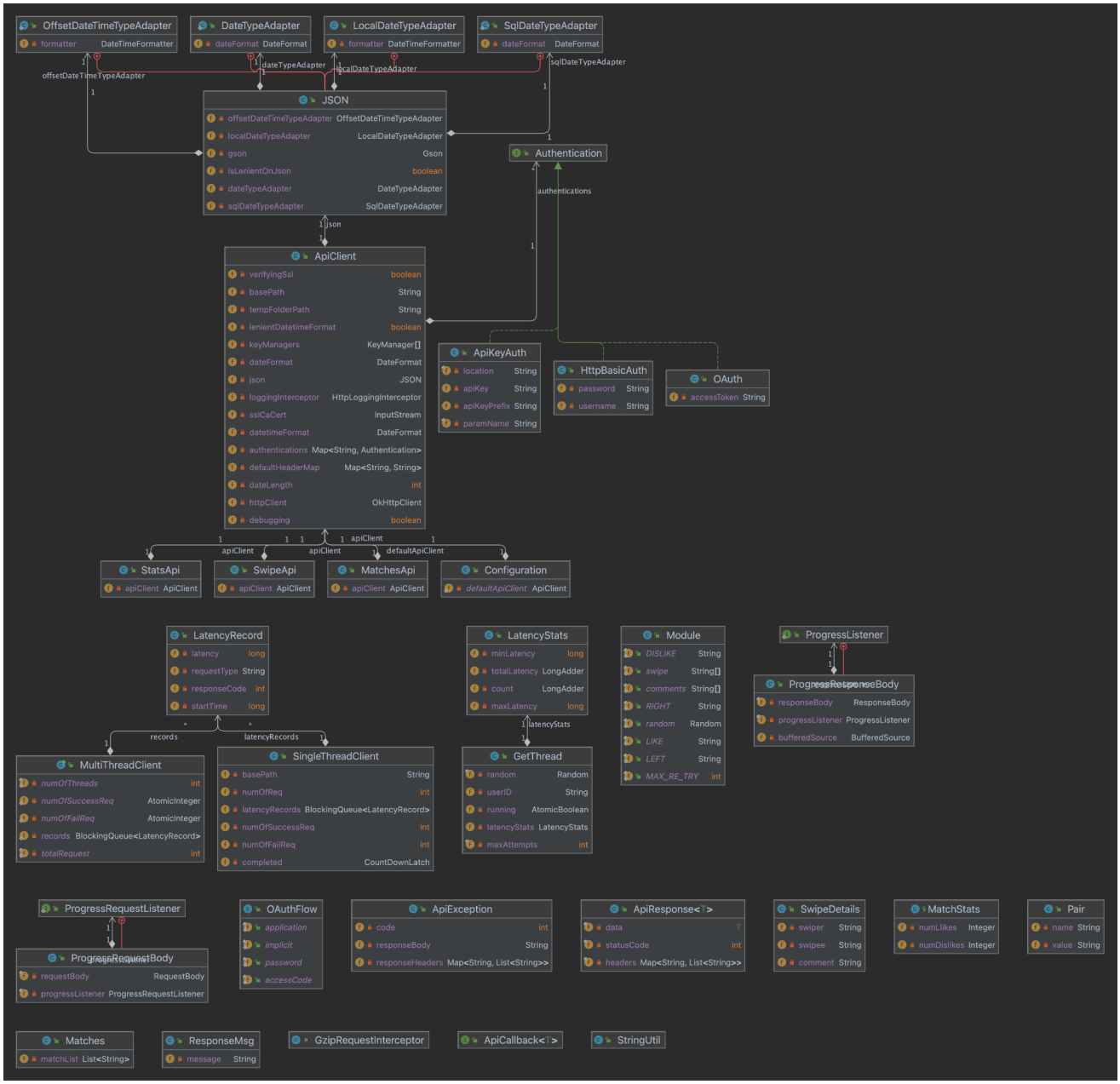
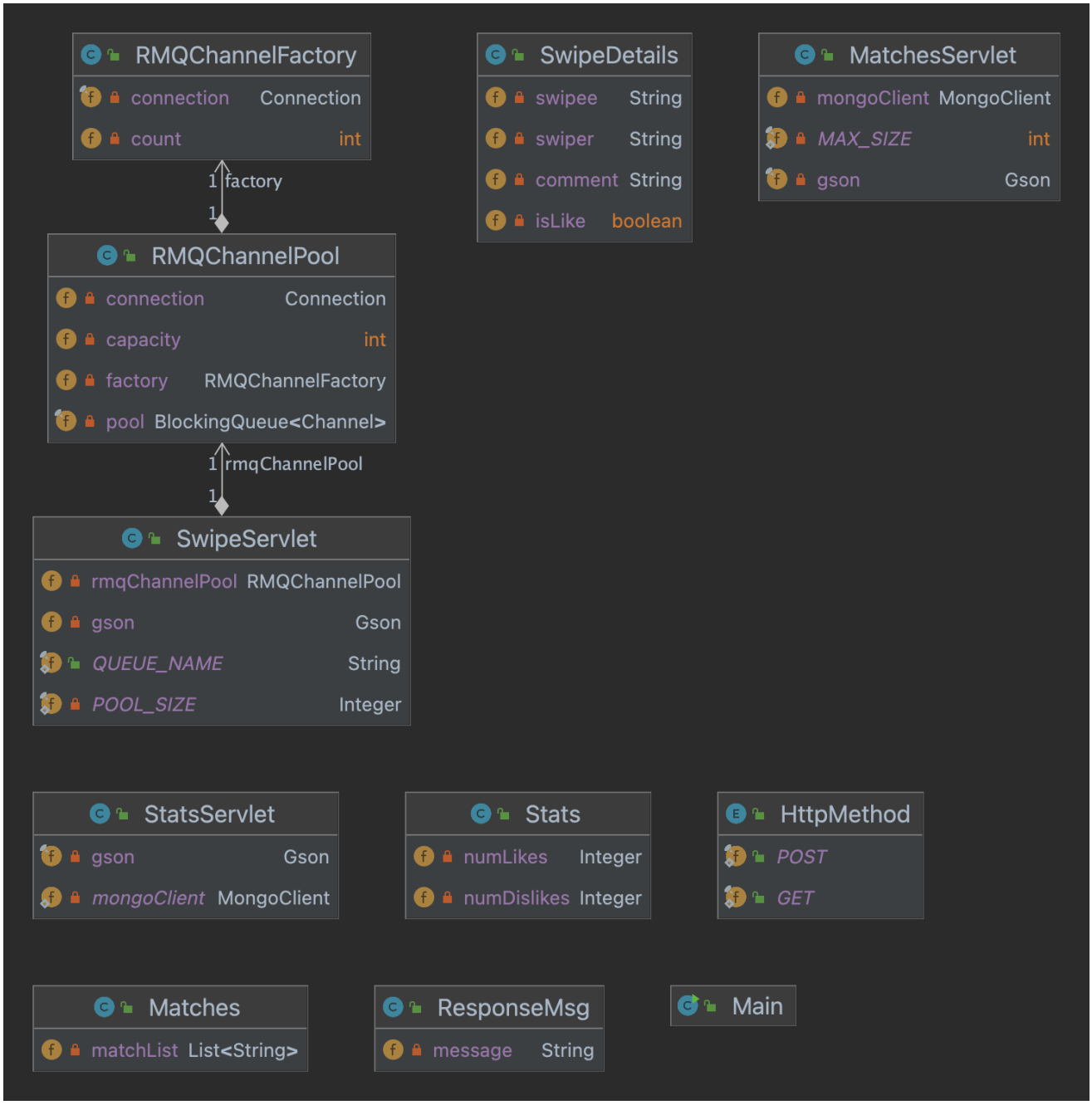
1. Git repo:https://github.com/Oliver1024/DistributedSystemDatingApp/tree/main/assignment3
2. Classes diagram, solution architecture, database design and deployment technology:
Client:



Consumer:


ConsumerRunnable
batchSize int
swipeBatch List<Document>
database MongoDatabase
executor ThreadPoolExecutor
processedMessageCount AtomicInteger
connection Connection
threadPoolSize int
processMatches(ConcurrentLinkedQueue<Document>) void
updateStatsCollection(MongoCollection<Document>, Map<Integer, ir
run() void
flushBatch(List<Document>) void
SwipeDetails
swiper Integer
isLike boolean
swipee Integer
comment String
getComment() String
getSwipee() Integer
getSwiper() Integer
setSwiper(Integer) void
getLike() boolean
setLike(boolean) void
SwipeRecord
swiper Integer
isLike boolean
likeOrDislikeMap Map<Integer, int[]>
listSwipeeRight Map<Integer, Set<Integer>>
listSwiperRight Map<Integer, Set<Integer>>
swipee Integer
addToLikeMap(Integer, Integer, boolean) void
MultiThreadConsumer
NUM_PER_THREADS Integer
mongoClient MongoClient
collectionExists(MongoDatabase, String) boolean
main(String[]) void
Constant
QUEUE_NAME String
NUM_PER_THREADS int
HOST_NAME String

Server:



Solution Architecture:

Client:

GetThread class implements the Runnable interface, with its primary responsibility being to send GET requests to the server. It randomly selects a userID from 1 to 50,000 and then sends a GET request to either the MatchesApi or StatsApi. It records the latency of each request and updates the LatencyStats object.

MultiThreadClient class creates an array of SingleThreadClient instances and launches them concurrently. It also starts a GetThread instance to perform GET requests concurrently with the SingleThreadClient instances. Once all SingleThreadClient instances have finished, the GetThread is stopped, and the program calculates and prints the total throughput, successful requests, failed requests, and latency statistics.

SingleThreadClient class, which also implements the Runnable interface, is responsible for sending POST requests to the server. Each instance of the class sends a predefined number of POST requests, with random data generated using the Module class. It calculates the latency of each request and adds it to the latencyRecords BlockingQueue. The class also tracks the number of successful and failed requests.

Servlet/tomcat server:

MatchesServlet: Handles GET requests to fetch the list of matches for a specific user. It connects to the MongoDB database, queries the "matches" collection to retrieve matchedSwipees, and sends the response as a JSON object.

StatsServlet: Handles GET requests to fetch the number of likes and dislikes for a specific user. It connects to the MongoDB database, queries the "stats" collection to retrieve numLikes and numDislikes, and sends the response as a JSON object.

SwipeServlet: Handles POST requests to submit a new swipe action (left or right) performed by a user. The servlet validates the input data, creates a SwipeDetails object, and sends it as a message to the RabbitMQ messaging system through the "swipe_queue".

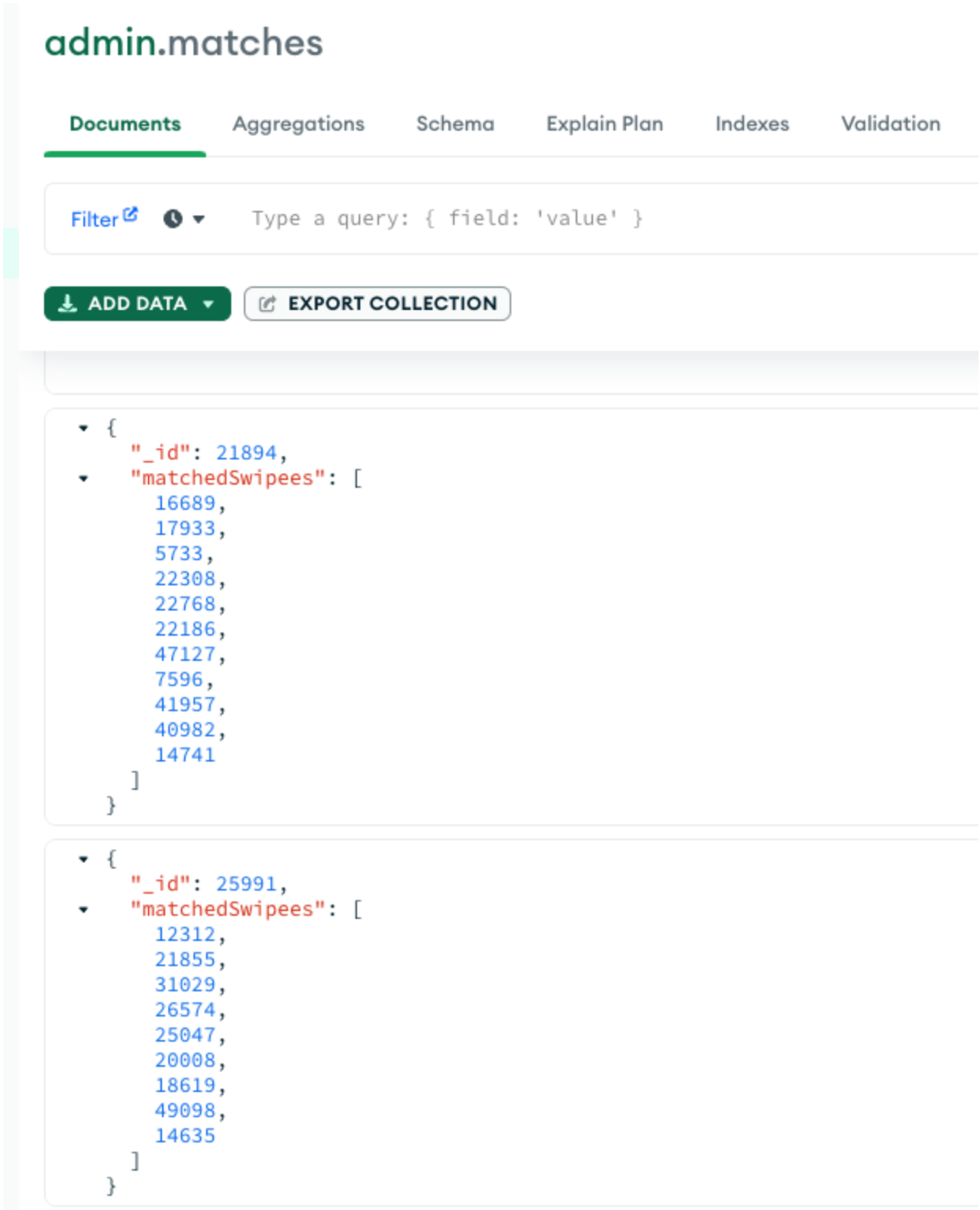
RabbitMQ Message Broker: A message broker that receives swipe data from a producer and distributes it to the consumer and the consumer then writing the data to MongoDB in EC2 instance.

MultiThreadConsumer: A multithreaded consumer application that processes swipe data, processes matches, and updates the MongoDB database.

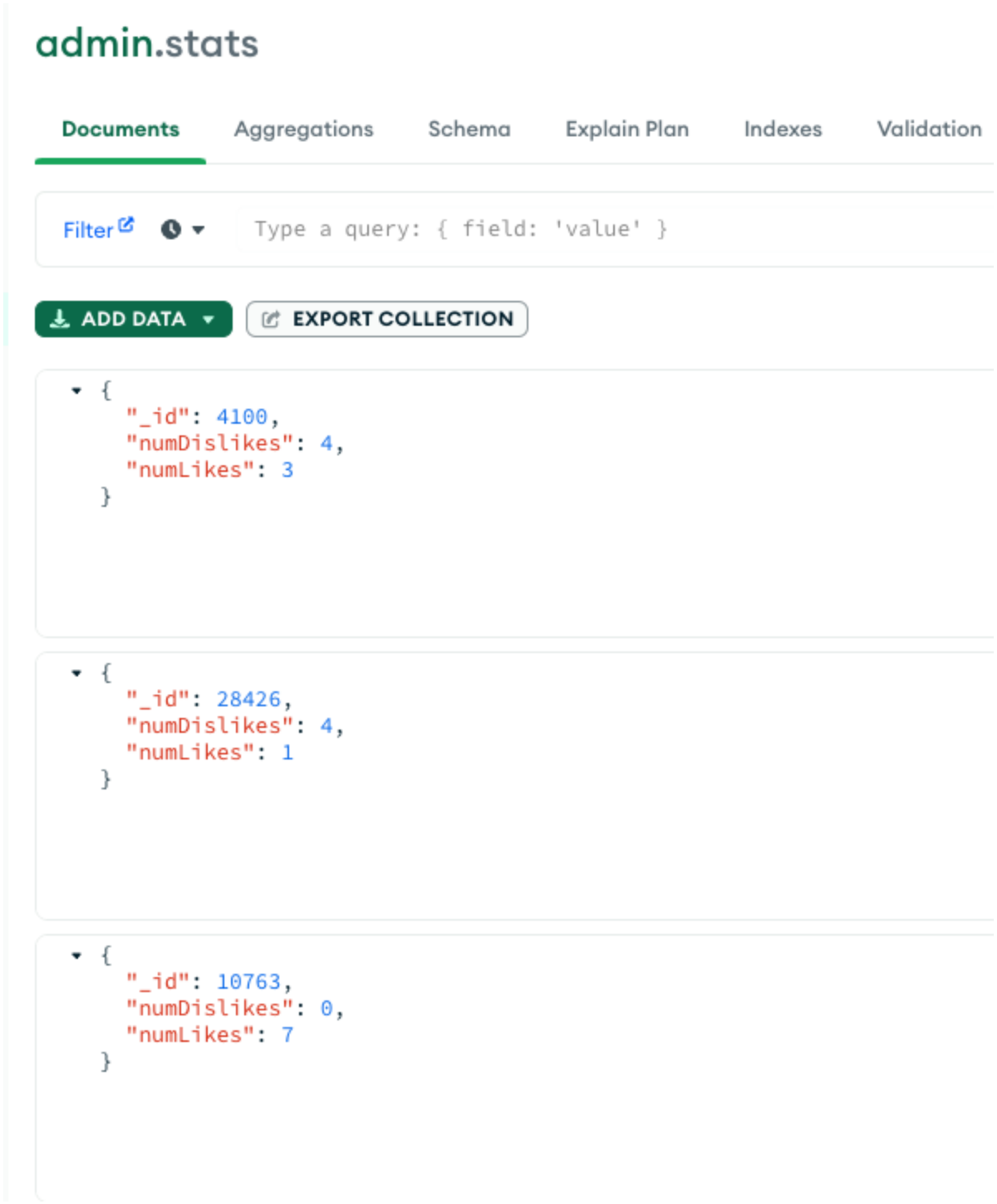
MongoDB: A NoSQL database that stores and manages the swipe data. Like stored the number of likes and dislikes and also for the swiper and swipee matches information.

Database Design:

matches collection: Stores mutual matches between users.
_id: user ID
matchedSwipees: list of mutually matched user IDs



stats collection: Stores the number of likes and dislikes for each user.
_id: user ID
numLikes: number of likes for the user
numDislikes: number of dislikes for the user

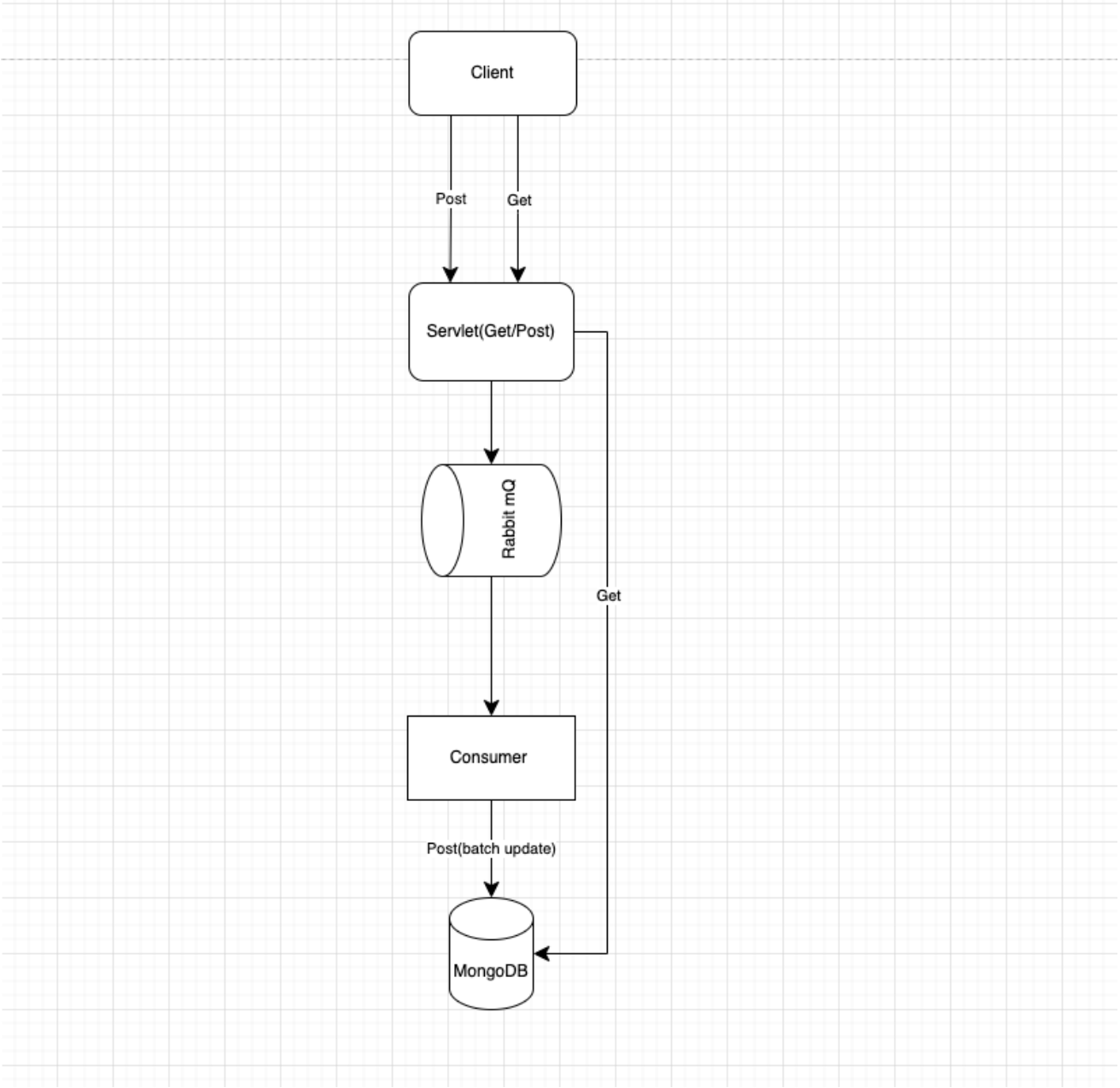


AWS Services used:

EC2: To host the RabbitMQ message broker and the MultiThreadConsumer application and also tomcat server is running in EC2 instances. Also instilled MongoDB in EC2 instance.

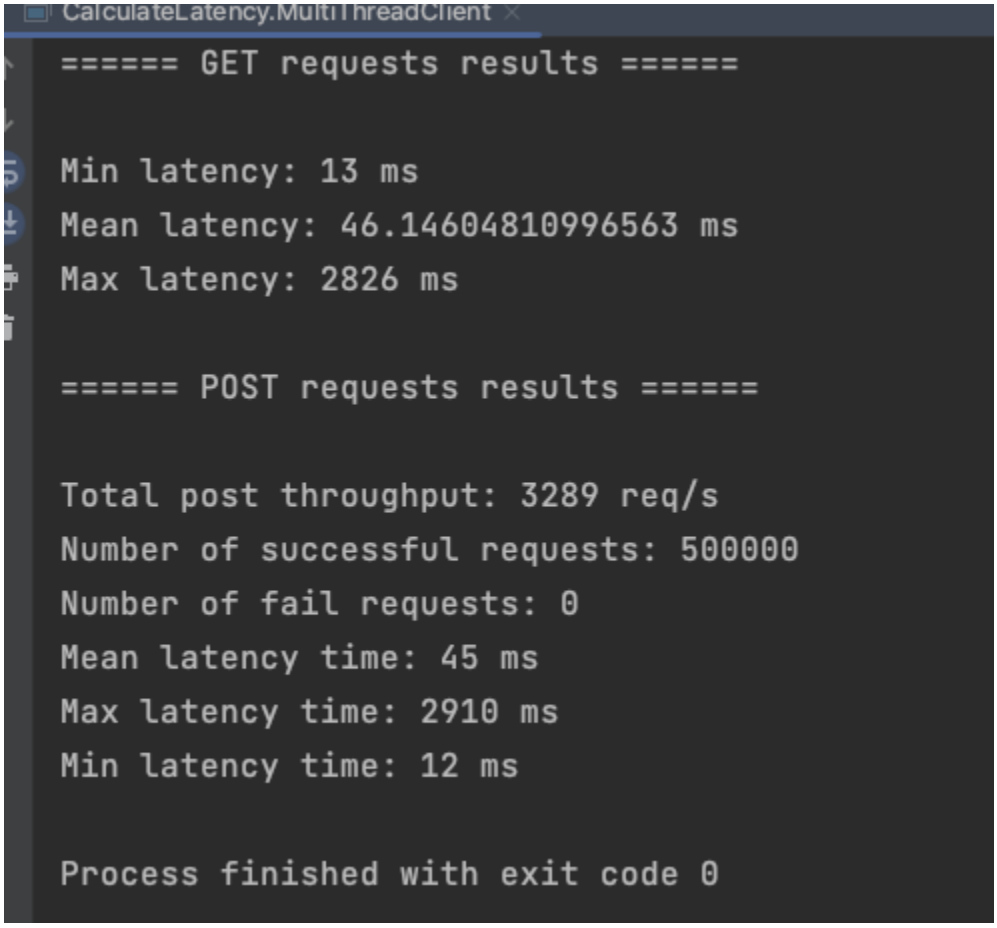
VPC & Security Groups: To define network access and security rules for the RabbitMQ and MultiThreadConsumer instances, Tomcat serverMongoDB instance.

And here's my diagram for the solution architecture.



3. Compare:

A3 screenhosts:



RabbitMQ™ RabbitMQ 3.11.9 Erlang 25.2.3

Overview | Connections | Channels | Exchanges | Queues | Admin

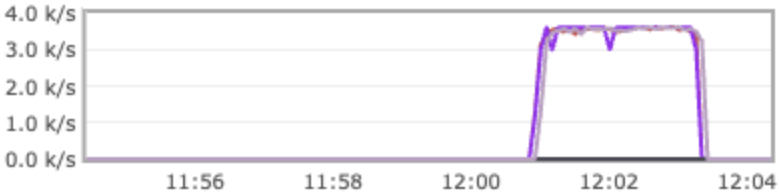
Overview

Totals

Queued messages last ten minutes ?

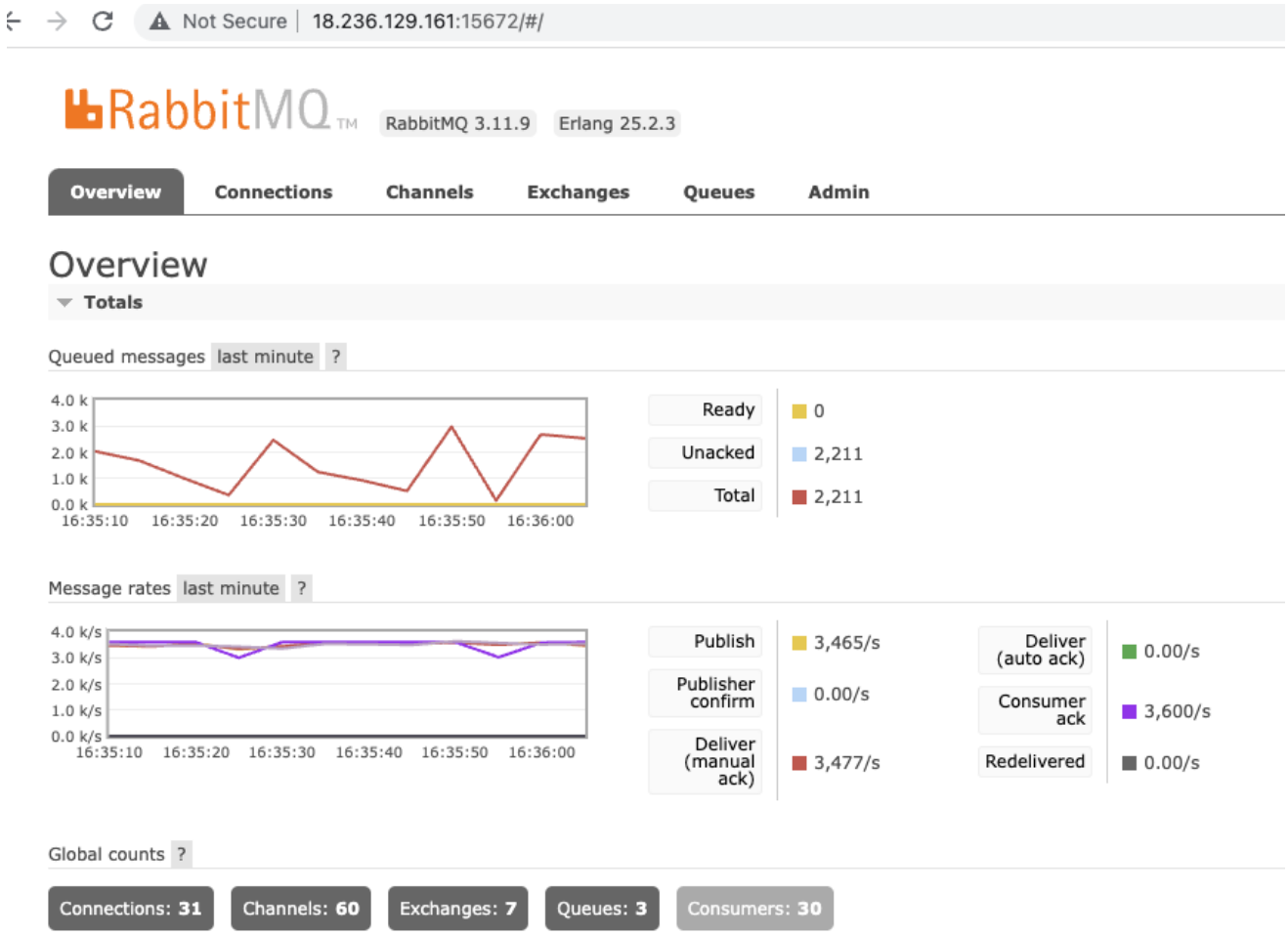
Ready	0
Unacked	2,000
Total	2,000

Message rates last ten minutes ?



Publish	0.00/s
Publisher confirm	0.00/s
Deliver (manual ack)	0.00/s

Global counts ?



A2 screenhosts:(From my A2 report)

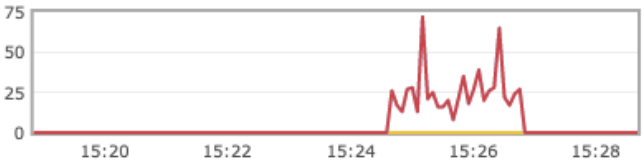
RabbitMQ™ RabbitMQ 3.11.9 Erlang 25.2.3

Overview | Connections | Channels | Exchanges | Queues | Admin

Queue swipe_queue_1

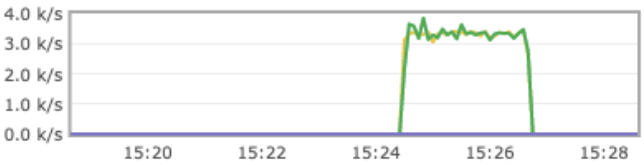
Overview

Queued messages last ten minutes ?



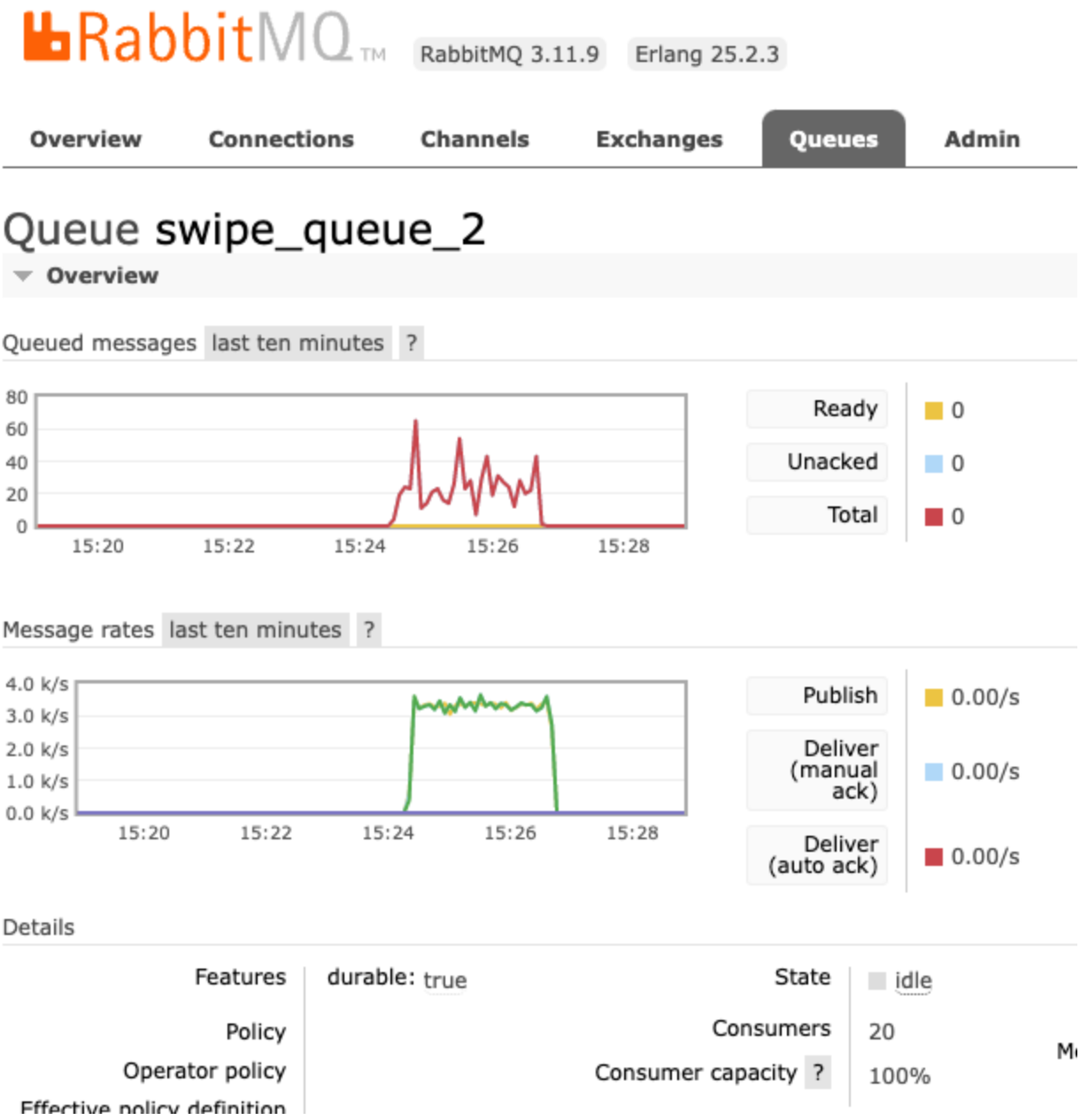
Ready	0
Unacked	0
Total	0

Message rates last ten minutes ?



Publish	0.00/s	Consumer ack
Deliver (manual ack)	0.00/s	Redelivered
Deliver (auto ack)	0.00/s	Get (manual ack)

Details



From the comparison, we can observe that the throughput for A2 and A3 are quite close. Similarly, the RabbitMQ screenshots show comparable results for the publish and consumer rate ack, with A2 around 3300 and A3 approximately 3250. For A2, I only printed the throughput and didn't record the latency. However, considering that the A3 latency is quite small and the throughput is very close to A2, it is reasonable to assume that the latencies for both A2 and A3 are nearly the same.

To achieve this level of performance, I conducted multiple trials with different configurations and varying numbers of threads for the client, Tomcat server, consumer, and database connections for the consumer. Initially, without batch writing, the data writing to the database was slow. However, after implementing

batch writing and adjusting the batch size, I was able to optimize the performance and achieve the current results.