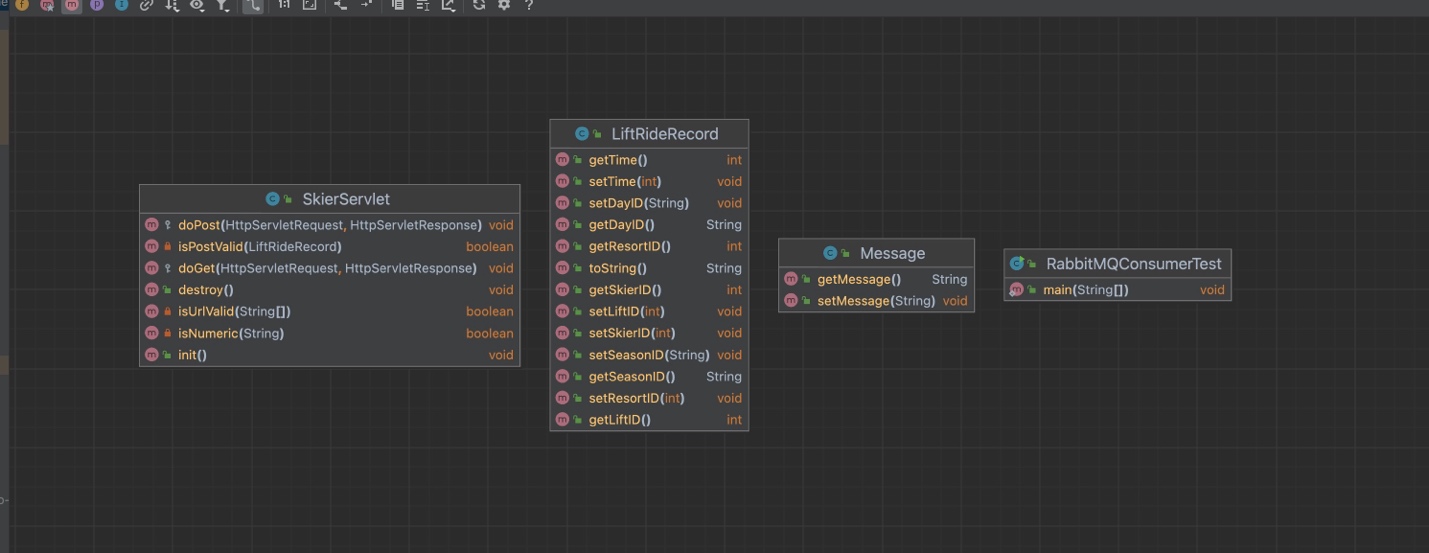
Assignment 3 Zegui Jiang

LIftServer Remain Same with Assignment 2



The server upgraded the functionality for processing messages. Firstly, it connects to RabbitMQ, then it sends the received message queue to RabbitMQ.

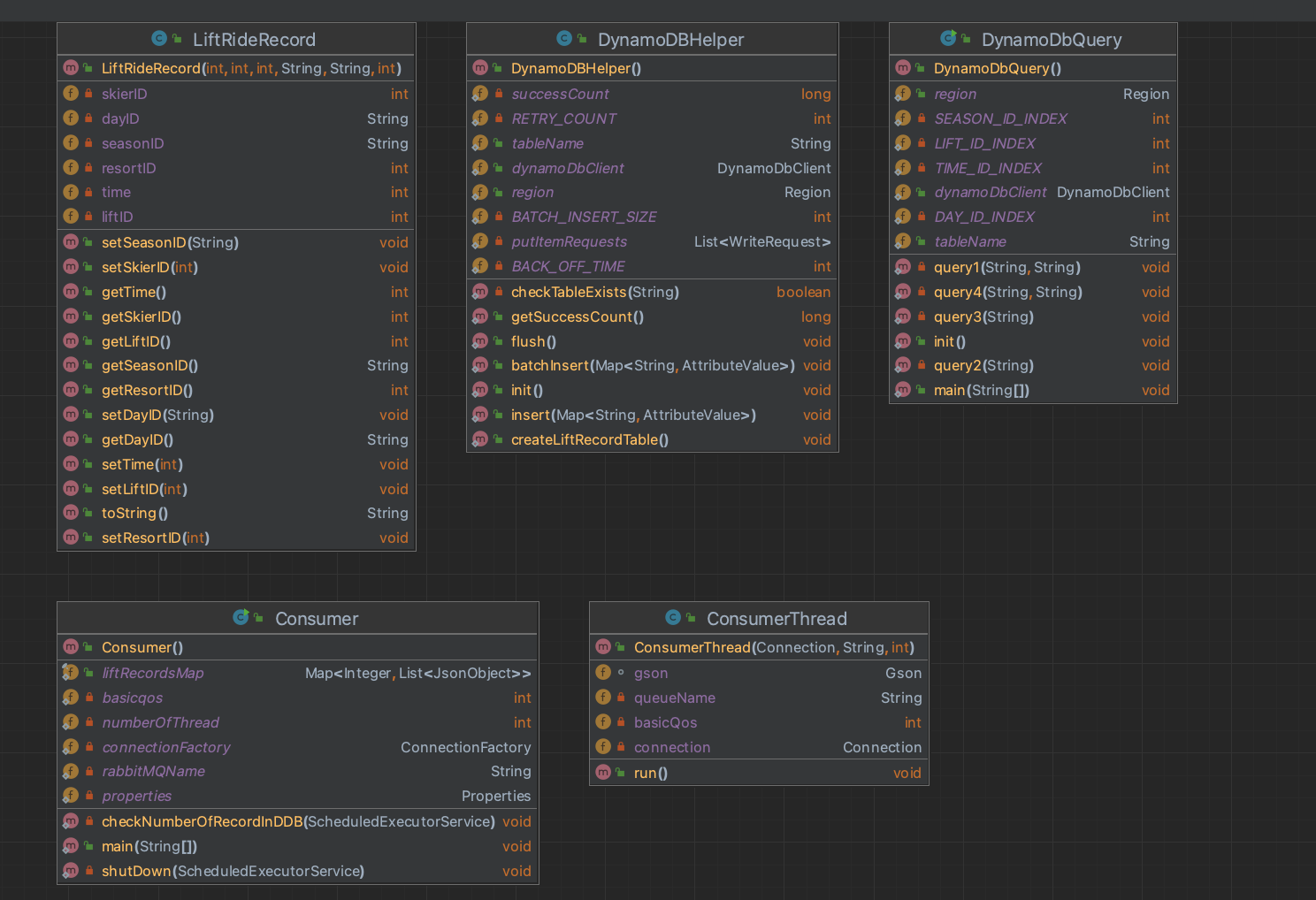
Init(): Reads and configures settings from the **rabbitmq.conf** configuration file found in resources. It extracts the RabbitMQ queue name and the number of channels from the properties file. All channels are stored in a channel pool, which is a Linked Blocking Queue.

Destroy(): Closes all open resources when the process is terminated.

IsPostValid and IsUrlValid: These functions are likely responsible for validating the incoming request's POST data and the URL, respectively.

doPost: Handles incoming POST requests by receiving a message and forwarding it to RabbitMQ.

RabbitMq Lift Record Consumer



The RabbitMQ consumer, building upon Assignment 2, has integrated DynamoDB as the database to store lift ride records.   
  
**DynamoDBHelper**

Init: Sets up the DynamoDbClient with the specified AWS region (Region.US\_WEST\_2). If initialization fails, it prints an error message.

1 - Create table:Checks if the LiftRideRecordTable table exists. If not, it creates the table with a specific schema designed to support queries related to ski lift rides.

This schema includes: Primary key: **skierID** (HASH) and **liftInfo** (RANGE) for identifying records.

**skierID**: skier id is partition key because most of query want to check attributes with different skier id

**LiftInfo**: This is little design in this table. I want to carry most info in to the table. Liftinfo is the concatenateString with symbol “:”, we can decouple when we want these information

Global Secondary Index (GSI): ResortAndDay with resortID (HASH) and dayID (RANGE) to support queries on unique skiers visiting a resort on a specific day. This index includes seasonID and skierID as non-key attributes. The table and index are provisioned with specific read and write capacity units.

Reason to enable GSI with ResortID and DayID, because the query “How many unique skiers visited resort X on day N?” required scan all over resort id, this will result fast scan.

2 - Inserting Records:

Single inserts (insert method): Inserts a single item into the LiftRideRecordTable. If the insertion fails, it prints an error message along with the item details.

Batch inserts (batch Insert and flush methods): Accumulates insert requests and performs a batch insert when the number of accumulated requests reaches a threshold (BATCH\_INSERT\_SIZE). The flush method processes the batch insert request; handling retries with exponential backoff (BACK\_OFF\_TIME) in case of unprocessed items. It keeps track of successfully processed items in success Count.

**DynamoDbQuery**

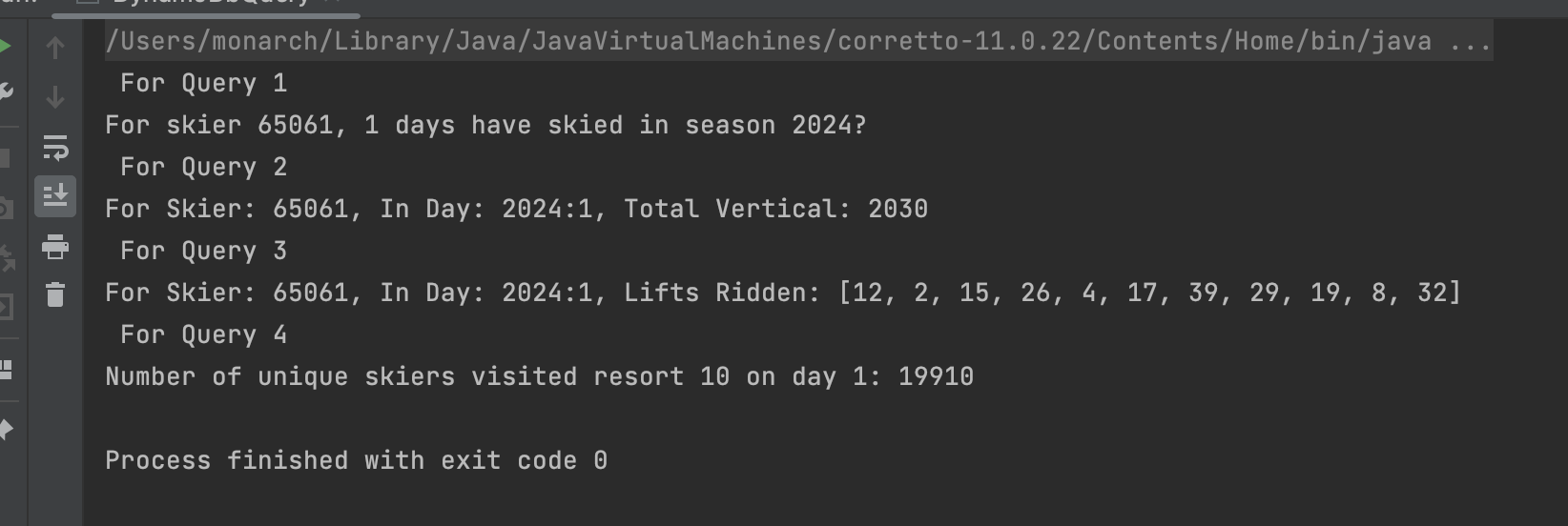
**Since I could not find a good way to use query in AWS console, I choose use java code to implement these four queries:** "For skier N, how many days have they skied this season?"

"For skier N, what are the vertical totals for each ski day?" (calculate vertical as liftID\*10)

"For skier N, show me the lifts they rode on each ski day"

"How many unique skiers visited resort X on day N?"

The result is:

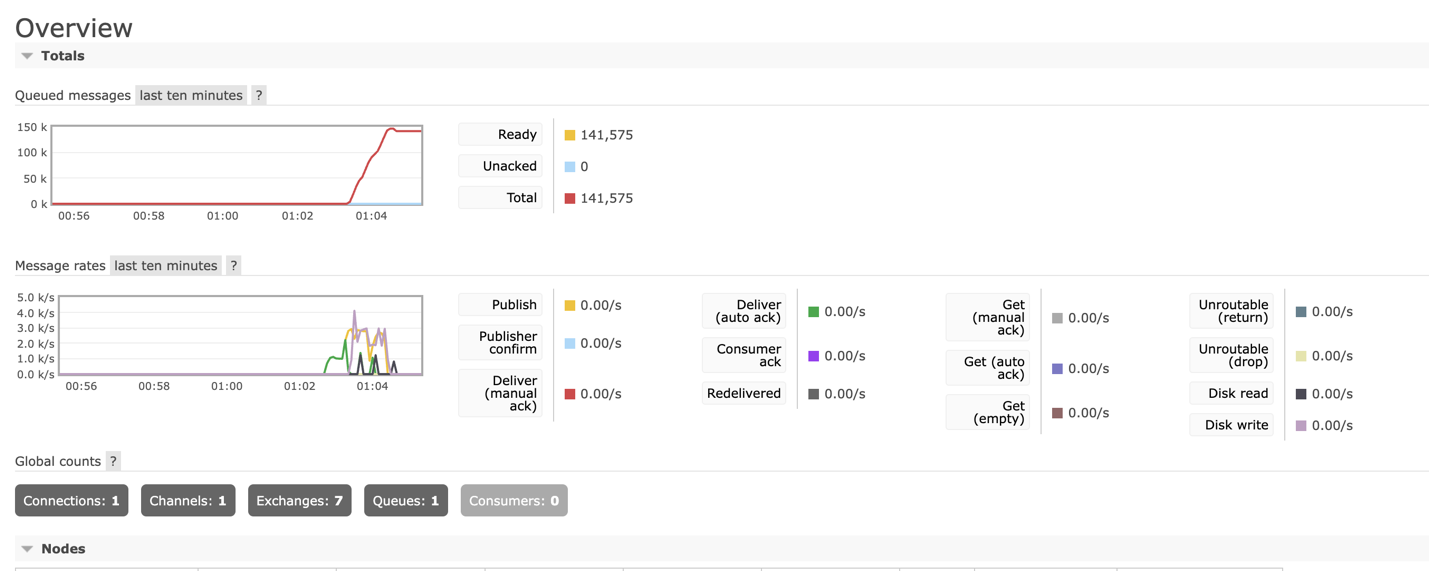


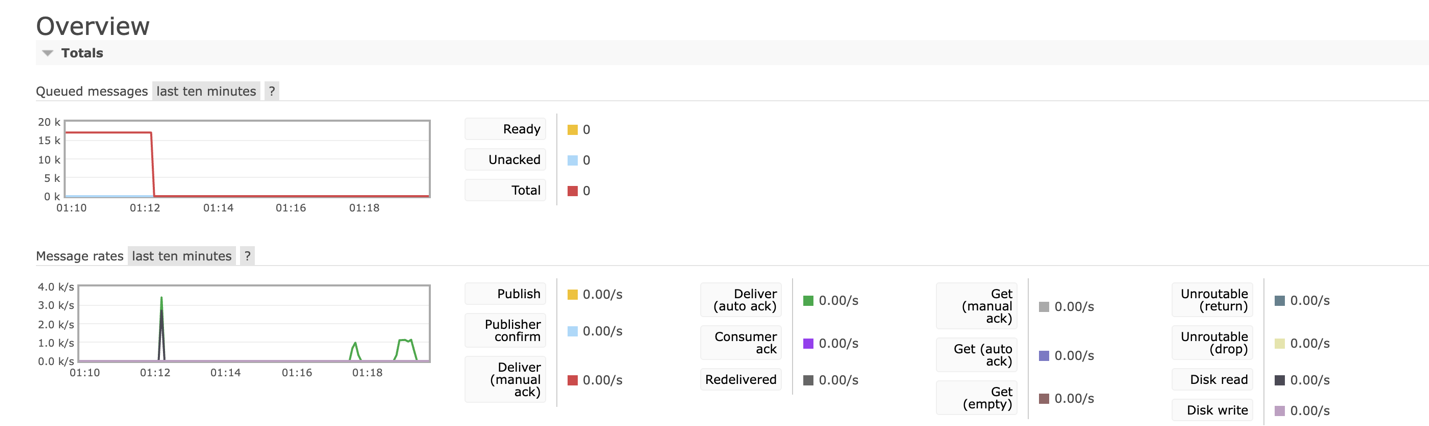
**Challenge on Dynamo DB deployment and problem solving.**

Single insert and batch insert:  
  
When I try using single insert, the speed of writing to the database becomes very slow, and the latency is very high. Therefore, I used batch write logic, which allows writing multiple records at the same time. Moreover, I implemented retry and back-off time logic to enhance stability.

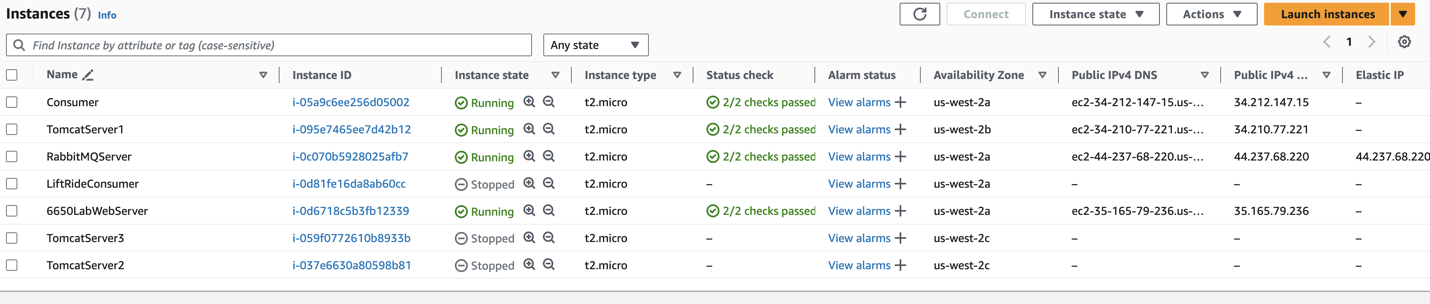
WCU&RCU：

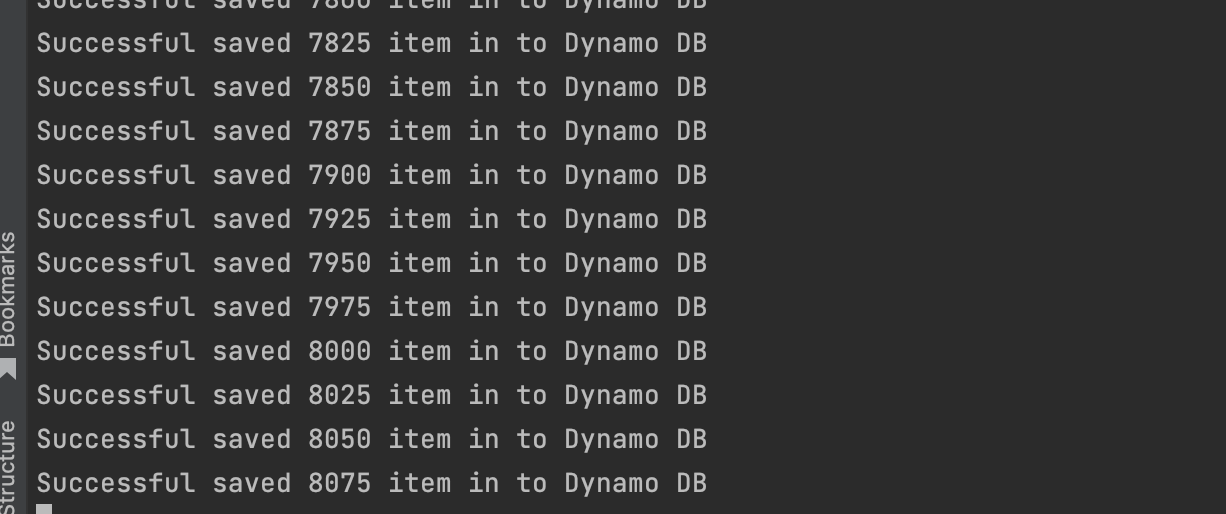
When setting the writeCapacityUnits and readCapacityUnits, I tried many combinations, but none worked well. There would be a lot of backlog in RabbitMQ, and the writing of consumers would also be blocked. Later, I adjusted the WCU to on-demand. DynamoDB will automatically adjust the size and speed of write and read, preventing blockages. RabbitMQ will also be healthier.

Before:  


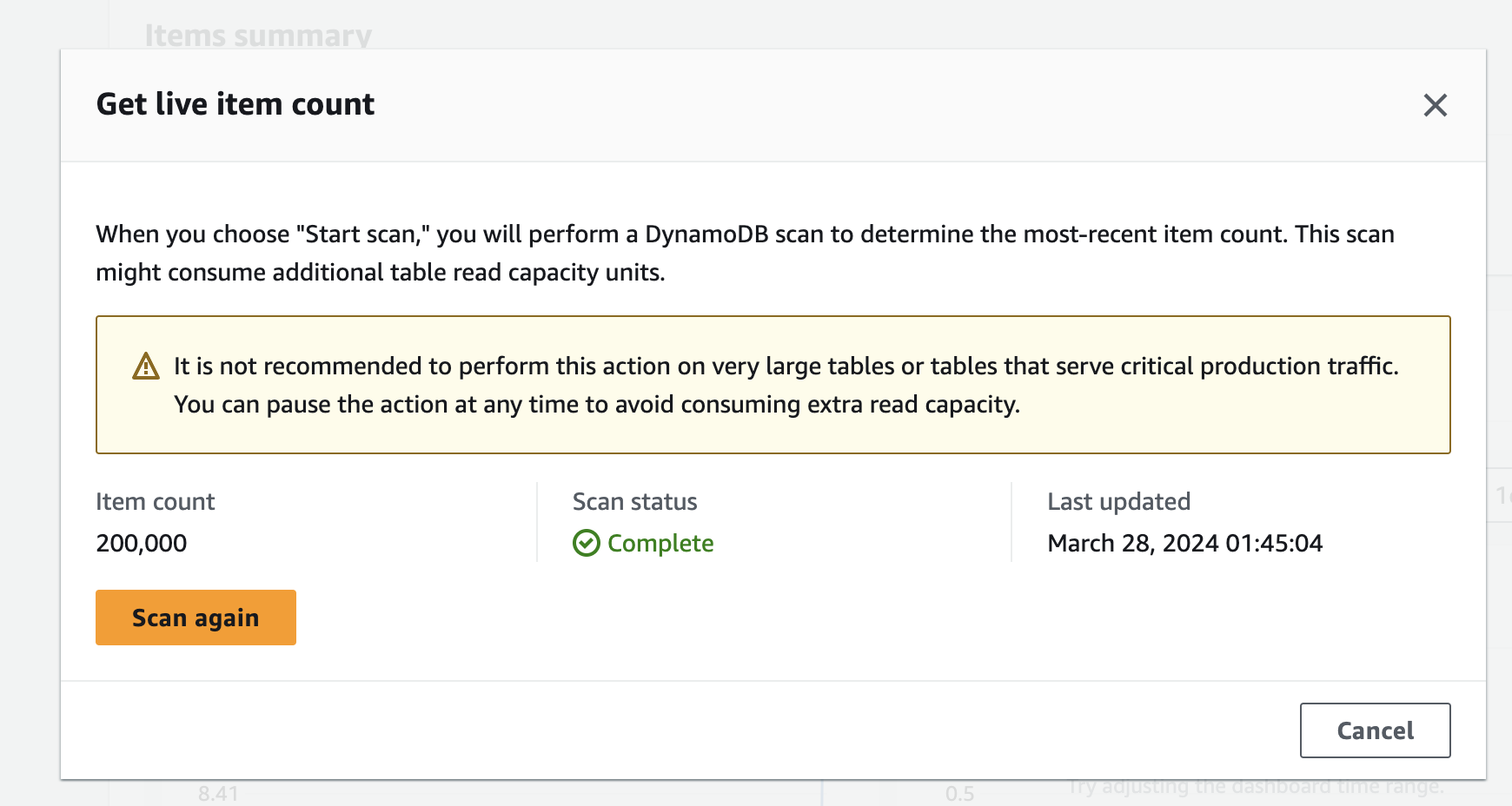
After:  


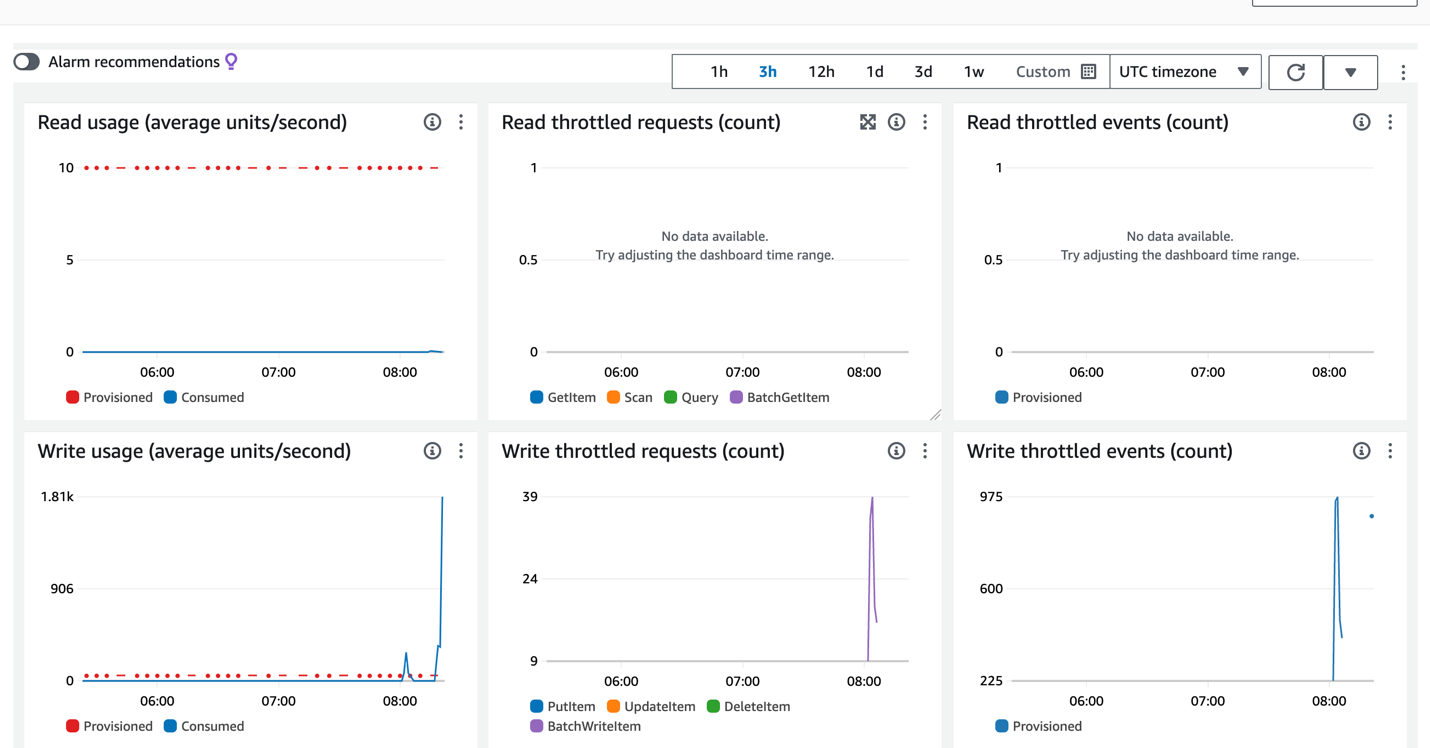
Deployment Topology:

**Client -> TomcatServer -> RabbitMq <- Consumer -> DynamoDB**  


**Saving log:**

**Saving result:**

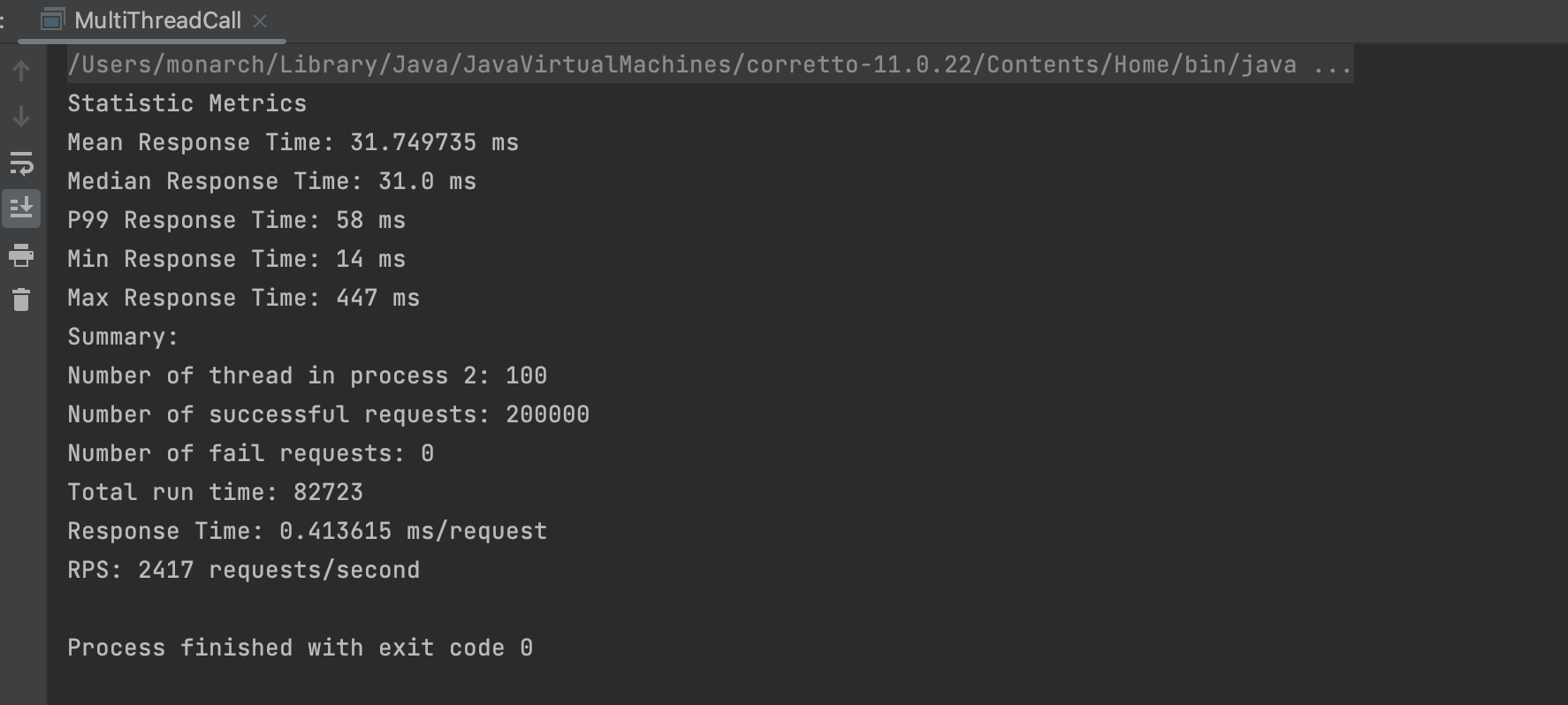


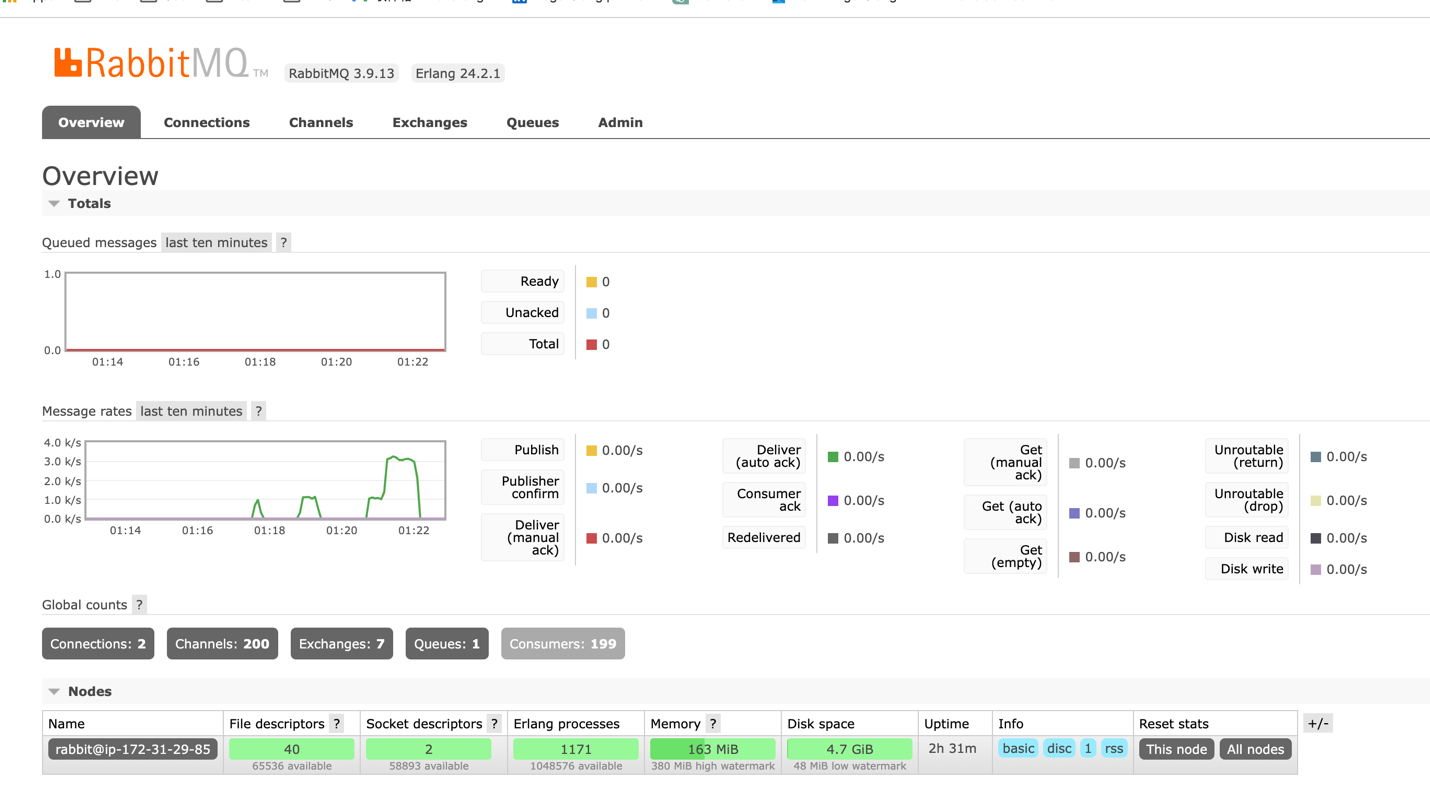


**Client test run and RMQ console screen shots showing your best throughput with ideally short, stable queue lengths.**

**Number of Thread in Client mode 2: 100**

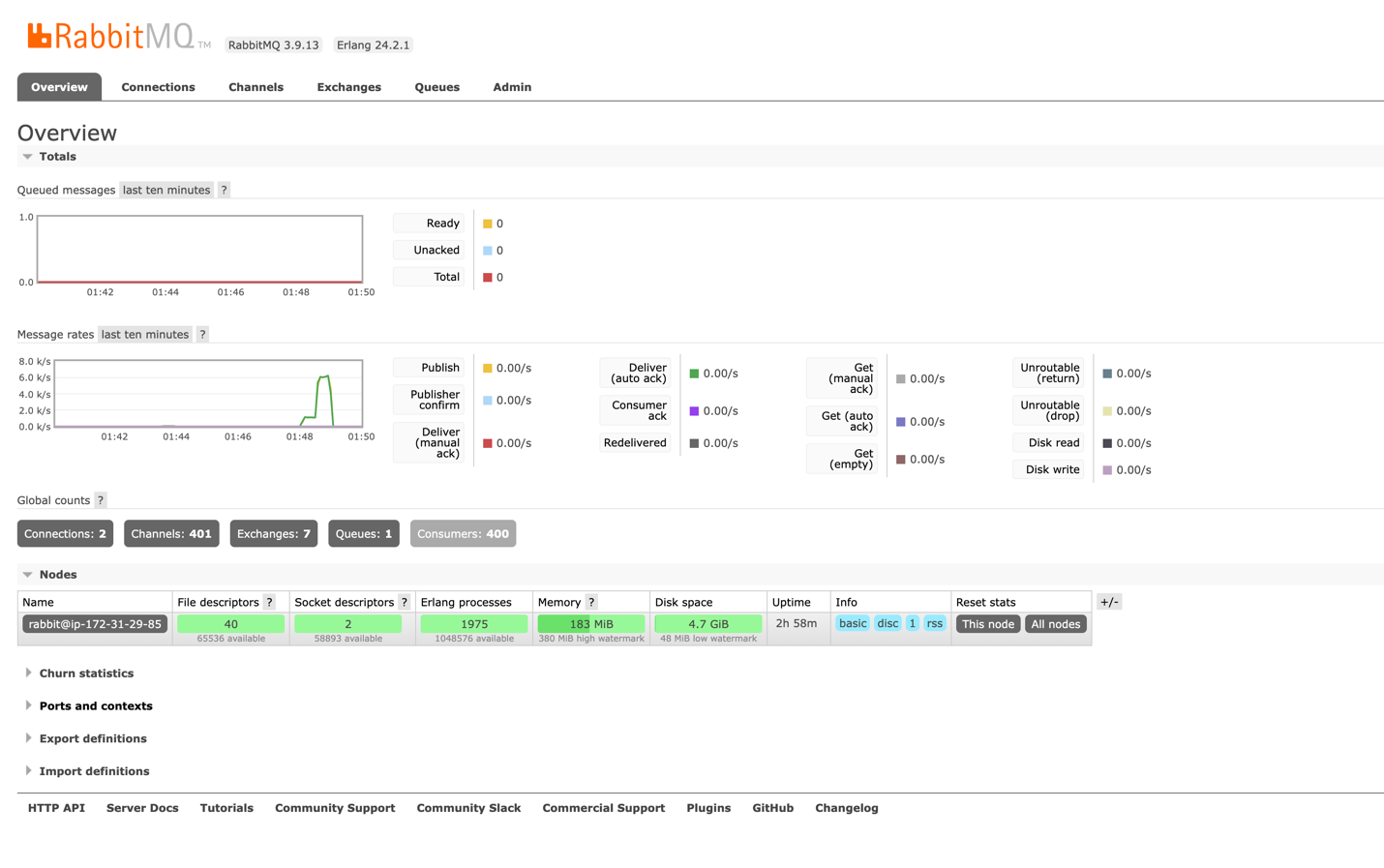
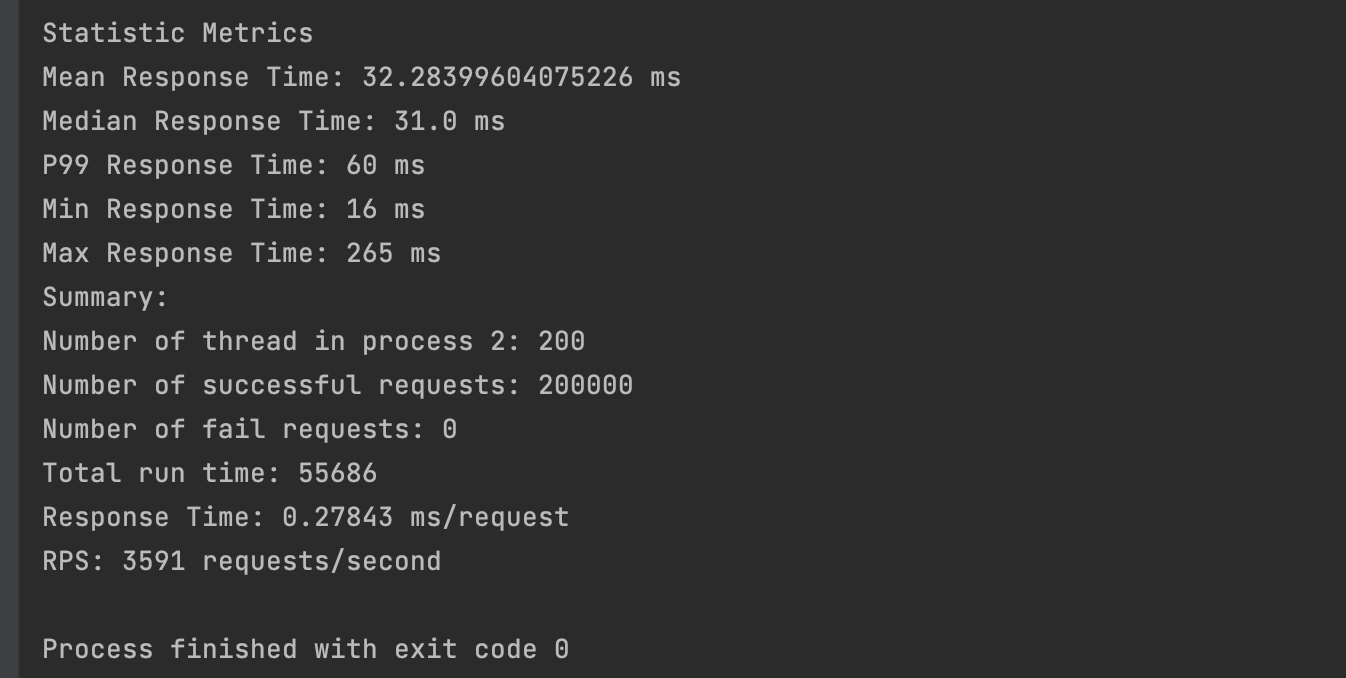
**Number of Thread in Consumer: 400**



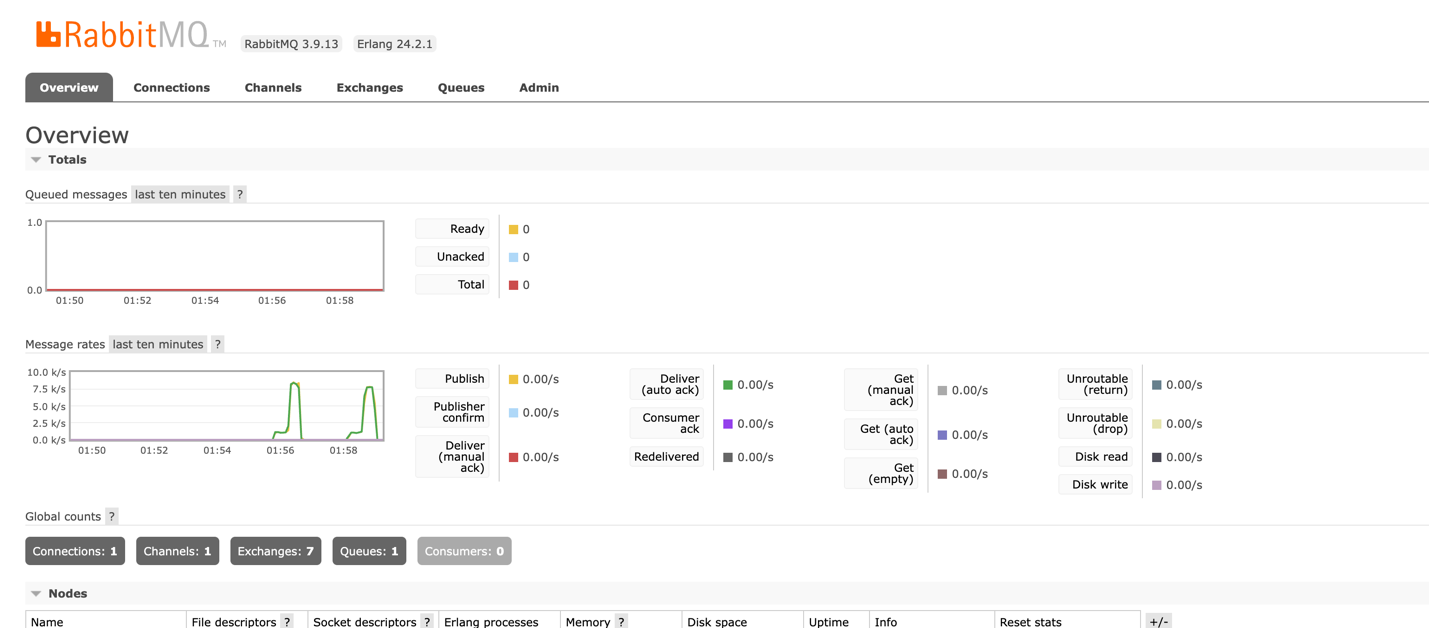
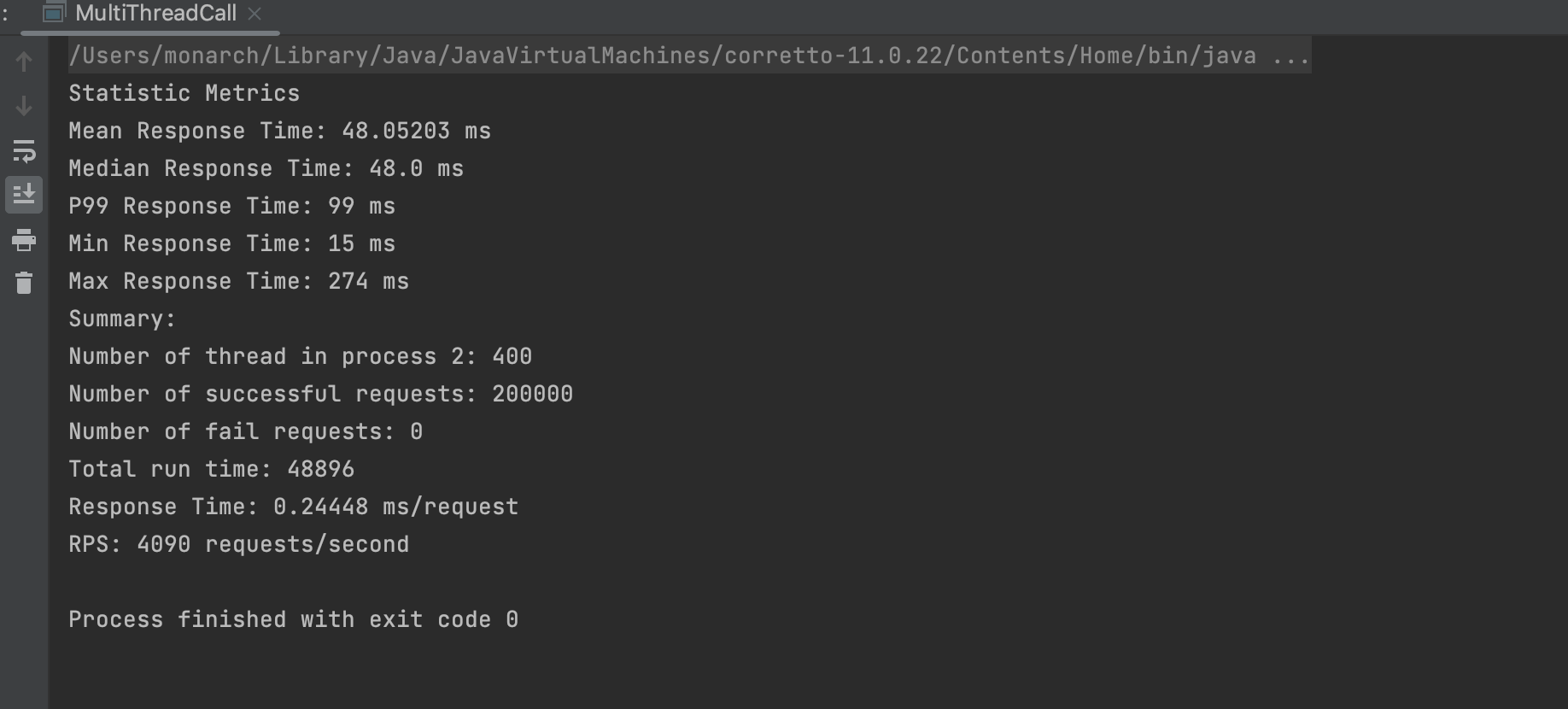


**Number of Thread in Client mode 2: 200**

**Number of Thread in Consumer: 400**

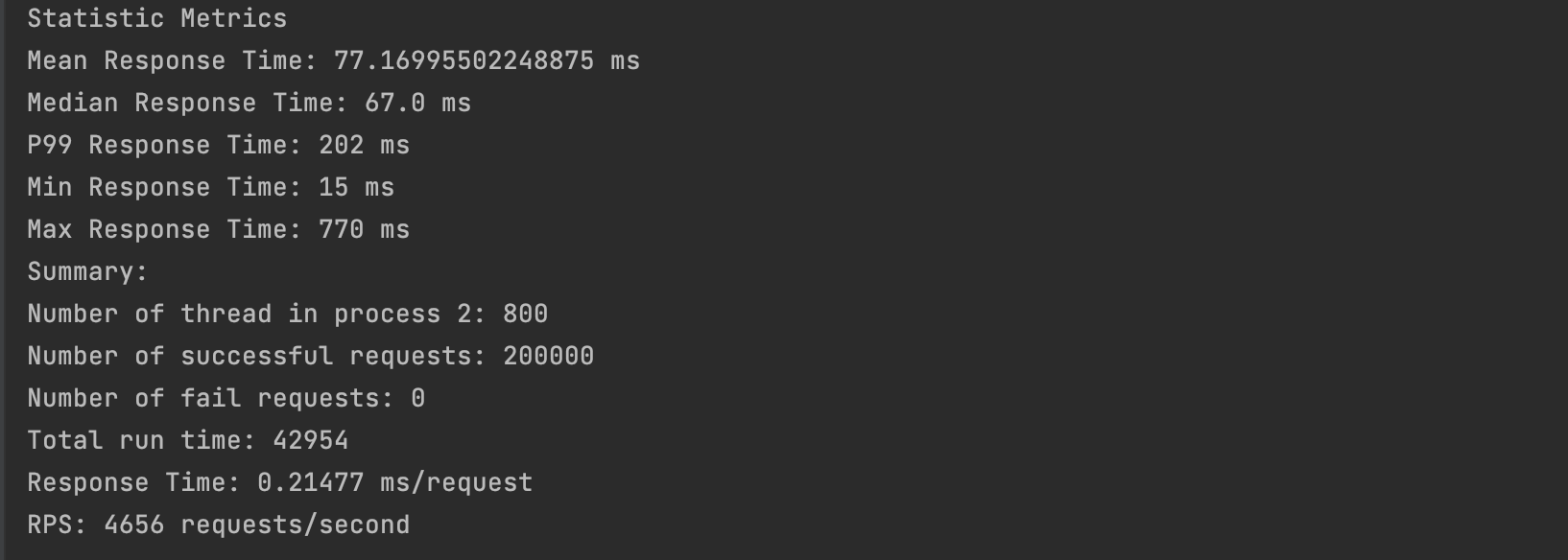


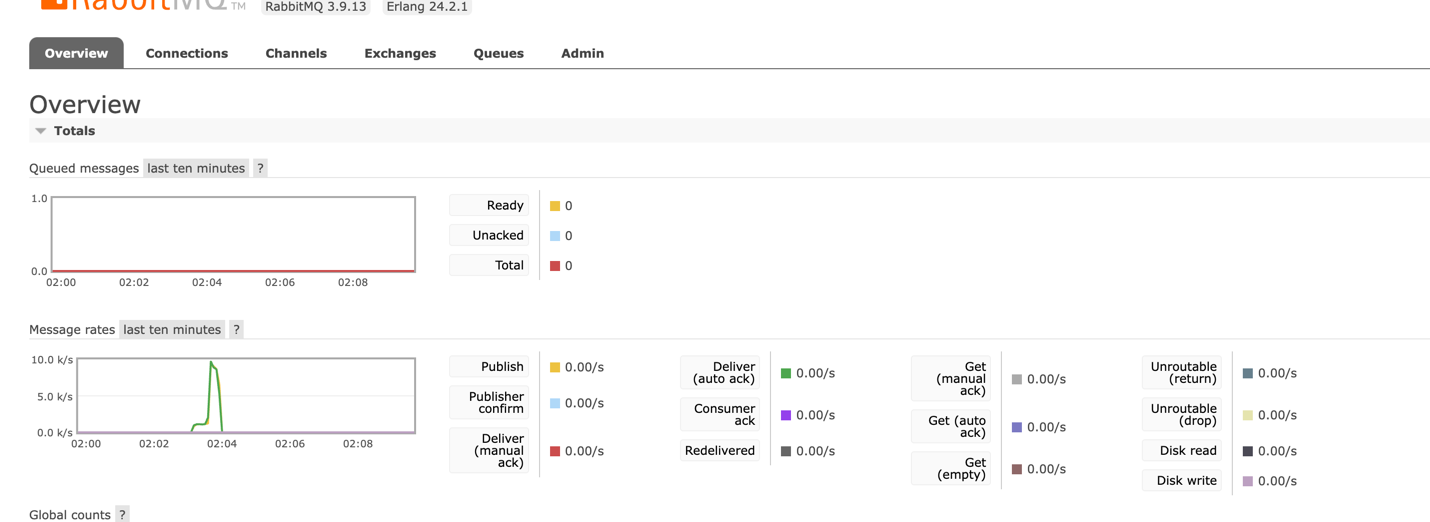
**Number of Thread in Client mode 2: 400**

**Number of Thread in Consumer: 400**

**Number of Thread in Client mode 2: 800**

**Number of Thread in Consumer: 400**





**Number of Thread in Client mode 2: 1000**

**Number of Thread in Consumer: 400**

