Cupcake Order & Delivery Service

IS213 Enterprise Solution Development

Assignment

G3-T8

HILYA SYAZWANI BINTE MOHAMED YUSOFF

GOH KIM HUI ASHTON

LIEW JUIN NENG IAN

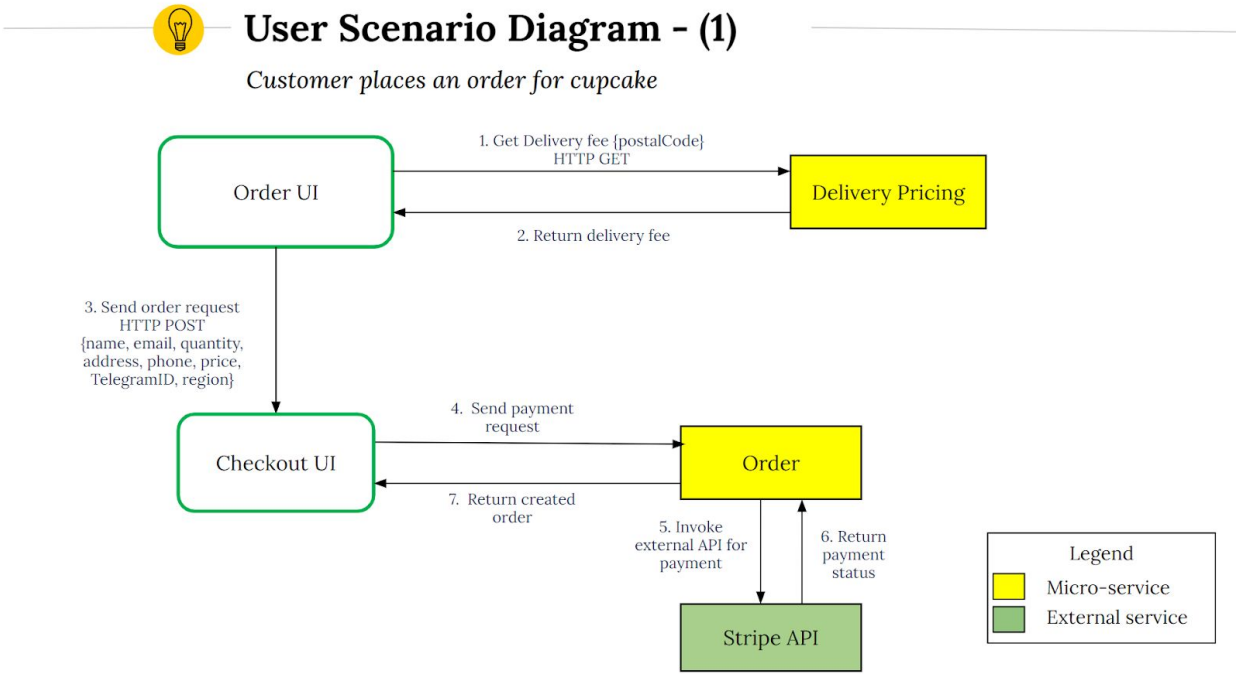
PHAM THU THUY

TAN TZE EN SUE-ANNE

1. Introduction

For the business scenario that we have chosen, we are building an enterprise solution for a cupcake bakery, named 1 Cupcake, which allows customers to place their order online via its website, and have their orders delivered to their doorstep. In the user scenarios that we are covering, the customer will begin by entering his details, retrieving delivery fee and sending the order request via the Order UI. In the next user scenario, the bakery completes the order and triggers the delivery request as well as sends the customer a message on the success status upon delivery completion. In addition, the bakery also checks the company's Twitter account periodically for new promotions and displays the latest promotion on its homepage.

2. User Scenario 1



Customer places an order:

1. Customer accesses the order UI, begins the order request by entering his order details including the region, then the UI invokes the Delivery Pricing microservice via HTTP GET to get the delivery pricing and the region.
2. The Delivery Pricing service returns region & pricing according to each region and displays on the order UI.
3. The Order UI passes customers' details to Checkout UI and displays customers' details on Checkout UI.
4. When the customer clicks "Proceed to checkout", an order with status "Unpaid" is created in the database. This is done via HTTP POST method via Order microservice.
5. The customer then clicks "Pay" to bring up the payment service, Stripe,that is invoked through Order microservice.
6. The customer then keys in the following:

a. Email: <Personal email address>
b. Card number: 4242 4242 4242 4242
c. MM/YY: <Any date that is beyond today>
d. CCV: <Any 3 numbers>

7. Stripe then returns the payment status to Order microservice.
8. If successful, the status of the order entry in the database will be changed to “Paid”, and the customer will be prompted with a success message.

2.1.(Micro)Services

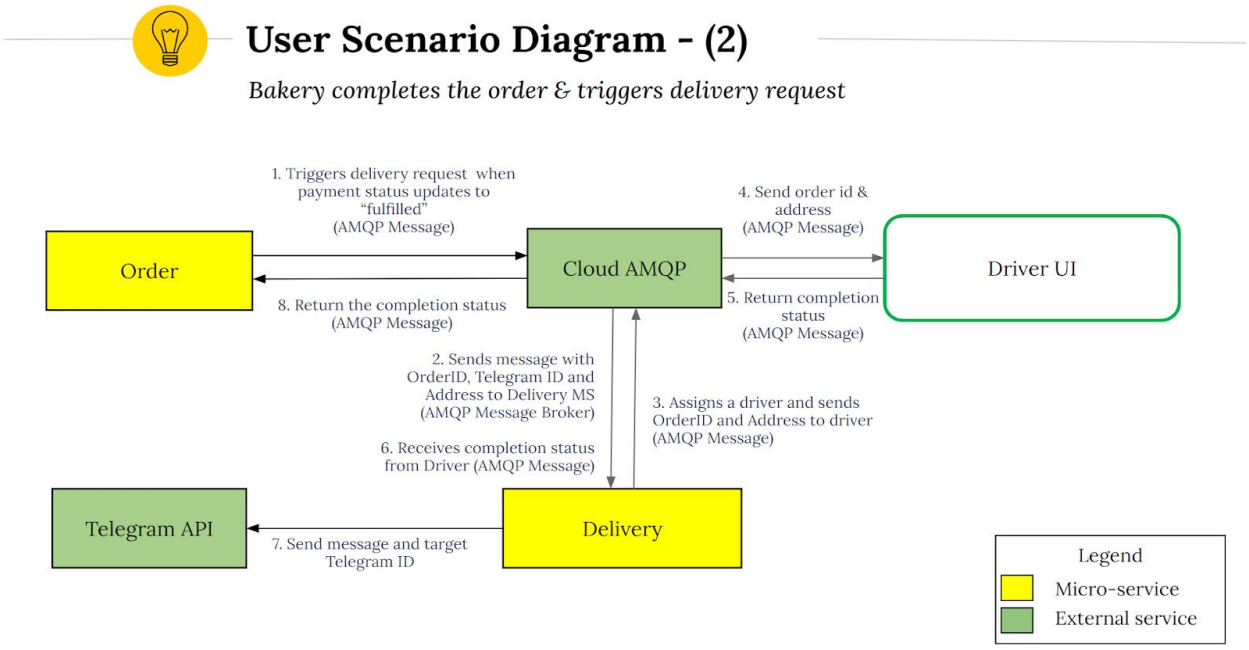
Service Name	Operational information	Description of the functionality	Input (if any)	Output (if any)
Delivery Pricing	HTTP GET	Using region, it returns delivery price for that region		{price}
Order	Add new order HTTP POST	[No additional description required for this operation as the Operation, Input & Output are clear enough for example]	{name, email,telegram_id, quantity, address, phone, postalCode, price}	{status of the order} and order details if any {bookingID, PassengerID, DriverID, DateTime, PickUpLocation, Destination, Price}
Stripe	Stripe API	Process the payment request and sends back the payment status to Order service	{stripeEmail, stripeToken}	Sends back same Charge object if successful, else error

2.2.Beyond the Labs

Order MS:

- Hosting of the database on Amazon RDS
- External service - Stripe API
- Multiprocessing: The main process listens on Flask and the second process listens on AMQP.

3. User Scenario 2



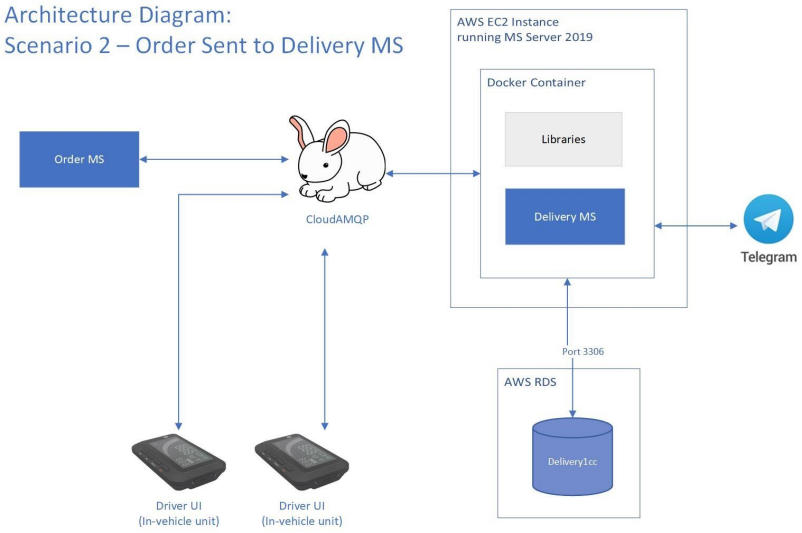
Bakery completes order and triggers delivery request:

1. Upon successful order creation, the Order service invokes the Delivery service to trigger the delivery fulfillment process.
2. The Delivery service takes an order from Order UI, writes it to database & assigns to any driver logged in to the microservice. It sends the OrderID & Address of the customer and displays it on the Driver UI for the driver.
3. Upon completion of delivery, Driver UI returns the completion status to the Delivery microservice.
4. The Delivery service invokes the external Telegram API to send the customer a message of the delivery completion status.
5. The Delivery service then returns the delivery completion status to Order service.

3.1. (Micro)Services

Service Name	**Operational information**	**Description of the functionality**	**Input (if any)**	**Output (if any)**
Order	AMQP Topic	Sends order to the Delivery MS	NA	JSON of order details: {telegram_id:<string>, email: <string>, name: <string>, quantity: <int>, address: <string>, phone: <string>, region: <string>}
Delivery	AMQP Topic	Assign Orders to Drivers: Takes an order from Order UI, writes it to the database and assigns the order to any available drivers. If none are available, it will keep trying to reallocate the order until a driver is available to receive it. Driver Verification: Checks if the Driver that is trying to log in has a valid username. Telegram Notification: Sends a Telegram message to customer when Driver notifies of delivery completion	For Orders: {“Order”: [OrderID, Address, Telegram]} For Validating Drivers: {“Validate”: [DriverID]} For Notification that Driver has logged off: {“Exit”: [DriverID]} For Notification that Driver has completed Order: {“Completed”: [DriverID, OrderID]}	Driver MS: For validating driver: [“Validate”, status<boolean>] For assigning of job: [“Order”, Address<string>] Order MS: ["order_complete",[completed_orderid, "Order has been delivered!"]] Telegram API: chatID, orderID+message <string>
Driver UI	AMQP Topic	Console-based user interface for Driver to log in, receive order status, complete order and notify Delivery MS of order completion. Time spent for a Driver to fulfill a job request is simulated with a timer.	For validating driver: [“Validate”, status<boolean>] For assigning of job: [“Order”, Address<string>]	For Notifying Delivery that Driver has logged off: {“Exit”: [DriverID]} For Notification that Driver has completed Order: {“Completed”: [DriverID, OrderID]} For Validating Drivers: {“Validate”: [DriverID]}
Telegram API	Telegram API	Sends notification of completed delivery to specified Telegram ID	chatID, orderID+message <string>	Sends actual Telegram message

3.2.Architecture Diagram



3.3. Rationale

Delivery Microservice is supposed to be a hub for multiple Drivers to connect to via their custom In-vehicle unit. As writing an app for the hypothetical in-vehicle unit is out of the scope of this project, we have chosen to use the Windows CLI to display the information for the Driver UI. The advantage of using this is that there is no need to handle the refreshing of the content. Furthermore, it would not make sense to design a web UI for it given the use-case. In the scope of this project, Driver UI will be running from the local machine and it will be able to connect to Delivery UI which is hosted on AWS EC2. Multiple physical machines can run copies of Driver UI and they will be differentiated so long as they key in a different DriverID when connecting to Delivery UI via the Driver UI interface.

For this scenario, we used AMQP Topic exchange to implement the communication between components. Topic is chosen as Topic gives the most flexibility in the possible communication patterns. Topic message allows us to use the routing key to differentiate between different Drivers, and this key is generated dynamically when the driver logs in to the system.

Our team planned to move Delivery MS to EC2, but Driver UI would remain on the local machine. In order to achieve this, we had to separate the AMQP broker (previously existing on localhost) and we found that Cloud AMQP can do this very easily, so we did not need to host it and open the port in EC2 server ourselves.

The sequence that we moved the components online was:

1. Set up the new database in RDS, connect to it via MySQL and run the local database's SQL to recreate the database on RDS. Modify code of Delivery MS to connect to RDS database.
2. Set up Cloud AMQP Broker and modify configuration of Driver, Order and Delivery UI to use Cloud AMQP
3. Containerize Driver UI on local machine
4. Containerize Delivery UI on local machine

5. Set up the server on AWS EC2, connect to the server and install Docker.
6. Copy the Delivery MS source code and dockerfiles to server, build and run the image from Powershell

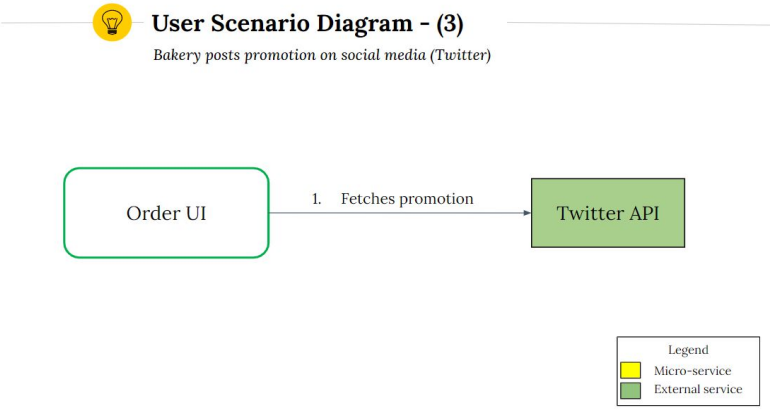
As Delivery MS communicates with Order MS, we did this in the sequence indicated to ensure that at any stage of the process, Order MS can still communicate with a working Delivery MS, with minimal changes to the code.

We chose Microsoft Server 2019 as we had issues creating Linux containers with Docker on our laptops, which could not be rectified towards the end of the project period. In addition, we planned to keep the Delivery service running 24/7 on the hosted EC2 Instance and eventually we might need to write a batch file or similar to check if the service is running periodically and restart it if it is down, and we have more confidence doing this in a Windows environment. Eventually, we clarified with Eng Kit that it is not necessary to have the service actually running for the grading, but since the environment has already been set up, we stayed with it.

3.4. Beyond the Labs

1) Telegram API - sends completion message to customer after order has been placed
2) Multiprocessing of 2 concurrent tasks:
    a) Main process - AMQP callback function for listening to incoming messages
    b) “Fantasma” process - A worker that checks the database at regular intervals to check if there are any jobs that were written to the database but were not assigned to Drivers at time of arrival (Due to no available drivers, etc). This process then re-assigns an available driver to the incomplete job.
3) Hosting of the database remotely on Amazon RDS
4) Hosting of the AMQP Broker remotely on AMQP Cloud.
5) Hosting of the Delivery Microsystem on AWS EC2 Instance running Microsoft Server 2019.

4. User Scenario 3



Bakery posts promotion on social media (Twitter):

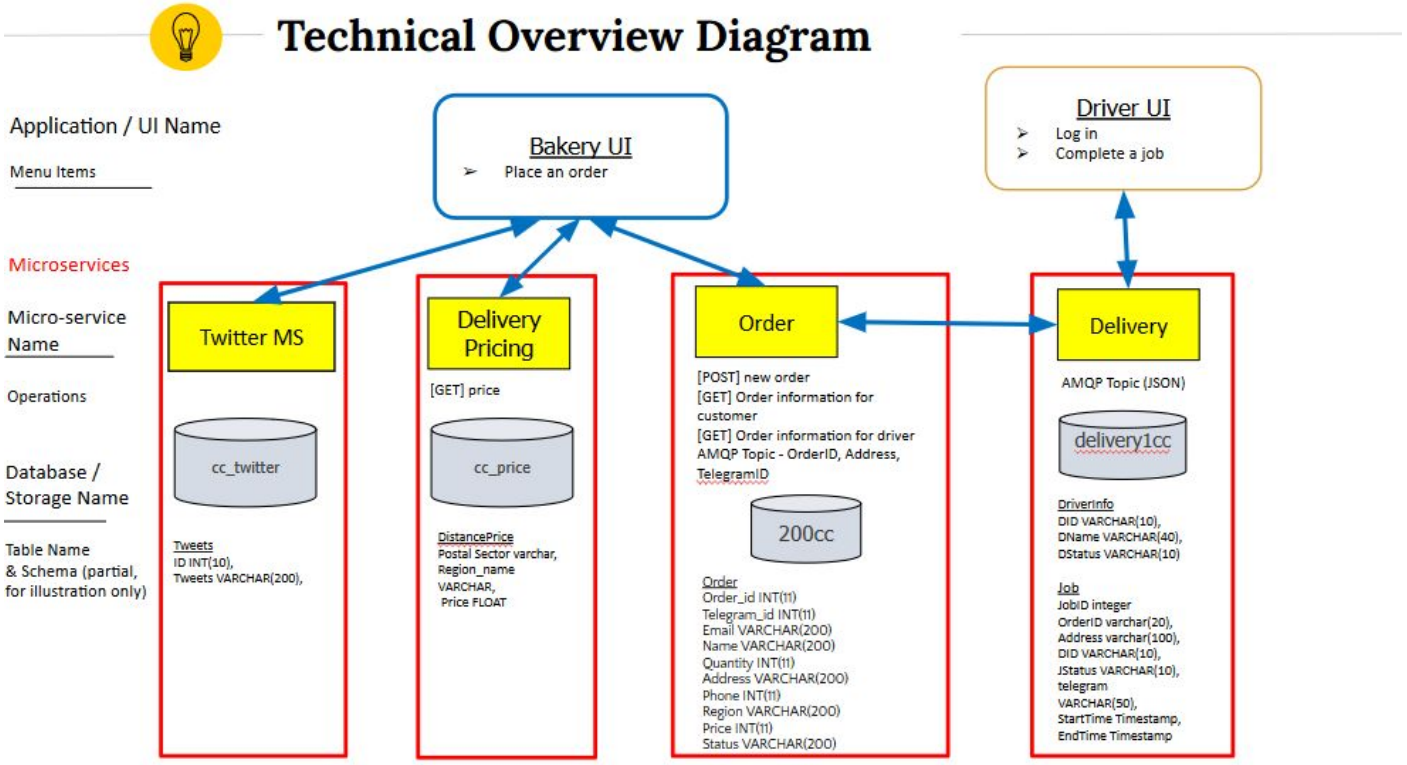
1. Bakery UI checks the company Twitter account periodically for new promotions and displays the latest promotion on the UI Homepage.

Service Name	Operational information	Description of the functionality	Input (if any)	Output (if any)
Twitter	Twitter API	[No additional description required for this operation as the Operation, Input & Output are clear enough for example]		

4.1.Beyond The Labs

Order UI: Twitter API - fetches tweets from our business' twitter account to be displayed in our promotion banner on the Bakery UI.

5. Technical Overview Diagram



6.Appendix

Related Screenshots for Scenario 1: Customer places an order for cupcake

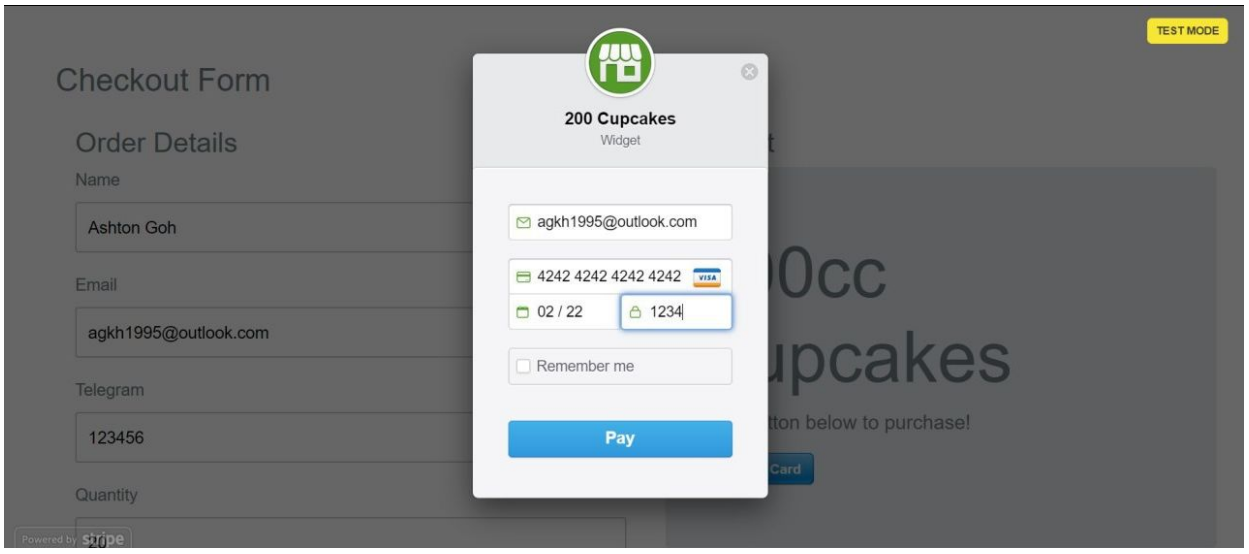


Figure 1.1: Display of Stripe payment option on Bakery UI

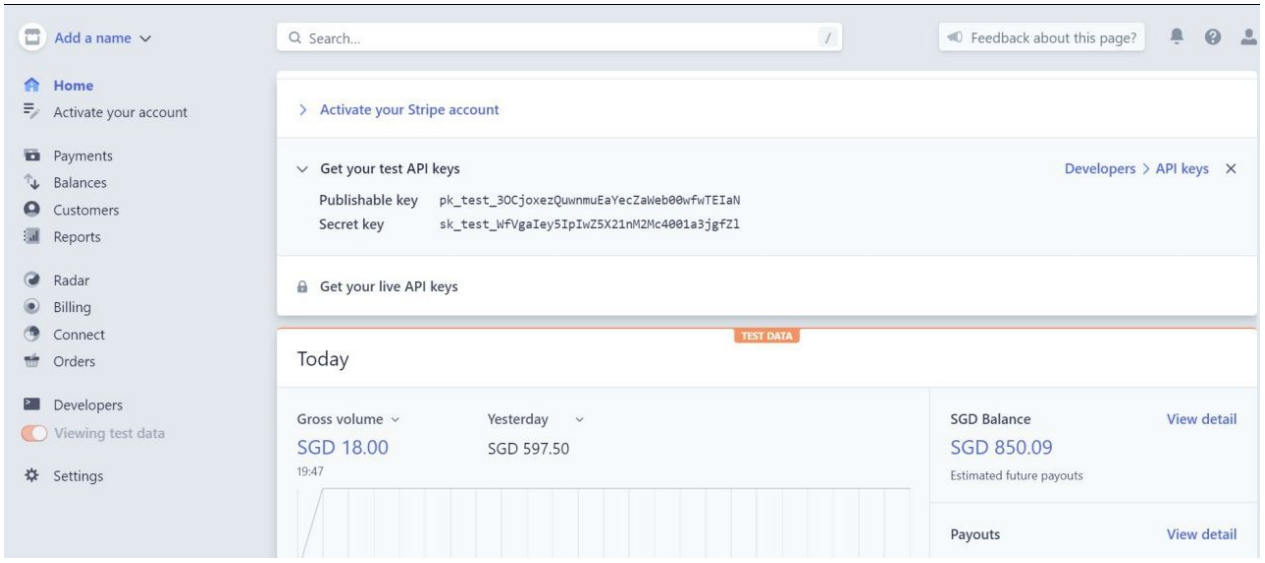


Figure 1.2: Stripe Console

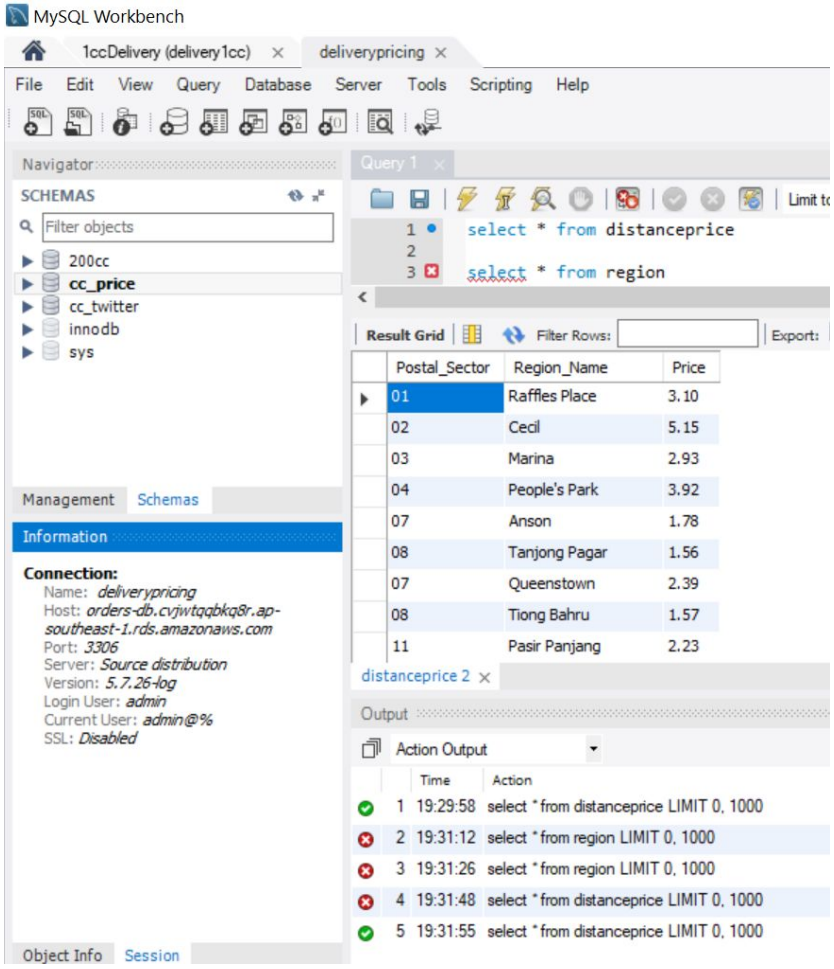


Figure 1.3: Databases hosted on RDS

Related Screenshots for Scenario 2: Bakery completes order and triggers delivery request

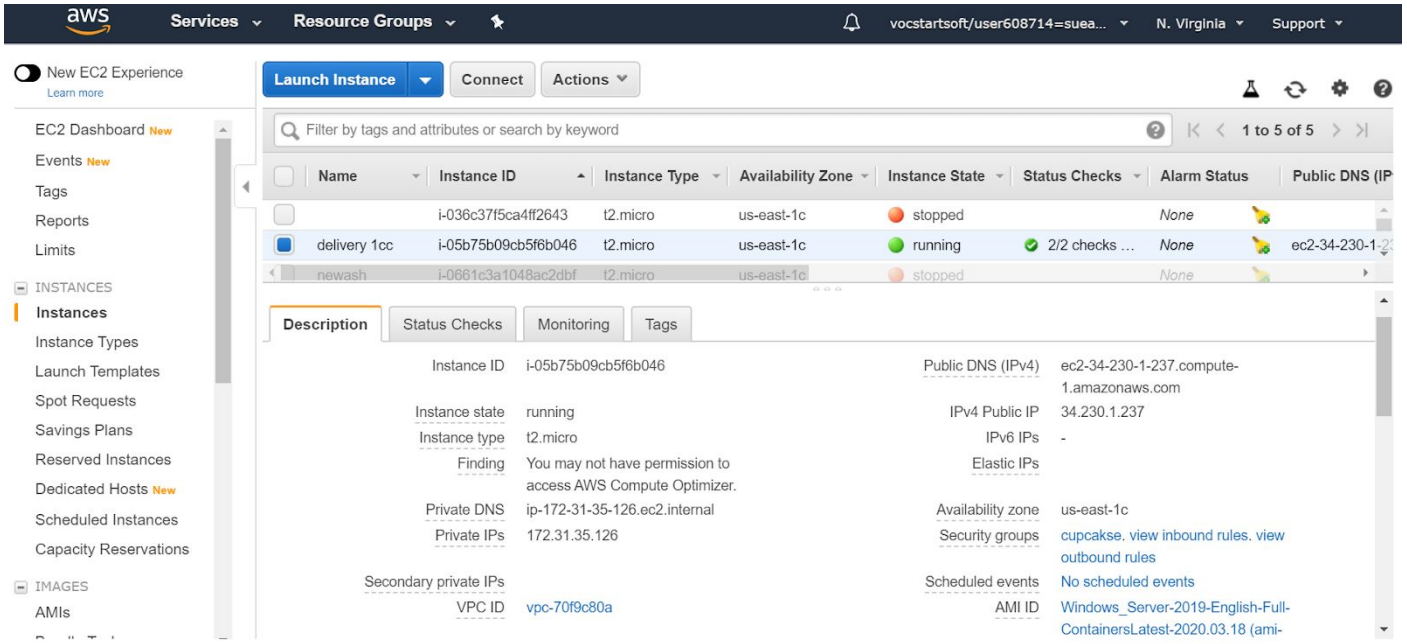


Figure 2.1: Instance running Delivery MS in AWS EC2 (the service will only be started upon request).

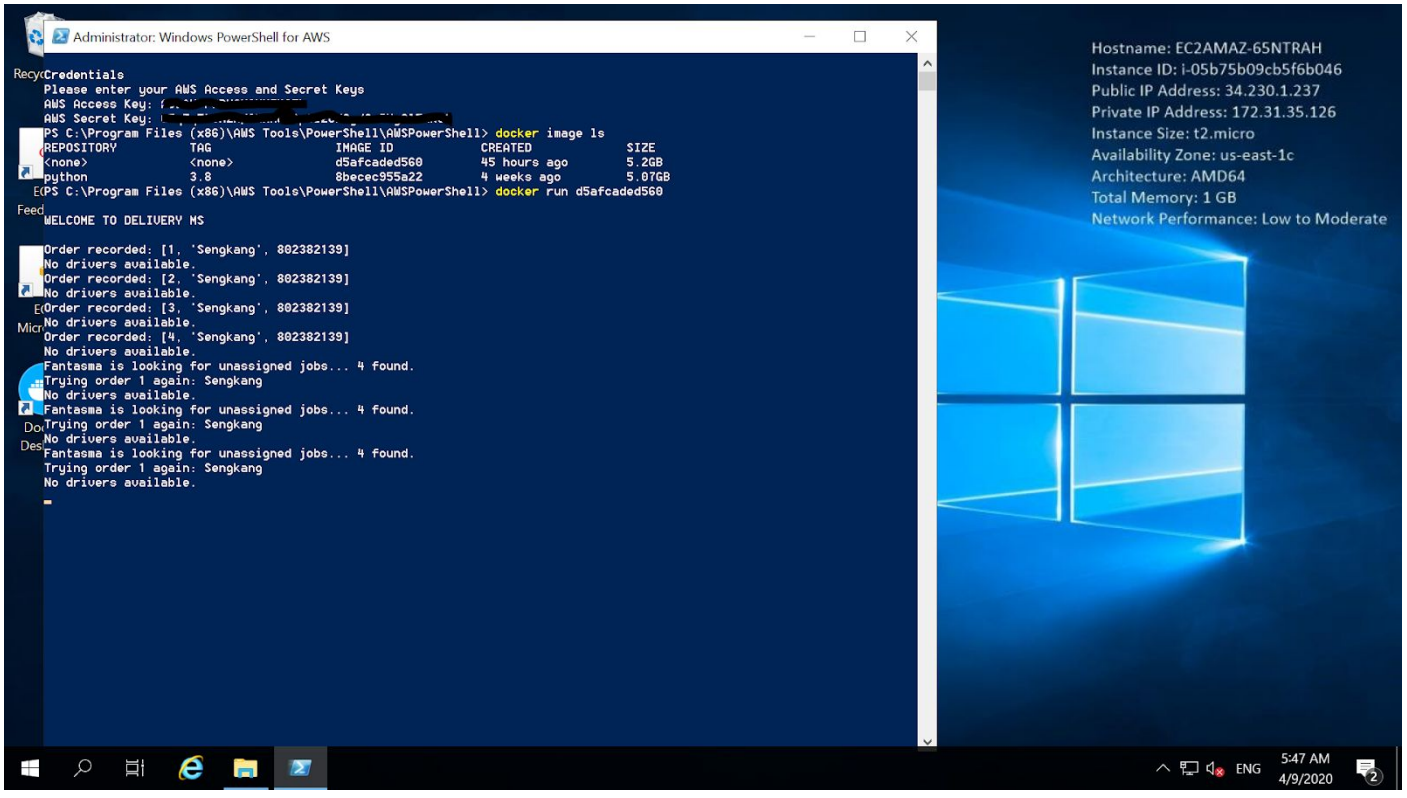


Figure 2.2: Delivery MS running on MS Server 2019 receiving orders from Order MS (no Drivers are logged in)

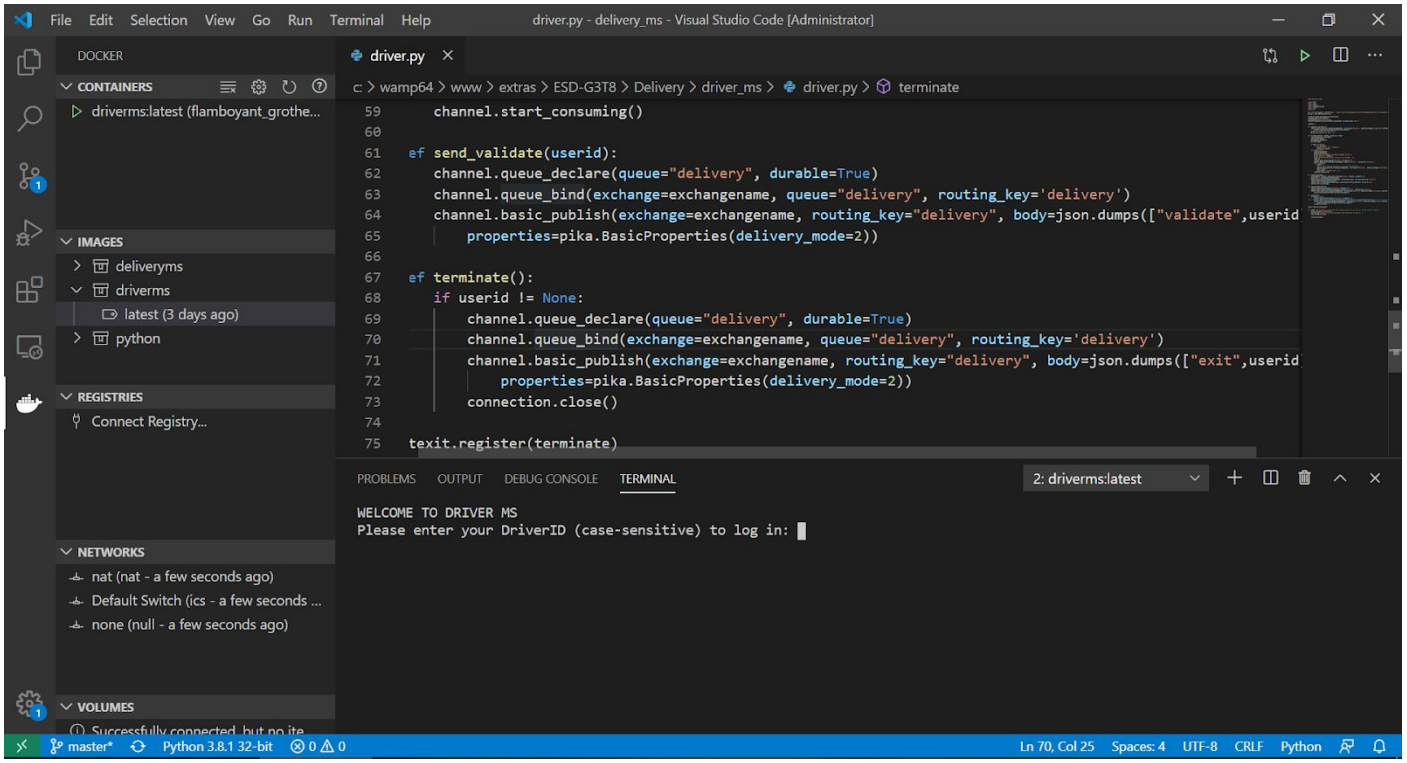


Figure 2.3: Driver MS running on local machine, but with connection to Cloud AMQP

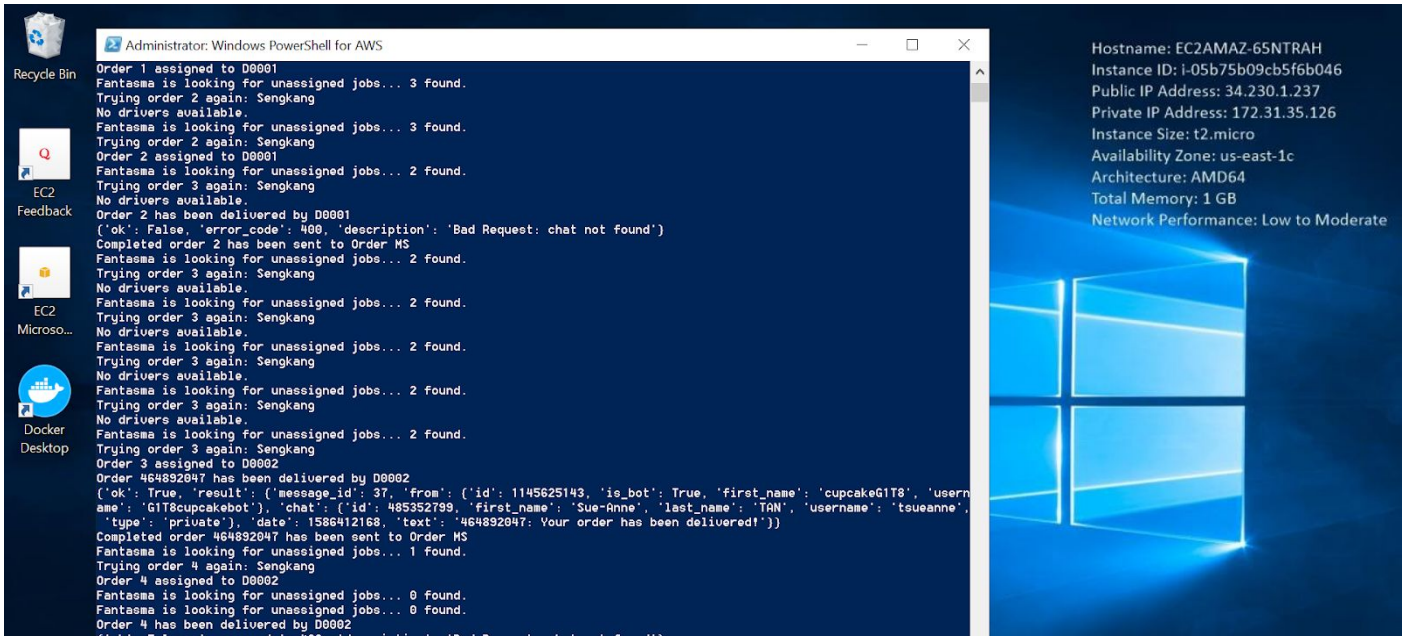


Figure 2.4: Delivery MS running with 2 drivers connected, showing Fantasma reassigning the leftover jobs.

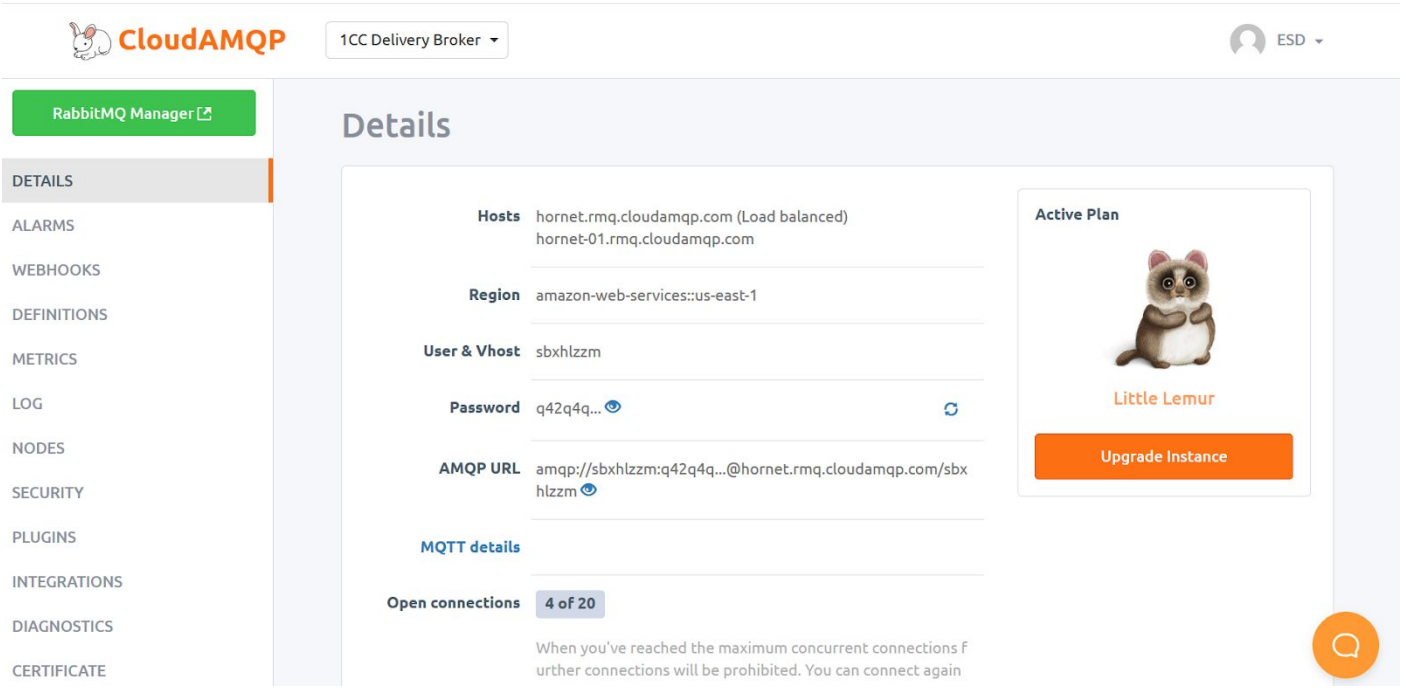


Figure 2.5: Cloud AMQP Console showing 4 connections (Order, Delivery and 2 Drivers)

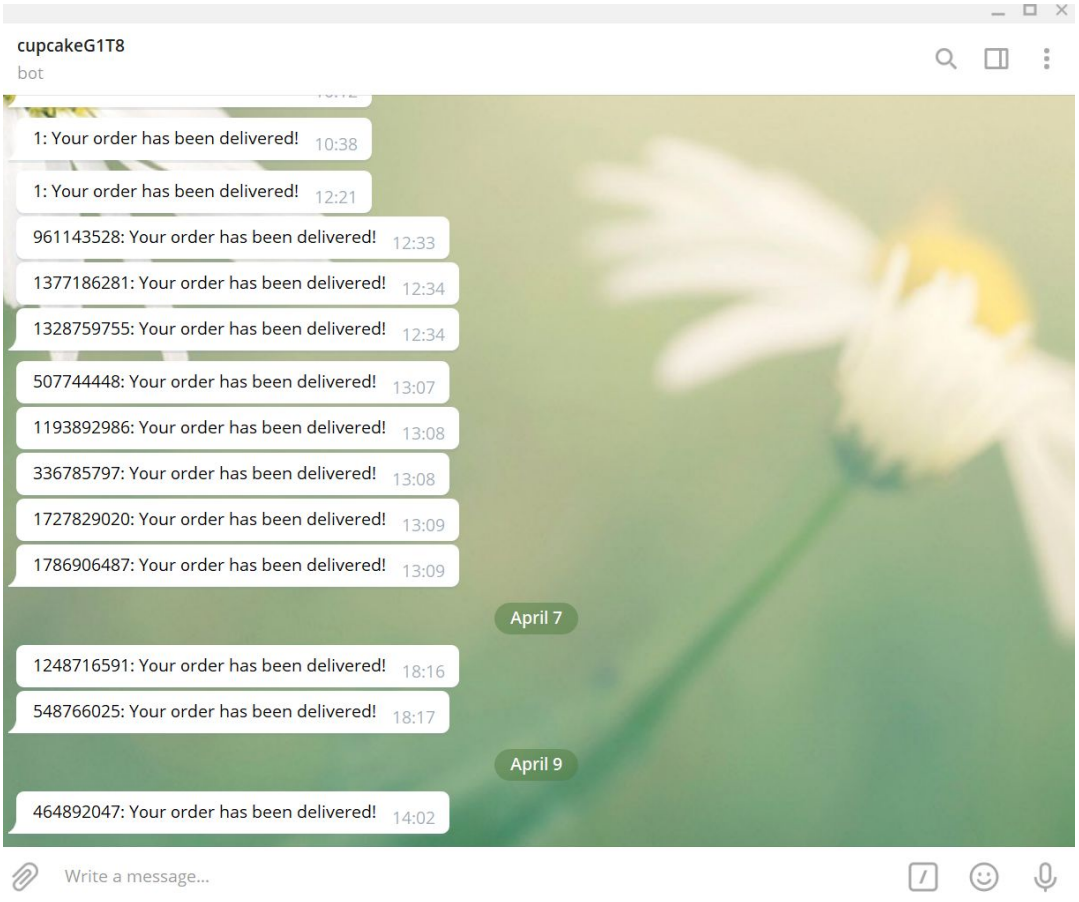


Figure 2.6: Notification sent to Telegram chat

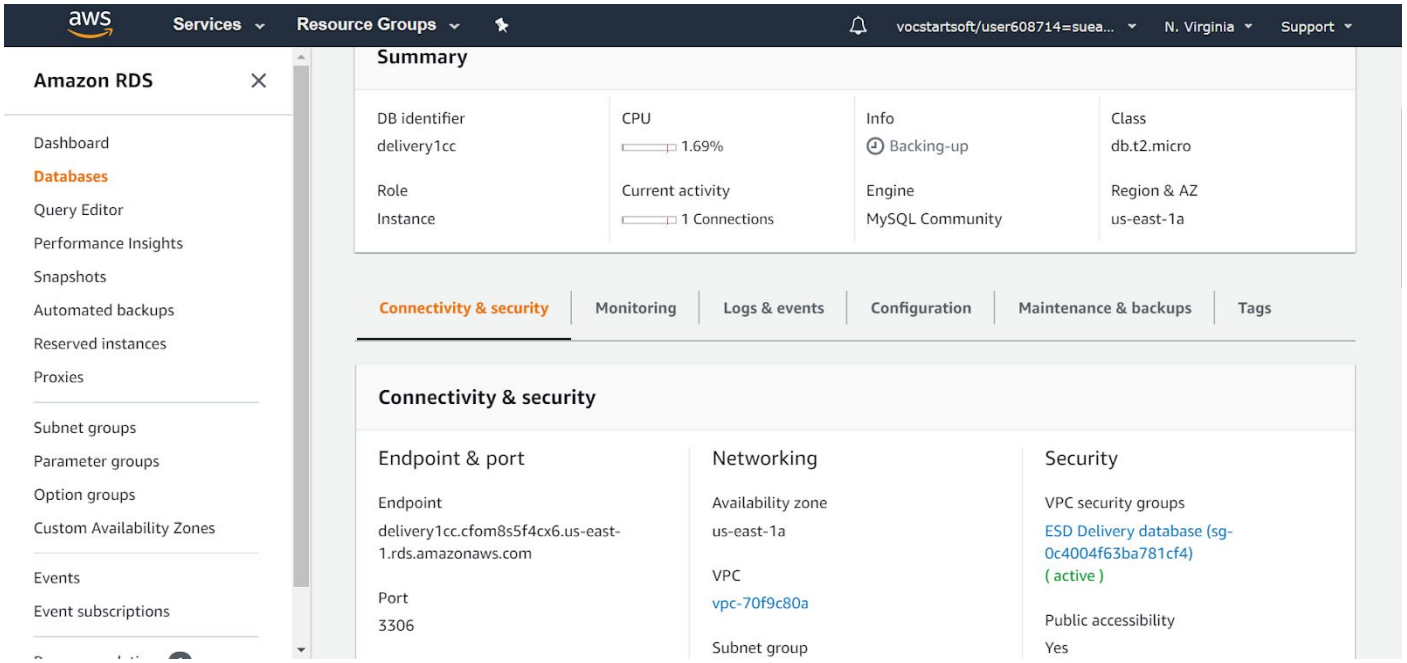


Figure 2.7: RDS Console showing Delivery database

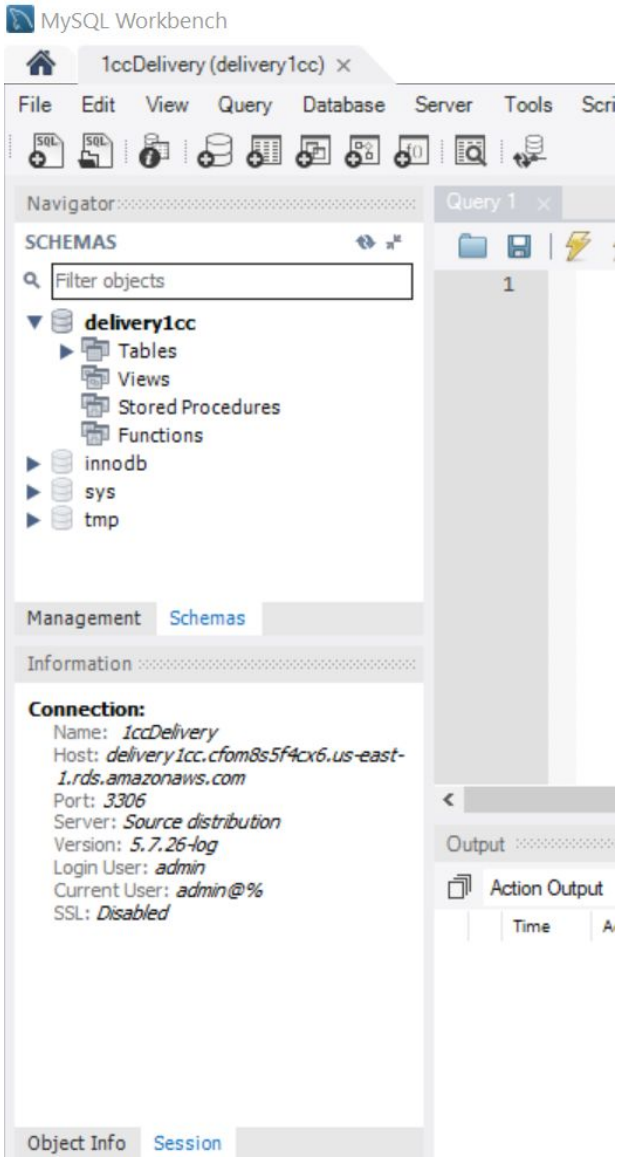


Figure 2.8: MySQL connected to Delivery database