CS6650 Assignment 3

Github:https://github.com/Alkatripathi-004/CS6650-Assignment1/tree/main/assignment3

Database Design Document: Chat System Persistence

1. Database Choice Justification

For this distributed chat system, we selected Amazon DynamoDB (NoSQL) over traditional RDBMS solutions (MySQL/PostgreSQL). This decision was driven by the following trade-offs and requirements:

1. Write Throughput & Scaling:
    - Requirement: The system must handle high-velocity concurrent writes (thousands of messages/second) from multiple load-balanced server instances.
    - Justification: DynamoDB provides single-digit millisecond latency at any scale. Unlike RDBMS, which requires complex sharding to scale writes horizontally, DynamoDB automatically partitions data across physical servers based on the Partition Key. Using On-Demand Capacity mode allows the database to instantly accommodate the burst traffic of our load tests without manual provisioning.
2. Data Model Fit:
    - Requirement: Chat messages are immutable time-series data. The primary access patterns are Key-Value lookups (Room History) and Range Queries (Time Windows).
    - Justification: The relational overhead (ACID compliance, Joins, Foreign Keys) is unnecessary for chat logs. DynamoDB's Key-Value architecture is optimized exactly for retrieving ordered lists of items, matching our "Room History" requirement perfectly.
3. Operational Overhead:
    - Requirement: The system requires high availability and minimal maintenance.
    - Justification: As a Serverless offering, DynamoDB removes the need for OS patching, connection pool tuning, or managing master-slave replication lag, allowing us to focus purely on application logic and query optimization.

2. Complete Schema Design

Table Name: `ChatMessages`

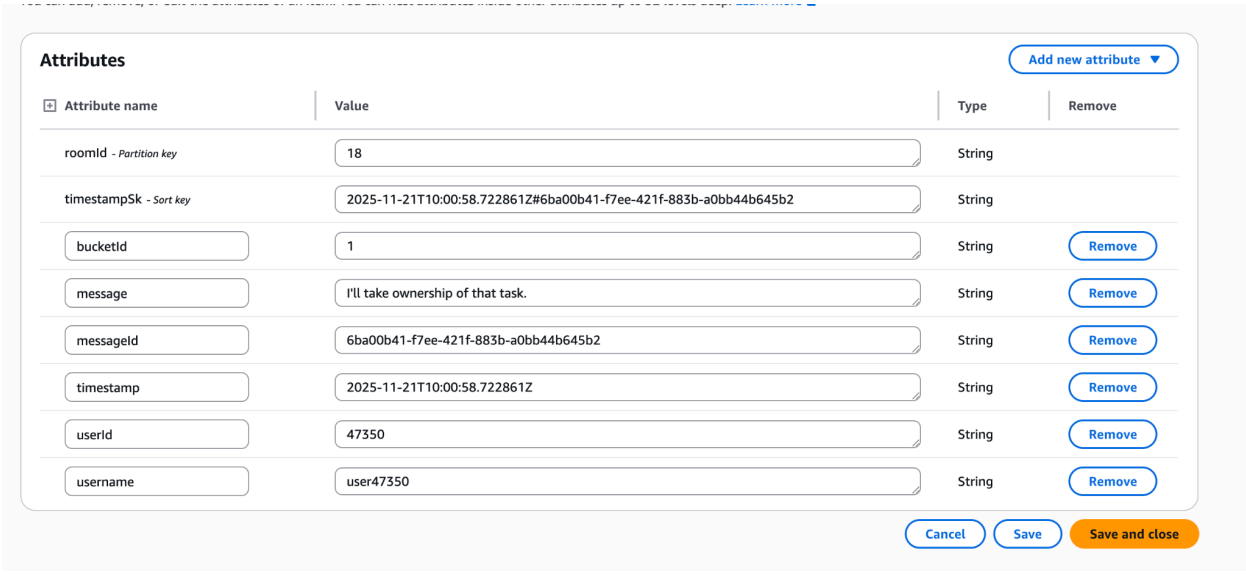
Primary Key Structure

- Partition Key (PK): `roomId` (String)
    - *Reasoning:* This distributes data evenly across partitions. Since users query history by Room, this allows efficient retrieval of all messages for a specific room in a single query.
- Sort Key (SK): `timestampSk` (String)

- *Format:* `ISO-8601-Timestamp#UUID` (e.g., `2025-11-21T10:00:00Z#a1b2-c3d4`)
- *Reasoning:*
    1. Ordering: DynamoDB automatically stores items in sorted order by SK. This meets the requirement for "Ordered list of messages" without client-side sorting.
    2. Idempotency: Appending the UUID ensures uniqueness. If a client retries sending a message at the exact same millisecond, the composite key remains identical, allowing DynamoDB to treat it as an idempotent overwrite rather than creating a duplicate.

Attributes

Attribute Name	Type	Description
`roomId`	String	(PK) The room identifier.
`timestampSk`	String	(SK) Composite key for sorting and uniqueness.
`userId`	String	ID of the sender.
`username`	String	Display name of sender.
`message`	String	The message payload.
`timestamp`	String	Raw ISO timestamp for analytics parsing.
`bucketId`	String	(GSI PK) A random shard ID (0-4) used for Write Sharding.



3. Indexing Strategy

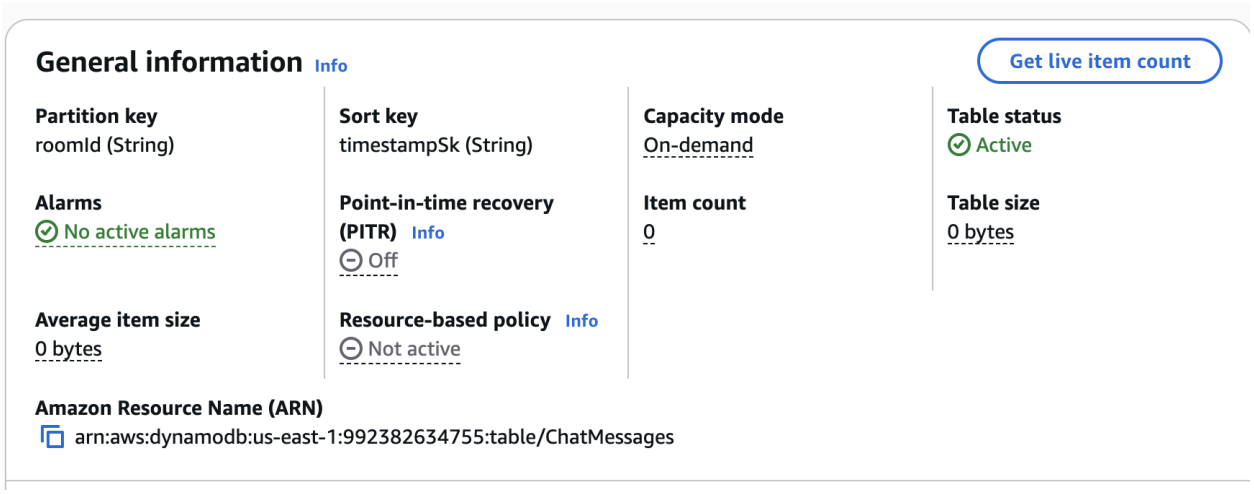
To support the required access patterns that do not fit the Main Table's PK/SK structure, we implemented two Global Secondary Indexes (GSIs).

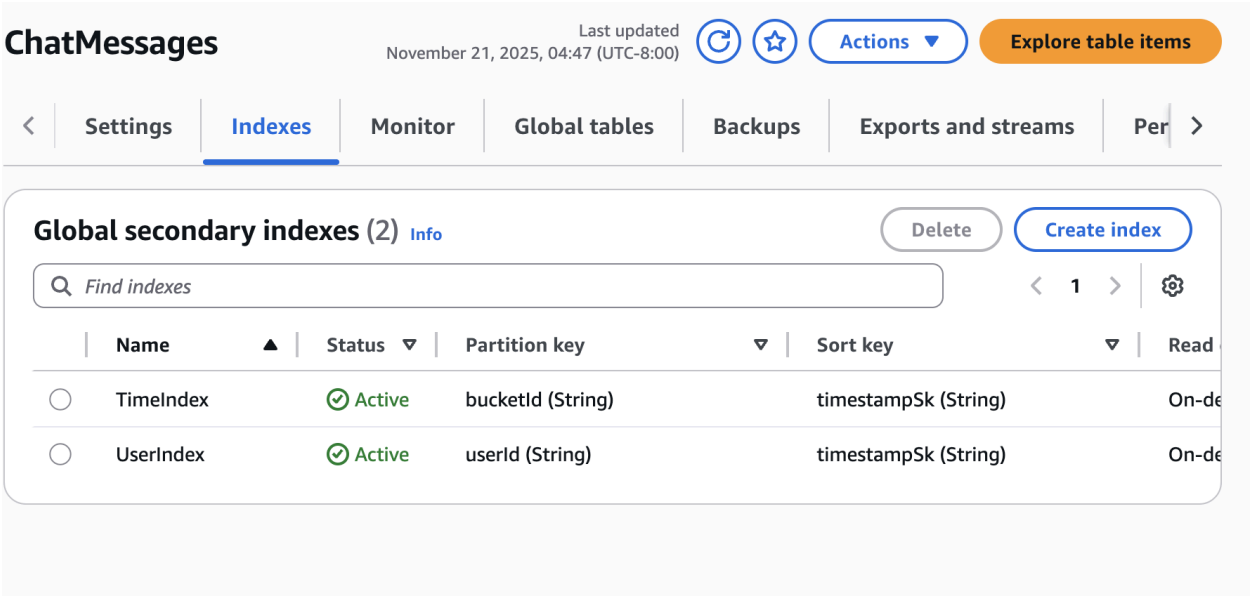
Index 1: `UserIndex`

- Goal: "Get user's message history across all rooms" (< 200ms).
- Partition Key: `userId`
- Sort Key: `timestampSk`
- Projection: `ALL`
- Justification: The main table is partitioned by Room. Finding a user's messages would require a full table scan ($O(N)$).. This GSI effectively creates a "Materialized View" organized by User, making the lookup $O(1)$ or $O(k)$ where k is the user's message count.

Index 2: `TimeIndex` (Write Sharding Strategy)

- Goal: "Count active users in time window" (< 500ms) & High-Throughput Analytics.
- Partition Key: `bucketId` (Values: "0", "1", "2", "3", "4")
- Sort Key: `timestampSk`
- Projection: `ALL`
- Justification:
  - The Hot Partition Problem: If we indexed strictly by `timestamp`, all concurrent writes (e.g., 4,000/sec) would target the *same* partition tip (the current second), causing write throttling.
  - The Solution (Scatter-Gather): We assign a random `bucketId` (0-4) to every message. This spreads the write load across 5 physical partitions.
  - Read Strategy: To query a time window, the application executes 5 parallel queries (one per bucket) and aggregates the results in memory. This ensures both high write throughput and fast, bounded-time analytics queries.





4. Scaling Considerations

1. Write-Behind Buffering:
    - The application implements a Write-Behind pattern using an internal `BlockingQueue`. This acts as a shock absorber. If traffic spikes briefly above DynamoDB's provisioned limits, the queue absorbs the load, preventing 500 Errors at the API level.
2. Capacity Modes:
    - Current: On-Demand Mode is used to handle the unpredictable spikes of the load test without manual intervention.
    - Future/Prod: We would switch to Provisioned Capacity with Auto-Scaling for cost optimization once traffic patterns are predictable.
3. Time To Live (TTL):
    - To prevent the table from growing infinitely (which increases storage costs and backup times), we can enable DynamoDB TTL on the `timestamp` attribute to automatically expire and delete messages older than a specific retention period (e.g., 1 year).

5. Backup and Recovery Approach

1. Point-in-Time Recovery (PITR):
    - We enable PITR on the `ChatMessages` table. This allows continuous backups with per-second granularity. In the event of accidental logical corruption (e.g., a bad script deleting rows), we can restore the table to any state in the last 35 days.
2. On-Demand Backups:
    - Before any major schema change or migration, an On-Demand Backup is triggered via AWS Backup to create an immutable snapshot of the data for compliance and long-term archiving.
3. Dead Letter Queue (DLQ):

    - Data integrity is preserved via an application-level DLQ. If the database is unreachable or writes fail repeatedly after exponential backoff, failed messages are routed to a holding area (in-memory or SQS) for later inspection and replay, ensuring zero data loss.

Error Handling

1. Database Layer

We use the Resilience4j library to protect the application from DynamoDB failures.

- Circuit Breaker:
    - Mechanism: Using `@CircuitBreaker(name = "dynamoDB")`.
    - Behavior: If writes to DynamoDB start failing (e.g., >50% failure rate), the Circuit Breaker Opens. Subsequent write attempts are immediately blocked without waiting for timeouts, saving system resources.
    - Recovery: After a wait duration (10s), it enters a "Half-Open" state to test if the database is back online.
- Exponential Backoff:
    - Mechanism: Using `@Retry` with `enableExponentialBackoff=true`.
    - Behavior: If a write fails due to a transient error (e.g., network blip), it retries automatically with increasing delays (100ms -> 200ms -> 400ms).
- Dead Letter Queue (DLQ):
    - Mechanism: The `fallbackWrite` method in `DynamoDBBatchWriter`.
    - Behavior: If the Circuit Breaker is open or all retries fail, messages are not lost. They are routed to the `DlqService` (in-memory queue), which can be inspected via the `/api/analytics/dlq` endpoint.

2. Concurrency Layer

We protect the server from crashing under heavy load when the database is slow.

- Bounded Buffer:
    - Mechanism: `LinkedBlockingQueue` in `MessagePersistenceService` is capped at 50,000 items.
    - Behavior: Prevents `OutOfMemoryError` if the DB is down for an extended period.
- Rate-Limited Error Logging:
    - Mechanism: The `lastLogTime` check in `persistAsync`.
    - Behavior: If the buffer fills up, we drop messages to keep the server alive. Crucially, we only log this error once every 5 seconds to prevent disk I/O and CPU from choking on millions of log statements (preventing the "Death Spiral").

3. Data Integrity Layer (Idempotency)

We handle duplicate messages caused by network retries.

- Database Level Idempotency:
    - Mechanism: Composite Primary Key (`roomId` + `timestamp#messageId`).
    - Behavior: If a message is sent twice, DynamoDB simply overwrites the existing record with the exact same data. No duplicate rows are created.
- Consumer Level Deduplication:
    - Mechanism: `processedMessageIds` Set in `RabbitMQConsumerService`.
    - Behavior: We track IDs in memory to prevent processing the same message logic twice in the short term.

Consumer Modifications

Batch Size Optimization:

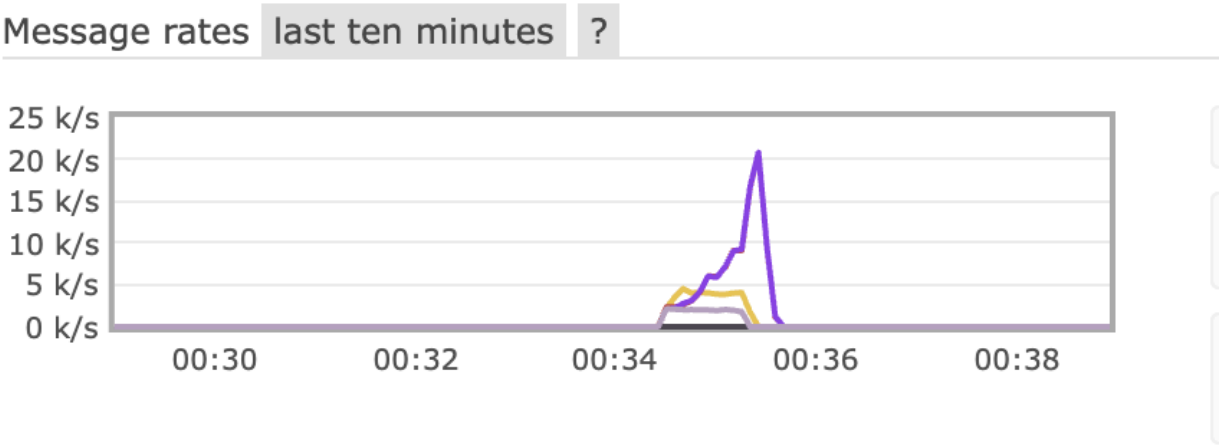
Running tests with different batch size and flush intervals to find optimal batch size and flush interval.

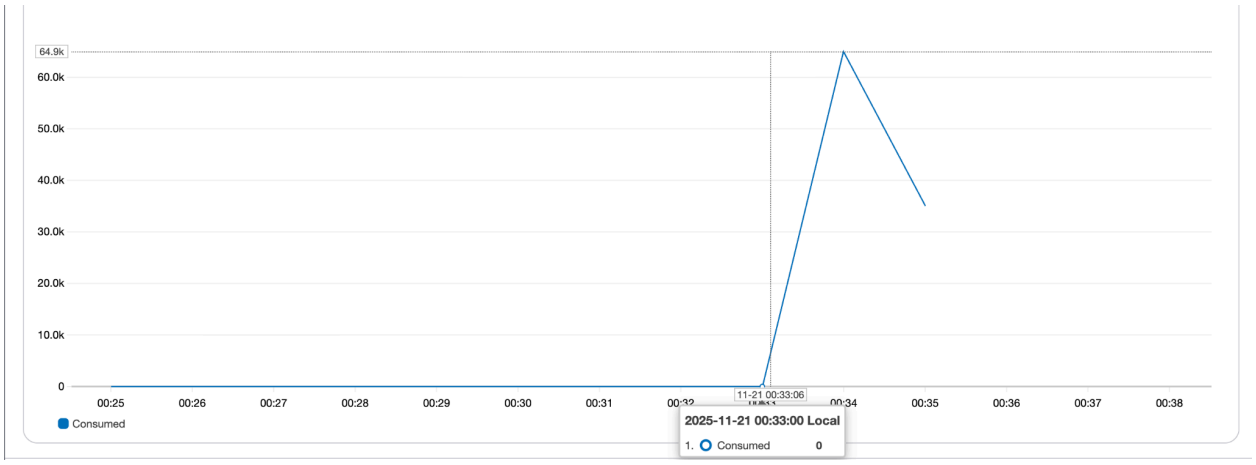
Test 1 => Batch Size: 100, Flush interval: 100

Result:

1. Saw the queue getting backed up.
2. Very high p50, p95 and p99 latency.

```
--- Load Test Results ---
Total Successful Messages: 100000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 65.56 seconds
Throughput: 1525.25 messages/second

------------------------------------------------
          CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   23084.65 ms
Min Response Time:    352 ms
Max Response Time:    36083 ms
------------------------------------------------
P50 (Median) Latency: 23945 ms
P95 Latency:          33125 ms
P99 Latency:          34633 ms
------------------------------------------------
```





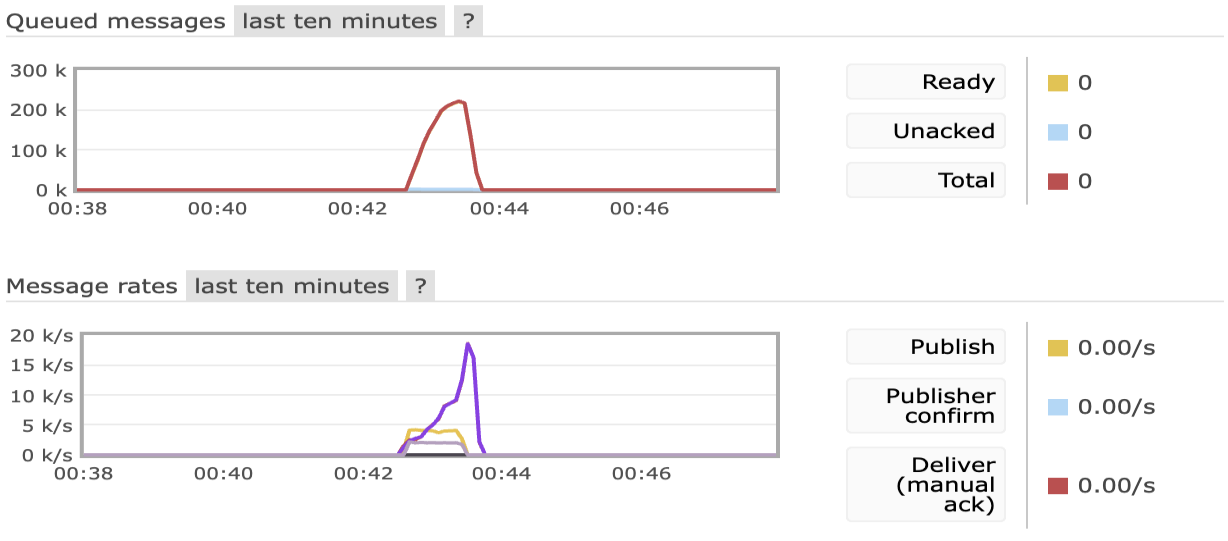
Test 2

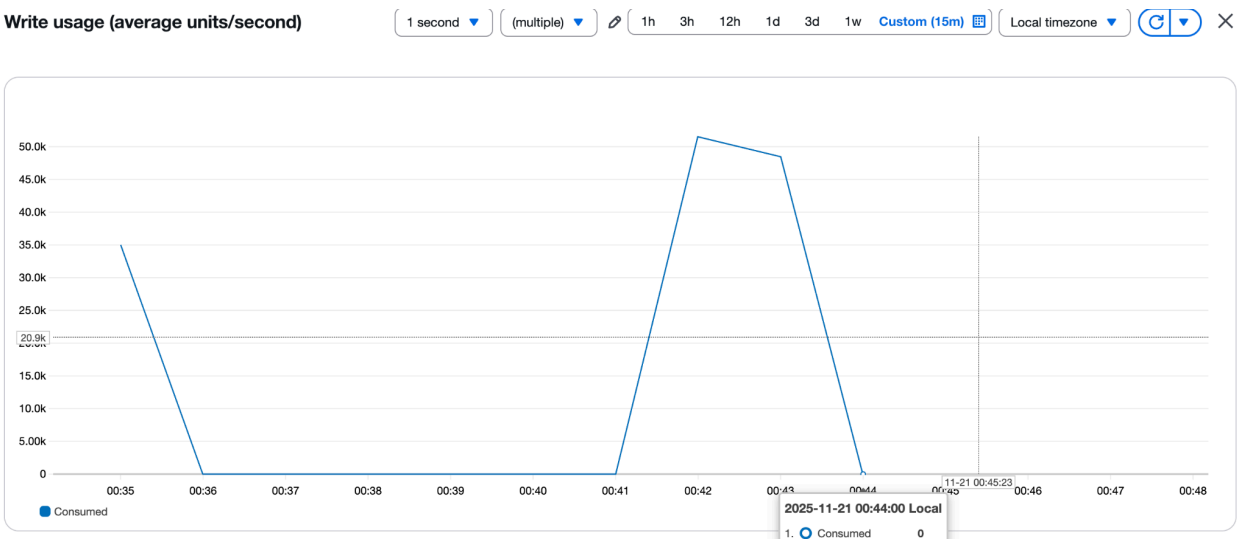
Batch Size: 500
Flush: 100

Result:

1. Queue getting backed up with thousands of messages.
2. DBWriter not able to flush the messages fast enough.
3. High p50, p95, p99 latency

```
--- Load Test Results ---
Total Successful Messages: 100000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 63.59 seconds
Throughput: 1572.52 messages/second

------------------------------------------------
          CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   22663.05 ms
Min Response Time:    359 ms
Max Response Time:    33394 ms
------------------------------------------------
P50 (Median) Latency: 24031 ms
P95 Latency:          31040 ms
P99 Latency:          32407 ms
------------------------------------------------
```





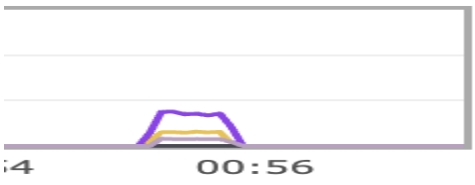
Test 3:

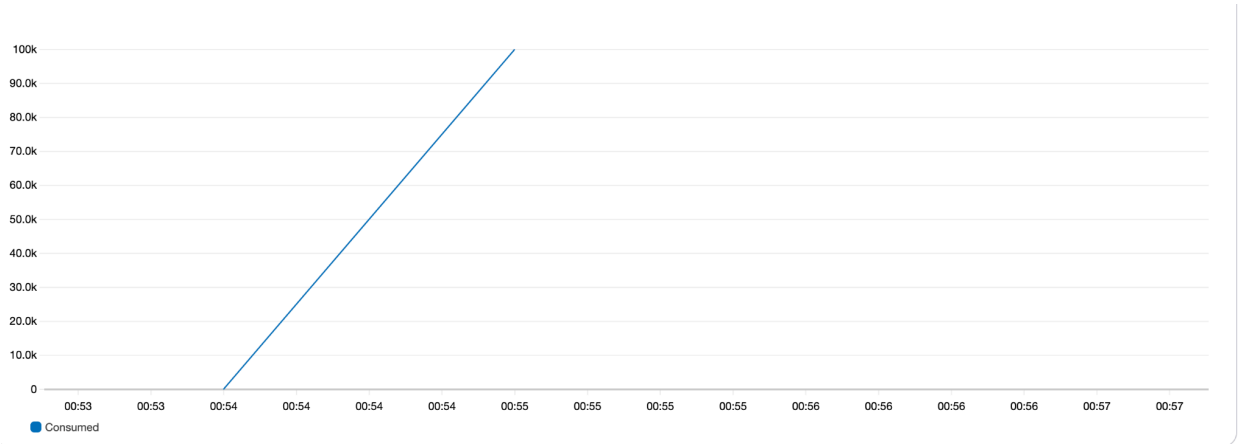
Batch Size: 1000
Flush : 200ms

Result:

1. Improved results compared to previous two tests.
2. Queue not getting backed up.
3. DBWriter still taking to flush out all the messages.
4. Latency lower than previous two tests but still high

```
--- Load Test Results ---
Total Successful Messages: 100000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 35.67 seconds
Throughput: 2803.40 messages/second

------------------------------------------------
          CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   1154.46 ms
Min Response Time:    69 ms
Max Response Time:    11544 ms
------------------------------------------------
P50 (Median) Latency: 240 ms
P95 Latency:          6829 ms
P99 Latency:          8984 ms
------------------------------------------------
```





Test 4

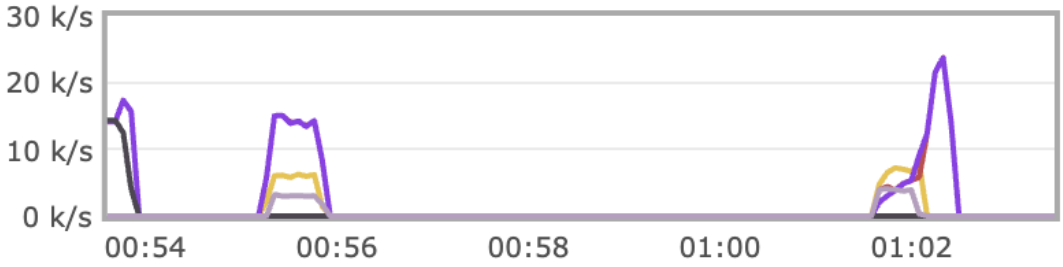
Batch 5000
Flush 500

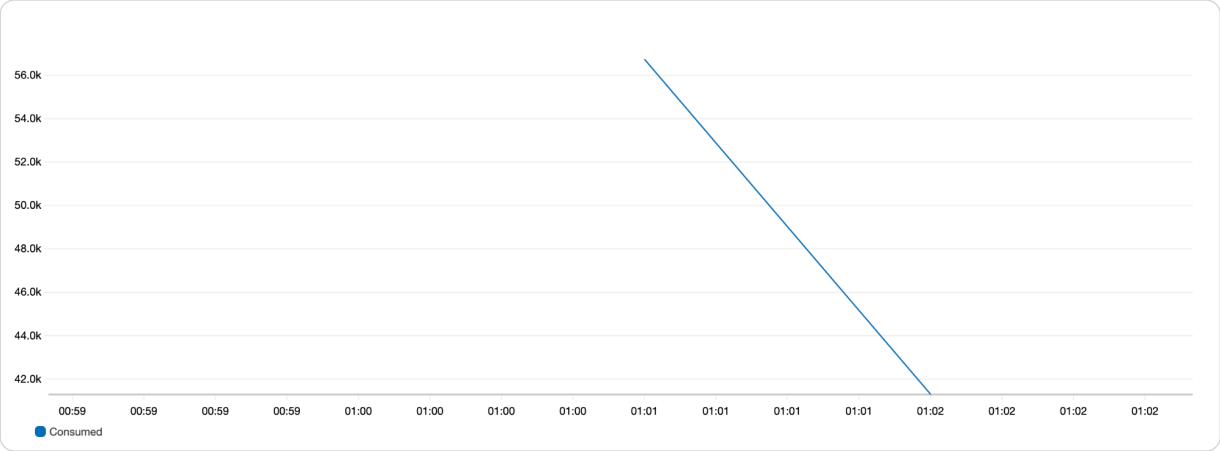
Result:

1. High latency for p50, p95, p99
2. Queue getting backed up
3. DBWriter taking too long to flush out all the messages.

```
--- Load Test Results ---
Total Successful Messages: 100000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 48.28 seconds
Throughput: 2071.12 messages/second

------------------------------------------------
          CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   25570.94 ms
Min Response Time:    382 ms
Max Response Time:    37046 ms
------------------------------------------------
P50 (Median) Latency: 26295 ms
P95 Latency:          30245 ms
P99 Latency:          32115 ms
------------------------------------------------
```

Message rates last ten minutes ?





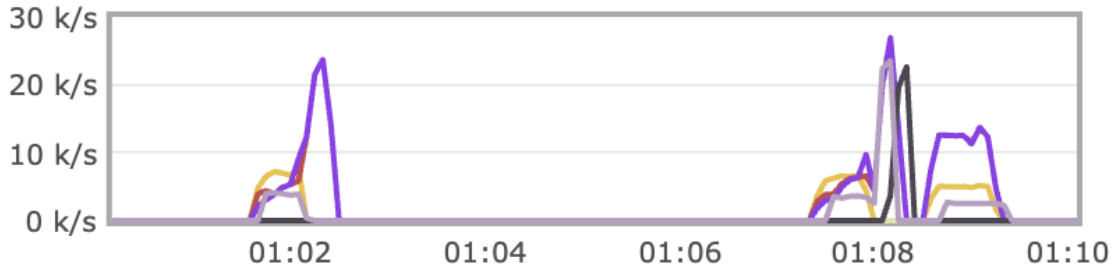
Test 5
Batch 2000
Flush Interval: 200

Result:

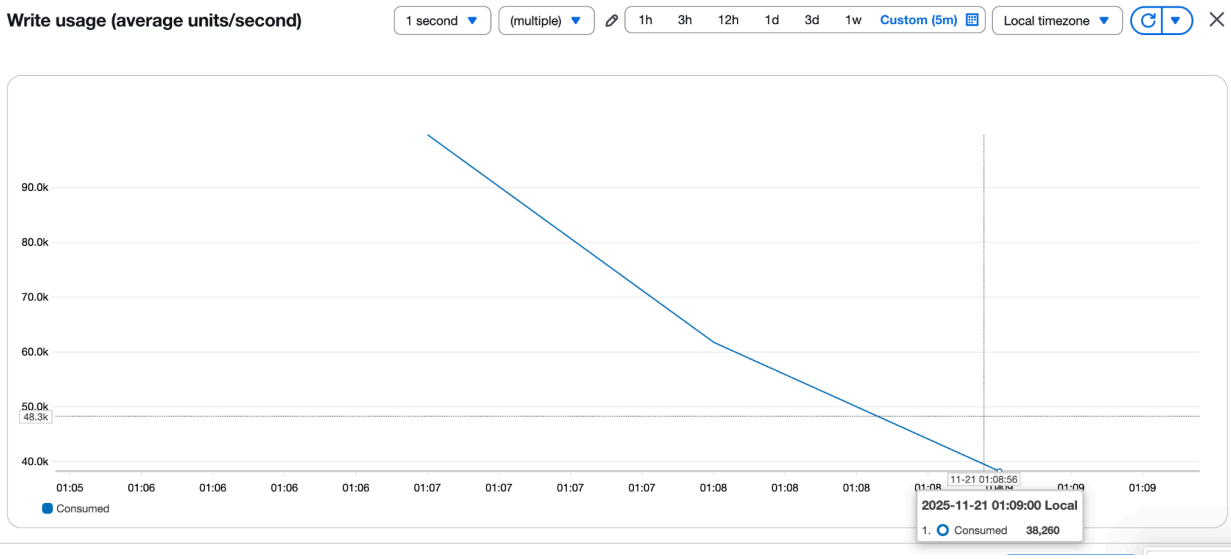
1. Minimal queue depth.
2. Improved latency for p50, p95, p99
3. High throughput.
4. Selecting this for load tests

```
--- Load Test Results ---
Total Successful Messages: 100000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 40.14 seconds
Throughput: 2491.40 messages/second

------------------------------------------------
          CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   170.01 ms
Min Response Time:    68 ms
Max Response Time:    3064 ms
------------------------------------------------
P50 (Median) Latency: 92 ms
P95 Latency:          604 ms
P99 Latency:          1337 ms
------------------------------------------------
```

Message rates last ten minutes ?



Write usage (average units/second)



Load Testing

Baseline Test

Messages: 500000

1. Console showing
    a. Total throughput
    b. p50, p90, p99 latencies
    c. Result of analytics queries

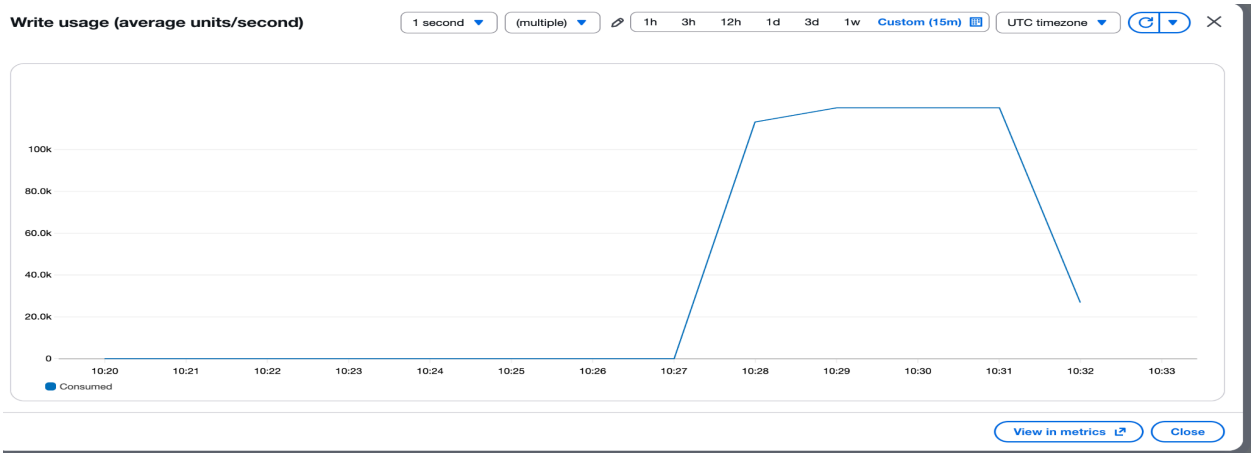
```
--- Load Test Results ---
Total Successful Messages: 500000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 250.14 seconds
Throughput: 1998.86 messages/second

------------------------------------------------
           CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   100.04 ms
Min Response Time:    66 ms
Max Response Time:    3145 ms
------------------------------------------------
P50 (Median) Latency: 75 ms
P95 Latency:          121 ms
P99 Latency:          871 ms
------------------------------------------------
```

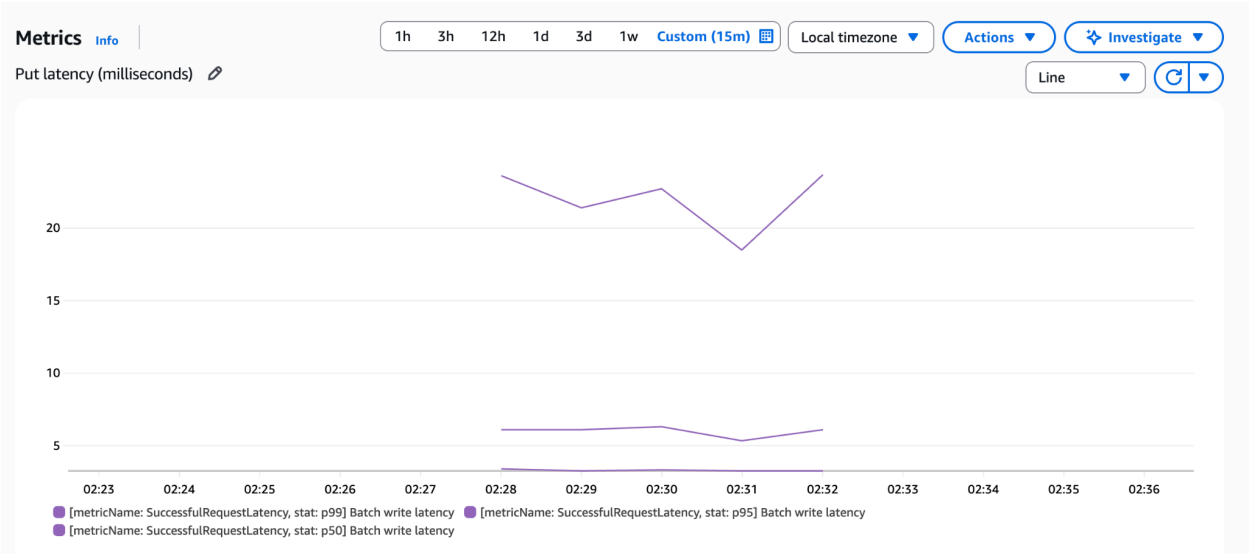
```
--- PHASE 2: ANALYTICS & PERSISTENCE VERIFICATION ---
Waiting 30 seconds to allow Write-Behind buffer to flush to DynamoDB...

[Query 1] Fetching System Analytics Stats...
Response:
{
  "top_active_users" : [ {
    "22513" : 5
  }, {
    "39321" : 4
  }, {
    "96029" : 3
  }, {
    "35006" : 3
  }, {
    "96139" : 3
  } ],
  "window_start" : "2025-11-21T09:32:24.730070047Z",
  "throughput_msg_per_sec" : "2932.25",
  "unique_active_users" : 17282,
  "top_active_rooms" : [ {
    "1" : 1125
  }, {
    "16" : 1081
  }, {
    "9" : 1079
  }, {
    "2" : 1055
  }, {
    "8" : 1010
  } ],
  "total_messages_in_window" : 18957,
  "window_end" : "2025-11-21T10:32:24.730063486Z"
}
```

```
[Query 2] Fetching History for Room 1...
>> Messages found in Room 1: 4559

[Query 3] Fetching History for User ID '1'...
>> Messages found for User 1: 10

[SUCCESS] Analytics API verification completed.
=========== PERFORMANCE TEST COMPLETE ===========
```

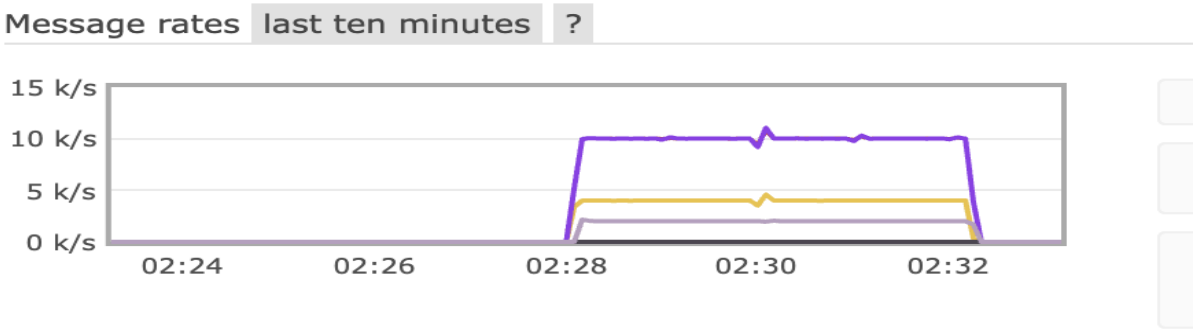
DB Metrics:

Average Writes





Queue Depth:



2. Stress Test

Total messages: 1000000

Results:

1. Higher throughput compared to first test
2. Bad latency compared to the previous test. We see higher p99 latency here.
3. Queue getting backed up.
4. DBWriter takes time to flush out the messages resulting in an increase in messages in the blocking queue.

Console Screenshots

```
--- Load Test Results ---
Total Successful Messages: 1000000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 457.41 seconds
Throughput: 2186.20 messages/second

------------------------------------------------
           CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   179.89 ms
Min Response Time:    66 ms
Max Response Time:    6177 ms
------------------------------------------------
P50 (Median) Latency: 76 ms
P95 Latency:          754 ms
P99 Latency:          2102 ms
------------------------------------------------
```

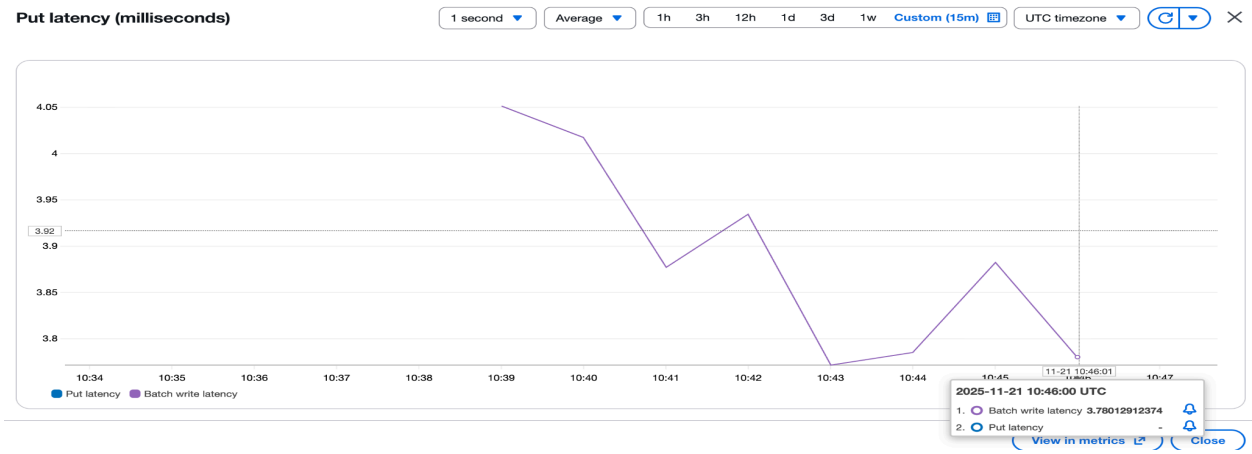
```
--- PHASE 2: ANALYTICS & PERSISTENCE VERIFICATION ---
Waiting 30 seconds to allow Write-Behind buffer to flush to DynamoDB...

[Query 1] Fetching System Analytics Stats...
Response:
{
  "top_active_rooms" : [ {
    "1" : 1125
  }, {
    "16" : 1081
  }, {
    "9" : 1079
  }, {
    "2" : 1055
  }, {
    "8" : 1010
  } ],
  "total_messages_in_window" : 18957,
  "window_end" : "2025-11-21T10:47:00.510515014Z",
  "top_active_users" : [ {
    "22513" : 5
  }, {
    "39321" : 4
  }, {
    "96029" : 3
  }, {
    "35006" : 3
  }, {
    "96139" : 3
  } ],
  "window_start" : "2025-11-21T09:47:00.510521809Z",
  "throughput_msg_per_sec" : "2932.25",
  "unique_active_users" : 17282
}
```

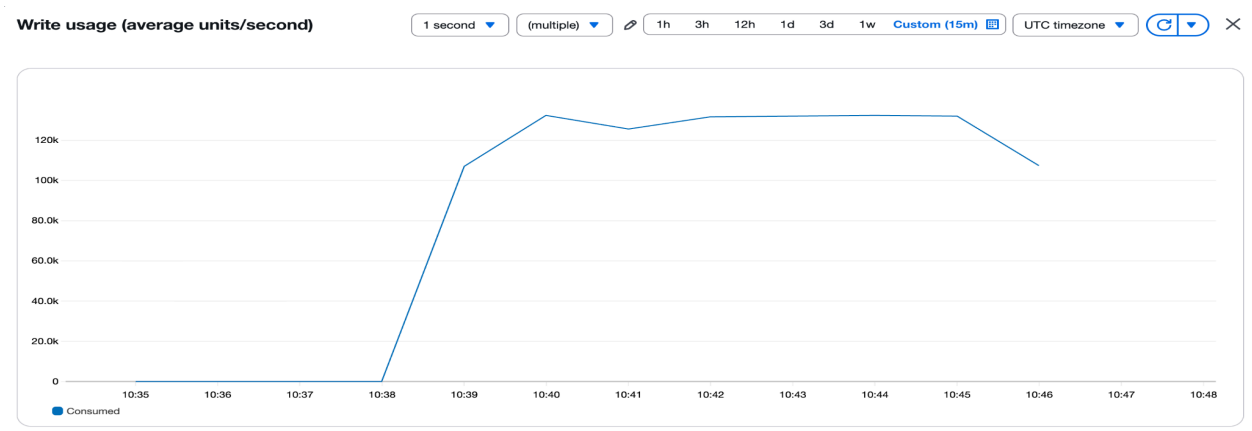
```
[Query 2] Fetching History for Room 1...
>> Messages found in Room 1: 4559

[Query 3] Fetching History for User ID '1'...
>> Messages found for User 1: 22

[SUCCESS] Analytics API verification completed.
========== PERFORMANCE TEST COMPLETE ==========
```

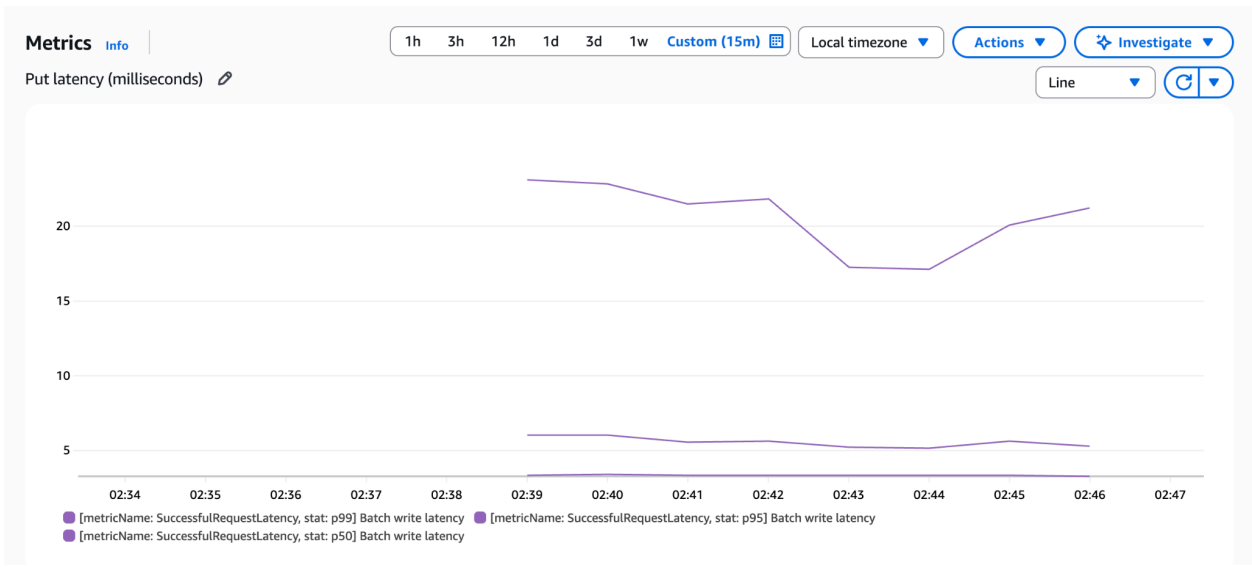
Queue:

DB Metrics:

1. Sustained write throughput
2. Low latency < 20ms







3. Endurance Test

Run (Success):

Time Duration: 25 Minute:

I saw failures during high throughput because of out of memory errors and dropped connections.

TradeOff:

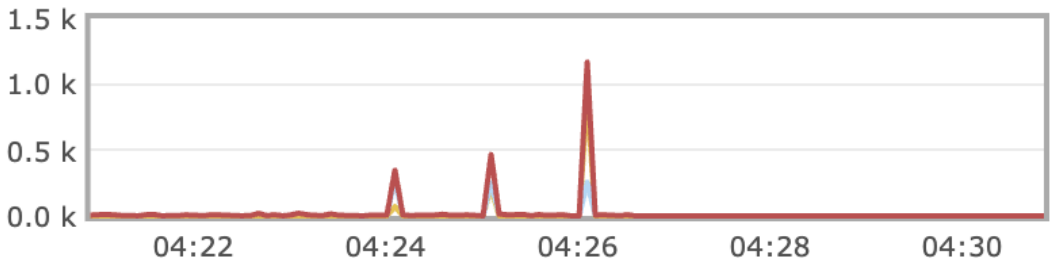
1. Lower throughput for sustained load.
2. Increasing the flush batch size.
3. Increased number of server instances leads to higher throughput.
4. Creating a baseline read/write throughput for ondemand DynamoDB.

```
--- Load Test Results ---
Total Successful Messages: 1500000
Total Failed Messages: 0
Total Initial Connections: 256
Total Reconnections: 0
Total Runtime: 1500.20 seconds
Throughput: 999.87 messages/second

-----------------------------------------------
           CLIENT SIDE LATENCY METRICS
-----------------------------------------------
Mean Response Time:   220.27 ms
Min Response Time:    64 ms
Max Response Time:    14976 ms
-----------------------------------------------
P50 (Median) Latency: 70 ms
P95 Latency:          130 ms
P99 Latency:          7226 ms
-----------------------------------------------
```

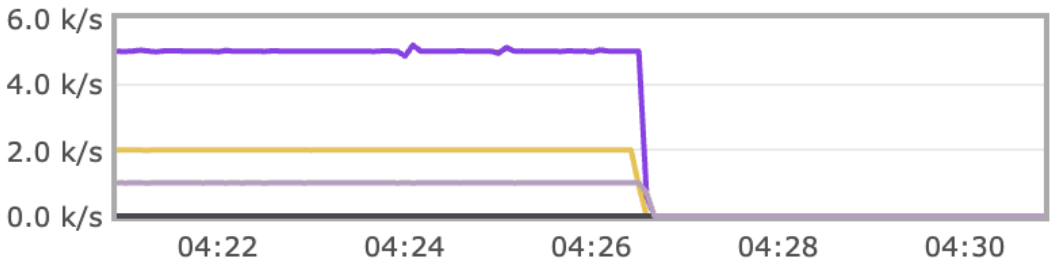
```
--- PHASE 2: ANALYTICS & PERSISTENCE VERIFICATION ---
Waiting 30 seconds to allow Write-Behind buffer to flush to DynamoDB...

[Query 1] Fetching System Analytics Stats...
Response:
{
  "top_active_rooms" : [ {
    "2" : 1056
  }, {
    "11" : 1002
  }, {
    "9" : 994
  }, {
    "7" : 992
  }, {
    "5" : 978
  } ],
  "total_messages_in_window" : 18964,
  "window_end" : "2025-11-21T12:26:44.004459346Z",
  "top_active_users" : [ {
    "49695" : 4
  }, {
    "36635" : 4
  }, {
    "98967" : 4
  }, {
    "68979" : 4
  }, {
    "65802" : 4
  } ],
  "window_start" : "2025-11-21T11:26:44.004490734Z",
  "throughput_msg_per_sec" : "27.11",
  "unique_active_users" : 17278
}
```

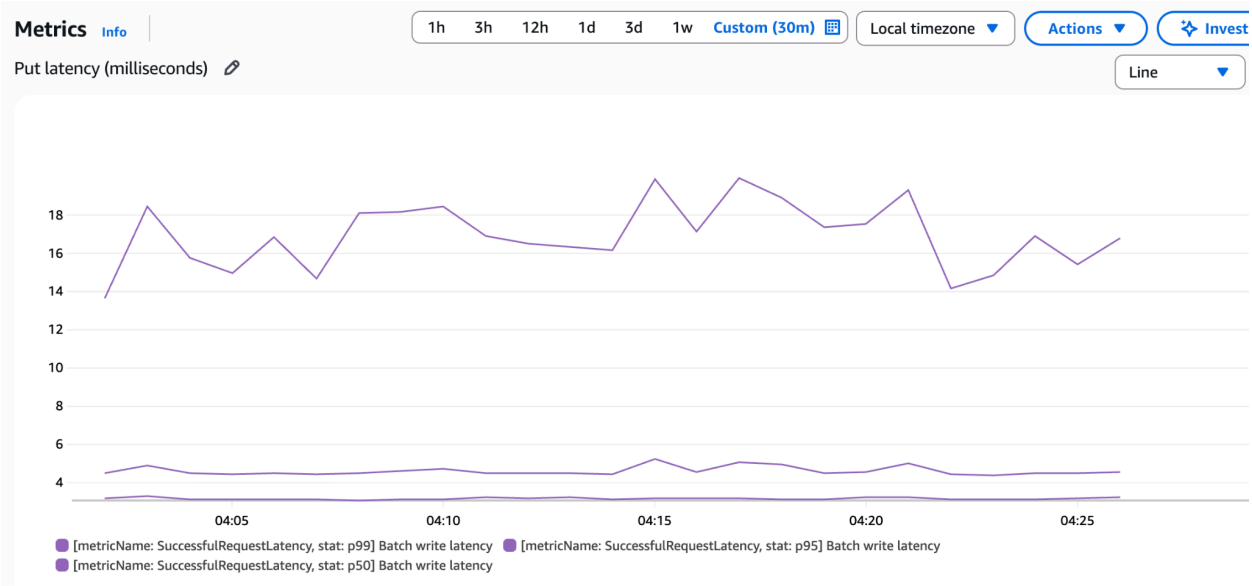
```
[Query 2] Fetching History for Room 1...
>> Messages found in Room 1: 4557

[Query 3] Fetching History for User ID '1'...
>> Messages found for User 1: 64

[SUCCESS] Analytics API verification completed.
=========== PERFORMANCE TEST COMPLETE ===========
```

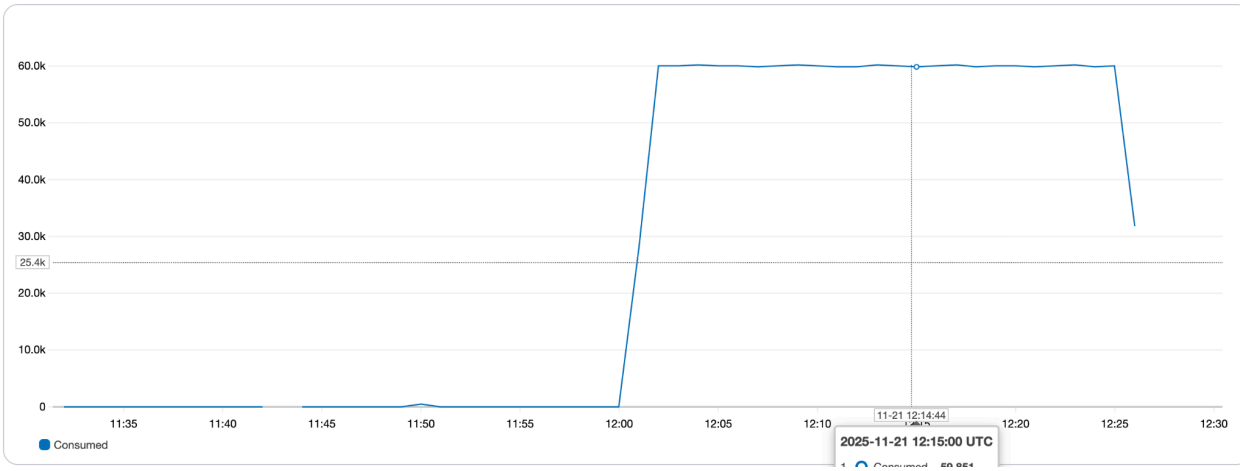
Queued messages last ten minutes ?



Message rates last ten minutes ?







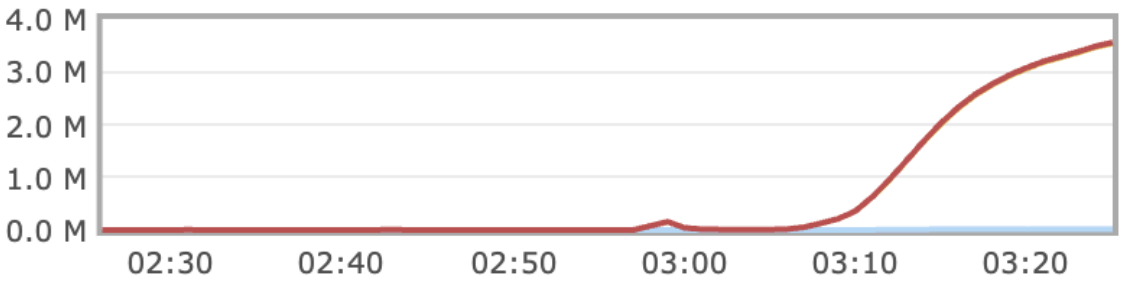
Previous Run (Failed):

1. Reconnection errors.
2. Java Application Heap - OutofMemory errors. Seems the blocking queues that we created for message buffers ended up hogging a lot of memory and the messages were not getting flushed out fast enough.
3. Queue was getting backed up and the messages were not being polled fast enough. Java application out of memory issues leading to lot of reconnections and dropped connections.
4. High latency is also observed during DB writes.

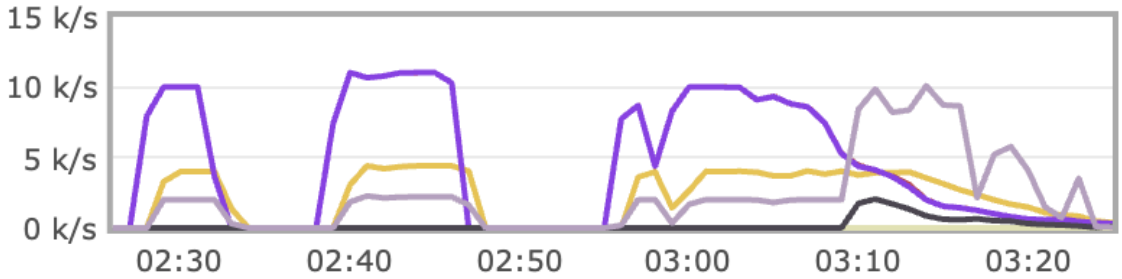
```
ngWebSocketHandlerDecorator : Transport error in StandardWebSocketSession[id=7f3989a2-6697-81
b8-fa14-a8980a7d7f2f, uri=ws://chat-app-alb-2109924358.us-east-1.elb.amazonaws.com/chat/18]

java.io.IOException: java.io.IOException: Broken pipe
        at org.apache.tomcat.websocket.WsRemoteEndpointImplBase.sendMessageBlock(WsRemoteEndp
ointImplBase.java:321) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsRemoteEndpointImplBase.sendMessageBlock(WsRemoteEndp
ointImplBase.java:257) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsSession.sendCloseMessage(WsSession.java:770) ~[tomca
t-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsSession.doClose(WsSession.java:589) ~[tomcat-embed-w
ebsocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsRemoteEndpointImplBase.sendMessageBlock(WsRemoteEndp
ointImplBase.java:319) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsRemoteEndpointImplBase.sendMessageBlock(WsRemoteEndp
ointImplBase.java:250) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsRemoteEndpointImplBase.sendPartialString(WsRemoteEnd
pointImplBase.java:223) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.apache.tomcat.websocket.WsRemoteEndpointBasic.sendText(WsRemoteEndpointBasic.j
ava:48) ~[tomcat-embed-websocket-10.1.20.jar!/:na]
        at org.springframework.web.socket.adapter.standard.StandardWebSocketSession.sendTextM
essage(StandardWebSocketSession.java:217) ~[spring-websocket-6.1.6.jar!/:6.1.6]
        at org.springframework.web.socket.adapter.AbstractWebSocketSession.sendMessage(Abstra
ctWebSocketSession.java:108) ~[spring-websocket-6.1.6.jar!/:6.1.6]
        at com.chat.cs6650assignment3.serverv3.ChatWebSocketHandler.handleTextMessage(ChatWeb
SocketHandler.java:78) ~[!/:1.0.0-SNAPSHOT]
        at org.springframework.web.socket.handler.AbstractWebSocketHandler.handleMessage(Abst
ractWebSocketHandler.java:43) ~[spring-websocket-6.1.6.jar!/:6.1.6]
        at org.springframework.web.socket.handler.WebSocketHandlerDecorator.handleMessage(Web
SocketHandlerDecorator.java:75) ~[spring-websocket-6.1.6.jar!/:6.1.6]
        at org.springframework.web.socket.handler.LoggingWebSocketHandlerDecorator.handleMess
age(LoggingWebSocketHandlerDecorator.java:56) ~[spring-websocket-6.1.6.jar!/:6.1.6]
        at org.springframework.web.socket.handler.ExceptionWebSocketHandlerDecorator.handleMe
```

room.1	classic	D TTL Lim DLX	running	847	800	1,647	0.00/s	7.8/s	7.8/s
room.10	classic	D TTL Lim DLX	running	504	800	1,304	0.00/s	19/s	19/s
room.11	classic	D TTL Lim DLX	running	107	800	907	0.00/s	5.2/s	5.2/s
room.12	classic	D TTL Lim DLX	running	398	800	1,198	0.00/s	13/s	13/s
room.13	classic	D TTL Lim DLX	running	1,658	800	2,458	0.00/s	6.6/s	6.6/s
room.14	classic	D TTL Lim DLX	running	4,139	800	4,939	2.8/s	5.2/s	5.2/s
room.15	classic	D TTL Lim DLX	running	1,013	800	1,813	0.00/s	10/s	10/s
room.16	classic	D TTL Lim DLX	running	1,119	800	1,919	0.00/s	9.2/s	9.2/s
room.17	classic	D TTL Lim DLX	running	1,818	800	2,618	0.00/s	12/s	12/s
room.18	classic	D TTL Lim DLX	running	2,288	800	3,088	0.00/s	10/s	10/s
room.19	classic	D TTL Lim DLX	running	226	800	1,026	0.00/s	14/s	14/s
room.2	classic	D TTL Lim DLX	running	1,392	800	2,192	0.00/s	11/s	11/s
room.20	classic	D TTL Lim DLX	running	2,021	800	2,821	0.00/s	18/s	18/s
room.3	classic	D TTL Lim DLX	running	1,973	800	2,773	4.0/s	9.0/s	9.0/s
room.4	classic	D TTL Lim DLX	running	1,680	800	2,480	0.00/s	7.8/s	7.8/s
room.5	classic	D TTL Lim DLX	running	2,283	800	3,083	0.00/s	8.2/s	8.2/s
room.6	classic	D TTL Lim DLX	running	1,134	800	1,934	0.00/s	8.6/s	8.6/s
room.7	classic	D TTL Lim DLX	running	1,308	800	2,108	0.00/s	7.6/s	7.6/s

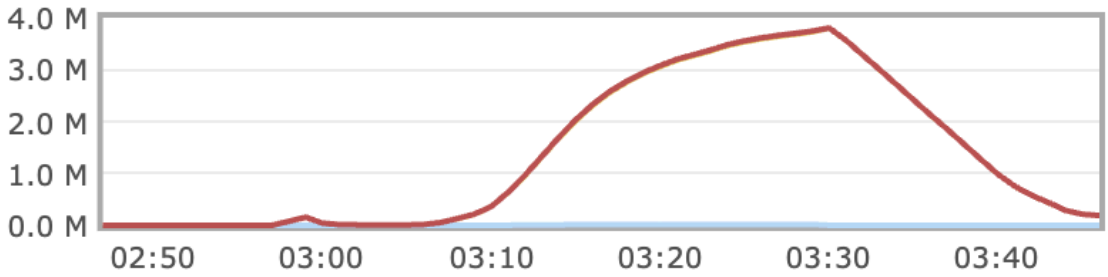
Queued messages last hour ?



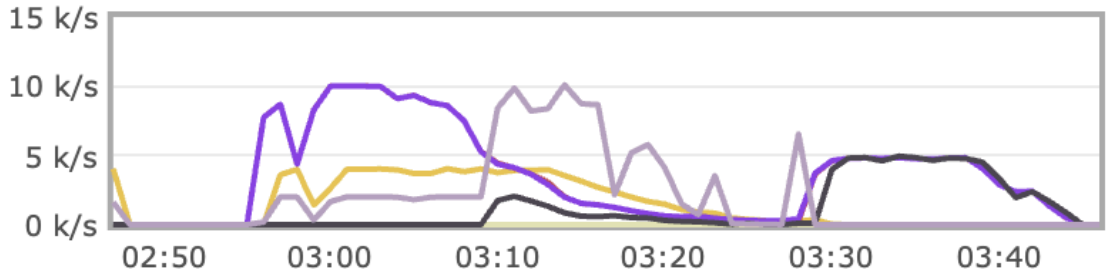
Message rates last hour ?

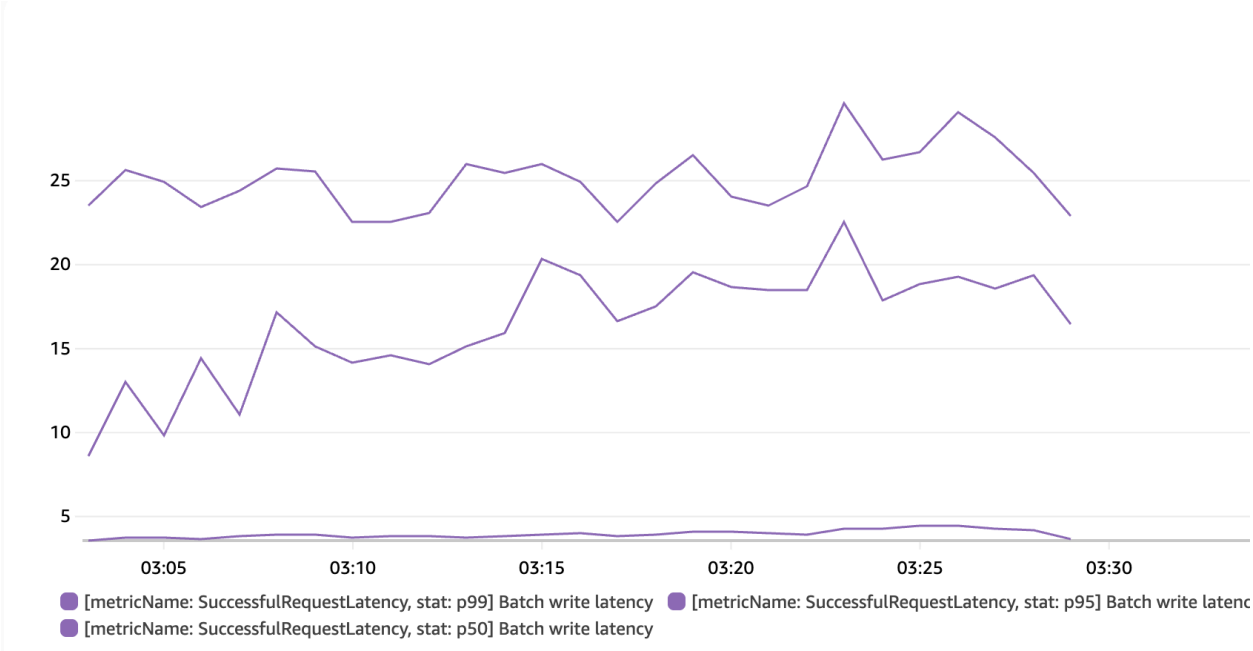


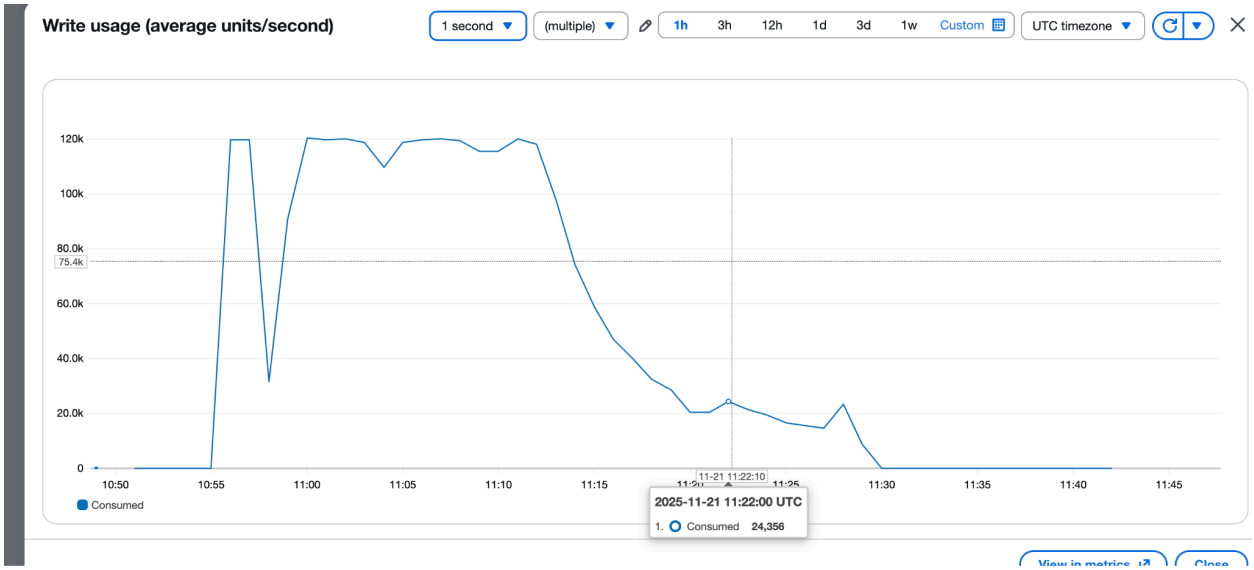
Queued messages last hour ?



Message rates last hour ?






Write usage (average units/second)
1 second
(multiple)
1h
3h
12h
1d
3d
1w
Custom
UTC timezone
120k
100k
80.0k
75.4k
60.0k
40.0k
20.0k
0
10:50
10:55
11:00
11:05
11:10
11:15
11:20
11:25
11:30
11:35
11:40
11:45
11-21 11:22:10
Consumed
2025-11-21 11:22:00 UTC
1. Consumed 24,356
View in metrics
Close

```
--- Load Test Results ---
Total Successful Messages: 1606378
Total Failed Messages: 1370377
Total Initial Connections: 256
Total Reconnections: 23617
Total Runtime: 2794.38 seconds
Throughput: 574.86 messages/second

------------------------------------------------
           CLIENT SIDE LATENCY METRICS
------------------------------------------------
Mean Response Time:   114803.44 ms
Min Response Time:    64 ms
Max Response Time:    1582275 ms
------------------------------------------------
P50 (Median) Latency: 960 ms
P95 Latency:          1199747 ms
P99 Latency:          1418555 ms
------------------------------------------------
```

```
080-exec-342] o.s.w.s.s.s.WebSocketHttpRequestHandler   : GET /chat
/8
Exception in thread "http-nio-8080-Poller" java.lang.OutOfMemoryEr
ror: Java heap space

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExce
ptionHandler in thread "http-nio-8080-Poller"
Exception in thread "http-nio-8080-Acceptor" 2025-11-21T11:52:13.8
27Z ERROR 1907 --- [CS66650_Assignment1] [o-8080-exec-357] o.a.coy
ote.http11.Http11NioProtocol      : Failed to complete processing
of a request

java.lang.OutOfMemoryError: Java heap space
```

Configuration Specifications

Here are the specific tuning parameters and configurations used to achieve the performance targets.

1. Database & Schema Configuration

- Database: Amazon DynamoDB (On-Demand Capacity Mode).
- Table: `ChatMessages`
    - PK: `roomId` | SK: `timestamp#messageId` (Composite Key).
- Indexes:
    - `UserIndex` (GSI): PK `userId` | SK `timestampSk` (For user history).
    - `TimeIndex` (GSI): PK `bucketId` | SK `timestampSk` (For analytics).
- Sharding: `NUM_SHARDS = 5` (Randomly assigned 0-4).

2. Performance Tuning (Batching & Buffering)

- Write-Behind Buffer: `LinkedBlockingQueue` capacity = 50,000 messages (Prevents OOM).
- Batch Size: 500 messages per logical batch (Split internally into DynamoDB's limit of 25).
- Flush Interval: 100ms (Ensures data is persisted quickly even under low load).

3. Thread Pool Isolation:

To prevent resource starvation, we separated workloads into distinct pools:

- Consumer Pool: Managed by Spring AMQP (RabbitMQ listeners).
- DB Writer Pool: Core: 10, Max: 50 threads (Handles I/O waiting).
- Analytics Pool: Core: 5, Max: 10 threads (Handles heavy GSI scans).

4. Resilience & Error Handling

- Circuit Breaker:
    - Threshold: 50% failure rate opens the circuit.
    - Wait Duration: 10 seconds before attempting recovery (Half-Open).
- Retry Policy:
    - Max Attempts: 3.
    - Strategy: Exponential Backoff (100ms multiplier).
- Logging Protection:
    - Buffer Overflow Logging: Rate-limited to once every 5 seconds to prevent CPU death spirals.

5. Caching

- Cache Provider: Caffeine.
- Policy: `AsyncCacheMode` enabled.
- Target: Analytics queries (`getAnalyticsInWindow`) are cached to prevent repeated heavy database scans.