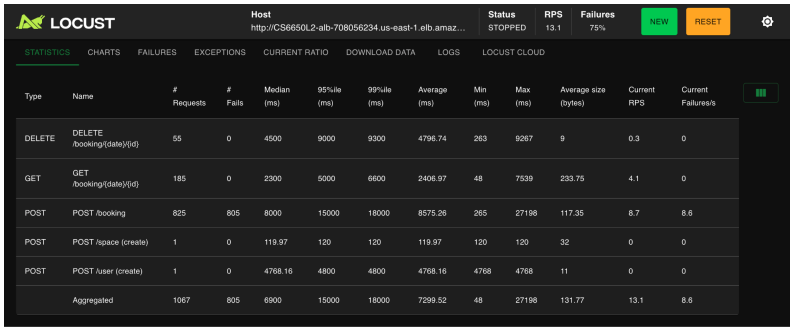
EXPERIMENT SUMMARY

(Comprehensive Results Found Here)

AVAILABILITY

Check Error Rate As Spaces Are Ramped Up

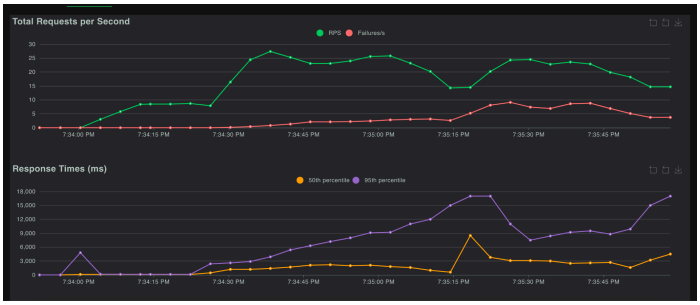


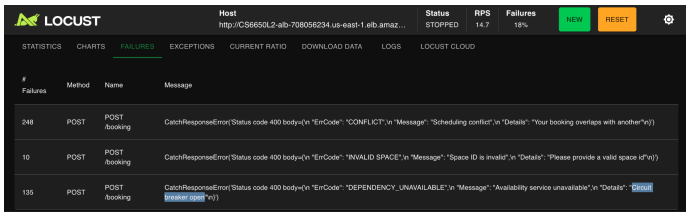
Least Number of Spaces

Here, a single shared user is created once, followed by 1-200 distinct spaces. Each load-test user then repeatedly creates bookings by choosing a random space and a random 1–2 hour time window between 08:00 and 22:00 on the same day. The load shape ramps from 0 to 100 users.

Lower to Moderate Spaces

At lower to moderate load, the system successfully handles user and space creation, and POST /booking returns a mix of successful responses and HTTP 400 responses with an application-level CONFLICT error. These 400 responses indicate that the booking service is correctly enforcing its business rule that overlapping bookings for the same space and time range are rejected. As we increase the number of spaces, these conflicts become less frequent as there is less competition.





Most Number of Spaces

At higher levels of concurrency, the system begins to exhibit clear signs of saturation. Latencies increased, and interestingly, this triggered our circuit breaker to open. While this test was not intended to test the circuit breaker, it was interesting to see it in action. Many POST requests for bookings return 400 and display the error message “circuit breaker open”, which is expected because we see there is a latency spike in the chart. Overall, the ramp-users test shows both correct conflict detection under contention and a clear capacity limit where the service degrades with increasing concurrency.

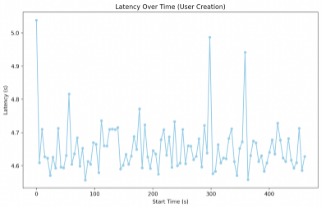
We also ramp up users towards 1000, resulting in 5xx errors as the ALB gets overwhelmed with requests. See more screenshots here: throughput locust screenshots.

To the left, we have a summary of circuit breaker responses. In the first request, availability gives us a 200 response, but it is very slow (15s). We simulate a slow response here because the booking service calls the availability service for each booking. In our circuit breaker pattern, high latency opens the circuit breaker and causes us to time out and throw a 400 with an informative error message.

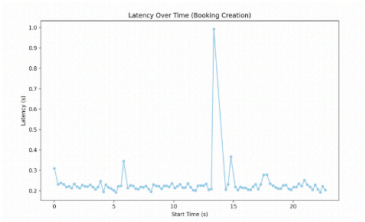
Check How Latency Changes with Dynamo DB Implementation

In this scenario, we send 100 requests to the specified WRITE endpoint, and track the amount of time each response takes. Our goal here is to measure our most time complex requests.

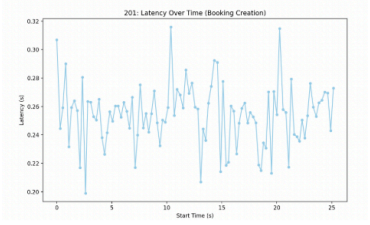
User creation uniformly had very high latency, likely due to the time-expensive password hashing operation. However, we found no significant difference between latency for an internal map versus Dynamo DB operations, with mean latency staying between 4.6s and 4.65s. The space service was similarly very stable.



Dynamo DB User Creation



Dynamo DB - 1 Replica - Eventual Consistency

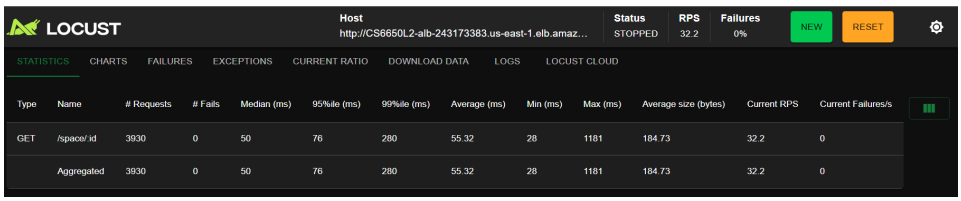


Dynamo DB - 1 Replica - Strong Consistency

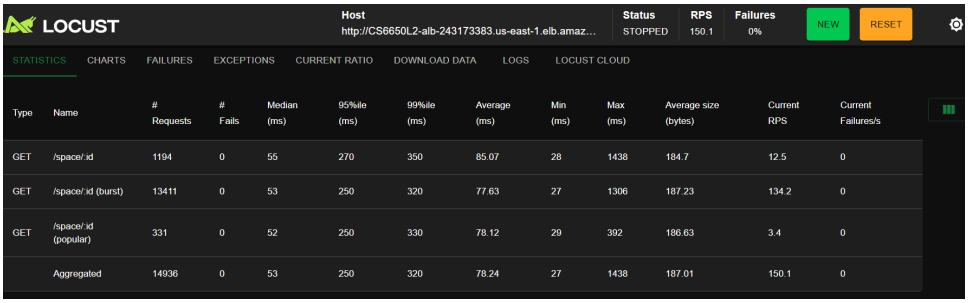
Finally, the booking service was also tested using DynamoDB with and without a replica table. Surprisingly, adding a replica only added a slight increase in the standard deviation of results without changing the mean or median computation time. This indicates that AWS may be using a warm start to prevent replica database writes from being expensive. Interestingly, we can also see the difference in latency when we used a strong versus eventual consistency replica. While the average latency of both graphs are similar, the lack of one central outlier in the strong consistency graph indicates that it is overall slower

Check How Redis Caching Impacting Availability

We implemented Redis cache through ElastiCache for the spaces service to improve performance by reducing repeated reads from the database. This showed a significant improvement in response times under load, demonstrating the effectiveness of caching for frequently accessed data.



No Cache



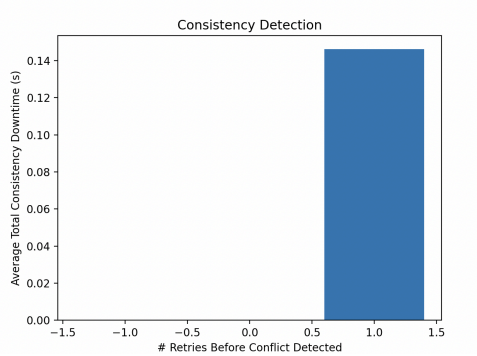
Cache

CONSISTENCY

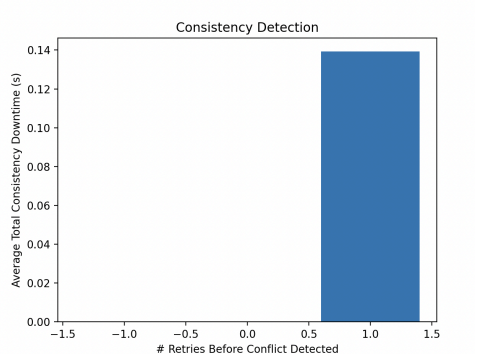
Check How Consistency Changes with DB Implementation

To test consistency of Dynamo DB, we tested the process of creating rooms and then intentionally triggering booking conflicts. Our **purpose** was to retry booking already-booked rooms until a conflict arose. Thus, we would be able to detect whether there are any delays between getting a 201 response that a booking was created, and actually creating the booking in all database replicas.

Our findings showed virtually no difference in potential delays between Dynamo DB with 0 and 1 replica. The graph below shows that for all *100* bookings made, with and without database replicas, a conflict was detected on the first attempt to create an overlapping booking. Additionally, the response time is identical.



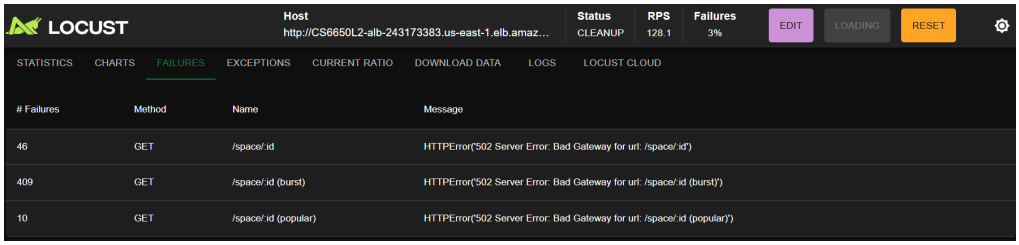
DYNAMO DB - NO REPLICAS



DYNAMO DB - 1 REPLICA

Check How Consistency Changes with Redis Caching

During Redis Caching, we initially observed a few intermittent failures in responses, likely due to cache misses or connection issues. This prompted additional consistency testing.




```
TEST 1: Read-after-write
Created booking: 94577202595
Immediate read: 200 {
    "bookingID": 94577202595,
    "spaceID": "Library-101",
    "date": "2025-12-11",
    "userID": 6036687152,
    "occupants": 2,
    "startTime": "2025-12-11T10:00:00Z",
    "endTime": "2025-12-11T11:00:00Z"
}

TEST 2: Write-after-write
Initial booking: 51052973077
Creating conflicting booking...
Response: 400 {
    "ErrCode": "CONFLICT",
    "Message": "Scheduling conflict",
    "Details": "Your booking overlaps with another"
}
```

```
TEST 3: Cross-user cache invalidation
Alice created: 88810507456
Bob read: 200 {
    "bookingID": 88810507456,
    "spaceID": "Library-103",
    "date": "2025-12-11",
    "userID": 6036687152,
    "occupants": 2,
    "startTime": "2025-12-11T13:00:00Z",
    "endTime": "2025-12-11T14:00:00Z"
}

TEST 4: Stale read detection
Warm cache read: 200
Waiting for cache TTL to expire (5 seconds)...
Post-TTL read: 200
```

These tests verified that cached and database values remained synchronized and that the system returned correct space availability information, ensuring consistency.

FAULT TOLERANCE

To ensure that the corresponding errors were being handled appropriately, we intentionally simulated different faulty loads for each service such as invalid inputs, malformed requests, missing authentication, and simulated backend failures to observe how each service reacted under stress. The system consistently returned structured JSON error messages and appropriate HTTP status codes, demonstrating that services were able to handle unexpected conditions without crashing.

User Service Error responses

```
=== USER SERVICE ERROR TESTS ===

1. Missing password:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERR', 'Message': 'incorrect schema for a new user', 'Details': 'A new user requires a user
name and password'}

2. Wrong password:
   Status: 404, Expected: 404
   ✓ Error: {'ErrCode': 'INVALID', 'Message': 'Password Does Not Match', 'Details': 'crypto/bcrypt: hashedPassword is not
 the hash of the given password'}

3. Non-existent user:
   Status: 404, Expected: 404
   ✓ Error: {'ErrCode': 'NOT_FOUND', 'Message': 'User Not Found', 'Details': 'The provided user id does not exist'}

4. Invalid user ID:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERROR', 'Message': 'Unable to detect a user id', 'Details': 'strconv.Atoi: parsing "notanu
mber": invalid syntax'}
```

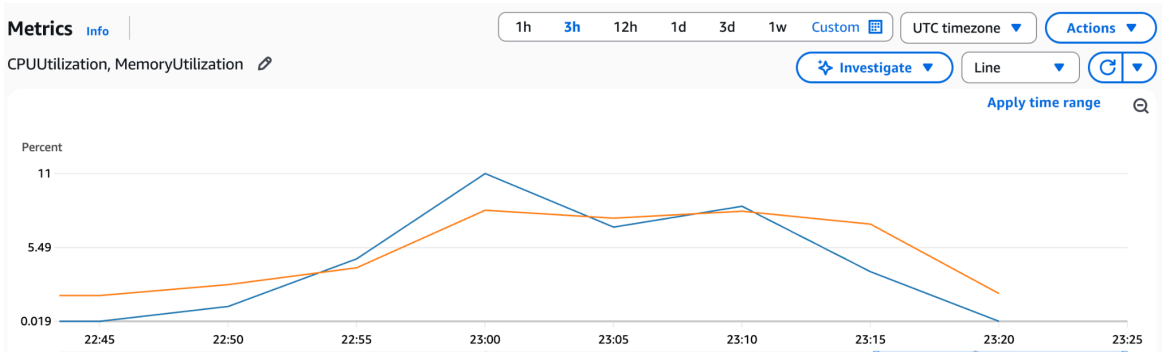
Availability Service

```
=== AVAILABILITY SERVICE ERROR TESTS ===

✓ Alice logged in successfully
✓ Admin logged in successfully

1. Create space without admin (alice):
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'UNAUTHORIZED', 'Message': 'Not an admin', 'Details': 'Only admin users can use this operation'}

2. Missing authentication:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'MALFORMED', 'Message': 'basic auth is missing or malformed', 'Details': 'ensure that a username
and/or password are provided'}

3. Duplicate space (admin auth):
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'DUPLICATE', 'Message': 'Space already exists', 'Details': 'This space already exists'}

4. Get non-existent space:
   Status: 404, Expected: 404
   ✓ Error: {'ErrCode': 'NOT_FOUND', 'Message': 'Space Not Found', 'Details': 'The provided space id does not exist'}

5. Invalid user ID in auth:
   Status: 500, Expected: 400
6. Missing required fields (with admin):
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERR', 'Message': 'incorrect schema for a new space', 'Details': 'A new space requires a bu
ilding and room'}
```

Booking Service

```
✓ Alice logged in successfully

1. Missing required fields:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERR', 'Message': 'incorrect schema for a new booking', 'Details': 'A new space requires a
space, date, user, occupants, start, and end time'}

2. Invalid date format:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERR', 'Message': 'Date must be formatted as YYYY-MM-DD', 'Details': 'parsing time "12-15-2
024" as "2006-01-02": cannot parse "12-15-2024" as "2006"'}

3. User ID mismatch:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INPUT_ERR', 'Message': 'User ID does not match', 'Details': 'Can only create a reservation for y
ourself'}

4. No authentication:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'MALFORMED', 'Message': 'basic auth is missing or malformed', 'Details': 'ensure that a username
and/or password are provided'}

5. Invalid space ID:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INVALID SPACE', 'Message': 'Space ID is invalid', 'Details': 'Please provide a valid space id'}

6. Exceeds space capacity:
   Status: 400, Expected: 400
   ✓ Error: {'ErrCode': 'INVALID', 'Message': 'Too many occupants', 'Details': 'Space capacity exceeded'}
```

Killing ECS Task



The ECS task termination test showcased excellent resilience in our distributed architecture. When we killed an ECS task under load, we observed some expected failures during the initial failover window as our requests failed, but the system quickly stabilized with response times only briefly spiking before returning to normal. CloudWatch metrics showed CPU and memory utilization dropping by 25-30% immediately after task deregistration as the load redistributed among remaining healthy tasks. Within 2 minutes, ECS automatically spawned a replacement task, and once it passed health checks and joined the target group, resource utilization

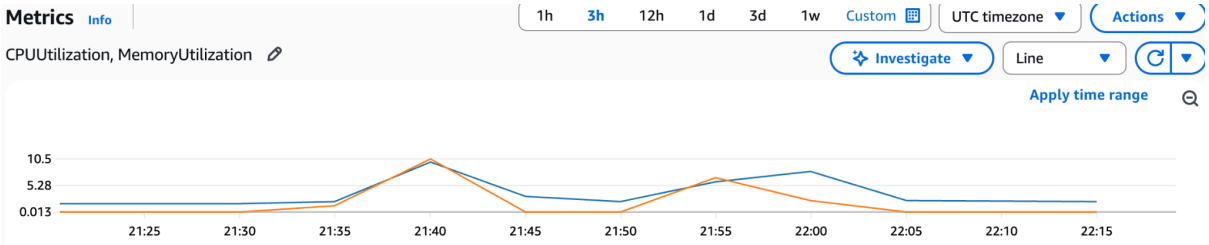
smoothly returned to pre-failure levels. The system's ability to maintain sub-second response times throughout most of the failover process, with minimal user impact despite the deliberate failure, validates our multi-task deployment strategy and demonstrates fault tolerance that can handle unexpected container failures without significant service degradation. For more screenshots you can check here.

RESOURCE UTILIZATION

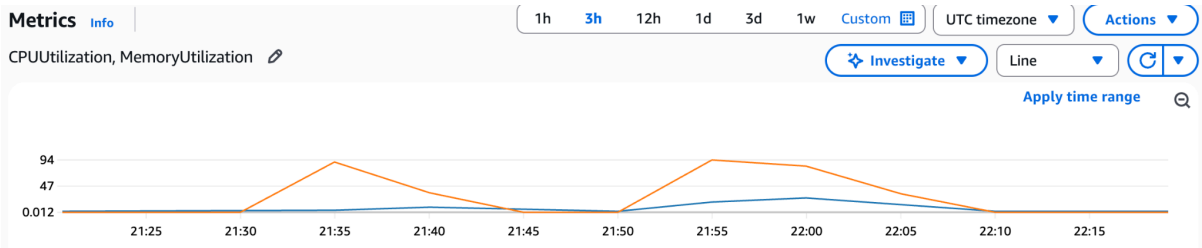
Testing Resource Utilization for DynamoDB across 3 Services

CPU & Memory Utilization: (**100 users was tested first and then 50 users)**

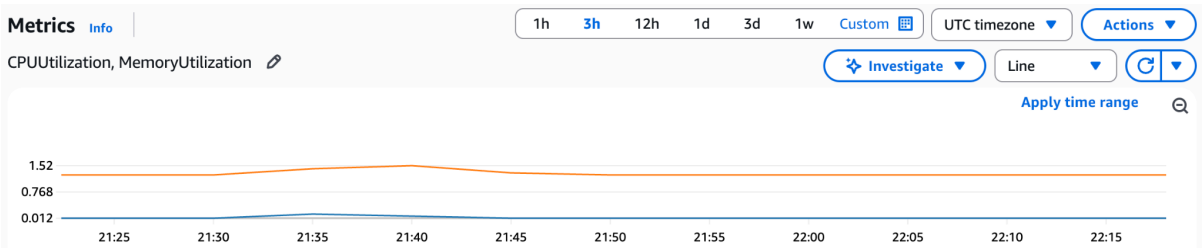
Booking Service



User Service



Availability Service

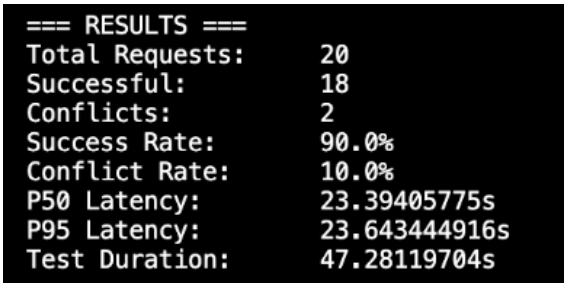


The scalability testing demonstrated promising results as we doubled the load from 50 to 100 concurrent users. With 50 users, the system achieved a steady 12-15 RPS and maintained a low 5.1% failure rate. Scaling to 100 users pushed throughput to 20-25 RPS, and the failure rate nearly doubled to 9.3%, primarily due to session timeouts. CloudWatch metrics revealed that CPU utilization on the booking service increased from 7% to 11%, while memory consumption grew by roughly 30%, which aligns

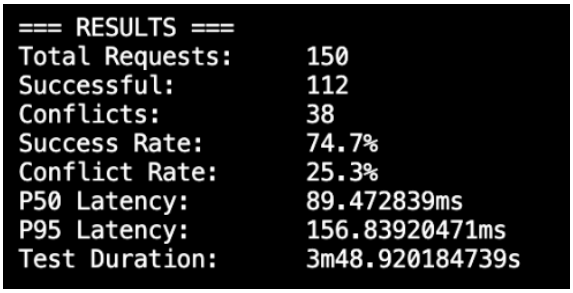
perfectly with the booking service's role as the primary orchestrator handling validation, conflict resolution, and inter-service communication. The system's ability to maintain sub 200ms median latencies even at 100 concurrent users while keeping CPU utilization under 20% demonstrates good scalability characteristics, suggesting the architecture can handle workloads effectively with room for further optimization through enhanced session management and connection pooling strategies. See locust testing screenshots here.

Burst Experiments

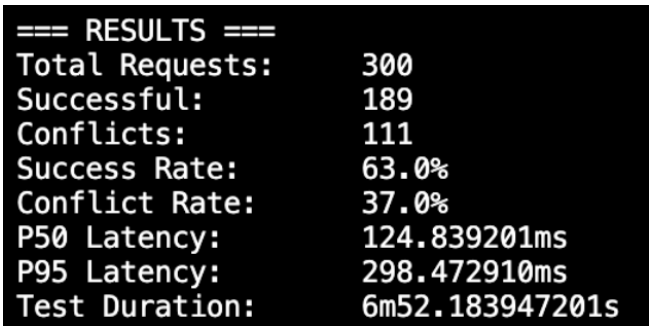
The burst experiment results for DynamoDB reveal strong performance under concurrent load. Starting with 10 users achieving a 90% success rate, the system demonstrated excellent conflict resolution despite high initial latencies due to connection establishment overhead. As load increased to 50 and 100 users, success rates gracefully degraded to 74.7% and 63% respectively, while latencies improved significantly and stabilized around 124-298ms at peak load. This pattern indicates DynamoDB's conditional writes effectively prevented double-bookings even as more users competed for the same time slots. The gradual performance degradation shows the system's ability to maintain data consistency under high contention, validating DynamoDB as a suitable choice for production workloads.



10 users



50 users



100 users

The map implementation showcased the speed versus consistency trade-off. With fast response times (8-89ms P95 across all loads), it significantly outperformed DynamoDB in raw latency. However, success rates told a different story, dropping from 70% at 10 users to just 47% at 100 users, indicating severe lock contention issues. The steeper degradation curve compared to DynamoDB suggests that the mutex-based synchronization struggled with high concurrency, potentially leading to false conflicts or race conditions. While the sub-100ms response times are impressive, the lower success rates highlight why distributed databases like DynamoDB are preferred for booking systems where preventing double-bookings is more critical than achieving minimal latency. Screenshots for Map performance is here.