PRACTICAL: 5

AIM:

Scenario:

The café is eager to launch a dynamic version of their website so that the website can access data stored in a database. Sofía has been making steady progress toward this goal. In a previous lab, you played the role of Sofía and created a DynamoDB database. The database table contains café menu details, and an index holds menu items that are flagged as specials. Then, in another lab, you created an API to add the ability for the website to receive mock data through REST API calls. In this lab, you will again play the role of Sofía. You will replace the mock endpoints with functional endpoints so that the web application can connect to the database. You will use Lambda to bridge the connection between the GET APIs and the data stored in DynamoDB. Finally, for the POST API call, Lambda will return an updated acknowledgment message.

Lab overview and objectives:

In this lab, you will use the AWS SDK for Python (boto3) to create AWS Lambda functions. Calls to the REST API that you created in the earlier Amazon API Gateway lab will initiate the functions. One of the Lambda functions will perform either an Amazon DynamoDB database table scan or an index scan. Another Lambda function will return a standard acknowledgment message that you will enhance later in a lab where implement Amazon Cognito.

After completing this lab, you should be able to:

- Create a Lambda function that queries a DynamoDB database table. · Grant sufficient permissions to a Lambda function so that it can read data from DynamoDB.
- Configure REST API methods to invoke Lambda functions using Amazon API Gateway.

THEORY:

This lab focuses on integrating AWS services—DynamoDB, Lambda, and API Gateway—to create a dynamic café website that can retrieve and display menu details stored in a database. **Amazon DynamoDB** serves as the backend database, storing menu items with attributes like name, description, price, and a flag for specials. A secondary index is used to efficiently query special menu items. **AWS Lambda** acts as the intermediary, running serverless functions to handle API calls and connect to DynamoDB. One Lambda function performs a database table scan or index scan to fetch menu data, while another function returns acknowledgment messages for POST requests.

Amazon API Gateway facilitates the communication between the web application and Lambda by exposing REST API endpoints. To achieve this, API methods (GET and POST) are configured to invoke the respective Lambda functions. The Lambda functions are granted **IAM permissions** to read data from DynamoDB. This architecture ensures a seamless, dynamic interaction between the web

application and the database, providing real-time data retrieval for menu items and a scalable solution for future enhancements like user authentication.

CODE:

update_config.py

```
import boto3
S3API = boto3.client("s3", region_name="us-east-1")
bucket_name = "c144426a3734344l9003220t1w816396249192-s3bucket-
dxyfssytdezp"

filename = "/home/ec2-user/environment/resources/website/config.js"
S3API.upload_file(filename, bucket_name, "config.js", ExtraArgs={'ContentType':
"application/js", "CacheControl": "max-age=0"})


print ("DONE")
```

get_all_products_code.py

```
import boto3, json
from boto3.dynamodb.conditions import Key
from boto3.dynamodb.conditions import Key, Attr, Not

TABLE_NAME_STR = 'FoodProducts'
INDEX_NAME_STR = 'special_GSI'
DDB = boto3.resource('dynamodb', region_name='us-east-1')

def lambda_handler(event, context):

    offer_path_str = event.get('path')
    if offer_path_str is not None:
        return scan_index(event, context)
    else:
        pass
    print("running scan on table")

    DDB = boto3.resource('dynamodb', region_name='us-east-1')

    TABLE = DDB.Table(TABLE_NAME_STR)

    response = TABLE.scan()
```

```
    data = response['Items']

    while 'LastEvaluatedKey' in response:
        response = TABLE.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
        print("We needed to paginate and extend the response")
        data.extend(response['Items'])

    #python return non standard JSON
    #so we need a helper to convet Decimal('595') and special returned by dynamo
    #to an integer like 595

    for item in data:
      item['price_in_cents_int'] = item.pop('price_in_cents')
      if item.get('special') is not None:
        item['special_int'] = item.pop('special')
      item['tag_str_arr'] = item.pop('tags')
      item['description_str'] = item.pop('description')
      item['product_name_str'] = item.pop('product_name')
      item['product_id_str'] = item.pop('product_id')

      if item['price_in_cents_int']:
          item['price_in_cents_int'] = int(item['price_in_cents_int'])
      if item.get('special_int') is not None:
          item['special_int'] = int(item['special_int'])

    return_me={"product_item_arr": data}

    return return_me

def scan_index(event, context):

    print("running scan on index")
    ## event and context not used
    TABLE = DDB.Table(TABLE_NAME_STR)

    response = TABLE.scan(
        IndexName=INDEX_NAME_STR,
        FilterExpression=Not(Attr("tags").contains("out of stock"))
    )

    data = response['Items']

    while 'LastEvaluatedKey' in response:
```

```
        response = TABLE.scan(
            ExclusiveStartKey=response['LastEvaluatedKey'],
            IndexName=INDEX_NAME_STR,
            FilterExpression=Not(Attr("tags").contains("out of stock"))
        )
        print("We needed to paginate and extend the response")
        data.extend(response['Items'])

    #python return non standard JSON
    #so we need a helper to convet Decimal('595') and special returned by dynamo
    #to an integer like 595
    for item in data:
      item['price_in_cents_int'] = item.pop('price_in_cents')
      item['special_int'] = item.pop('special')
      item['tag_str_arr'] = item.pop('tags')
      item['description_str'] = item.pop('description')
      item['product_name_str'] = item.pop('product_name')
      item['product_id_str'] = item.pop('product_id')

      if item['price_in_cents_int']:
          item['price_in_cents_int'] = int(item['price_in_cents_int'])
      if item.get('special_int') is not None:
          item['special_int'] = int(item['special_int'])

    return_me = {
        "product_item_arr": data
    }
    return return_me


#remove this line below once you have tested locally and wish to deploy
print(lambda_handler({}, None))
```

get_all_products_wrapper.py

```
import boto3
import subprocess

client = boto3.client('lambda', region_name='us-east-1')
ROLE = 'arn:aws:iam::816396249192:role/LambdaAccessToDynamoDB'
BUCKET = subprocess.getoutput('aws s3api list-buckets --query "Buckets[].Name" |
grep s3bucket | tr -d "," | xargs')

response = client.create_function(
```

```
    FunctionName='get_all_products',
    Runtime='python3.8',
    Role=ROLE,
    Handler='get_all_products_code.lambda_handler',
    Code={
        'S3Bucket': BUCKET,
        'S3Key': 'get_all_products_code.zip'
    }

)

print ("DONE")
```

create_report_wrapper.py

```
import boto3
import subprocess

client = boto3.client('lambda', region_name='us-east-1')
ROLE = 'arn:aws:iam::816396249192:role/LambdaAccessToDynamoDB'
BUCKET = subprocess.getoutput('aws s3api list-buckets --query "Buckets[].Name" |
grep s3bucket | tr -d "," | xargs')

response = client.create_function(
    FunctionName='create_report',
    Runtime='python3.8',
    Role=ROLE,
    Handler='create_report_code.lambda_handler',
    Code={
        'S3Bucket': BUCKET,
        'S3Key': 'create_report_code.zip'
    }

)

print ("DONE")
```

OUTPUT:

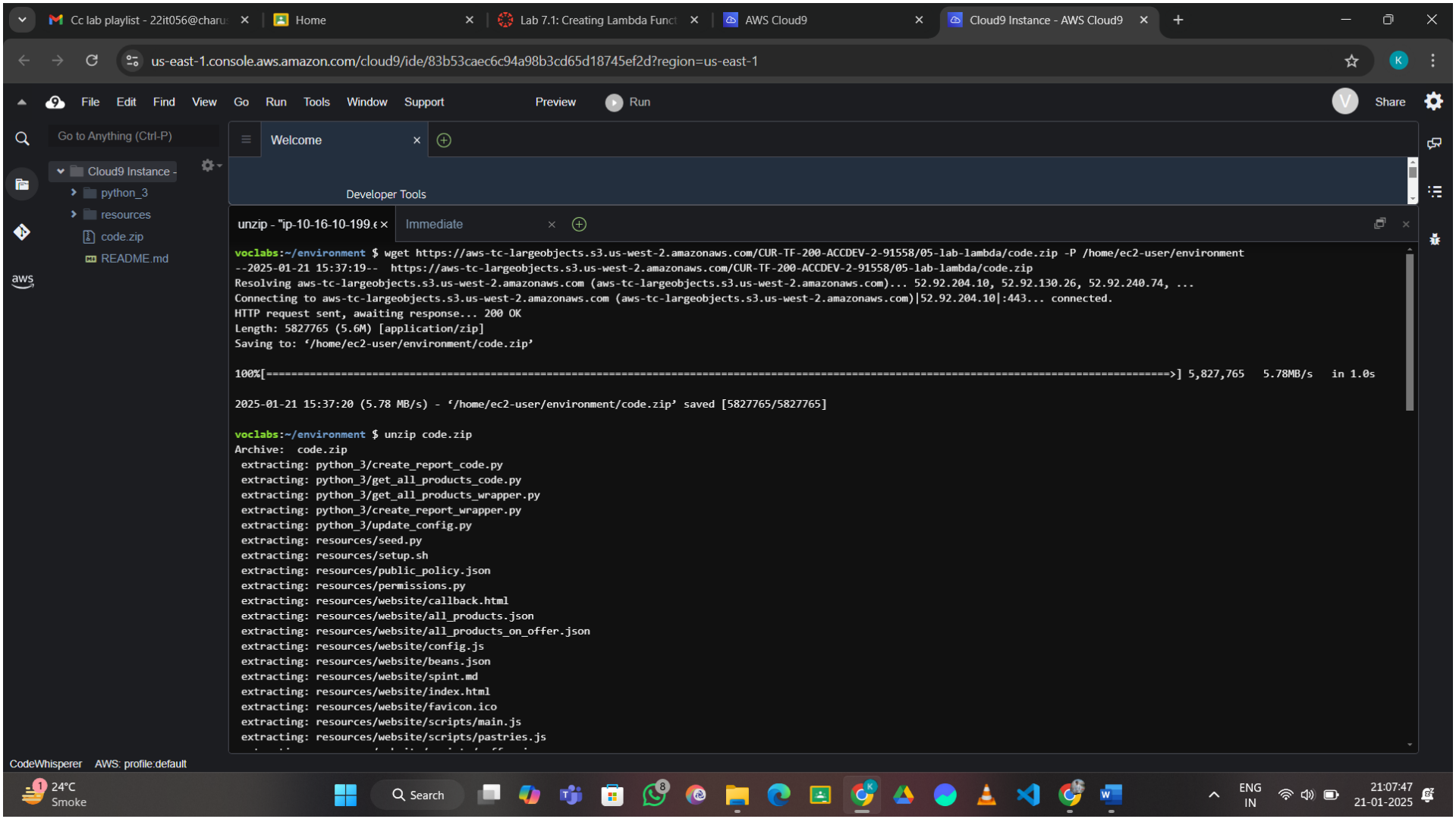

Figure 1: Download a zip file and extract it

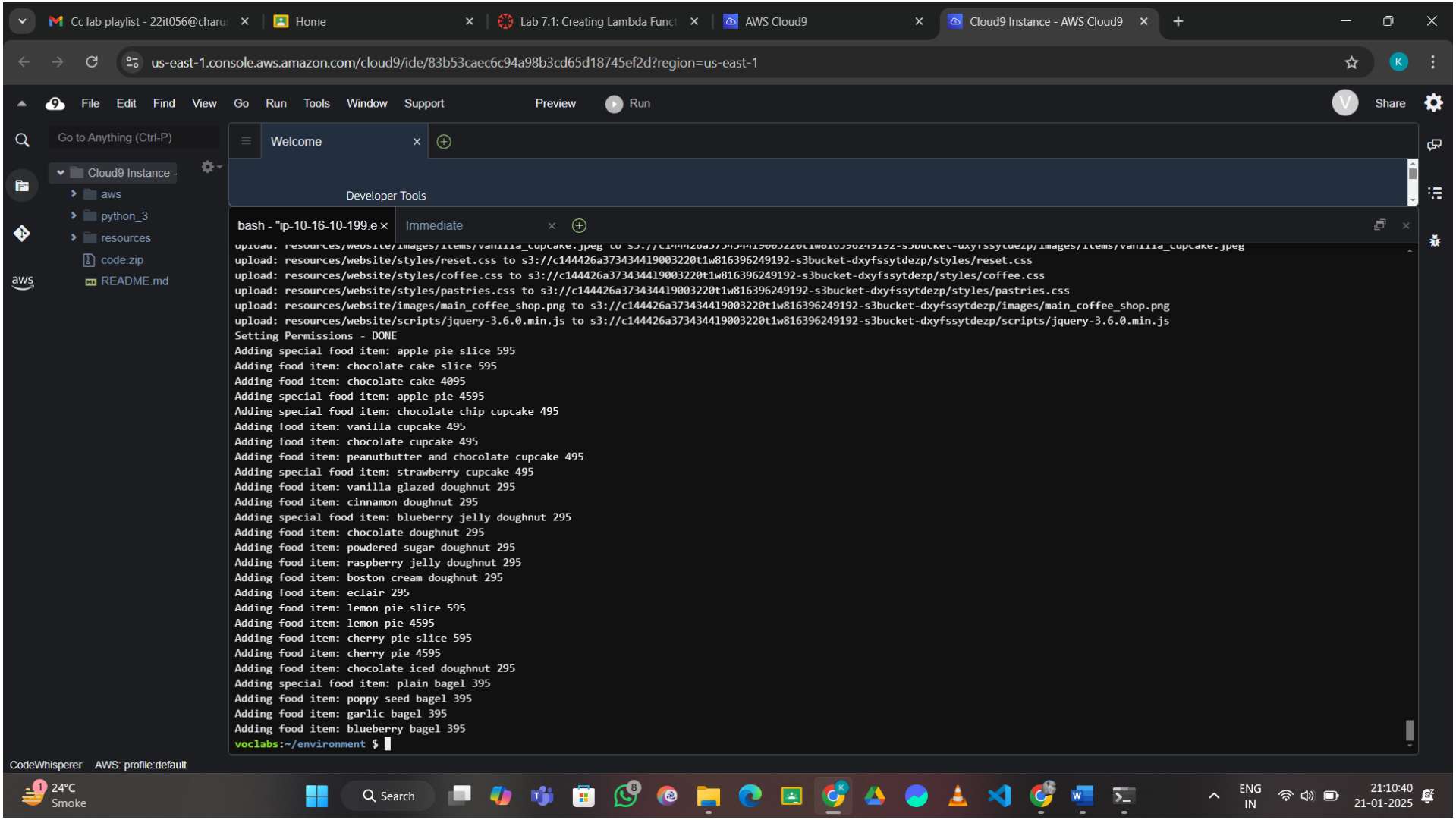

Figure 2: Set up resources with our ip address

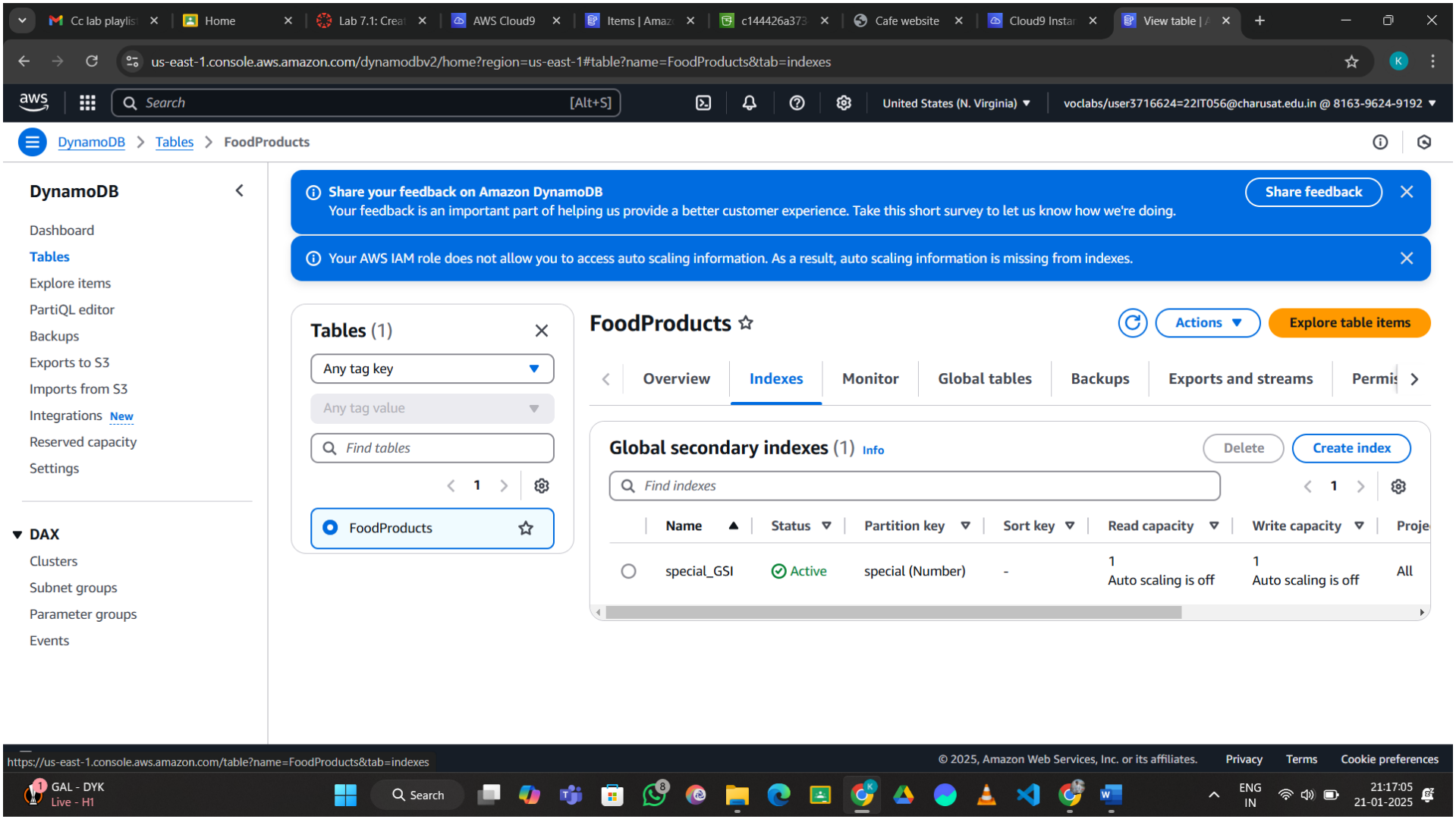

Figure 3: Check table in DynamoDB

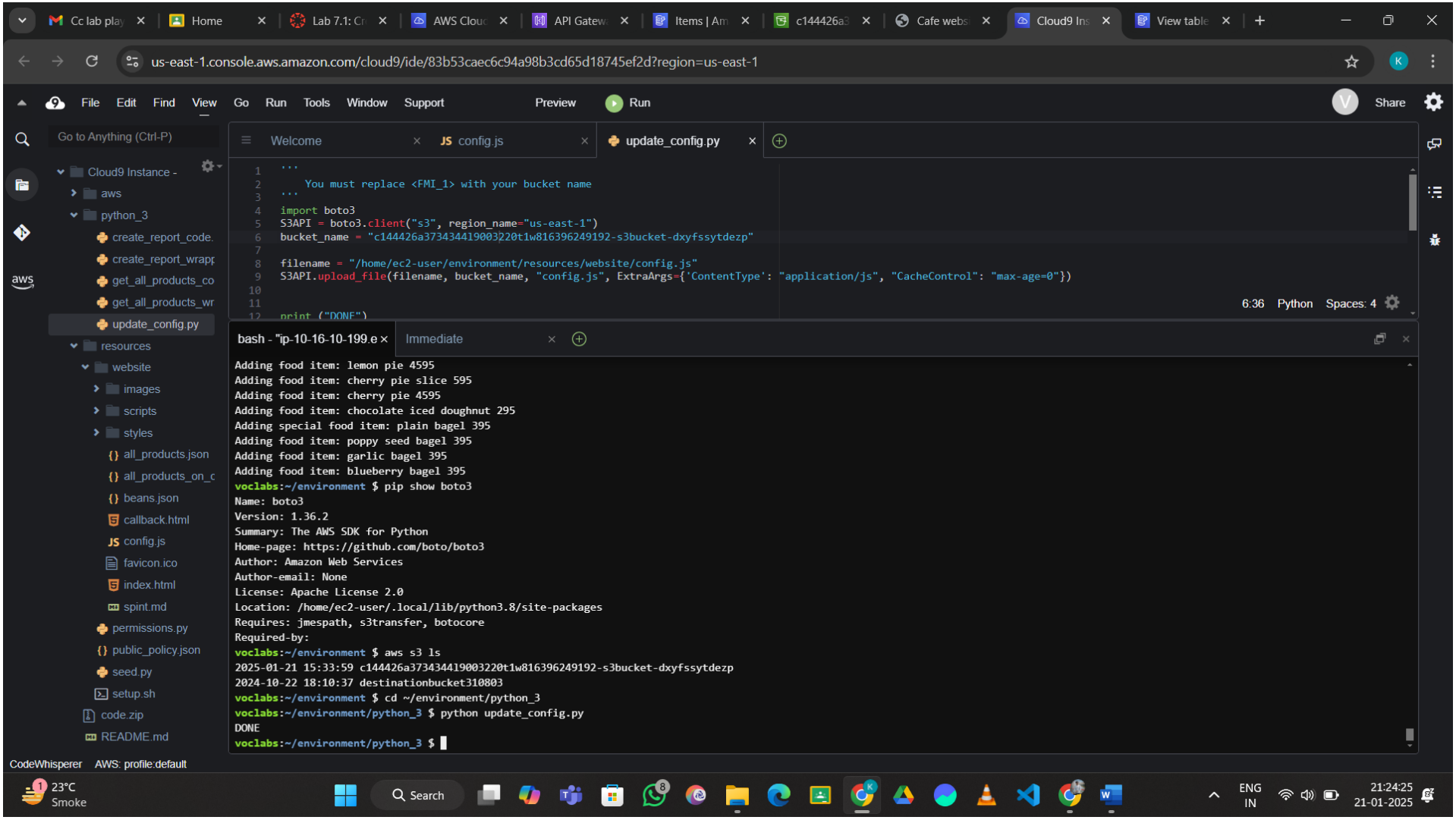

Figure 4: Update a confirmation in update_config.py file and run it

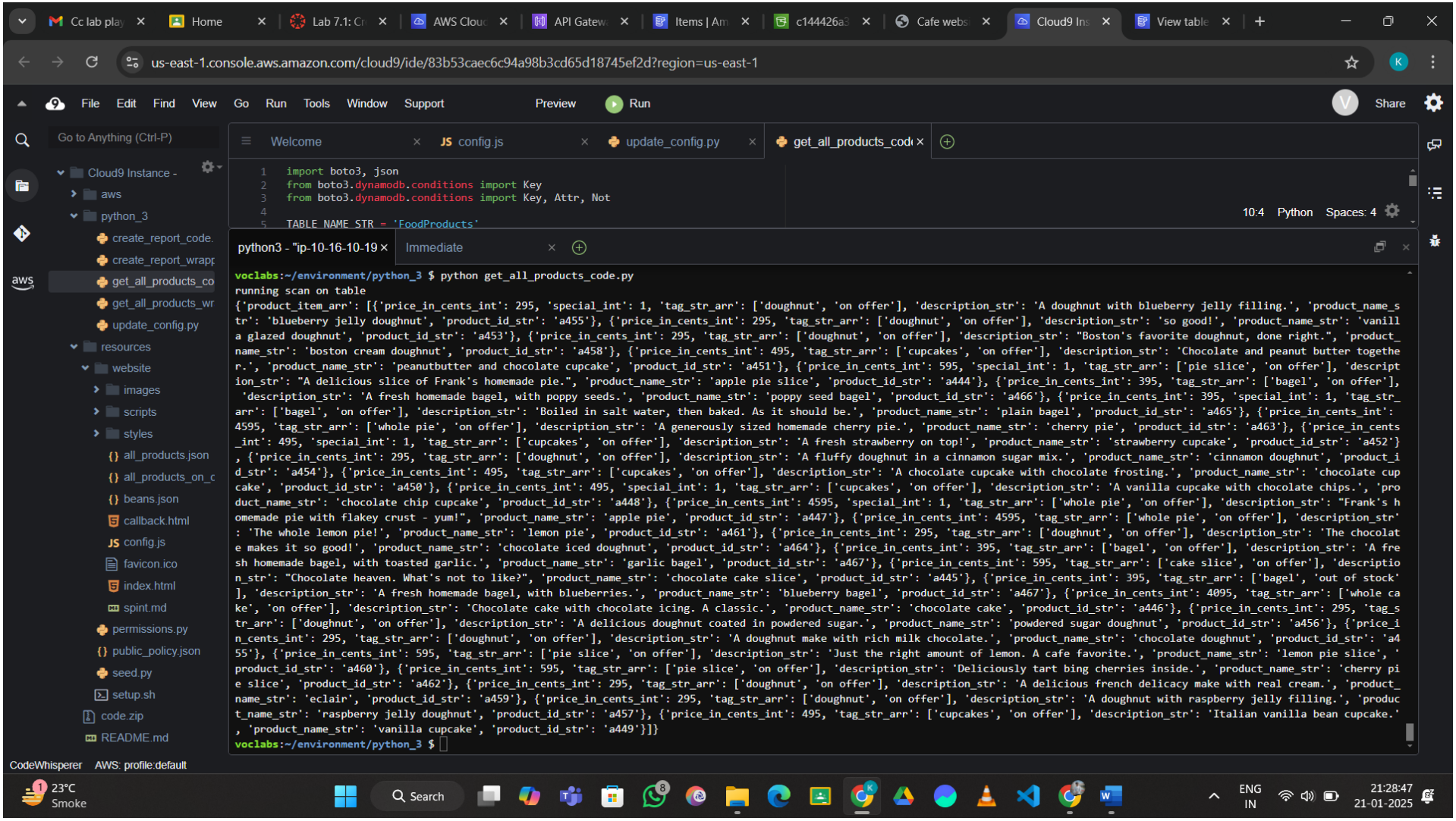

Figure 5: Update get_all_products_code.py file and run it

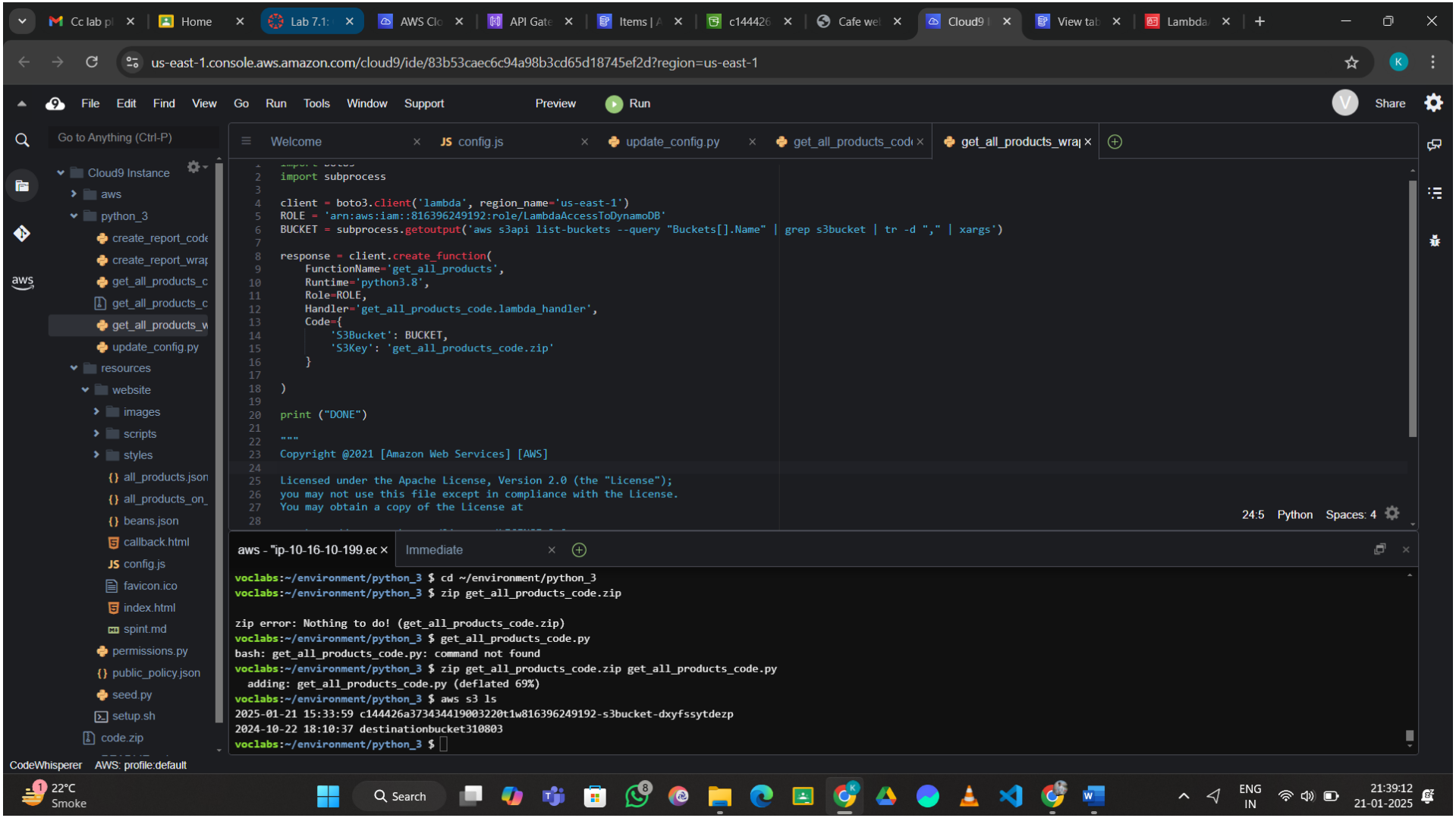

Figure 6: Download zip file in python3 directory and extract it

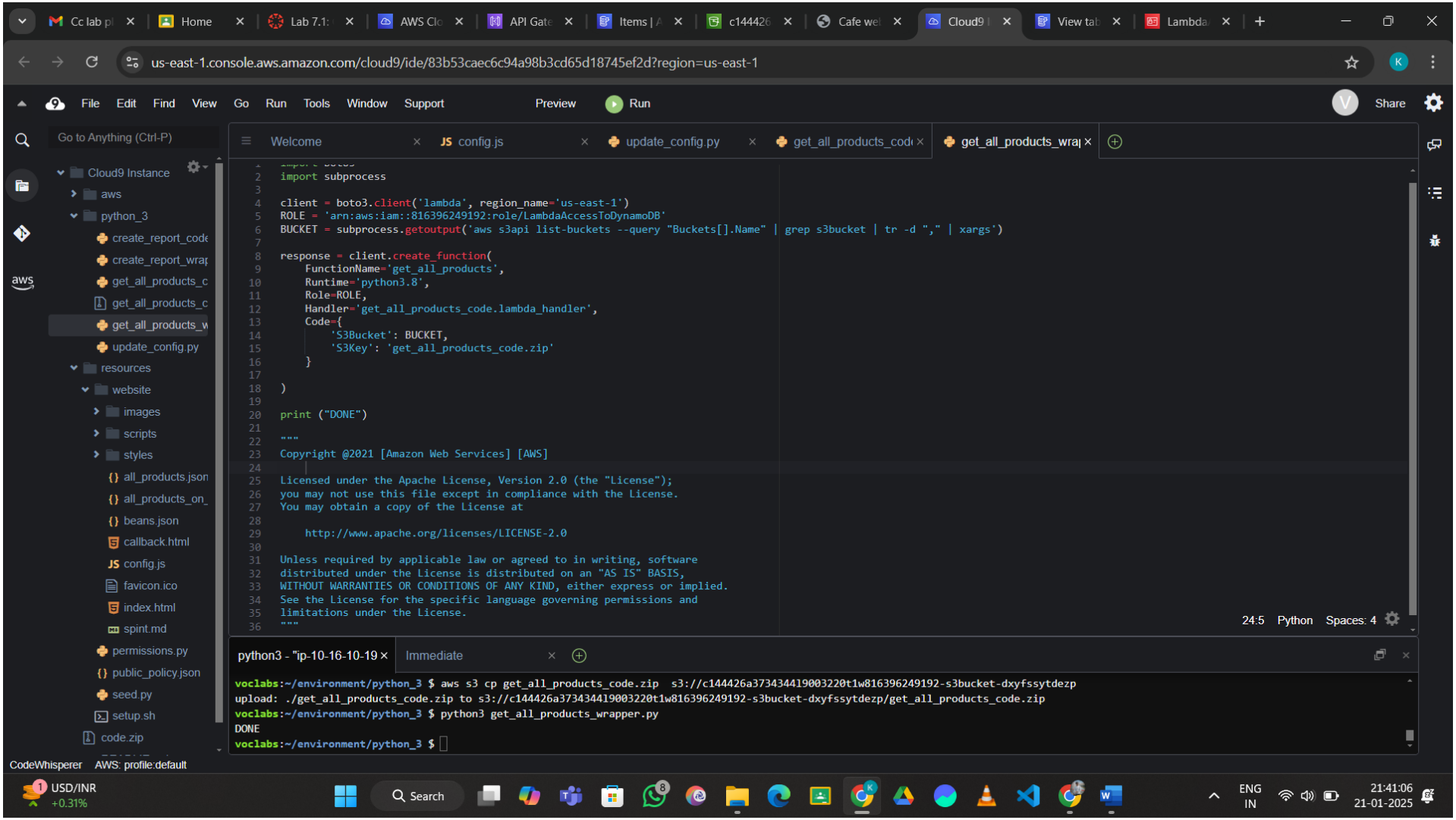

Figure 7: Update get_all_products_wrape.py file and run it

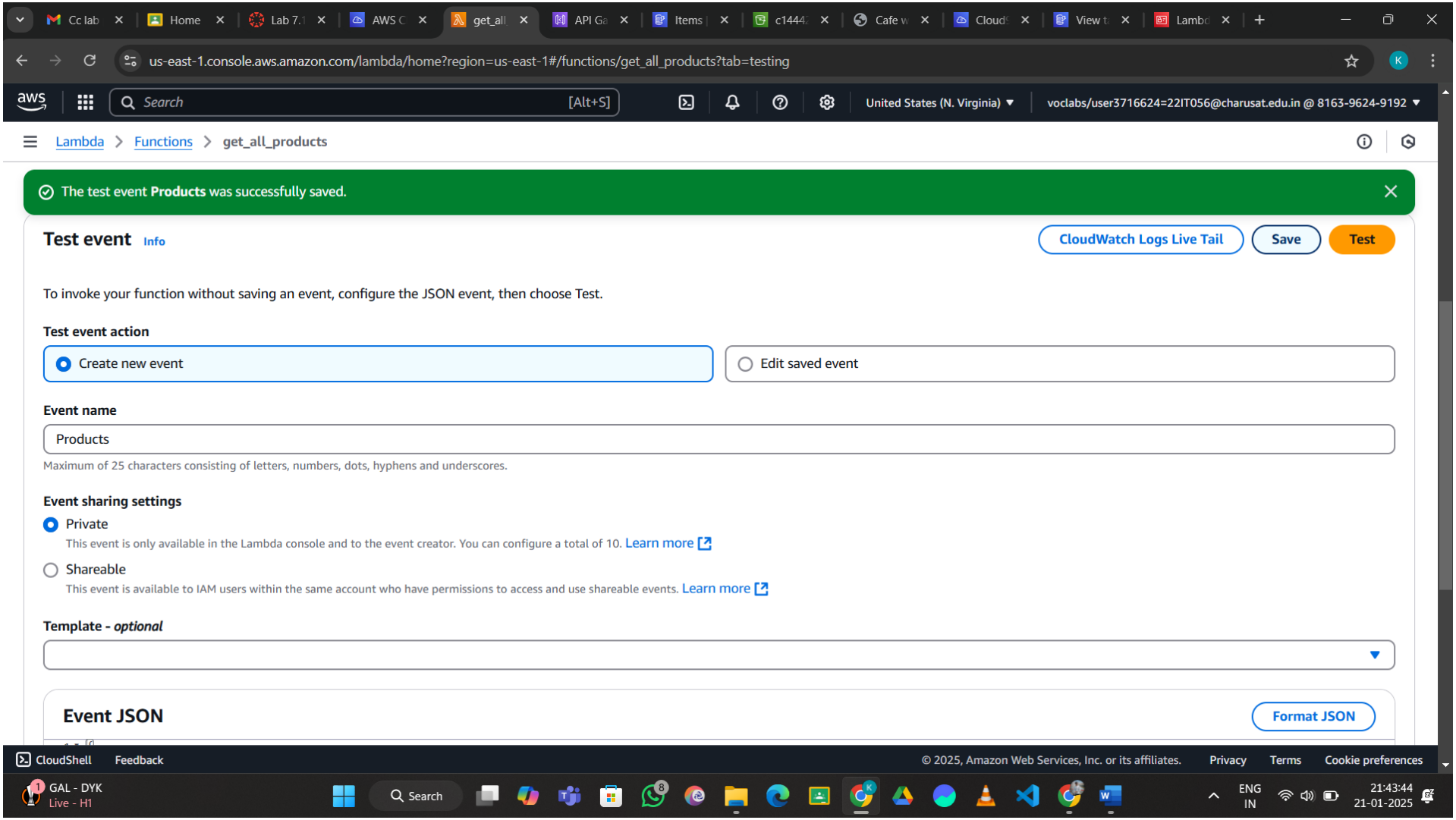

Figure 8: Test created product in lambda

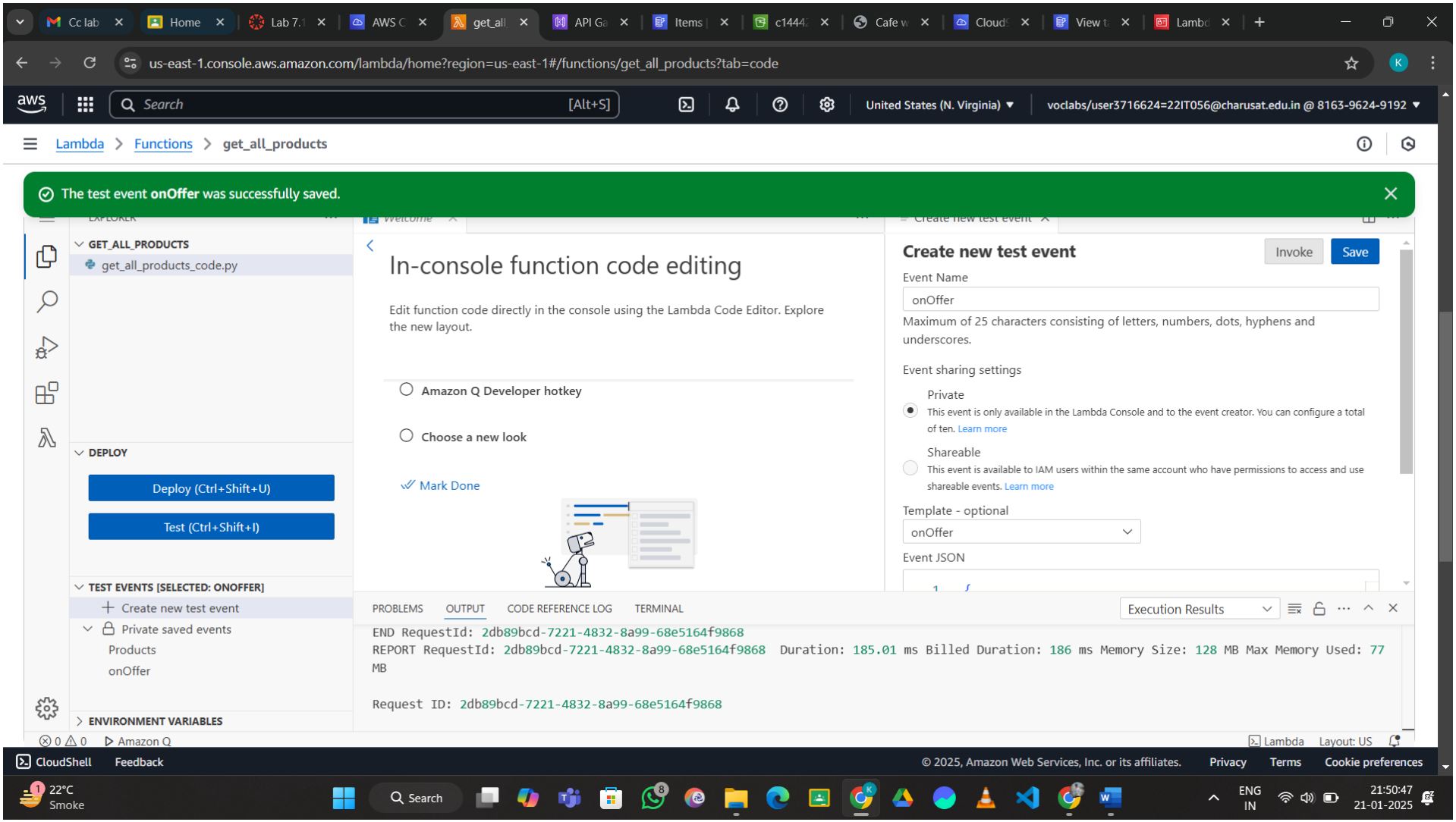

Figure 9: Create a new test onOffer for website

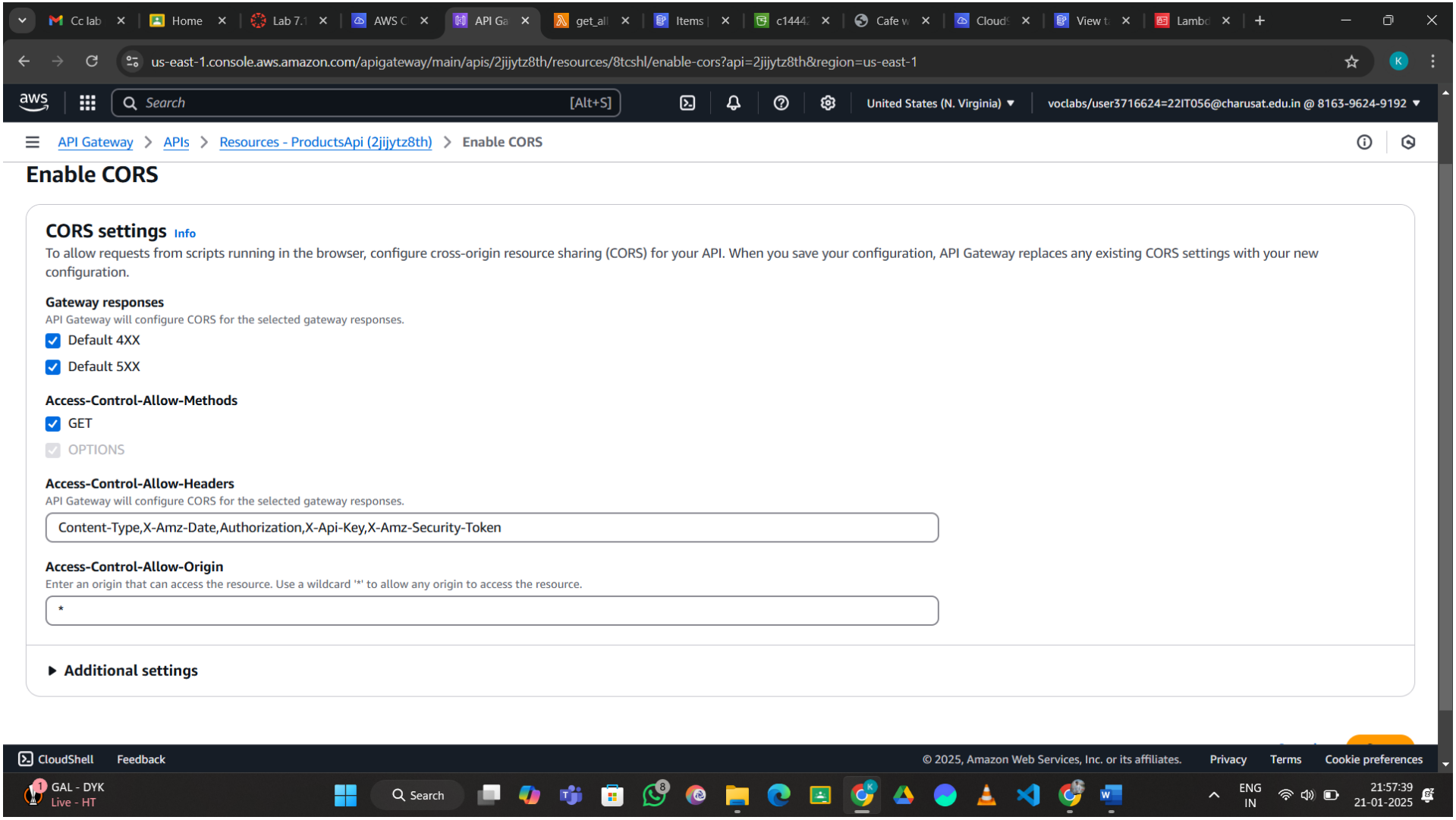

Figure 10: Enable CORS and update configuration

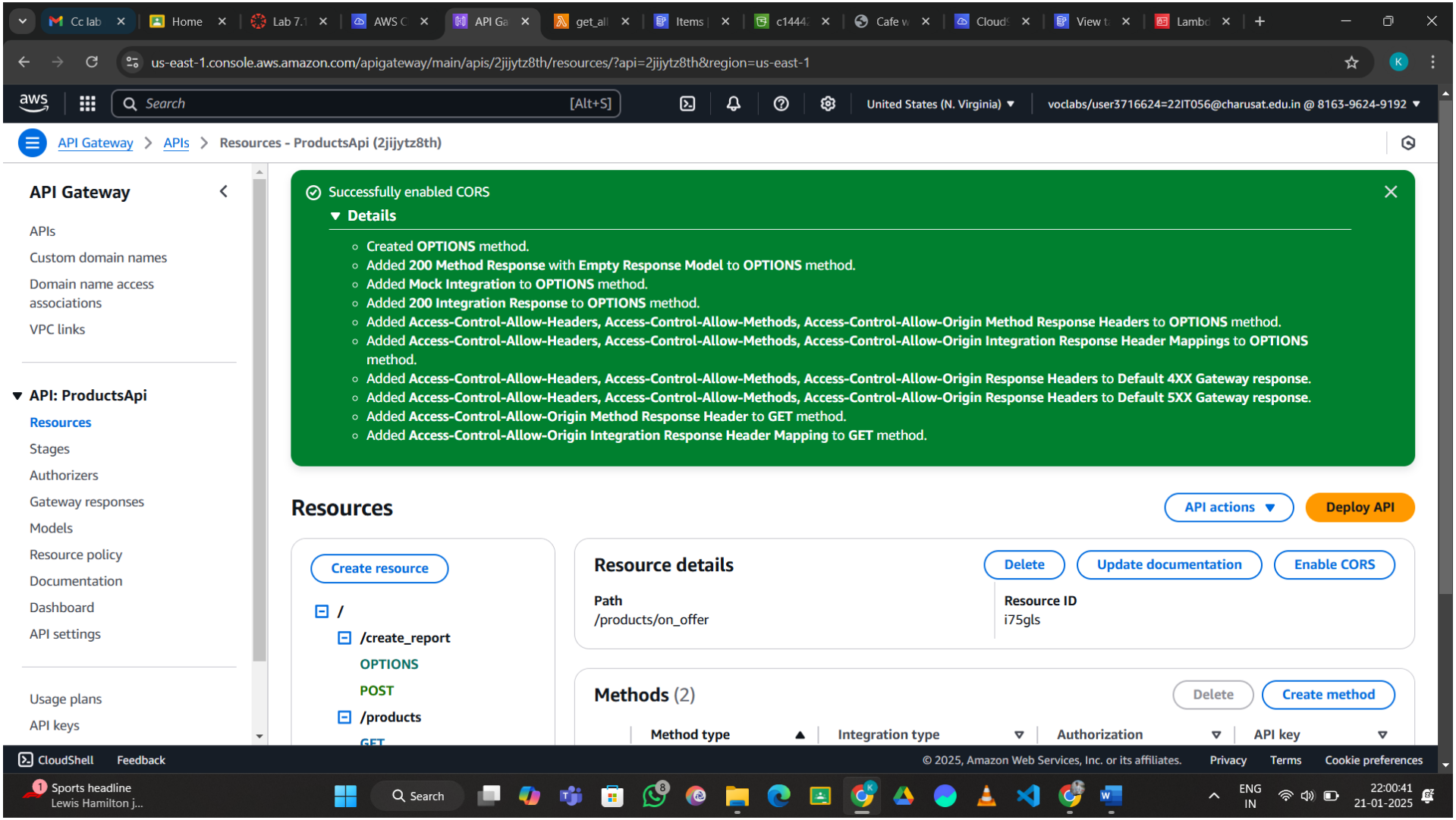

Figure 11: Update a access method

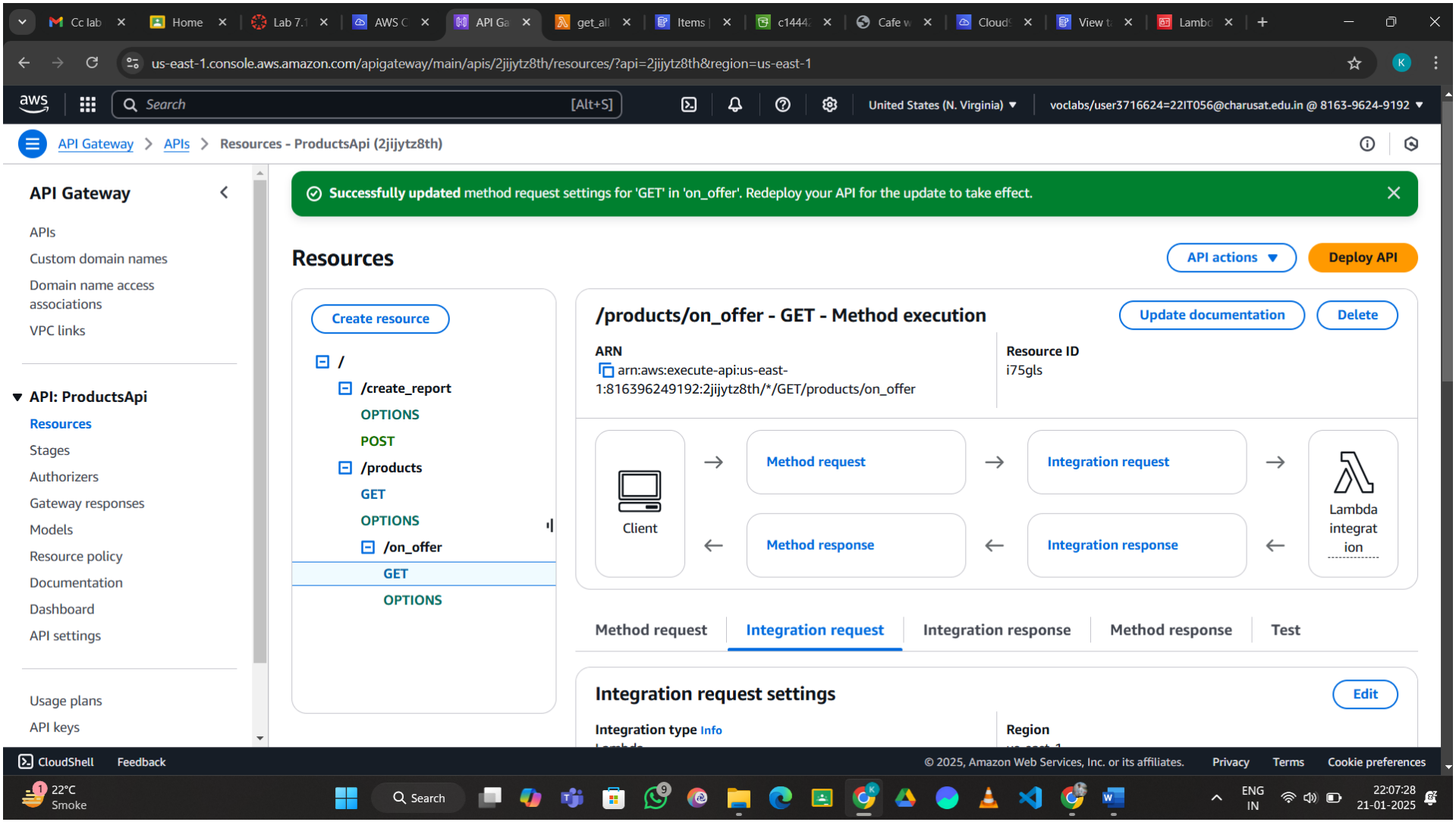

Figure 12: Retest updated on_offer request method

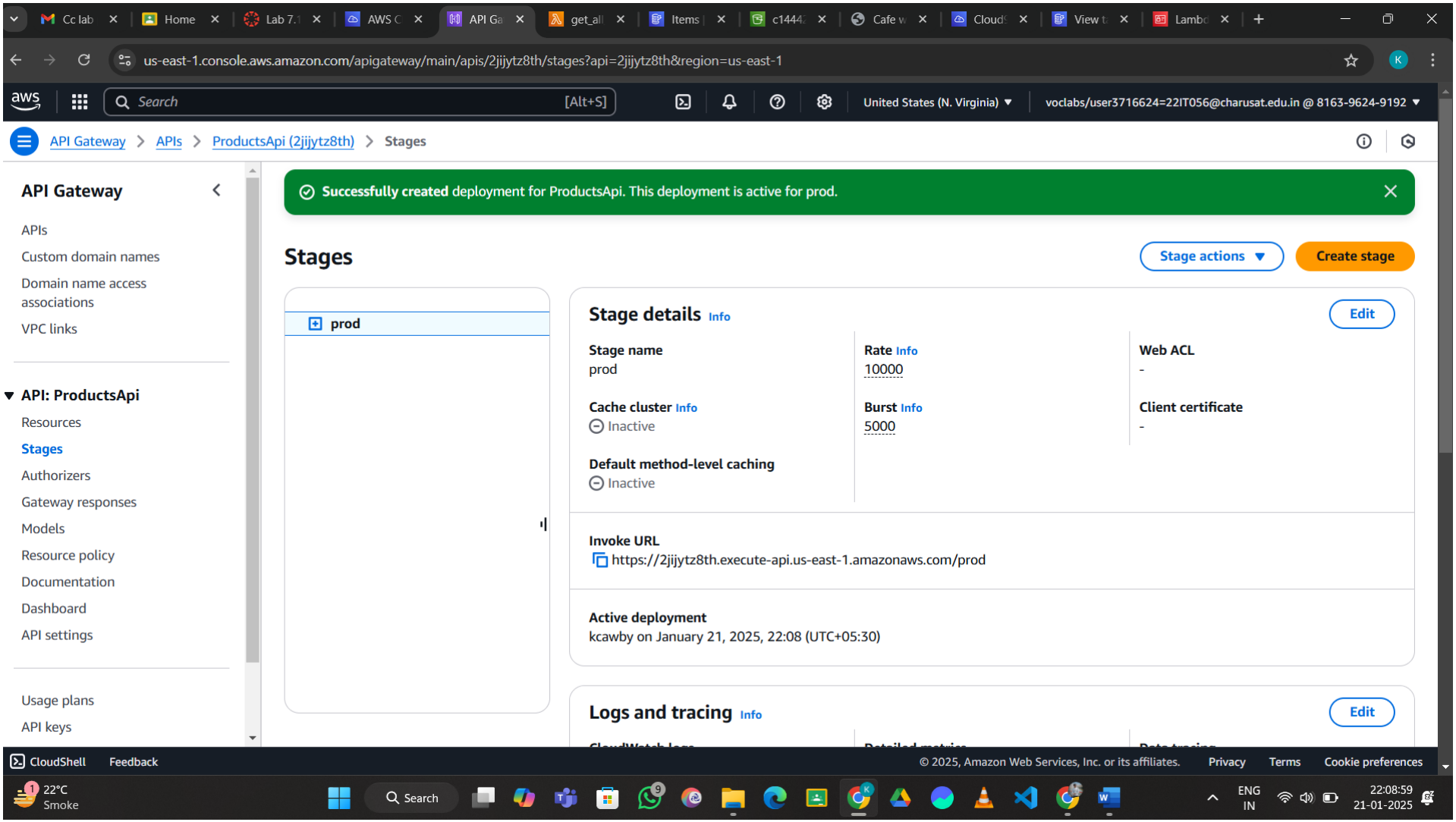

Figure 13: Deploy API

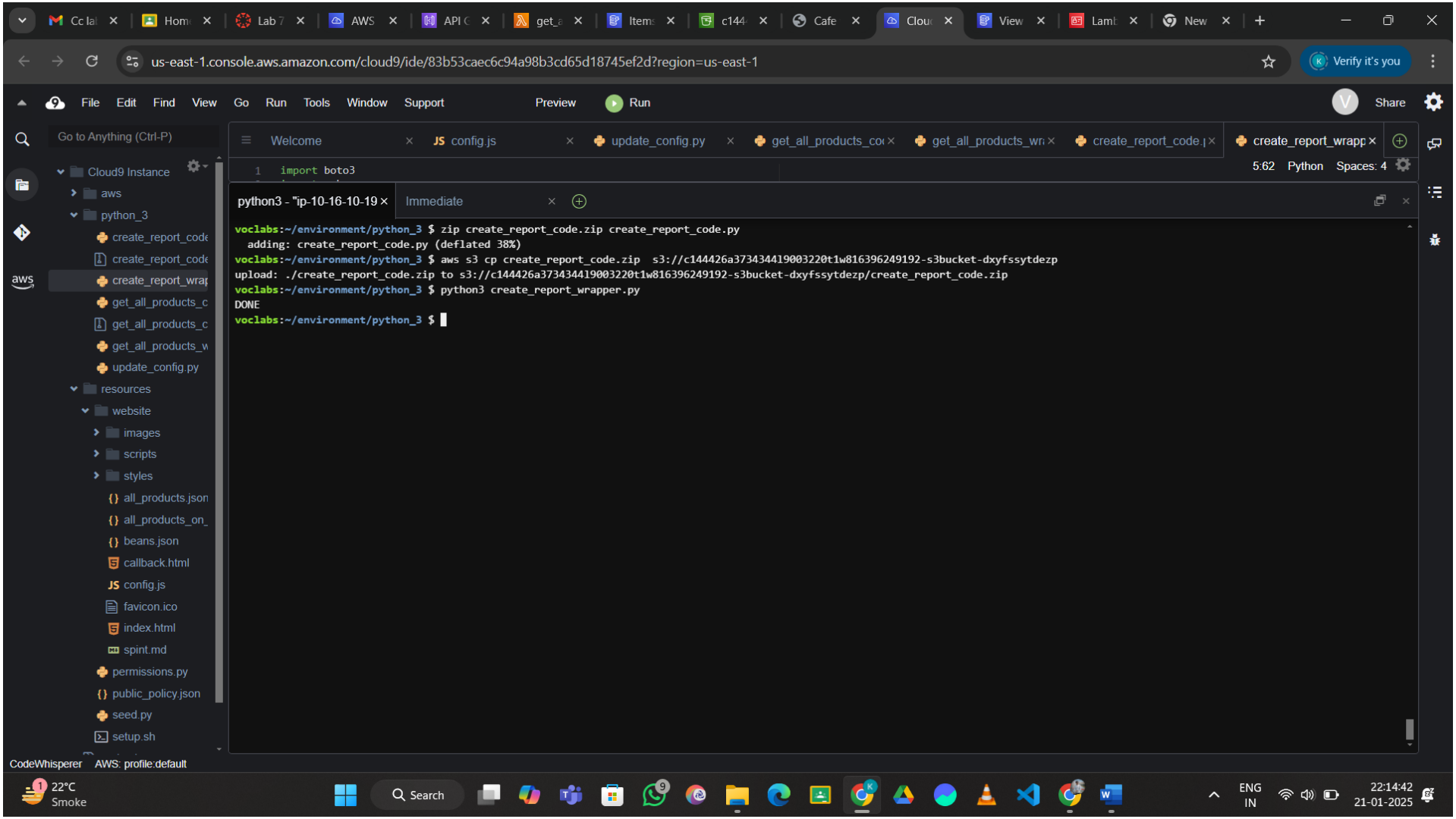

Figure 14: Update create_report_wrape.py file and run it

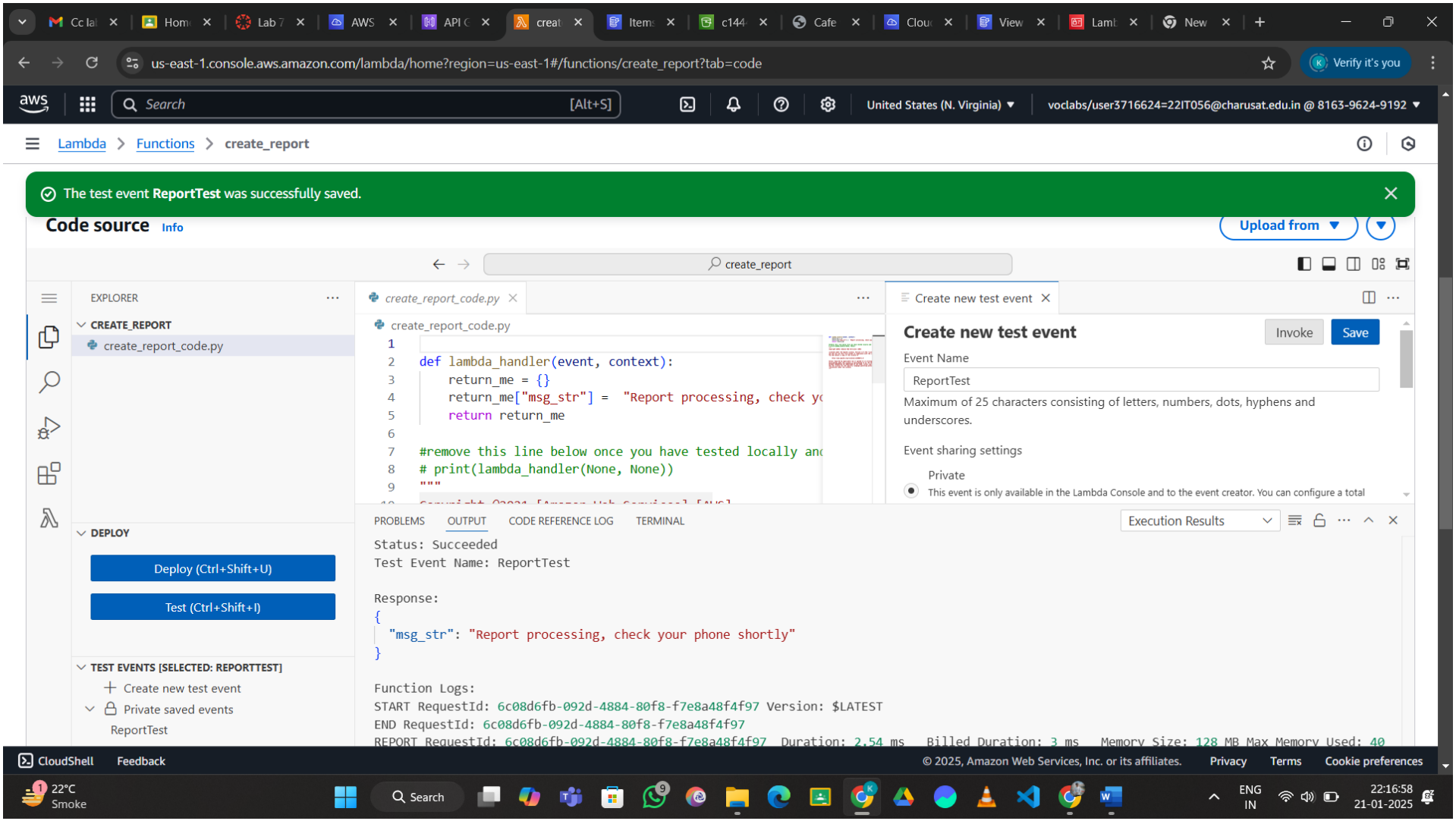

Figure 15: Test ReportTest

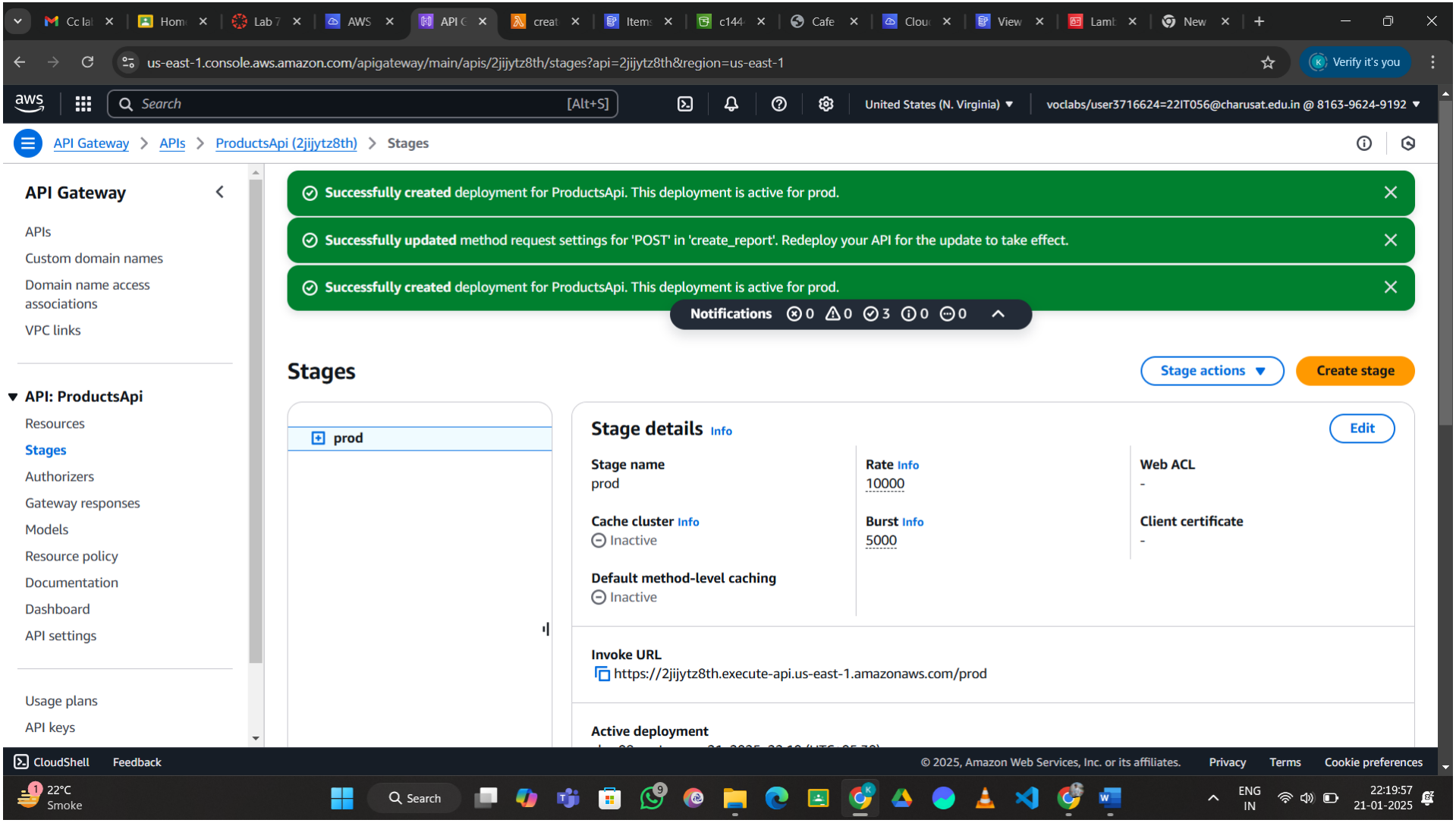

Figure 16: Update the request method and save it

LATEST APPLICATIONS:

1. AWS Lambda

- Dynamic content delivery for e-commerce, news, and media websites.
- Real-time data processing for IoT, sensor networks, and streaming applications.
- Event-driven microservices, reacting to changes in databases, file uploads, or queue messages.
- Automated workflows, such as image processing, file transformations, or notifications.
- Backend logic for chatbot and voice assistant integrations (e.g., Alexa skills).

2. Amazon DynamoDB

- High-performance NoSQL storage for gaming leaderboards, user sessions, or inventory management.
- Real-time analytics for streaming data and event tracking.
- Data storage for serverless applications with high read/write throughput requirements.
- Managing metadata and logs in media processing workflows.
- Supporting large-scale applications like ticketing systems and ride-sharing platforms.

3. Amazon API Gateway

- Backend for mobile and web applications, enabling secure REST/HTTP APIs.
- Real-time APIs for multiplayer gaming and collaborative tools.
- Integration with WebSocket APIs for chat apps and live data feeds.
- Exposing microservices for enterprise-scale distributed systems.
- Connecting serverless applications with external systems for hybrid architectures.

LEARNING OUTCOME:

By completing this lab, I will gain a comprehensive understanding of integrating AWS services to build a dynamic, serverless application. I will learn to create and deploy AWS Lambda functions to handle API requests and connect with Amazon DynamoDB for real-time data retrieval while designing and querying NoSQL databases and optimizing data retrieval using indexes. Additionally, I will develop skills in configuring and managing REST APIs with Amazon API Gateway to enable secure and scalable communication between the frontend and back end. This includes connecting API methods to Lambda functions, managing API security, and configuring AWS IAM roles and policies for controlled resource

access. Overall, this lab will equip me with practical knowledge to build event-driven architectures and implement real-world applications, such as dynamically updating a café menu on a website.

REFERENCE:

1. https://awsacademy.instructure.com/courses/104050