PCD - Homework 2

First component (microservice) - RDS & EC2

First microservice is responsible with CRUD operations for transactions.

The RDS Cloud Service was employed to store transaction information within a MySQL database. This database is linked to an EC2 instance hosting the microservice responsible for performing CRUD operations.

RDS Cloud Service: Managed MySQL database storage offering scalability, high availability, and automatic backups. Ideal for storing transaction data efficiently.

EC2 Instances: Flexible compute resources for hosting microservices, with options for easy scalability, cost-effectiveness, and compatibility with various application architectures.

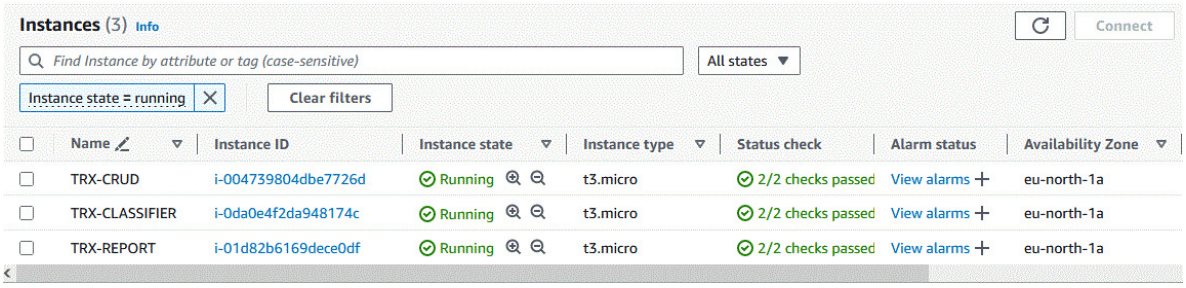

Figure 1: Deployed microservices

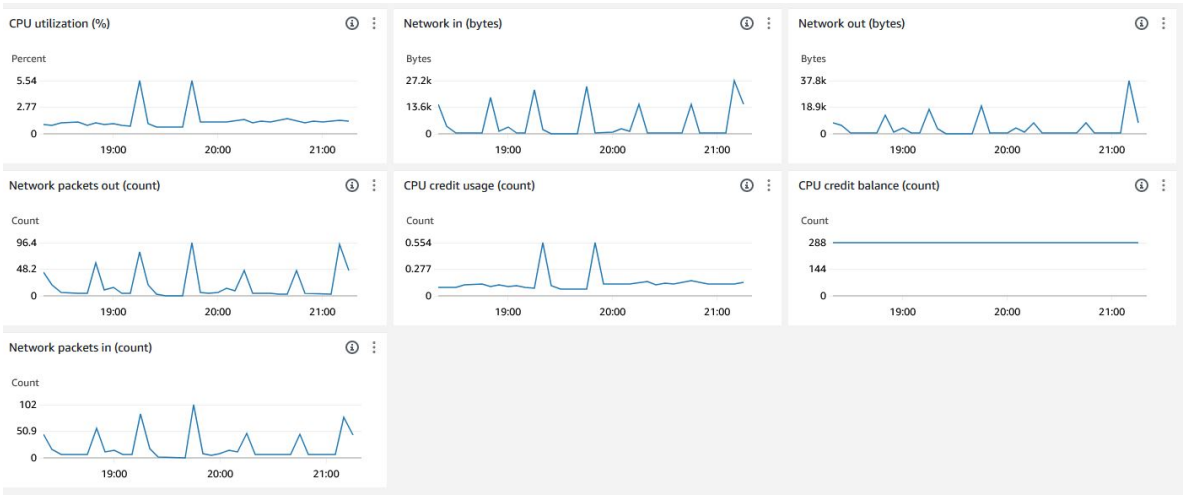

Figure 2: TRX-CRUD, Real Time Metrics

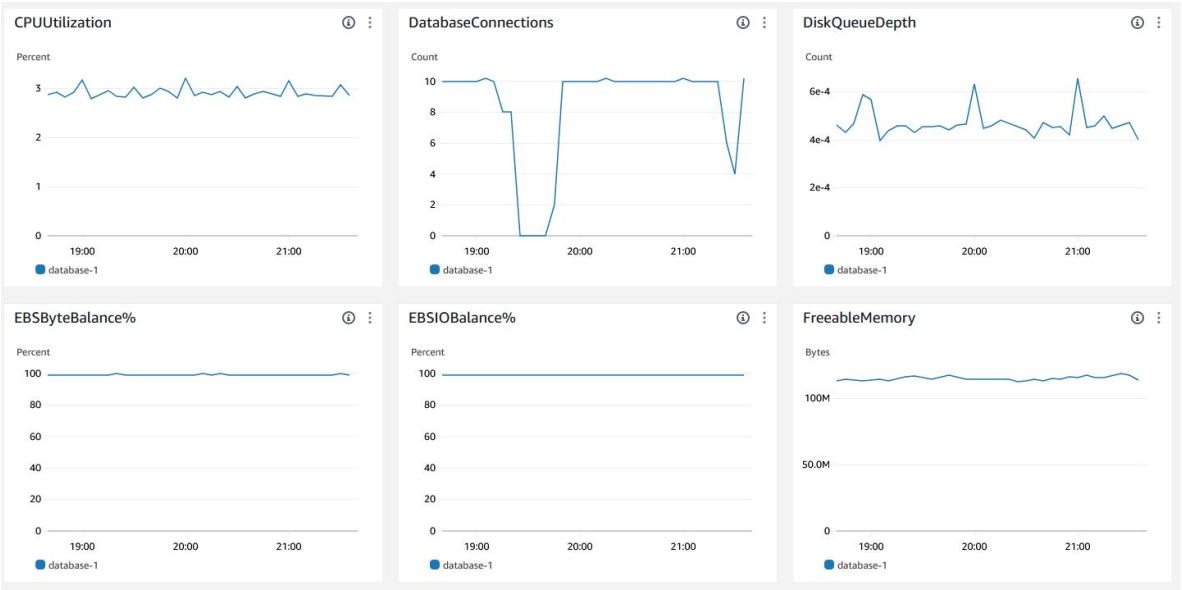

Figure 3: TRX-CRUD Database (RDS), Real Time Metrics

Second component (microservice) - API Gateway & Lambda Function & EC2

The second microservice is tasked with categorizing transactions. This categorization process can occur automatically based on transaction descriptions or manually using transaction IDs and category IDs.

The second microservice is hosted on a dedicated EC2 instance and offers endpoints for both manual and automatic categorization of transactions. However, for demo purposes, only manual categorization is currently available. Its functionality is initiated by calling an endpoint from the API Gateway, which in turn invokes the exposed endpoint of the microservice. The response is then returned to the caller, completing the transaction categorization process.

API Gateway: Fully managed service facilitating seamless creation, publication, and management of APIs. Offers robust security, traffic management, and monitoring features, simplifying integration and communication between applications and services.

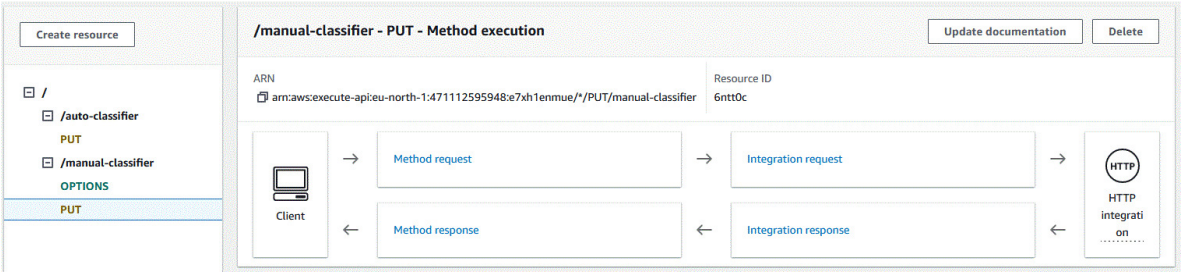

Figure 4: Manual Categorization

Third component (FaaS) - Lambda Function

The third component is a Function-as-a-Service (FaaS) platform that accepts a list of transactions as input and calculates both the total amount spent and the total amount received.

Lambda: Serverless computing service providing automatic scaling, pay-per-use pricing, and multi-language support. Perfect for event-driven and microservices architectures.

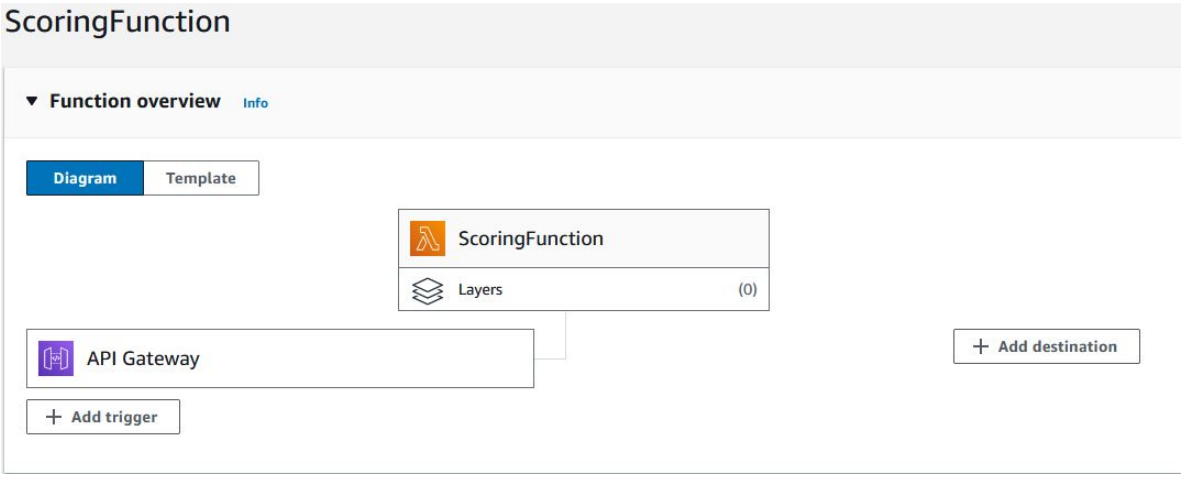

Figure 5: Scoring Lambda Function

Figure 6: Scoring Component, Real Time Metrics

Notification system

Implementation

This component represents a broadcast application that sends financial notifications to all the users connected. The application sends real-time notifications, without the clients making a specific request, through the use of WebSocket API. The WebSocket was used with the AWS API Gateway which has three main routes.

The *connect route* establishes the connection for the client and uses a Lambda function that adds the `connectionId` into the connections table. The *disconnect route* disconnects the client by calling the second Lambda function, which will erase the corresponding `connectionId` from the DynamoDB table. The *send notification route* will deal with the broadcasting operation: the stocks table can be

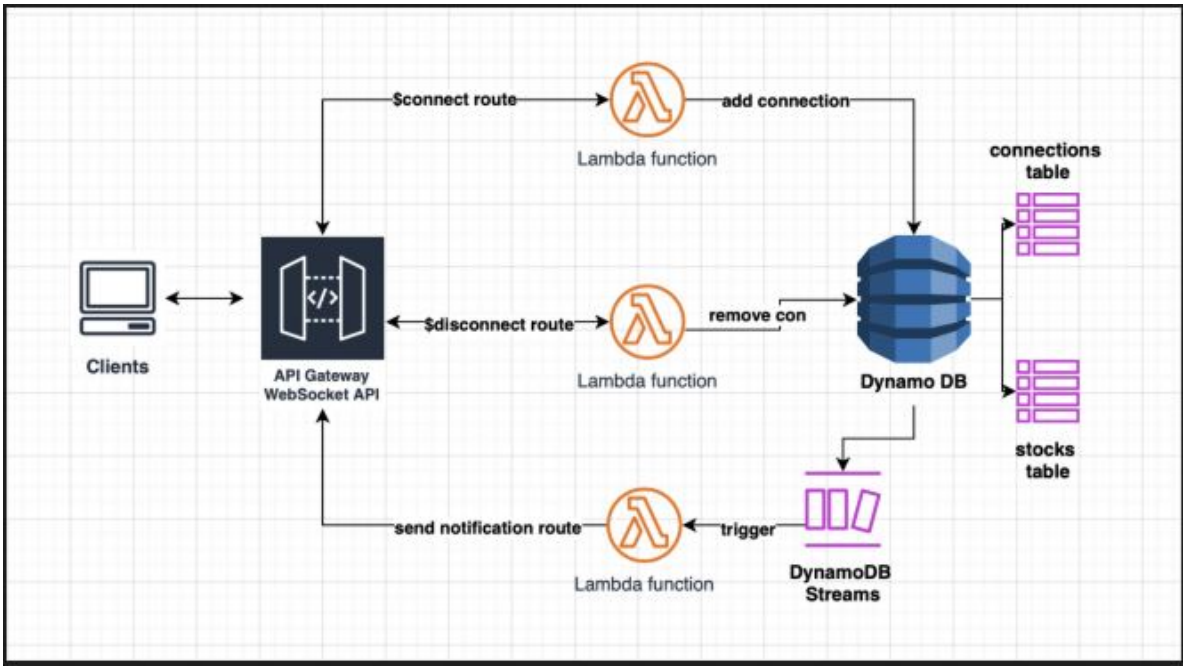

Figure 7: Component architecture

updated with new stocks, or the stock value can be modified. Through a DynamoDB stream, the modifications made to the table will trigger the `sendNotificationHandler` Lambda that will take the event it received from the stream and check which changes have been made.

The current implementation checks if the received records are either an insert operation or an update: because the stream was set to get the old and new images, the function can check what is the percent in which the values have been modified and write it in a JSON that will send to all the clients by iterating through the connections table.

For the third function, the `sendNotification` Lambda, its role had to be modified in order for it to read the records of the stream by updating the policy of the Lambda role to contain `"dynamodb:DescribeStream"` and `"dynamodb:GetRecords"` actions to the resource.

Considerations

DynamoDB

As a NoSQL database, DynamoDB offers high performance and scalability, which are essential for handling real-time data and a potentially large number of connections and notifications. Its fully managed service simplifies operational tasks like hardware provisioning, setup, configuration, replication, and scaling. The use of DynamoDB Streams to trigger Lambda functions ensures immediate processing of data changes, facilitating real-time notifications without polling the database.

AWS Lambda

Lambda functions provide a serverless execution environment, allowing to run code in response to triggers such as requests via API Gateway or database changes via DynamoDB Streams. This serverless approach is cost-effective since I pay only for the compute time I consume, and it automatically scales my application by running code in response to each trigger. By using Lambda for the connection, disconnection, and send notification functionalities, I ensure that my application can handle varying loads efficiently.

API Gateway with WebSocket support

API Gateway acts as a front door to manage incoming WebSocket connections and route messages to the appropriate backend services like Lambda functions. Its WebSocket support is crucial for establishing a persistent, full-duplex communication channel between the clients and the server, which is essential for real-time applications. API Gateway seamlessly integrates with other AWS services like Lambda and DynamoDB, simplifying the architecture and reducing the need for additional infrastructure management.

Metrics

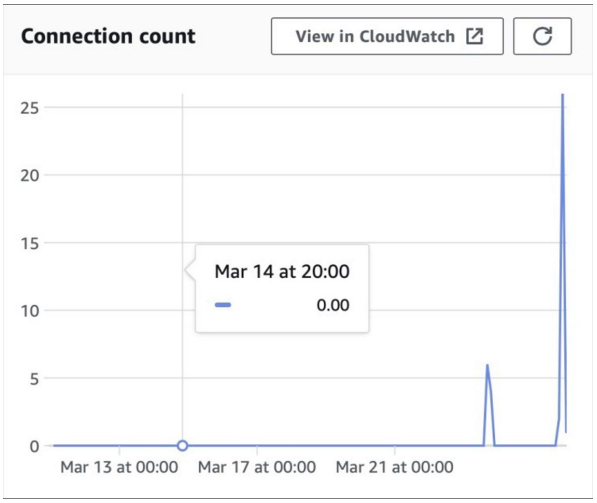

Figure 8: Connection Count

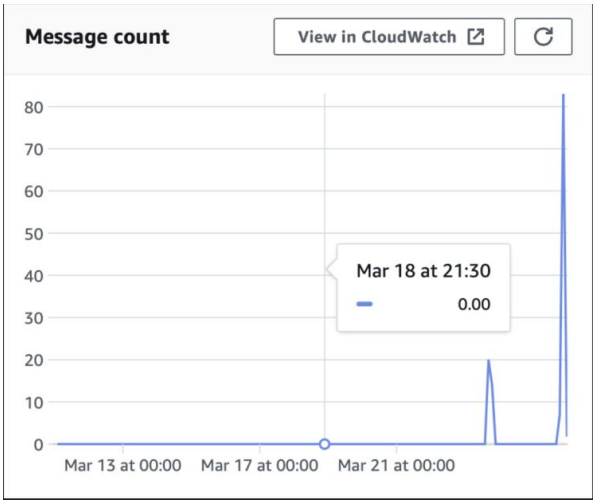

Figure 9: Integration Latency

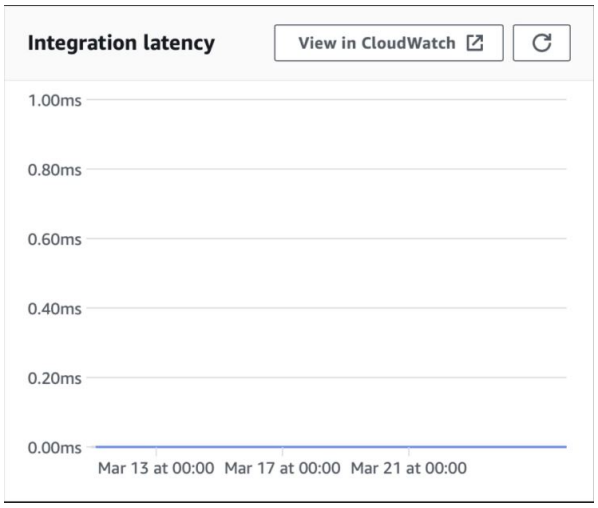

Figure 10: Message Count