CSCI 5410 Serverless Data Processing

Assignmnet-2

PART-A REPORT

The paper “Performance Evaluation of Distributed Systems in Multiple Clouds using Docker Swarm” presents a performance evaluation of a Docker Swarm-based distributed system [1]. The study showcases the design and simulated development of the system, focusing on attributes such as high availability, fault tolerance, automatic scalability, load balancing, and maintainability of services [1]. While the evaluation was conducted on a small Docker Swarm cluster, future prospects include assessing the system's performance on larger clusters and exploring its deployment in multiple clouds to analyze communication and interoperability. The findings highlight the effectiveness of Docker Swarm as a natural choice for distributed systems and provide insights for further advancements in this area.

A Docker swarm cluster requires managers to manage resources and services, with one designated as the leader and the others as followers. Workers handle basic operations such as task execution and container-related data traffic. Managers oversee workers, schedule tasks, and perform periodic health checks. Swarm manager nodes use the Raft Consensus Algorithm to manage the swarm state [1]. The leader sends periodic heartbeats to other nodes, and in case of leader failure i.e., no communication over a period of time called as election timeout then new leader is elected in the presence of quorum of managers [1].

The authors simulated development of the distributed system using Docker Swarm involved creating five Docker machines within VirtualBox and configuring them to form a cluster. Inbuilt high availability and fault tolerances were tested by abruptly stopping the viable leader one by one, at point docker swam cluster could not continue its normal as there are less than 2 managers for successful failover procedure [1]. The fault tolerance of Docker Swarm is achieved through the Raft Consensus Algorithm, which allows the managers in the cluster to coordinate. By having 2f+1 managers in the Docker Swarm cluster, the system can tolerate the failure of up to f managers [1]. This means that with an odd number of managers distributed across availability zones enhances fault-tolerance in the face of failures or maintenance scenarios [2].

The authors deployed two services (nginx and redis) in a Docker Swarm cluster and scaled them to five instances each [1]. Observing Docker Swarm's automatic load balancing, the replicas were efficiently allocated to nodes with optimal resources. Additionally, to test maintainability, the user abruptly stopped two nodes, and Docker Swarm automatically switched the instances running on those nodes to other available nodes. These experiments highlight Docker Swarm's capabilities in automatic scalability, load balancing, and maintaining services within a distributed system.

The user performed experiments comparing Docker Swarm and Google Kubernetes in terms of scalability performance on Cloud Container Cluster Common Benchmark framework by running 30,000 containers across 1,000 nodes [1]. The results showcased Docker Swarm's superior performance in container startup time and system responsiveness, particularly for small-scale workloads, while suggesting the usage of Google Kubernetes or other frameworks like Mesos for larger-scale deployments [1].

PART-B REPORT

GIT LAB Link - https://git.cs.dal.ca/pusuluru/csci-5410-summer-23-b00913674.git

1. Firestore has no collection(s) initially

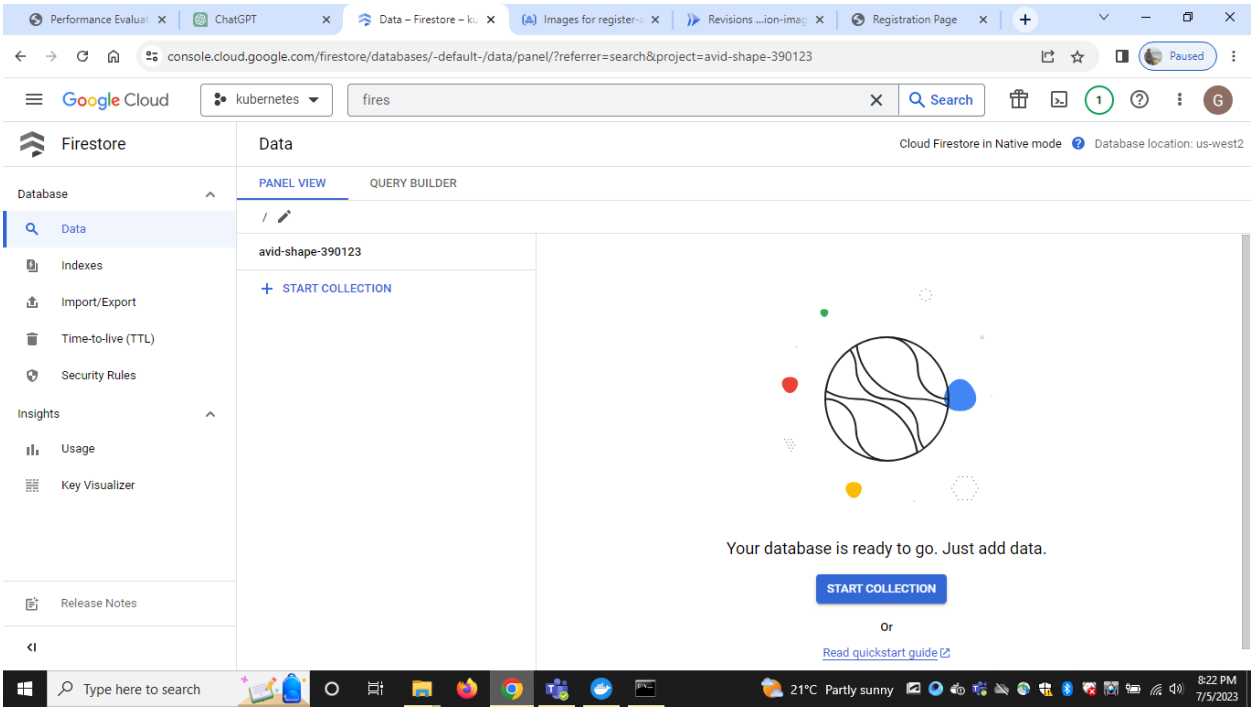


Figure 1: Firestore with no collections

2. Registration code docker image in artifact registry

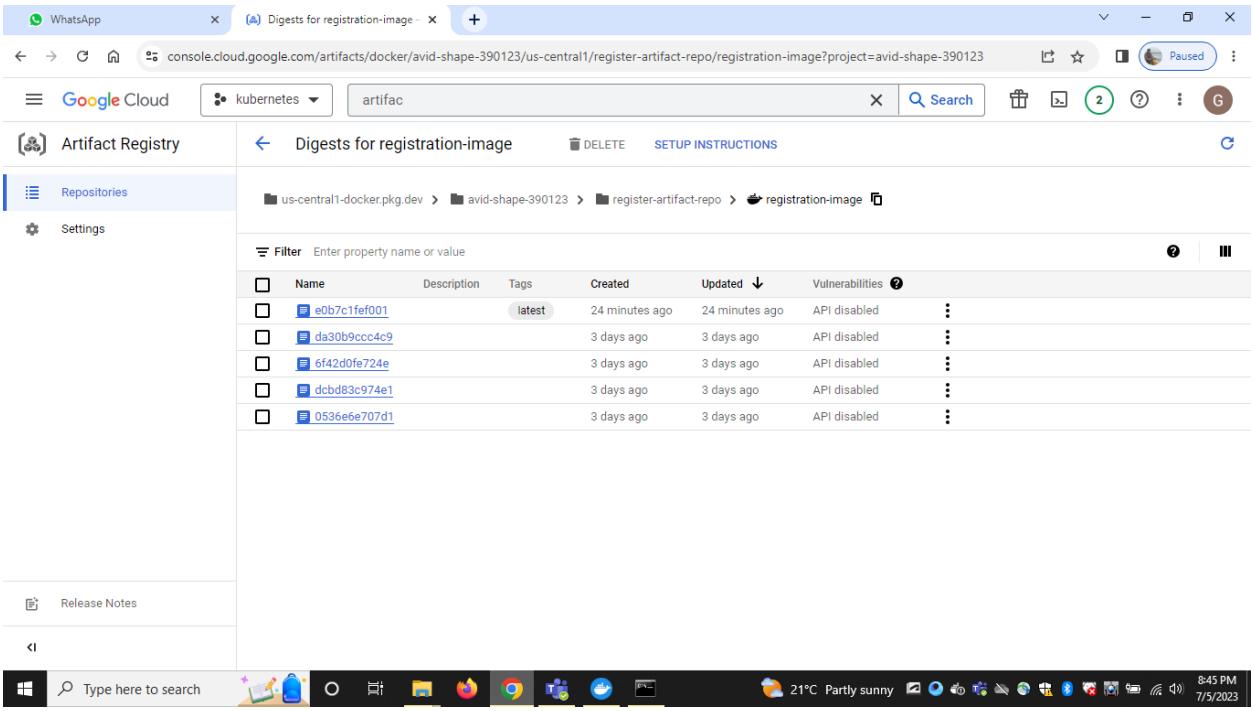


Figure 2: docker images of registration code under artifact registry

3. Login Code docker image in artifact registry

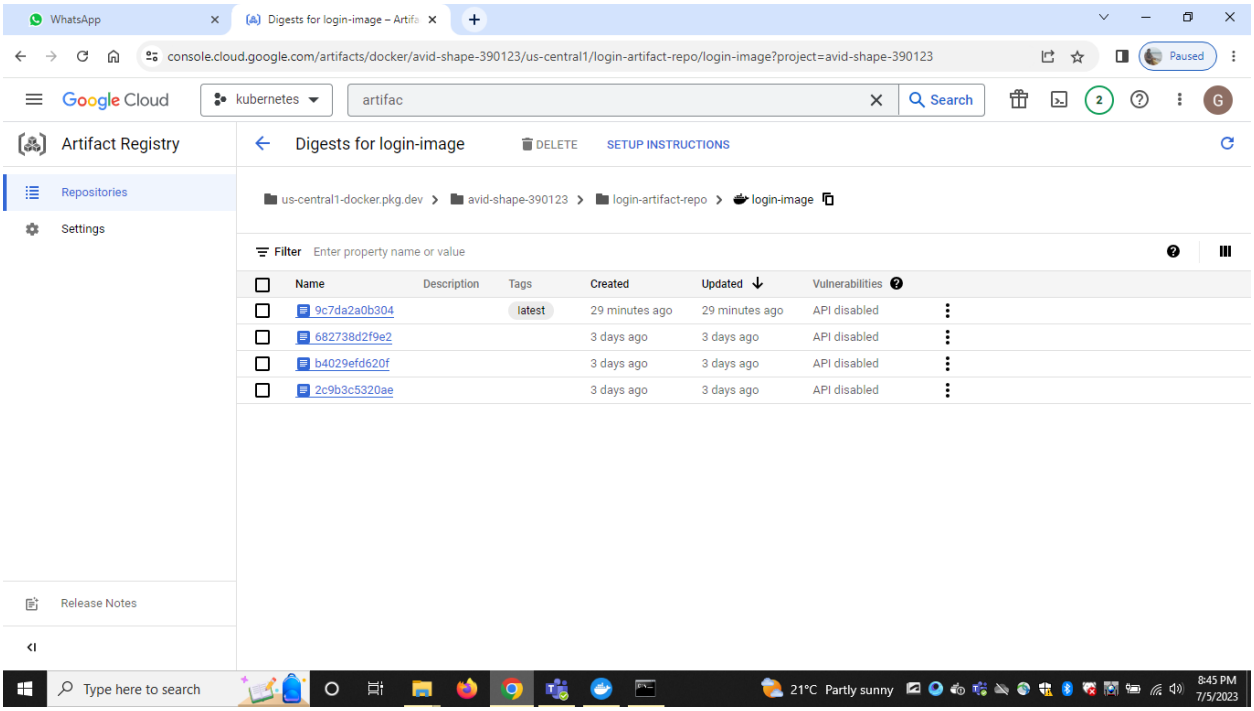


Figure 3: docker images of login page code under artifact registry

4. Online available users' visibility code docker image in artifact registry

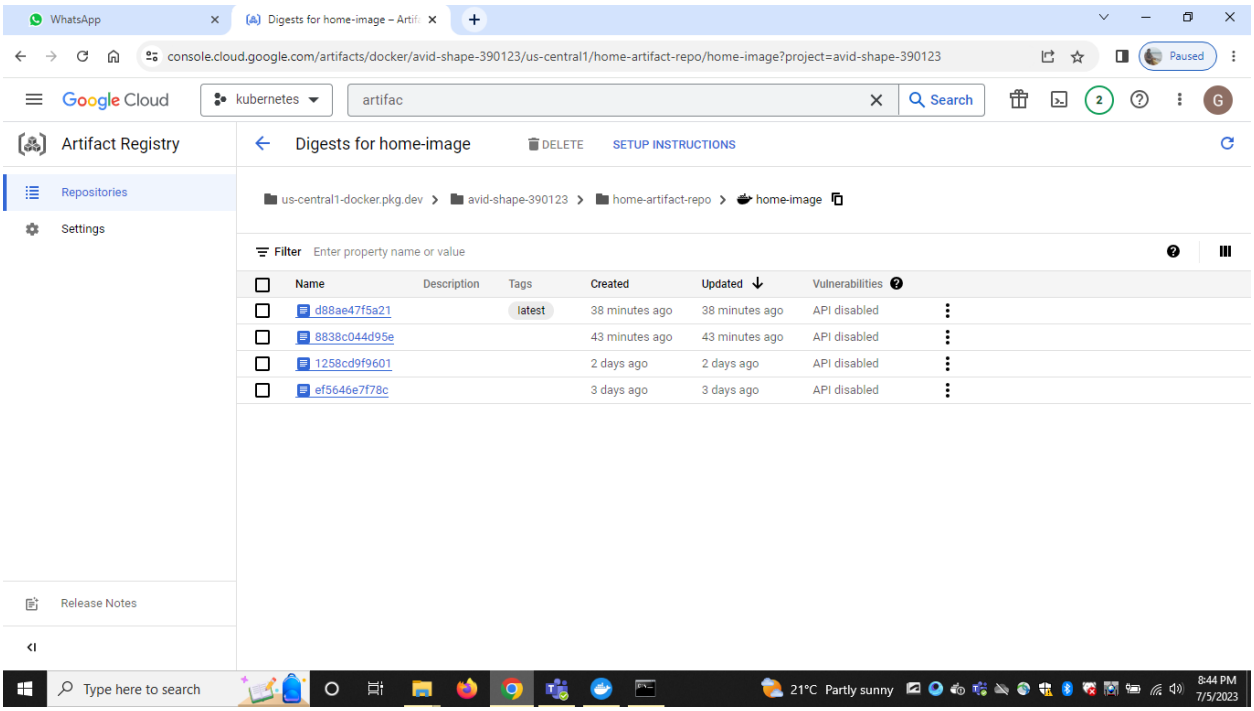


Figure 4: docker images of home page code under artifact registry

5. Registration Page after deploying docker image in the cloud run

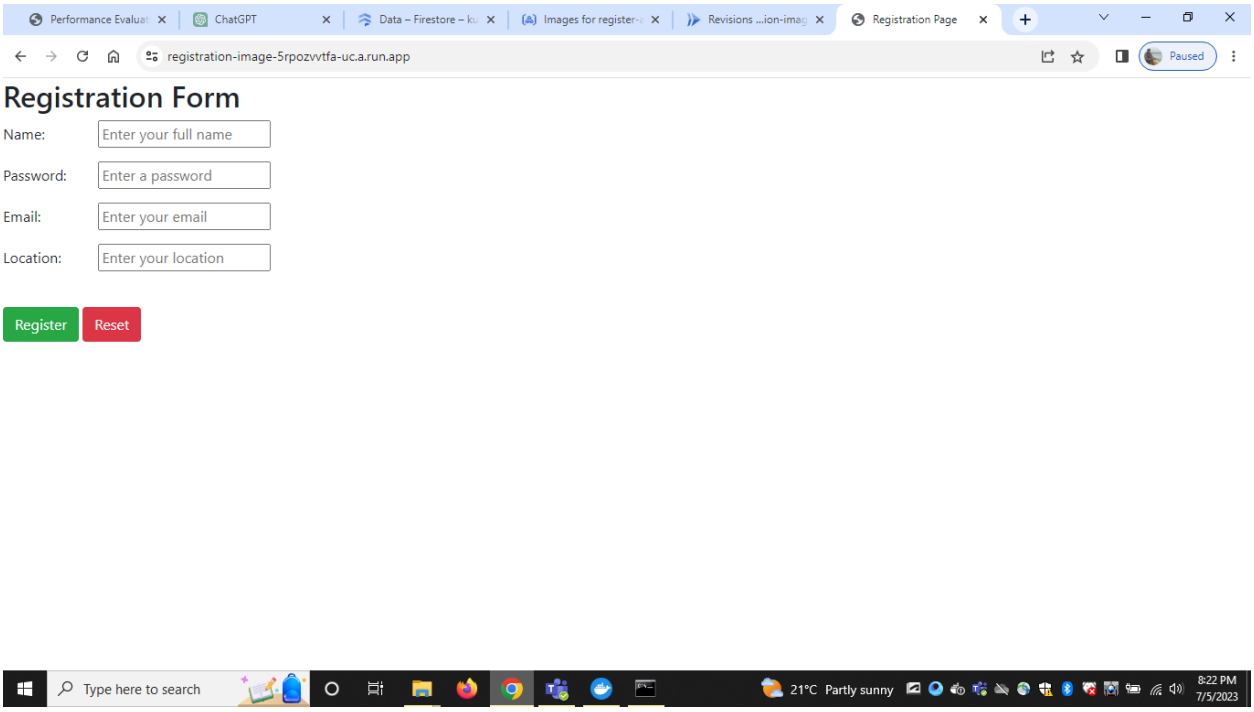


Figure 5: Registration form page

6. Registration Form Filled with details of user “Giri Sharan Reddy Pusuluru” and ready to submit

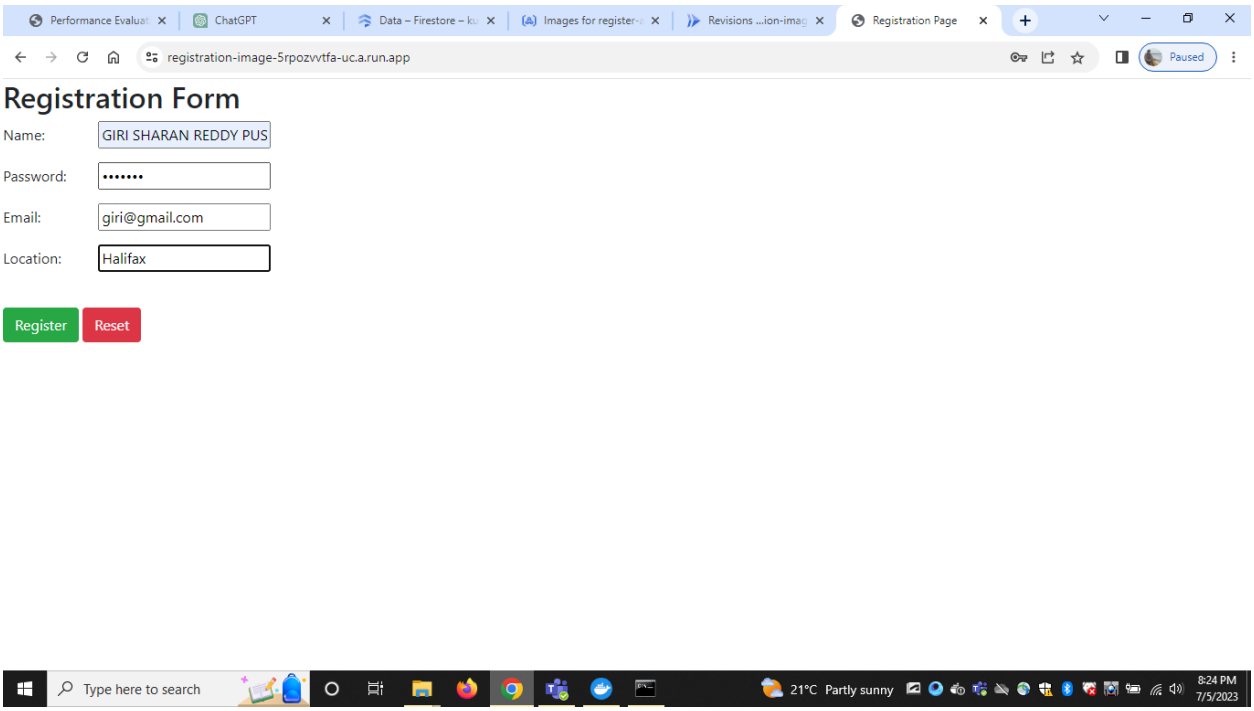


Figure 6: Details filled in registration form page

7. After submitting Registration Form, the details are getting stored in Firestore Database

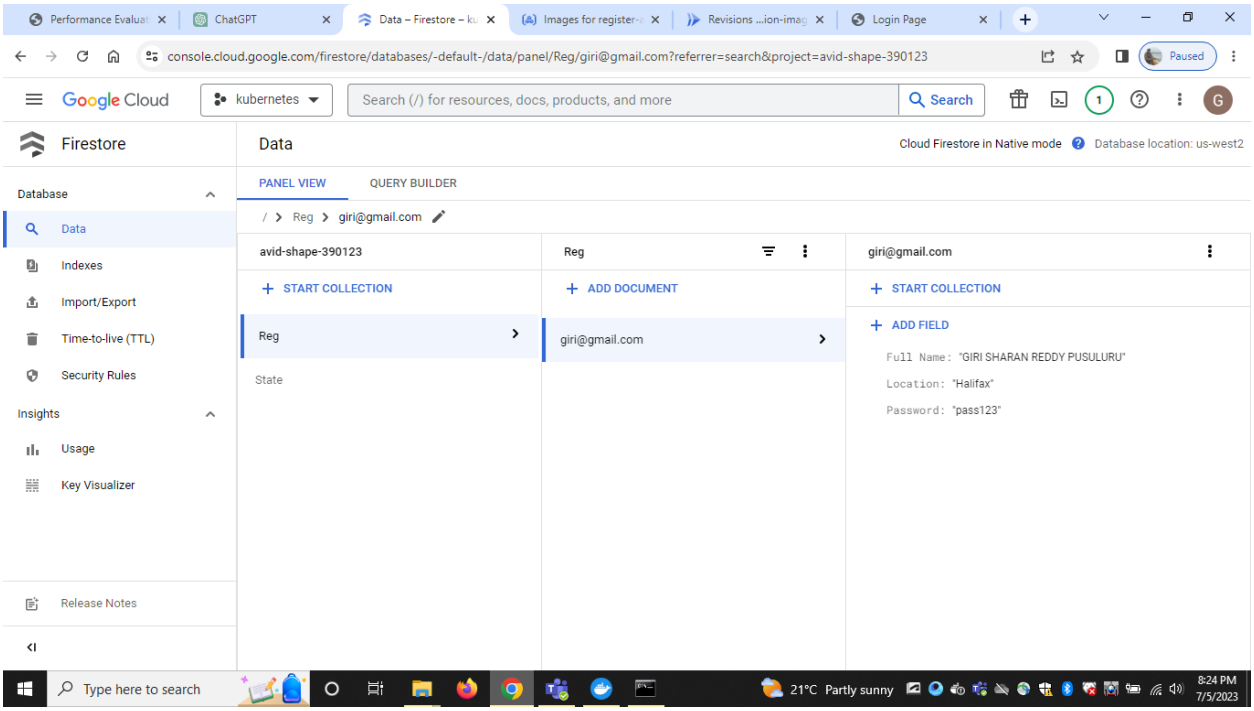


Figure 7: Details updated in firestore collection Reg after user registration

As the user just got registers and did not login yet so the field “Logged_In” is offline, and also keeping track of Login and Logout Timestamp

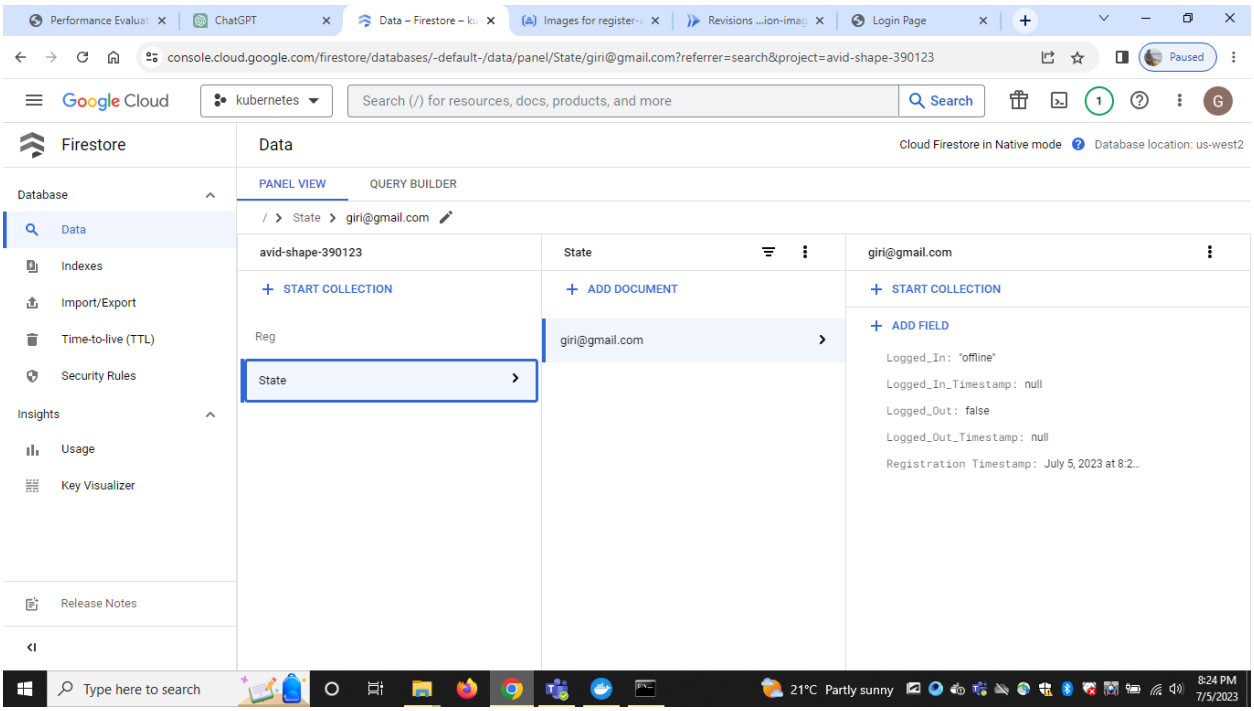


Figure 8: Details updated in firestore collection State after user registration

8. Logs from cloud run of registration service

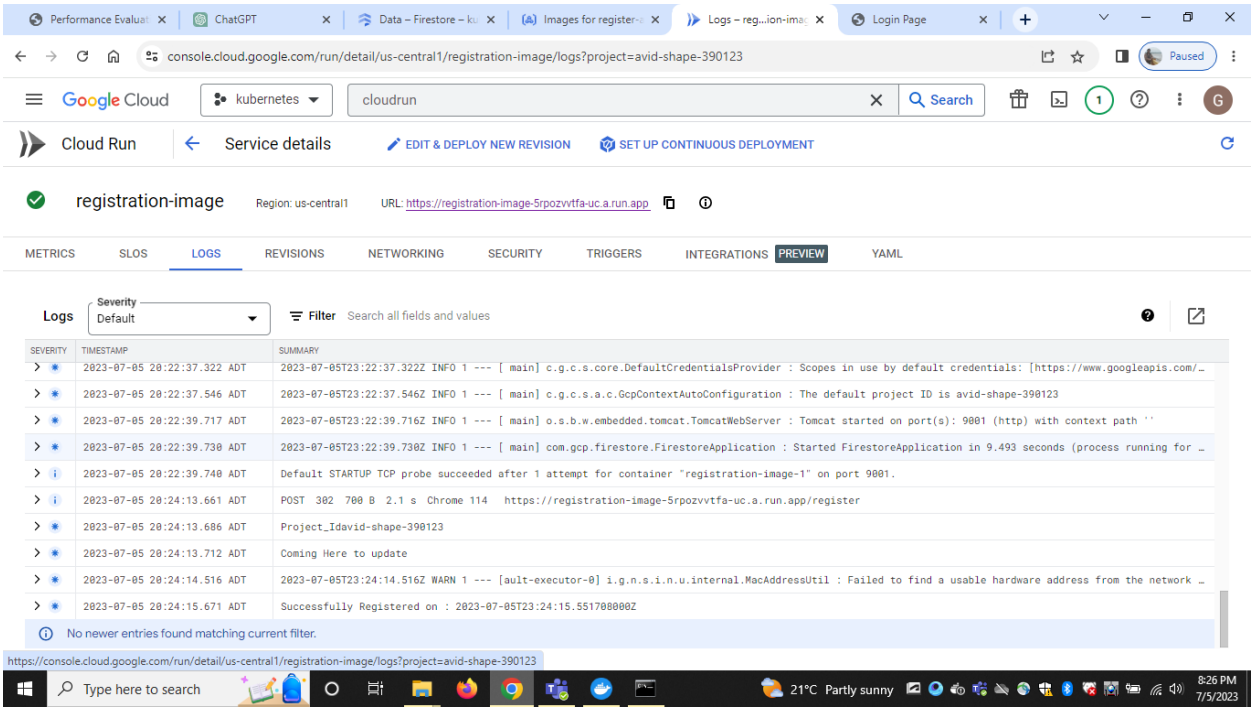


Figure 9: Logs in cloud run registration-image service

9. Once user is successfully registered, then will be redirected to Login page

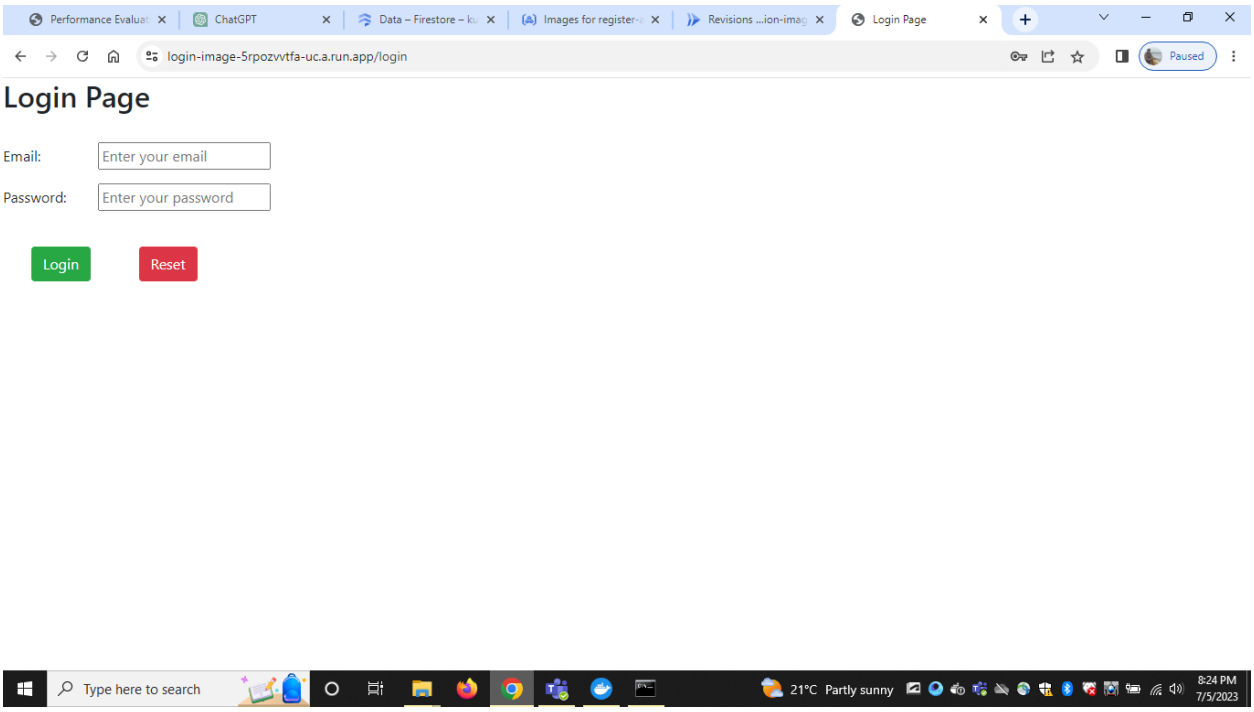


Figure 10: Login Page

10. Filled login page details to check for authentication. If the authentication is failed then he will be in the same page else redirected to home page

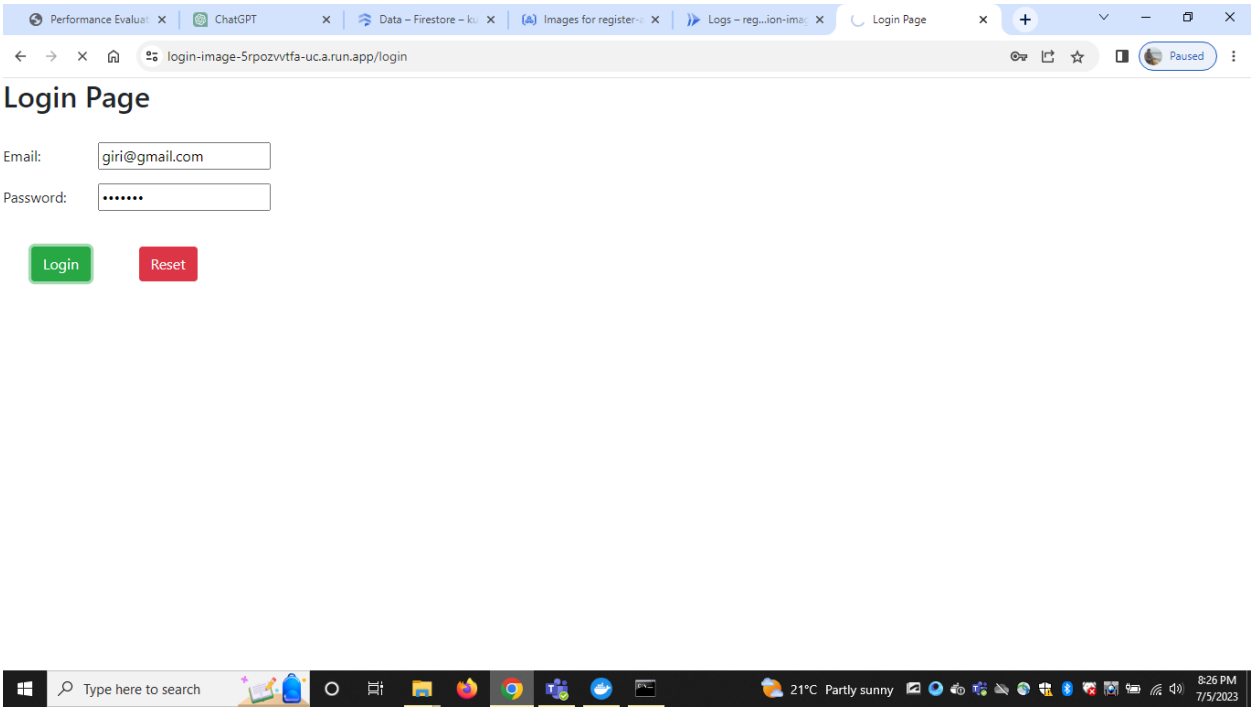


Figure 11: Details filled in login page for authentication

11. User Giri Sharan is successfully logged in and redirected to home page

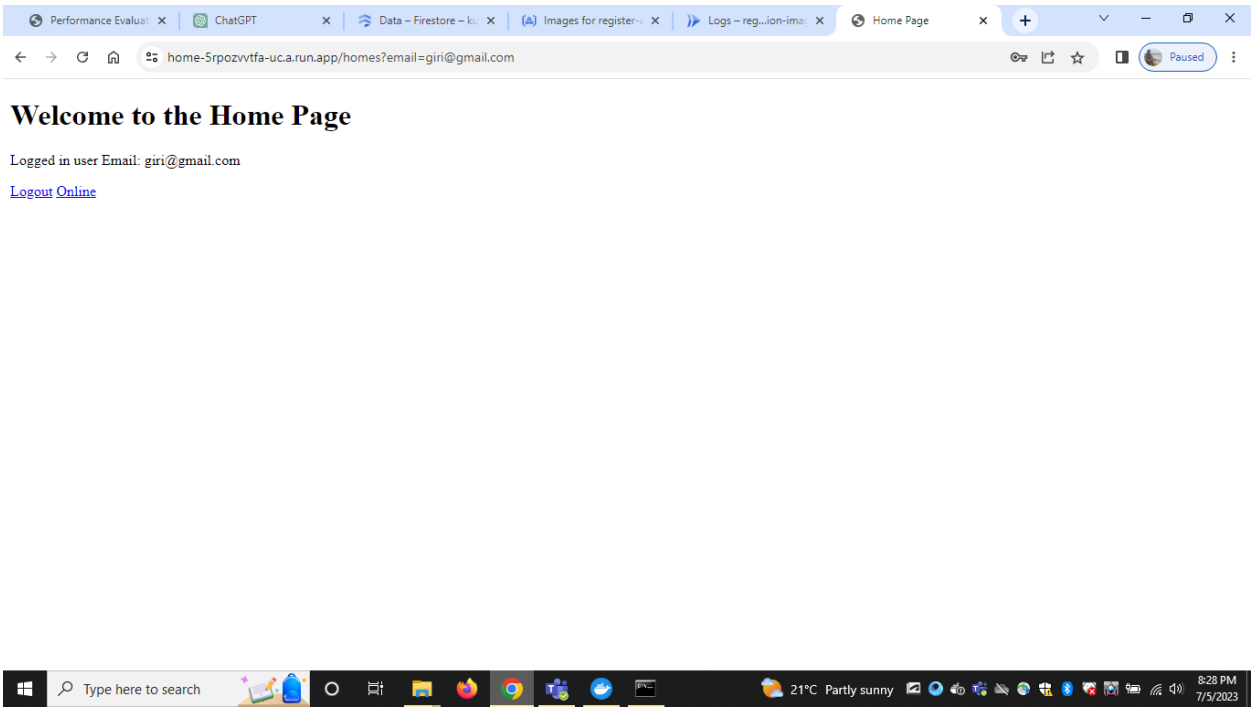


Figure 12: Home Page

12. Fields getting updated in Firestore Collection “State” of user specific document after user login

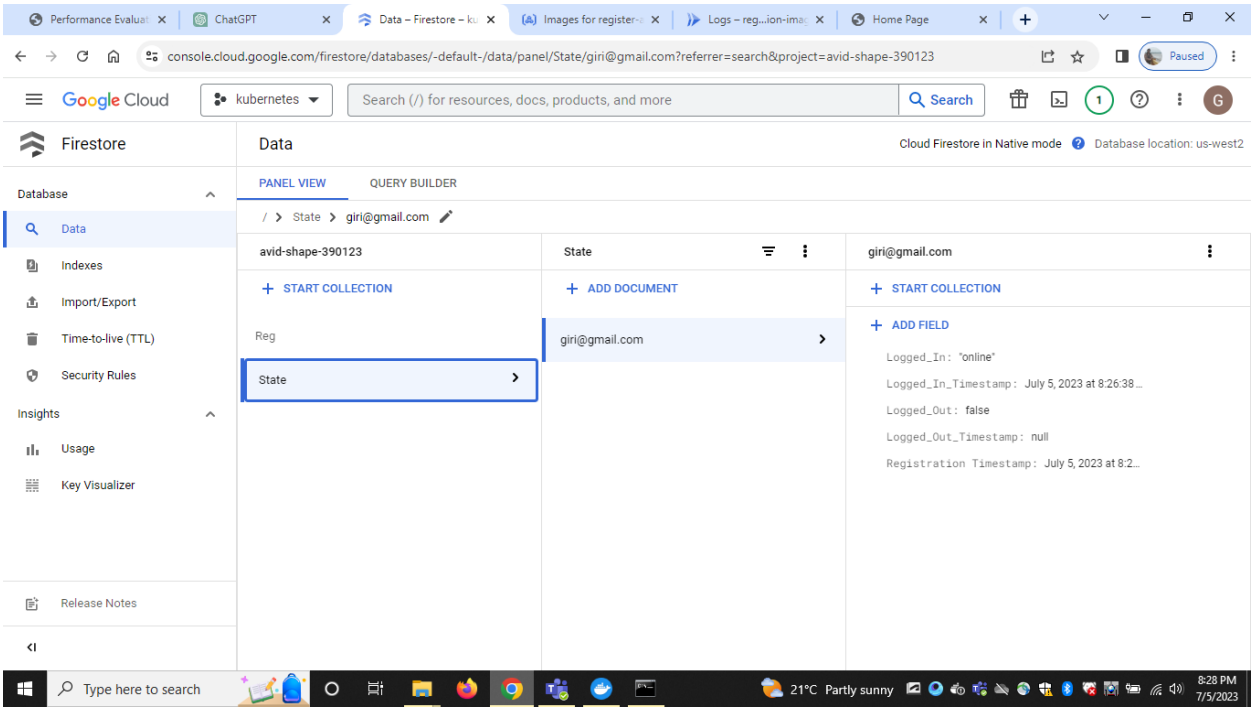


Figure 13: Fileds updated in State collection after user login

13. Registration Form Filled with details of user “Sankeerth” and then submitted

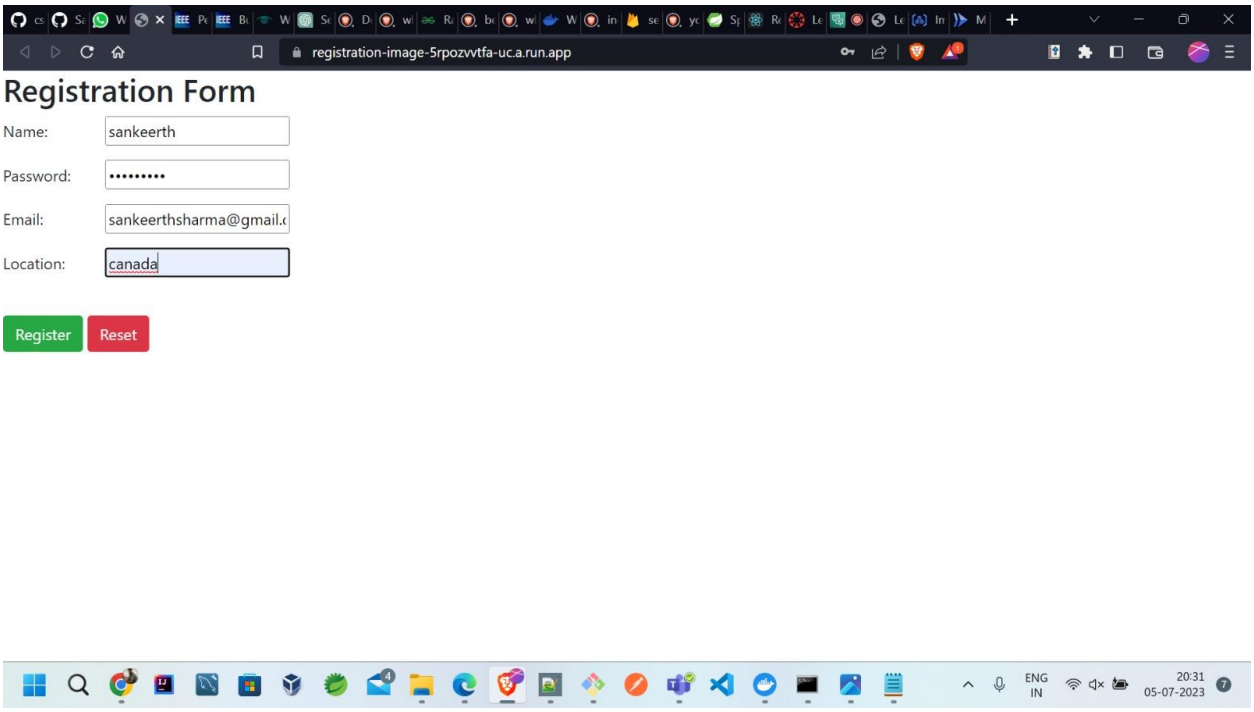


Figure 14: Registration of a user

14. User Tupper getting registered

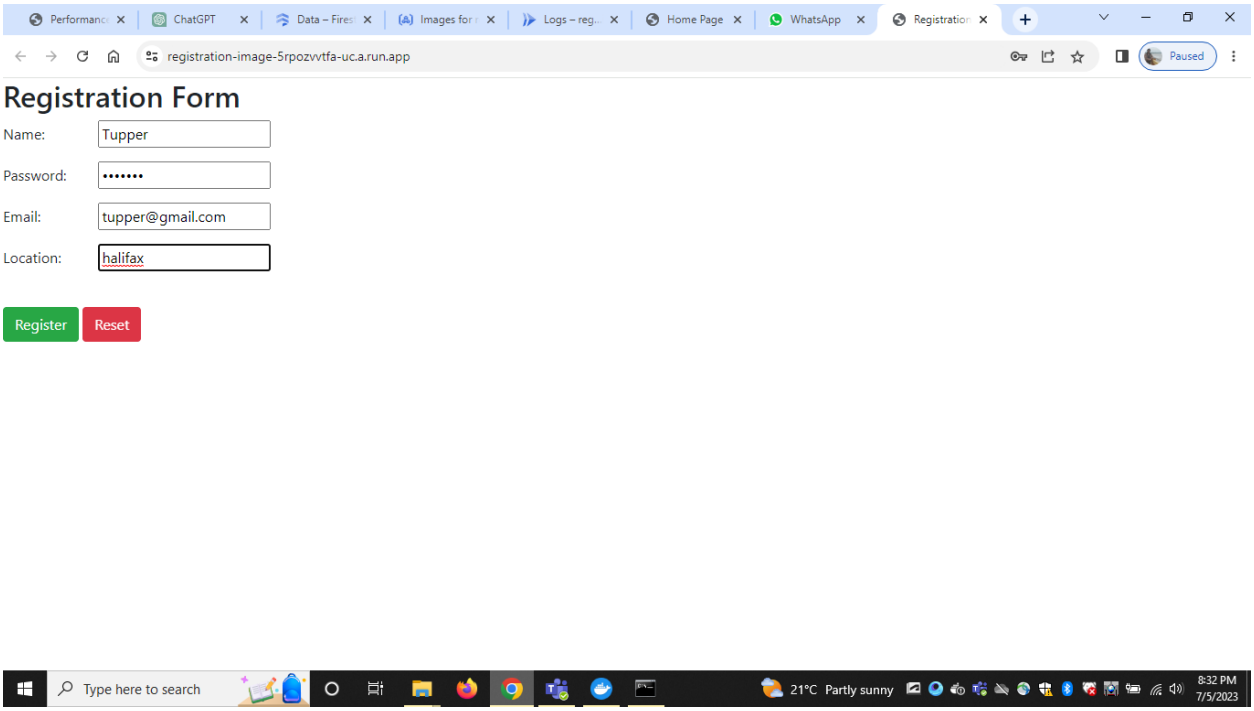


Figure 15: Registration of a user

15. Now user Sankeerth is logged in and clicking on button online in home page and able to see user Giri Sharan, who is also logged in but not Tupper because Tupper did not signin

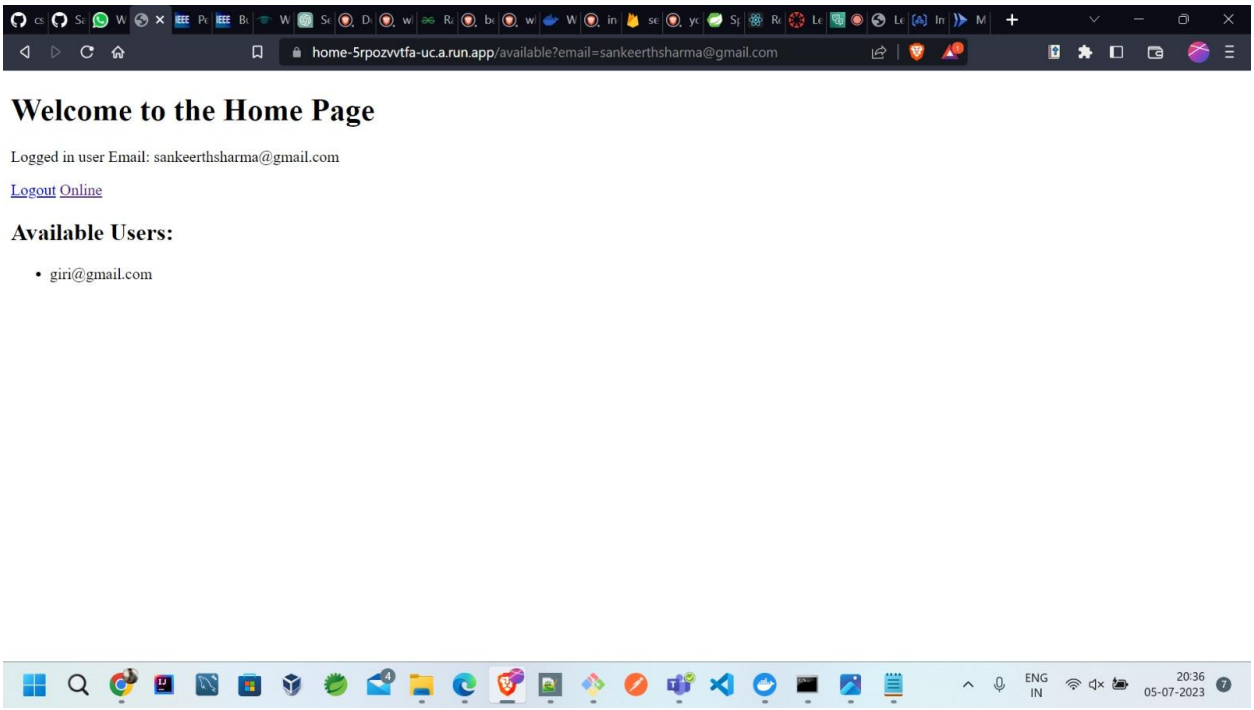


Figure 16: List of available users displayed in home page

16. User Sankeerth Field logged_in state got updated in Firestore collection of that user

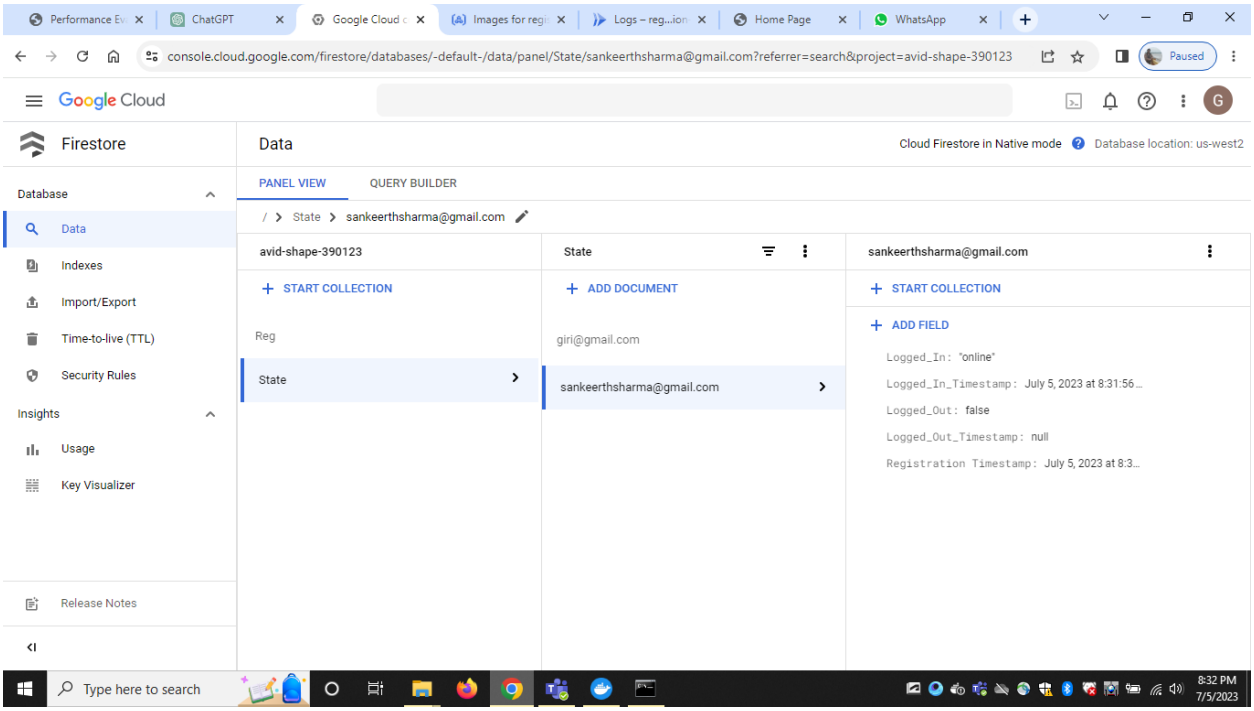


Figure 17: user State after login

17. Checking in user Giri Sharan by clicking on online button provided in home page and now he is able to user Sankeerth as he signed in just now

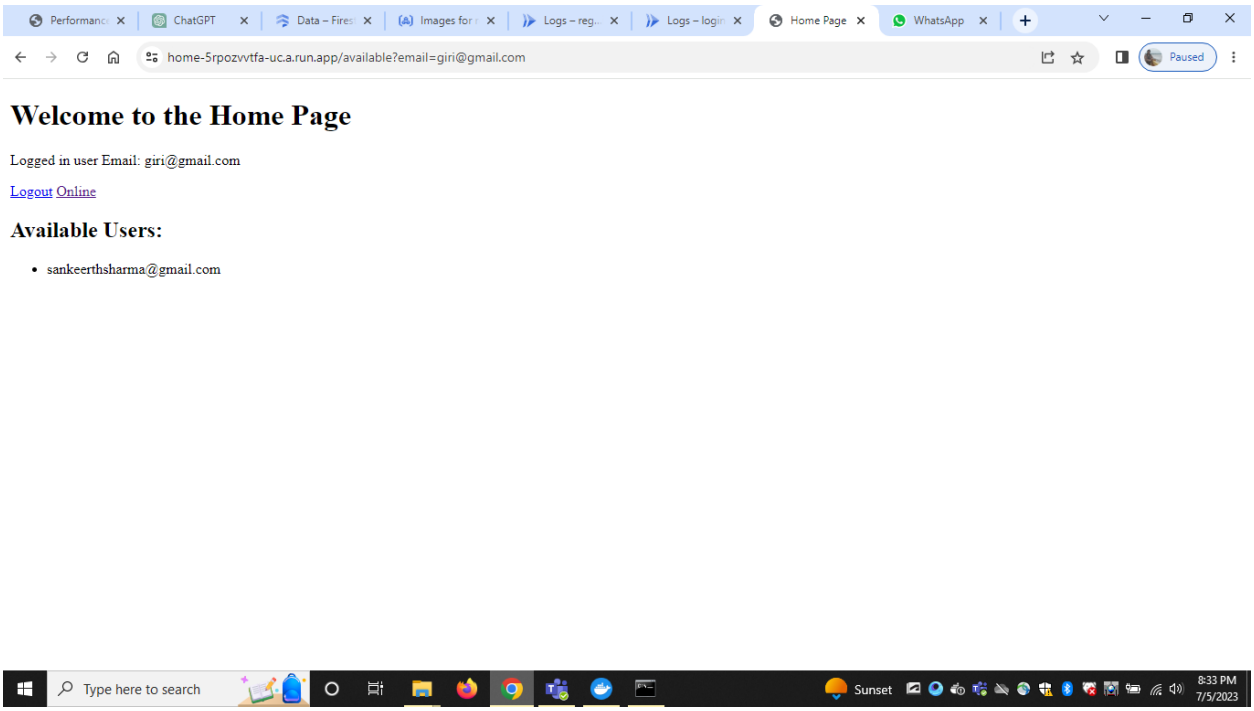


Figure 18: Displaying available users in home page

18. Reference that user Tupper is not logged in

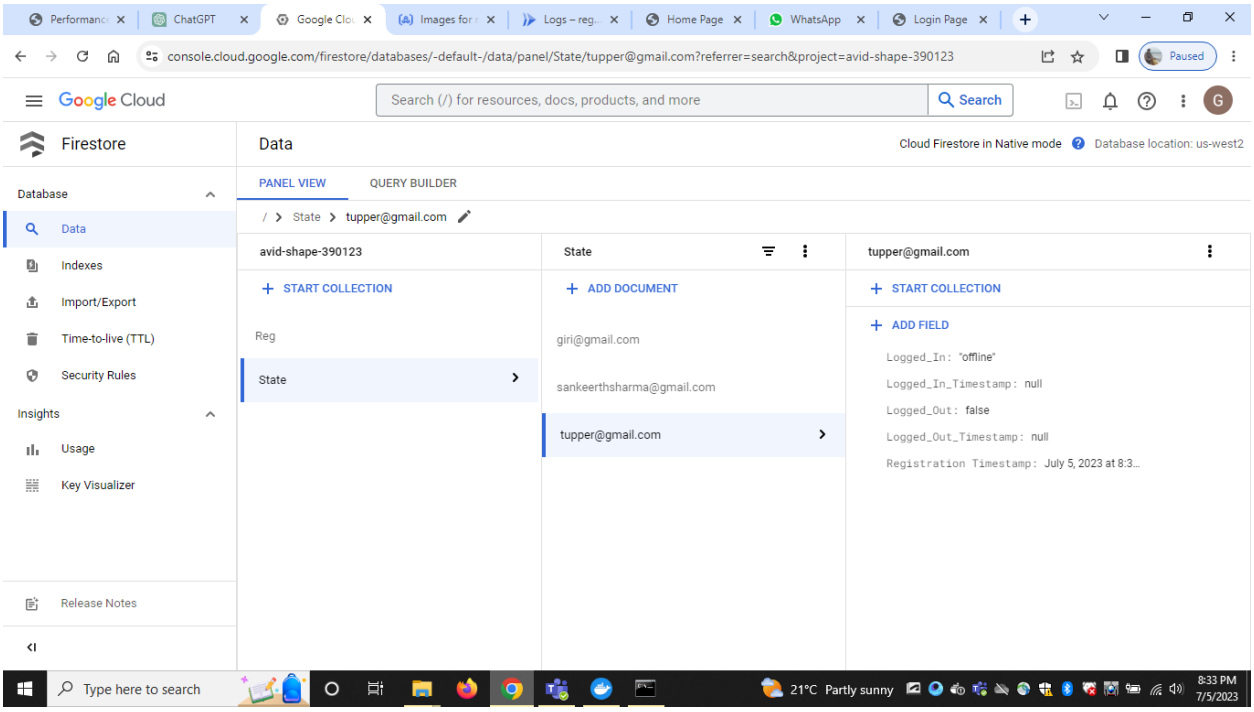


Figure 19: Reference that user is not logged in

19. Logs from cloud run of Login service

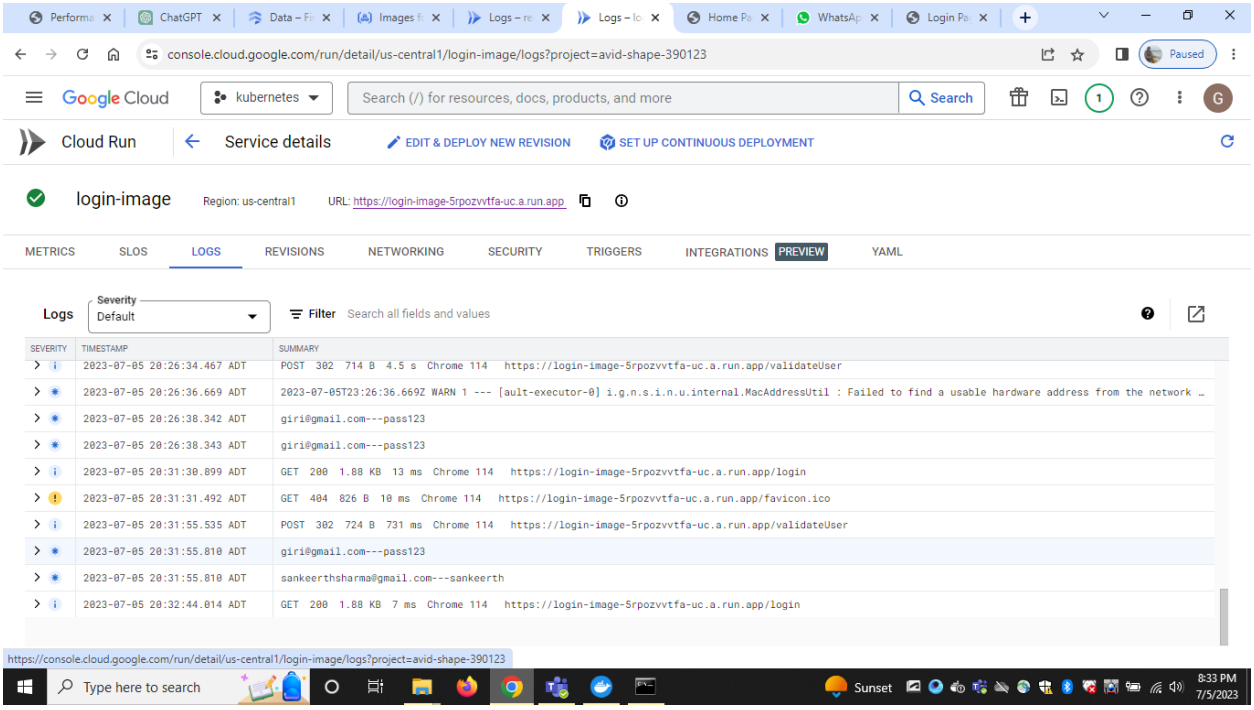


Figure 20: Logs of cloud run login-image service

20. Logs from cloud run of home page service

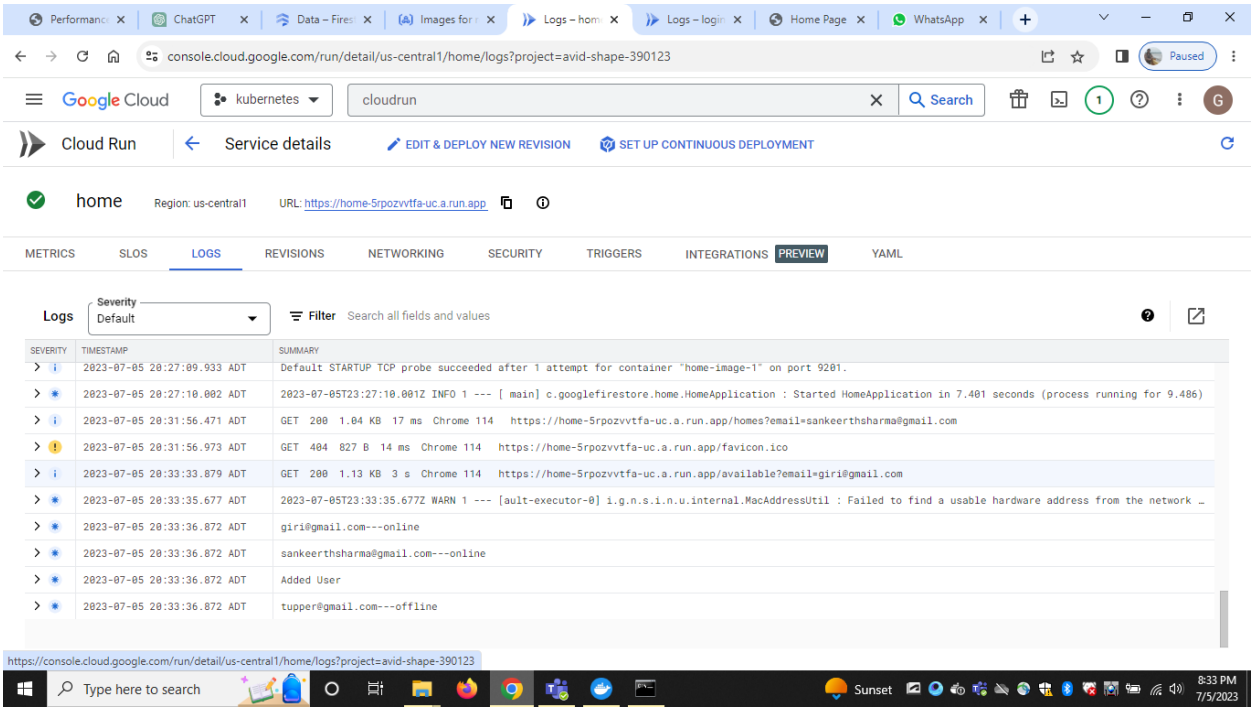


Figure 21: Logs of cloud run home service

21. Finally, the user can log out by clicking on button “logout” provided in home page

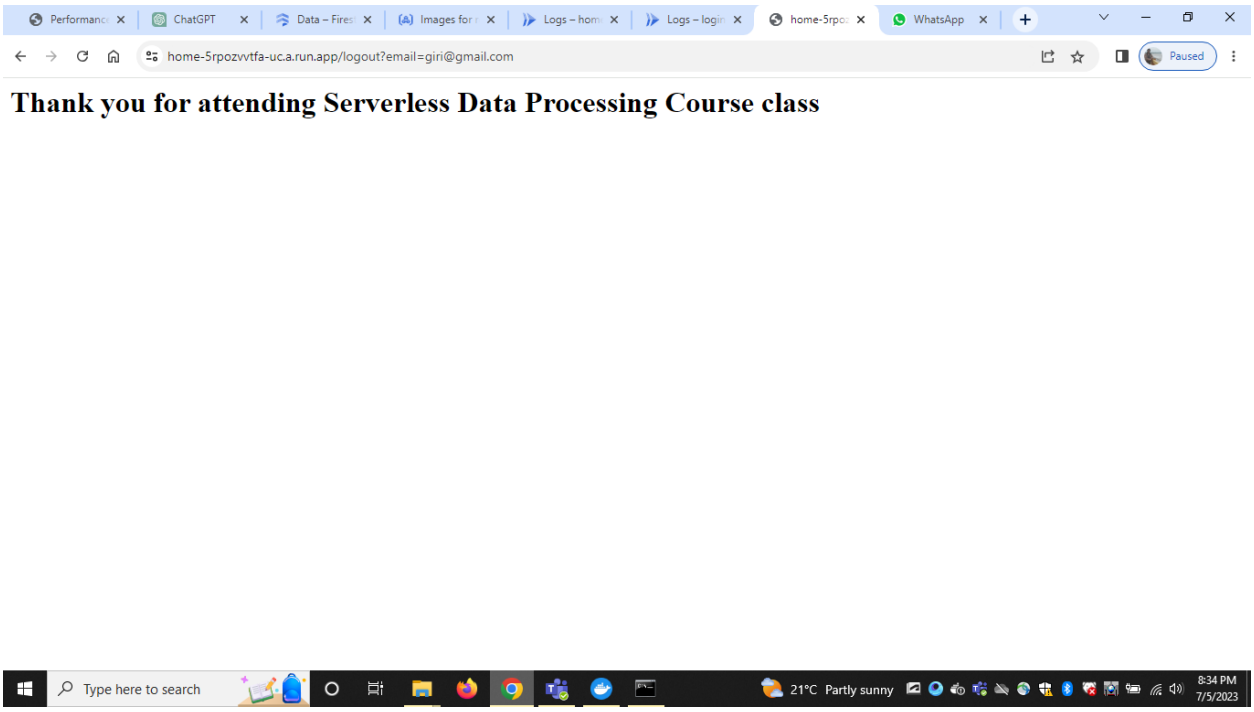


Figure 22: Display after user logout

22. Fields got updated after user gets signed out

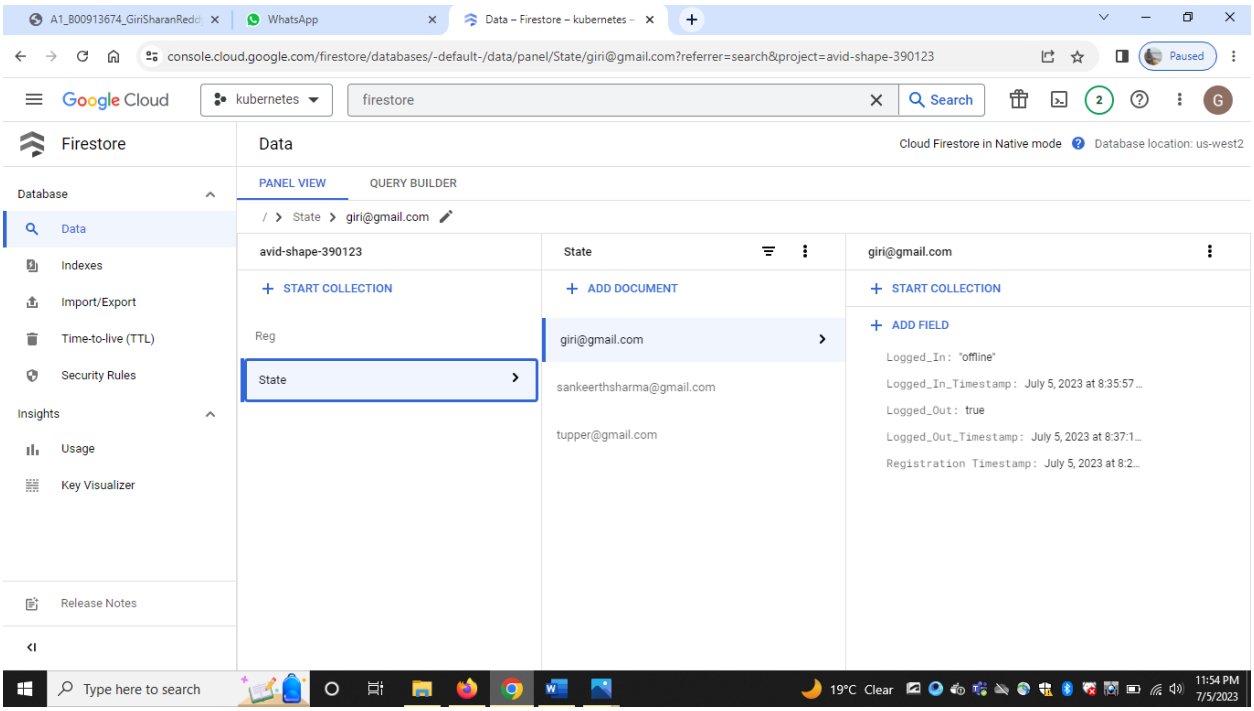


Figure 23: Fileds getting updated after user logout

Google Cloud Run is a powerful serverless compute platform that allows developers for building and deploying applications quickly and easily, without the need to manage infrastructure or worry about scaling, pay for the exact resources our application consumes during its execution. Google Cloud Run is apart from other similar services because of its ability to run both stateless HTTP-driven containers and long-running background tasks, providing developers with great flexibility[3]. Additionally, Cloud Run offers auto-scaling capabilities, ensuring that our application scales seamlessly to handle any level of traffic. It integrates seamlessly with other Google Cloud services, such as Cloud Build for automated builds and deployments. Other notable features of Google Cloud Run include easy deployment through a simple command-line interface or via continuous integration and delivery pipelines [3]. I have used to Continuous deploy feature of cloud run where when I push my code to Github then it will automatically build docker image and stores that in artifact registry and then deploy in Cloud run.

Yaml file used to build the pipeline

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: firestore
  namespace: '213864788863'
```

```
  selfLink:
/apis/serving.knative.dev/v1/namespaces/213864788863/services/firestore
  uid: a93118ce-f2c0-476b-832f-9ef64cf73f7c
  resourceVersion: AAX/yStraqU
  generation: 6
  creationTimestamp: '2023-07-06T03:08:04.335892Z'
  labels:
    managed-by: gcp-cloud-build-deploy-cloud-run
    gcb-trigger-id: c0151ee1-0fff-4371-8211-53f7d15033d8
    gcb-trigger-region: global
    commit-sha: 75d30bee54826e87f988870ee78e102b5d1aaba4
    gcb-build-id: e8fc1074-9300-4184-a55f-cc02997b1b8f
    cloud.googleapis.com/location: us-central1
  annotations:
    run.googleapis.com/client-name: gcloud
    serving.knative.dev/creator: giri.pusuluru@gmail.com
    serving.knative.dev/lastModifier: 213864788863@cloudbuild.gserviceaccount.com
    run.googleapis.com/client-version: 437.0.1
    run.googleapis.com/operation-id: d0e20354-e625-4e14-96e9-babc1909b59f
    run.googleapis.com/ingress: all
    run.googleapis.com/ingress-status: all
spec:
  template:
    metadata:
      name: firestore-00006-pex
      labels:
        managed-by: gcp-cloud-build-deploy-cloud-run
        gcb-trigger-id: c0151ee1-0fff-4371-8211-53f7d15033d8
        gcb-trigger-region: global
        commit-sha: 75d30bee54826e87f988870ee78e102b5d1aaba4
        gcb-build-id: e8fc1074-9300-4184-a55f-cc02997b1b8f
        run.googleapis.com/startupProbeType: Default
      annotations:
        run.googleapis.com/client-name: gcloud
        run.googleapis.com/client-version: 437.0.1
        autoscaling.knative.dev/maxScale: '100'
    spec:
      containerConcurrency: 80
      timeoutSeconds: 300
      serviceAccountName: 213864788863-compute@developer.gserviceaccount.com
      containers:
      - image: us-central1-docker.pkg.dev/avid-shape-390123/cloud-run-source-
deploy/firestore/firestore:75d30bee54826e87f988870ee78e102b5d1aaba4
        ports:
        - name: http1
```

```
          containerPort: 9001
        resources:
          limits:
            cpu: 1000m
            memory: 512Mi
        startupProbe:
          timeoutSeconds: 240
          periodSeconds: 240
          failureThreshold: 1
          tcpSocket:
            port: 9001
  traffic:
  - percent: 100
    latestRevision: true
status:
  observedGeneration: 6
  conditions:
  - type: Ready
    status: 'True'
    lastTransitionTime: '2023-07-06T03:32:45.427869Z'
  - type: ConfigurationsReady
    status: 'True'
    lastTransitionTime: '2023-07-06T03:32:21.709797Z'
  - type: RoutesReady
    status: 'True'
    lastTransitionTime: '2023-07-06T03:32:45.522597Z'
  latestReadyRevisionName: firestore-00006-pex
  latestCreatedRevisionName: firestore-00006-pex
  traffic:
  - revisionName: firestore-00006-pex
    percent: 100
    latestRevision: true
  url: https://firestore-5rpozvvtfa-uc.a.run.app
  address:
    url: https://firestore-5rpozvvtfa-uc.a.run.app
```

Artifact Registry is a tool provided by Google Cloud designed specifically for securely storing and managing software artifacts like container images and packages. It integrates with popular build and deployment tools, allowing easy building and deploying of applications. It offers fine-grained advanced features for artifact management, such as access controls, versioning, and tagging, and integrates seamlessly with Google Cloud services for easy deployment within a cloud infrastructure [4].

Docker Containers provide a lightweight and portable environment that encapsulates all the dependencies and components required to run an application, making it consistent and reliable across different environments. One key advantage of Docker containers is their isolation, ensuring that each container operates independently, preventing conflicts and providing security. Containers are highly scalable, allowing applications to be seamlessly scaled up or down based on demand [4]. They promote efficient resource utilization by running multiple containers on a single host, maximizing hardware efficiency. With Docker, applications become easily portable, as containers can be run on any system with Docker installed, irrespective of the underlying infrastructure [4].

Test case of registration form code:

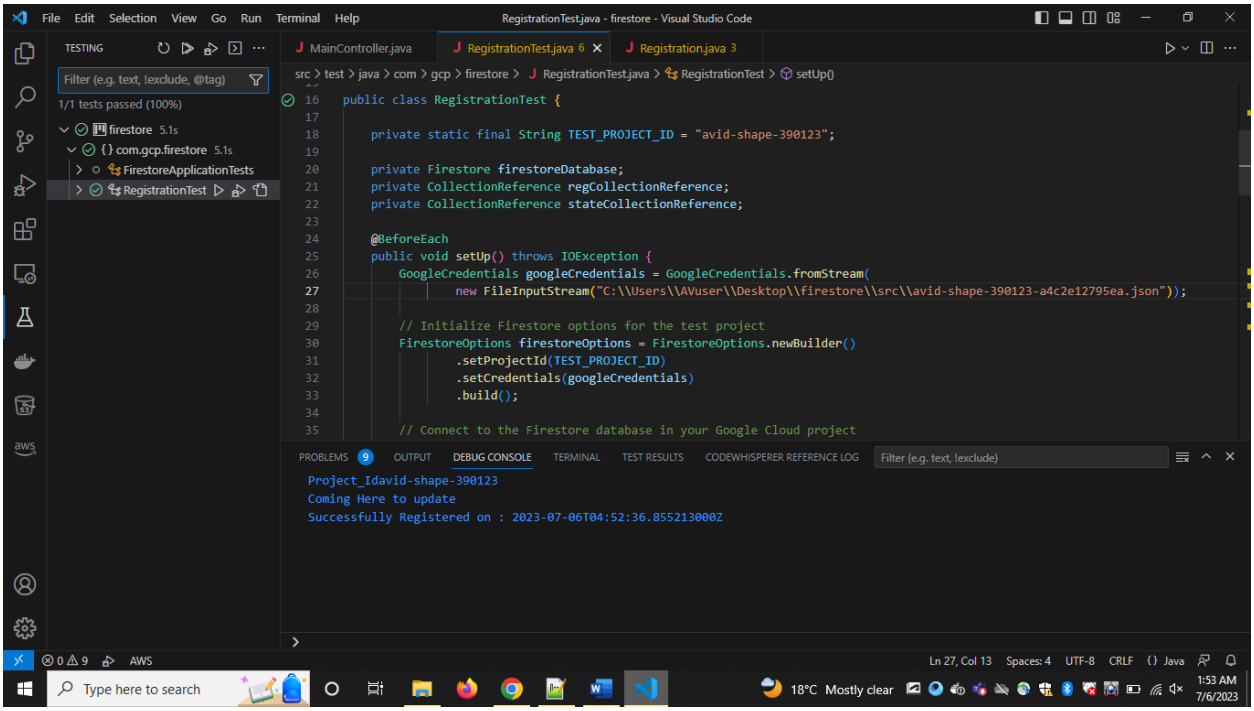


Figure 24: Test case for registration of a user

Test case of login page validation code:

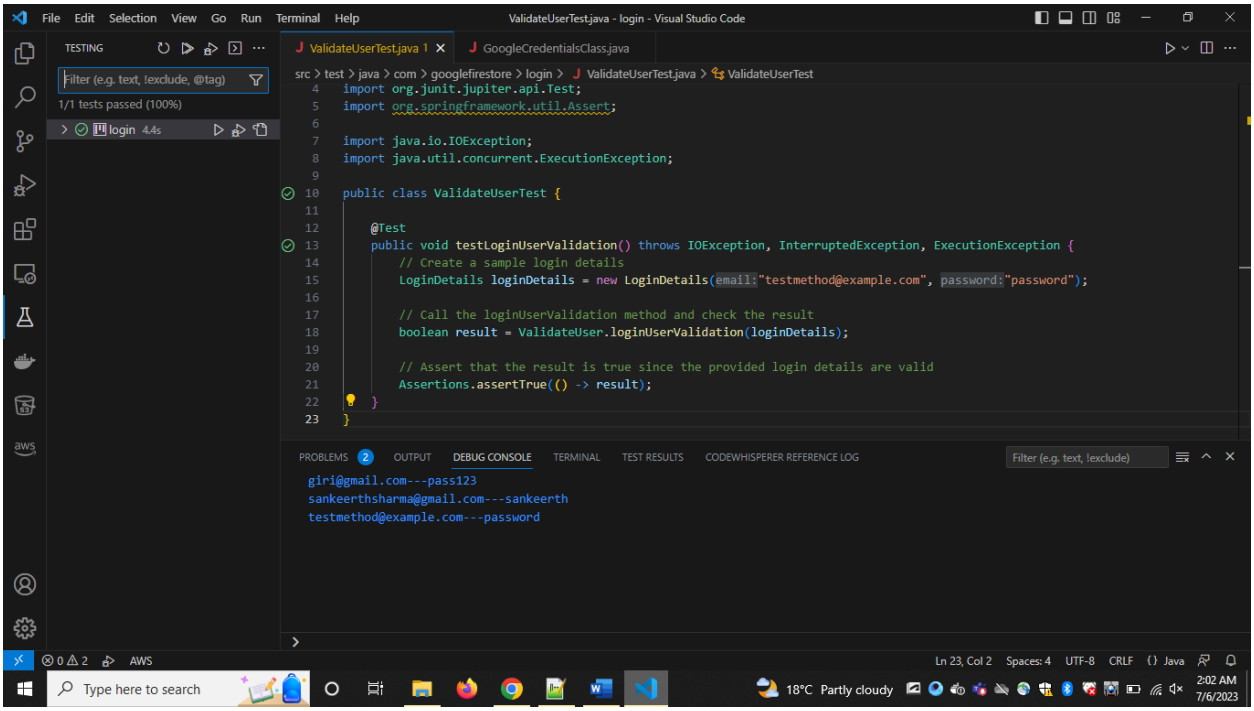


Figure 25: Test case for login of a user

Reference that user is created from test case of registration code

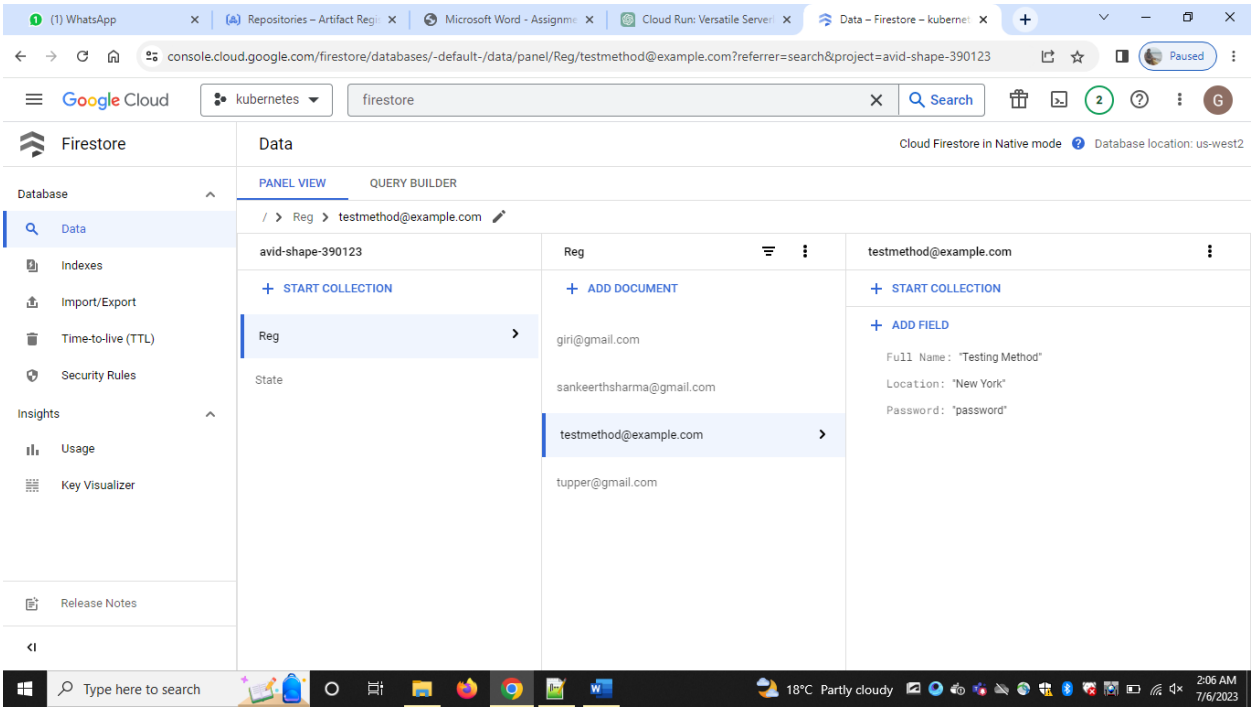
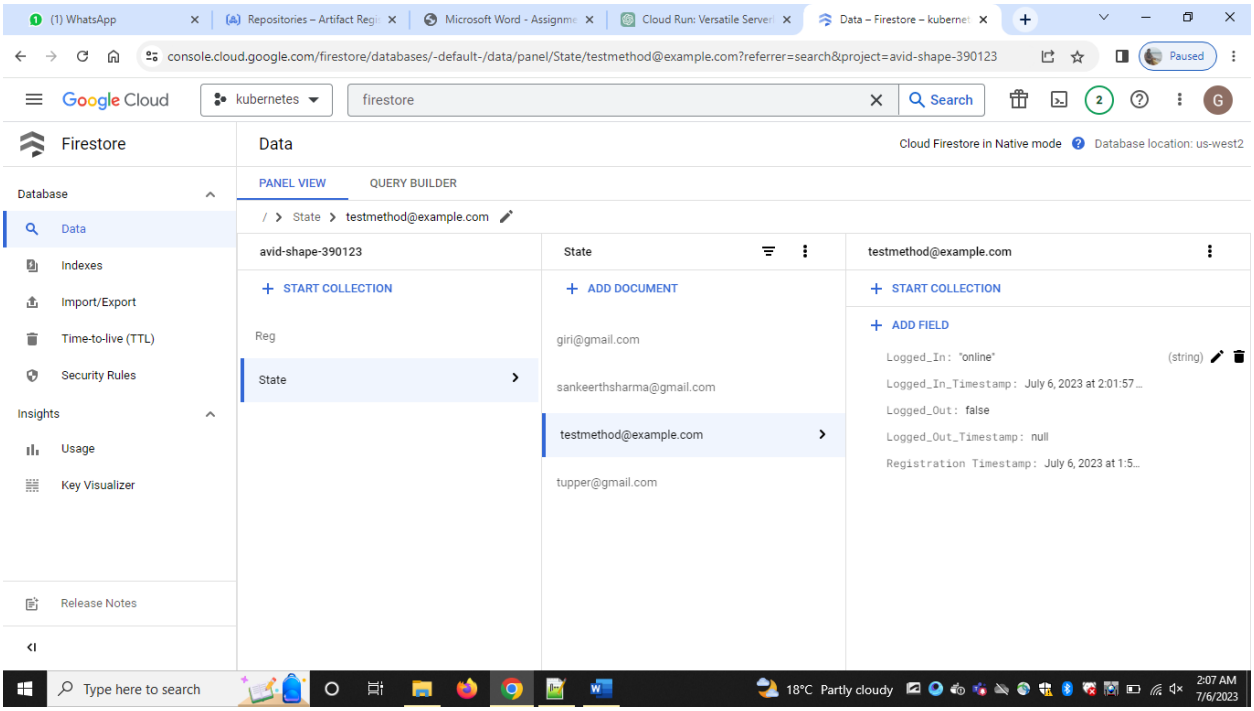


Figure 26: user details in firestore

Reference that user login details are validated successfully and the status of user is changed



Test case of checking available online users

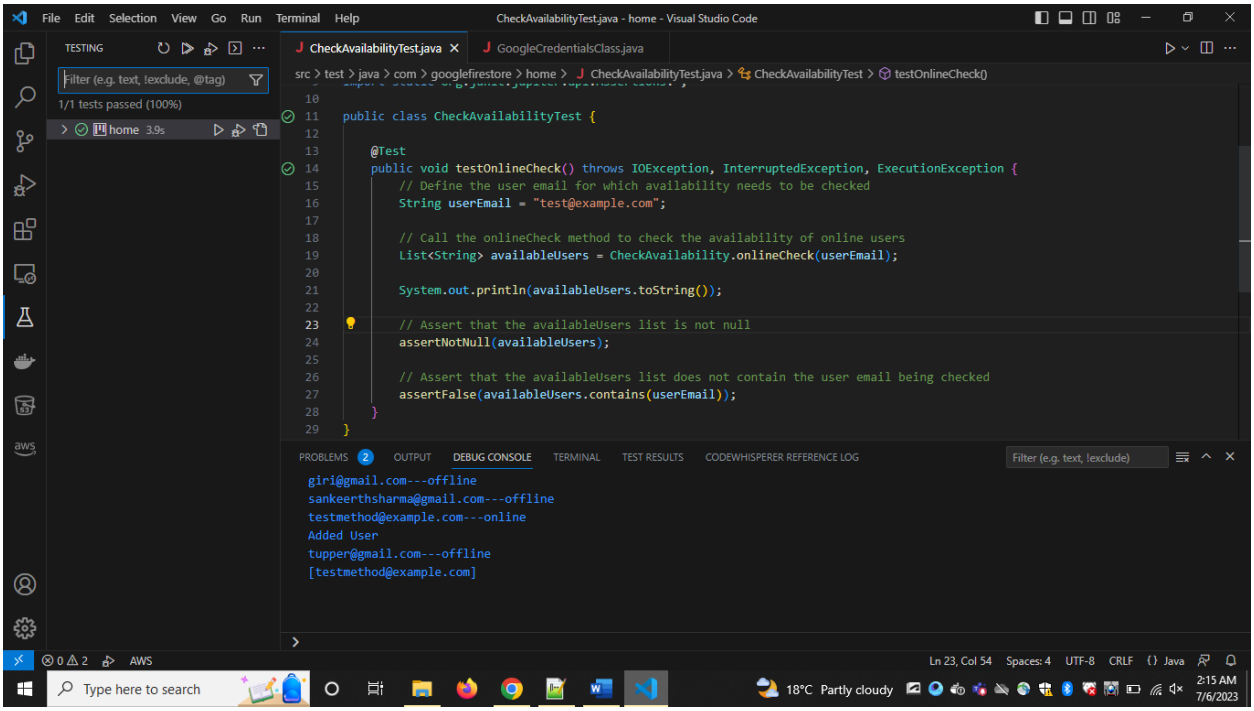


Figure 27: Test case for checking available online users

PART-C REPORT

Ride Request chat bot is built where it takes fulfills two types of requests

1) Taxi service booking
2) Self-driving car booking

Basic Terminology

Intent – An intent represents an action that the user wants to perform like booking a cab.

Utterances - These are the textual representations of what a user types or says in order to request a service. One intent can contain many different utterances, allowing users to trigger the bot using different phrases.

Slot - A slot type is a list of value(s) that Amazon Lex takes as input and are used to validate, compare or fulfill request.

Initial Greeting

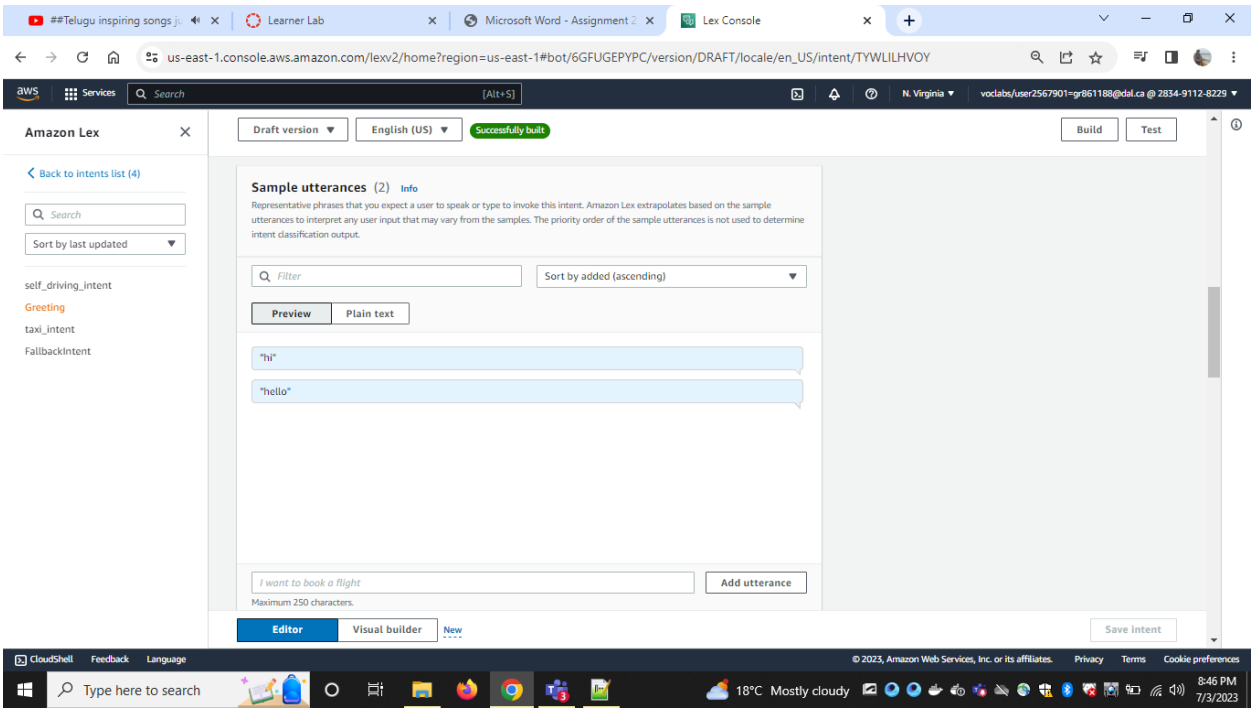


Figure 28: Greeting utterances

Reply for greeting utterances

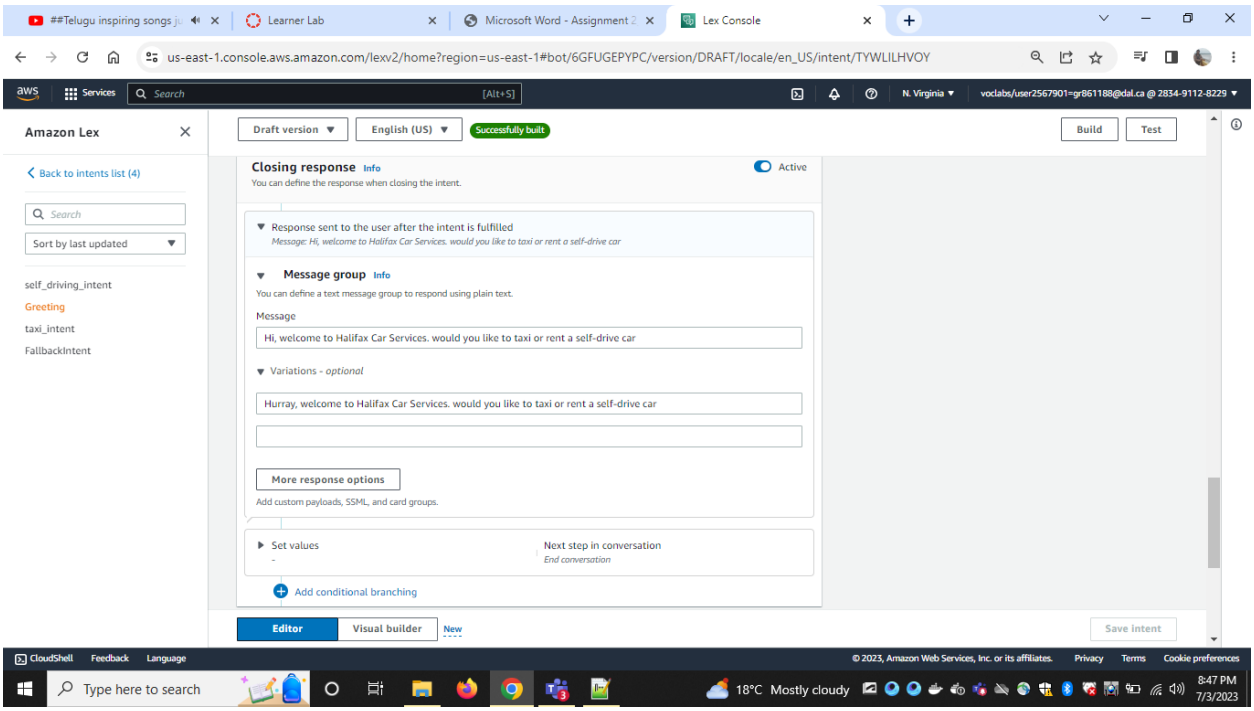


Figure 29: Reply for greeting utterances

Steps Involved in Building Taxi Service Bot

1. Sample Utterances for accepting the taxi/cab request

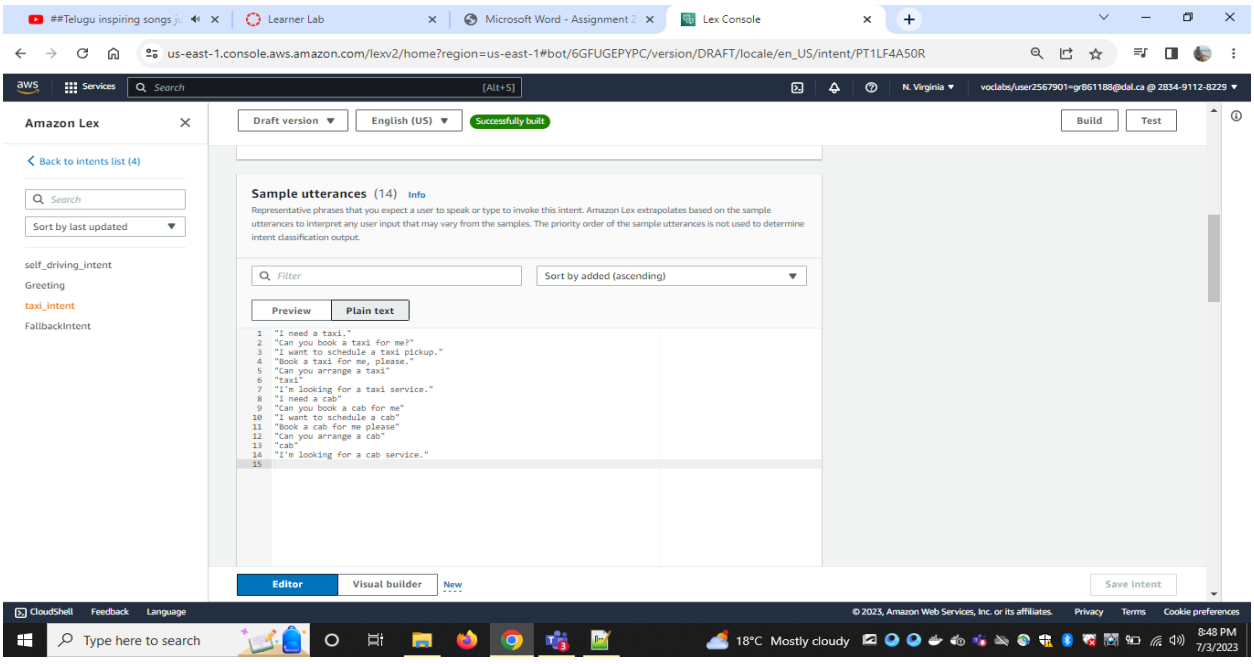


Figure 30: Sample utterances

2. Initial response from bot once user has requested for taxi service

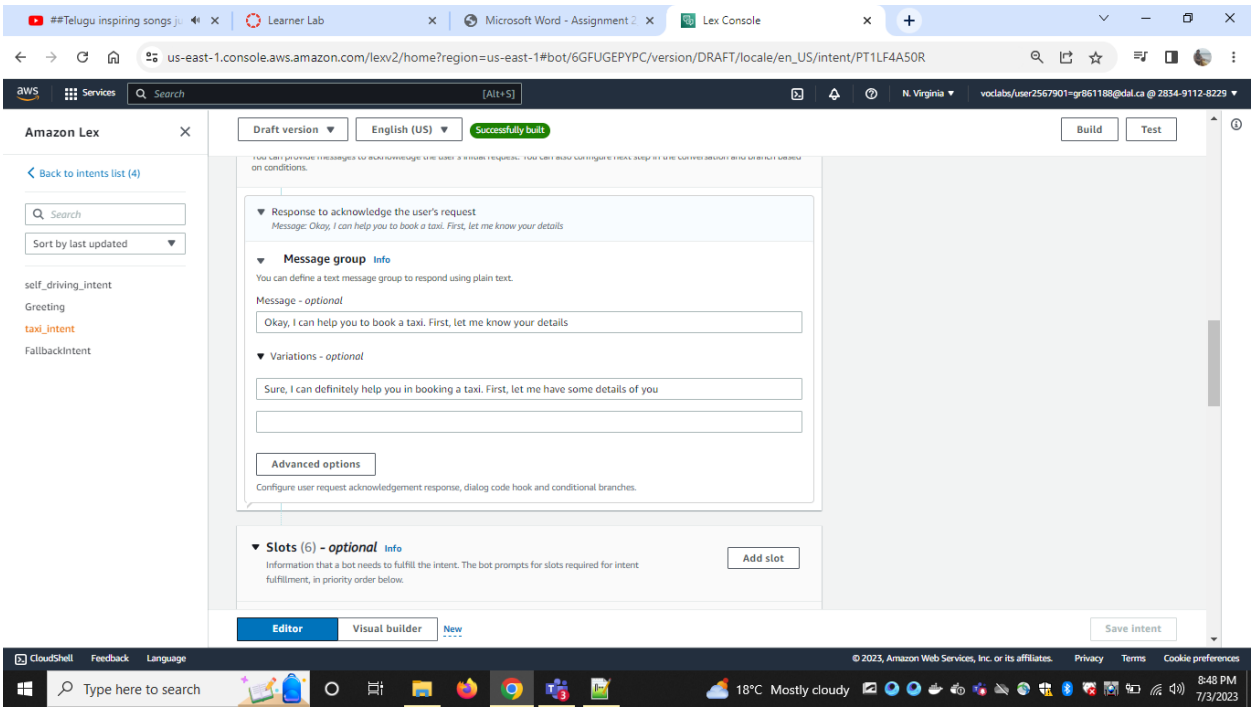


Figure 31: Initial response from bot

3. Inputs that are required in order to book a taxi

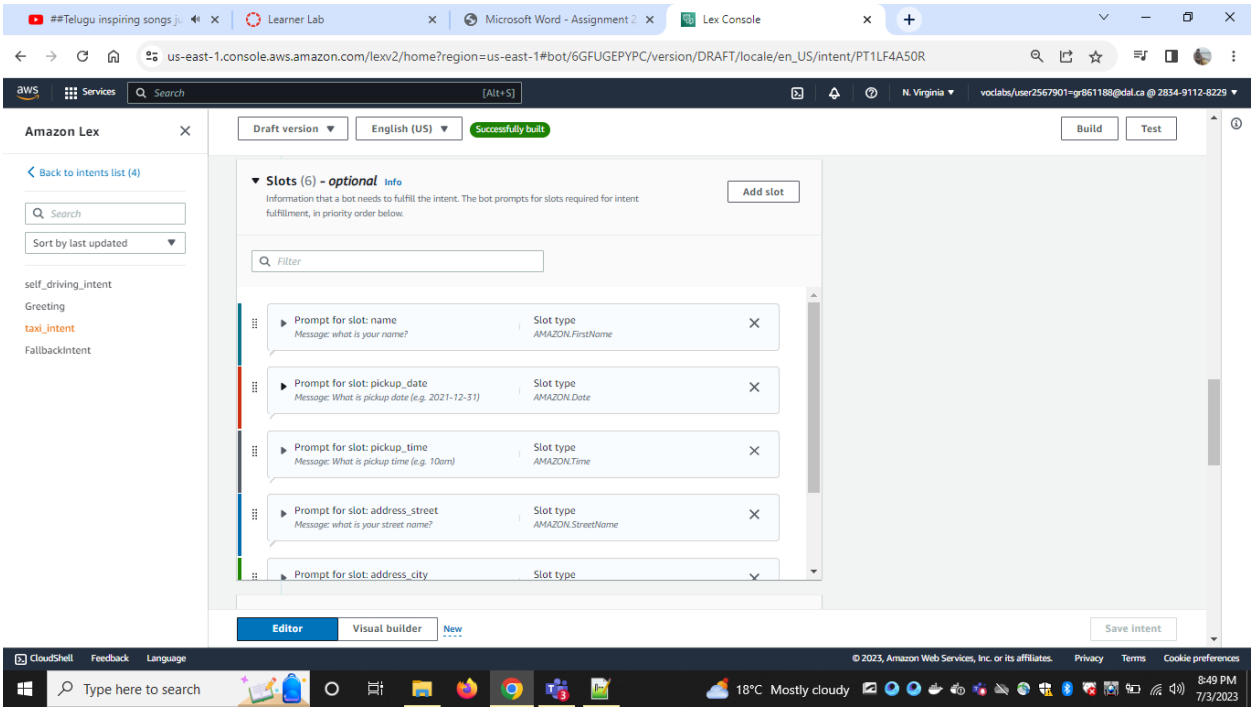


Figure 32: Slots for booking taxi

4. Once user provided all the slots, bot asks for confirmation

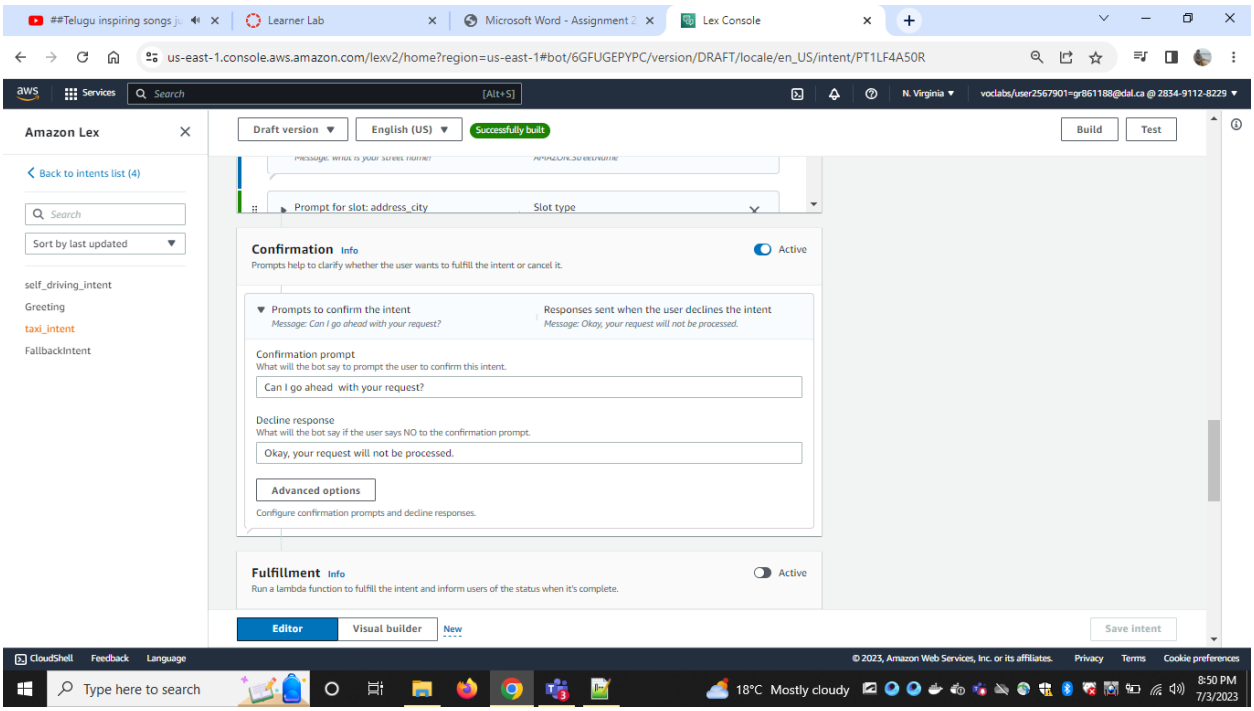


Figure 33: Request for getting confirmation

5. After confirmation, closing response is provided

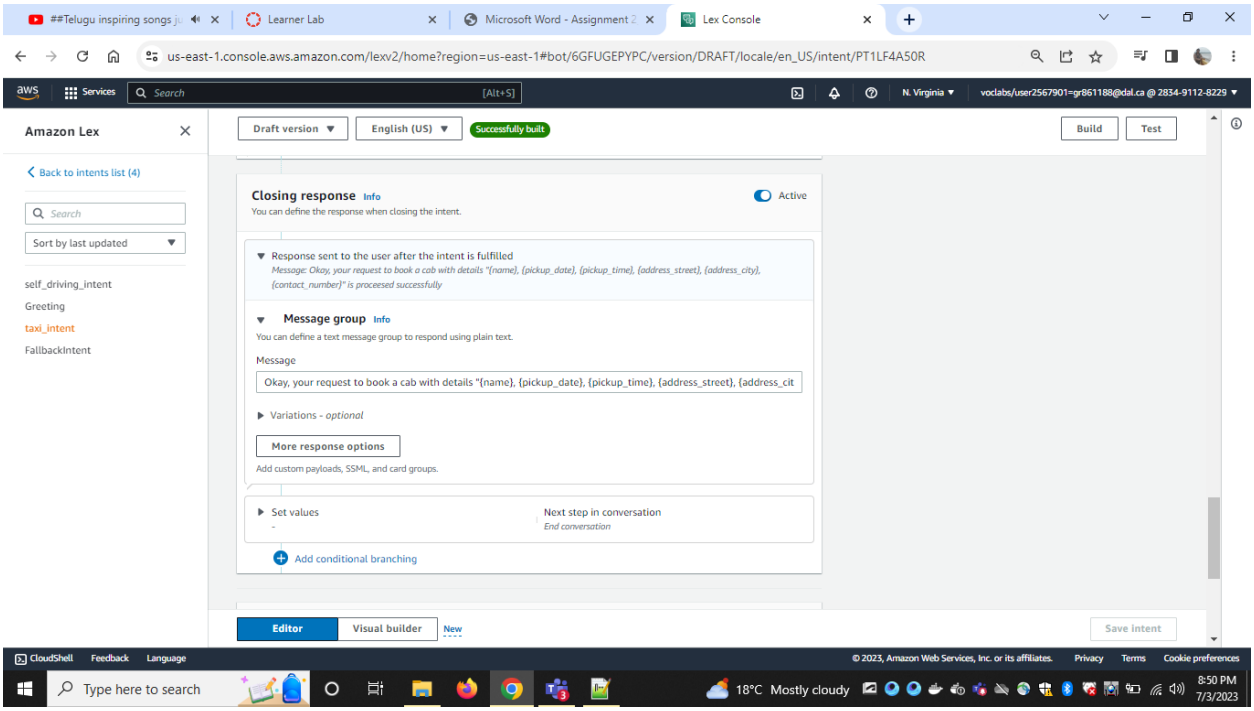


Figure 34: Closing message after service fulfillment

Taxi Booking Bot Execution

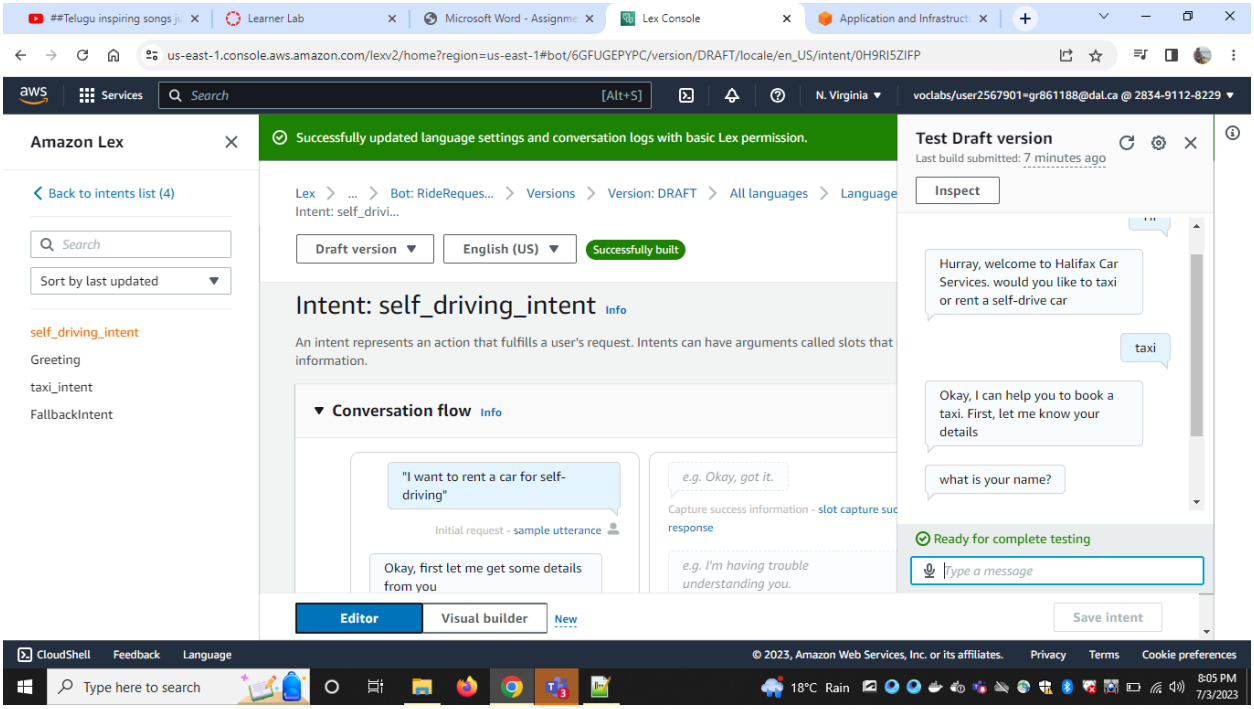


Figure 35: Bot exection-1

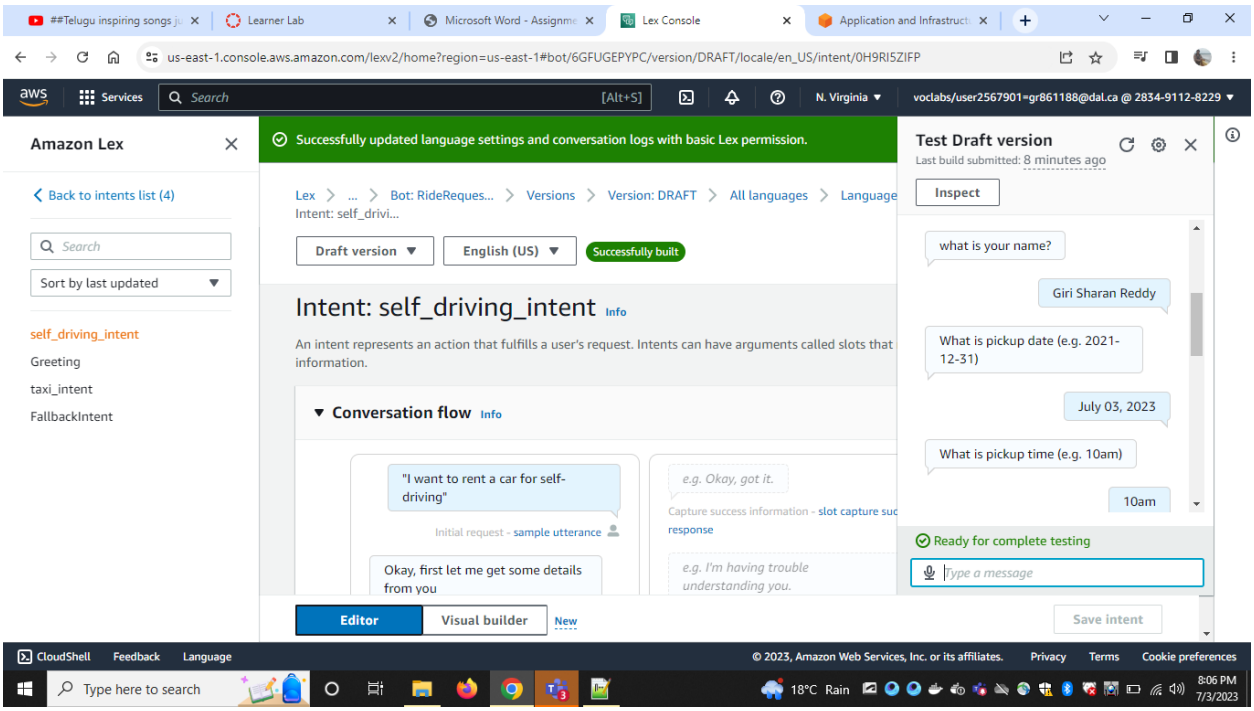


Figure 36: Bot execution-2

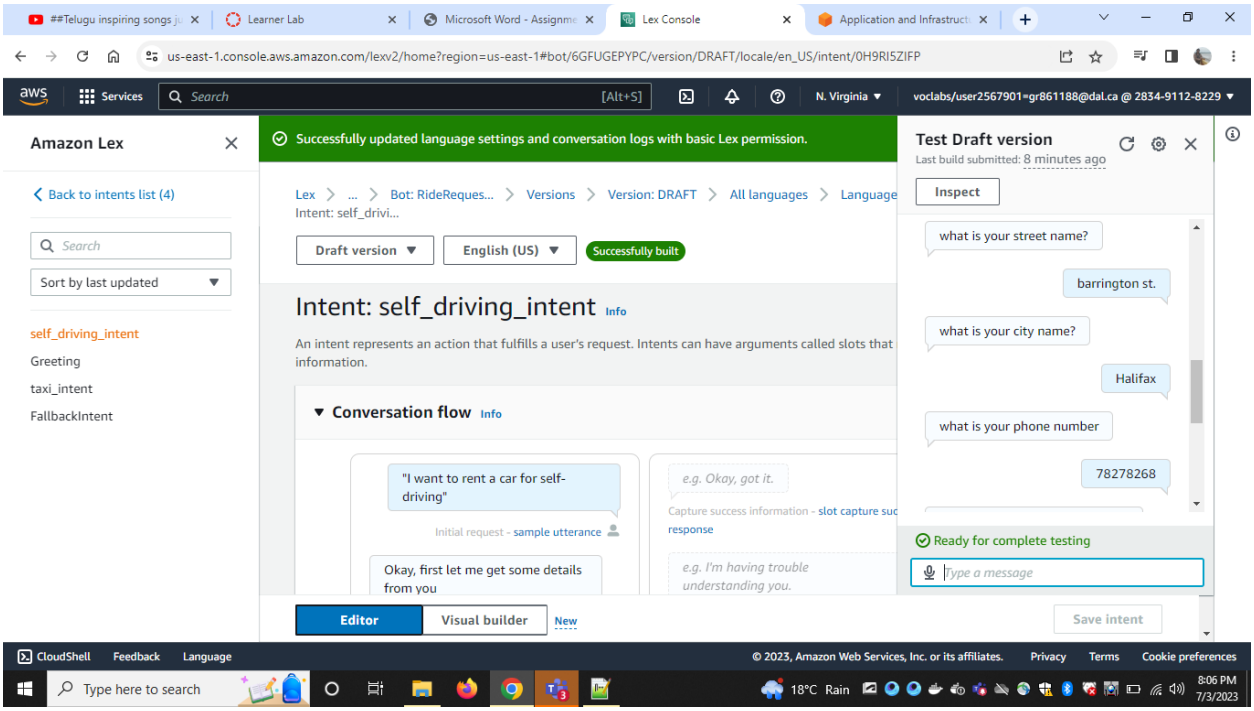


Figure 37: Bot execution-3

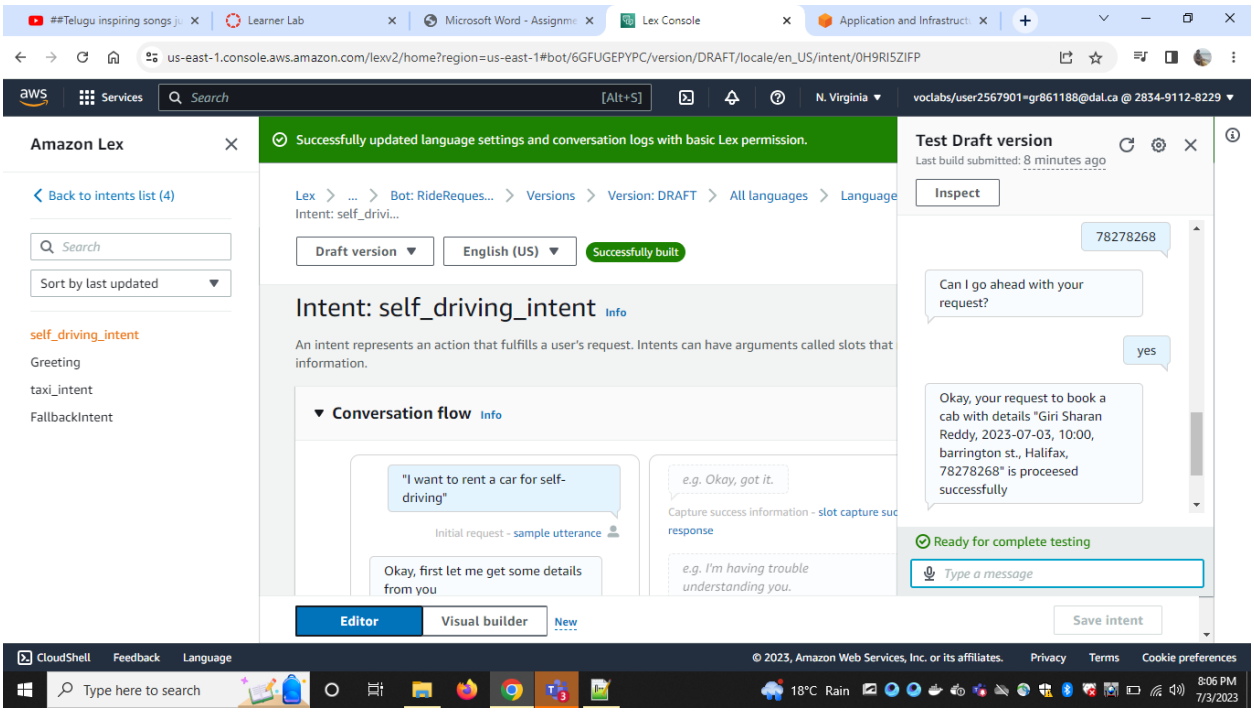


Figure 38: Bot exection-4

Steps Involved in Building Self-driving Car Booking Service Bot

1. Sample Utterances for accepting the self-driving car request

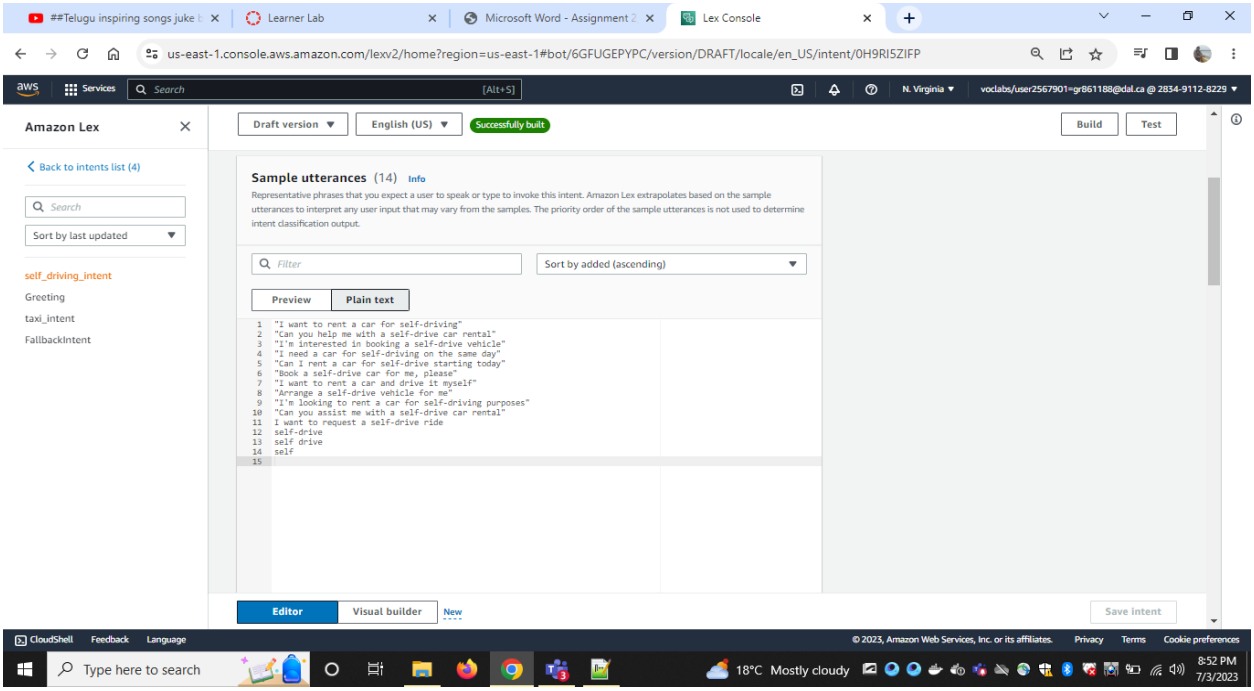


Figure 39: Sample utterances

2. Inputs that are required in order to book a taxi

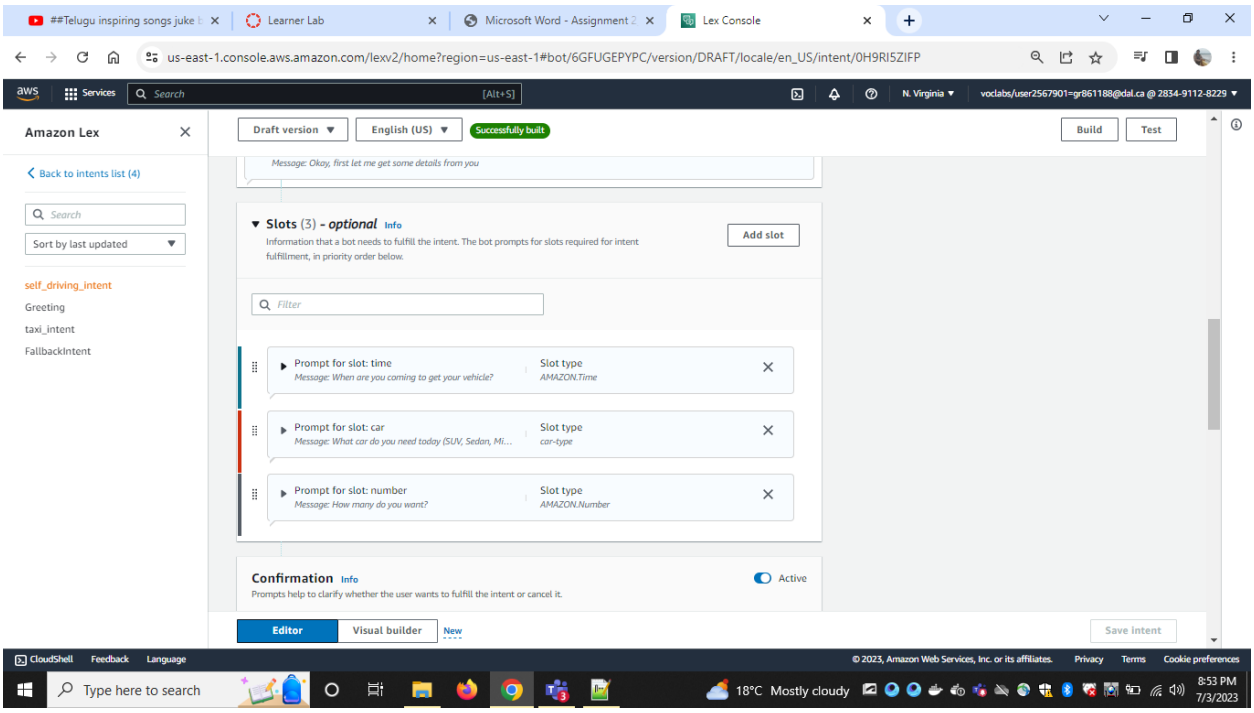


Figure 40: Slots for self-driving car service

3. Once user provided all the slots, bot asks for confirmation

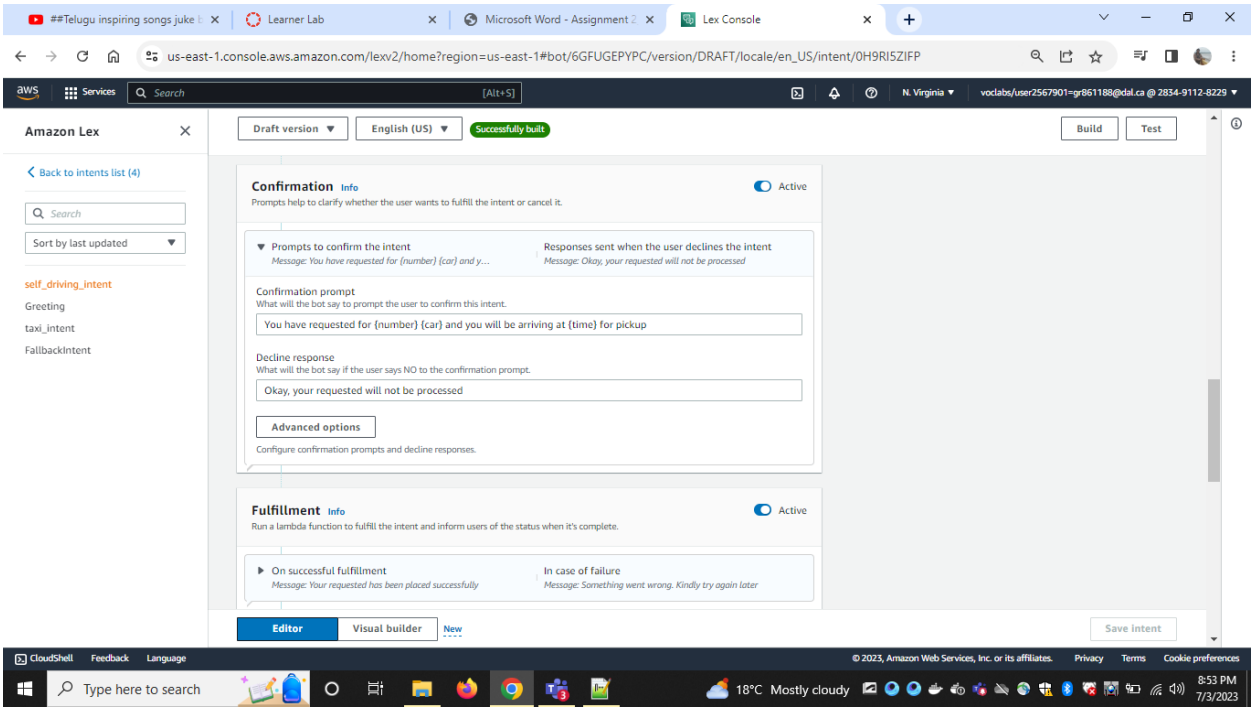


Figure 41: Request for getting confirmation

4. After confirmation, fulfillment response is provided else failure message

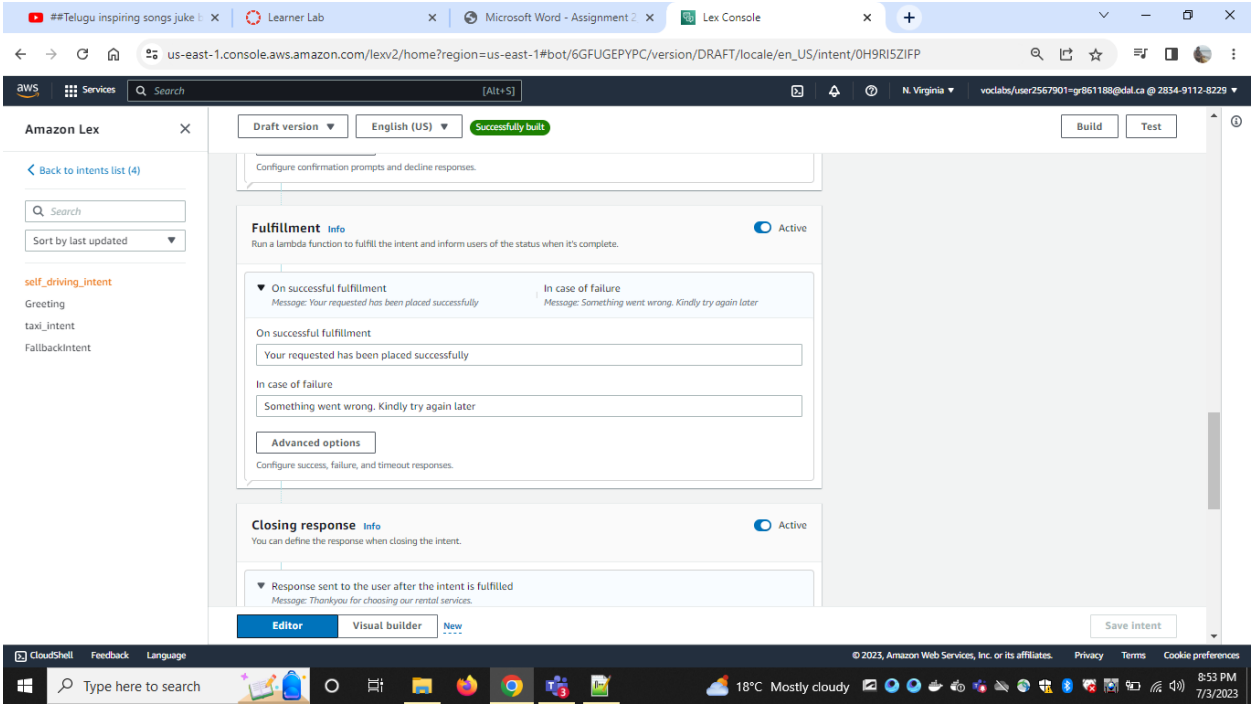


Figure 42: Fulfillment message to user

5. Closing response at the end after fulfillment of the service

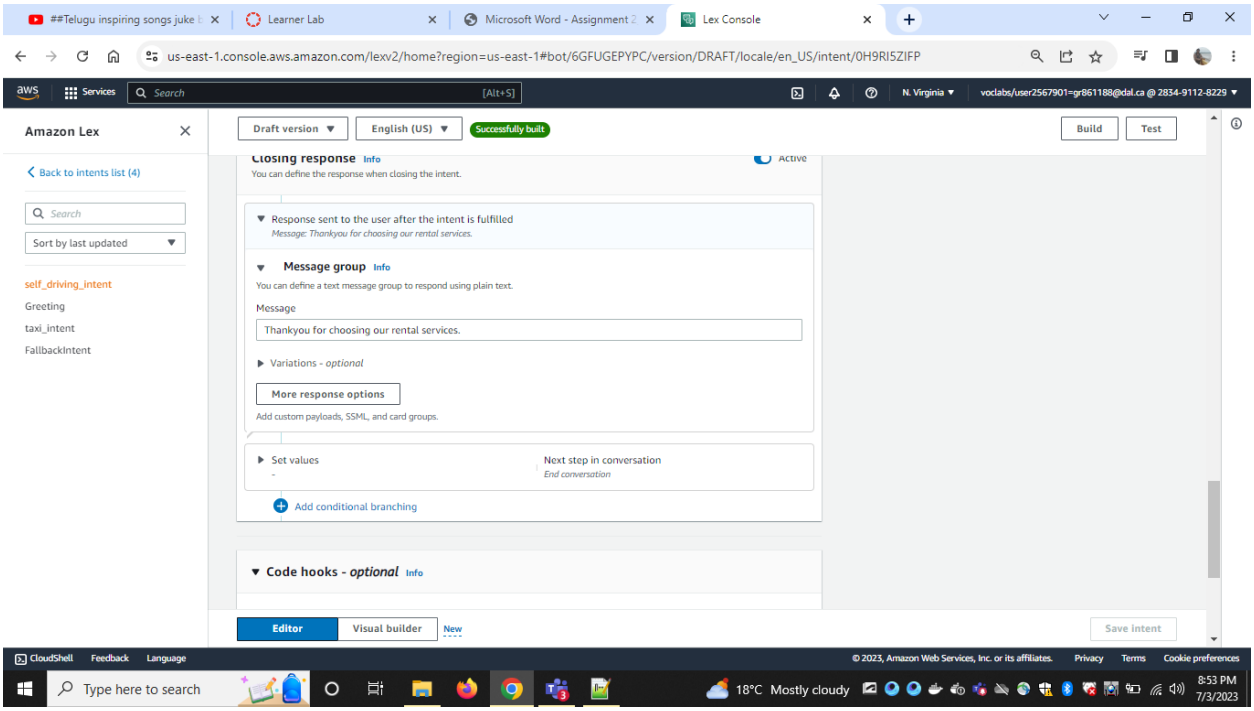


Figure 43: Closing response

Self-Driving Car Booking Bot Execution

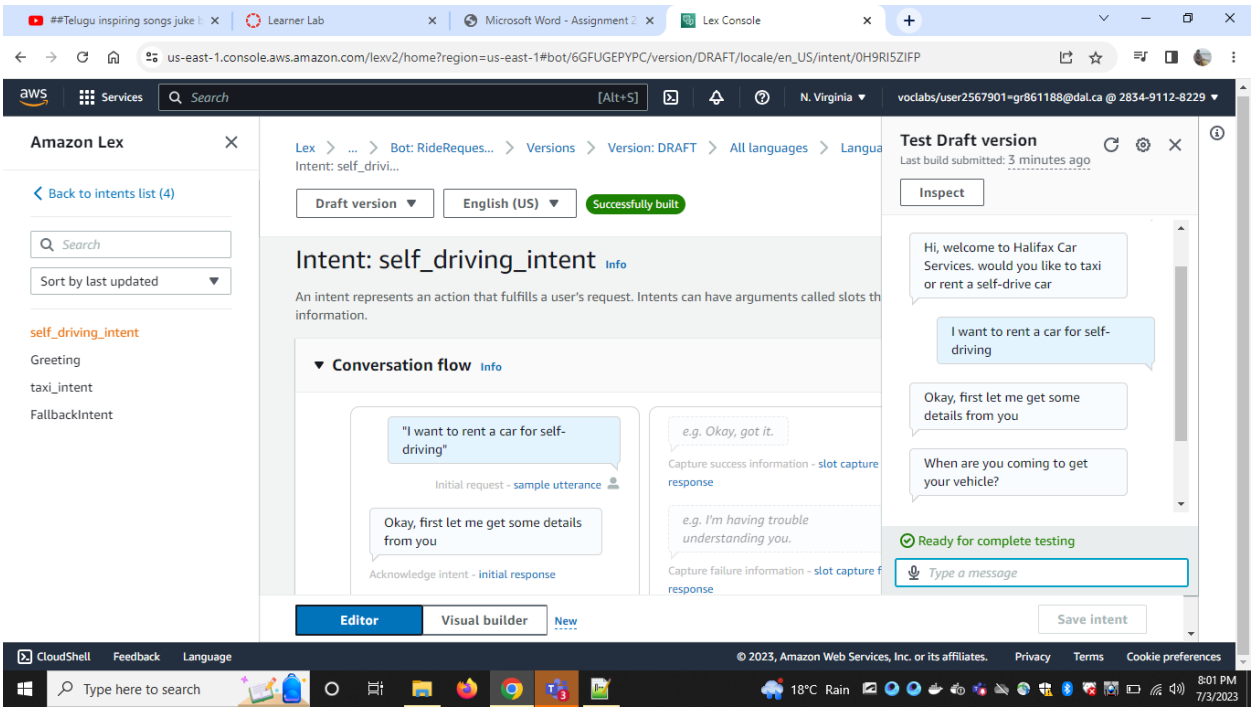


Figure 44: Bot execution-1

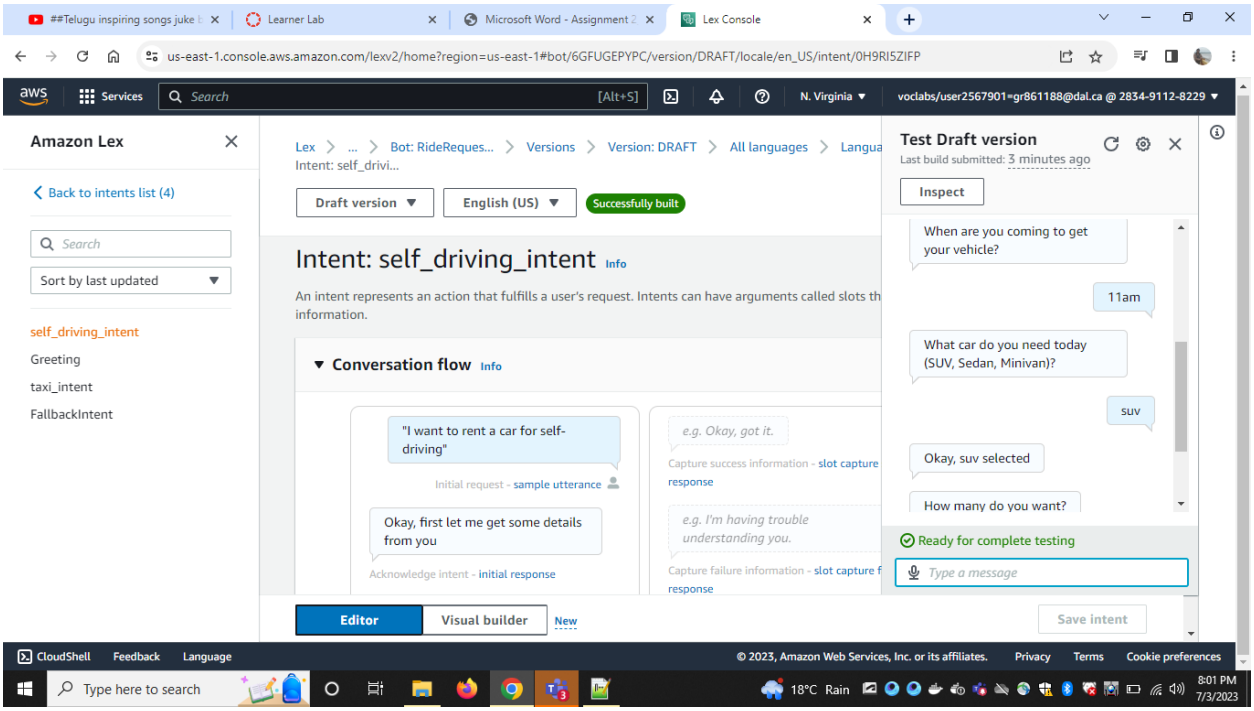


Figure 45: Bot execution-2

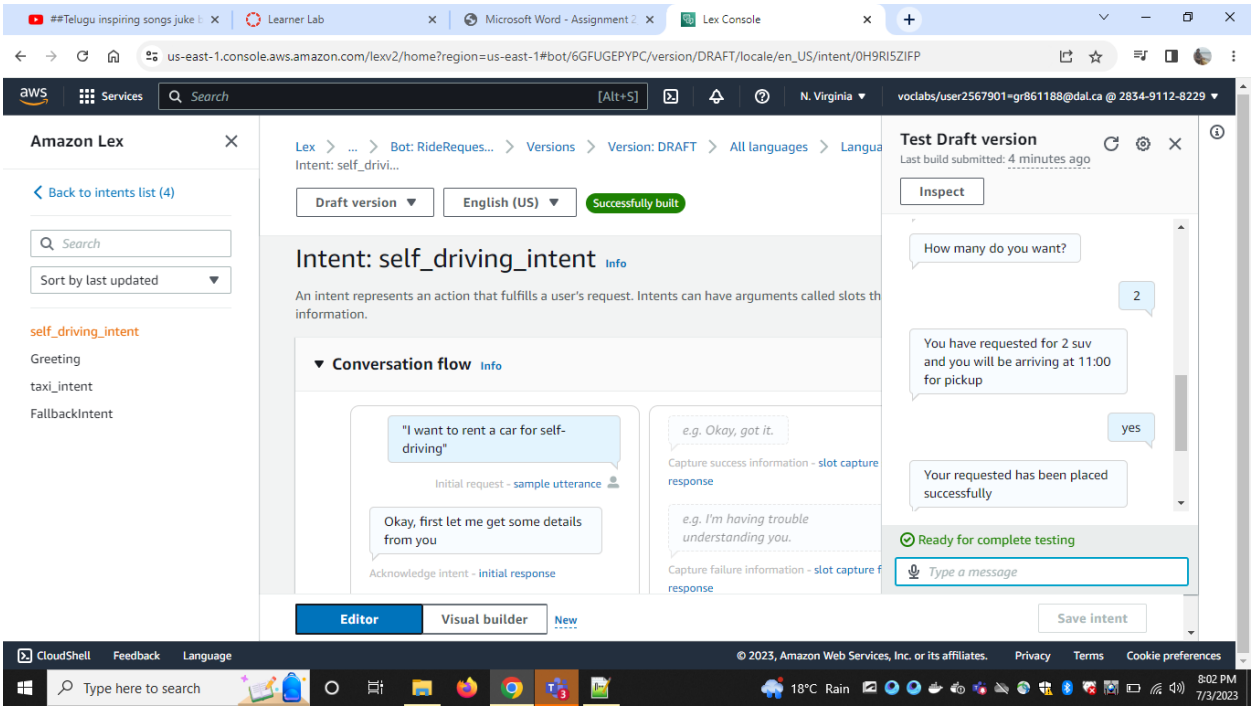


Figure 46: Bot execution-3

Procedure followed to build a bot

To build an Amazon Lex chatbot [6], we need to follow a step-by-step procedure. First, we should have the purpose of our chatbot by determining the main goal it should accomplish and the tasks or conversations it should handle. This will help us to understand the specific user inputs and intents your chatbot needs to recognize. Next, designing the conversation flow by creating a flowchart or diagram roughly that visualizes the interaction between the chatbot and users.

Once we have a clear understanding of your chatbot's purpose and conversation flow, we can proceed to set up the AWS Management Console. Sign in to the AWS Management Console and navigate to the Amazon Lex service [6].

In the Amazon Lex console, we can create a new bot by clicking on the "Create" button. Provide a name and description for our chatbot to help us in identifying and managing it later on. After creating chatbot, involves creating intents, which represent user goals or tasks, and defining sample utterances that users might those intents. We can also set up slots, which are used to gather specific pieces of information from users during the conversation.

Finally, we can test our chatbot using the built-in testing feature in the Amazon Lex console. This allows you to interact with your chatbot and see how it responds to different user inputs.

References

[1] N. Naik, “Performance evaluation of distributed systems in multiple clouds using docker swarm,” in *2021 IEEE International Systems Conference (SysCon)*, 2021, pp. 1–6.

[2] “Administer and maintain a swarm of Docker Engines,” *Docker Documentation*, 05-Jul-2023. [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/. [Accessed: 06-Jul-2023].

[3] “What is cloud run,” *Google Cloud*. [Online]. Available: https://cloud.google.com/run/docs/overview/what-is-cloud-run. [Accessed: 06-Jul-2023].

[4] “Artifact registry overview,” *Google Cloud*. [Online]. Available: https://cloud.google.com/artifact-registry/docs/overview. [Accessed: 06-Jul-2023].

[5] “What is a container?,” *Docker*, 11-Nov-2021. [Online]. Available: https://www.docker.com/resources/what-container/. [Accessed: 06-Jul-2023].

[6] *Amazon.com*. [Online]. Available: https://docs.aws.amazon.com/lexv2/latest/dg/what-is.html. [Accessed: 06-Jul-2023].