ASSIGNMENT 3 – PART A

Build an event-driven serverless application using AWS Lambda. In this assignment, you need to use AWS Lex, DynamoDb, and Lambda Functions.

GitLink:

https://git.cs.dal.ca/umatiya/csci5410-f23-b00899642/-/tree/A3

Steps I followed to build 2 ChatBots is mentioned below:

The common steps I took is given below:

Step 1:

- Create an Account and login to the AWS Academy as shown in fig 1 & 2.

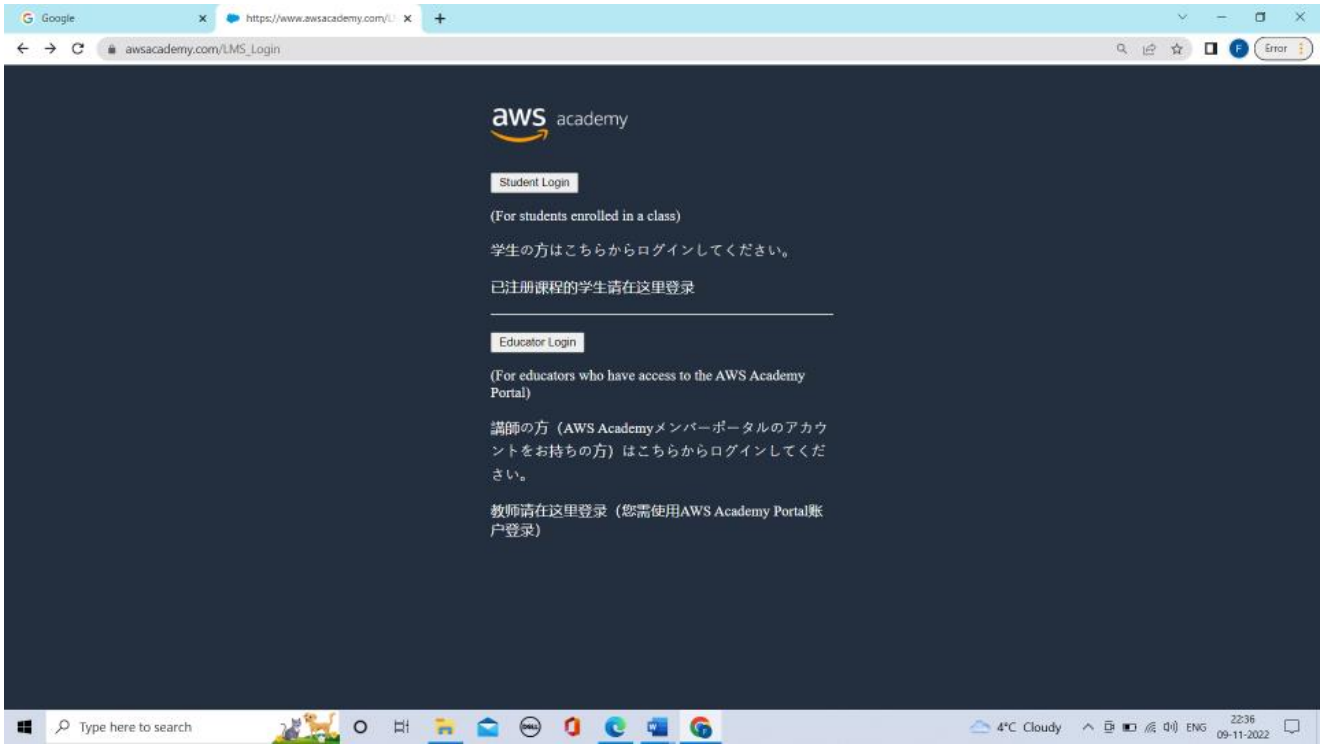

Fig 1: AWS Academy account login page

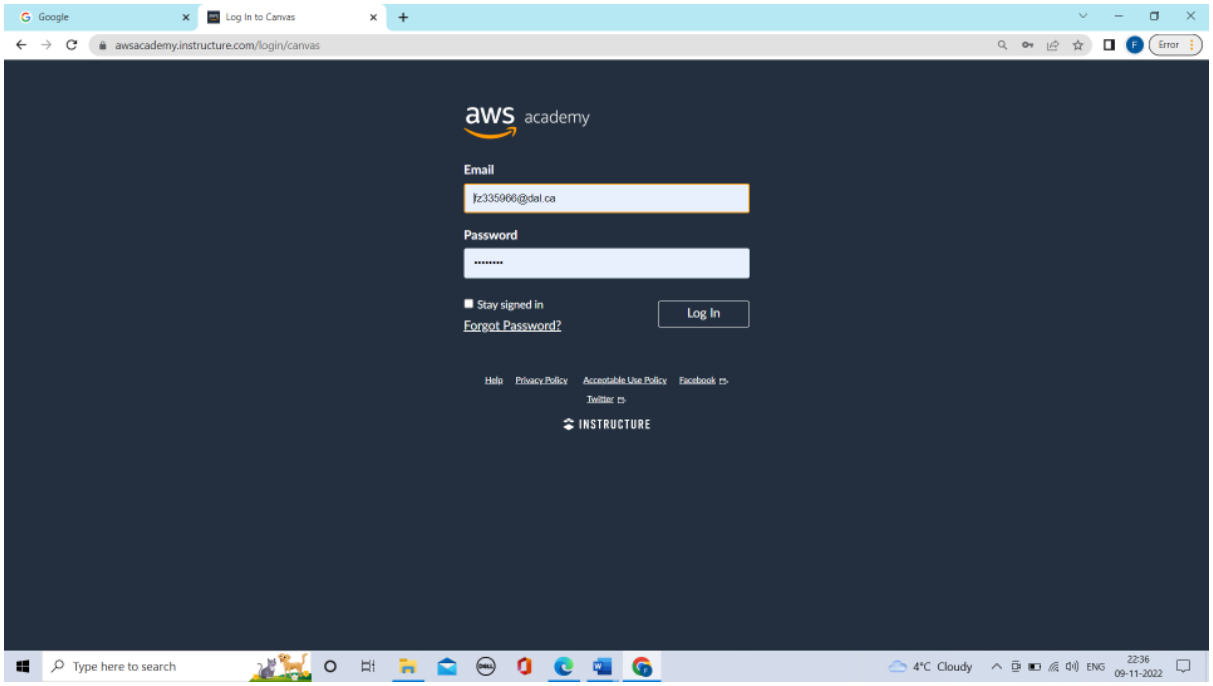

Fig 2: Student Account login page

- After login to the AWS academy, go to AWS Academy Learner Lab as shown in the fig 3.

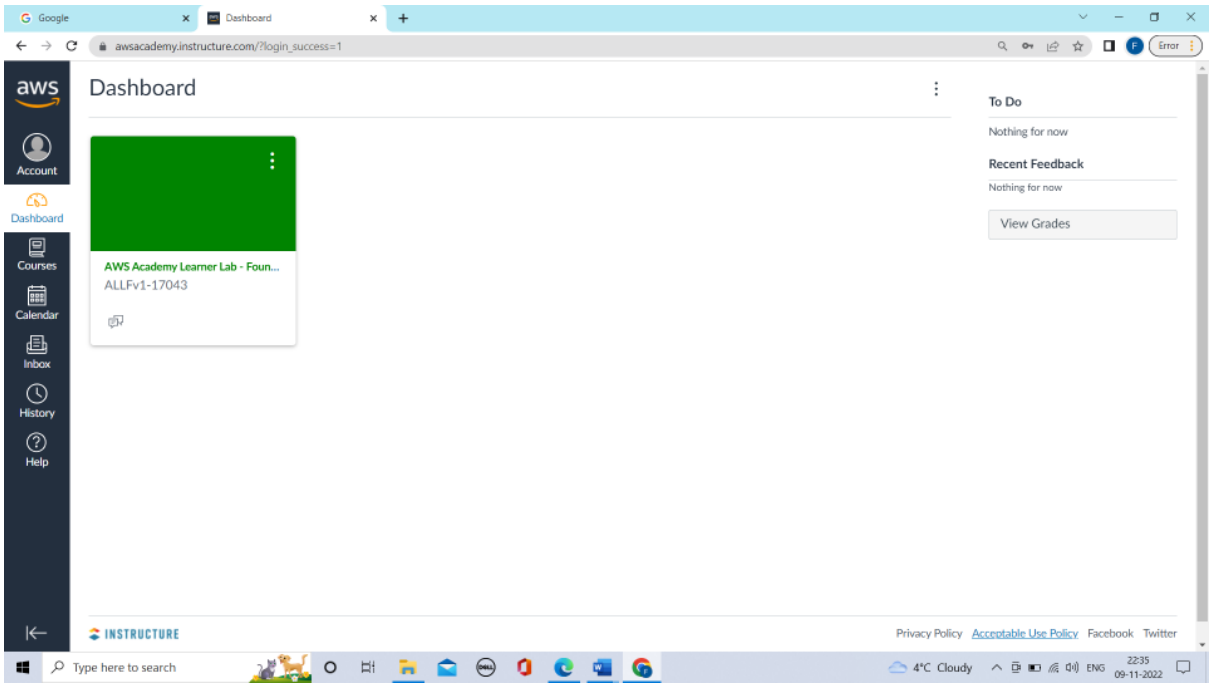

Fig 3: AWS Dashboard

- The first window you'll see open is the AWS Dashboard as shown in the fig 3.
- Click on the AWS Academy Learner Lab shown in the fig. 3 and the new window will be open which is shown in the fig 4.

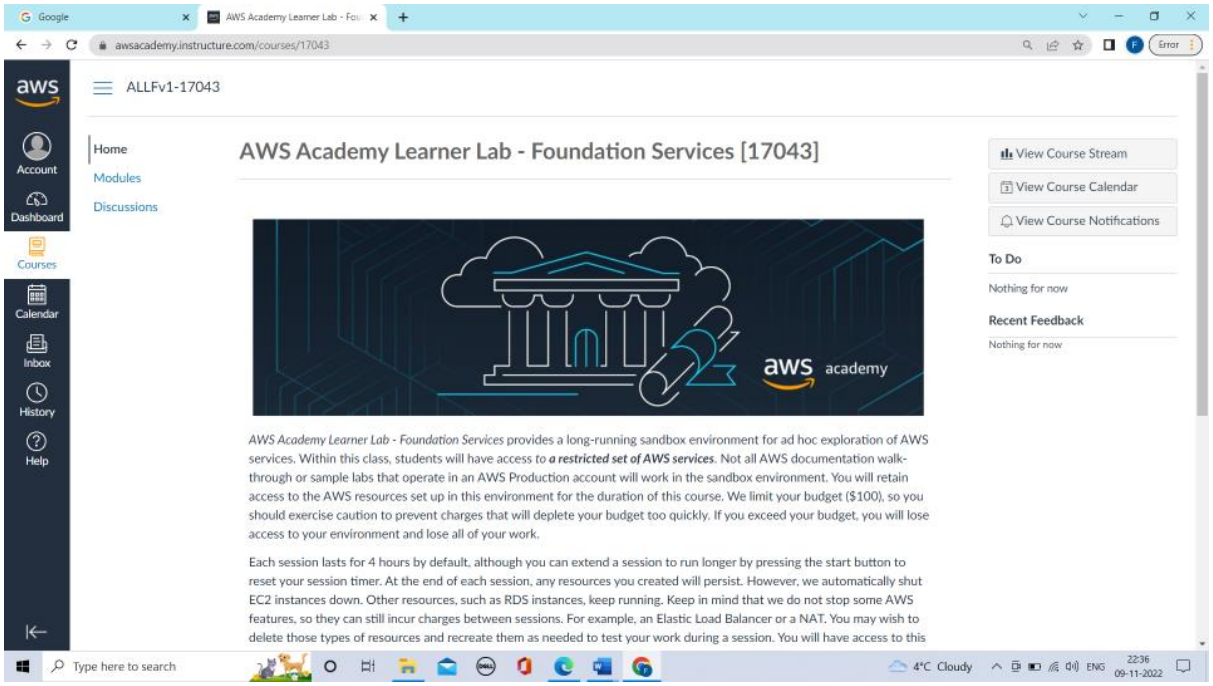

Fig 4: AWS home page

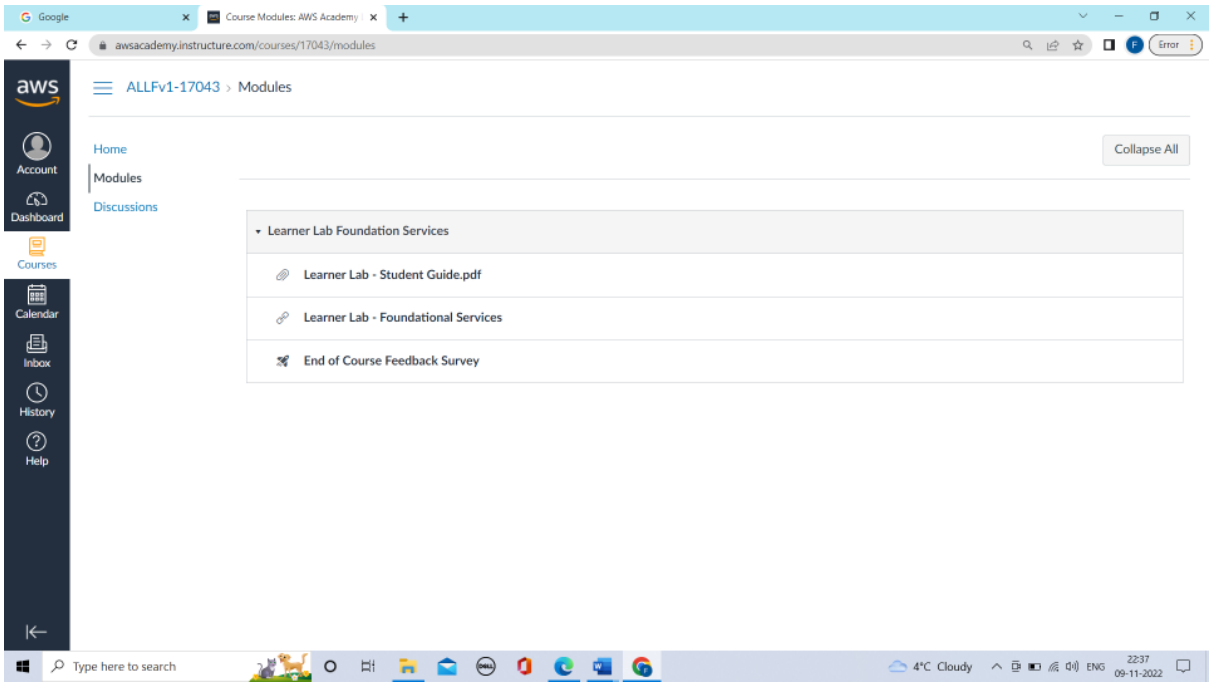

Fig 5: AWS Modules Page

- Go to modules as shown in the fig 4. After clicking on the modules, AWS modules page will open as shown in the fig 5.
- Go to "Learner Lab – Foundational Services" as shown in fig 5. After clicking on it, the start lab and end lab page will open as shown in the fig 6.
- Click on the Start Lab to start the lab (The red dot represents the lab is not started yet) as shown in the fig 6.

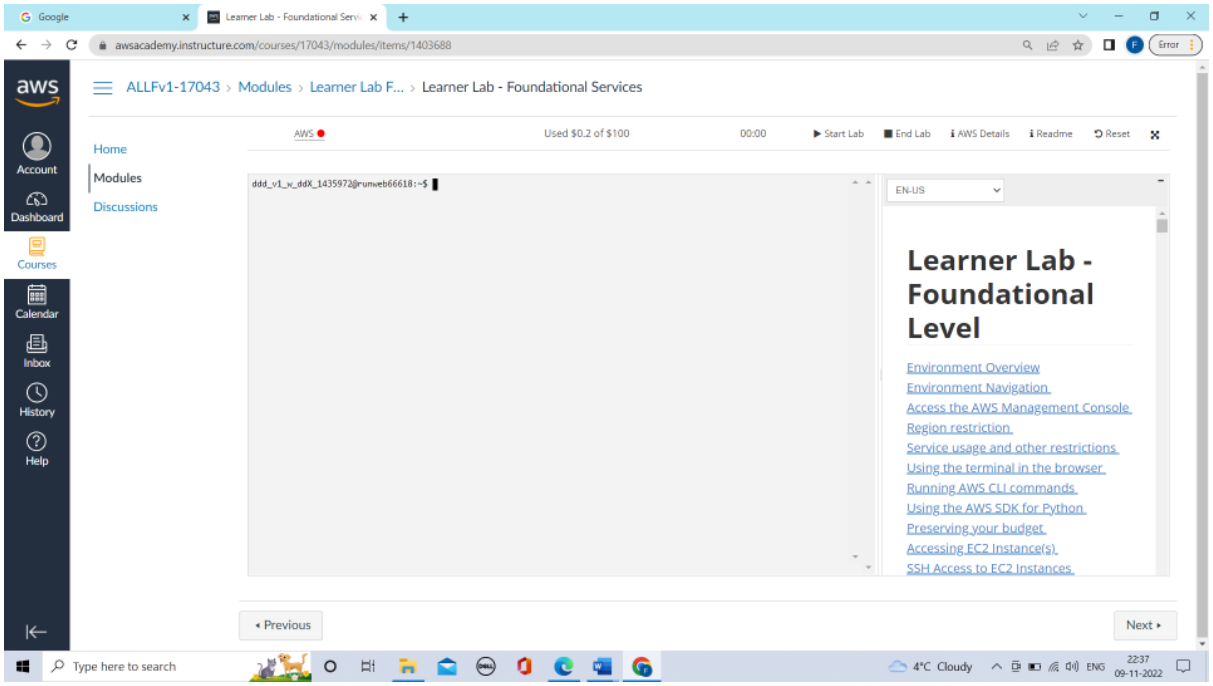

Fig 6: AWS End lab

- Lab has started because the green dot indicates the lab has started as shown in the fig 7.
- On clicking the AWS green icon which is shown in the fig. 7, it will redirect the user to the AWS Console Home as shown in the fig. 7.1. It contains all different services like Amazon Lex. DynamoDB, S3, Lambda and many more.

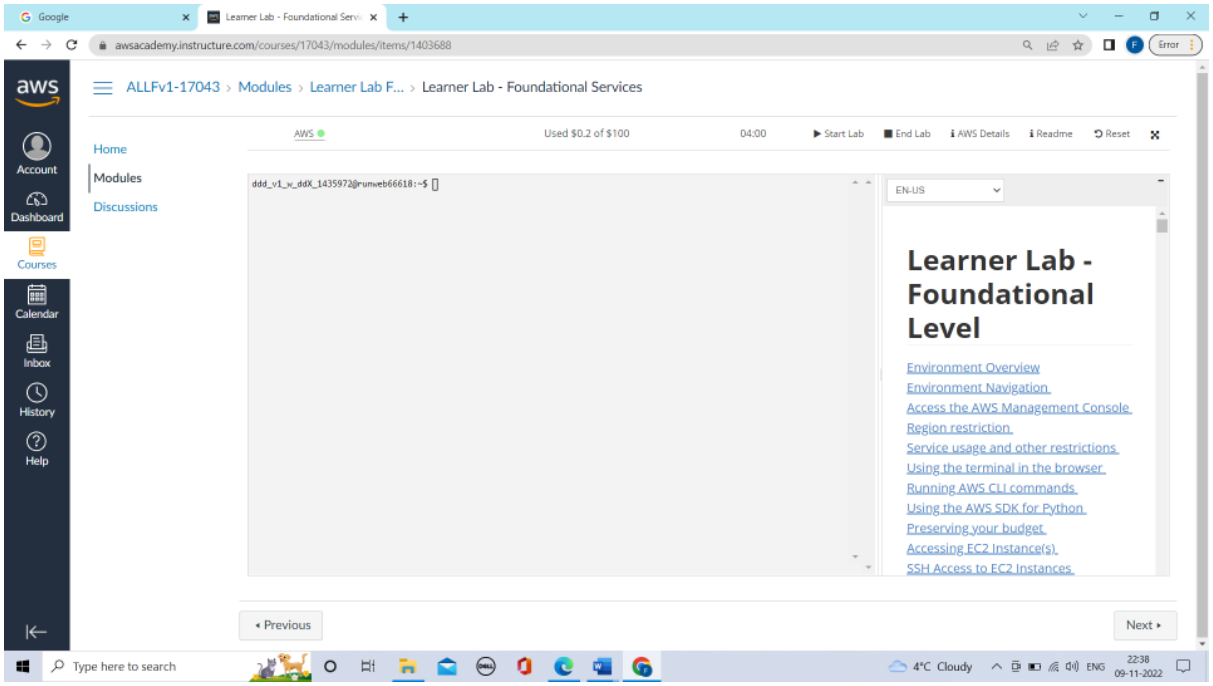

Fig 7: AWS start lab

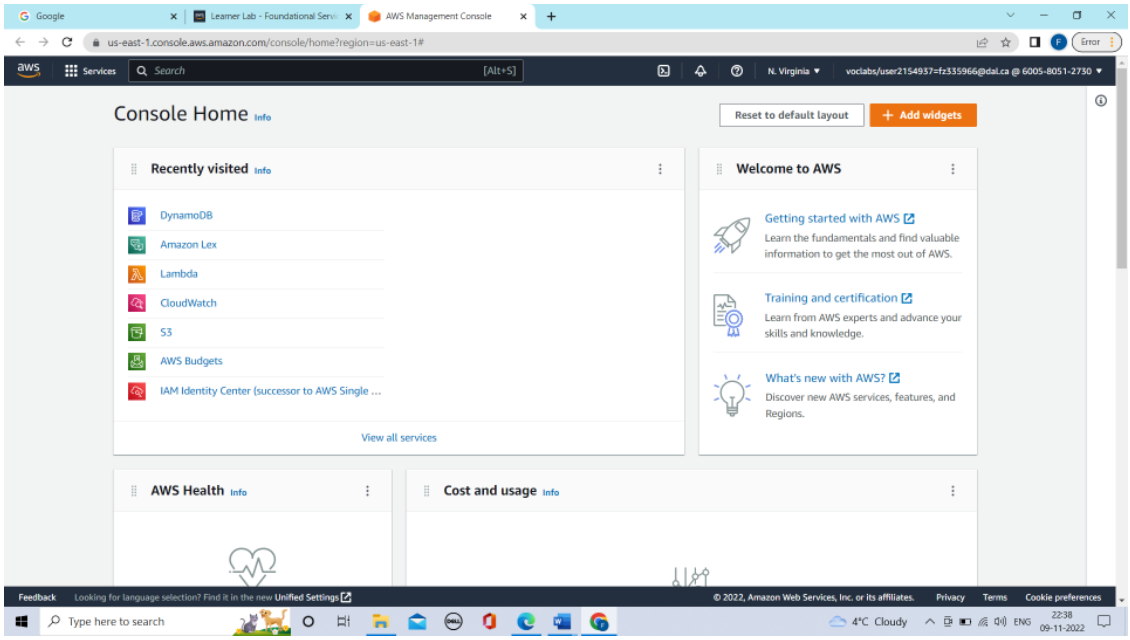

Fig 7.1: AWS Console Home

Step 2:

Create a custom chatbot [BotStdLookup] using AWS Lex, which can be used to verify student's identity (E.g., if a valid student is trying to access the system)

The steps I took is given below:

- Go to the Amazon Lex service to create a table as shown in the fig. 8. Click on "Create Table" as shown in the fig. 8 & 9.

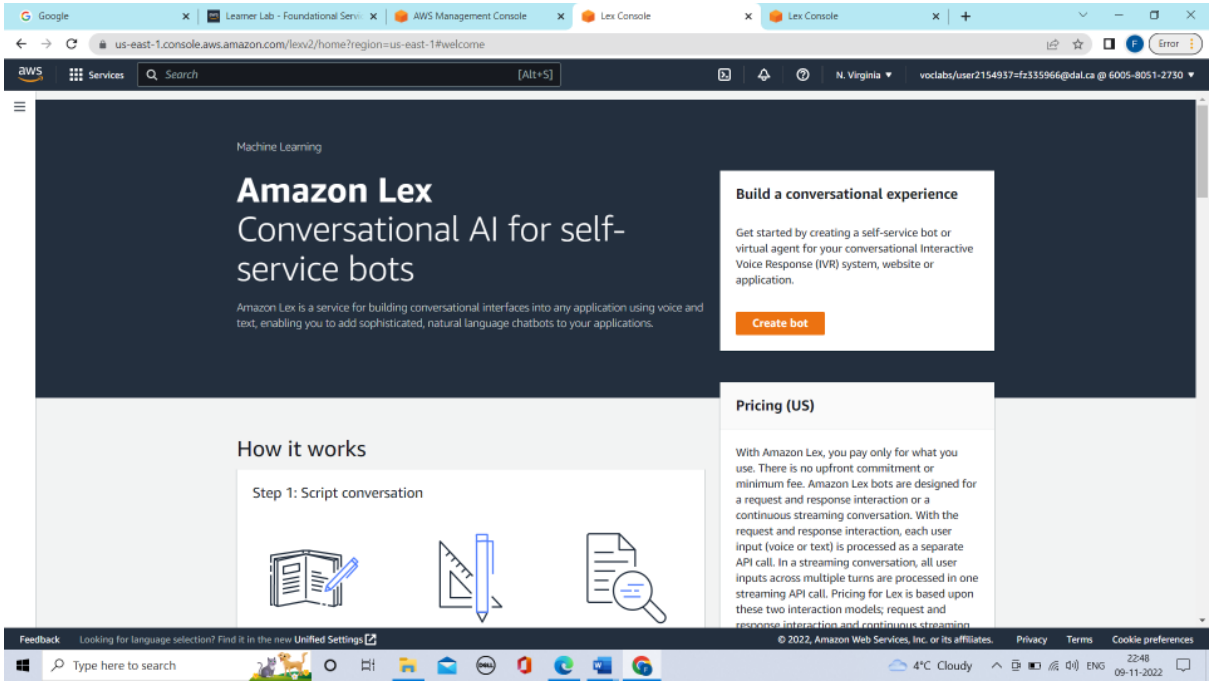

Fig 8: AWS Lex Home

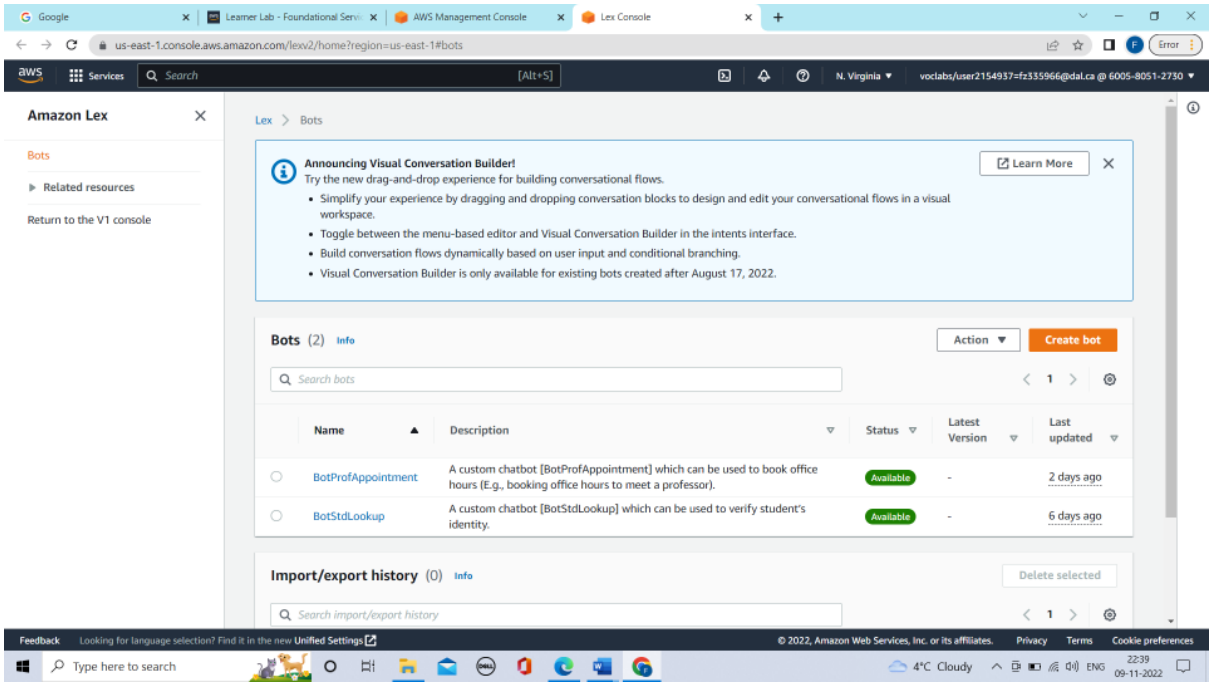

Fig 9: AWS Lex Home

- On clicking the "Create table", the configuration settings will open to provide details in order to create custom ChatBot as shown in the fig. 10, 11 & 12 and click on next page as shown in the fig 12.

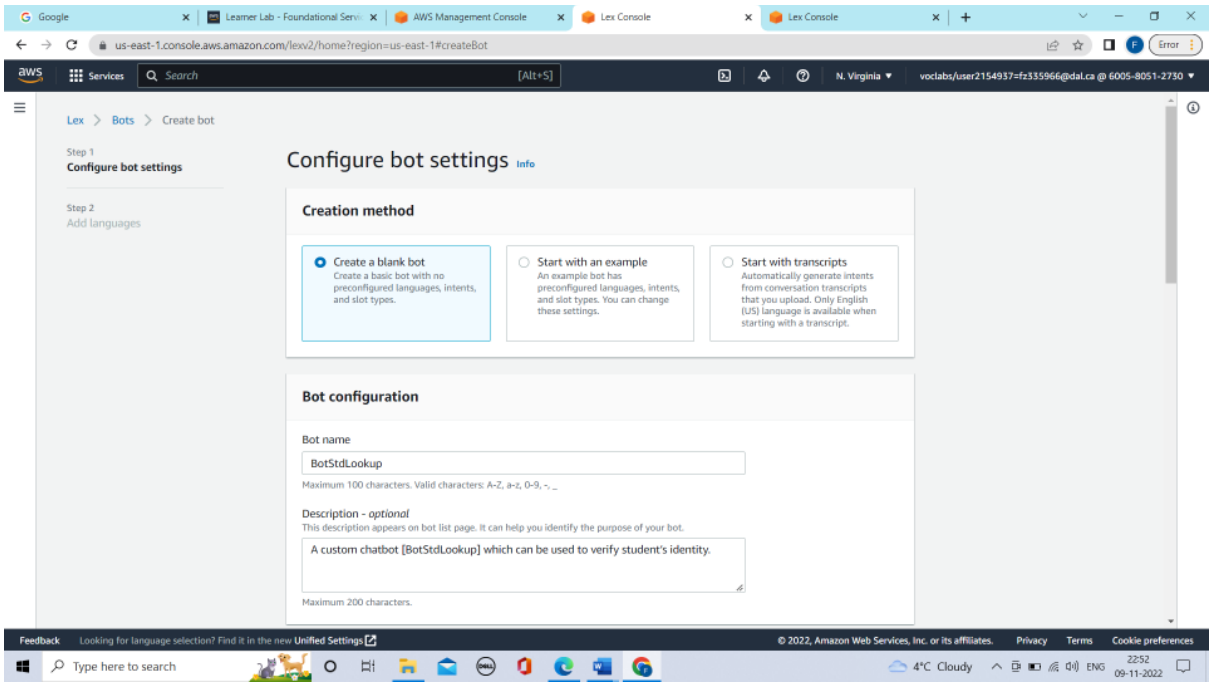

Fig 10: Configure bot settings page

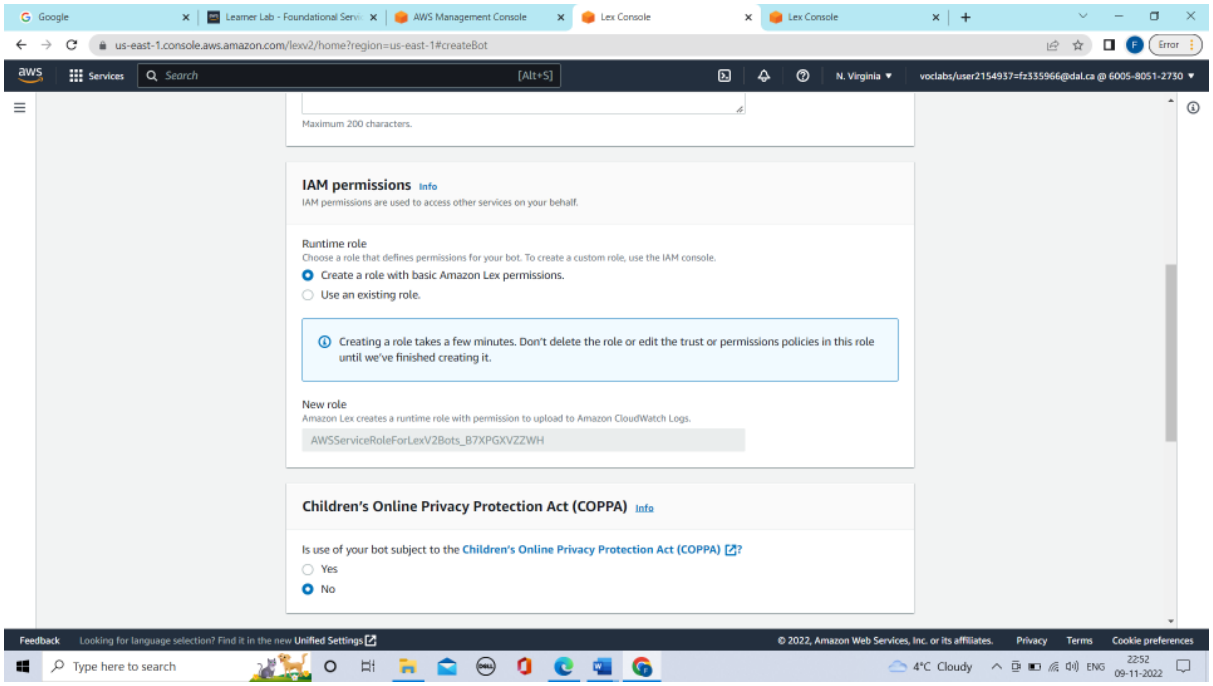

Fig 11: Configure bot settings page

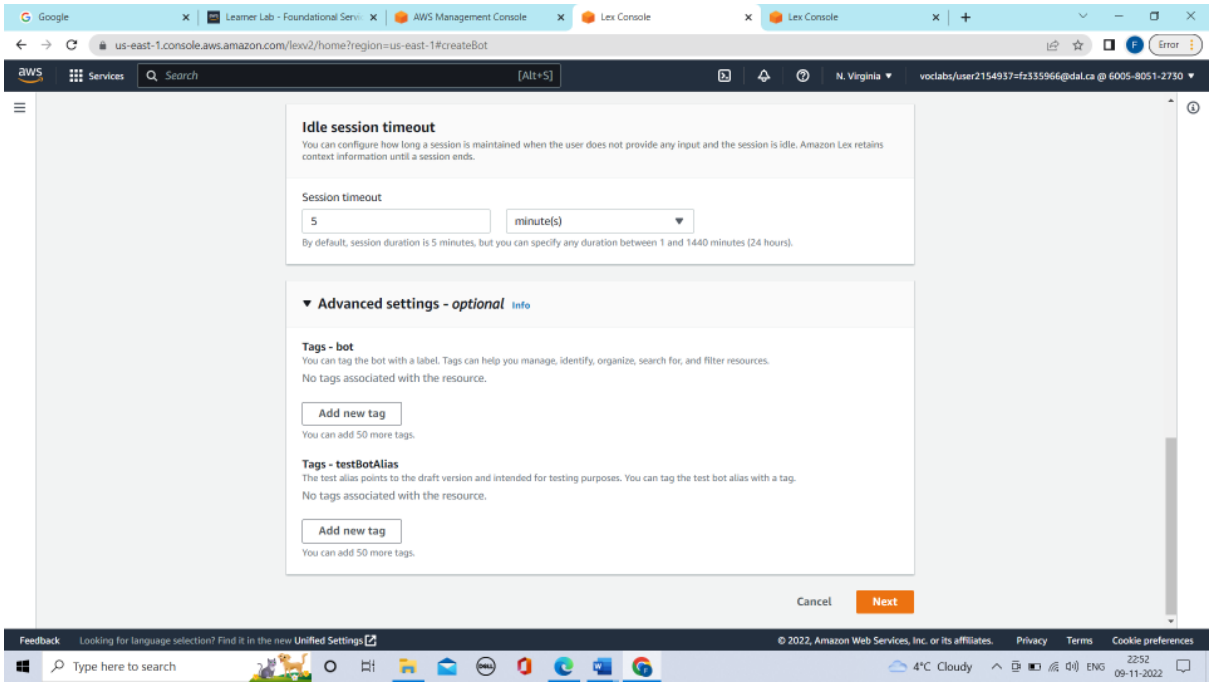

Fig 12: Configure bot settings page

- On the "Add language to bot" page, select "English" as the language and as the ChatBot is the text based application, select "This is only text based application" as shown in the fig. 13.
- After setting the language settings, the intents settings needs to made which is shown in the fig. 14.

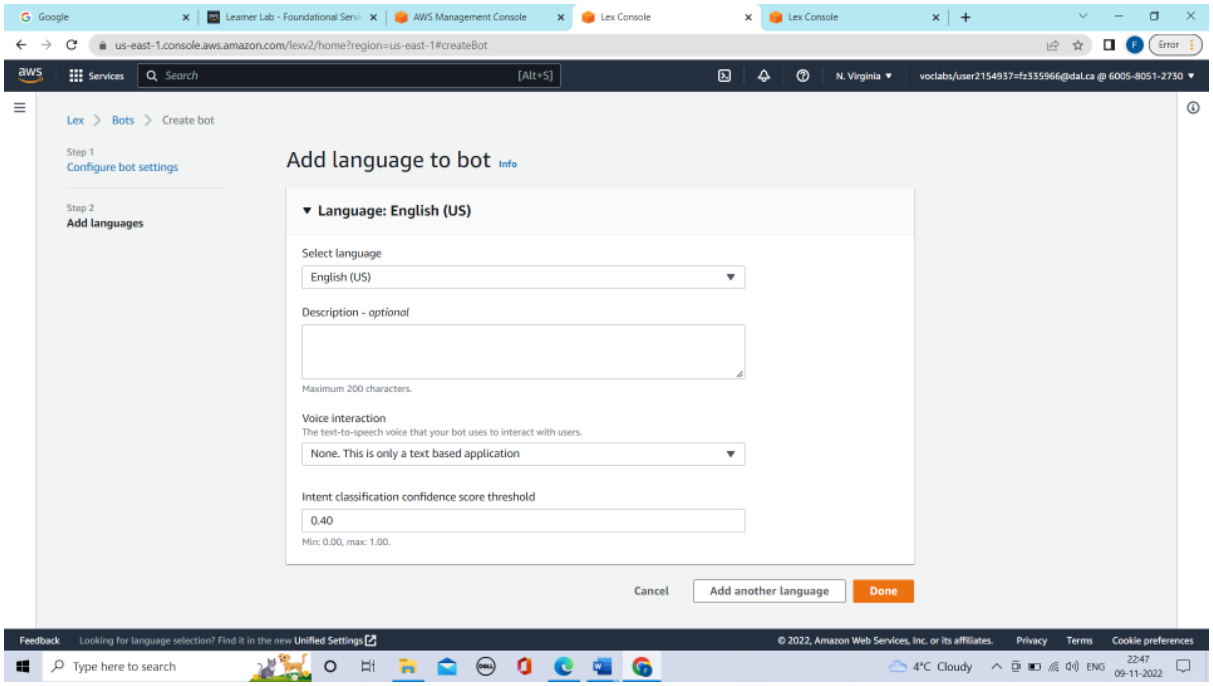

Fig 13: Add language to bot page

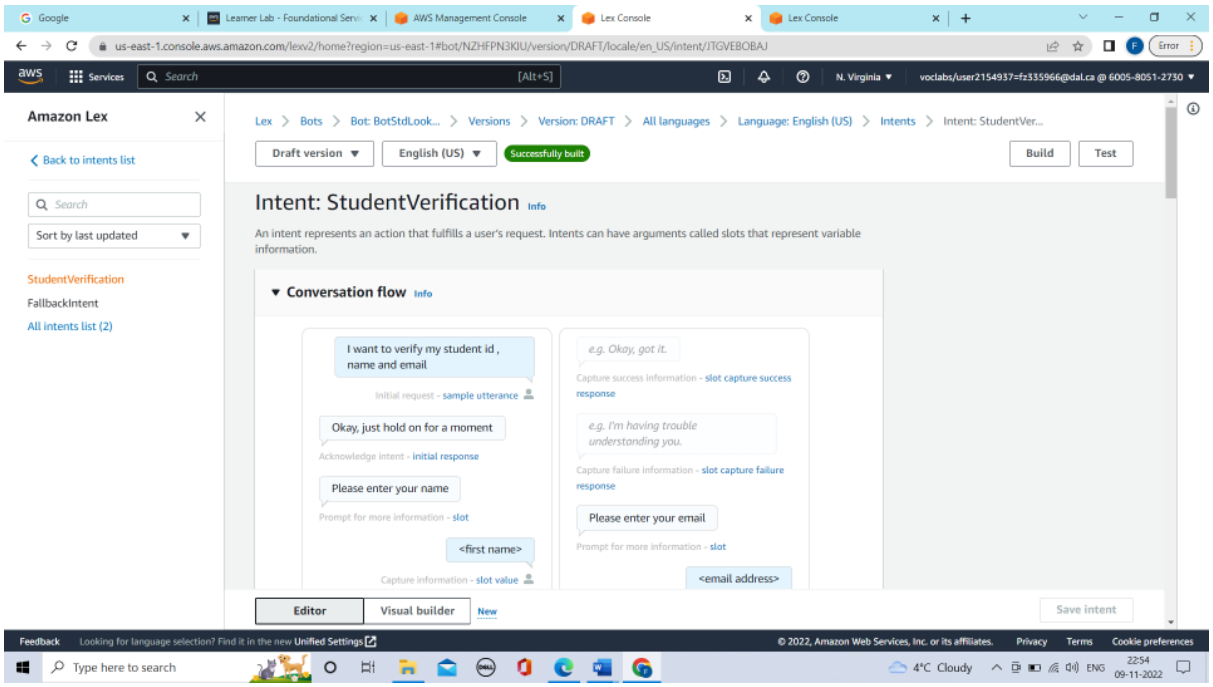

Fig 14: Add language to bot page

- The intent name provided is “BookAppointment” as shown in the fig. 14. The intent is nothing but the purpose of the Bot. The other intent details are filled as shown in the fig. 15
- User can make multiple uttrences like “hi”, “I’d like to verify my identity”, “verify id, name , email” and so on. The utterences is the text that user can start with or the replies from the users (Refer fig. 16 & 17).

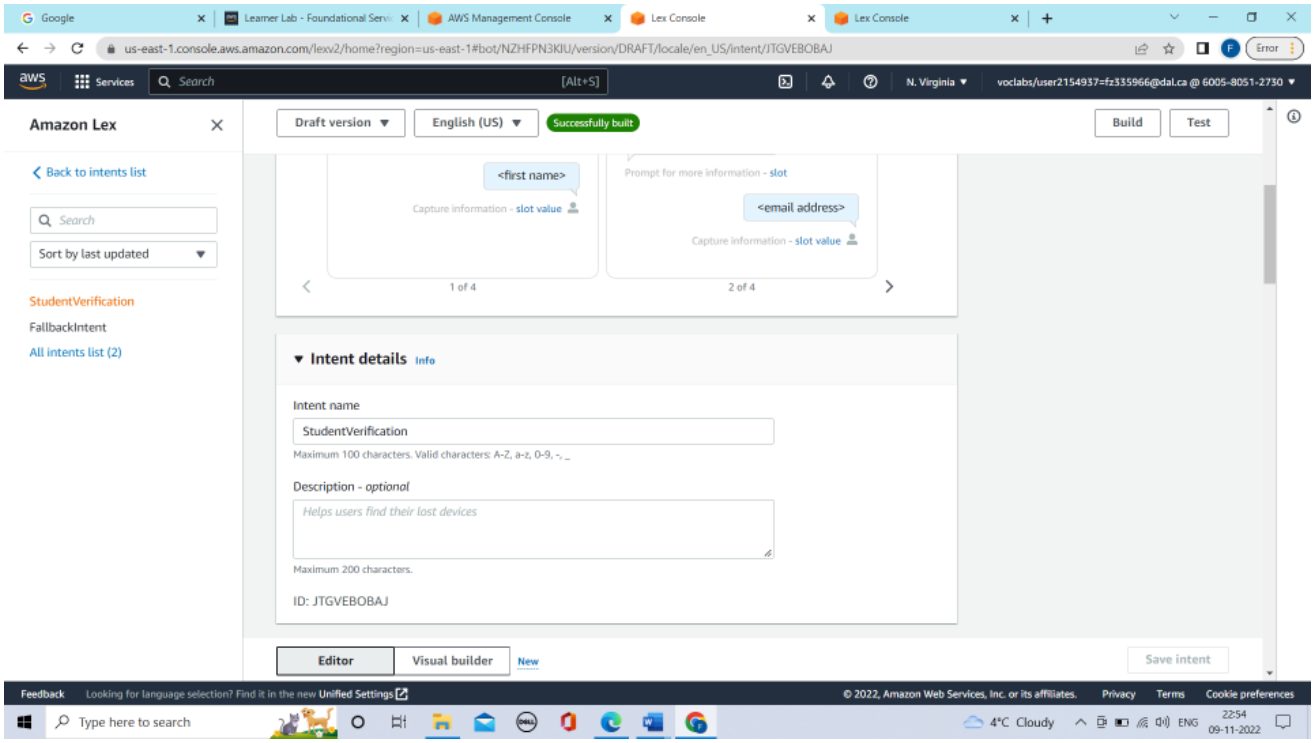

Fig 15: Add language to bot page

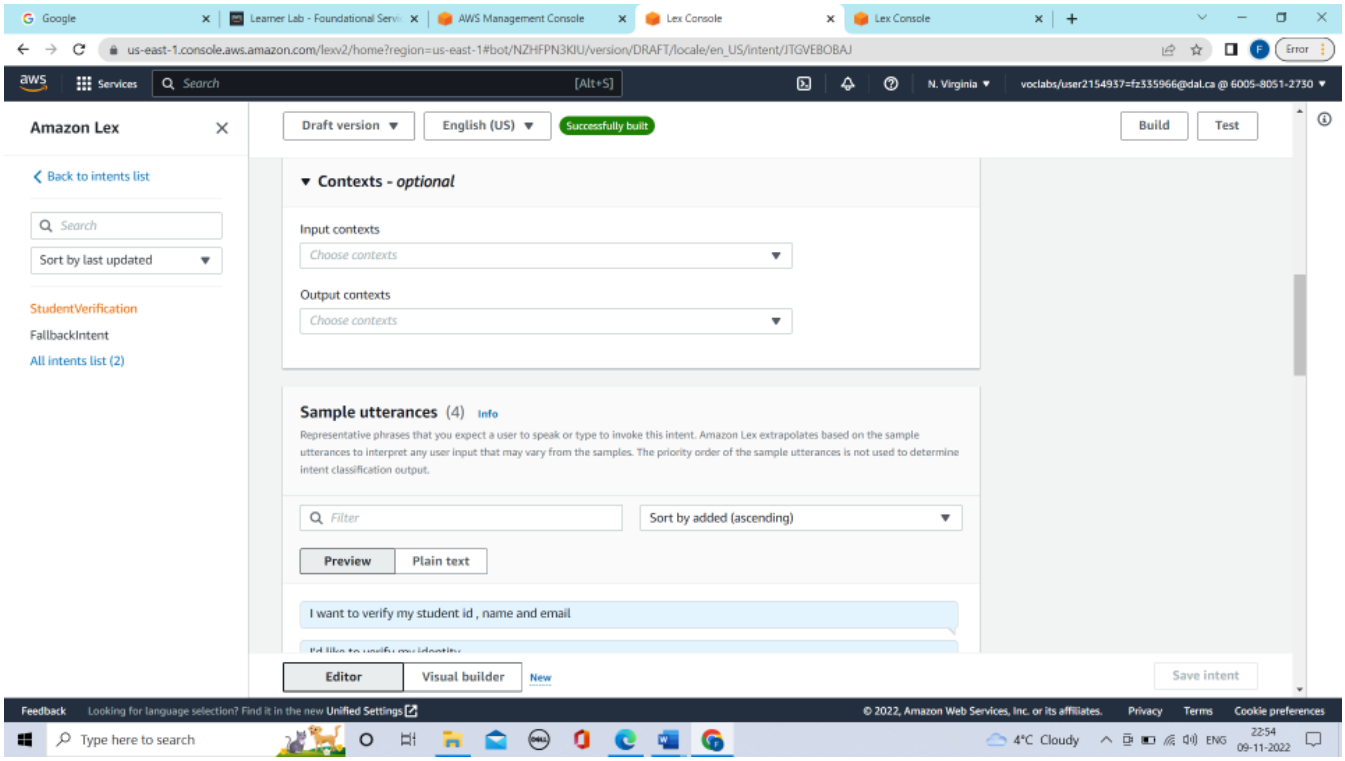

Fig 16: Contexts and sample utterance

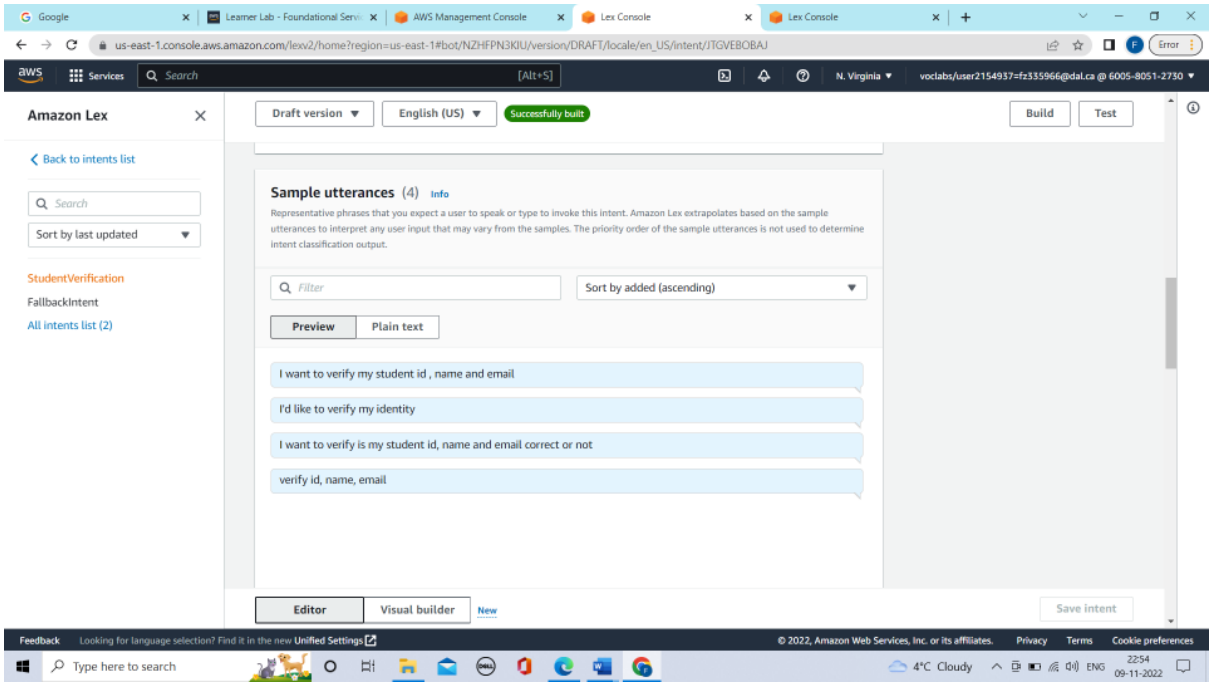

Fig 17: Sample utterance

- The user can set the initial response before starting the actual chat (Refer fig. 18). The slots are used to fulfill the bot intent. In order to verify the "Name" and "Email", the slots are formed to ask for users name and email for the verification (Refer fig. 19).

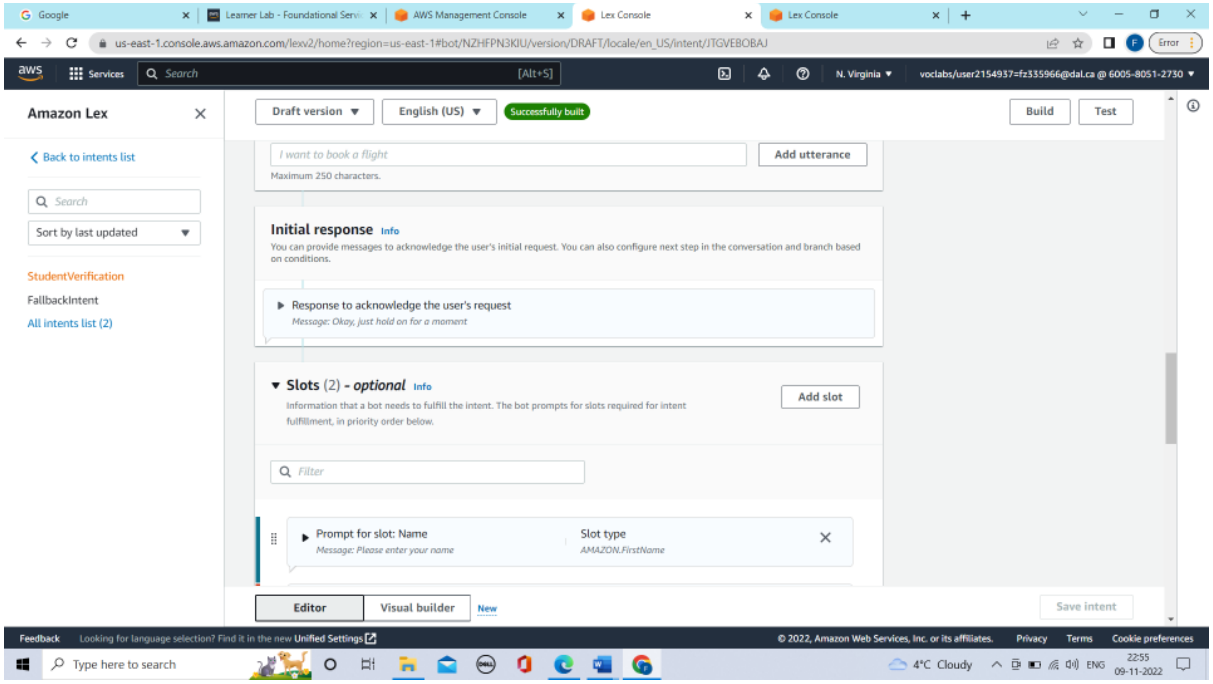

Fig 18: Initial Response

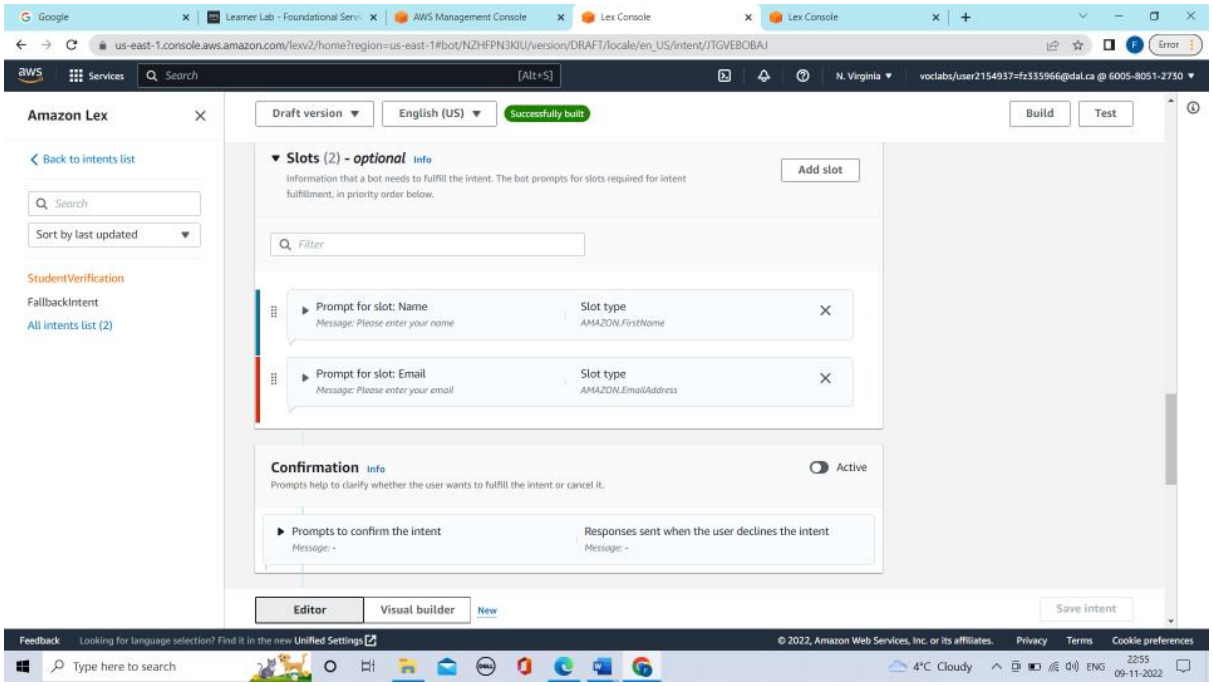

Fig 19: Slots

- Fulfillment given in fig. 20 allows users to know whether the intent is fulfilled or not.

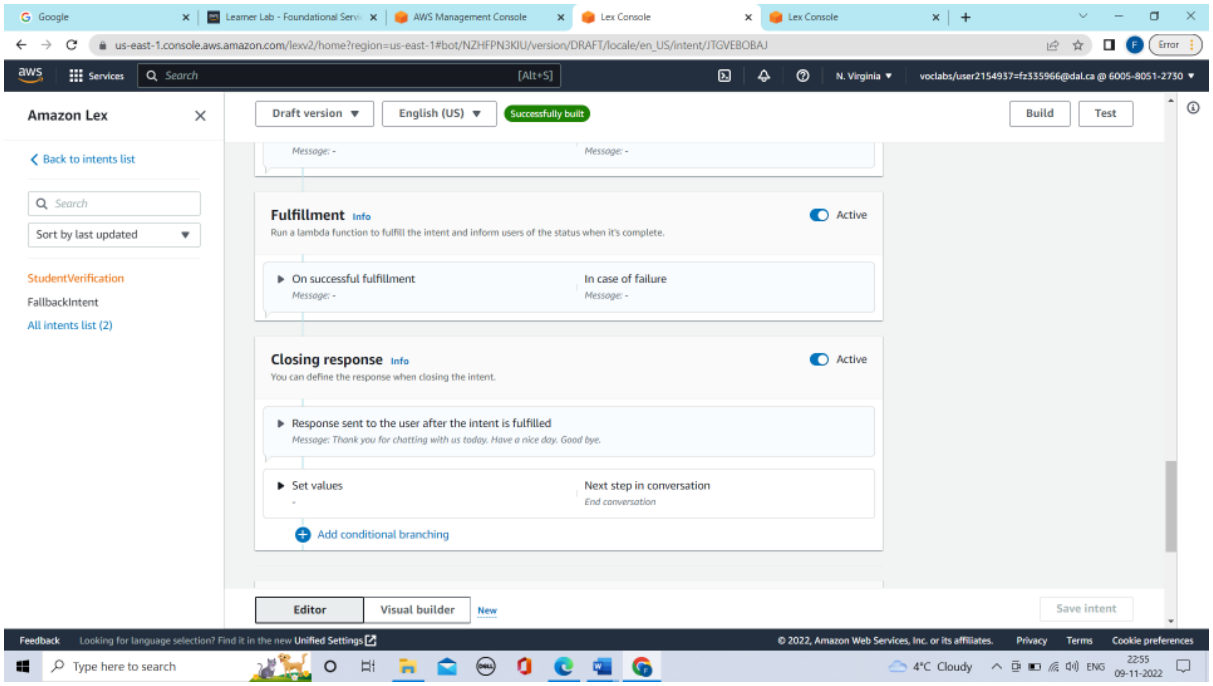

Fig 20: Fulfillment

- Closing response shown in the fig. 21 is nothing but the response sent to the user after the intent is fulfilled as shown in the fig. 20.

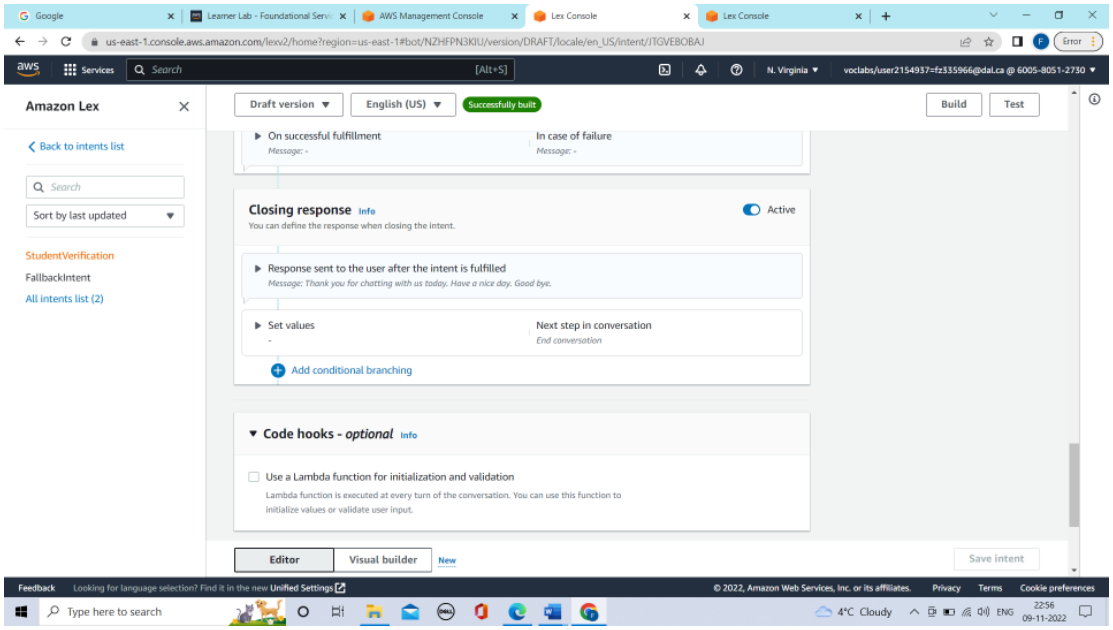

Fig 21: Closing Response

Step 3: Create a DynamoDB to store the data that needs to be verified.

Steps I followed is given below:

- Go to Amazon DynamoDB as shown in the fig. 22 to create a table. The DynamoDB is the database that helps to store the data as shown in the fig. 22.1. So when the students asks for the details verification like name, email and id, the ChatBot will look into the database which is DynamoDB to see if the data with the same name, email and id is present or not. If the details provided by the students matches with the details in the DynamoDB, that means the student details are verified otherwise its not.

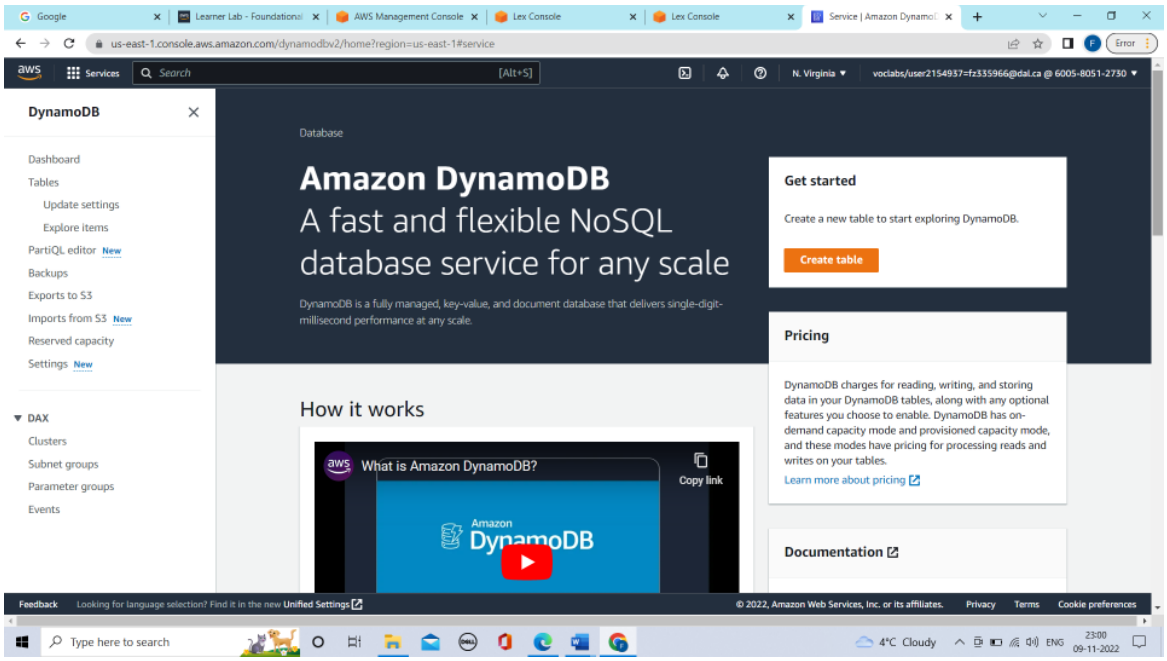

Fig 22: Amazon DynamoDB home page

- One can also create a table from DynamoDB dashboard as shown in the fig 22.

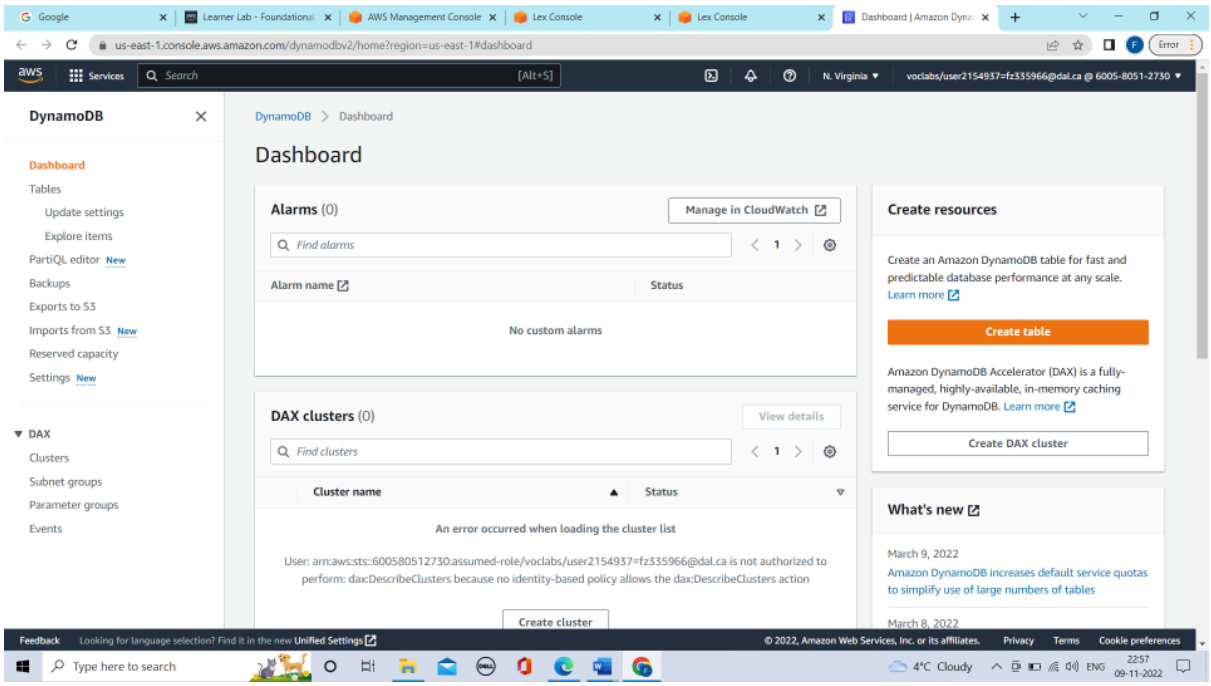

Fig 22.1: Amazon DynamoDB home page

- Fill the required details like table name and partial key as shown in the fig. 23, 24 and 25. The partial key is nothing but the primary key of the table. So if suppose user entered wrong name and email, the user will first be prompted as “incorrect email” rather than “incorrect name”. Because the details provided are first checked with the primary key.

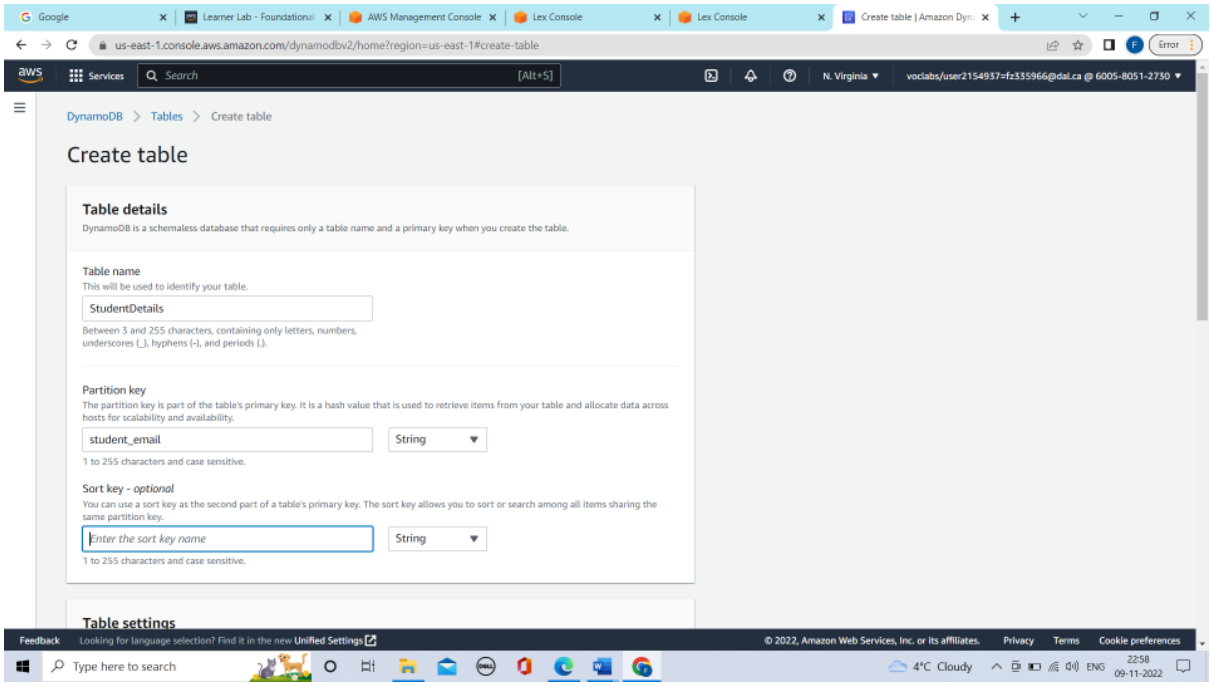

Fig 23: Create table page

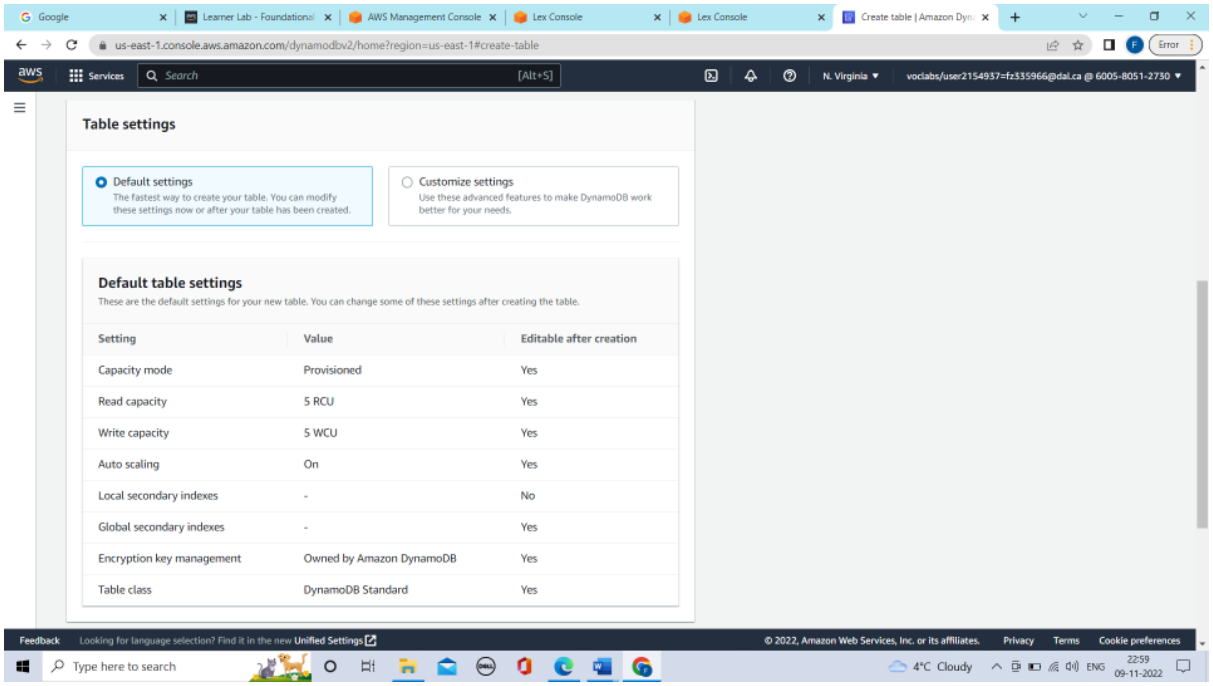

Fig 24: Create table page

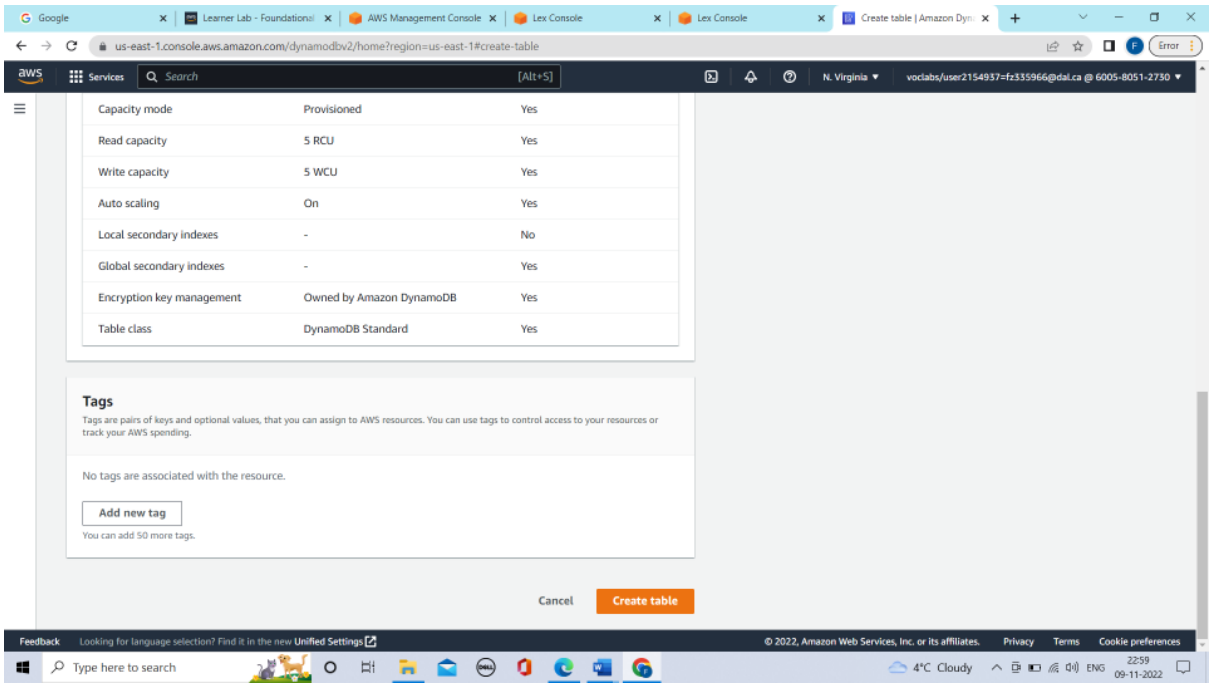

Fig 25: Create table page

- As we can see in the fig. 26, the table "StudentDetails" is formed.
- Go to the tables and we'll see "explore items" where we can add the attributes like name, email and id (Refer fig. 29).
- Some other table information is also provided in the pages (Refer fig. 27 & 28).

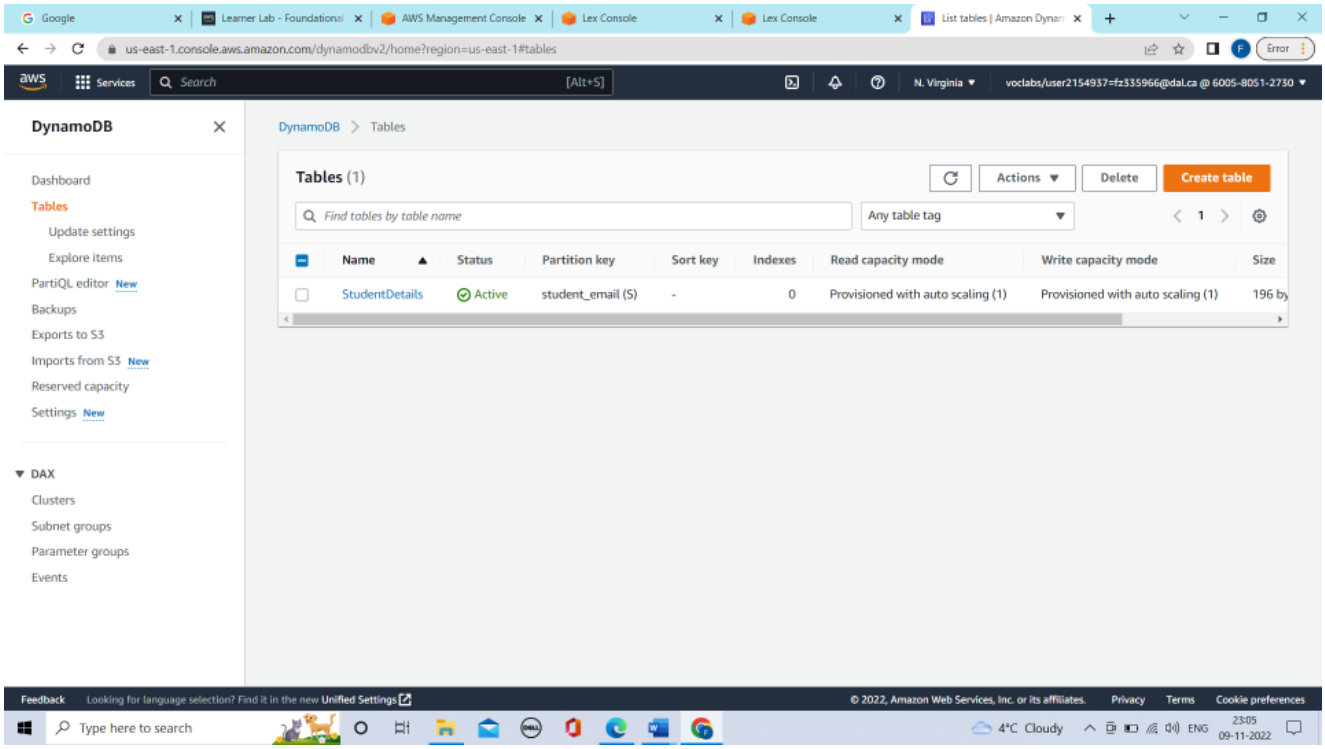

Fig 26: Tables page

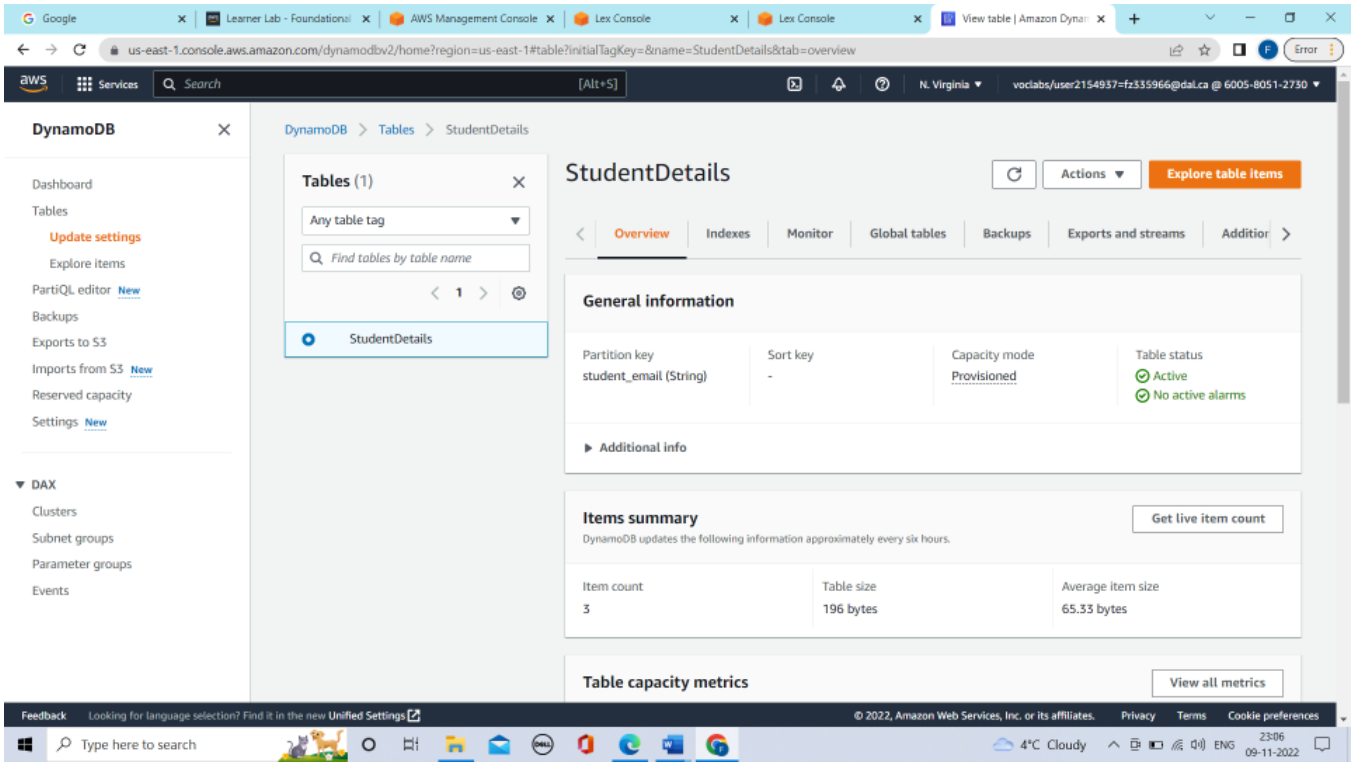

Fig 27: Explore items page

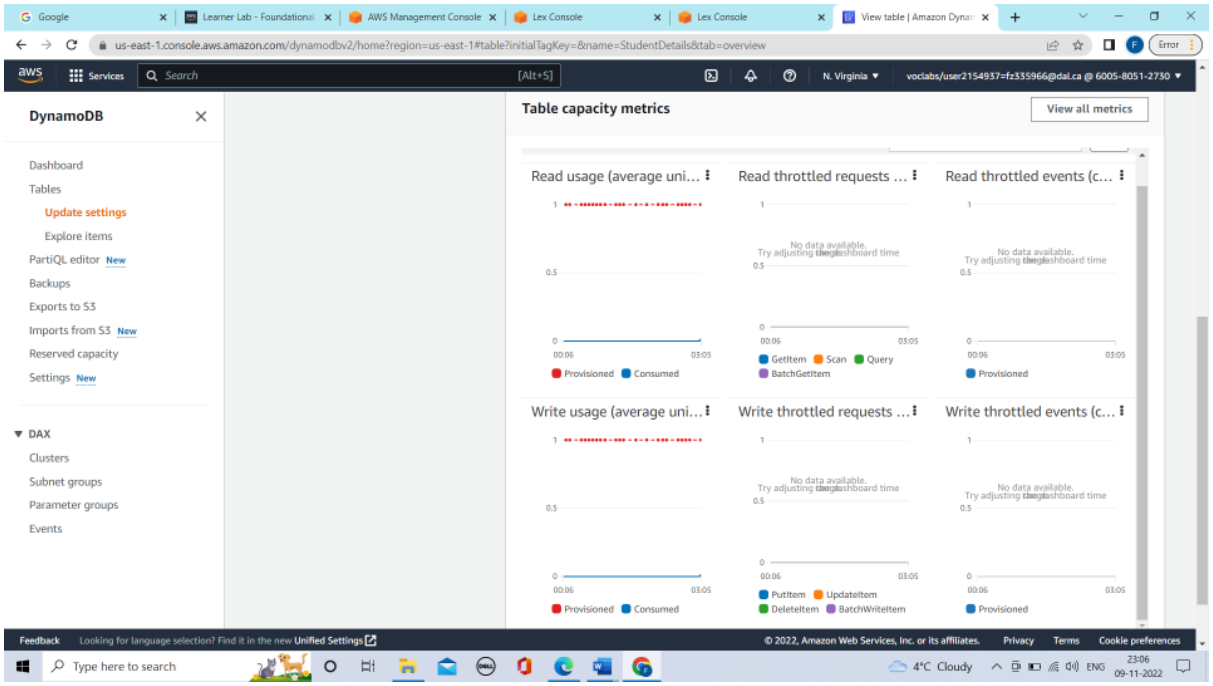

Fig 28: Explore items page

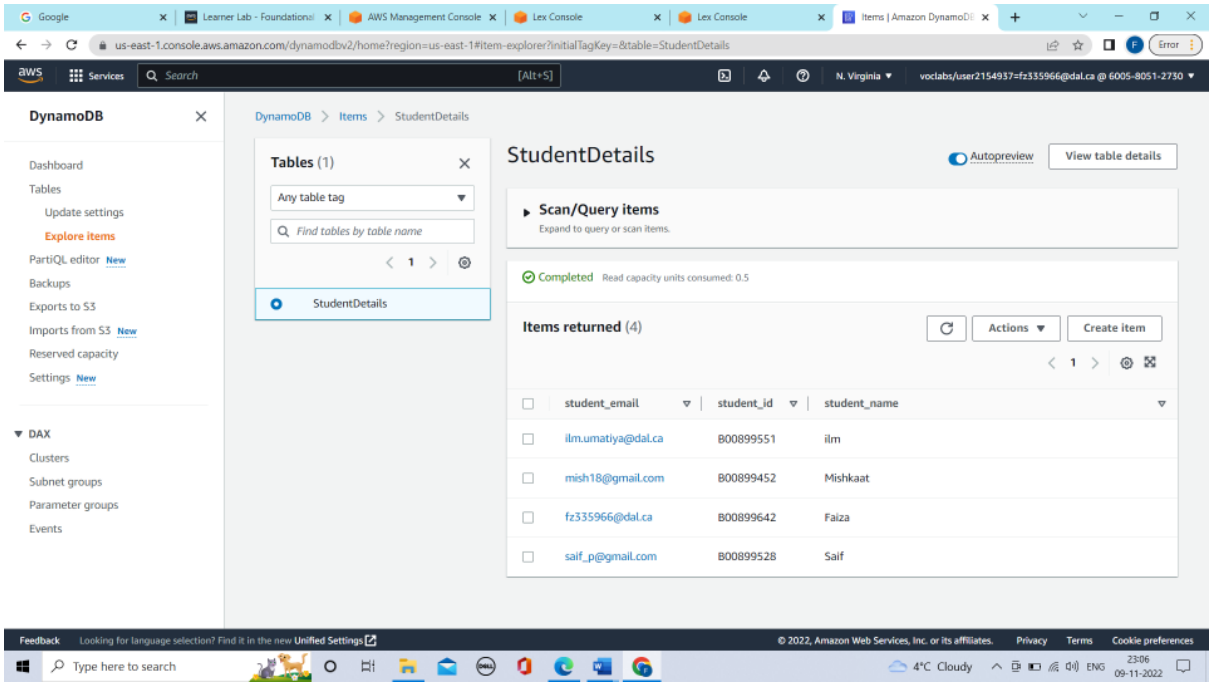

Fig 29: Explore items page

- Click on “Create item” as shown in the fig. 29.
- Refer fig. 30, to see how the attributes are added in an item. Attributes I considered for the student verification is “student_email”, “student_id” and “student_name”. Where student_email is the partial key.
- After adding the attributes click on “Create item” as shown in the fig. 30.

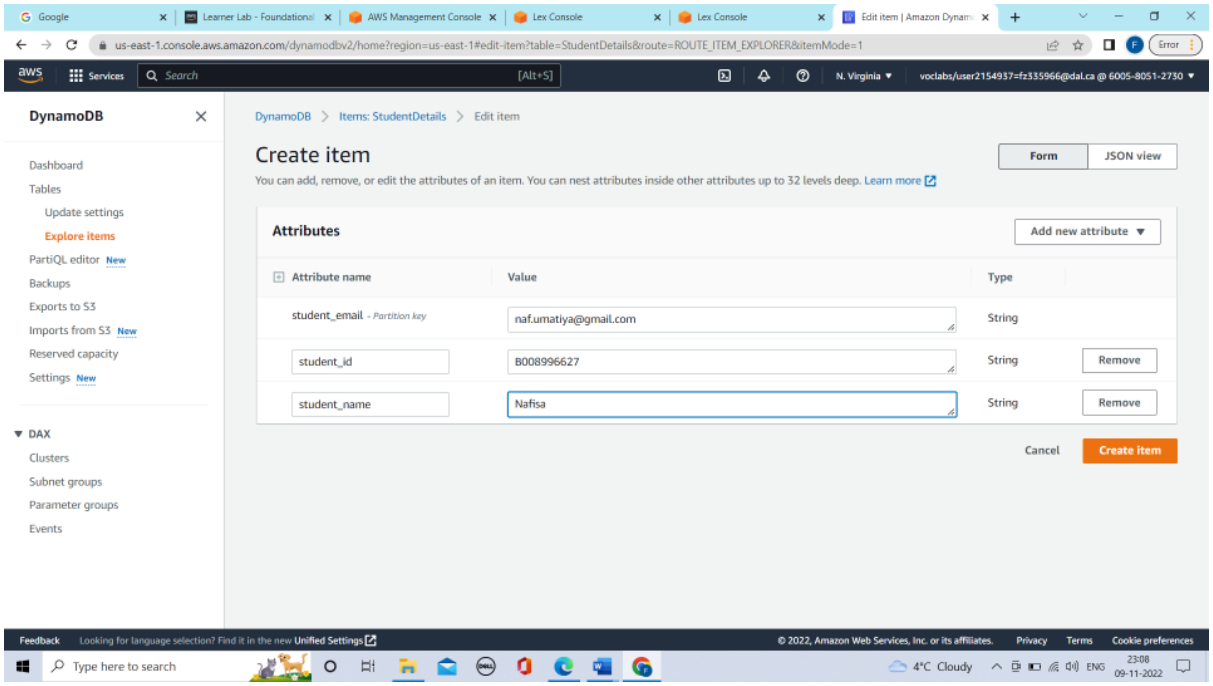

Fig 30: Create items page

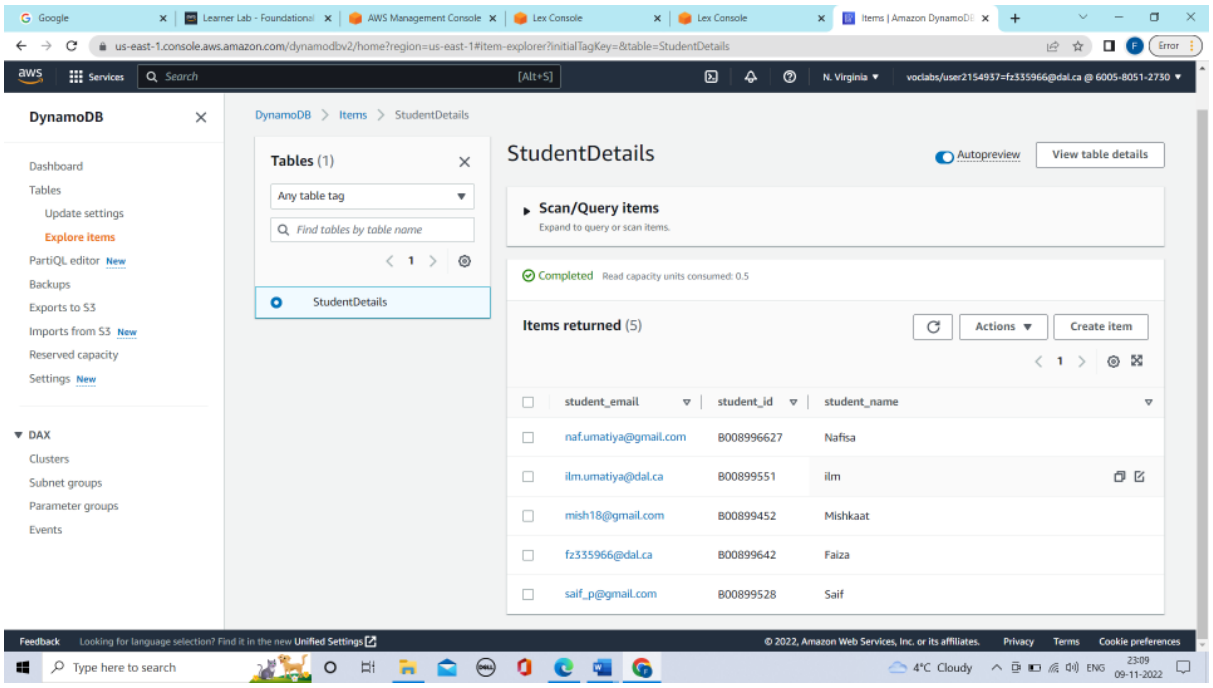

Fig 31: Create items page

Step 4: Create a Lambda function which communicates with the DynamoDB to retrieve records to perform user's name and email verification.

Steps I followed is given below:

- Go to AWS Lambda page and create the function by clicking on the option "Create function" as shown in the fig. 32.

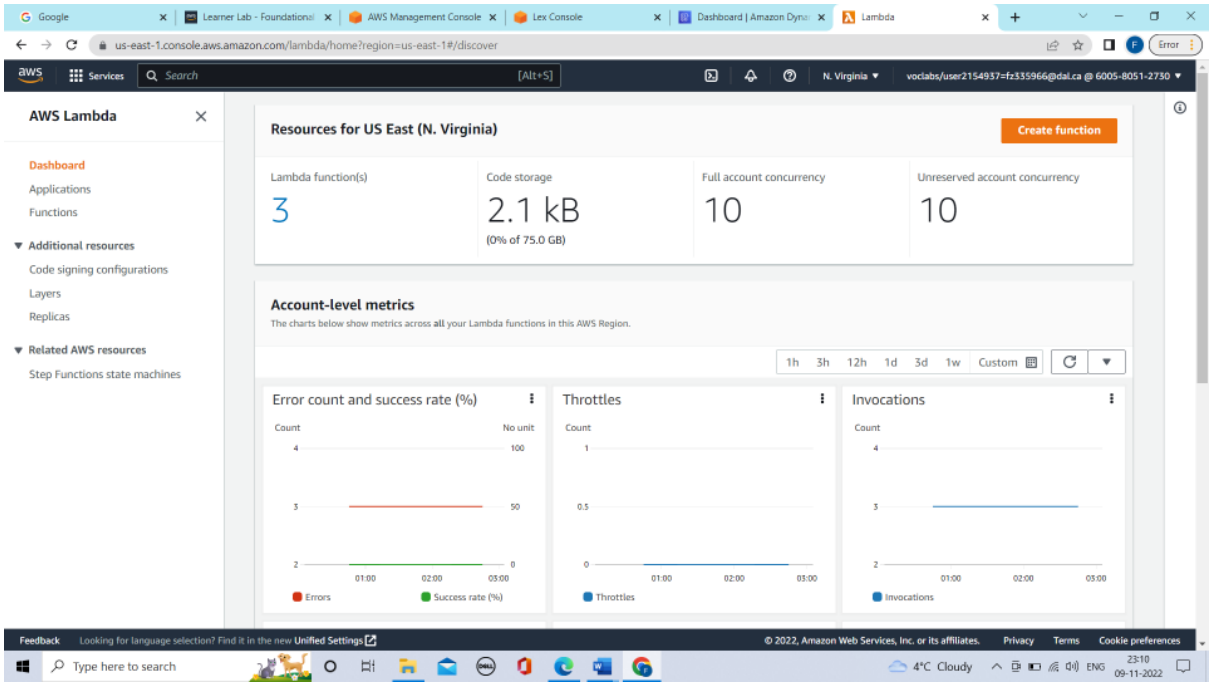

Fig 32: Create items page

- On clicking the "Create function", the user is directed to the page where we need to add details to create lambda function (Refer fig. 33).
- I named the function name as "verifyStudentIdentity" as shown in the fig, 33.
- Fill the other details like providing permissions as shown in the fig. 34.

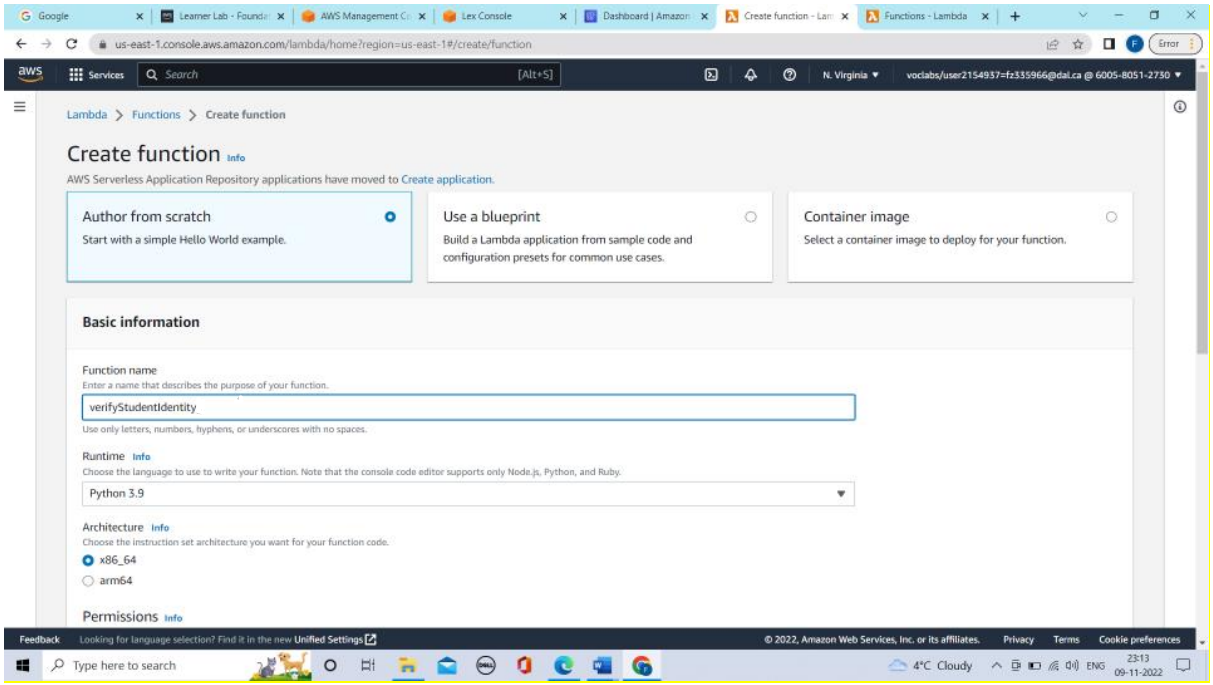

Fig 33: Create functions page

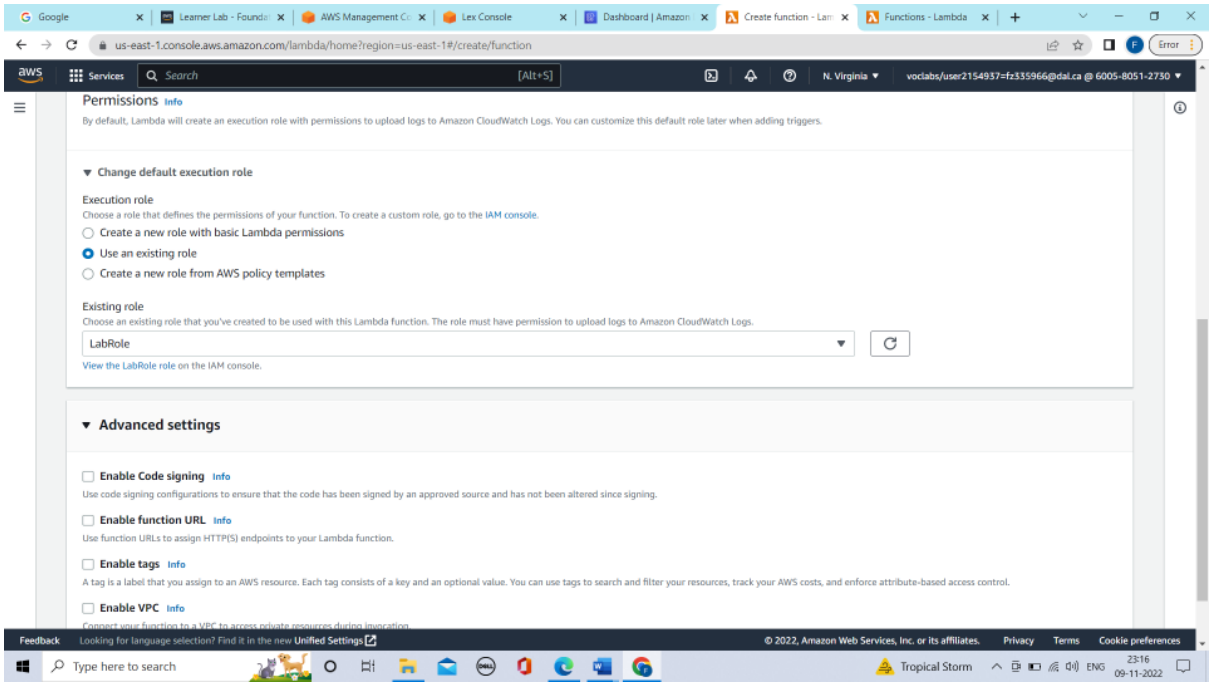

Fig 34: Create functions page

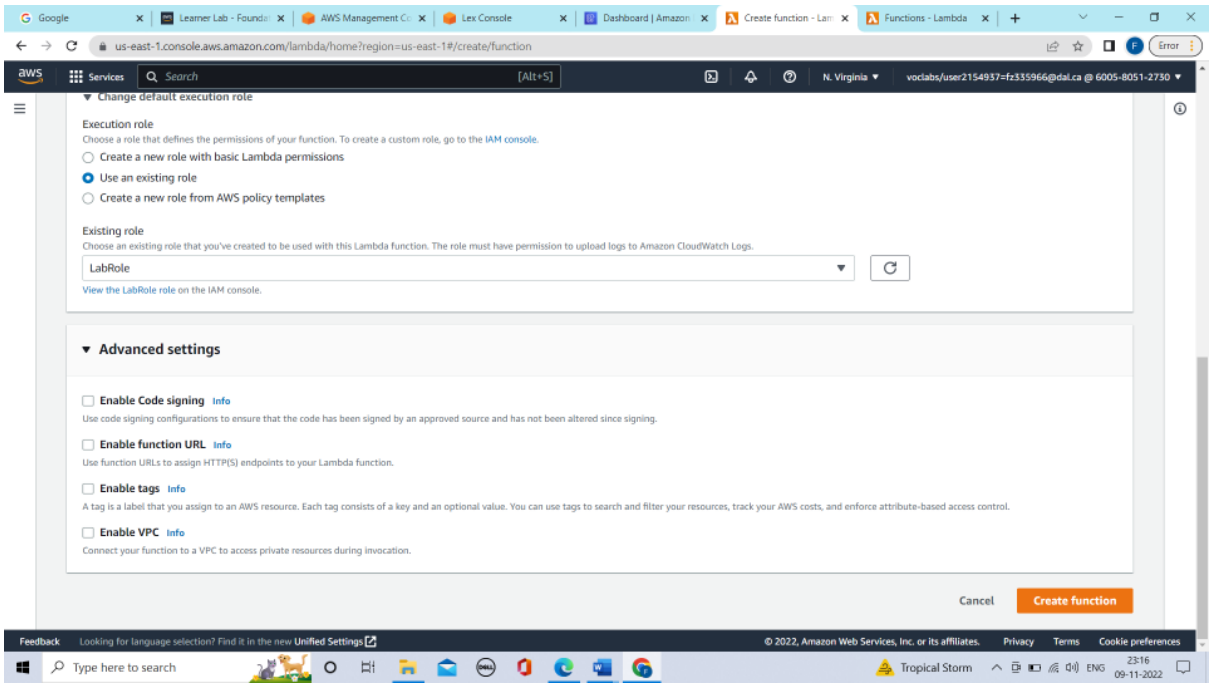

Fig 35: Create functions page

- Click on "Create function" as shown in the fig. 35 after filling the necessary details.
- "verifyStudentIdentity" function has been made as shown in the fig. 36.
- Fig. 37, 38, 39 & 40 represents the lambda function (verifyStudentDetails) code in python.
- In the fig 37, 38 & 39, I added slots, intent and dynamoDB names to get the data accurately. The condition is used to match the details provided by the user and the details present in the DynamoDB.

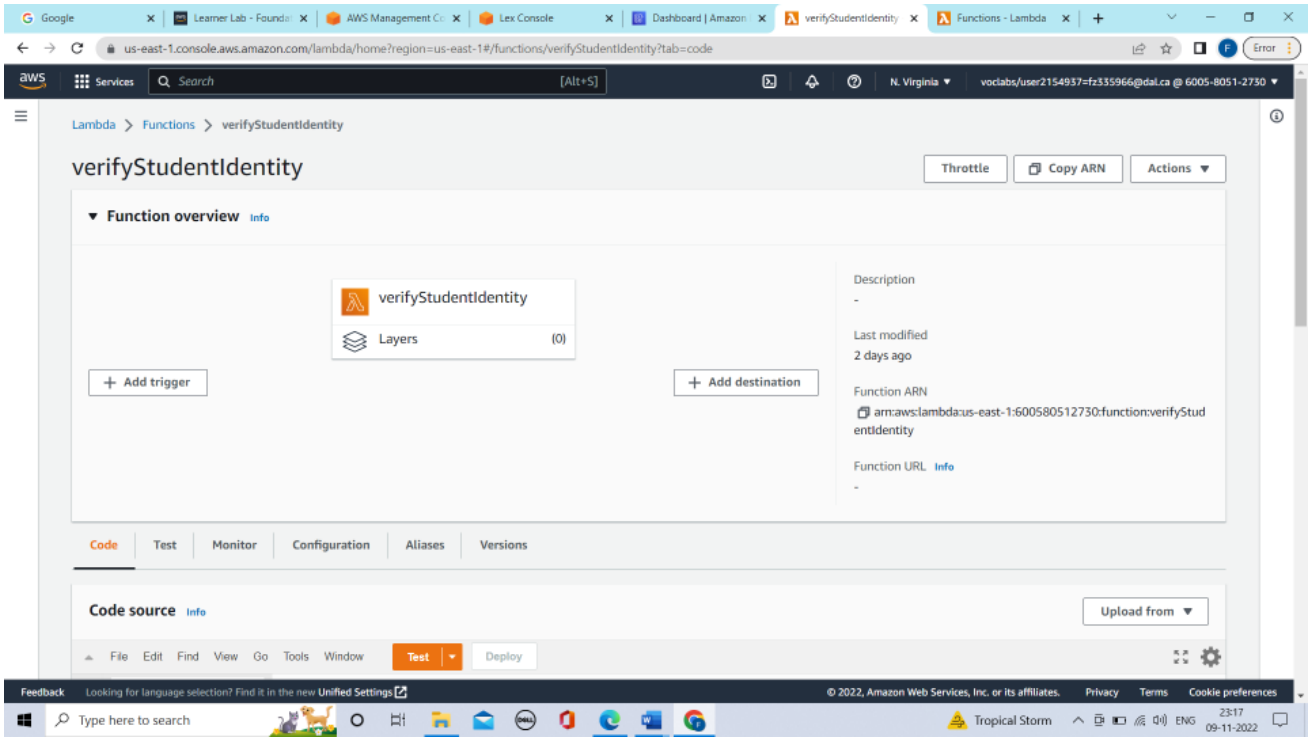

Fig 36: functions page

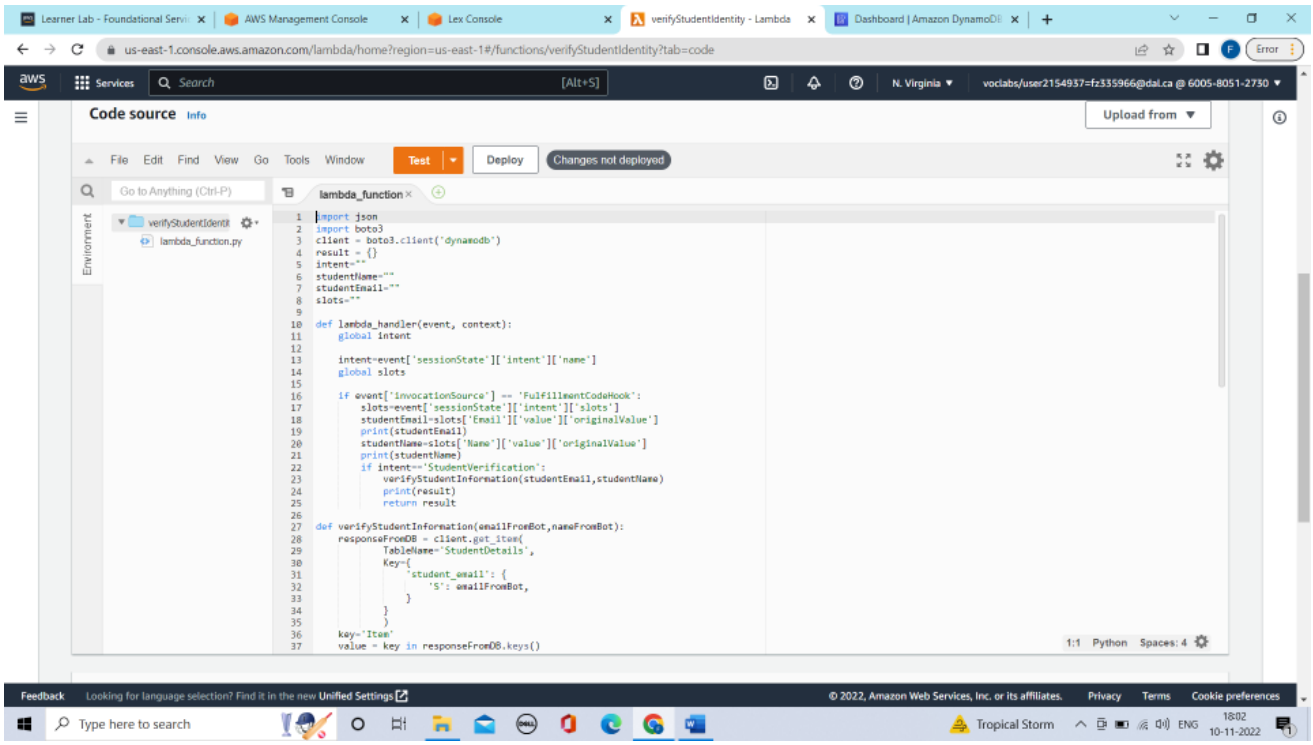

Fig 37: Lambda function code

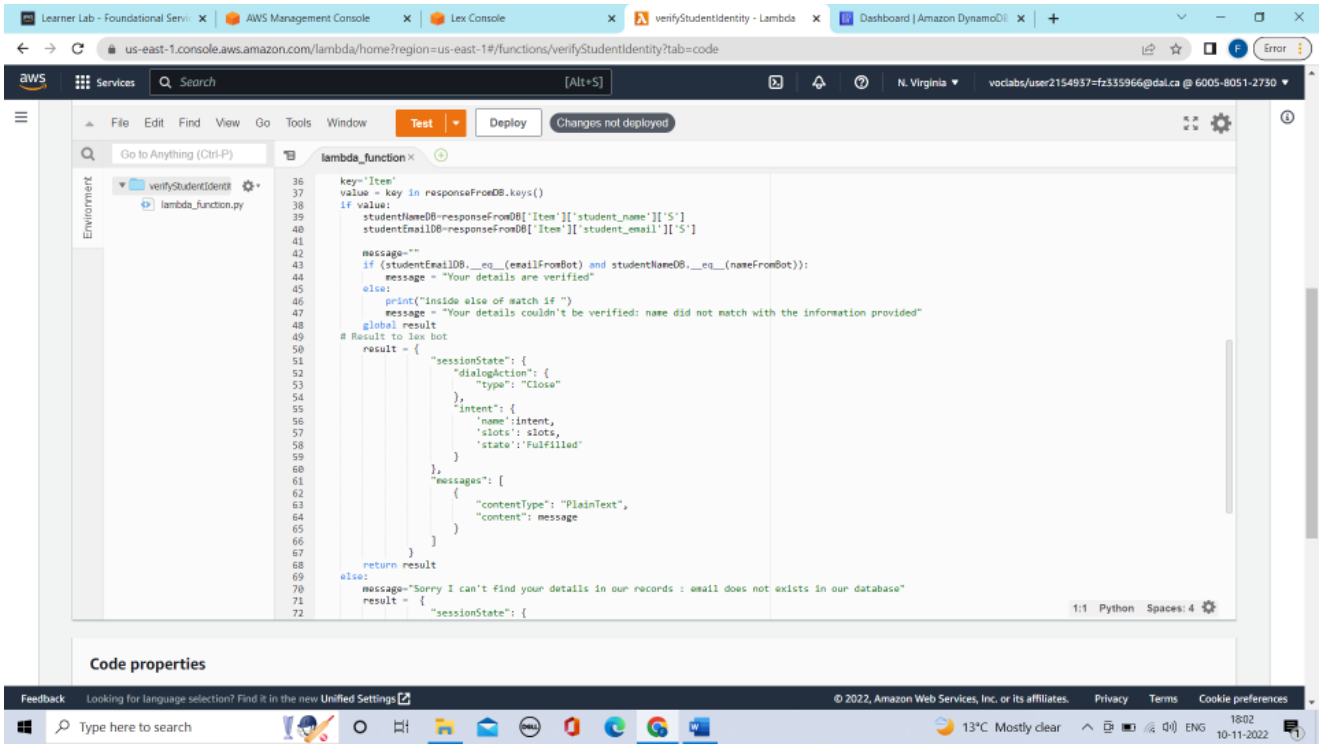

Fig 38: Lambda function code

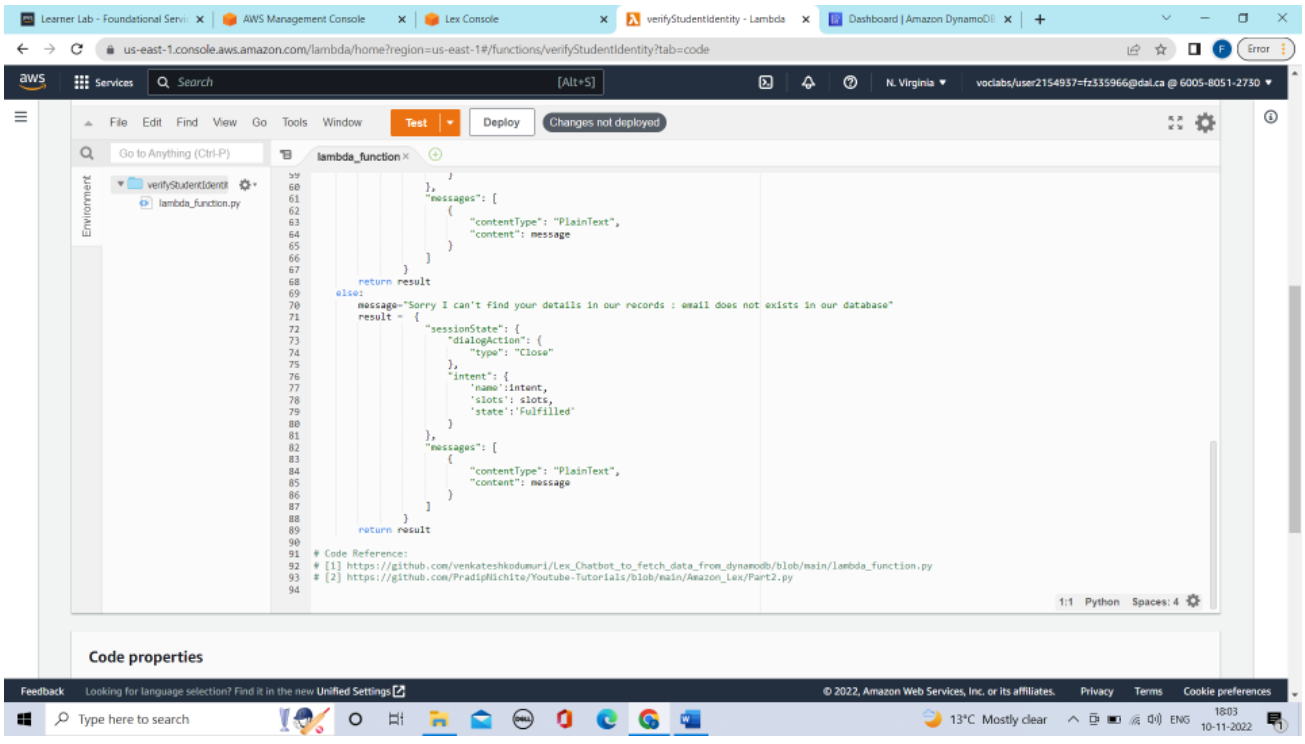

Fig 39: Lambda function code

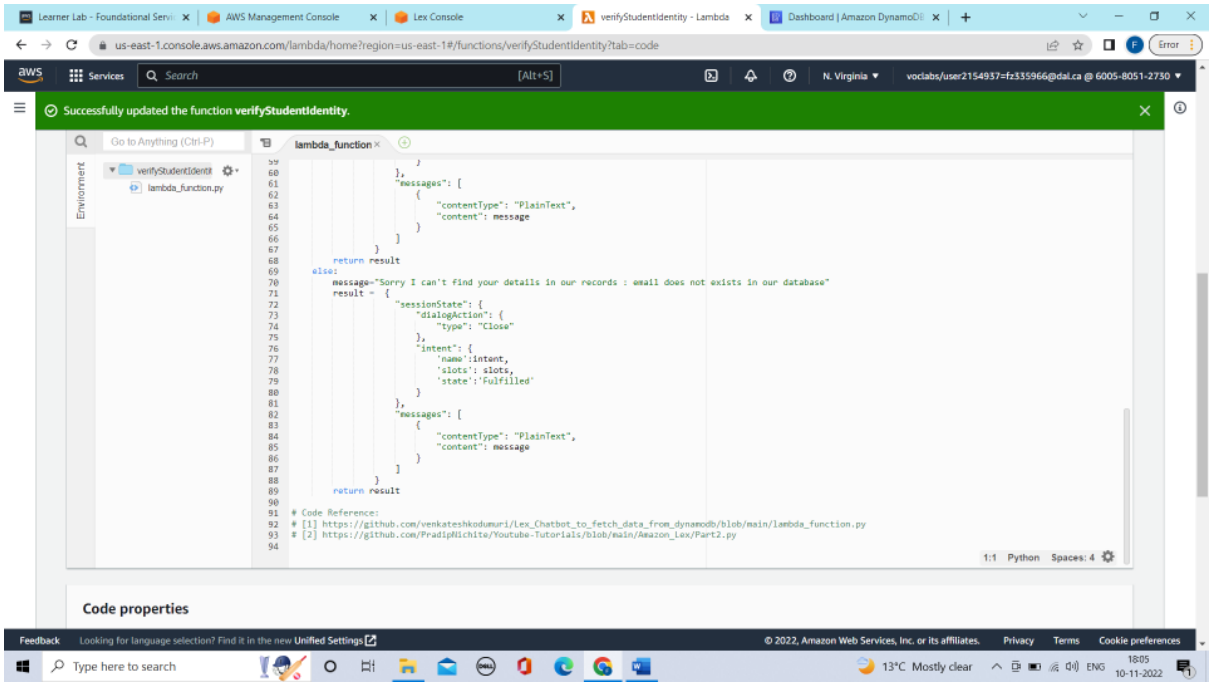

Fig 40:Deployed Lambda function code

- After writing the code, click on “Deploy” as shown in the fig. 39.
- The alert message will be after deploying as “Successfully updated the function verifyStudentIdentity” as shown in the fig. 40.
- After deploying, go to Amazon Lex intents page and click on “Build” as shown in the fig. 41.
- The status changes from “Building” (Refer fig. 41) to “Built” (Refer fig. 42) then we can click on the “Test” (Refer fig. 43).

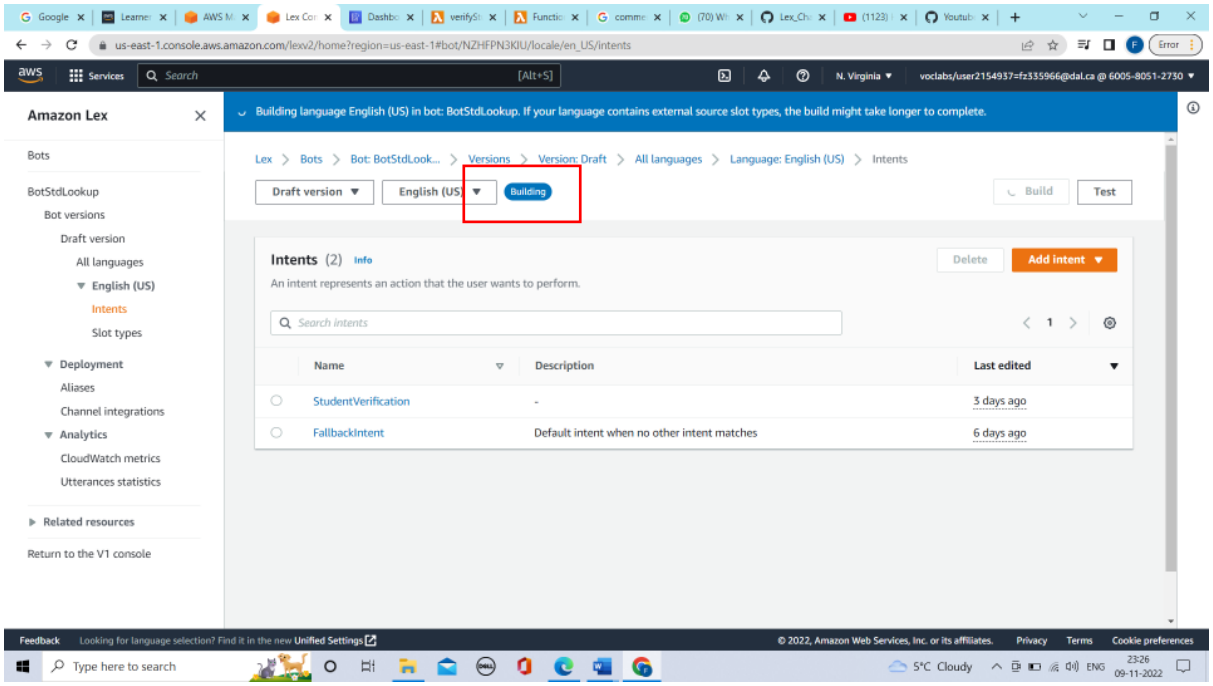

Fig 41: Amazon Lex intents page

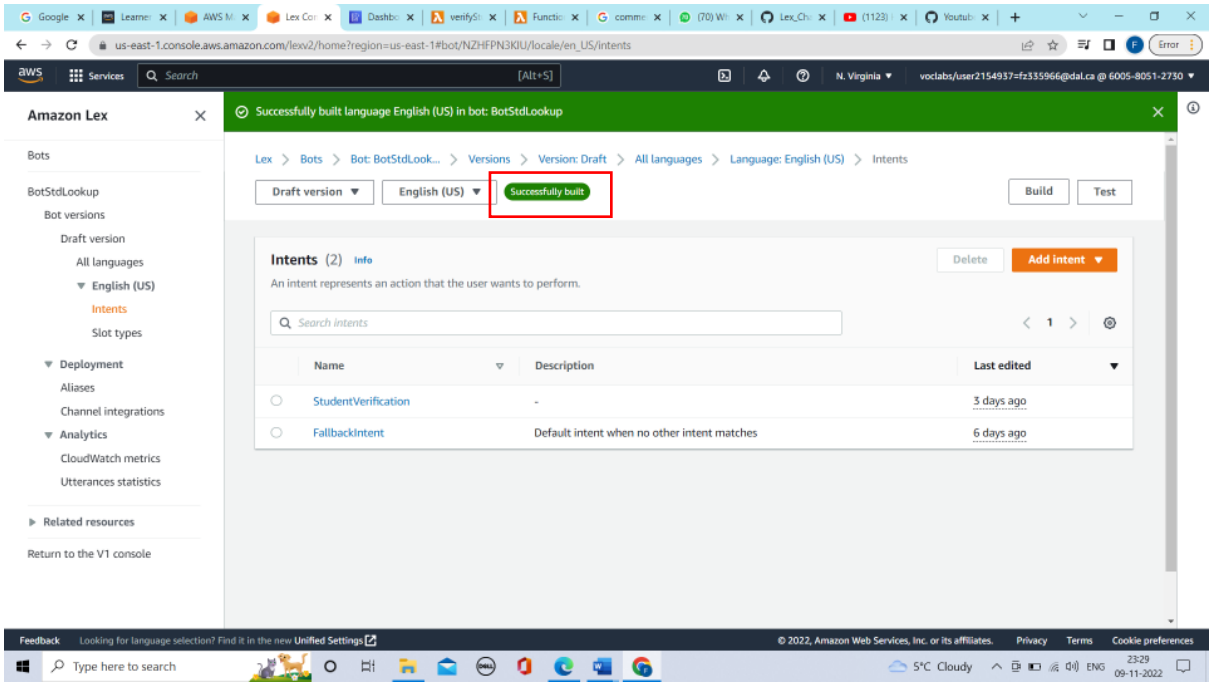

Fig 42: Amazon Lex intents page

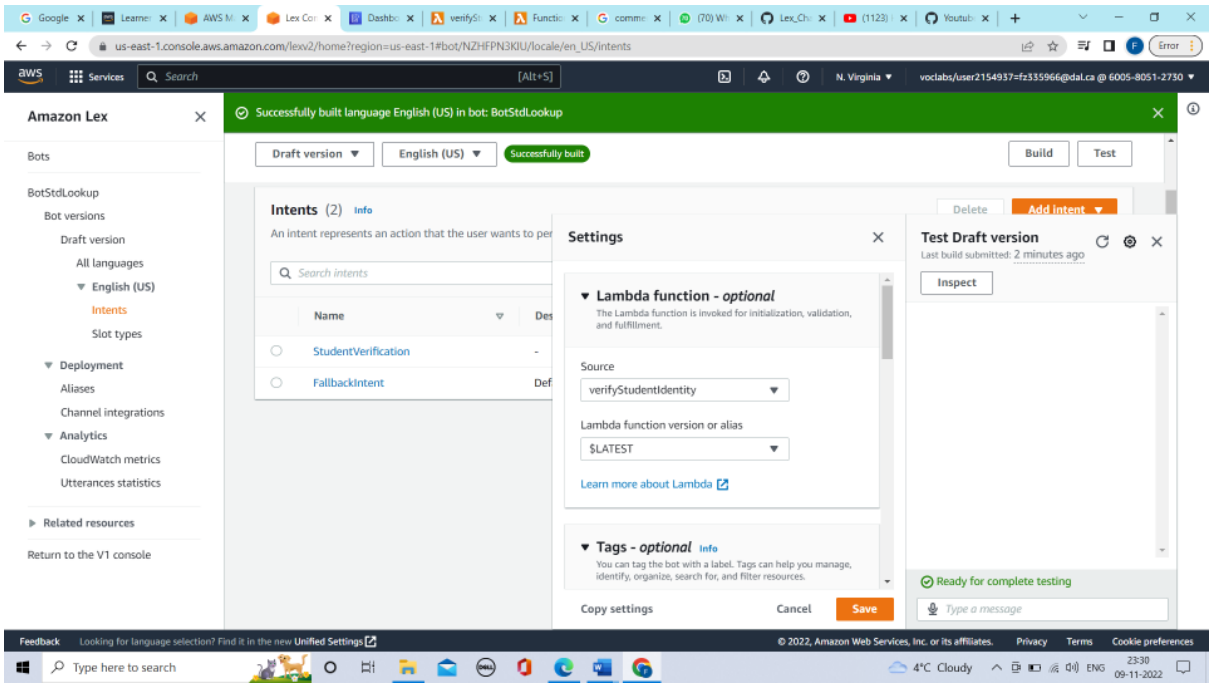

Fig 43: Amazon Lex intents page – test ChatBot

- The test ChatBot appears as shown in the fig. 43. We need to go to the settings options beside “Test Draft Version” to select the source as “verifyStudentIdentity” which is our lambda function and select the version of the function as “$LATEST” (Refer fig. 43, 44).
- Also verify the other settings mentioned in the fig. 45, 46, 47 & 48.

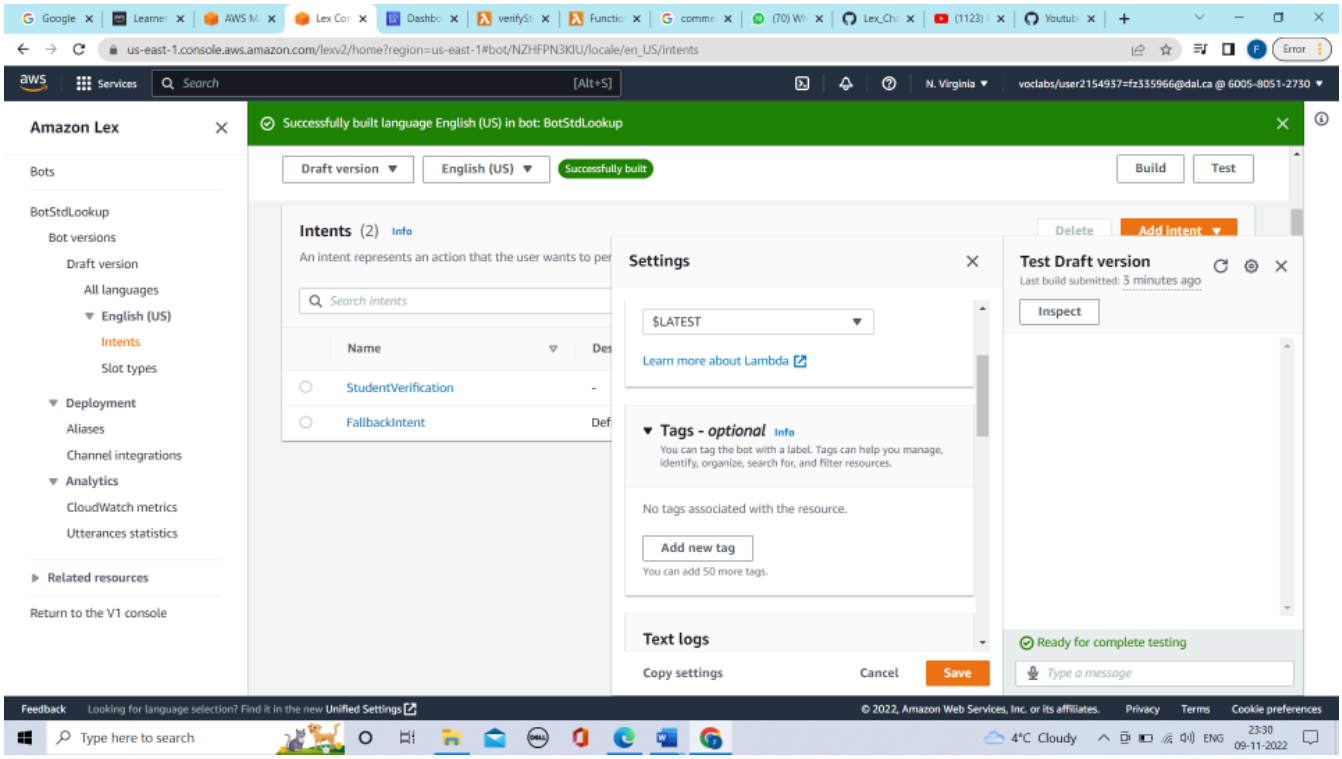

Fig 44: Amazon Lex intents page – test ChatBot

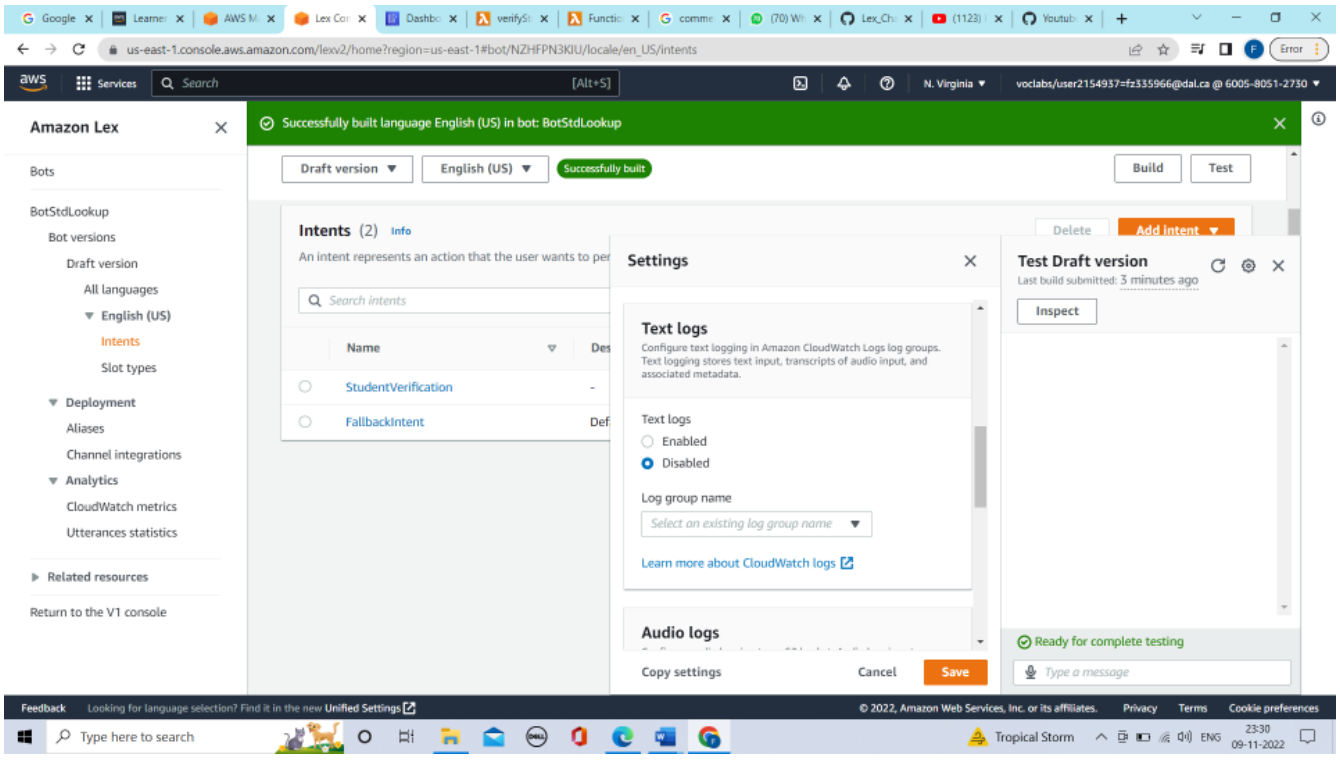

Fig 45: Amazon Lex intents page – test ChatBot

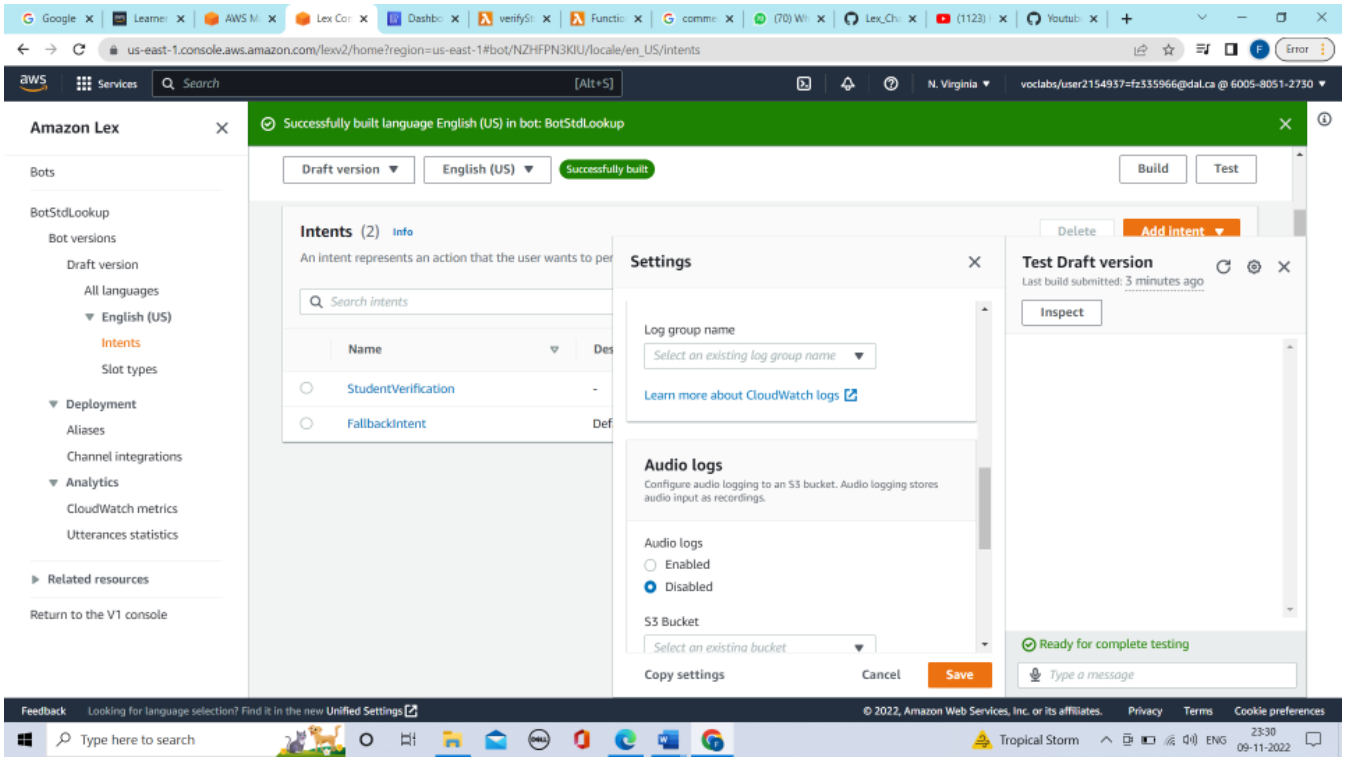

Fig 46: Amazon Lex intents page – test ChatBot

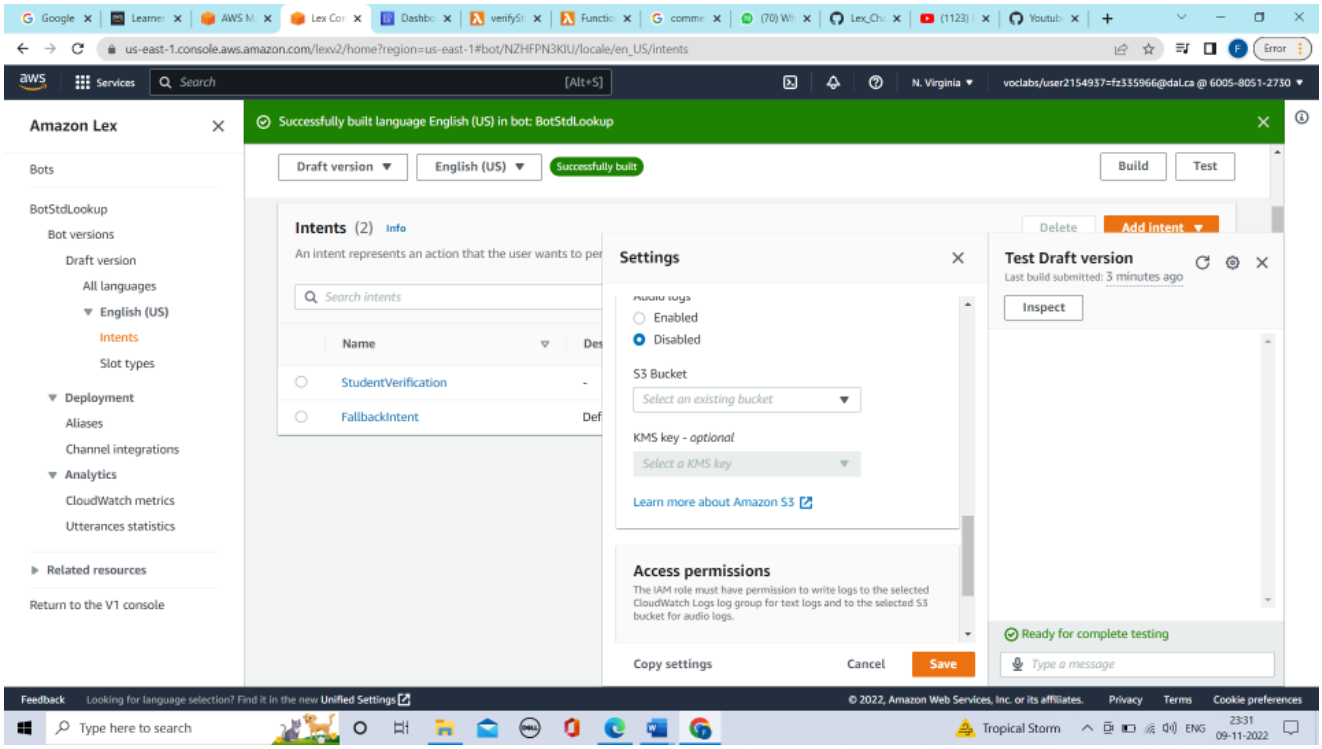

Fig 47: Amazon Lex intents page – test ChatBot

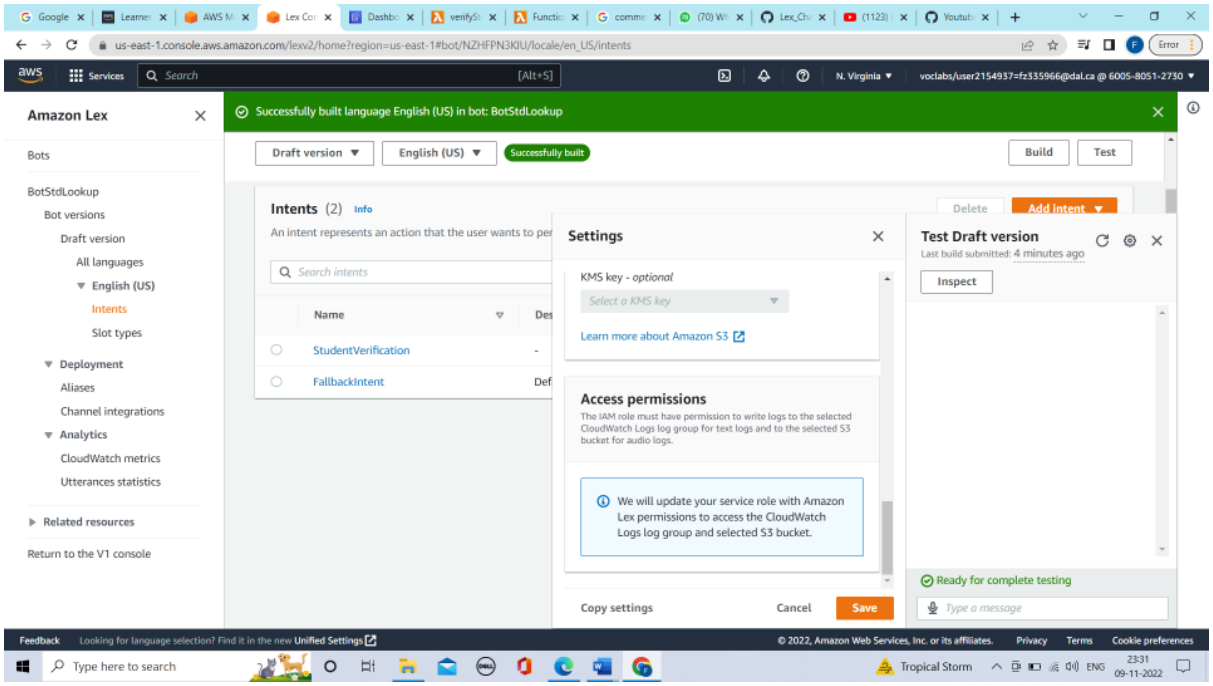

Fig 48: Amazon Lex intents page – test ChatBot

Step 4: Testcases for BotStdLookup

Test case 1:

- Fig. 49 depicts when the user provides same name and email that is present in the database.
- The ChatBot verifies the details using the lambda function by showing the message as "Your details are verified" as shown in the fig. 50.

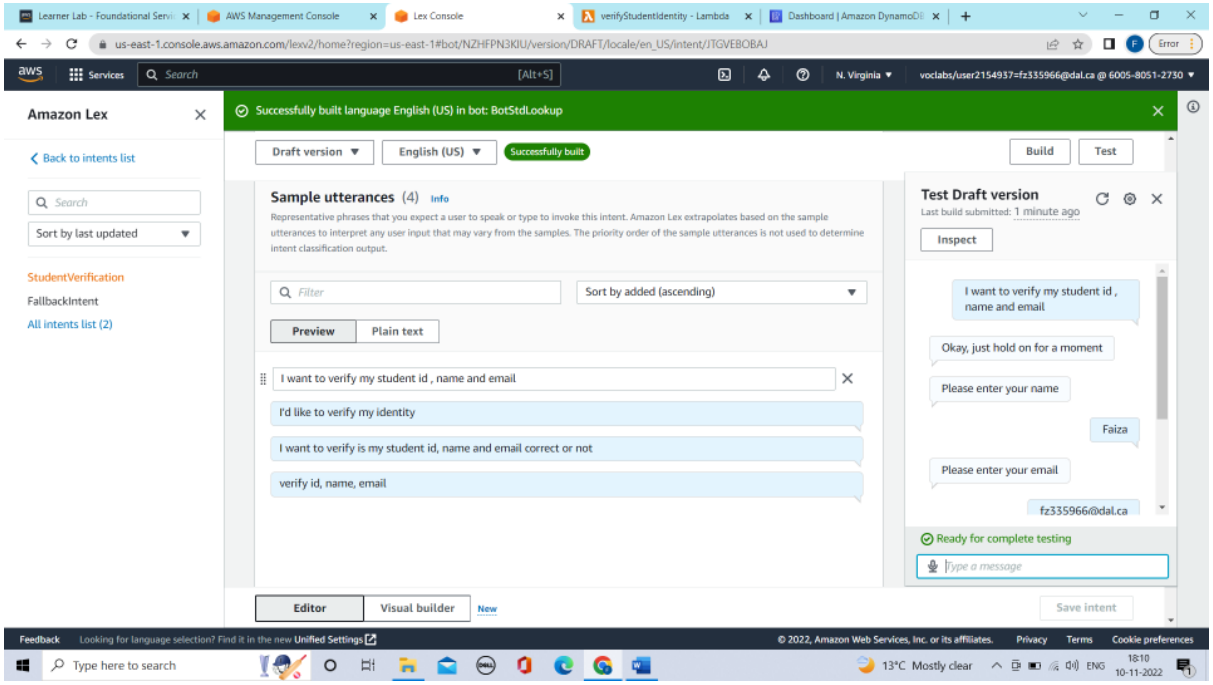

Fig 49: Amazon Lex intents page – test ChatBot

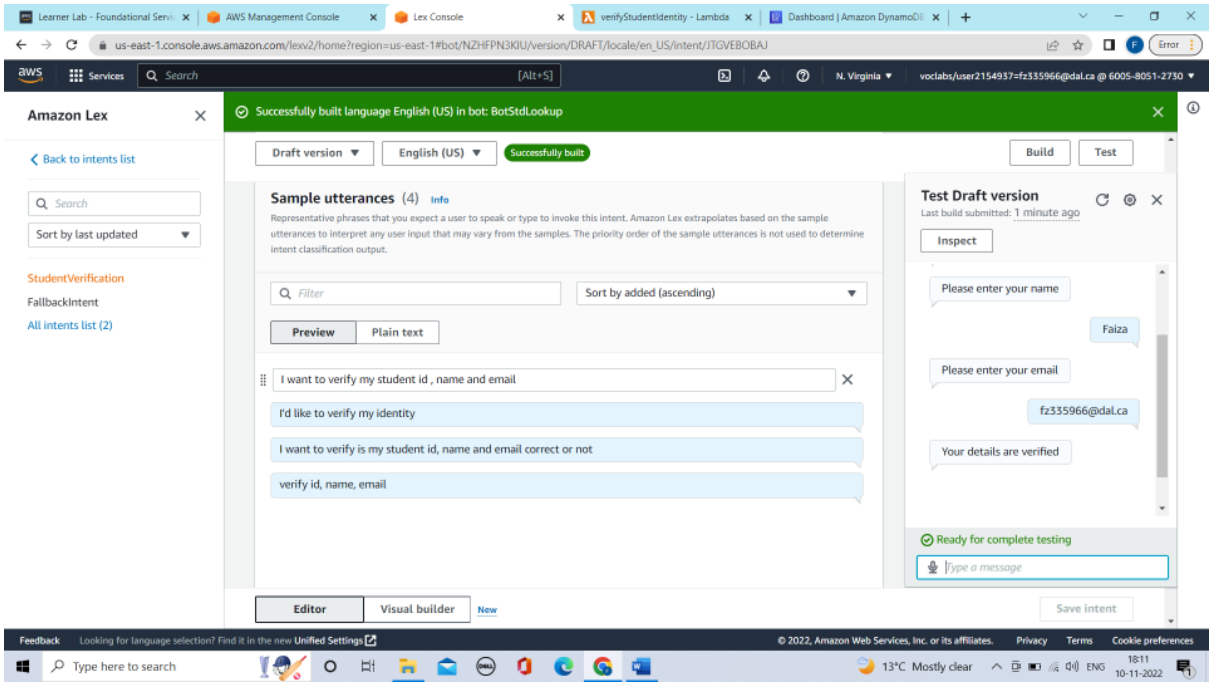

Fig 50: Amazon Lex intents page – test ChatBot

Test case 2:

- Fig. 51 depicts when the user provides incorrect name and email that is present in the database.
- The ChatBot verifies that the details present in the database is not same as the one user provided. So the ChatBot prompts user with the message “Sorry I can’t find your details in the records: email does not exists in our database” (Refer fig. 52).

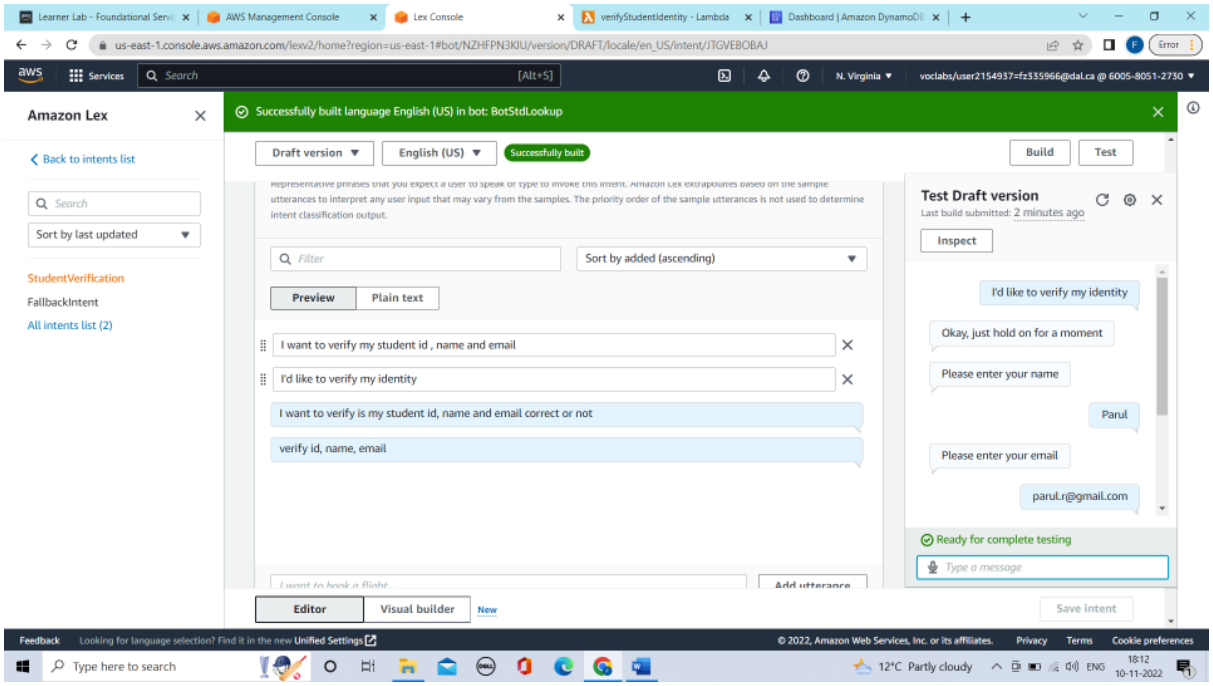

Fig 51: Amazon Lex intents page – test ChatBot

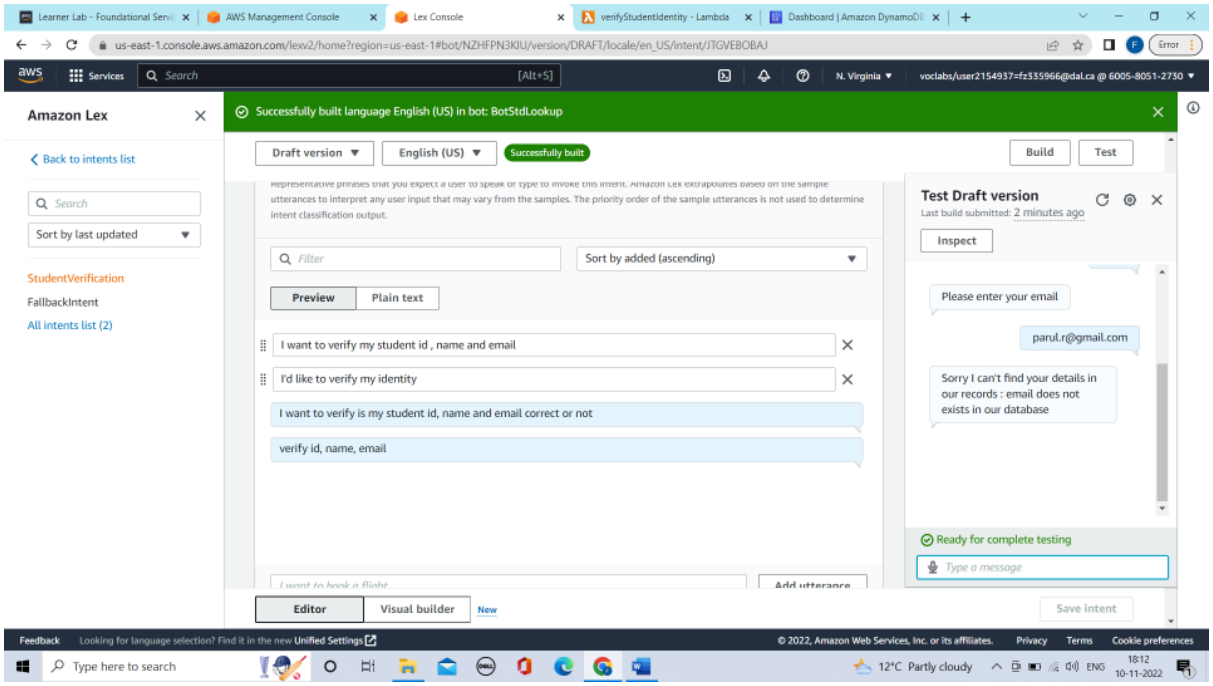

Fig 52: Amazon Lex intents page – test ChatBot

Test case 3:

- When the user enters the incorrect name or name in lowercase. For instance, the name in database is stored as “Nafisa” but the user types “nafisa” is ChatBot as shown in the fig. 53.
- This will prompt the user with an error message as “Your details could’nt be verified: name did not matched with the information provided” (Refer fig. 54).

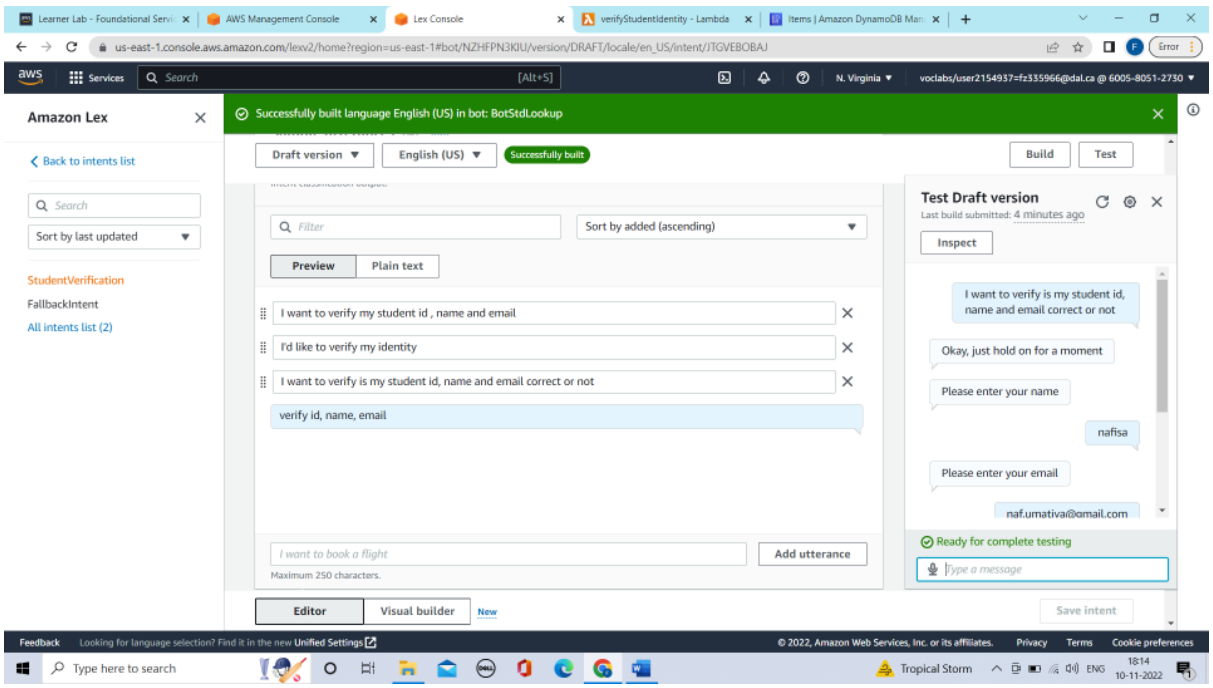

Fig 53: Amazon Lex intents page – test ChatBot

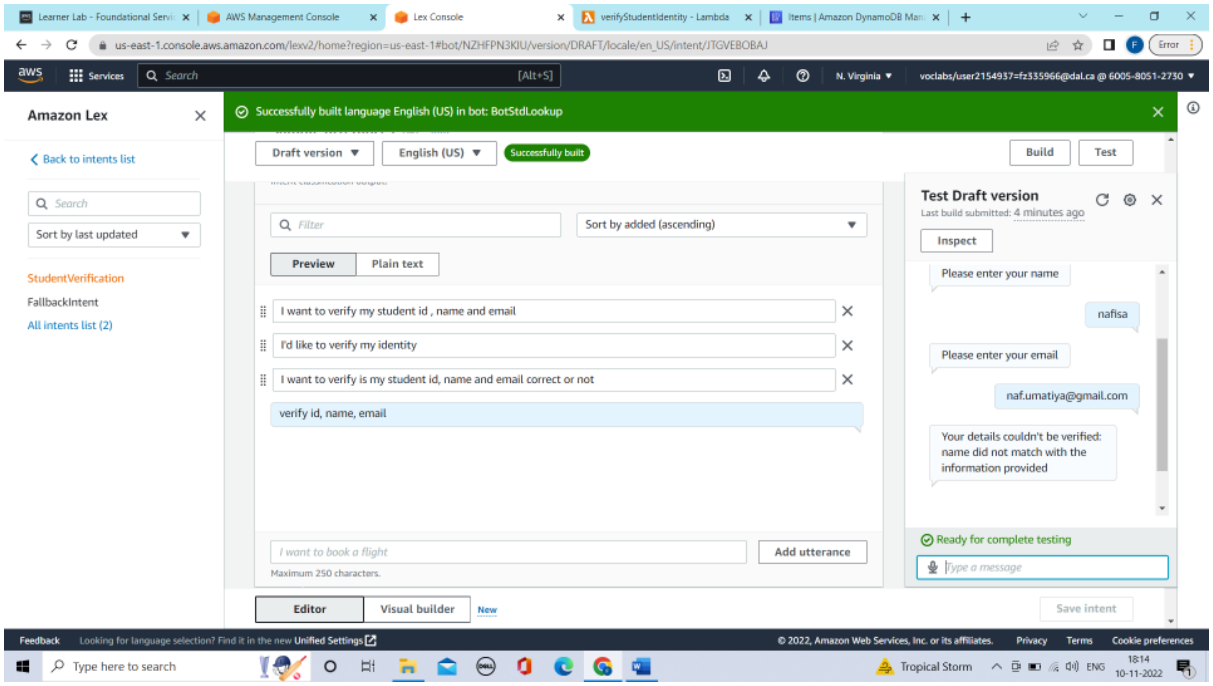

Fig 54: Amazon Lex intents page – test ChatBot

Test case 4:

- When the user enters correct name but incorrect email as shown in the fig. 55.
- The ChatBot prompts the user with the error message as “Sorry I can’t find the details in our records: email does not exists in our database” as shown in the fig. 56.

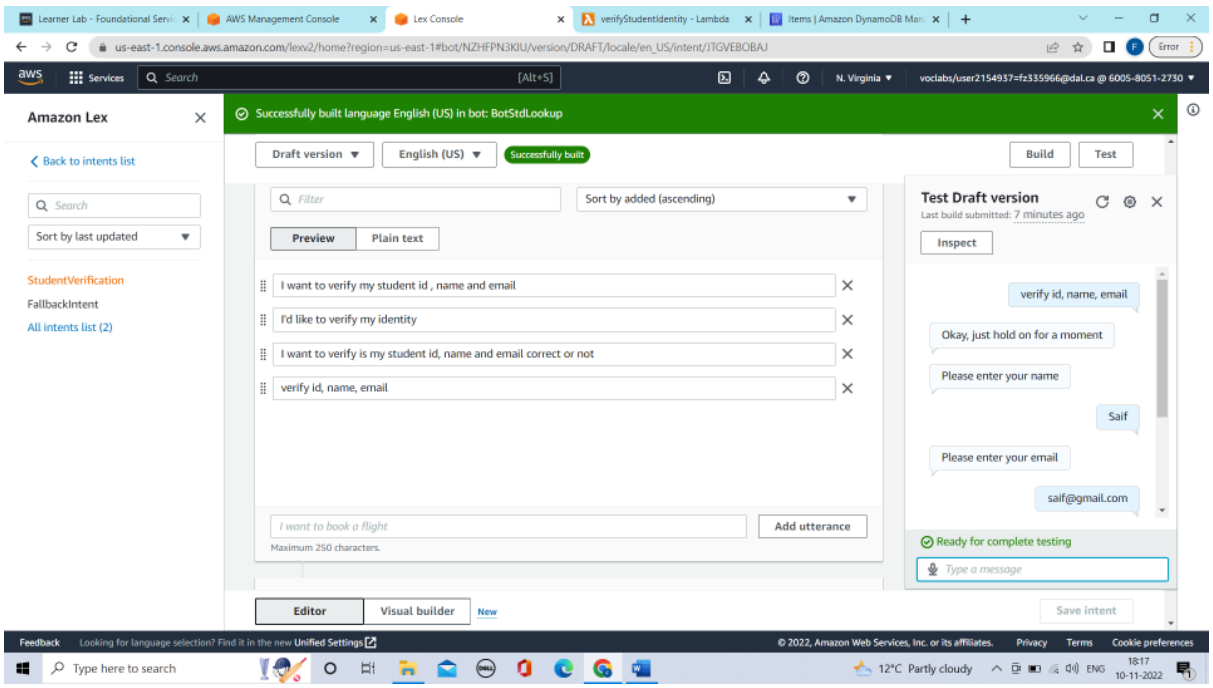

Fig 55: Amazon Lex intents page – test ChatBot

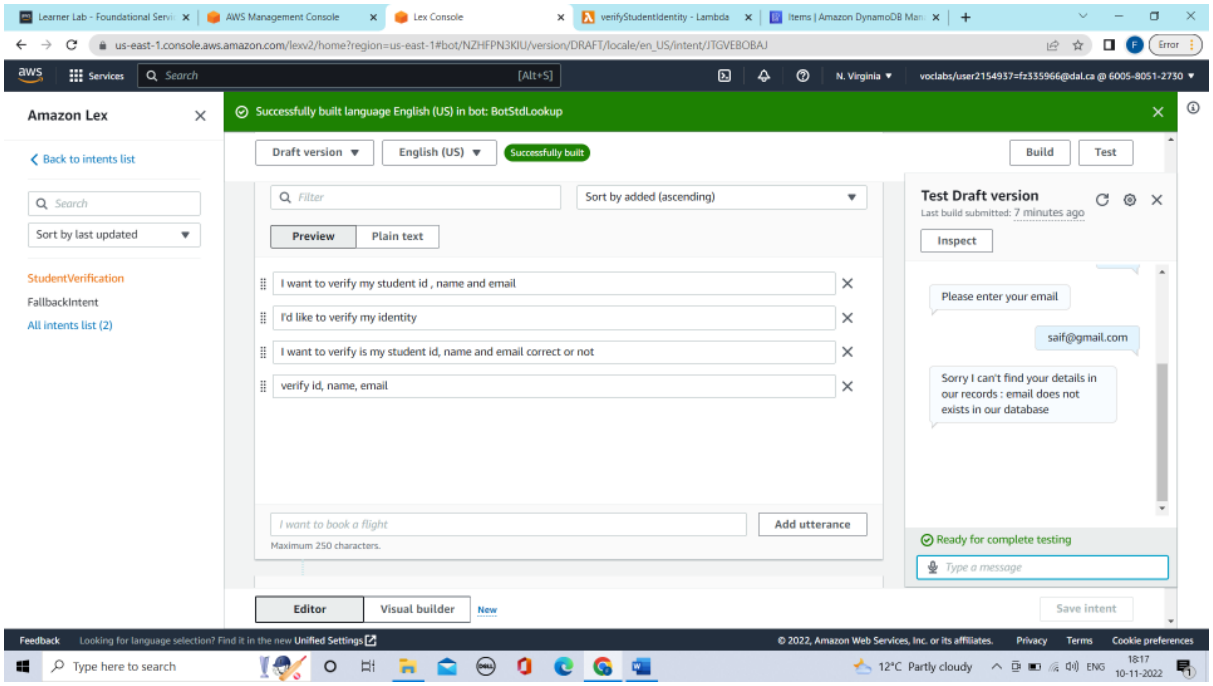

Fig 56: Amazon Lex intents page – test ChatBot

Step 5:

Create a custom chatbot [BotProfAppointment] using AWS Lex, which can be used to book office hours (E.g., booking office hours to meet a professor)

The steps I took is given below:

- Go to the Amazon Lex service to create a table as shown in the fig. 57. Click on "Create Table" as shown in the fig. 57 & 58.

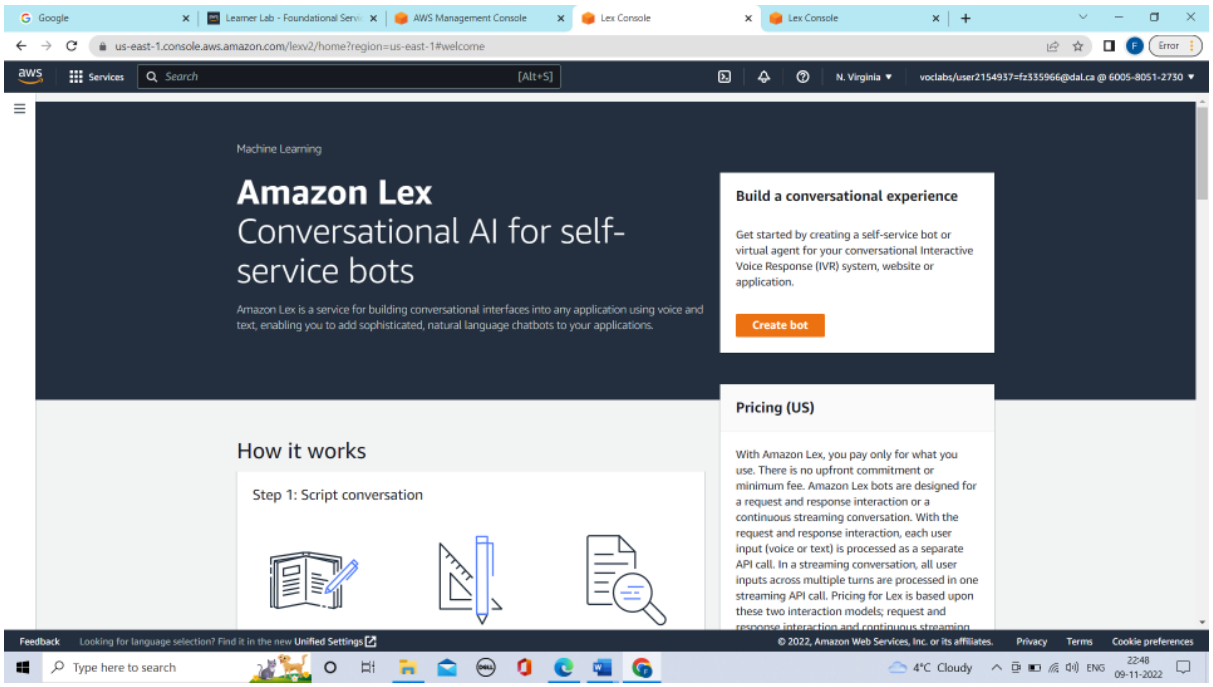

Fig 57: AWS Lex Home

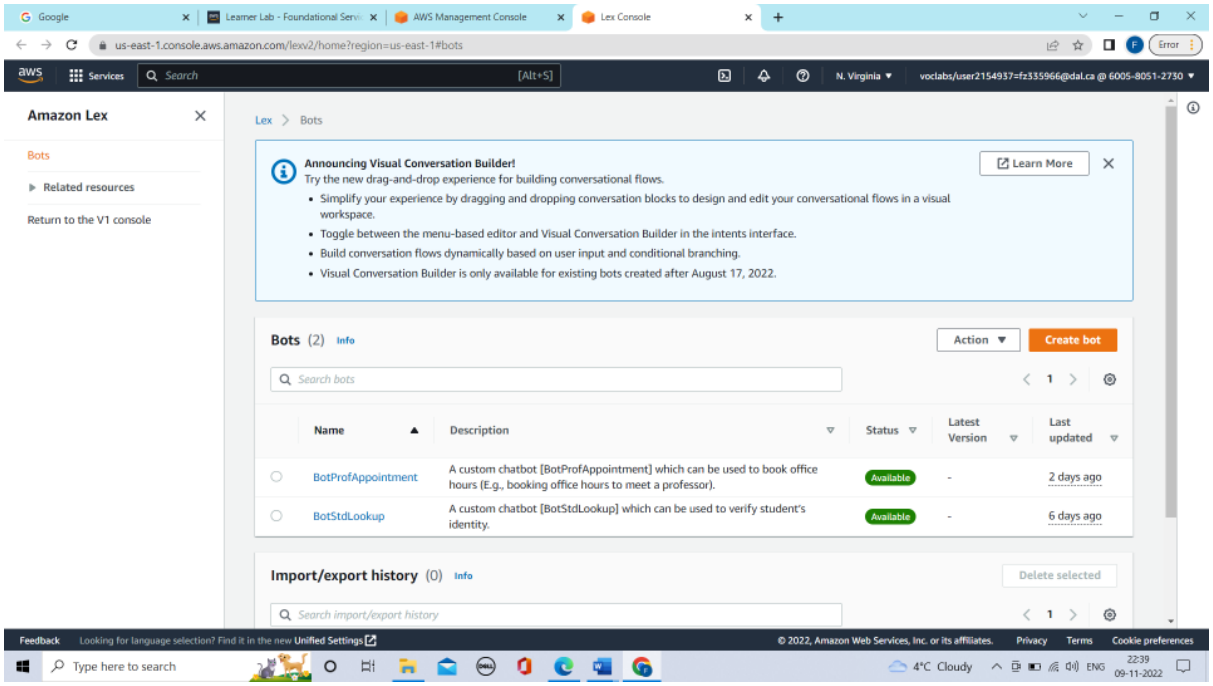

Fig 58: AWS Lex Home

- On clicking the "Create table", the configuration settings will open to provide details in order to create custom ChatBot as shown in the fig. 59, 60 & 61 and click on next page as shown in the fig 61.

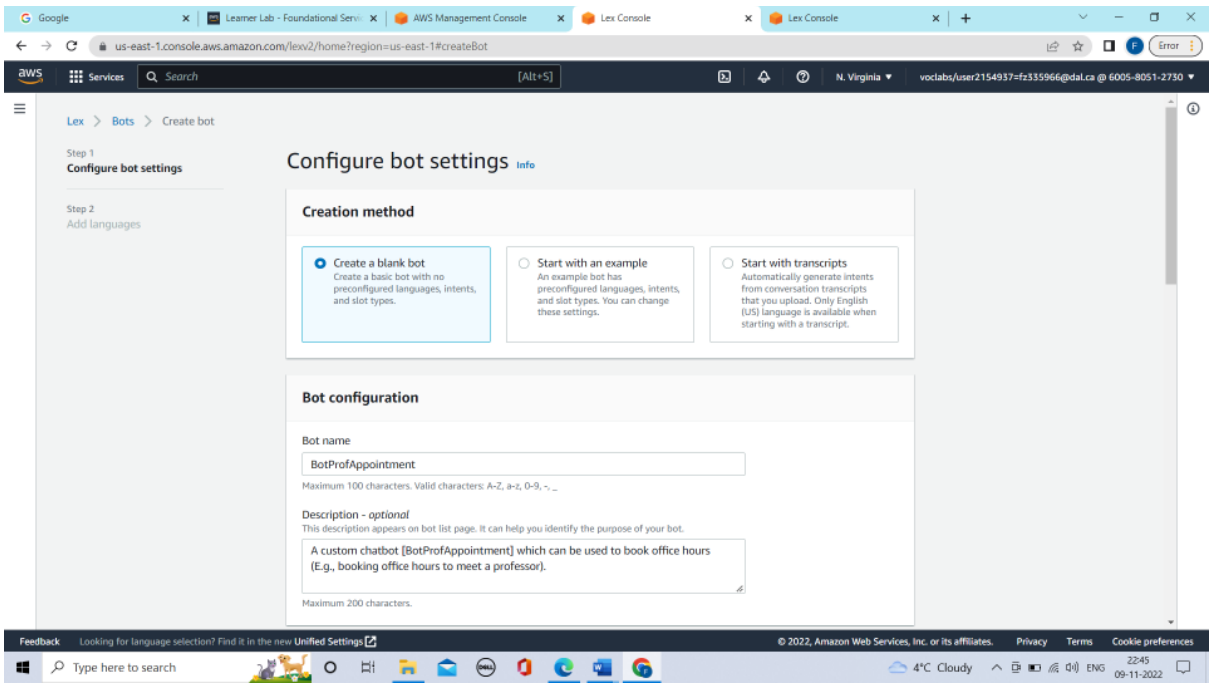

Fig 59: Configure bot settings page

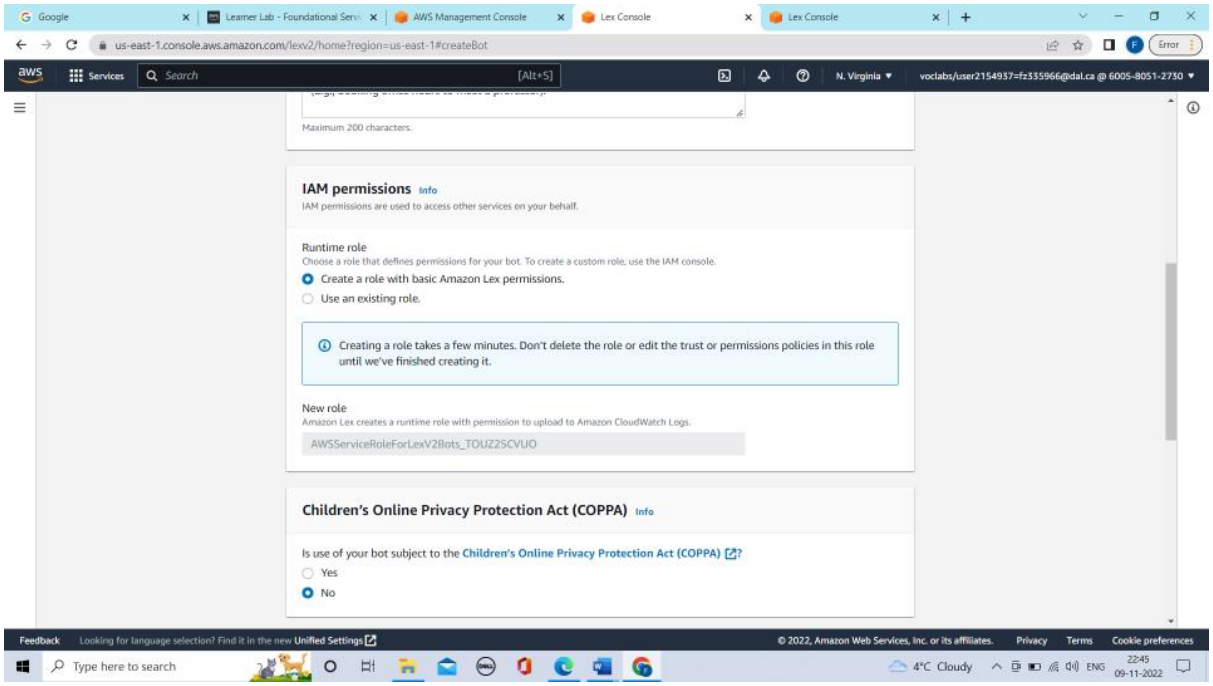

Fig 60: Configure bot settings page

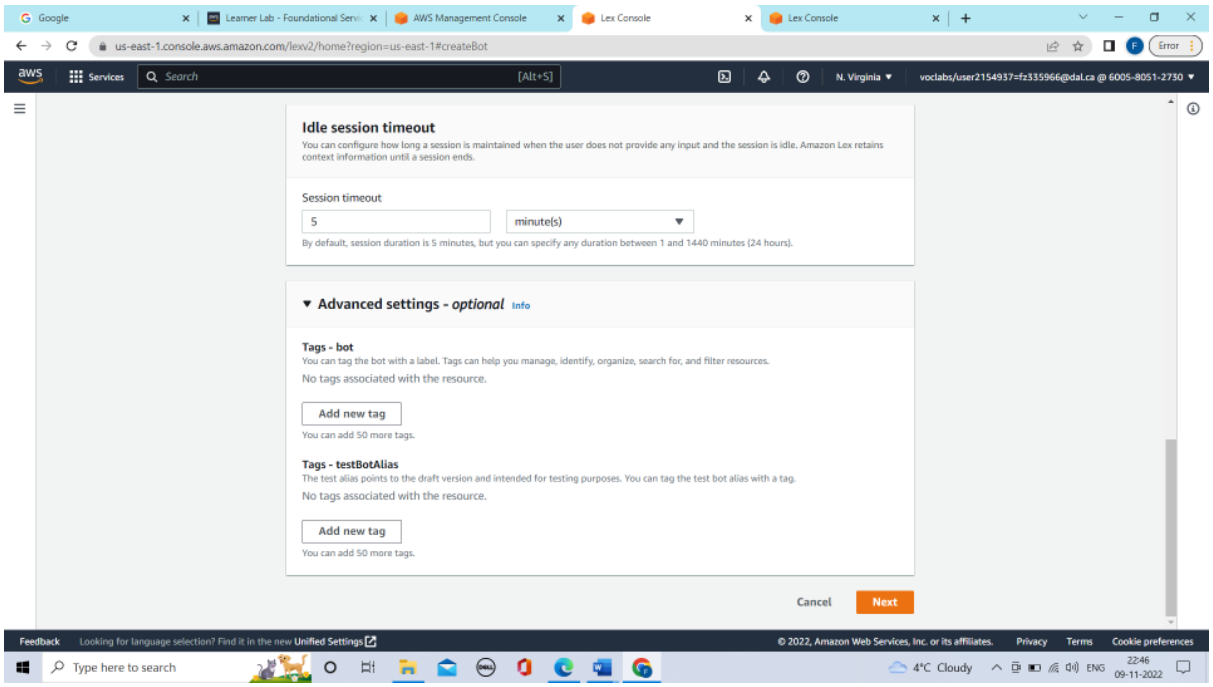

Fig 61: Configure bot settings page

- On the "Add language to bot" page, select "English" as the language and as the ChatBot is the text based application, select "This is only text based application" as shown in the fig. 62.
- After setting the language settings, the intents settings needs to made which is shown in the fig. 63.

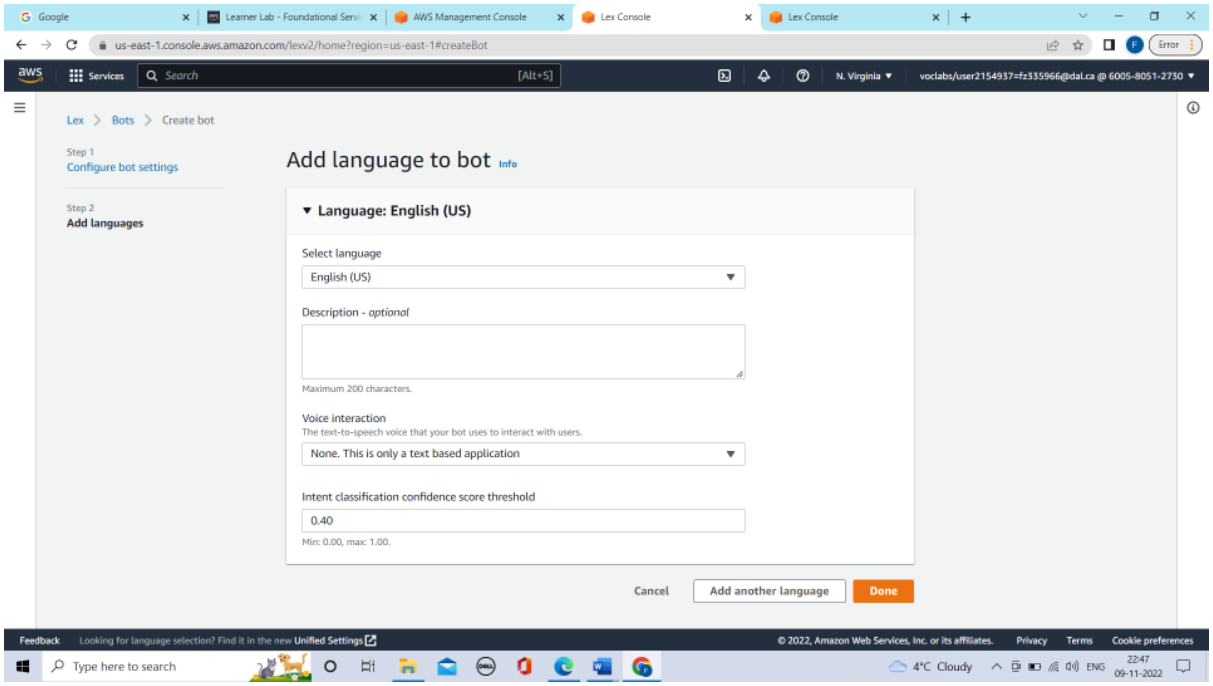

Fig 62: Add language to bot page

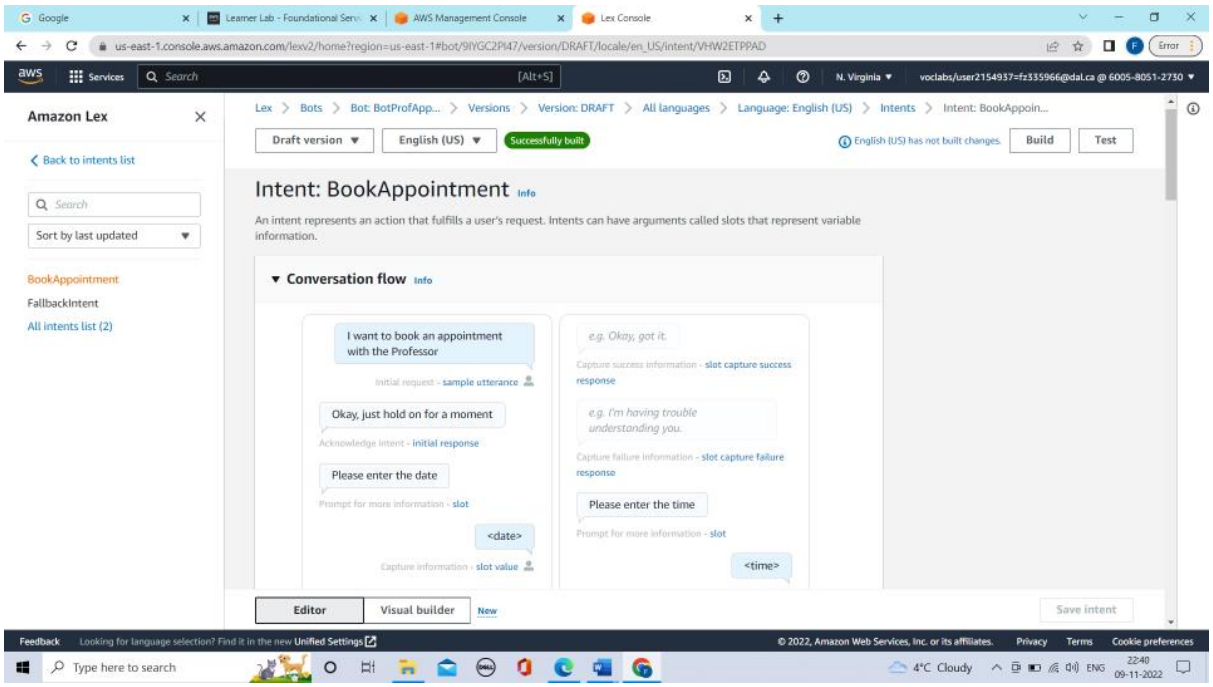

Fig 63: Add language to bot page

- The intent name provided is "BookAppointment" as shown in the fig. 63. The intent is nothing but the purpose of the Bot. The other intent details are filled as shown in the fig. 64
- User can make multiple uttrences like "hey", "I want to book an appointment with the Professor", "Book an appointment" and so on. The utterences is the text that user can start with or the replies from the users (Refer fig. 65 & 66).

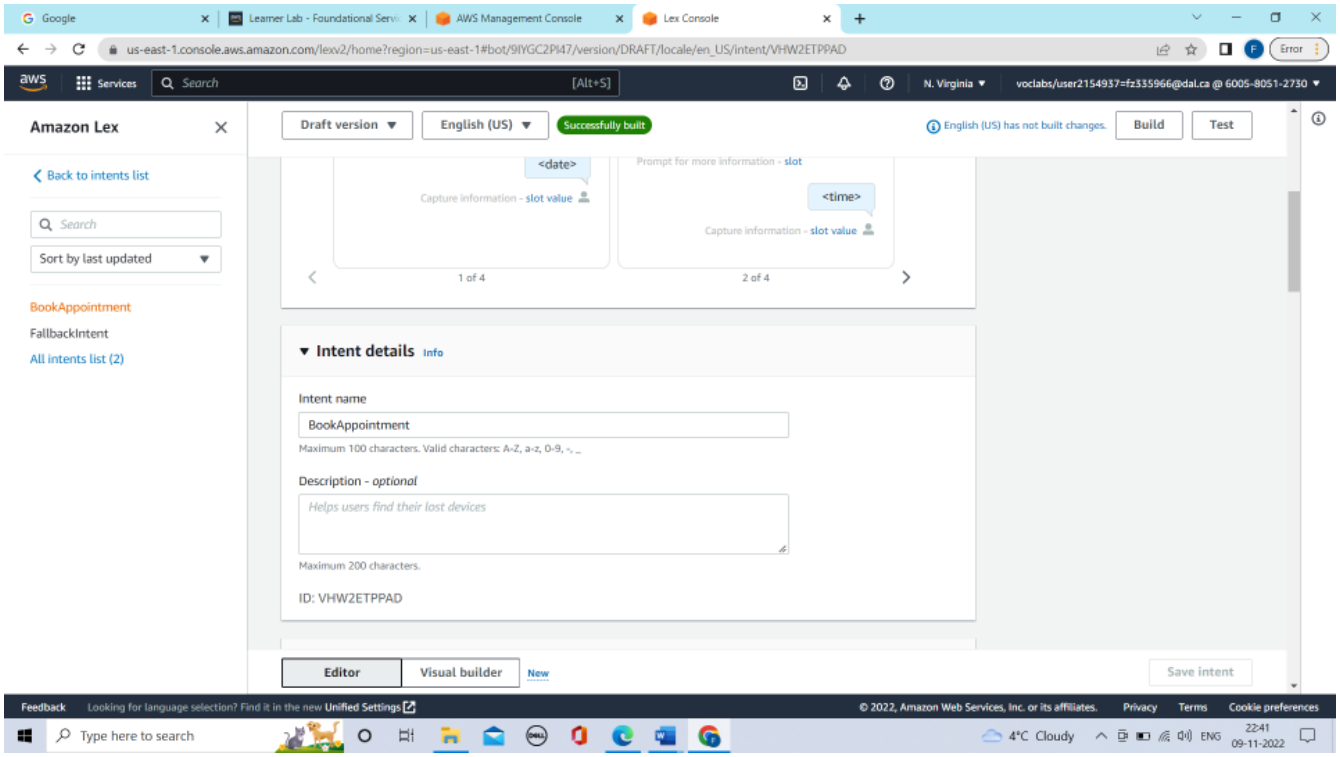

Fig 64: Add language to bot page

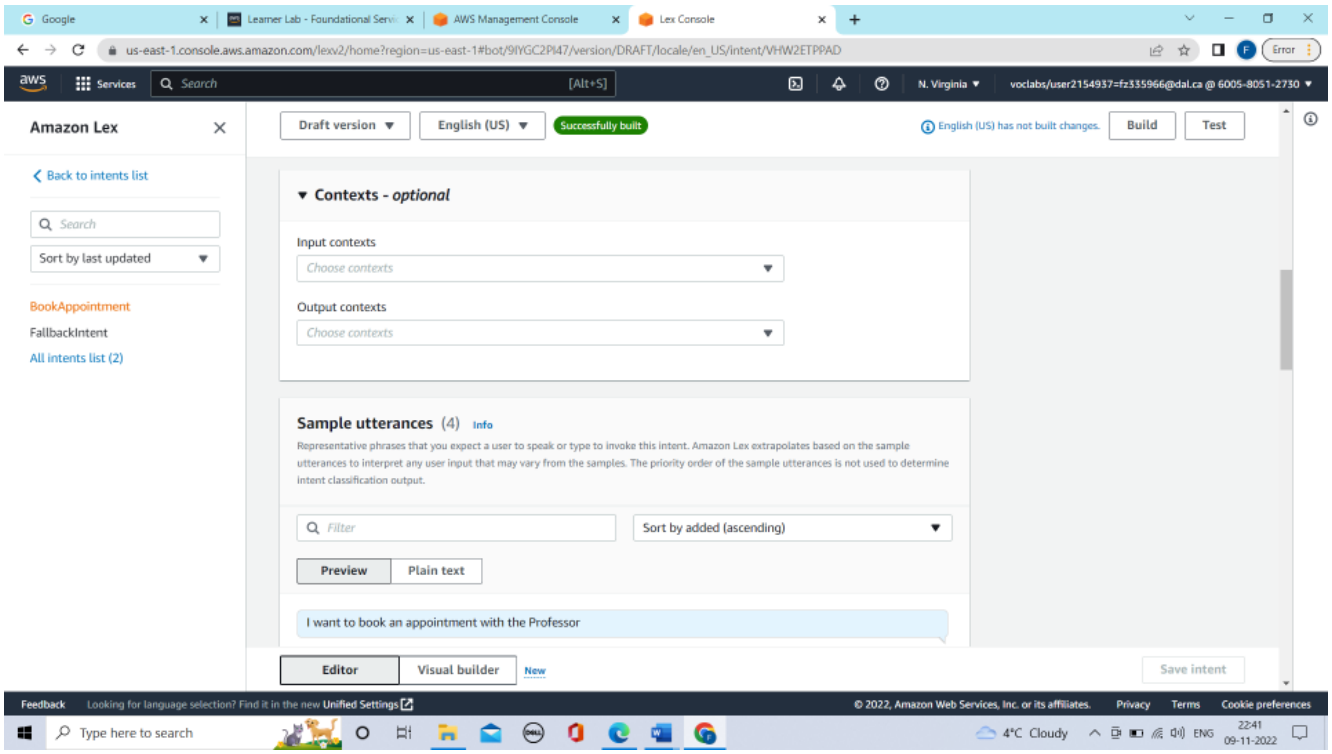

Fig 65: Contexts and sample utterance

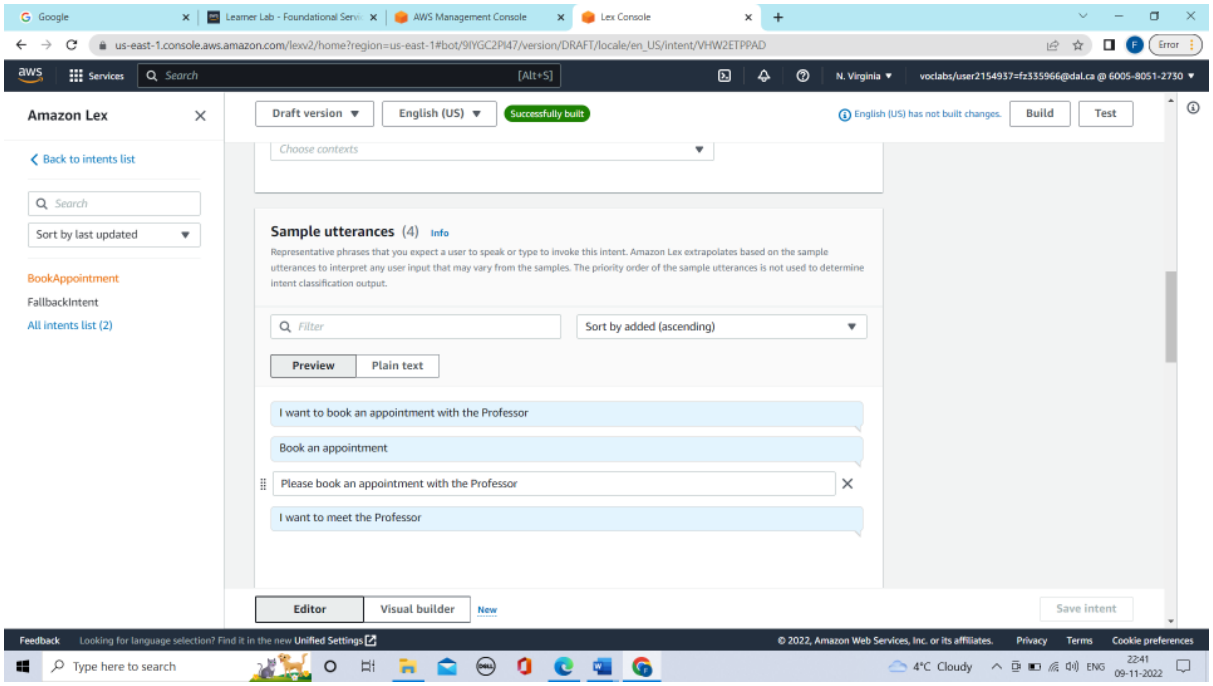

Fig 66: Sample utterance

- The user can set the initial response before starting the actual chat (Refer fig. 67). The slots are used to fulfill the bot intent. In order to book an appointment with the professor, the “Date” and “Time” are required. The slots ask user for their availability based on date and time (Refer fig. 68).

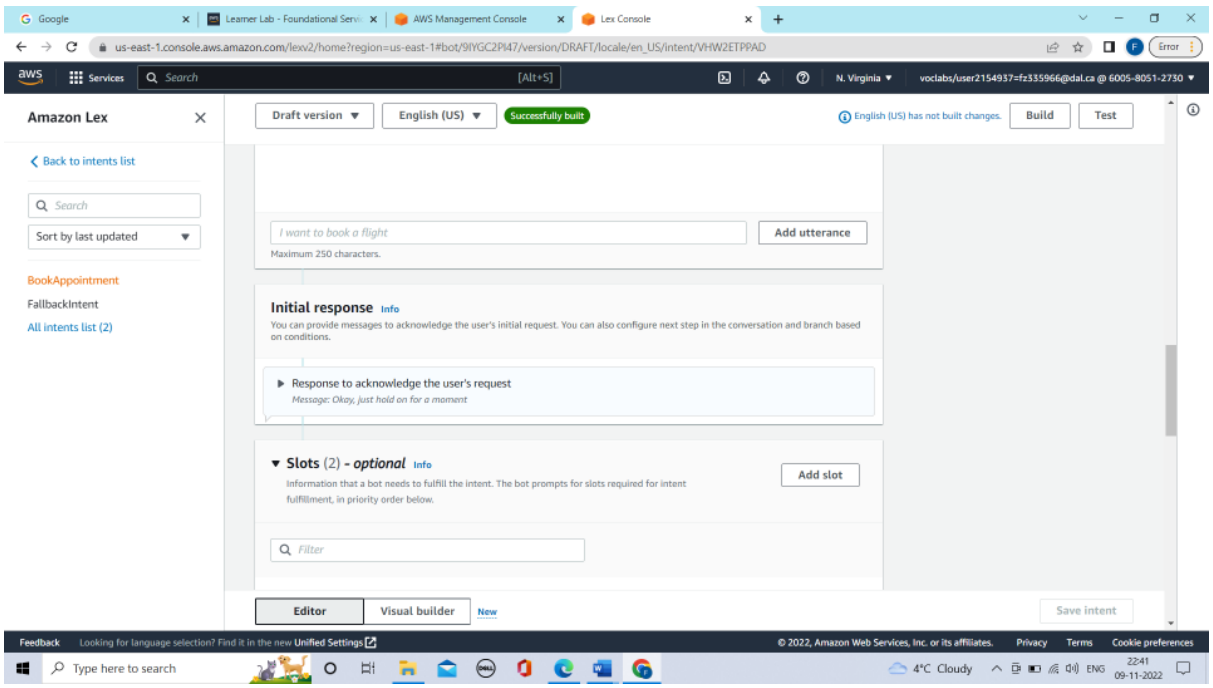

Fig 67: Initial Response

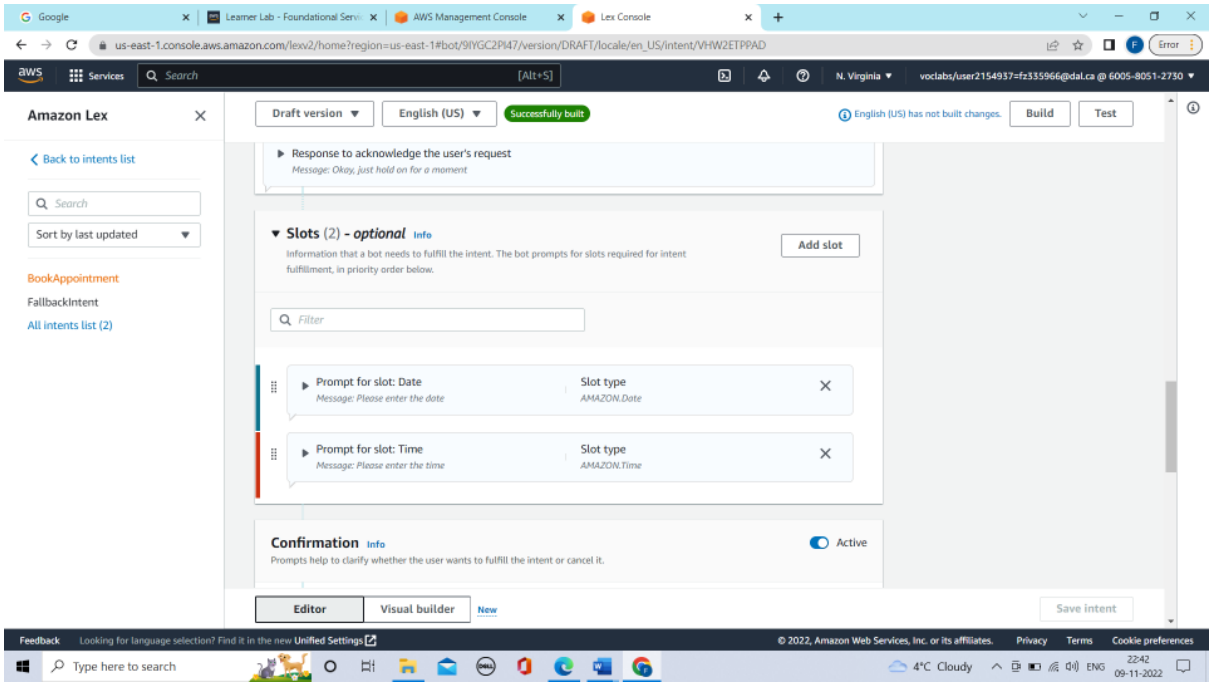

Fig 68: Slots

- Fulfillment given in fig. 69 allows users to know whether the intent is fulfilled or not

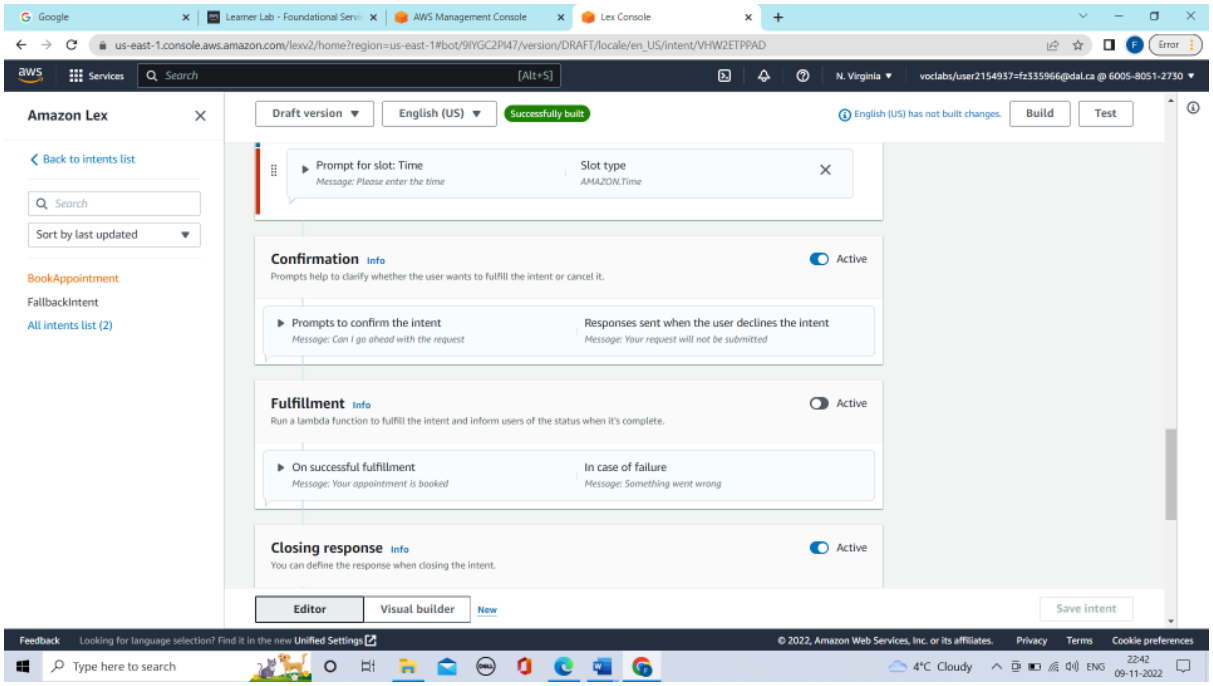

Fig 69: Fulfillment

- Closing response shown in the fig. 70 is nothing but the response sent to the user after the intent is fulfilled as shown in the fig. 69.

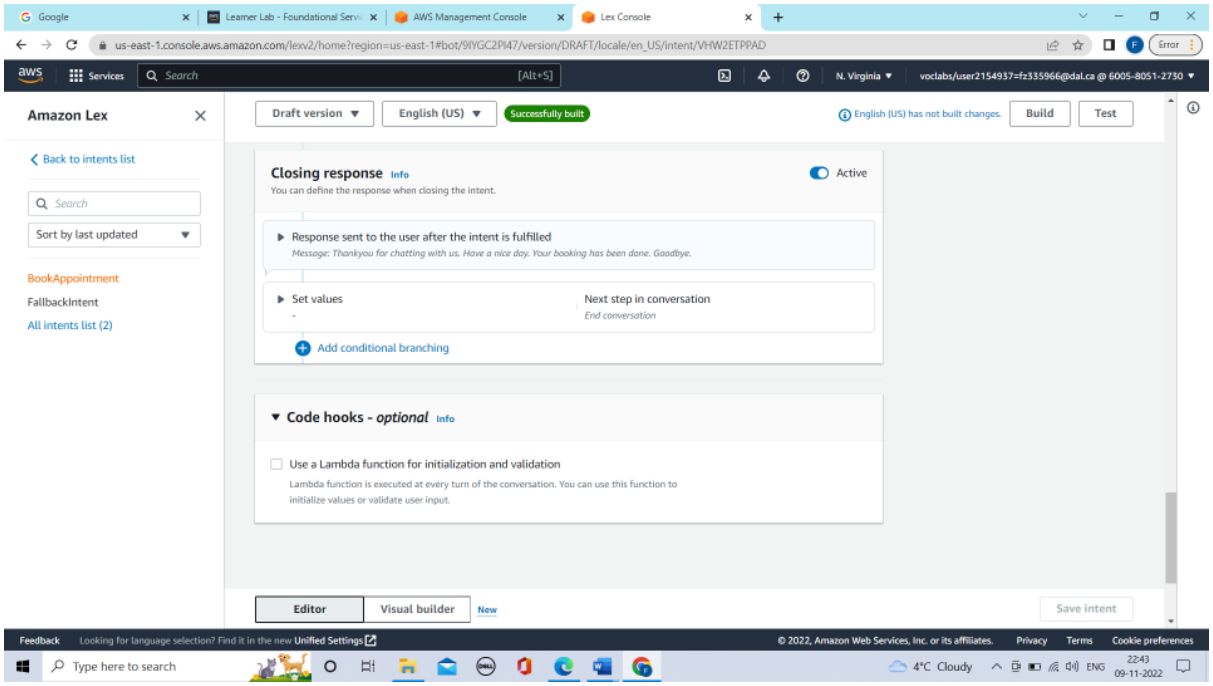

Fig 70: Closing Response

- After saving the intent, the user can see the saved intent which is mentioned in the fig. 71.

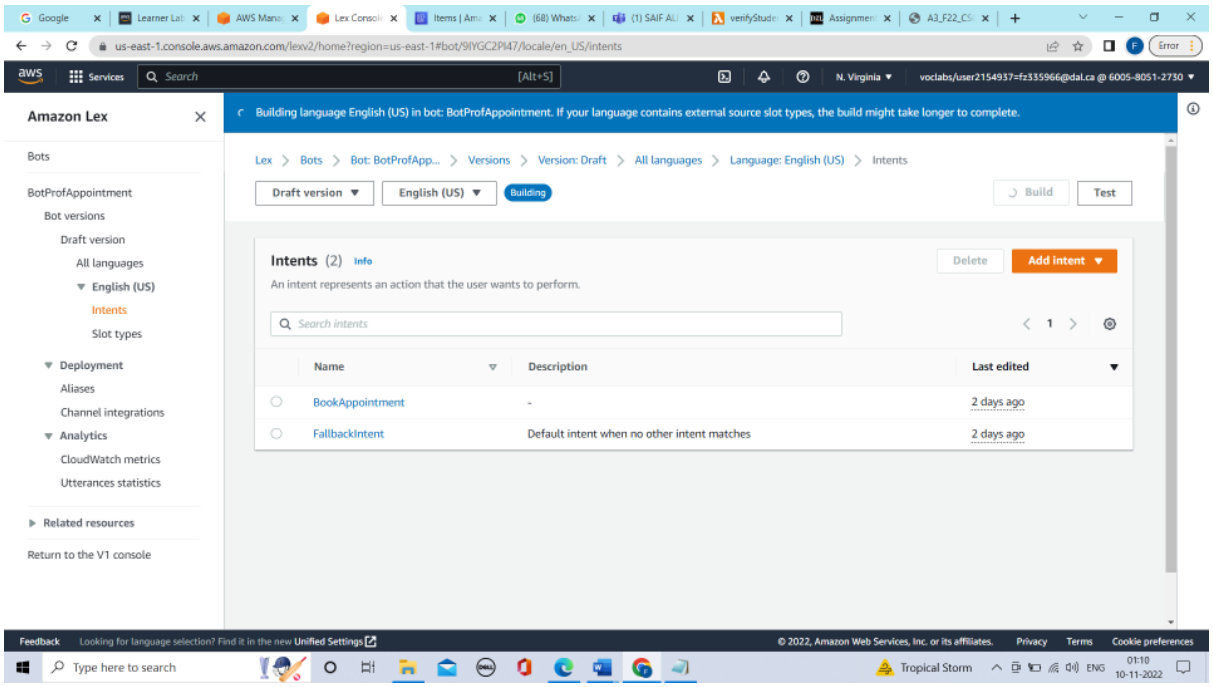

Fig 71: Saved intents page

- After saving the intent, build the ChatBot as shown in the fig. 71.
- On successfully building which is shown in the fig. 72, test the ChatBot.
- Below are the test cases of the ChatBot. Go to step 6.

Step 6: Testcases for BotProfAppointment

Test case 1:

- When the date and time provided by the user is accurate as shown in the fig. 72.
- The user went ahead with the request and the appointment was booked for that particular date by mentioning the closing response as “Thankyou for chatting with us. Have a nice day. Your booking has been done. Goodbye” (Refer fig. 73).

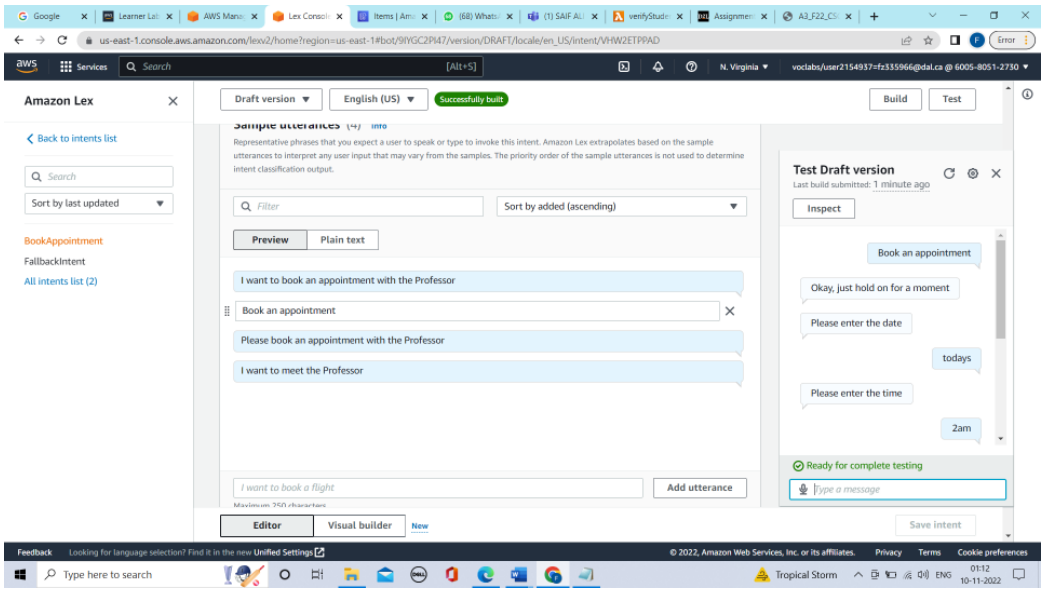

Fig 72: Amazon Lex – test ChatBot page

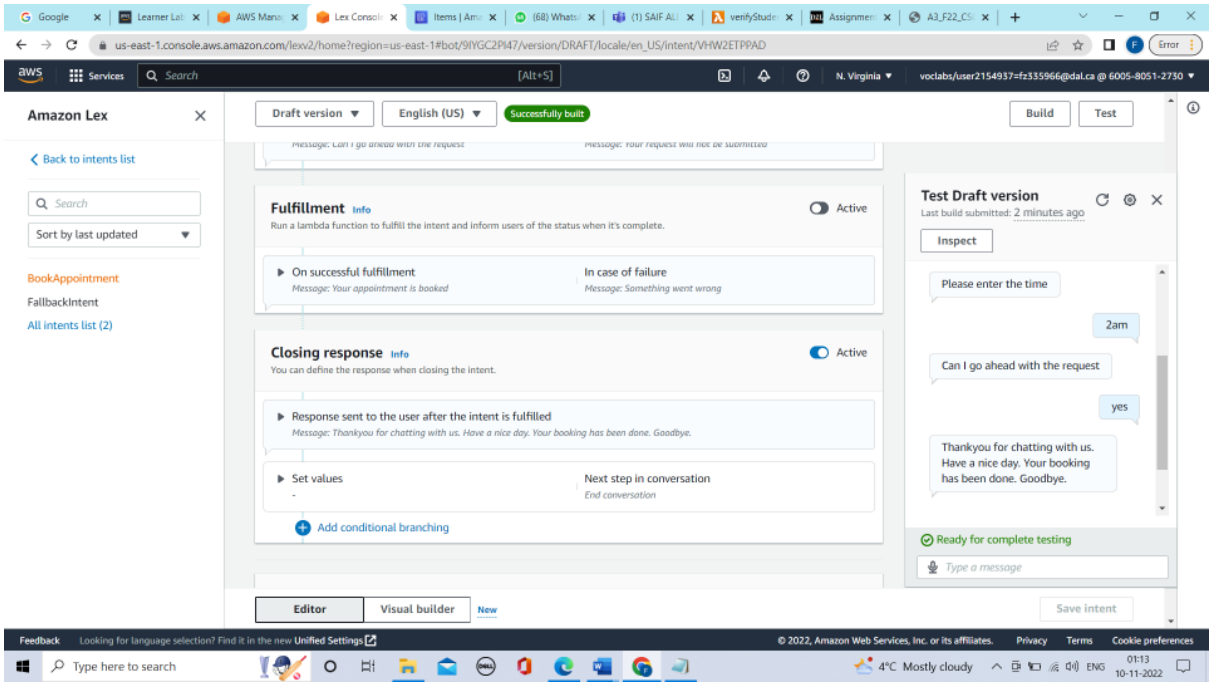

Fig 73: Amazon Lex – test ChatBot page

- The another similar example is given in fig. 74 & 75 but having different date format which is (DD/MM/YYYY).

- I used Amazon date and time to get the time what users enter as shown in the fig. 68 under slots.

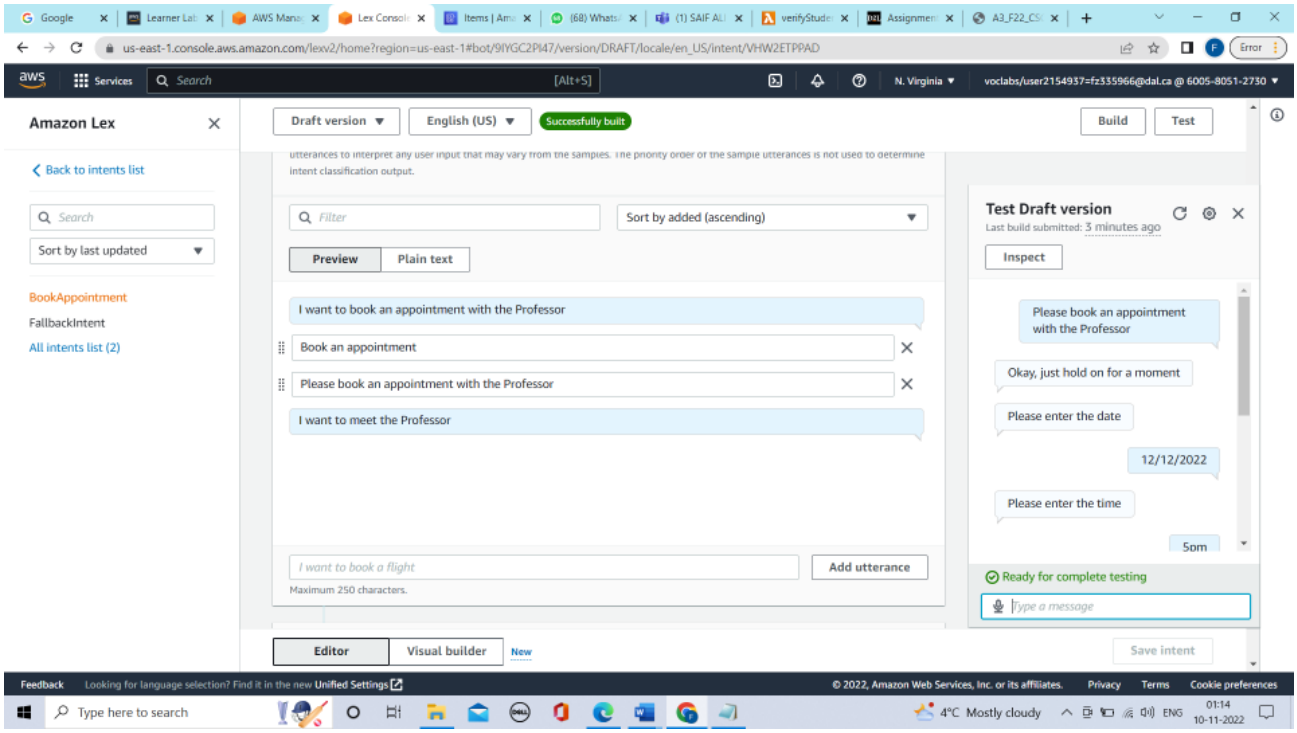

Fig 74: Amazon Lex – test ChatBot page

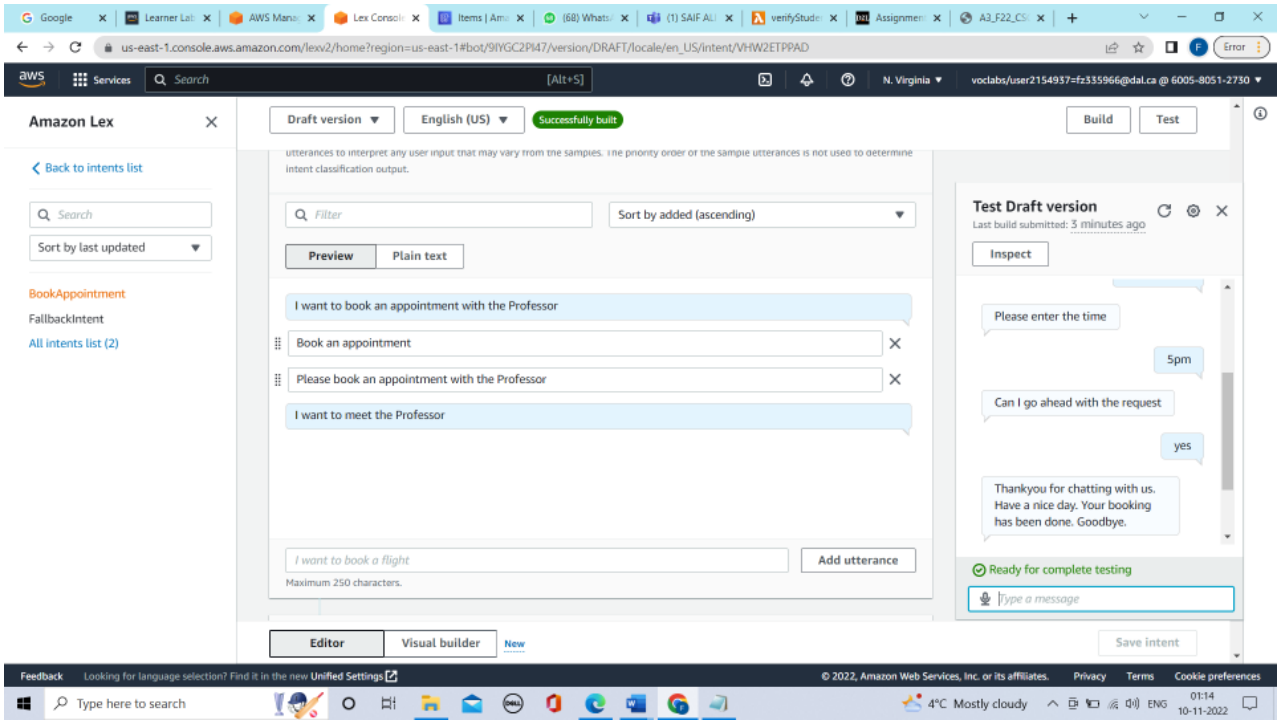

Fig 75: Amazon Lex – test ChatBot page

Test case 2:

- When the date and time provided is incorrect as shown in the fig. 76.
- The ChatBot askes to proceed with the request and the user proceeds. The ChatBot will still the book the appointment with the Professor even though the date provided is already passed. This is because there is no date and time validation logic mentioned using lambda function (Refer fig. 77).

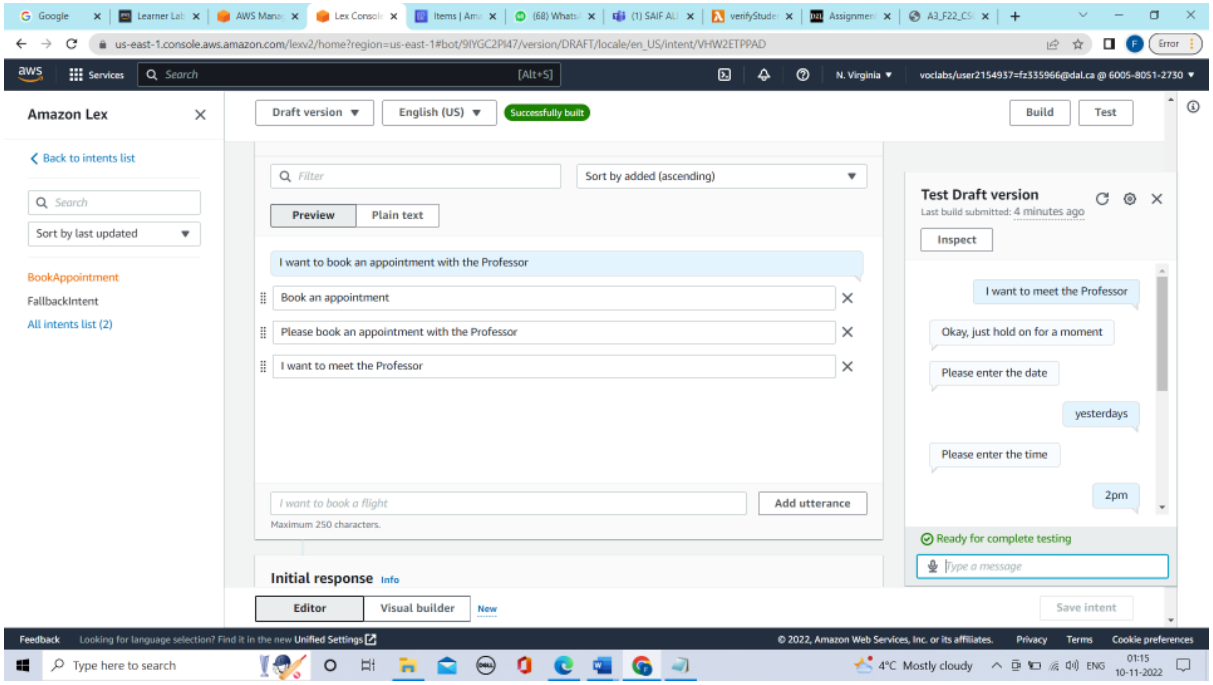

Fig 76: Amazon Lex – test ChatBot page

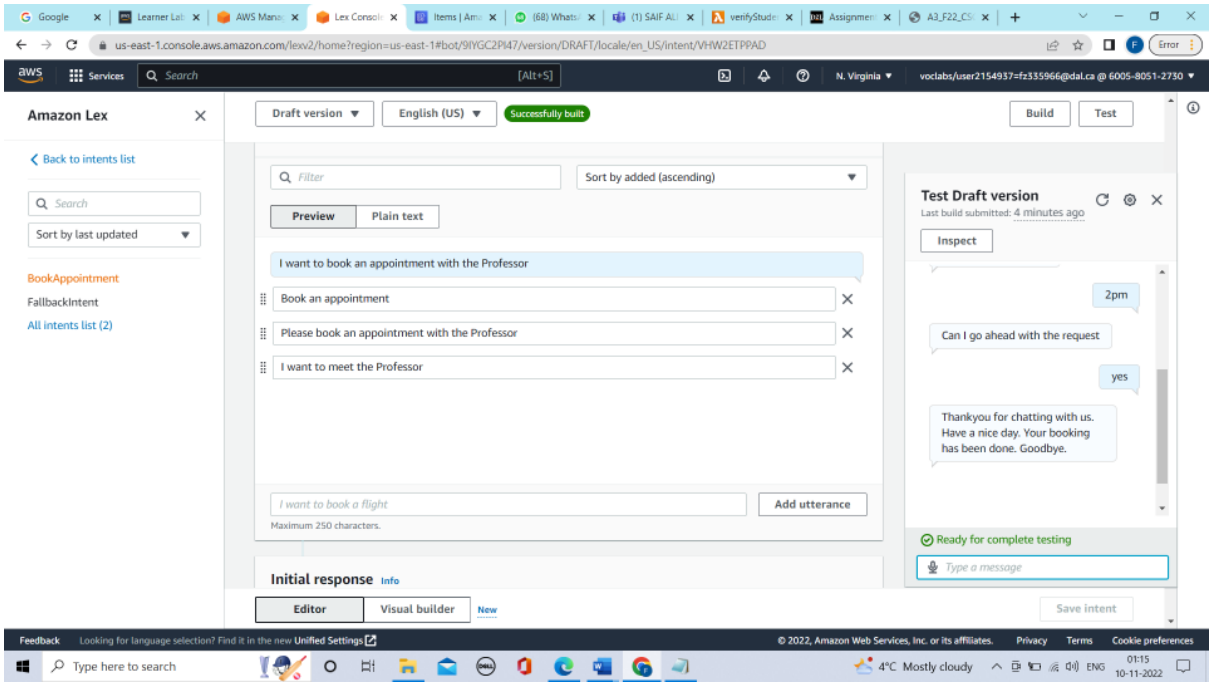

Fig 77: Amazon Lex – test ChatBot page

Test case 3:

- When the user provides time not in Amazon time format (Refer fig. 78) i.e. user does not mention pm and am which prompts the ChatBot to ask the user to enter the date again as shown in the fig. 79.
- The ChatBot does not proceed until the date and time is provided in Amazon format (Refer 79 & 80).

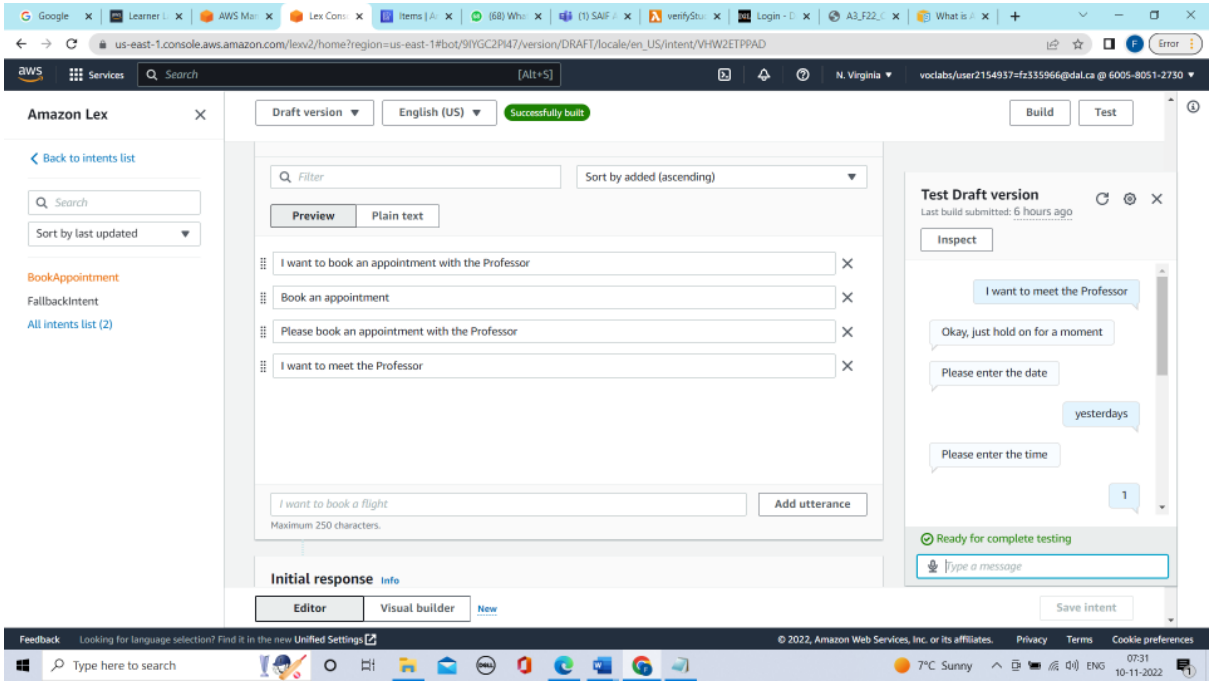

Fig 78: Amazon Lex – test ChatBot page

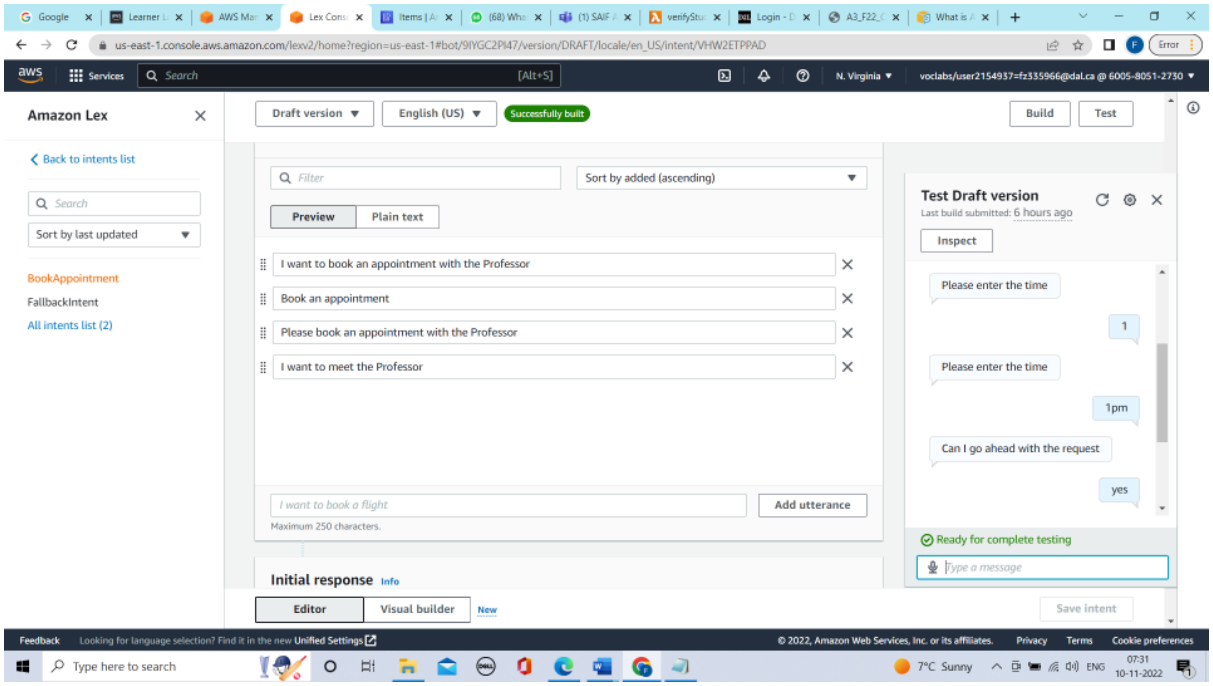

Fig 79: Amazon Lex – test ChatBot page

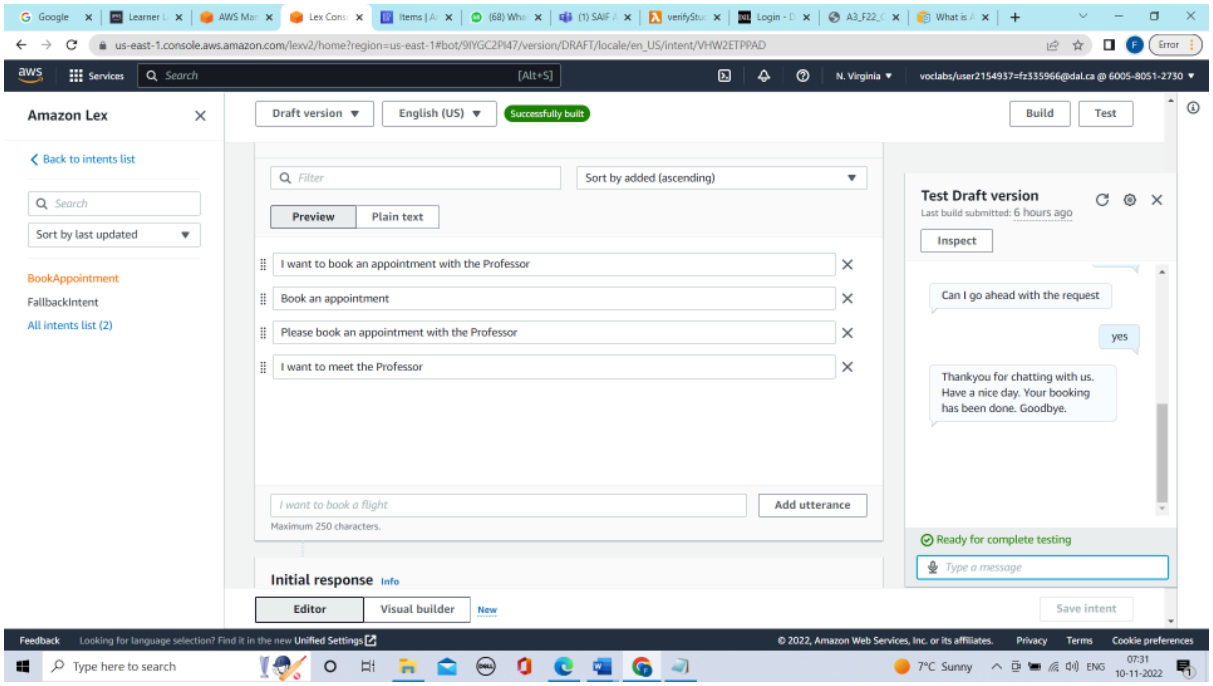

Fig 80: Amazon Lex – test ChatBot page

Test case 4:

- When the date and time provided is correct as shown in the fig. 81 & 82.
- However, when the ChatBot asks to proceed with the request, the user declines. And the request will not be submitted which means the appointment with the professor will not be booked as shown in the fig. 82.

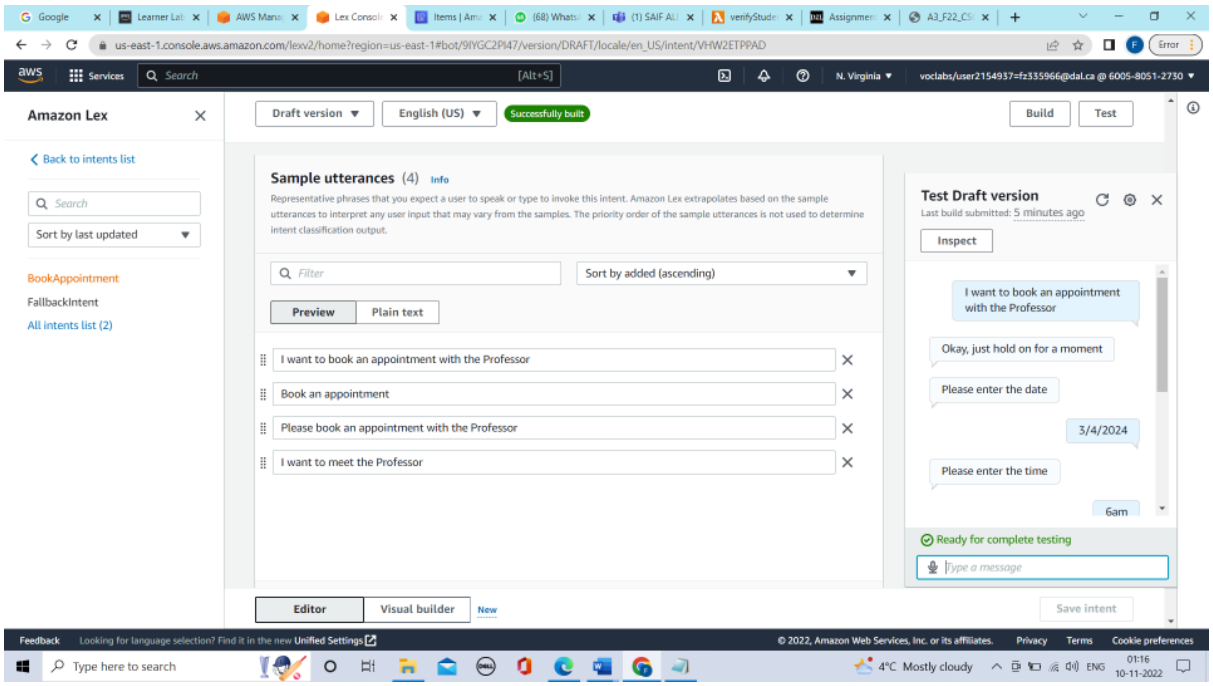

Fig 81: Amazon Lex – test ChatBot page

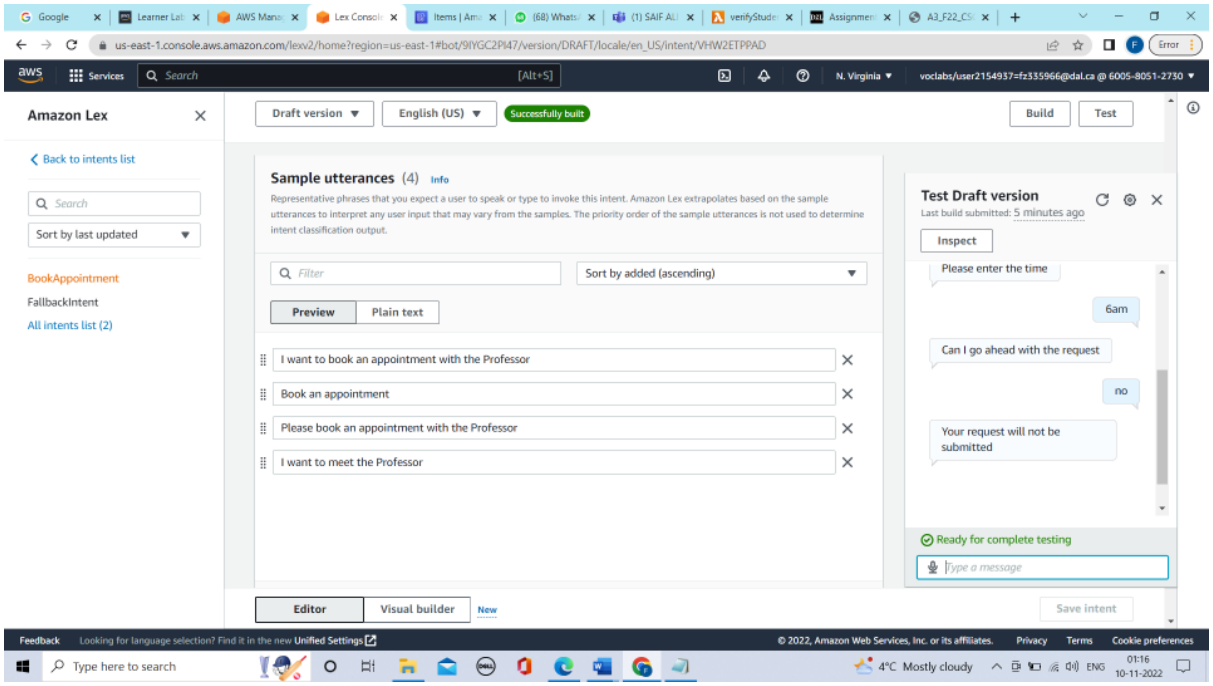

Fig 82: Amazon Lex – test ChatBot page

Test case 5:

- User enters correct date and time to book an appointment as shown in the fig. 83.
- But the user does not proceed with the request. So, the appointment with the Professor is not made (Refer fig. 84).

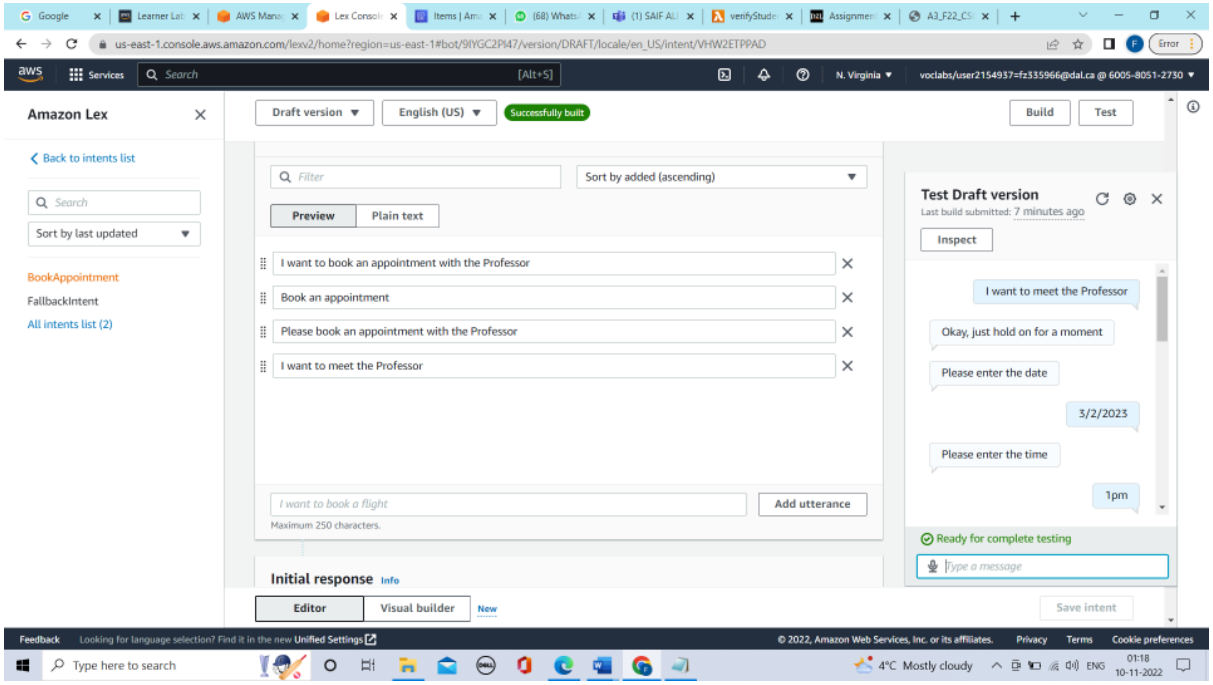

Fig 83: Amazon Lex – test ChatBot page

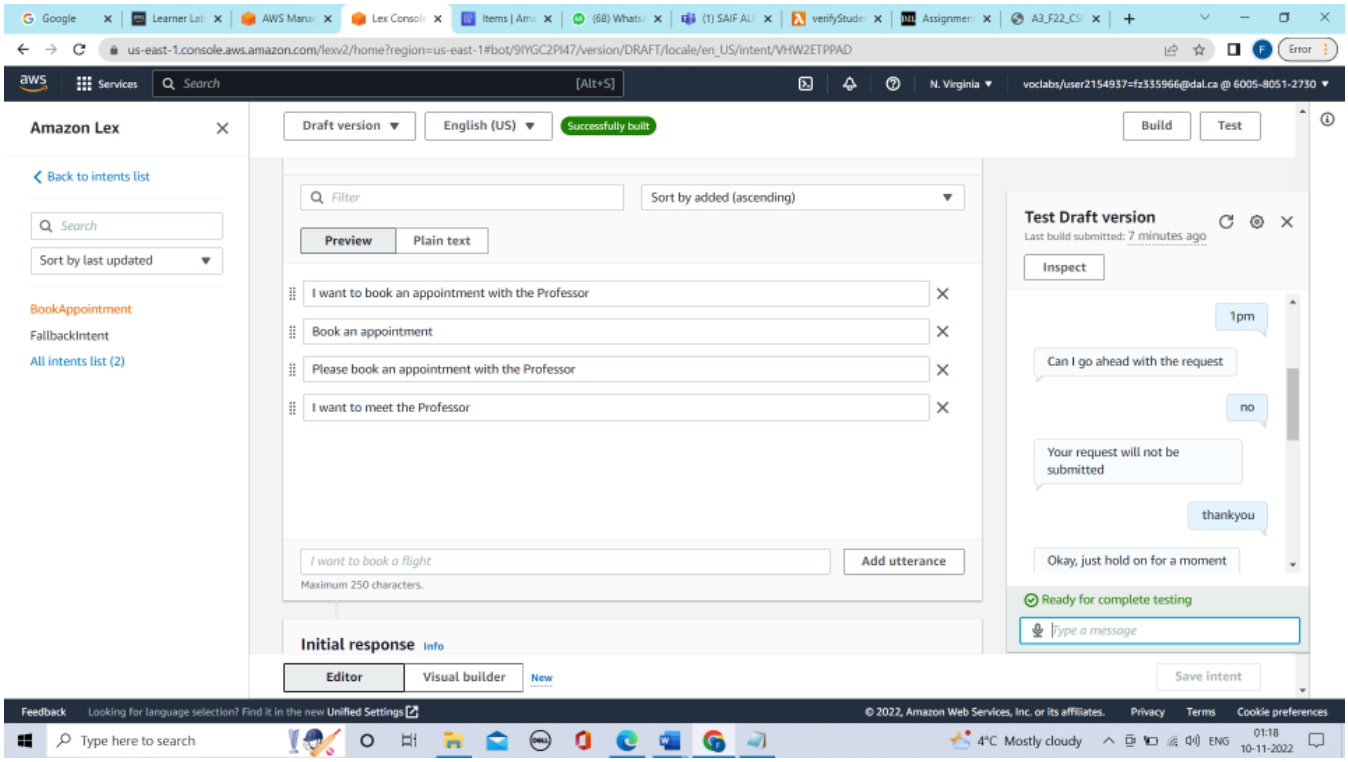

Fig 84: Amazon Lex – test ChatBot page

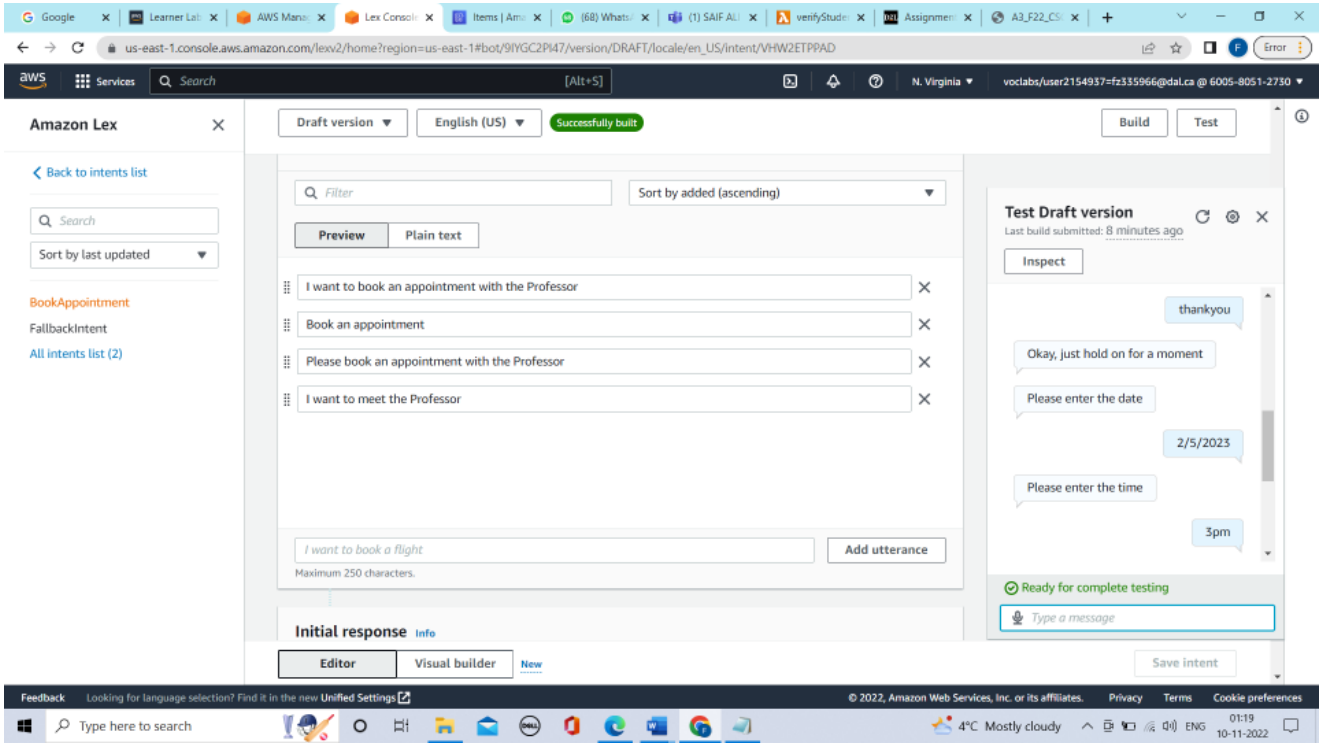

Fig 85: Amazon Lex – test ChatBot page

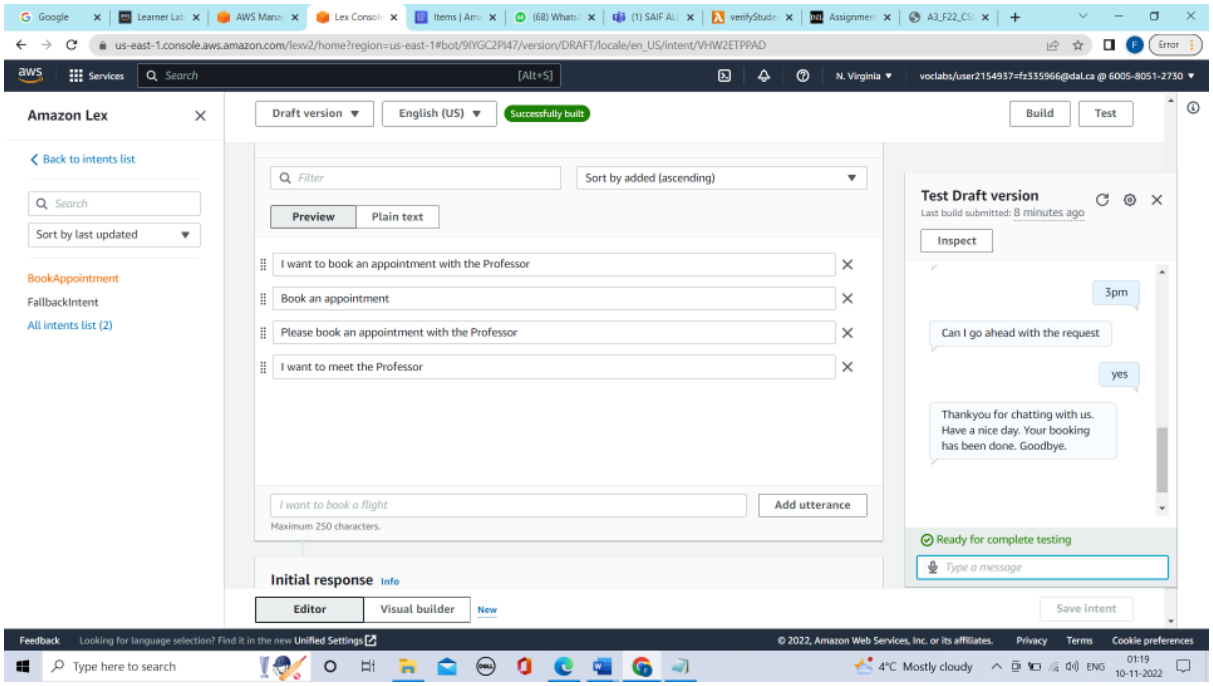

Fig 86: Amazon Lex – test ChatBot page

- The user mentioned "Thankyou", but the ChatBot prompted to enter date and time incase user wants to book the appointment again (Refer fig. 85 & 86).
- Another case is when the user mentions "Thankyou" after booking an appointment. This also makes ChatBot prompt user to enter date and time incase user wants to book the appointment again (Refer 87, 88 & 89).
- Fig. 88 also shows that the user can't proceed with the booking until he/she enters the correct format for date and time.

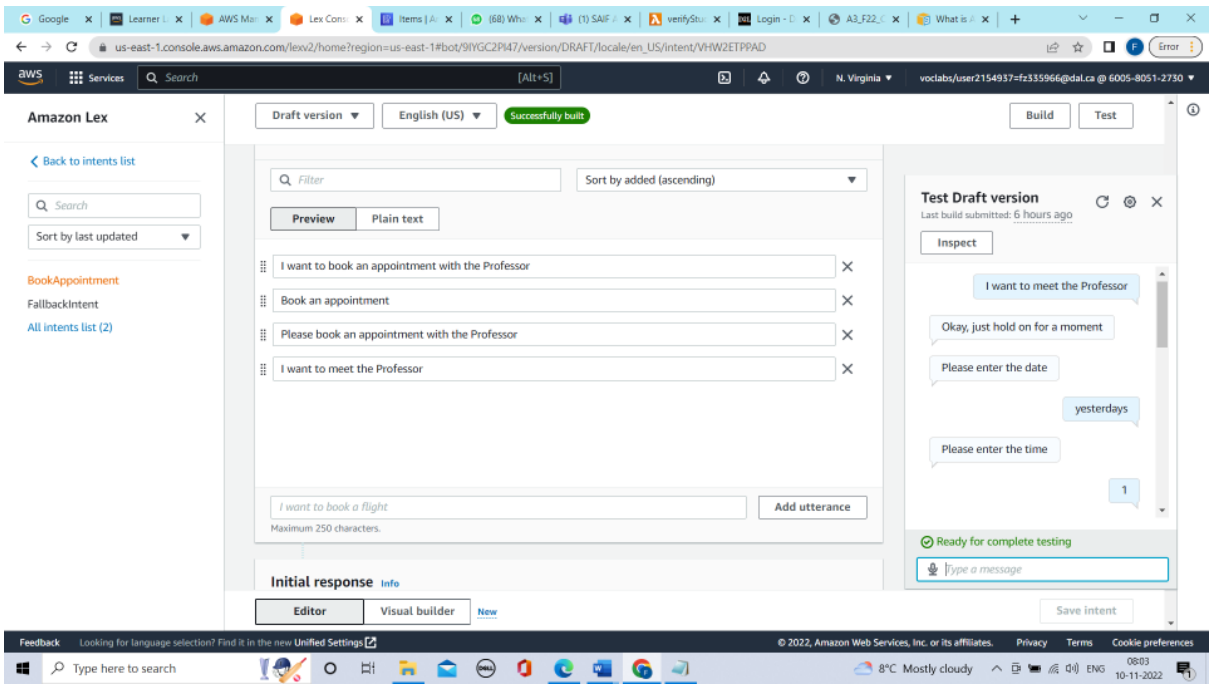

Fig 87: Amazon Lex – test ChatBot page

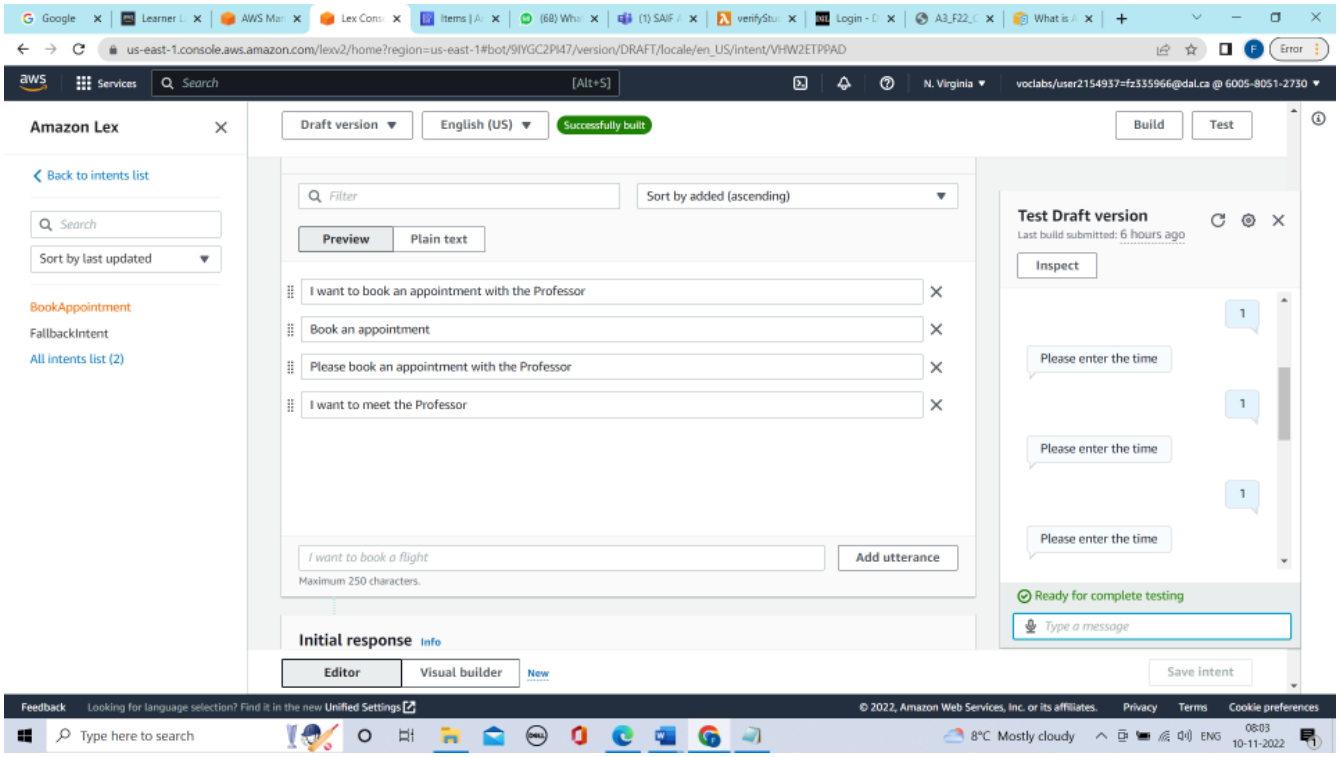

Fig 88: Amazon Lex – test ChatBot page

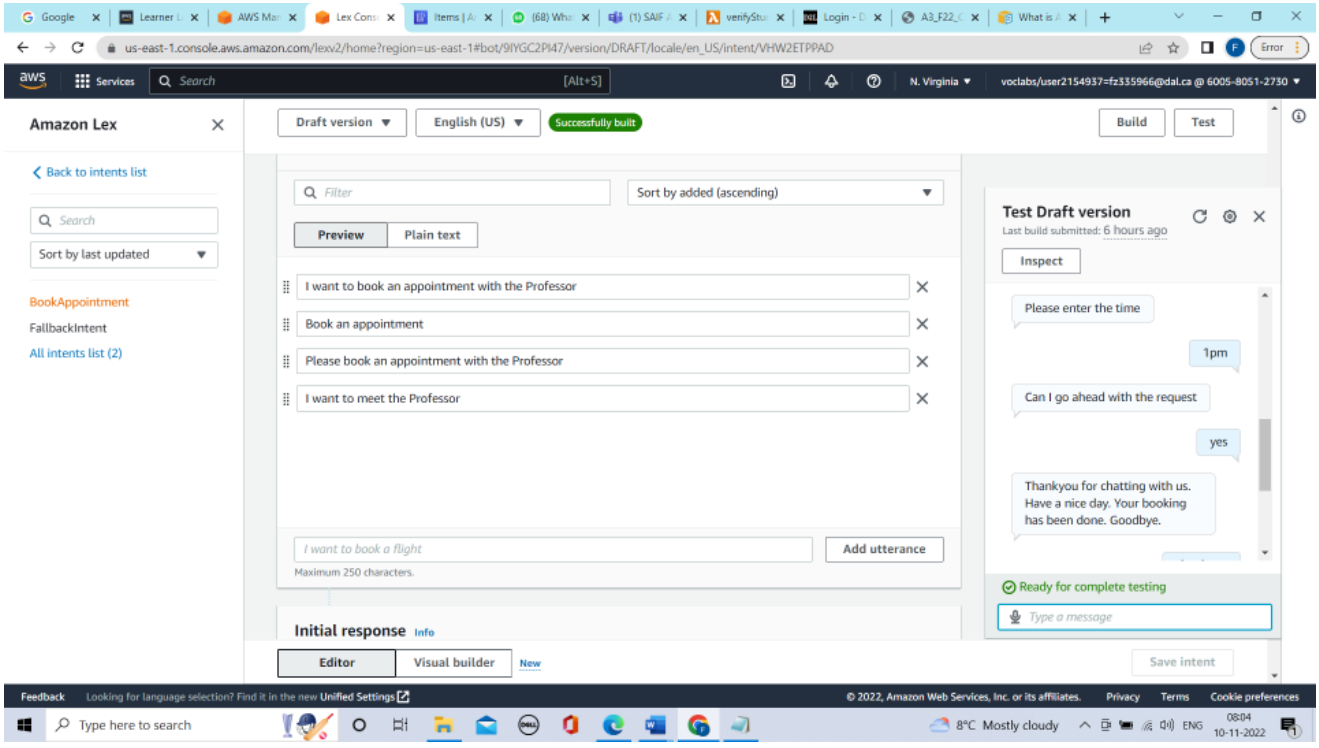

Fig 89: Amazon Lex – test ChatBot page

Program Script:

lambda_function.py

```
import json
import boto3
client = boto3.client('dynamodb')
result = {}
intent=""
studentName=""
studentEmail=""
slots=""

def lambda_handler(event, context):
    global intent

    intent=event['sessionState']['intent']['name']
    global slots

    if event['invocationSource'] == 'FulfillmentCodeHook':
        slots=event['sessionState']['intent']['slots']
        studentEmail=slots['Email']['value']['originalValue']
        print(studentEmail)
        studentName=slots['Name']['value']['originalValue']
        print(studentName)
        if intent=='StudentVerification':
            verifyStudentInformation(studentEmail,studentName)
            print(result)
            return result

def verifyStudentInformation(emailFromBot,nameFromBot):
    responseFromDB = client.get_item(
            TableName='StudentDetails',
            Key={
                'student_email': {
                    'S': emailFromBot,
                }
            }
            )
    key='Item'
    value = key in responseFromDB.keys()
    if value:
        studentNameDB=responseFromDB['Item']['student_name']['S']
        studentEmailDB=responseFromDB['Item']['student_email']['S']

        message=""
        if (studentEmailDB.__eq__(emailFromBot) and
studentNameDB.__eq__(nameFromBot)):
            message = "Your details are verified"
```

```
        else:
            print("inside else of match if ")
            message = "Your details couldn't be verified: name did not match with
the information provided"
        global result
    # Result to lex bot
        result = {
                    "sessionState": {
                        "dialogAction": {
                            "type": "Close"
                        },
                        "intent": {
                            'name':intent,
                            'slots': slots,
                            'state':'Fulfilled'
                        }
                    },
                    "messages": [
                        {
                            "contentType": "PlainText",
                            "content": message
                        }
                    ]
                }
        return result
    else:
        message="Sorry I can't find your details in our records : email does not
exists in our database"
        result =  {
                    "sessionState": {
                        "dialogAction": {
                            "type": "Close"
                        },
                        "intent": {
                            'name':intent,
                            'slots': slots,
                            'state':'Fulfilled'
                        }
                    },
                    "messages": [
                        {
                            "contentType": "PlainText",
                            "content": message
                        }
                    ]
                }
        return result
```

```
# Code Reference:
#[1]https://github.com/venkateshkodumuri/Lex_Chatbot_to_fetch_data_from_dynamodb/blob/main/lambda_function.py
#[2]https://github.com/PradipNichite/Youtube-Tutorials/blob/main/Amazon_Lex/Part2.py
```

Code Reference:

[1] *Amazon.com.* [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/gs2-create-bot-create.html. [Accessed: 08-Nov-2022].

[2] *Amazon.com.* [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/gs2-create-bot-intent.html. [Accessed: 08-Nov-2022].

[3] *Amazon.com.* [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/gs2-create-bot-slot-types.html. [Accessed: 08-Nov-2022].

[4] *Amazon.com.* [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/gs2-create-bot-configure-intent.html. [Accessed: 08-Nov-2022].

[5] *Amazon.com.* [Online]. Available: https://docs.aws.amazon.com/lex/latest/dg/gs2-build-and-test.html. [Accessed: 08-Nov-2022].

[6] Amazon.com. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SampleData.CreateTables.html. [Accessed: 08-Nov-2022].

[7] Amazon.com. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Accessed: 08-Nov-2022].

[8] lambda_function.py at main · venkateshkodumuri/Lex_Chatbot_to_fetch_data_from_dynamod.

[9] Nichite, P. (n.d.). *Part2.Py at main · PradipNichite/YouTube-Tutorials*.

ASSIGNMENT - 3 PART B

TITLE: Quantifying Permissiveness of Access Control Policies.

Reference:

W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince and T. Bultan, "Quantifying Permissiveness of Access Control Policies," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 1805-1817, doi: 10.1145/3510003.3510233.

Summary:

More and more people rely on software services today, making it crucial to secure confidential data and uphold privacy [page 1805]. Without any verification, this information can be saved by utilising cloud services with the aid of access control specification languages [page 1805]. This forces programmers to create complex policies filled with errors that have serious implications, such as disclosing sensitive information like names, addresses, and credit card pins [page 1805]. AWS Simple Storage Service (S3), which is publicly available, contains this information [page 1805]. By creating automated verification tools to evaluate access control regulations, sensitive data must be protected immediately [page 1805]. The motivation of this study is to describe how permissive the policy is and to quantify how permissive other access control policies are in comparison [page 1805]. Users are assisted in securing their services via access control policies by services like Microsoft Azure and Google Cloud Platform (GCP) [page 1805].

The study suggests measuring permissiveness in terms of the number of requests and actions that the policy permits [page 1807]. Based on the set of recognised resources and valid AWS S3 actions, the outcome is presented [page 1807]. AWS users may desire a combination of public and private access to their data, which can result in complex policies that cannot be implemented without a thorough understanding of AWS policy [page 1808]. It is challenging to create a policy that can match the user's goal [page 1808]. However, the number of users allows us to assess how permissive a policy is [page 1808]. By comparing the policy semantics with the user's intents, this enables users to validate [page 1808]. This has the effect of granting access to a particular group of people [page 1807]. We can obtain the userId count by adding only the known userIds to the constraints [page 1808]. Therefore, the quantitative analysis can be used to validate the number of permissions if an AWS user wants to establish a policy with a specific number of permissions [page 1808]. Using the verification and validation technique, the policy model is intended to produce complicated policy specifications that can be efficiently examined [page 1808]. If a request is approved under one regulation but rejected by another, the policy (P) grants, indicating that the request was ultimately turned down [page 1808]. The permissiveness increases with the amount of the request [page 1808]. By introducing the variables, the SMT encoding is applied to encode the set of possible requests [page 1809]. By evaluating how permissive a particular policy or related policies are, we can quantify the policy [page 1809]. More requests are likely to be granted if the probability is higher, whereas fewer requests are likely to be granted if the probability is lower [page 1809]. The likelihood of 0 denotes a policy that rejects all requests, while the probability of 1 denotes a policy that grants all requests [page 1809].

In order to provide compact encoding constraints for policy acts, the heuristic approach transforms the set of equality and inequality constraints [page 1810]. The valid actions (V) are reduced to a considerably smaller set of range constraints to achieve this [page 1810]. Elastic Compute Cloud (EC2), Identity Access Management (IAM), and Simple Storage Service (S3) are the three services that we take into consideration [page 1811]. The first level is the constraint of activities to a set of actions, and the second is the constraint of resources and actions in relation to one another [page 1811]. This makes it easier to specify the kinds of resources that each action can work with [page 1811]. Based on this methodology, an open-source programme called QUACKY may be used to translate policies into SMT formulae and then send the formulas to a model counting constraint solver to quantify permissiveness or relative permissiveness [page 1811]. The experiment's objective is to assess the QUACKY's performance using a heuristic methodology [page 1813]. Following evaluation, we can compare policies with and without type constraints [page 1813]. It was shown that the constraint transformation heuristic reduced minimum times by between 16% and 76% for S3 and EC2, and maximum durations by between 75% and 96% for S3 and EC2 [page 1813]. The average revealed a drop of between 78% and 92% for S3 and EC2, respectively [page 1813]. Time and permissiveness are traded off in the Microsoft Azure Policies [page 1815]. LoginAdmin is more permissive to specific requests than LoginUser [page 1815].

Constraints slow down the procedure for each AWS service, but they also boost permissiveness's effects by drastically reducing the number of requests that may be made [page 1813]. This is because type constraints limit the range of possible actions and constrain actions to act only on particular resource types [page 1813]. QUACKY overestimates the permissiveness of a policy in the absence of type constraints to model the semantics of the policy language [page 1813]. To determine whether a given policy is permissive, our method uses model counting constraint solvers [page 1815]. This method was applied to AWS policies, and its efficacy was experimentally assessed using AWS policies that we had gathered from online discussion boards [page 1815]. Our findings show that our quantitative permissiveness analysis methodology may be used in real-world settings [page 1815]. Future research will examine the applicability of quantitative analysis methods to additional policy issues, such as policy repair [page 1815].

My Views:

On compute clouds, a variety of software services are used [page 1805]. Popular cloud service providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), which protect user information by formulating access control policies [page 1805]. These policies grant authorised access and can be written in a variety of access control specification languages, such as eXtensible Access Control Markup Language (XACML) [2] or AWS Identity and Access Management (IAM) [1] [page 1805]. There are many other helpful languages, but they risk errors and unwanted access to cloud data without validation and verification [page 1805]. To ensure that a policy is correct, it is therefore required to design an automated verification technique that can assess access control policies for cloud computing [page 1805]. When there is a discrepancy between a specification and a policy, it is not always the case that the policy is incorrect; the specification may also be incorrect [page 1805].

AWS establishes a policy language in which policies use declarative statements to either grant or deny access [page 1806]. A statement is a 5-tuple, i.e., principal, effect, action, resource, and condition [page 1806]. A conditional operator, a condition key, and a condition value make up a condition [page 1806]. The security of the user's data in the cloud is ensured by this formal model [page 1806]. The policies are straightforward for typical users, but this may not always be the case [page 1807]. The written policies may be difficult to understand and complicated [page 1807]. In that scenario, one may take into account the policies gathered from the forums to assess their applicability [page 1807]. Alphanumeric, ':', '-', '_', and '/' characters are permitted as field values in the policy [page 1807].

The paper discusses an automated method for calculating the permissiveness of access control policies by converting the policies to SMT formulas and then utilising a model counting constraint solver [page 1806]. Additionally, it suggests expanding the formal model and using an automated method to compare the relative permissiveness of different policies [page 1806]. For the purpose of enhancing model counting performance, a heuristic approach was used to modify formulas taken from policies [page 1806]. Additionally, there is an open-source application called QUACKY that uses an automated method to assess policies published in the Azure and AWS Identity and Access Management (IAM) policy languages [page 1806]. A publicly accessible policy dataset that includes hundreds of policies created by applying mutation techniques to dozens of real-world policies that were obtained from the AWS forums and the Azure documentation [page 1806]. On the dataset, QUACKY was experimentally evaluated [page 1806]. In order to determine which policy is more permissive than others, it is crucial to quantify the permissiveness of policies using a model counting constraint solver [page 1805]. This would not only lessen the issue of data protection but also enhance the performance of the model counts when utilising the heuristic approach [page 1805].

Reference:

[1] iam [n.d.]. AWS IAM Policy Language. http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html.

[2] XACML 2003. eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Standard. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.