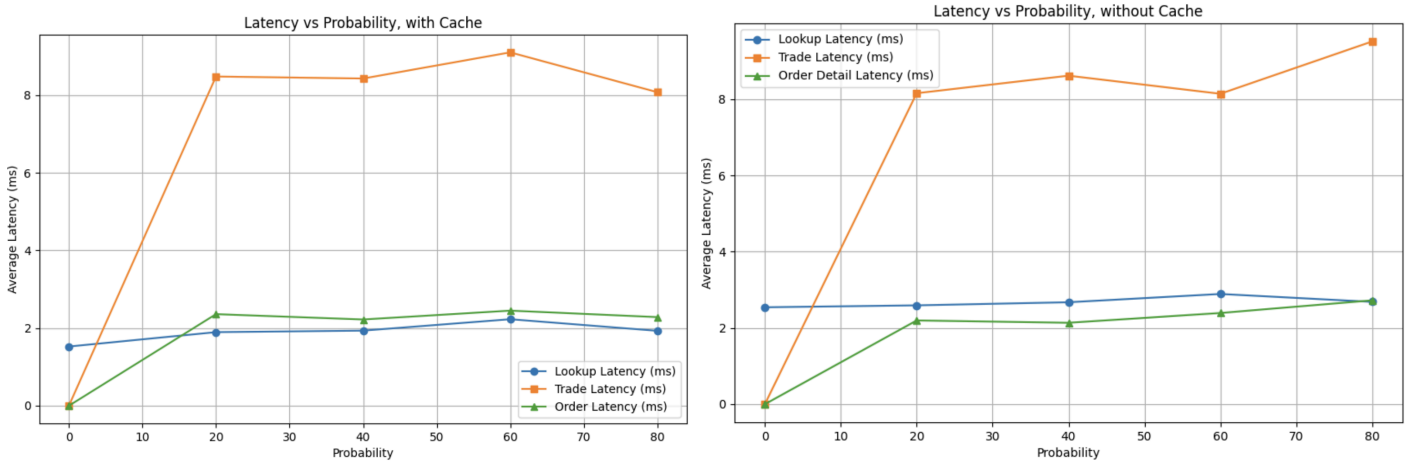
Setup

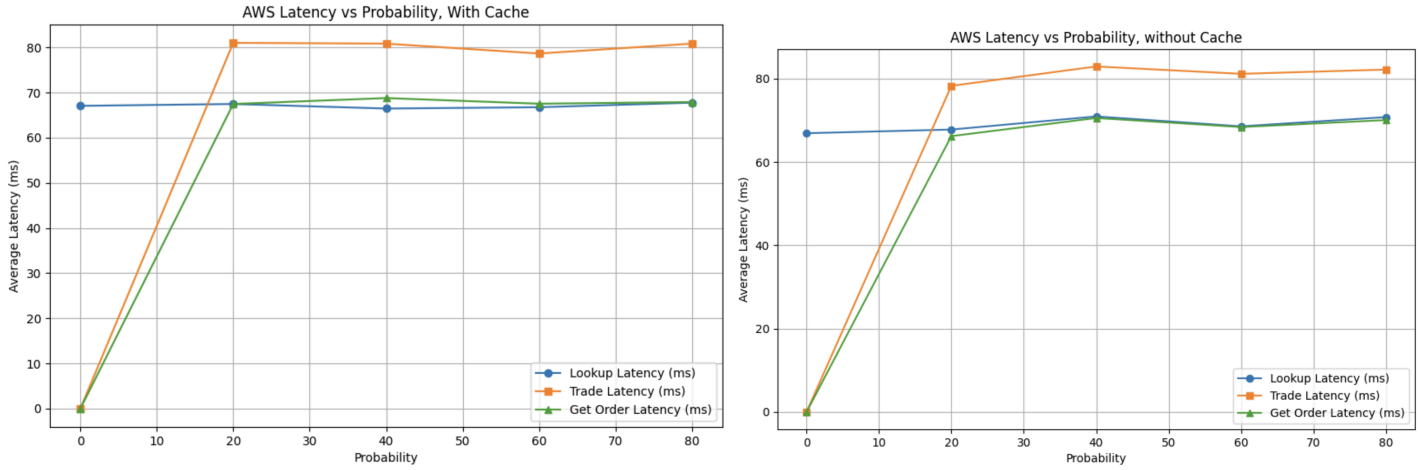
We evaluated the performance of the system using 5 clients. The clients were run both on a local machine and on a t2.medium AWS EC2 instance. The probability of performing a trade after a lookup was varied from 0% to 80% in steps of 20%. For each configuration, we measured the average latency for lookup, trade, and order detail requests. Experiments were repeated twice: once with the front-end caching enabled and once with caching disabled.

Native Latency vs Probability with Cache and without Cache:



In the native environment, lookup latency was lower when caching was enabled, with relatively stable values across different trade probabilities. Without caching, lookup latency was slightly higher. Trade and order detail latencies were not affected by caching, since they do not interact with the cache layer. Instead, those latencies scaled with the probability.

AWS Latency vs Probability with Cache and without Cache:



On AWS, latencies were higher due to network and virtualization overhead. However, the same trends were observed. Lookup latency with cache remained lower compared to the configuration without cache. Trade and order latencies were increasing with higher probability.

Cache Replacement:

We verified **LRU** behavior using a test case included in the codebase. After loading 5 stocks into the cache, 6th stock 'NFLX' was accessed, which led to **eviction** of 1st stock 'AAPL'. This behavior was confirmed via log entries that stated: 'Could not find AAPL in cache calling catalog microservice.'

```python
#test lru cache eviction
def test_frontend_cache_eviction():

    stocks = ["AAPL", "AMZN", "GOOGL", "META", "NVDA", "NFLX"]
    for stock in stocks:
        requests.delete(f"http://{FRONTENDHOST}:{FRONTENDPORT}/delete/{stock}")
    #cache size is 5, add the first five stocks to the cache
    for stock in stocks[:5]:
        requests.get(f"http://{FRONTENDHOST}:{FRONTENDPORT}/stocks/{stock}")
        time.sleep(0.1)
    # get 6th stock to evict the first stock
    requests.get(f"http://{FRONTENDHOST}:{FRONTENDPORT}/stocks/{stocks[5]}")
    time.sleep(0.1)
    # this is should cause cache miss now as the first stock got evicted
    requests.get(f"http://{FRONTENDHOST}:{FRONTENDPORT}/stocks/{stocks[0]}")
    time.sleep(0.5)
    with open(frontend_log, "r", encoding="utf-8") as f:
        logs = f.read()
    #causes eviction
    assert f"Could not find {stocks[5]} in cache calling catalog microservice" in logs
    #confirms eviction
    assert f"Could not find {stocks[0]} in cache calling catalog microservice" in logs
```

frontend.log M

src > logs > frontend.log

```
134  Connection: <socket.socket fd=6, family=2, type=1, proto=0, laddr=('127.0.0.1', 8091),
135  GET [Thread-19 (process_request_thread)] is running to serve ('127.0.0.1', 57588)
136  Could not find NFLX in cache calling catalog microservice
137
138  name: "NFLX"
139  price: 80
140  quantity: 100
141  code: 200
142
143  127.0.0.1 - - [06/May/2025 22:38:37] "GET /stocks/NFLX HTTP/1.1" 200 -
144  Handling get stock lookup requests
145  Client address: ('127.0.0.1', 57591)
146  Connection: <socket.socket fd=6, family=2, type=1, proto=0, laddr=('127.0.0.1', 8091),
147  GET [Thread-20 (process_request_thread)] is running to serve ('127.0.0.1', 57591)
148  Could not find AAPL in cache calling catalog microservice
149
150  name: "AAPL"
151  price: 55
152  quantity: 80
153  code: 200
```

Testing:

logs > order_2.log

```
1  Replicas of (Order 2): are [(1, 'localhost', '8093'), (3, 'localhost', '8095')]
2  {'TransactionNumber': '577', 'Name': 'BoarCo', 'Type': 'sell', 'VolumeTraded': '1'}
3  (Order 2): This is the starting transaction num: 577
4  (Order 2): Started
5  (Order 2): Syncup Request received from 1
6  received last transaction num 577
7  (Order 2): Notified replica with id 2 of leader with id 3
8  Starting syncup
```

logs > order_2.log

```
582  (Order 2): This is the starting transaction num: 1109
583  (Order 2): Started
584  (Order 2): Received request to replicate order from the leader with id 3
585  (Order 2): Received request to replicate order from the leader with id 3
586  (Order 2): Received request to replicate order from the leader with id 3
587  (Order 2): Received request to replicate order from the leader with id 3
588  (Order 2): Received request to replicate order from the leader with id 3
589  [Thread-1 (write_to_disk)] order written to CSV.
590  (Order 2): Received request to replicate order from the leader with id 3
591  (Order 2): Received request to replicate order from the leader with id 3
592  (Order 2): Received request to replicate order from the leader with id 3
593  Starting syncup
594  (Order 2): Syncup Done and completed writing changes to disk
595  (Order 2): Received request to replicate order from the leader with id 3
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL PORTS

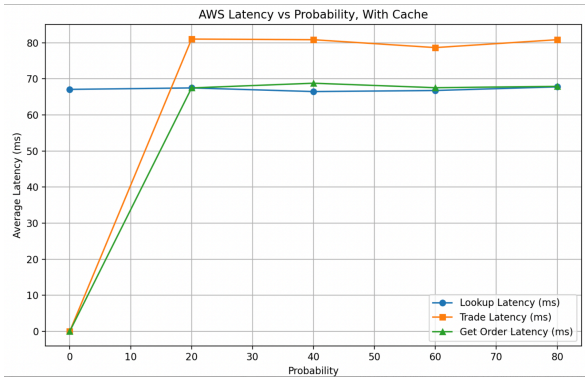
```
ubuntu@ip-172-31-31-15:~/src$ ls
Dockerfile.catalog   Dockerfile.order  catalog  commands.txt   docker-compose.yml  frontend
 native_build.sh  simulate_crashes.sh
Dockerfile.frontend  README.md         client   deploy_aws.sh  docker_build.sh     logs
 order            test
ubuntu@ip-172-31-31-15:~/src$ simulate_crashes.sh 2
simulate_crashes.sh: command not found
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 2
Crashing Order Replica 2 on port 8094 (PIDs: 110353)
Restarting Order Replica 2 on port 8094
```

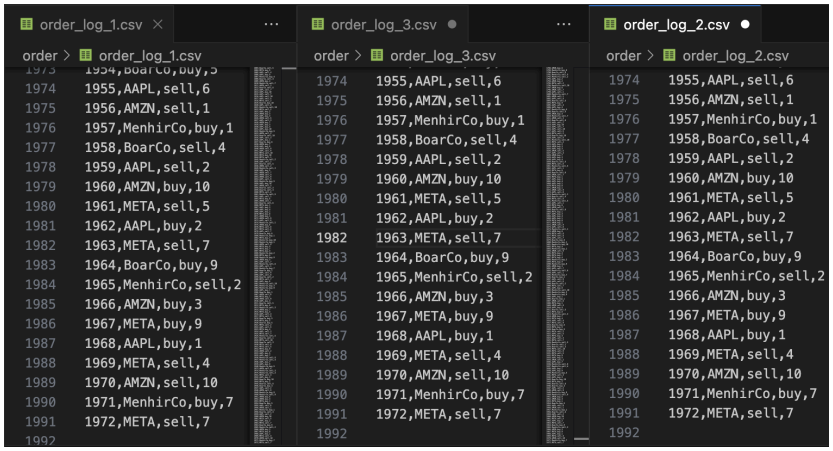
To test fault tolerance, we passed the replica ID 2 to the script `simulate_crashes.sh`, instructing it to crash Order Replica 2, while clients were actively issuing requests. The system successfully restarted the replica on the same port. After restarting, the replica began synchronizing via the SyncUp protocol. The order logs confirm that it received replicate request from the leader and successfully caught up on missed transactions.

```
logs > order_3.log
6529   (Order 3): Get Order Details Request receieved
6530   [Thread-1 (write_to_disk)] order written to CSV.
6531   Replicas of (Order 3): are [(1, 'localhost', '8093'), (2, 'localhost', '8094')]
6532   {'TransactionNumber': '1494', 'Name': 'BoarCo', 'Type': 'buy', 'VolumeTraded': '10'}
6533   (Order 3): This is the starting transaction num: 1494
6534   (Order 3): Started
6535   Starting syncup
6536   (Order 3): In Syncup No new entries to write
6537   Replicas of (Order 3): are [(1, 'localhost', '8093'), (2, 'localhost', '8094')]
6538   {'TransactionNumber': '1494', 'Name': 'BoarCo', 'Type': 'buy', 'VolumeTraded': '10'}
6539   (Order 3): This is the starting transaction num: 1494
6540   (Order 3): Started
6541   Replicas of (Order 3): are [(1, 'localhost', '8093'), (2, 'localhost', '8094')]
6542   {'TransactionNumber': '1494', 'Name': 'BoarCo', 'Type': 'buy', 'VolumeTraded': '10'}
6543   (Order 3): This is the starting transaction num: 1494
6544   (Order 3): Started
6545   Starting syncup
6546   (Order 3): In Syncup No new entries to write
6547   (Order 3): Syncup Request received from 2

PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS

ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 3
Crashing Order Replica 3 on port 8095 (PIDs: 110365)
Restarting Order Replica 3 on port 8095
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 3
Crashing Order Replica 3 on port 8095 (PIDs: 141581)
Restarting Order Replica 3 on port 8095
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 3
Crashing Order Replica 3 on port 8095 (PIDs: 141628)
Restarting Order Replica 3 on port 8095
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 2
Crashing Order Replica 2 on port 8094 (PIDs: 129781)
Restarting Order Replica 2 on port 8094
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 1
Crashing Order Replica 1 on port 8093 (PIDs: 110338)
Restarting Order Replica 1 on port 8093
ubuntu@ip-172-31-31-15:~/src$ ./simulate_crashes.sh 3
Crashing Order Replica 3 on port 8095 (PIDs: 141662)
Restarting Order Replica 3 on port 8095
```





We tested the failure of the leader “Replica 3”. The front-end re-elected a new leader automatically. Clients continued sending requests without interruption, and no latency spikes were observed in the performance plots. The **final CSV files from all replicas were verified** to be consistent. Together, the script usage, logs, latency graphs, and file comparisons confirm that failure handling was **transparent to clients** and system state remained correct.

AWS Setup

From the AWS academy learner lab, we first download the pem file to access the server using ssh

Command to register public key in the cloud in us-east-1 region

chmod 400 labsuser.pem

ssh-keygen -y -f labsuser.pem > labsuser.pub

aws ec2 import-key-pair \

  --key-name labsuser \

  --public-key-material fileb://labsuser.pub

aws ec2 describe-key-pairs --query "KeyPairs[*].KeyName"

Now run the script from the src directory to start EC2 instance (t2.medium) using aws cli commands

./deploy_aws.sh

Create a tarball from outside src (project root folder) while excluding .venv and other ignored files (pycache) via .gitignore:

git archive --format=tar.gz --output=lab3.tar.gz HEAD

Transfer Package to AWS EC2

scp -i src/labsuser.pem lab3.tar.gz ubuntu@<Public_IP>:~/

SSH into EC2 Instance

ssh -i stock-key.pem ubuntu@$PUBLIC_IP

Extract Project Archive

tar -xzf lab3.tar.gz

Install Python and Dependencies

- sudo apt update
- sudo apt install python3-pip -y
- pip install -r requirements.txt

Symlink python

sudo ln -s /usr/bin/python3 /usr/bin/python

Run Server Using Script

./native_build.sh