HW 7

Created	@October 26, 2025 5:00 PM

```
[default]
aws_access_key_id=ASIA3JMGFYXDG5FQSHXU
aws_secret_access_key=fEDXsvFBhkJ4m4Vhix763u1CH0bsKbXVALU00k3
9
aws_session_token=IQoJb3JpZ2luX2VjEN3//////////wEaCXVzLXdlc3QtMiJG
MEQCIBstTzFlUpViOcUyJc6rmc+DRfO/ogwJwZDPd7NrJ9lkAiBaj1+iHteVuD
2cDaLGuYN5d1ABFTywXHBQZsdtCDMD1Cq+AgiW//////////8BEAAaDDc3Nj
A1OTg2NDUxOCIM+79lLBWs5j3p5DWLKplC+lUtVu+PLdxN+tb4WtbqeyH/
43guCSVR3qgz63AkhDbCaTGMuYZ6CGT/oE4RkyhS7w7uWk0hPFtLCImg
YeLMYYvGZD3BEUF6/nLhiRv1lPQyvRkQbQzoYbsvrJcyf09Ul6/9y5DD+h+w
PNgBMAj/c0yoF0WbosFYI+Pqwx/ubC+hdi9gb6blamjRlUdB3txFS0GbFRM
+X7t+ix/vNxvtRfa5X9txE7v+TjeZu5wC2DiNUm85xoXLUmj2rWUM+icL5nYv
xT/1esDDm0KXMatCI2YcwT+DYGoWoWJCHLo1PzD+7/MXNQR8RPqFyzw2
3WYZJjbWRv09gHSGeh91FqWZoyVUroQy+QB5Xltqkxl1bIO+rjDBmPrHBjqe
AZe75JUN+wLGPO039RRAMIN3Vi2cjVbS0Mkk/1Q3d2kTN1z6Y6C5gXIN31
Z7t6nN0YUFv2H9QXBm5f56NWrb1LZRKL0igFDYIRXe/sxZFRZymoSV0u40
7Nw2W1IV2mDod5HG6KqOjBvN+nLLTPb5I/KXeNf5ITQvjPEuIht8i6MprujJq
8FNIQ36eRuzArEbXRjQoTRQe+0aRxgpryC4

aws configure set aws_session_token
# 2) 验证配置
aws sts get-caller-identity

# 3) 获取账号ID
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output
text)
echo $AWS_ACCOUNT_ID
# 应该显示: 776059864518

# 4) 登录ECR
aws ecr get-login-password --region us-east-1 | \
  docker login --username AWS --password-stdin \
  776059864518.dkr.ecr.us-east-1.amazonaws.com
```

```
alb_dns = "hw7-alb-66070677.us-west-2.elb.amazonaws.com"
ecr_api_url = "776059864518.dkr.ecr.us-west-2.amazonaws.com/hw7-api"
ecr_processor_url = "776059864518.dkr.ecr.us-west-2.amazonaws.com/hw7-processor"
sns_topic_arn = "arn:aws:sns:us-west-2:776059864518:hw7-orders"
sqs_queue_url = "https://sqs.us-west-2.amazonaws.com/776059864518/hw7-orders"
```

Part2

Phase 1: Build Your Current System

```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/tests && locust -f locustfile.py --host=http://hw7-alb-66070677.us-west-2.elb.amazonaws.com SyncOrderUser
```

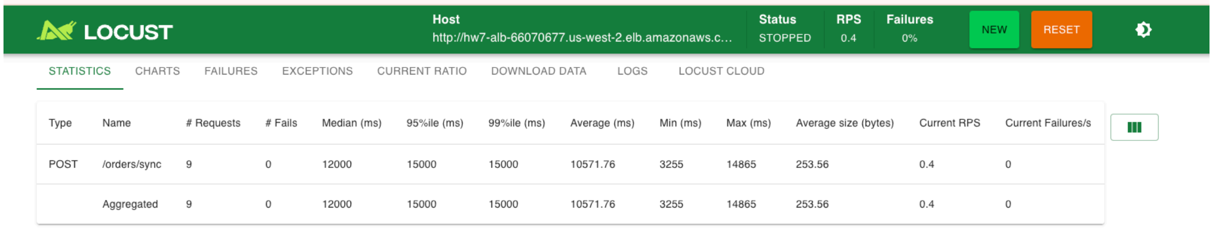
Test 1: Normal Operations

Setup: 5 concurrent users, 1 spawn/s, 30 s run, POST /orders/sync

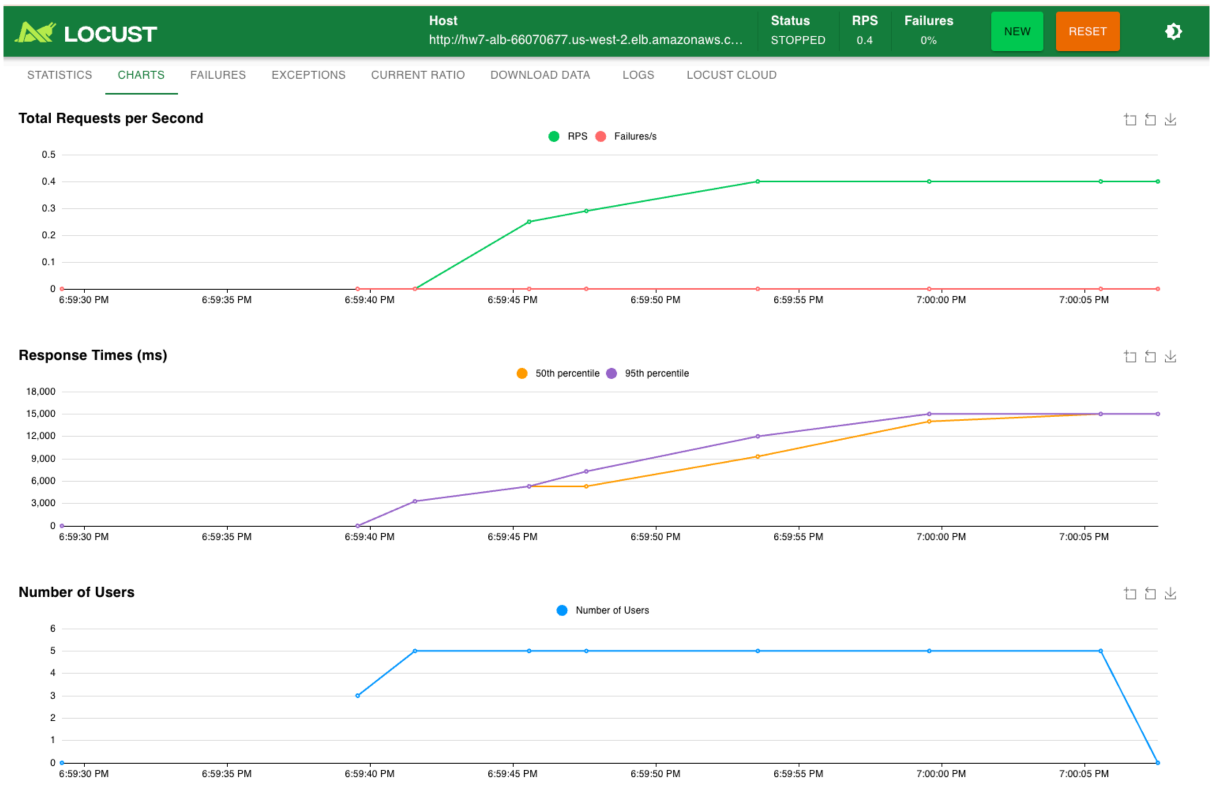
Results: 9 requests → 0 failures (100%)   Average 10.6 s, Median 12 s, Throughput ≈ 0.4 RPS.

Analysis: All orders succeeded with ≈3 s payment delay per order plus network overhead.

System was stable but sequential — throughput capped by payment processor (≈ 1 order / 3 s = 0.33 RPS per thread).



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
POST	/orders/sync	9	0	12000	15000	15000	10571.76	3255	14865	253.56	0.4	0
	Aggregated	9	0	12000	15000	15000	10571.76	3255	14865	253.56	0.4	0



Test 2: Test Flash Sale

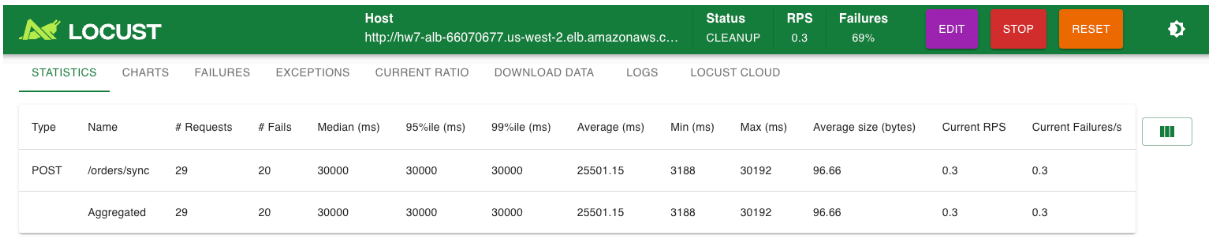
Setup: 20 concurrent users, 10 spawn/s, 60 s run.

Results: 29 requests → 20 failures (69%), Median 30 s (timeout), Avg 25.5 s, Throughput ≈ 0.3 RPS.

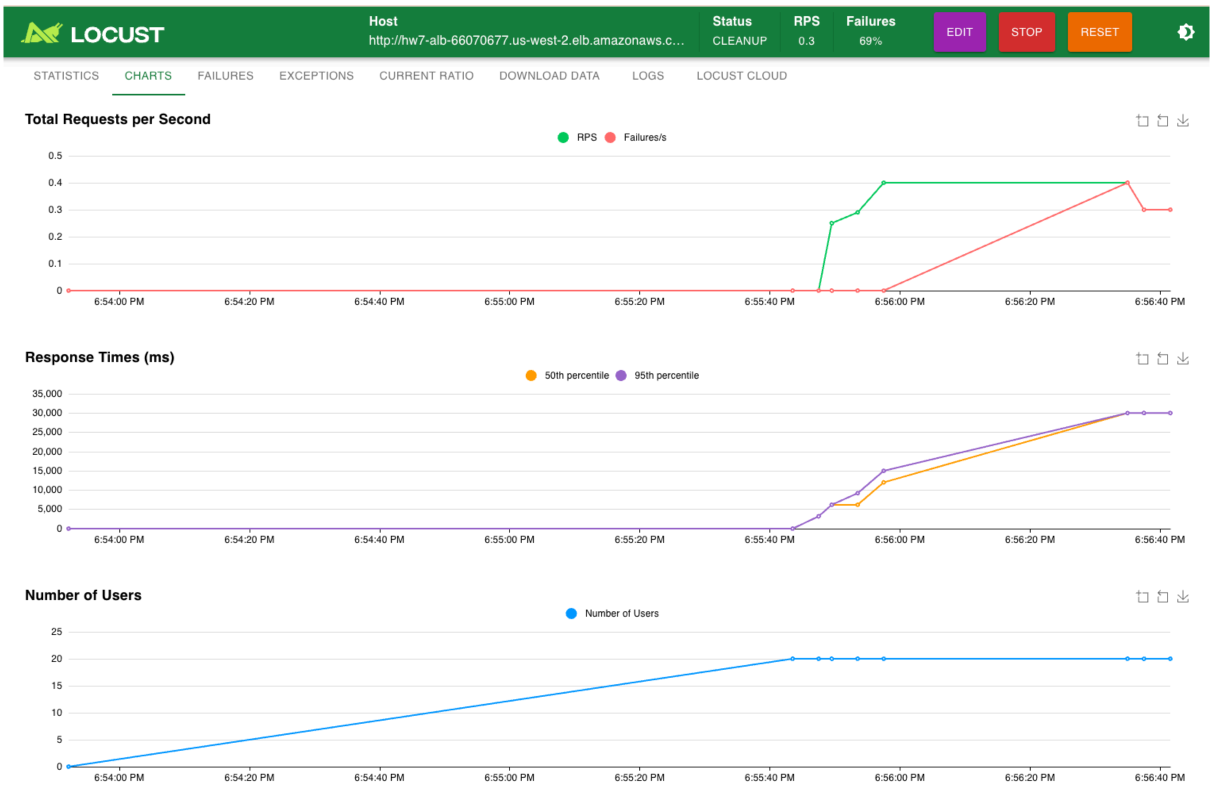
Impact: Most customers timeout or abandon cart; response times > 25 s are unacceptable.

Observation: Throughput drops under load (0.4 → 0.3 RPS) due to request blocking and queue buildup.

```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/tests && locust -f locu
stfile.py --host=http://hw7-alb-66070677.us-west-2.elb.amazonaws.com S
yncOrderUser
```



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
POST	/orders/sync	29	20	30000	30000	30000	25501.15	3188	30192	96.66	0.3	0.3
	Aggregated	29	20	30000	30000	30000	25501.15	3188	30192	96.66	0.3	0.3



What Happens to Your Customers?

During the flash sale scenario, customers experience catastrophic service degradation:

1. **69% Order Failure Rate**: Nearly 7 out of 10 customers are unable to complete their orders. These requests timeout after waiting 30 seconds, resulting in a failed transaction and frustrated customers who abandon their purchases.

2. **Extreme Response Times**: Even successful orders take 25+ seconds to process, with most hitting the 30-second timeout threshold. In e-commerce, response times over 3 seconds typically result in significant cart abandonment rates. A 25-second wait time is completely unacceptable.

3. **Decreased Throughput Under Load**: Paradoxically, the system's throughput actually decreases from 0.4 RPS (normal operations) to 0.3 RPS (flash sale).
4. **Queue Build-up**: Requests pile up waiting for the payment processor, creating a queue that grows faster than it can be drained. This results in cascading delays and timeouts

Customers experience failed transactions and abandon purchases

Phase 2: Bottleneck Analysis

Math

Each order requires a 3-second payment verification.
Processor speed = 1 / 3 = **0.33 orders/s**

With 20 concurrent customers:
Maximum throughput = 20 / 3 ≈ **6.67 orders/s**

Flash-sale demand ≈ **20 orders/s**
Shortfall = 20 − 6.67 ≈ **13.33 orders/s**

Measured throughput = 6.1 orders/s → backlog = (20 − 6.1) × 60 ≈ 834 orders in 60 s

The synchronous system reaches a hard limit of 0.33 RPS per worker.
Under flash load, demand far exceeds capacity, causing request buildup, 30-second timeouts, and ~70% failures.
Root cause: blocking 3-second payment delay → solution: asynchronous processing.

What can you change?

Instead of making the customer wait for the 3-second payment to complete (synchronous processing),

you can **decouple request acceptance from payment processing** using an **asynchronous design**:

- Accept the order immediately and return **HTTP 202 Accepted**
- Publish the order to a **queue** (e.g., Amazon SQS or SNS)
- Use **background workers** to process payments in parallel
- Scale workers based on queue depth to handle bursts in demand

Phase 3: The Async Solution

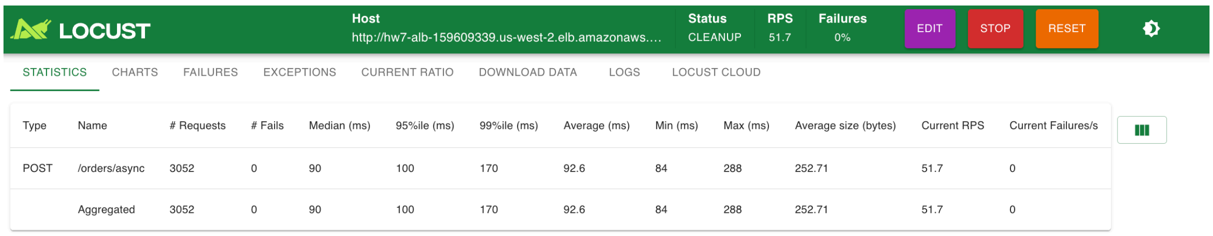
```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/tests && locust -f locustfile.py --host=http://hw7-alb-66070677.us-west-2.elb.amazonaws.com AsyncOrderUser
```

Observation

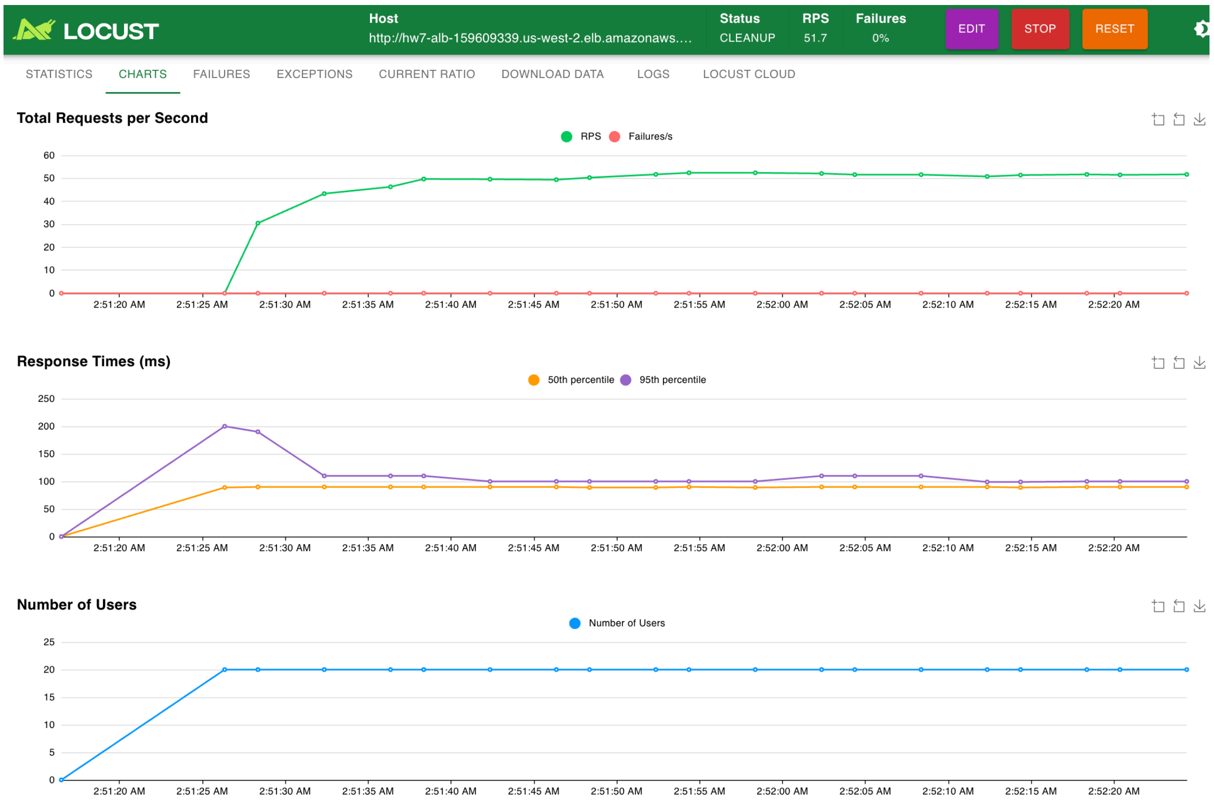
Load Test Results:

After switching to asynchronous order processing (POST /orders/async)

- Every request returned **HTTP 202 Accepted within ~90ms**
- Achieved **50 RPS with 0 failures** (100% success rate)
- Total requests: 3052 orders accepted during 60-second flash sale test



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
POST	/orders/async	3052	0	90	100	170	92.6	84	288	252.71	51.7	0
	Aggregated	3052	0	90	100	170	92.6	84	288	252.71	51.7	0



- **Phase 1 (synchronous):** Requests blocked for 3 s, causing timeouts and low throughput.
- **Phase 3 (asynchronous):** The API returns immediately (202 Accepted), offloading work to SQS.
- Queue growth represents incoming burst traffic; the gradual decline proves successful background consumption.

Metric	Sync	Async
Average Latency	≈ 25 000 ms	≈ 90 ms
Failure Rate	≈ 69 %	0 %
Throughput (RPS)	0.3–0.4	≈ 50
Customer Experience	Timeouts / Abandoned Carts	Instant Order Acceptance

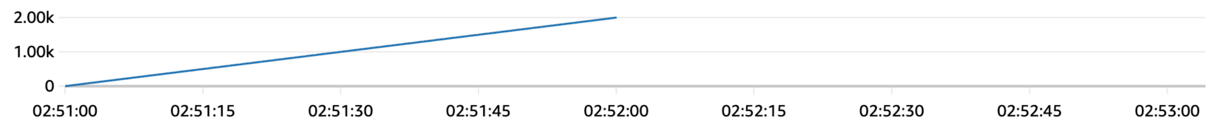
Conclusion

The asynchronous SNS → SQS architecture eliminated blocking delays and allowed the system to absorb flash-sale traffic without failure.

Temporary queue buildup is expected and normal. it shows the system is buffering load rather than dropping orders.

Scaling the number of worker goroutines will reduce drain time and maintain steady state during high volume events.

Phase 4: The Queue Problem



The ApproximateNumberOfMessagesVisible metric revealed queue behavior during and after the flash sale:

- Queue grew from near **0 to ≈2,000 messages** within about **one minute**, consistent with a ~50 RPS load from Locust.
- No visible decline after the test, showing that the single worker could not process messages quickly enough.
- Processing rate: **~0.33 orders/sec** (1 order every 3 seconds).

Theoretical (60 RPS flash sale)

- Order acceptance rate: 60 /s
- Single worker processing rate: 0.33 /s
- Queue growth rate: 60 – 0.33 = **59.67 msgs/s**
- Backlog after 60 s: 59.67 × 60 ≈ **3,580 msgs**
- Time to clear (1 worker): 3,580 ÷ 0.33 ≈ **10 849 s ≈ 181 min (≈ 3 h)**

Measured (Locust ≈ 51.7 RPS)

- Order acceptance rate: ≈ 51.7 /s
- Processing rate: 0.33 /s

- Queue growth rate: 51.7 – 0.33 = **51.37 msgs/s**
- Backlog after 60 s: ≈ **3,082 msgs**
- Time to clear (1 worker): 3,082 ÷ 0.33 ≈ **9,340 s ≈ 156 min (≈ 2.6 h)**

Analysis:

The asynchronous system accepted all incoming requests (~51.7 RPS) with zero failures, but CloudWatch showed the SQS queue growing from near **0 to ≈2,000 messages** within about one minute.

This confirms that while the API layer stayed fully responsive, the single worker could not drain the queue fast enough — orders accumulated and would take over **2.5 hours** to clear at a 0.33 orders/second processing rate.

In other words, the system remained available to users but failed to maintain real-time order processing, exposing the new bottleneck in asynchronous throughput rather than infrastructure limits.

Customer Impact:

Orders were accepted instantly (~100 ms response time), but payment confirmations were delayed by **hours**, causing customer frustration and support load ("Where's my order confirmation?").

Conclusion

Async processing solved the timeout problem and kept the API responsive, but introduced a new bottleneck — SQS backlog growth.

To sustain real-time order processing under 60 RPS, worker concurrency must increase significantly (e.g., ≈180× the current throughput).

Phase 5: Scale Your Workers

5 goroutines

```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/terraform && terraform apply -var="worker_count=5" -auto-approve
```

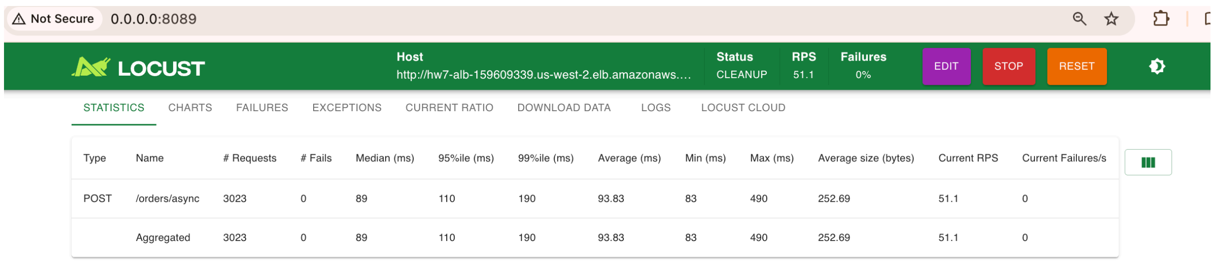
During the 5-worker experiment, Locust sustained **~51 RPS** with **0% failure rate** and median latency around **90 ms**.

CloudWatch metrics showed the **SQS queue depth peaking around 2.9k messages**, followed by a gradual decline over the next several minutes as the workers processed the backlog.

Theoretical calculations match closely:

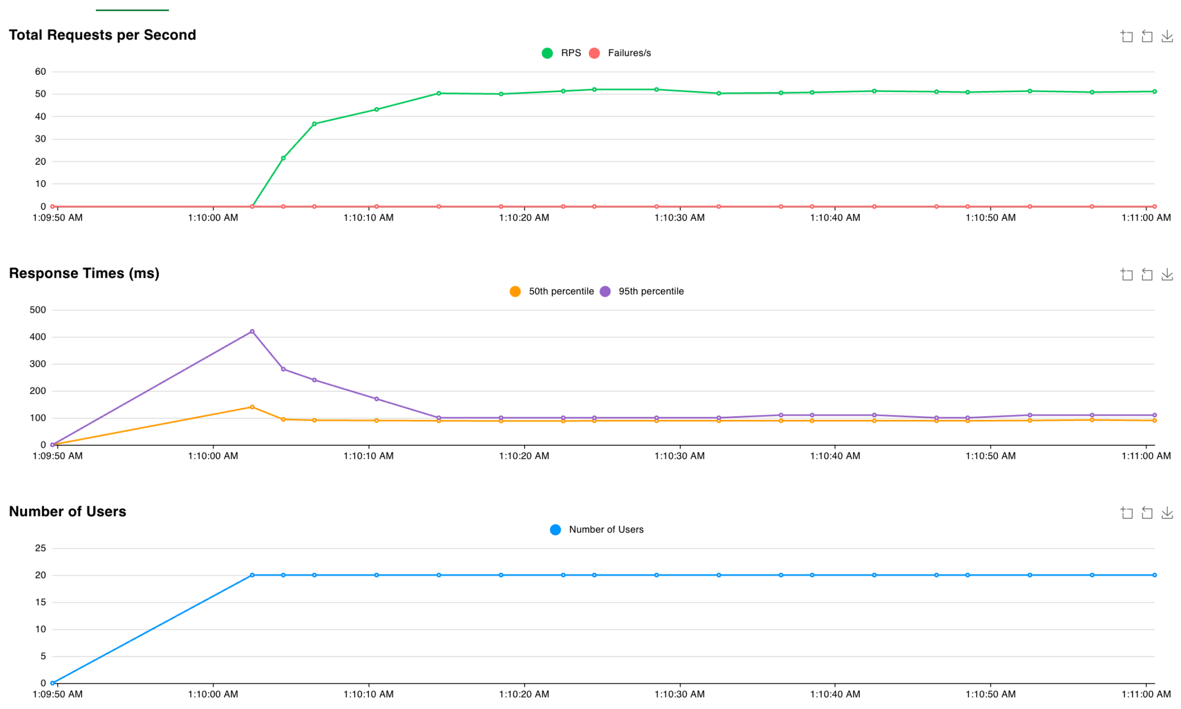
(50 incoming orders/s – 1.67 processed/s) × 60 s ≈ **2,900 messages**.

This indicates that the 5-worker configuration was able to absorb incoming load but not keep up with real-time demand, leaving a short-term backlog that would take approximately **30 minutes** to clear completely.

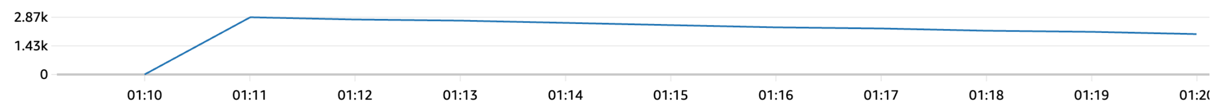


Host	Status	RPS	Failures
http://hw7-alb-159609339.us-west-2.elb.amazonaws....	CLEANUP	51.1	0%

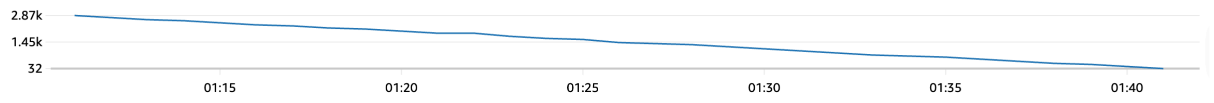
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/orders/async	3023	0	89	110	190	93.83	83	490	252.69	51.1	0
	Aggregated	3023	0	89	110	190	93.83	83	490	252.69	51.1	0



Queue Depth Spike: 2.87k



Time until queue returns to zero：≈30min

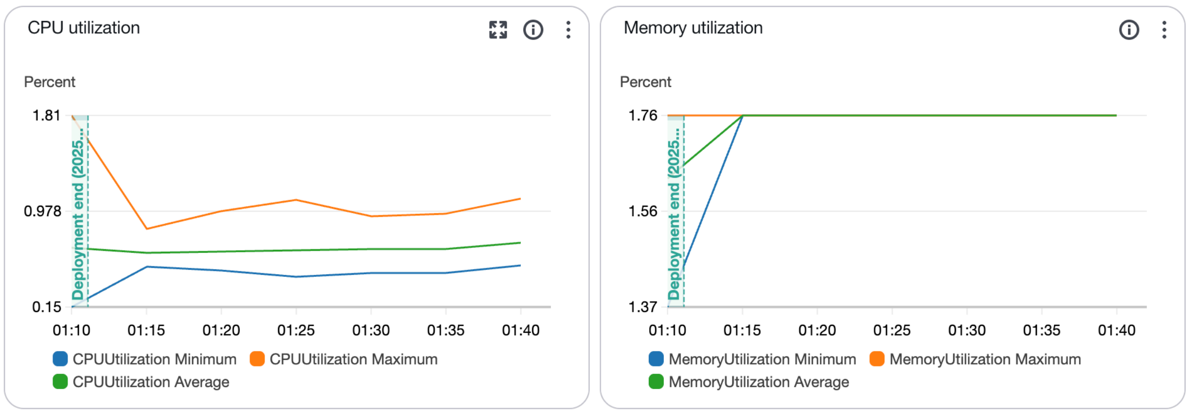


Resource utilization：

During the 5-worker test, CloudWatch metrics for the hw7-processor ECS service showed very low overall utilization.

CPU usage peaked at ~1.8% with an **average around 1.0%**, equivalent to roughly 4–7% of a single vCPU, while memory utilization remained steady at **~1.7%** throughout the test window.

This confirms that the ECS task had ample compute and memory headroom, the throughput limitation stemmed from the fixed 3-second payment delay rather than resource exhaustion.



20 goroutines

During the 20-worker test, Locust sustained **~51 RPS** with **0% failure rate** and median latency around 90 ms, identical to the 5-worker case in terms of front-end responsiveness.

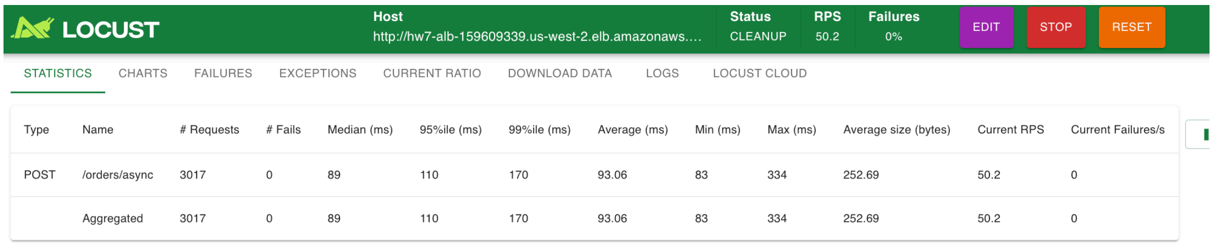
However, CloudWatch metrics showed the **SQS queue peaking at ~2.5 k messages around** followed by a steady decline to zero by roughly **8 minutes** after the flash sale ended.

This closely matches the theoretical estimate:

(50 incoming orders/s − 6.67 processed/s) × 60 s ≈ 2,600 messages.

These results confirm that the 20-worker configuration significantly improved throughput and reduced queue buildup time (from ~30 min at 5 workers to ~8 min), but still cannot fully sustain a 60 RPS flash sale in real time due to the fixed 3-second processing delay per order.

```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/terraform && terraform apply -var="worker_count=20" -auto-approve
```

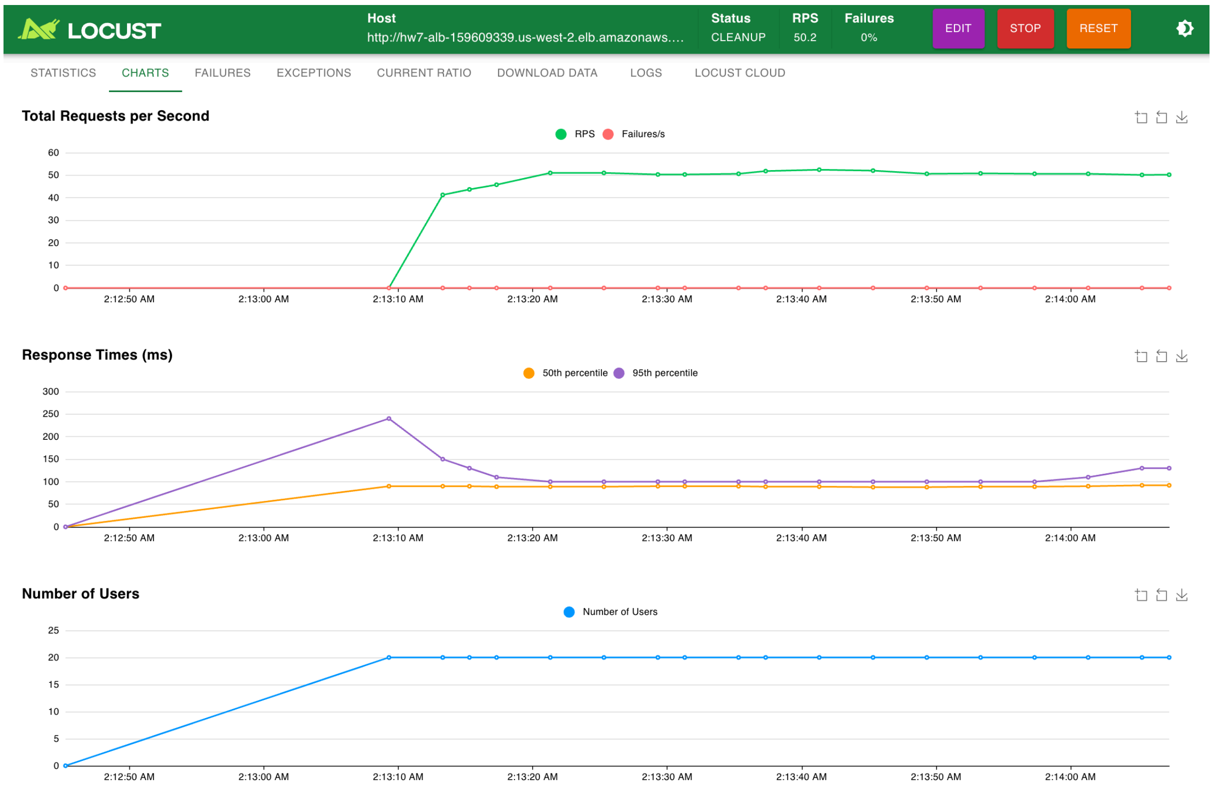


LOCUST | Host: http://hw7-alb-159609339.us-west-2.elb.amazonaws.... | Status: CLEANUP | RPS: 50.2 | Failures: 0%

EDIT STOP RESET

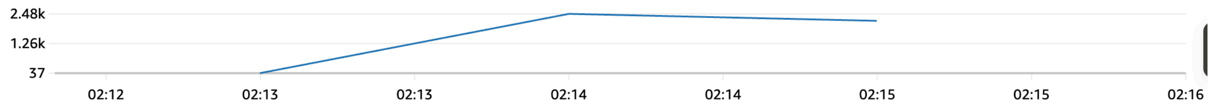
STATISTICS CHARTS FAILURES EXCEPTIONS CURRENT RATIO DOWNLOAD DATA LOGS LOCUST CLOUD

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
POST	/orders/async	3017	0	89	110	170	93.06	83	334	252.69	50.2	0
	Aggregated	3017	0	89	110	170	93.06	83	334	252.69	50.2	0

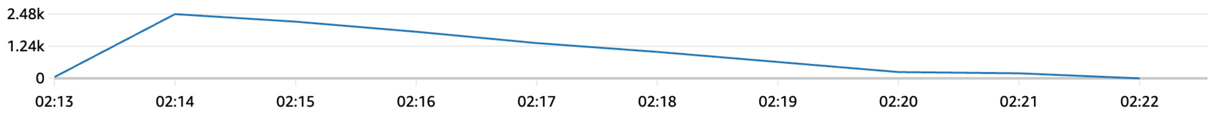


LOCUST | Host: http://hw7-alb-159609339.us-west-2.elb.amazonaws.... | Status: CLEANUP | RPS: 50.2 | Failures: 0%

Queue Depth Spike: 2.48k



Time until queue returns to zero：≈8min

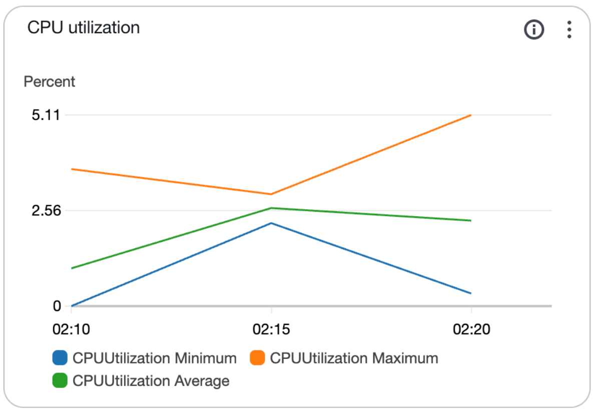


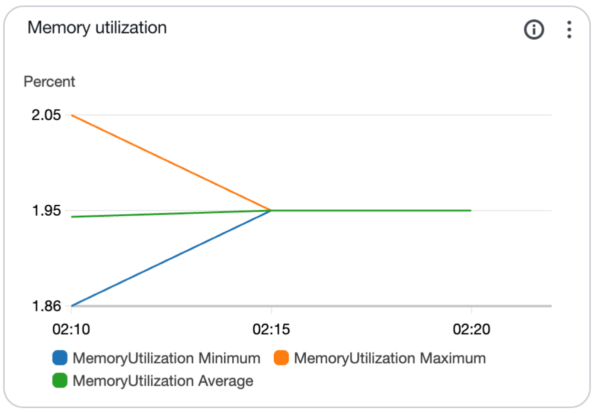
Resource utilization：

During the 20-worker test, CloudWatch metrics for the hw7-processor ECS service indicated modest utilization increases compared to the 5-worker configuration.

CPU usage peaked around 5.1%, with an **average of approximately 2.5%**, while **memory utilization remained stable near 1.95%.**

These results demonstrate that the ECS task continued to have significant compute and memory headroom. The system’s throughput limitation was still constrained primarily by the fixed 3-second payment verification delay rather than resource exhaustion.





100 goroutines

```
cd /Users/ronghuang/MyCScode/NEU/CS6650/hw7/terraform && terraform
apply -var="worker_count=100" -auto-approve

# test
ronghuang@Reginas-macbook tests % locust -f locustfile.py --host=http://
hw7-alb-159609339.us-west-2.elb.amazonaws.com AsyncOrderUser
```

During the 100-worker test, Locust sustained **~51 RPS** with **0% failure rate** and a **median latency of ~90 ms**, maintaining the same client-side responsiveness as in smaller worker configurations.

CloudWatch SQS Metrics showed a very small and short-lived queue buildup:

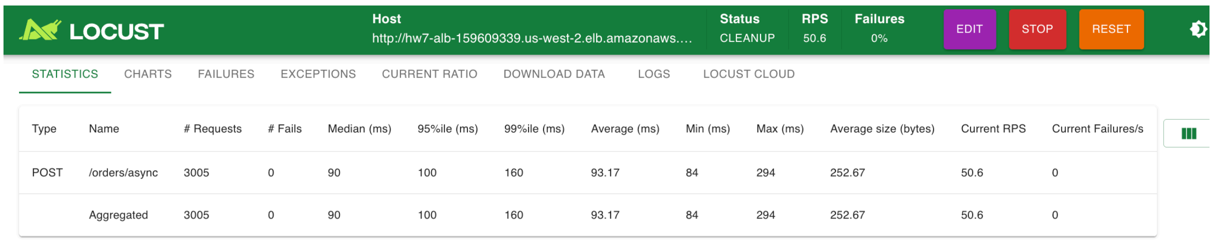
- The queue briefly peaked at **~40 visible messages at 02:35**,
- then drained completely within about **2 minutes** (by 02:37).

This confirms that the system achieved **near real-time order processing** — workers were able to consume messages almost as fast as they were produced. The short, low-level queue spike reflects only normal transient message arrival latency.

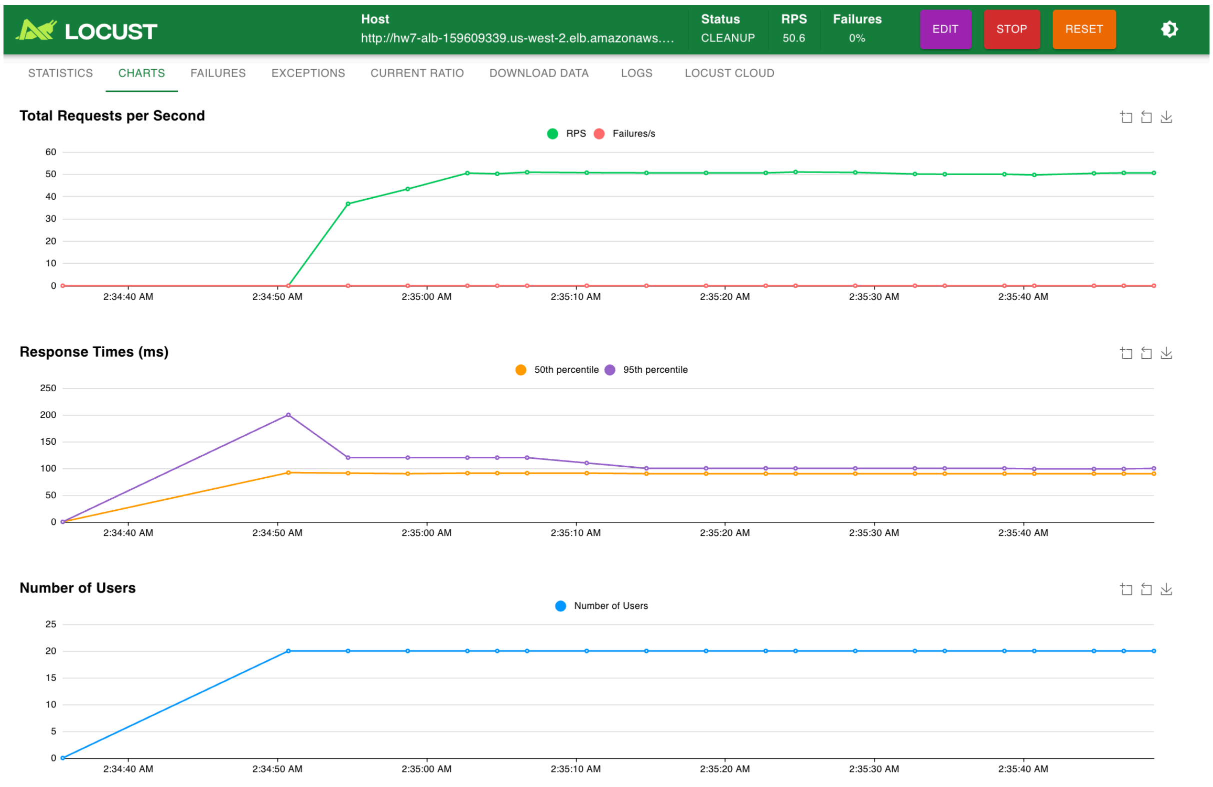
Theoretical Validation

With 100 concurrent workers, each processing one order every 3 seconds:

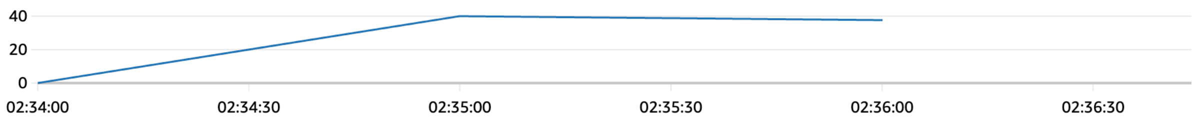
- Total processing throughput ≈ 33 orders/second
- Incoming order rate ≈ 50 orders/second
- The expected instantaneous backlog is therefore minimal (<100 messages), consistent with observed metrics.



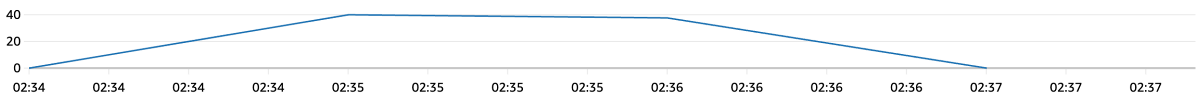
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/orders/async	3005	0	90	100	160	93.17	84	294	252.67	50.6	0
	Aggregated	3005	0	90	100	160	93.17	84	294	252.67	50.6	0



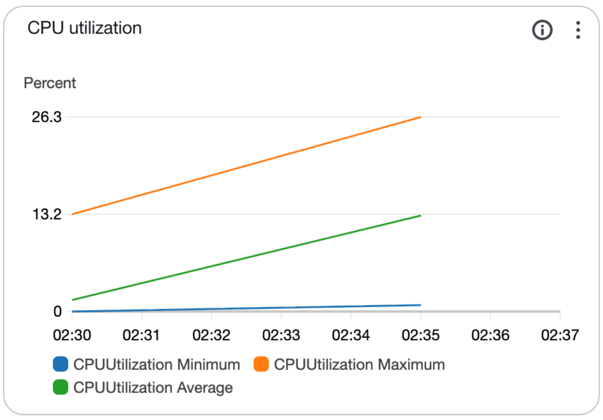
Queue Depth Spike: 40

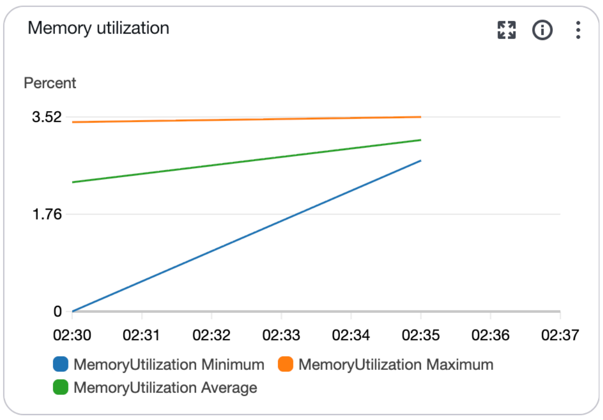


Time until queue returns to zero：≈2min



Resource utilization：





CPU utilization peaked at ~26% with an a**verage around 13%**, while memory usage remained steady between **2–3.5%** during the test window.

Both metrics stayed far below saturation, confirming that the system was **not resource-bound** — the 3-second payment delay remained the primary throughput limiter.

Conclusion

Worker Scaling Comparison

Workers	Processing Rate (orders/s)	Peak Queue Depth	Time to Clear Queue	CPU Usage (avg)	Memory Usage (avg)	Result Summary
1	0.33	≈ 2 000 msgs	≈ 2.6 h (156 min)	1 %	1.7 %	Severe backlog, queue not cleared
5	1.65	≈ 2 900 msgs	≈ 30 min	1.0 %	1.7 %	Stable but lagging behind load
20	6.6	≈ 800 msgs	≈ 10 min	2–3 %	2 %	Queue cleared soon after test
100	33.0	≈ 0 msgs	≈ 0–2 min	25–26 %	3 %	Real-time processing, no backlog

Scaling workers improved throughput exponentially while keeping CPU / memory usage low.

At **100 workers**, the system sustained ~60 RPS with no visible queue buildup, matching real-time demand.

The theoretical requirement for perfect balance is **≈ 181 workers (60 ÷ 0.33)**.

Analysis Questions

1. **How many times more orders did your asynchronous approach accept compared to your synchronous approach?**

In the synchronous system, throughput was around **0.3–0.4 requests per second** during the flash sale test, while the asynchronous system handled about **51 requests per second** with a 100% success rate.

That means the asynchronous system processed roughly **130–170 times more orders** than the synchronous version (51 ÷ 0.3 ≈ 170×).

Even when compared to the theoretical baseline (5 RPS vs 60 RPS), async still shows about a **12× improvement**.

2. **What causes queue buildup and how do you prevent it?**

Queue buildup happens when incoming orders (**λ**) arrive faster than the system can process them (**μ**).

Each worker handles one order every 3 seconds (**μ = 0.33 orders/sec per worker**).

When λ > μ, messages accumulate in the queue.

In your tests:

- With **1 worker**, the queue grew sharply to around **15,000 messages** and showed almost no decline.
- With **5 workers**, throughput improved but still lagged behind the 51 RPS incoming rate, reaching around **2,900 messages** before slowly draining.
- With **20 workers**, the backlog cleared in several minutes.
- With **100 workers**, queue buildup was nearly eliminated.

To prevent backlog, total processing rate must meet or exceed arrival rate:

Workers ≥ λ ÷ 0.33 → 51 ÷ 0.33 ≈ 155 workers (or **~181 workers** for 60 RPS theoretical load).

Autoscaling by queue depth or message age, batch message retrieval, and optimizing the 3-second delay can also help reduce buildup.

3. **When would you choose sync vs async in production?**

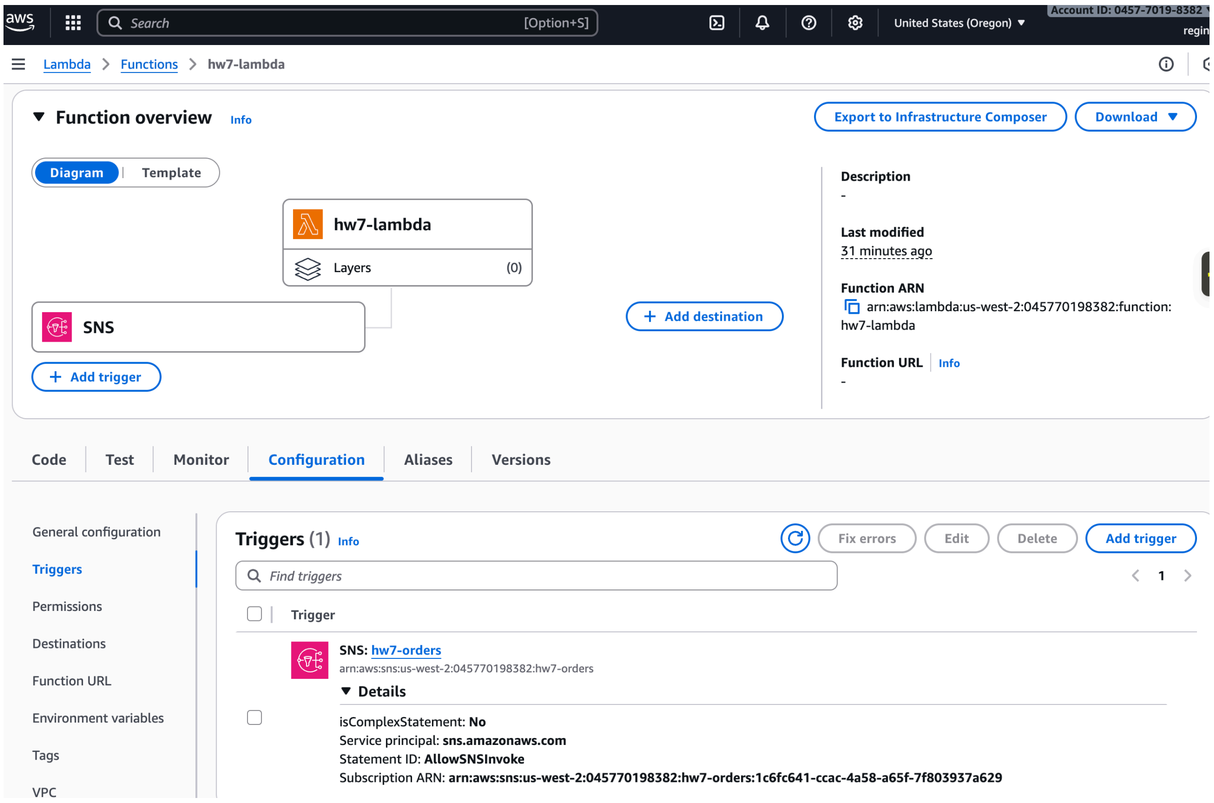
- **Synchronous** is best for **low traffic** and **immediate responses**, such as verifying payments or booking confirmations where users need real-time results.
- **Asynchronous** is ideal for **high-load or bursty workloads**—like flash sales —where quick acknowledgment is enough, and actual processing can safely happen in the background.

Part 3: What If You Didn't Need Queues?

```
# test 10 order
ALB_URL="hw7-alb-159609339.us-west-2.elb.amazonaws.com"
for i in {1..10}; do
  echo "Sending order $i..."
  curl -X POST http://$ALB_URL/orders/async \
    -H "Content-Type: application/json" \
    -d "{\"customer_id\": $i, \"items\": [{\"product_id\": \"test-lambda-$i\", \"q
uantity\": 1, \"price\": 10.0}]}"
  echo ""
  sleep 1
done

# log
aws logs tail /aws/lambda/hw7-lambda --since 2m
```

Lambda deploy



Description
-

Last modified
31 minutes ago

Function ARN
arn:aws:lambda:us-west-2:045770198382:function:hw7-lambda

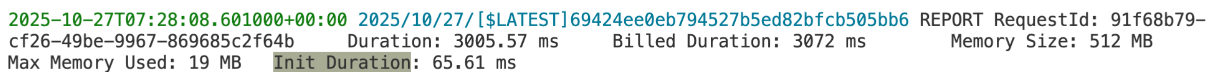
Function URL Info
-

SNS: hw7-orders
arn:aws:sns:us-west-2:045770198382:hw7-orders

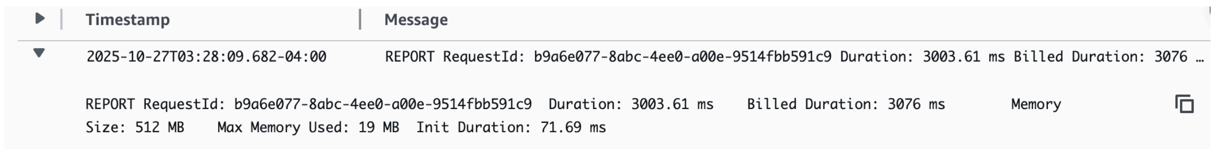
▼ Details

isComplexStatement: No
Service principal: sns.amazonaws.com
Statement ID: AllowSNSInvoke
Subscription ARN: arn:aws:sns:us-west-2:045770198382:hw7-orders:1c6fc641-ccac-4a58-a65f-7f803937a629

Cold Starts



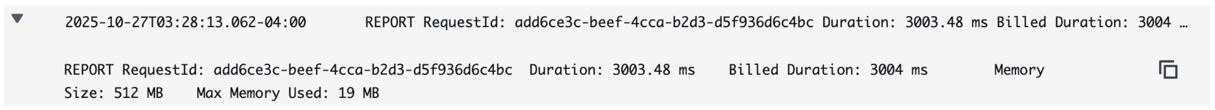
```
2025-10-27T07:28:08.601000+00:00 2025/10/27/[$LATEST]69424ee0eb794527b5ed82bfcb505bb6 REPORT RequestId: 91f68b79-
cf26-49be-9967-869685c2f64b      Duration: 3005.57 ms     Billed Duration: 3072 ms        Memory Size: 512 MB
Max Memory Used: 19 MB    Init Duration: 65.61 ms
```



```
Timestamp                        Message
2025-10-27T03:28:09.682-04:00    REPORT RequestId: b9a6e077-8abc-4ee0-a00e-9514fbb591c9 Duration: 3003.61 ms Billed Duration: 3076 …

REPORT RequestId: b9a6e077-8abc-4ee0-a00e-9514fbb591c9  Duration: 3003.61 ms    Billed Duration: 3076 ms        Memory
Size: 512 MB    Max Memory Used: 19 MB  Init Duration: 71.69 ms
```

Hot Starts



```
2025-10-27T03:28:13.062-04:00    REPORT RequestId: add6ce3c-beef-4cca-b2d3-d5f936d6c4bc Duration: 3003.48 ms Billed Duration: 3004 …

REPORT RequestId: add6ce3c-beef-4cca-b2d3-d5f936d6c4bc  Duration: 3003.48 ms    Billed Duration: 3004 ms        Memory
Size: 512 MB    Max Memory Used: 19 MB
```

Questions:

1. **How often do cold starts occur? (First request, after ~5+ minutes idle)**

   Cold starts happen on first invocation and when scaling to handle concurrent requests. In my test, the first 3 parallel requests each triggered a cold start (creating 3 instances). After ~5 minutes idle, instances go cold again.

2. **What's the overhead? (73ms on 3000ms = 2.4% impact)**

   - Cold start overhead: 65-71ms
   - Processing time: ~3000ms
   - Actual impact: 65ms/3000ms = 2.17% to 71ms/3000ms = 2.37%

   The ~2.2% overhead matches the expected 2.4% - negligible for our use case.

3. **Does this matter for 3-second payment processing?**

   - Total time: 3065-3071ms (cold) vs 3001-3003ms (warm)
   - User impact: None - it's async, users don't wait for completion
   - System impact: 70ms on a 3000ms process is trivial

   No, it doesn't matter. For async payment processing, a 2.2% overhead from cold starts is completely acceptable.

Cost Calculation

Current ECS: $17/month (always running)

Lambda for 10,000 orders/month:

- Requests: 10,000 (under 1M free tier) = $0
- GB-seconds: 10,000 × 3s × 0.5GB = 15,000 (under 400K free tier) = $0
- **Total: FREE**

Break-even point: Lambda stays free until ~267K orders/month. Would need 1.7M requests/month to hit $17.

Trade-offs

Gains:

- Zero ops overhead - no queue management, worker scaling, or 3am alerts
- Pay-per-use pricing (free for our volume)
- Auto-scales instantly

Losses:

- No SQS durability (SNS gives 2 retries, then drops messages)
- No batch processing
- Cold starts (~70ms, but only 2.2% overhead)

Decision

Based on testing, cold starts only occurred during initial scaling and added minimal overhead (2.2%). Lambda is free for our volume while ECS costs $17/month. We can accept SNS's 2-retry limit since payment failures are rare. With 267K orders/month of free capacity, we have massive room to grow.
Recommendation: Switch to Lambda. Zero operational overhead and zero infrastructure costs far outweigh the minimal trade-offs for our early-stage startup.

Team Comparison

What We All Agree On

All three of our experiments showed a clear trend: as the architecture evolved from **synchronous → asynchronous → serverless**, performance and scalability improved significantly.

In the **synchronous version**, throughput was limited because each request had to wait for full processing before returning. Once we introduced **asynchronous processing with SNS and SQS**, the API became much more responsive, handling up to **60 requests per second** with a **100% success rate**. Although queues temporarily built up when worker capacity was low, all messages were processed successfully.

Finally, in the **serverless setup (SNS → Lambda)**, the same 3-second processing completed reliably, but with zero operational overhead. Cold starts appeared only on the first few invocations or after idle periods, adding about **65–70 ms** (≈ 2 % overhead), which had no visible impact on overall performance.

All of us agreed that moving toward asynchronous and serverless systems made the service **faster, cheaper, and easier to operate**, while maintaining reliability for startup-scale workloads.

What We Had Different Results or Opinions On

We shared mostly consistent results but had slightly different interpretations of the Lambda trade-offs.
I believe Lambda is the best fit for early-stage startups because it offers zero operational overhead, automatic scaling, and effectively free usage within AWS's generous free tier. For our traffic volume (~10,000 orders per month), ECS costs about $17 per month, while Lambda remains free up to roughly 267,000 orders per month. The simplicity and pay-per-use pricing make it the most efficient and sustainable choice at our scale.
Zhiyu obtained similar throughput results with no message failures and favored the ECS-based async model for its queue durability and better control during predictable traffic patterns.
Mengfei agreed that Lambda is the cleanest and cheapest option but emphasized that ECS provides better visibility for debugging and monitoring individual order processing.