EXPERIMENT 4

Aim: To study and Implement Platform as a Service using AWS/Microsoft Azure.

Objective:

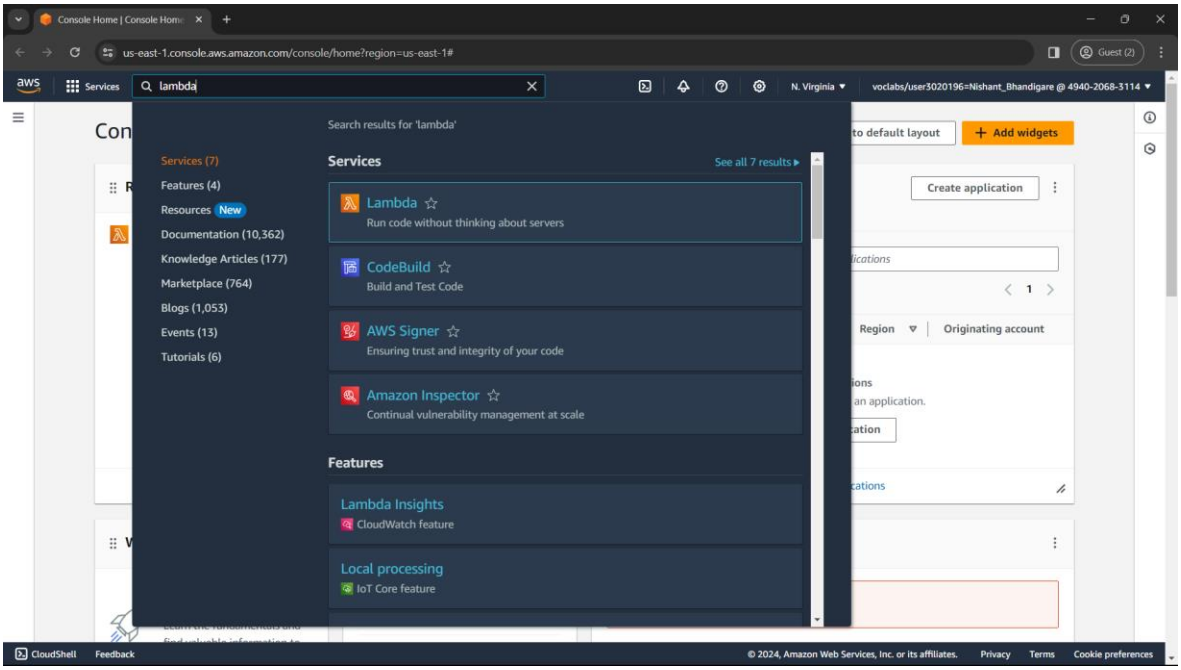
- Describe microservices and Serverless concepts
- Discuss event-driven architectures
- Describe the features and benefits of AWS Lambda functions
- Discuss how to configure AWS Lambda functions
- Identify how to monitor AWS Lambda functions
- Describe best practices for working with AWS Lambda
- Identify the features and benefits of additional AWS Serverless services

Theory: To embark on a study and implementation of Platform as a Service (PaaS) using AWS or Microsoft Azure, it's essential to grasp the foundational concepts and functionalities inherent in these cloud platforms. PaaS provides developers with a complete platform for building, deploying, and managing applications without the complexity of managing underlying infrastructure. Both AWS and Microsoft Azure offer robust PaaS solutions tailored to various application development needs. The theory behind this endeavor revolves around understanding the core components of PaaS offerings, such as development tools, middleware, database management systems, and runtime environments, provided by AWS and Azure. These platforms offer services like AWS Elastic Beanstalk, AWS Lambda, Azure App Service, and Azure Functions, enabling developers to focus on application development without worrying about server management, scalability, or infrastructure maintenance. Key concepts include selecting the appropriate PaaS services based on application requirements, configuring and deploying applications, integrating with other cloud services, and monitoring application performance. Through this study and implementation, participants gain insights into the agility, scalability, and efficiency offered by PaaS solutions, empowering them to accelerate application development and deployment while reducing operational overhead. This experiment serves as a practical exploration of cloud-native development paradigms and equips participants with the skills to leverage AWS or Microsoft Azure effectively for building and managing modern applications in the cloud.

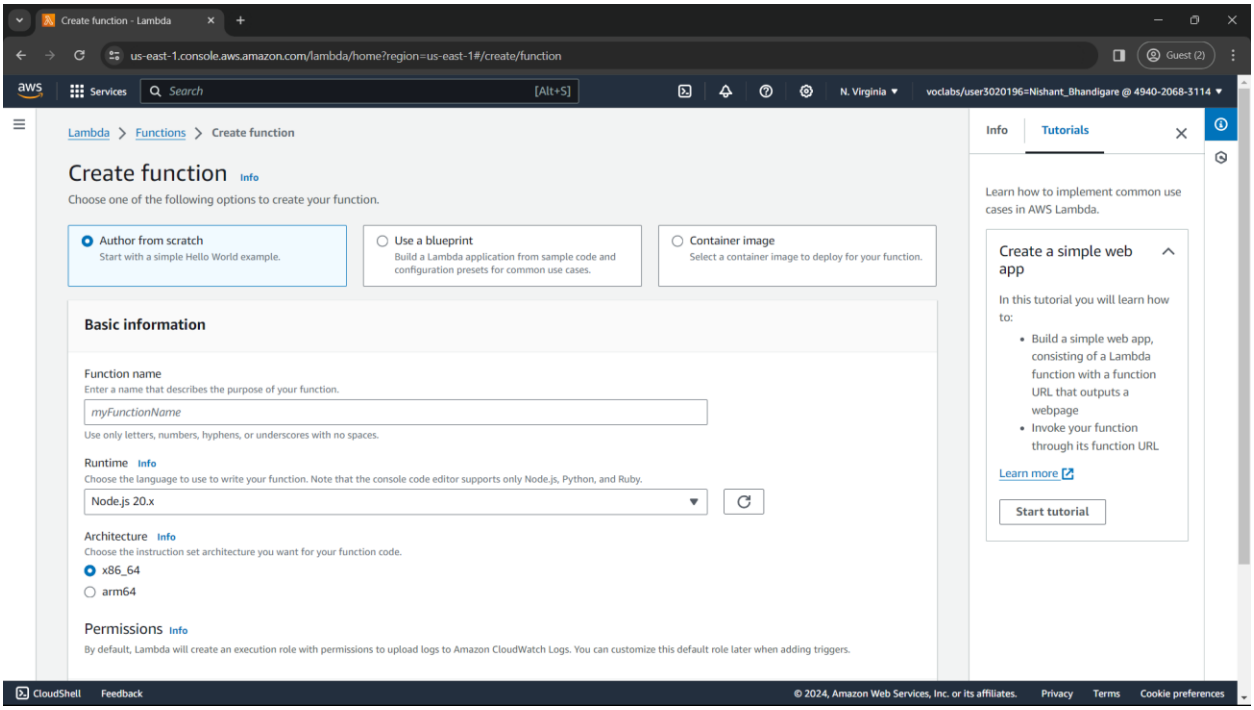
Implementation And Output:

Task 1: Creating a Lambda function

1. In the **AWS Management Console**, on the **Services** menu, enter **Lambda**. From the search results, choose **Lambda**.

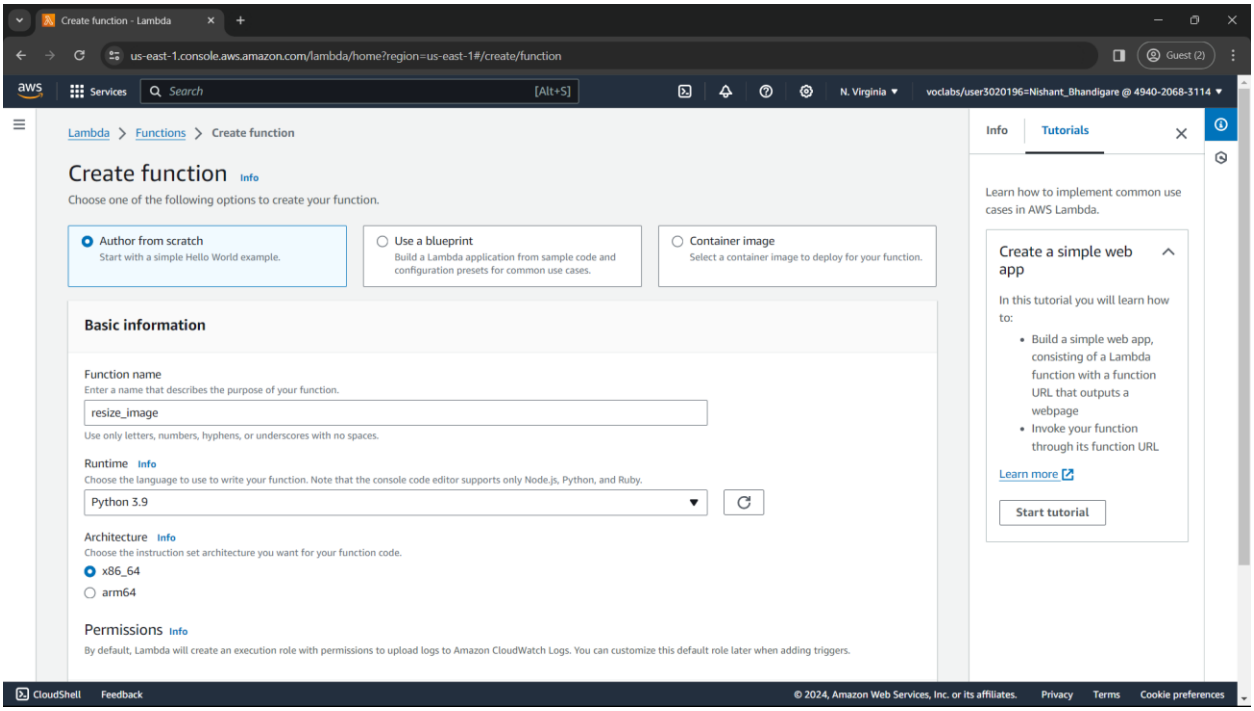


2. Choose **Create Function**.

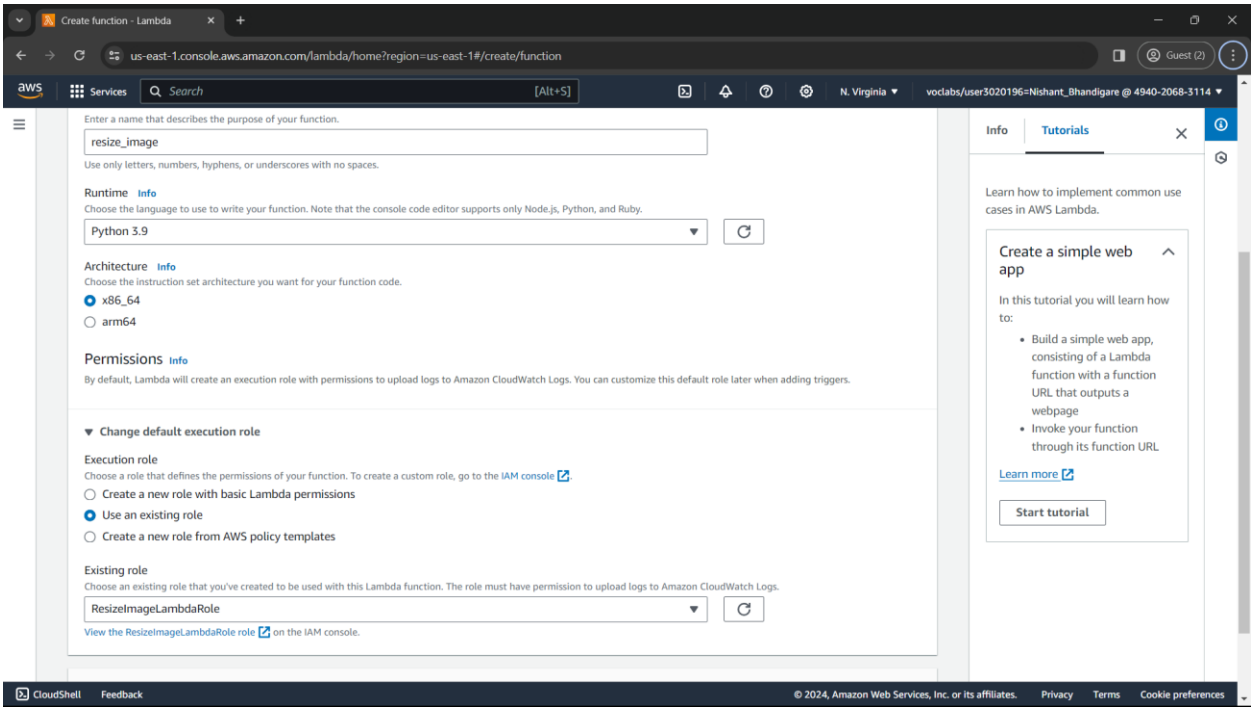


Edit the following:
Function name: resize_image

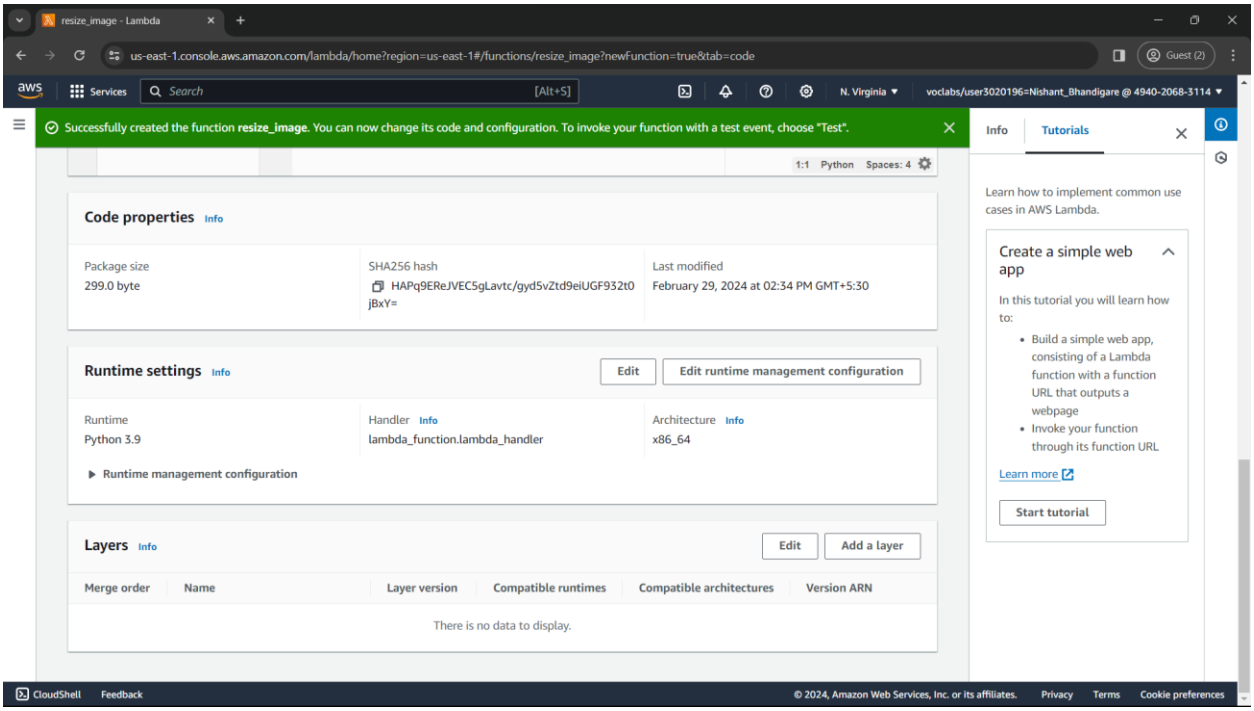
3. **Runtime:** Python3.9



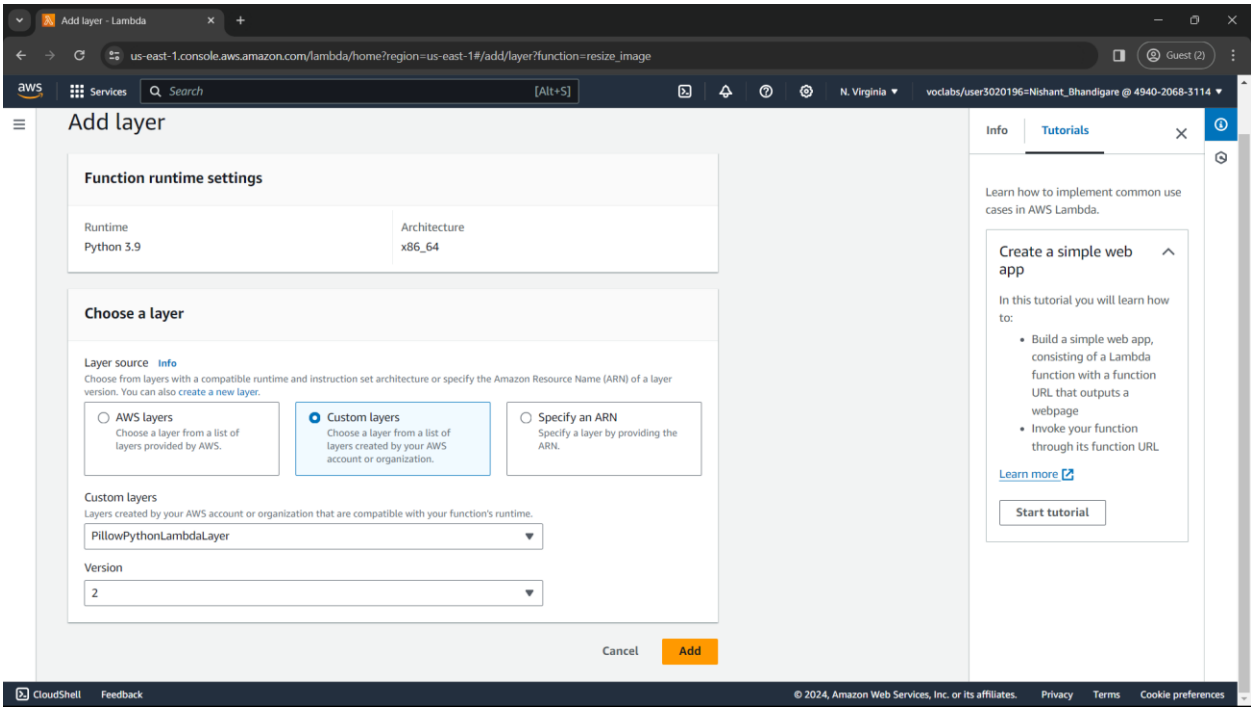
4. Expand the **Change default execution role** section. Choose **Use an existing role**. In the dropdown list for **Existing Role**, choose **ResizeImageLambdaRole**.



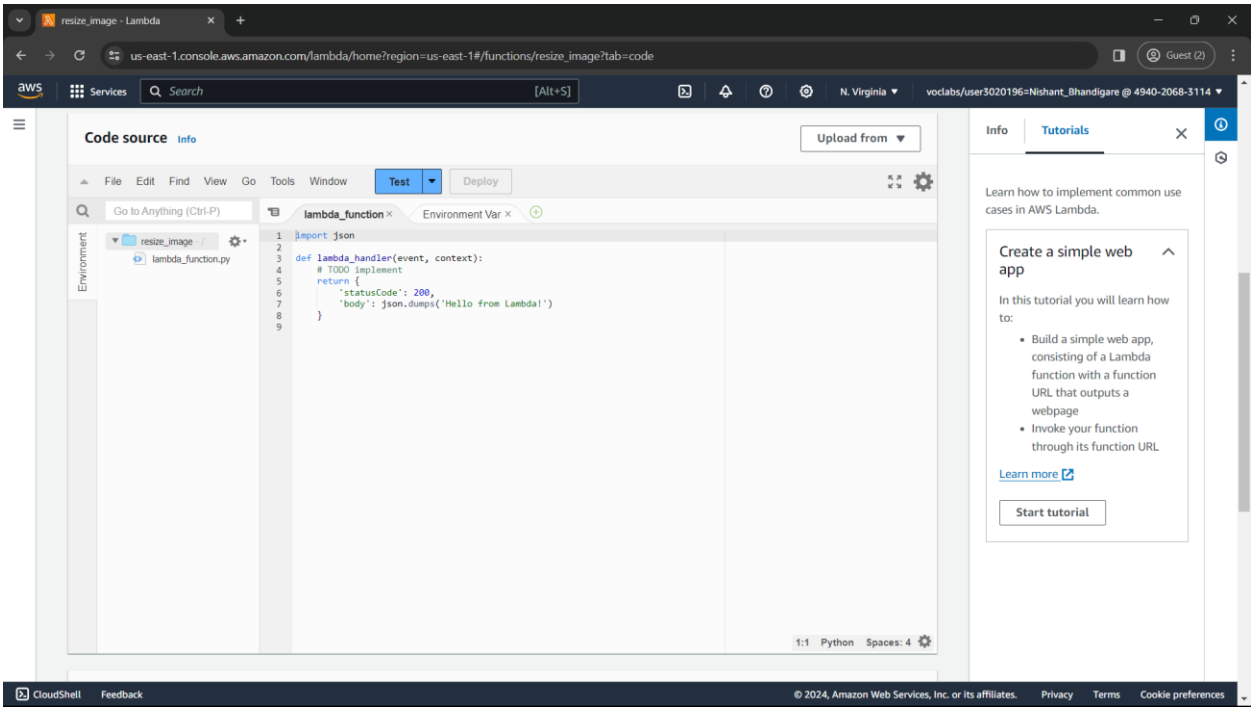
5. Choose **Create function**.
6. Scroll to the **Layers** section.



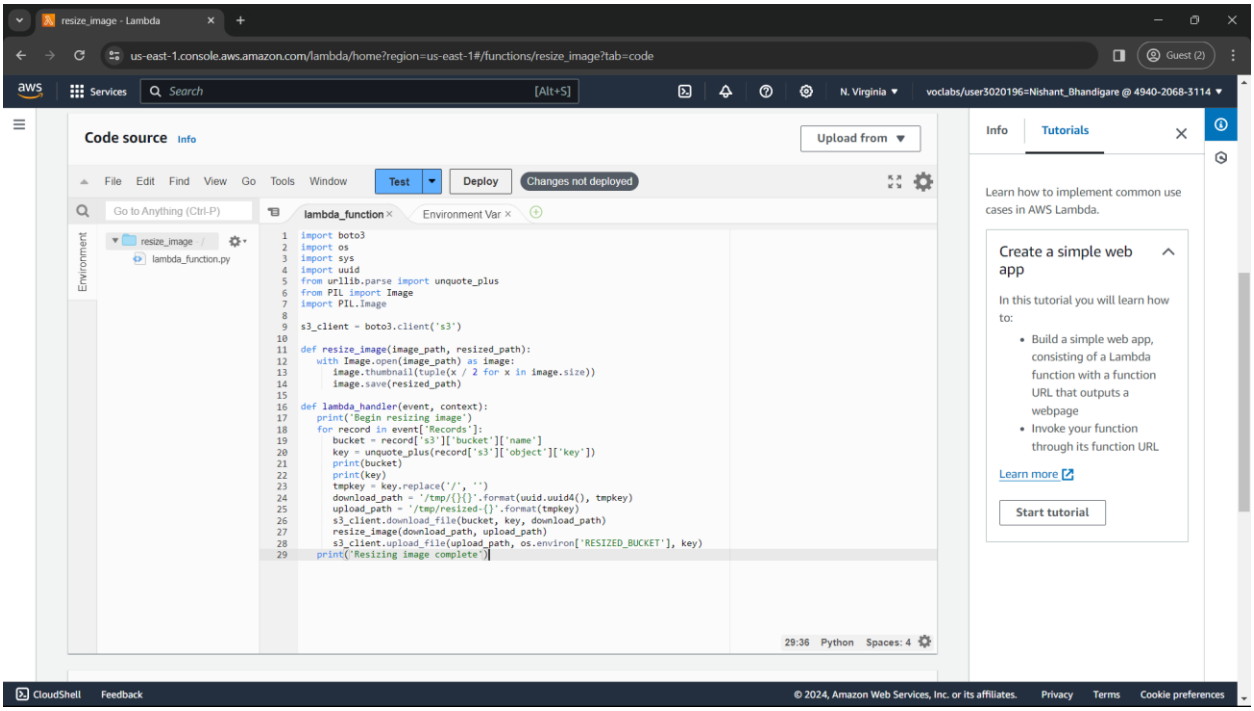
7. Choose **Add a layer**.
8. Choose **Custom layers**.
9. From the **Custom layers** dropdown list, choose **PillowPythonLambdaLayer**.
10. From the **Version** dropdown list, choose **1** or the option with the highest version number.
11. Choose **Add**.



12. Scroll to the **Code source** section of the page.



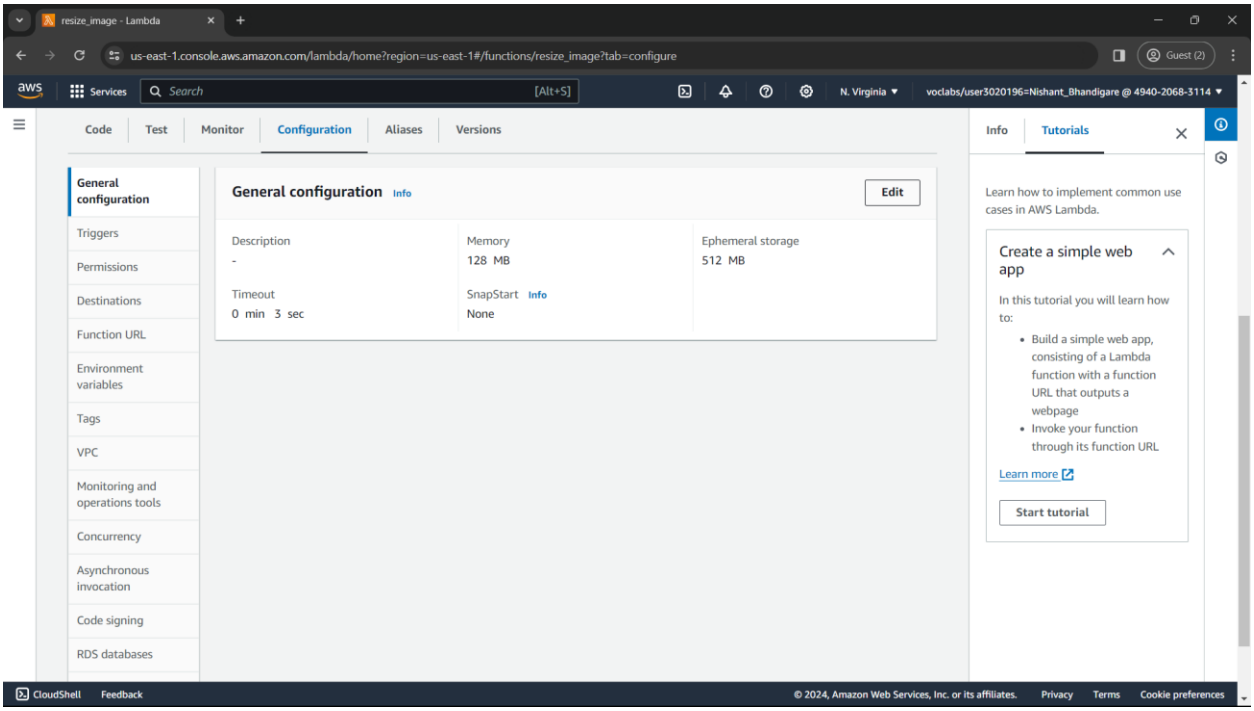
13. Copy the code below, and replace the default source code in the lambda_function.py file with it.



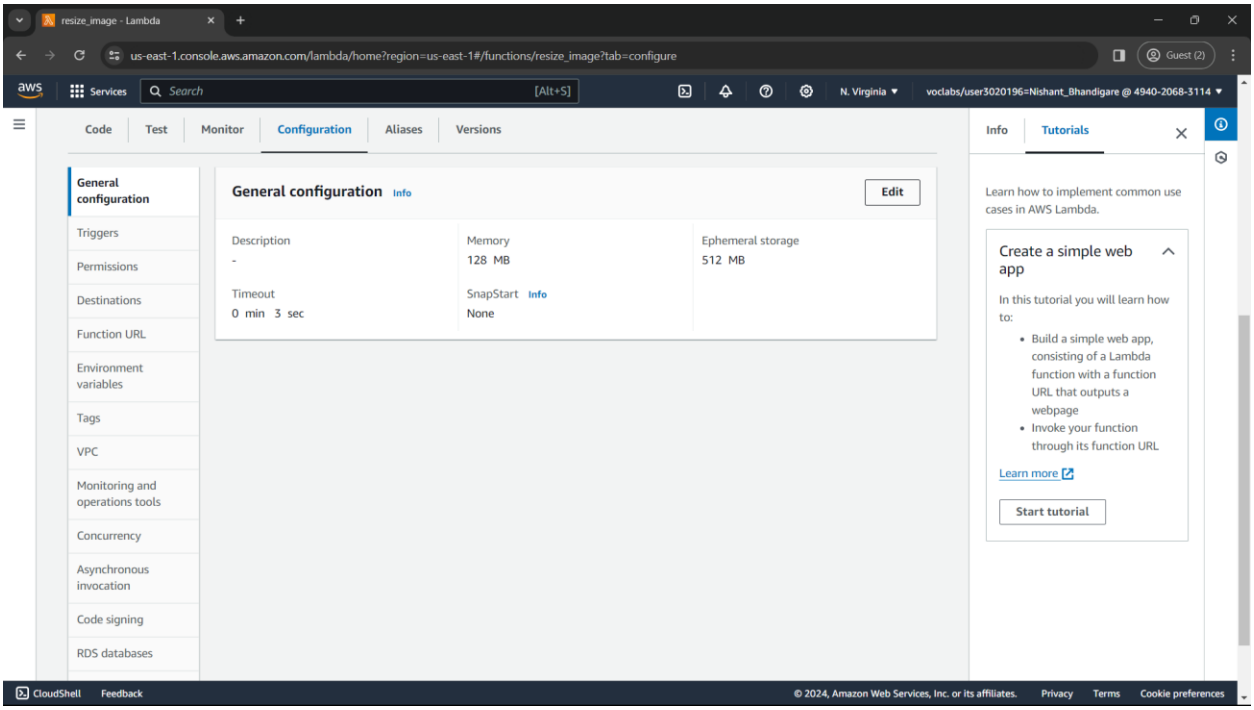

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image
s3_client = boto3.client('s3')
def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)
def lambda_handler(event, context):
```

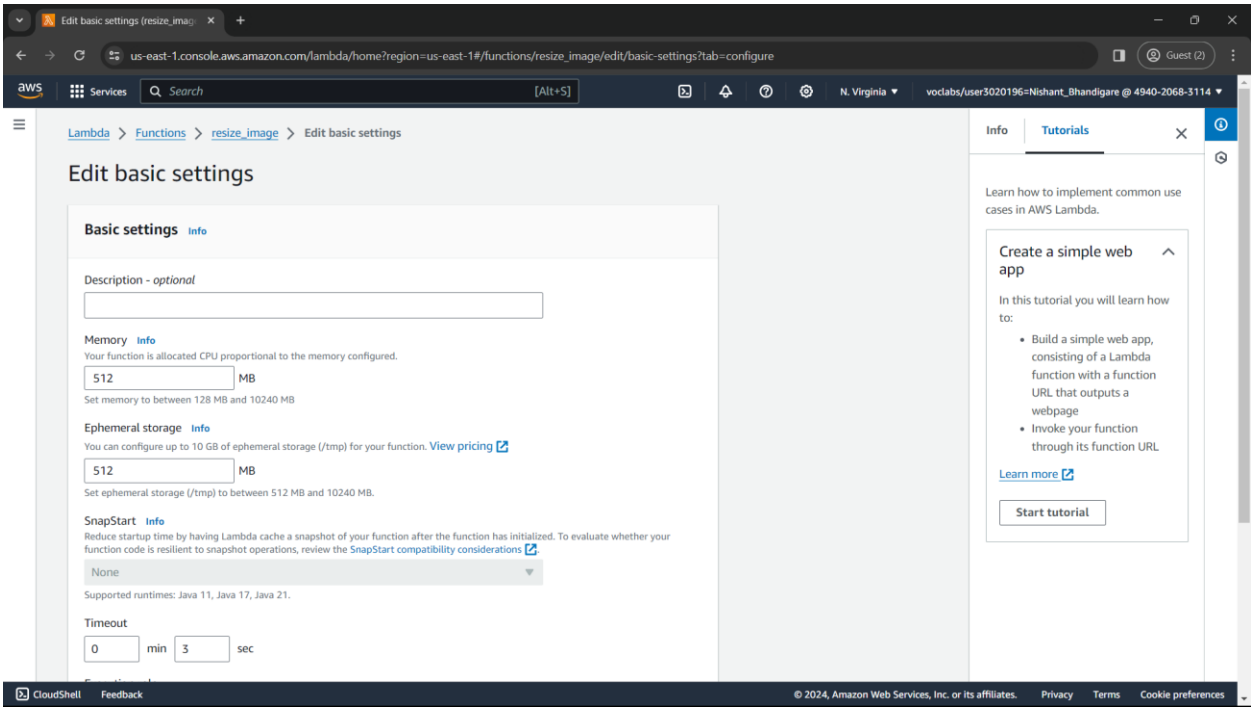
```
print('Begin resizing image')
for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    print(bucket)
    print(key)
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, os.environ['RESIZED_BUCKET'], key)
print('Resizing image complete')
```

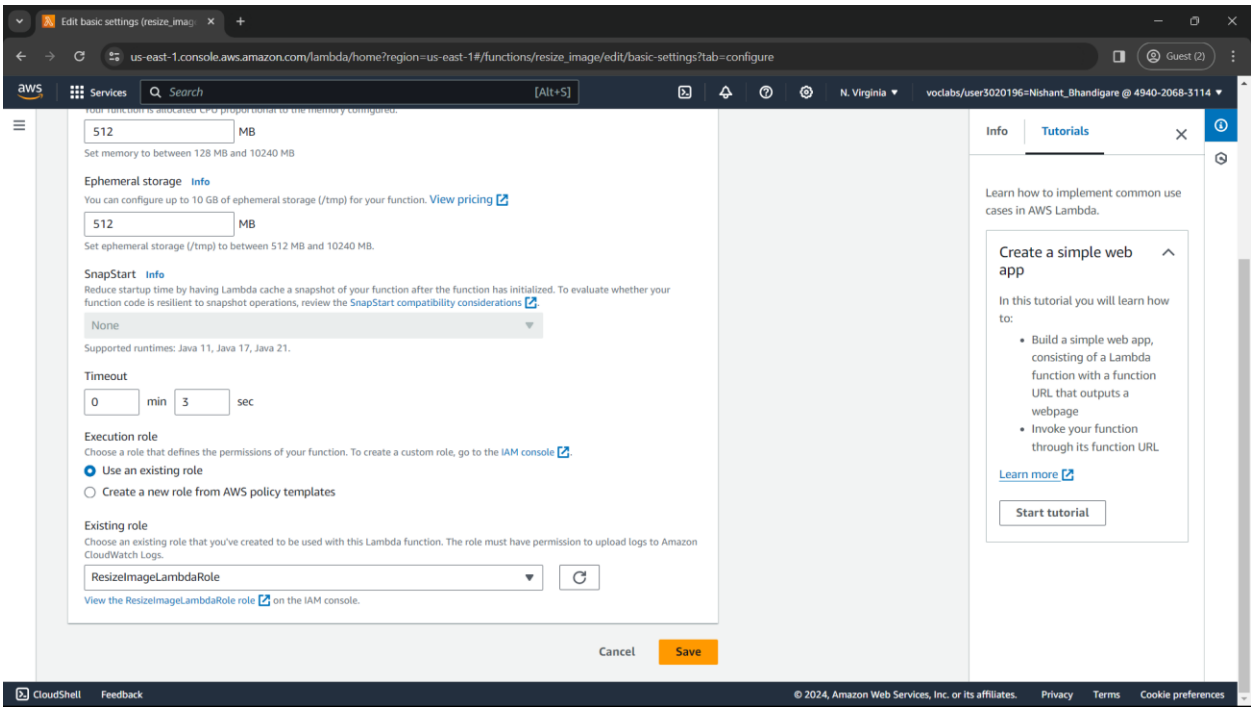
14. To deploy your Lambda function, choose **Deploy**.
15. Choose the **Configuration** tab, and then choose **General configuration**. Choose **Edit**.



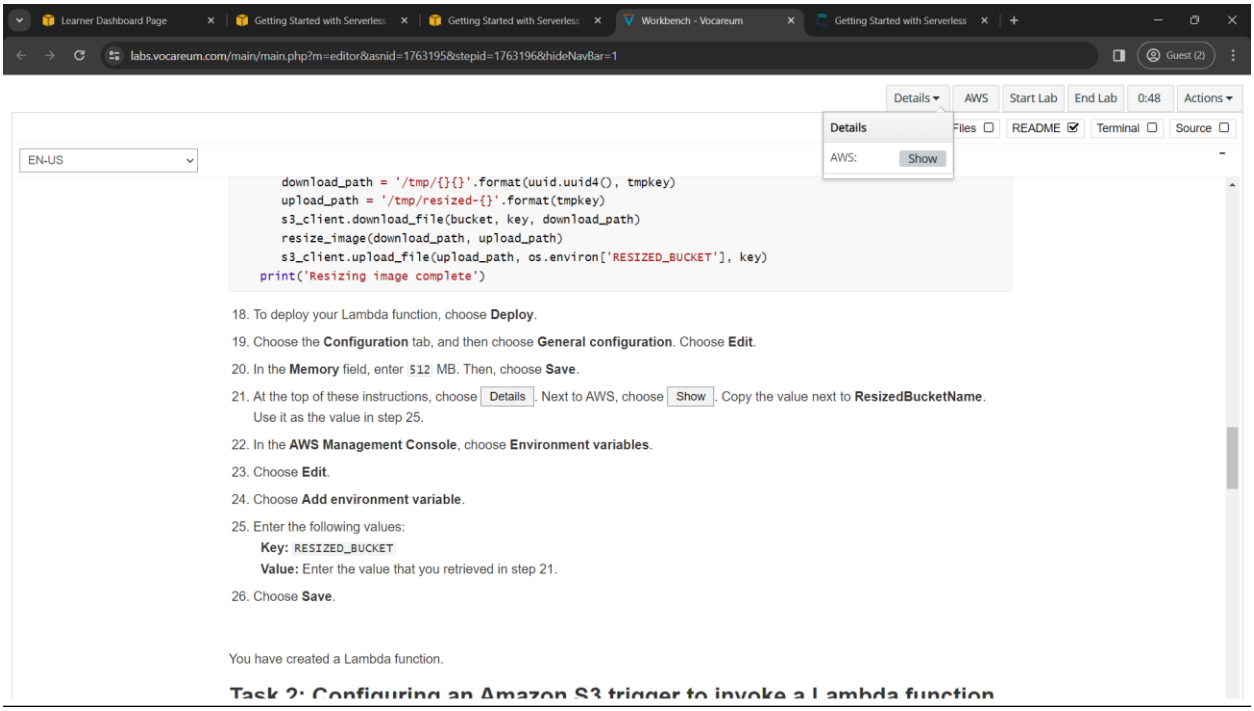
16. In the **Memory** field, enter 512 MB. Then, choose **Save**.

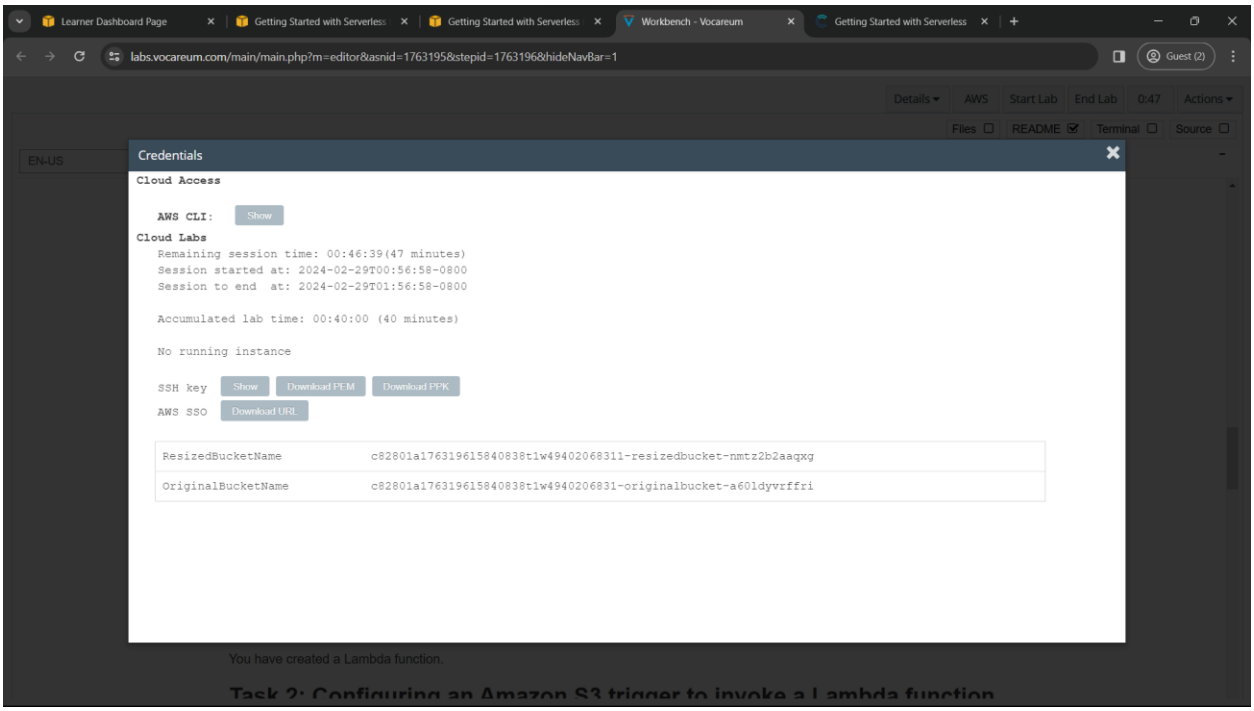




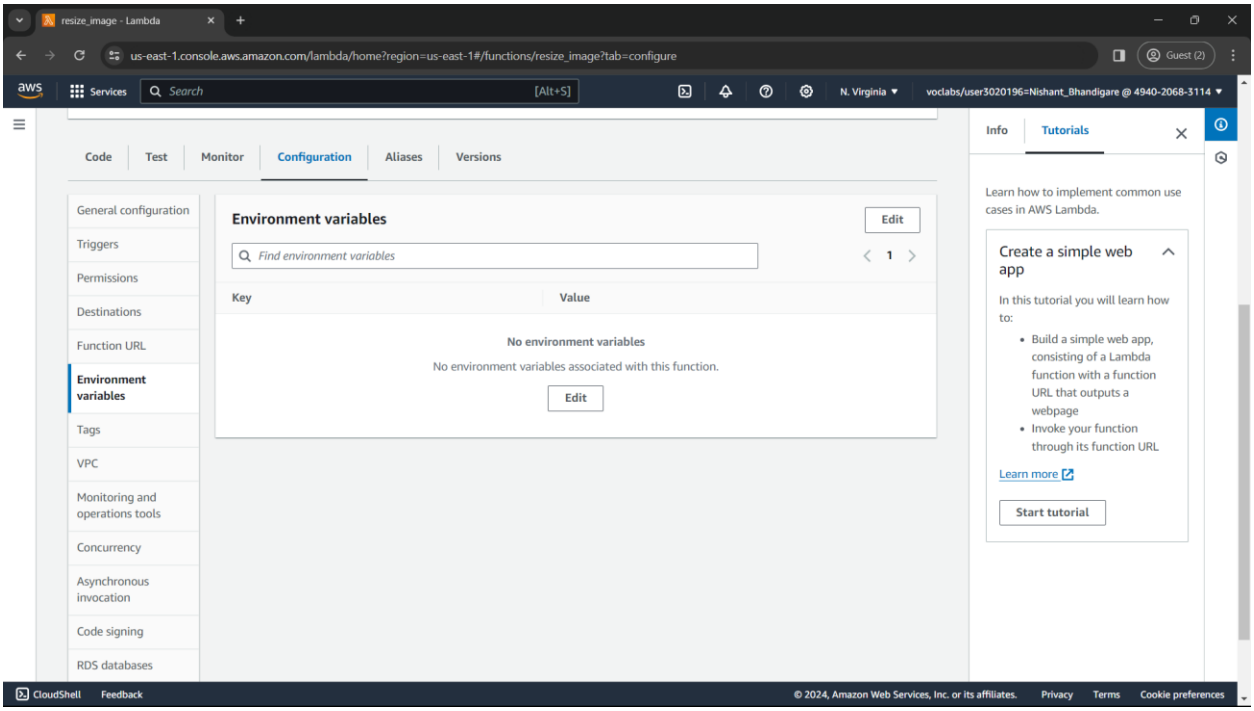


17. At the top of these instructions, choose Details. Next to AWS, choose Show. Copy the value next to **ResizedBucketName**. Use it as the value in step 21.

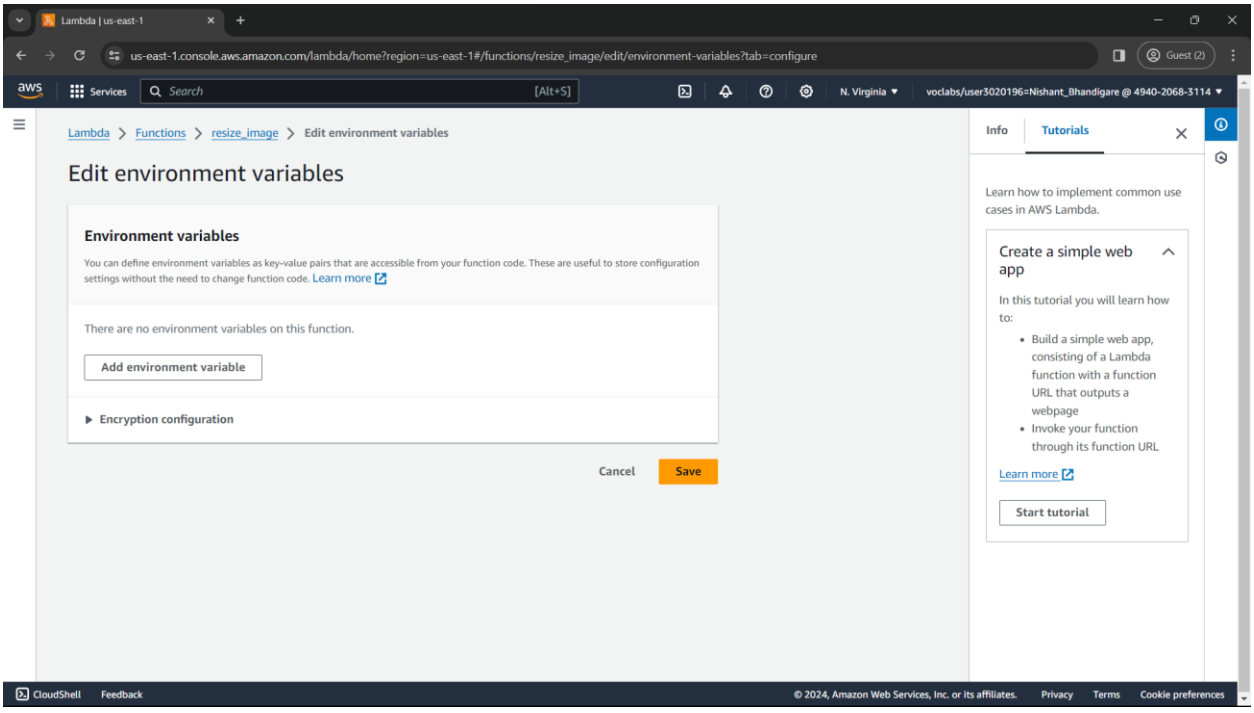




18. In the **AWS Management Console**, choose **Environment variables**.



19. Choose **Edit**.
20. Choose **Add environment variable**.

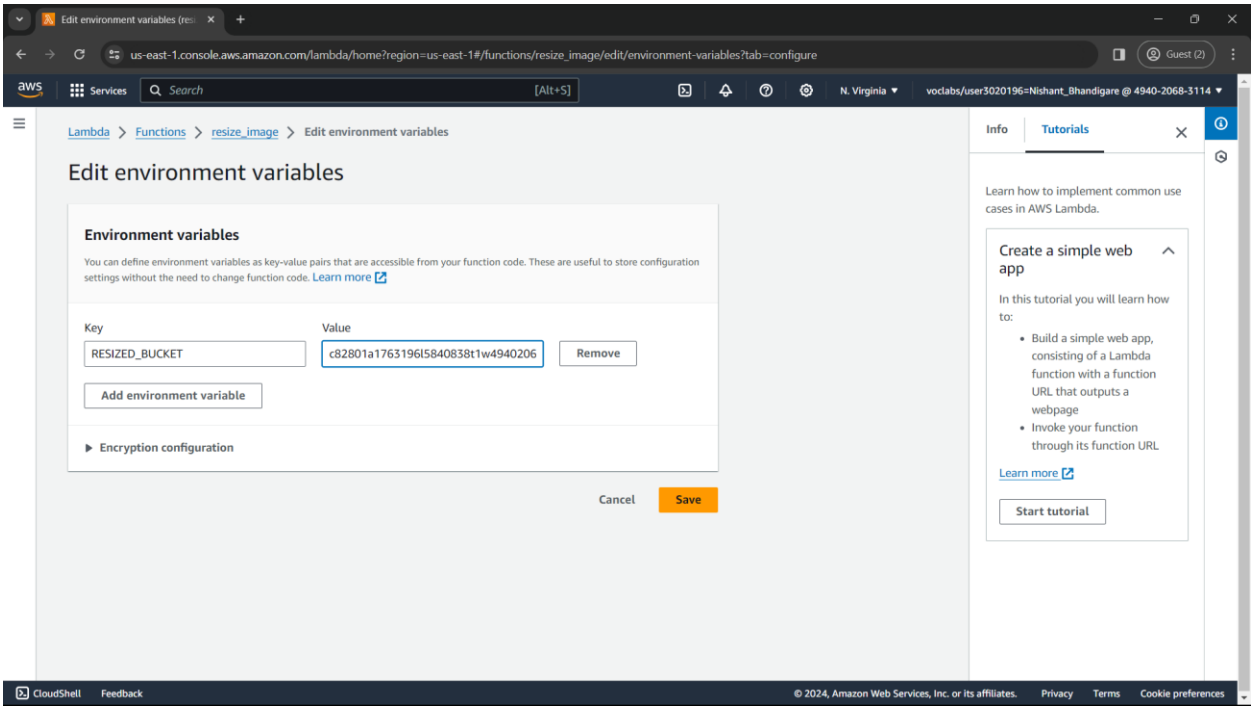


Enter the following values:

Key: RESIZED_BUCKET

21. **Value:** Enter the value that you retrieved in step 17.

22. Choose **Save**.

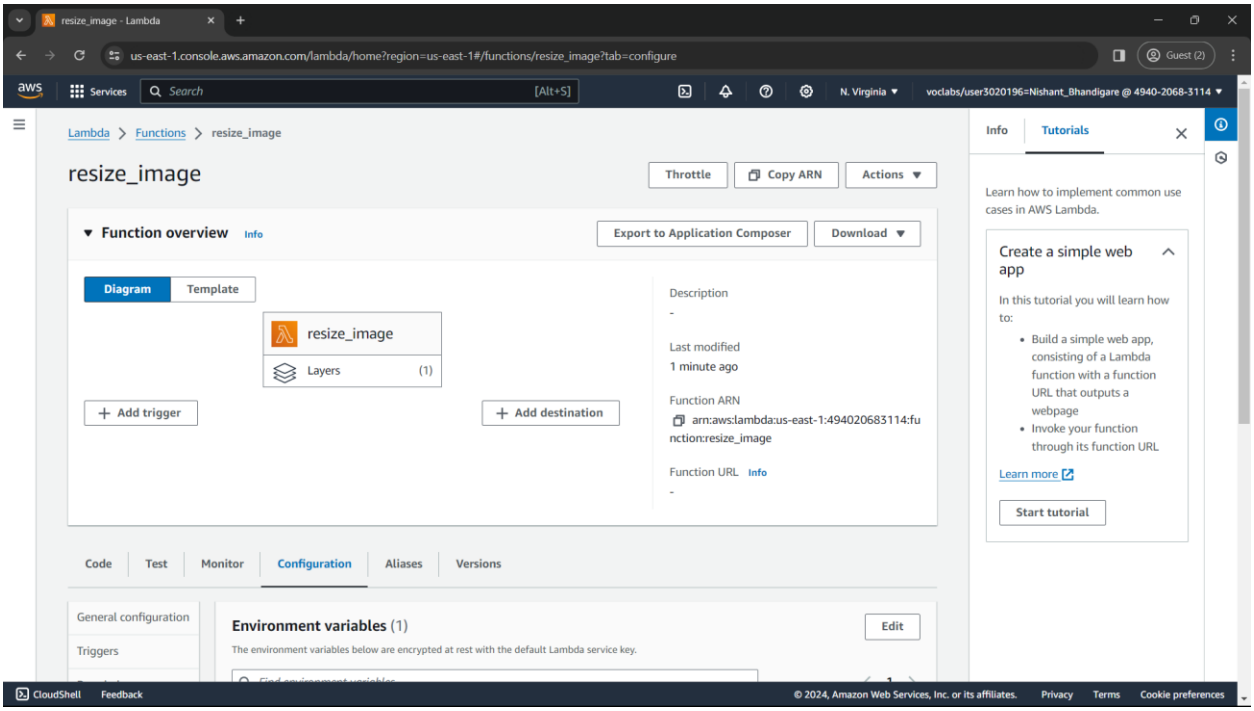


You have created a Lambda function.

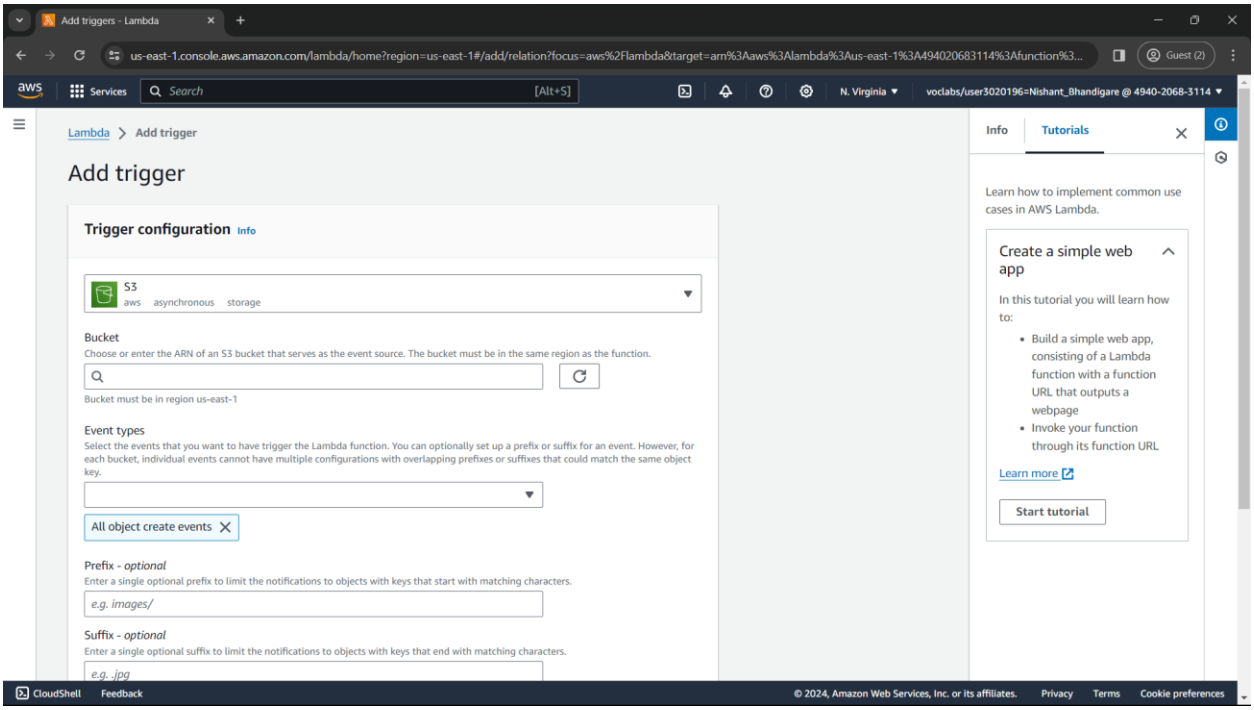
Task 2: Configuring an Amazon S3 trigger to invoke a Lambda function

In this task, you configure an S3 trigger on an existing S3 bucket and your Lambda function. The Lambda function resizes images and places them in another bucket.

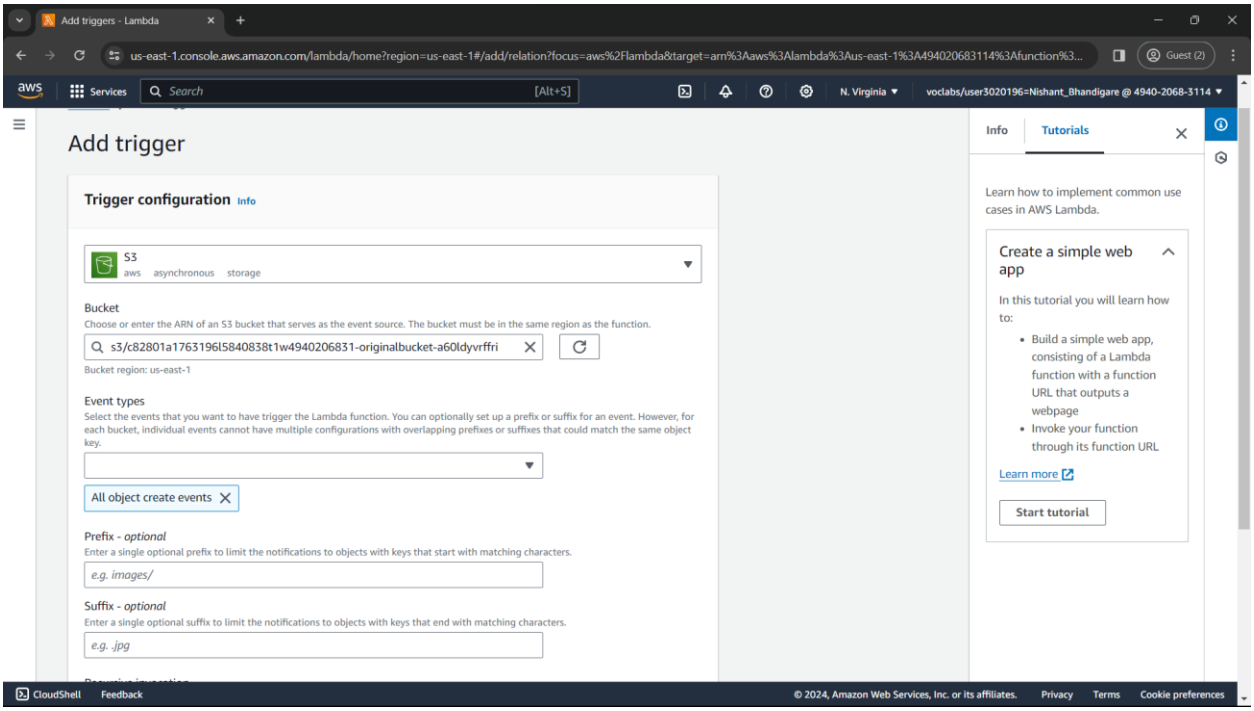
23. In the **Function overview** section of the Lambda console near the top of the page, choose **Add trigger**.



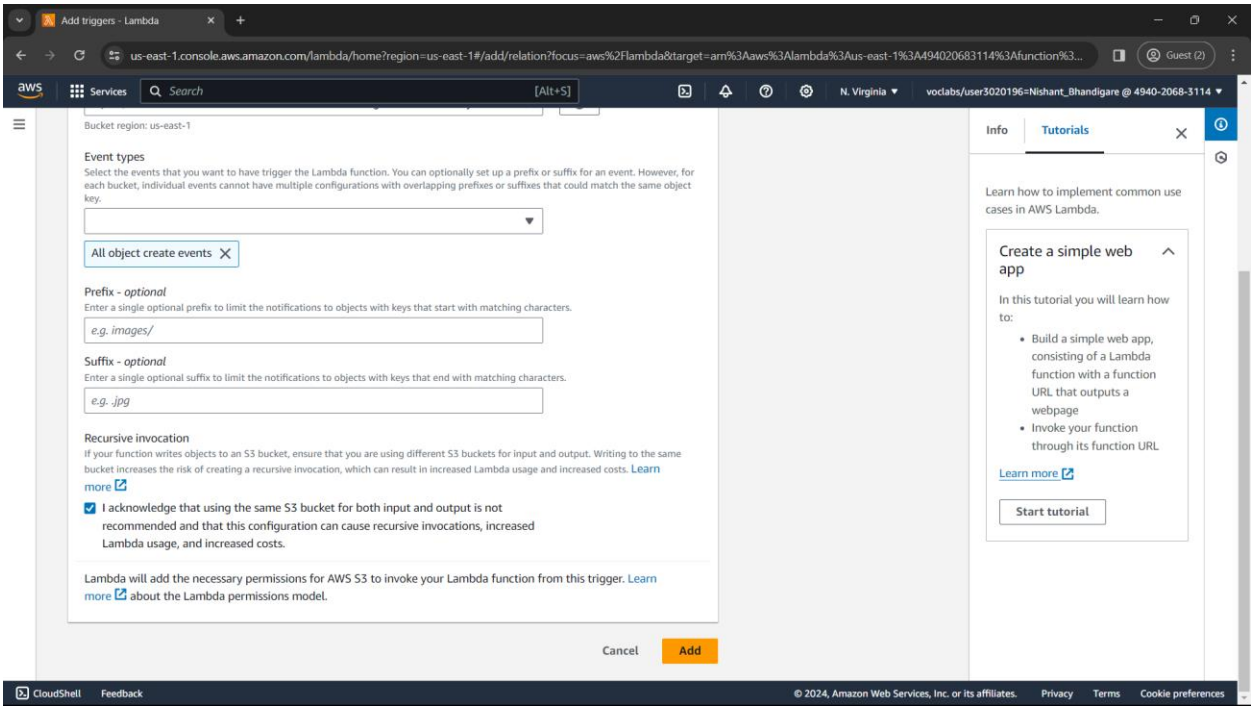
24. In the **Trigger configuration** section, choose **S3** from the dropdown list.



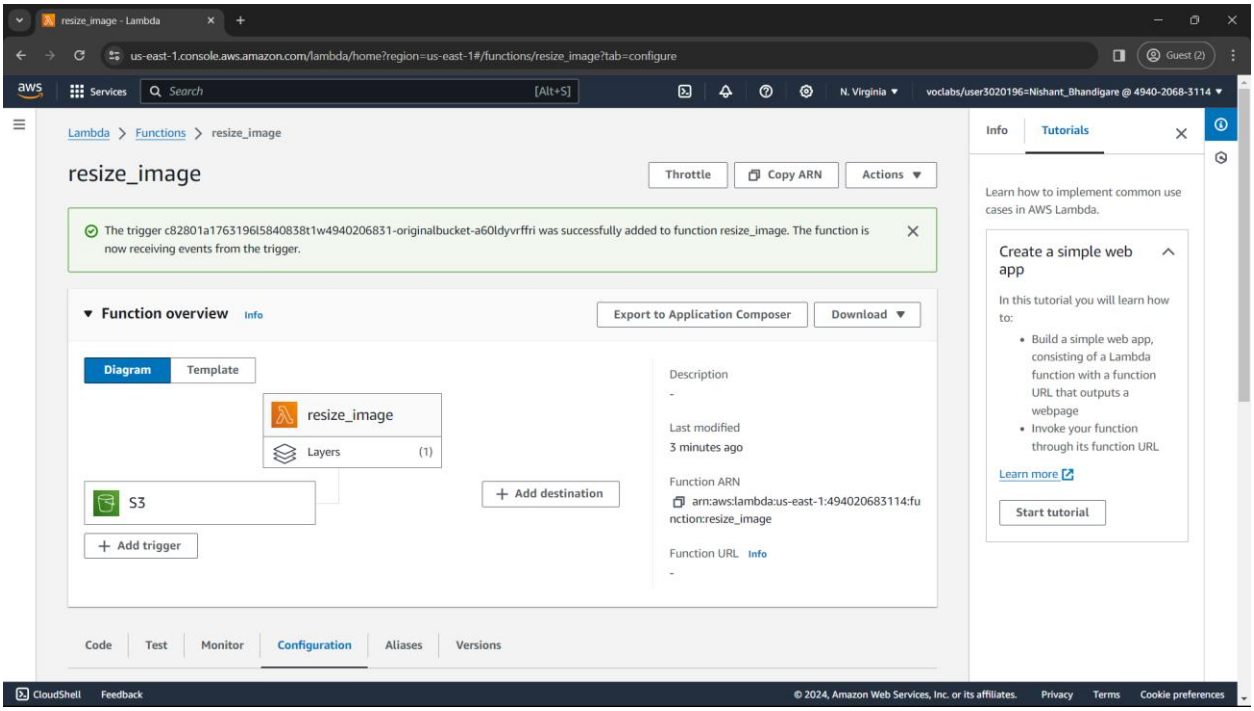
25. For Bucket, choose the bucket with **original** in the name.
26. For Event Type, choose **All object create events**.
27. Acknowledge the notification for **Recursive invocation** by selecting the check box.



28. Choose **Add**.



You have configured your Lambda function to be initiated when a new object is uploaded to the S3 bucket.

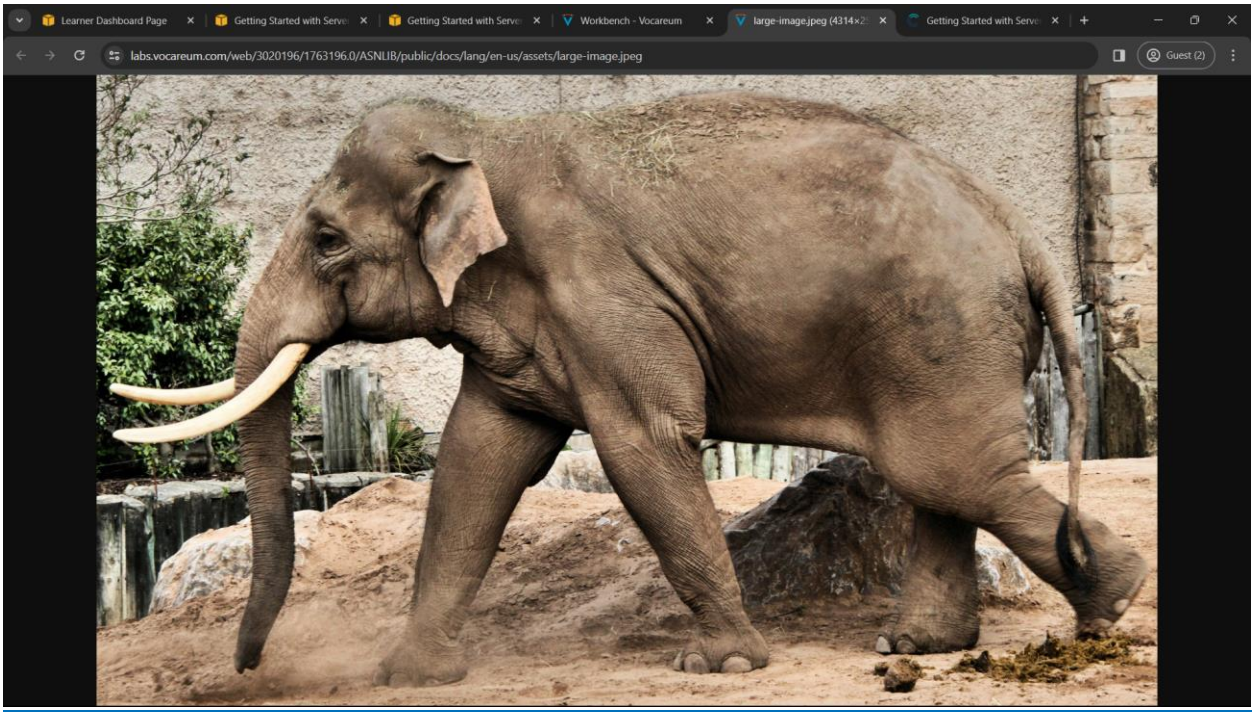


Task 3: Uploading an image to the Amazon S3 bucket

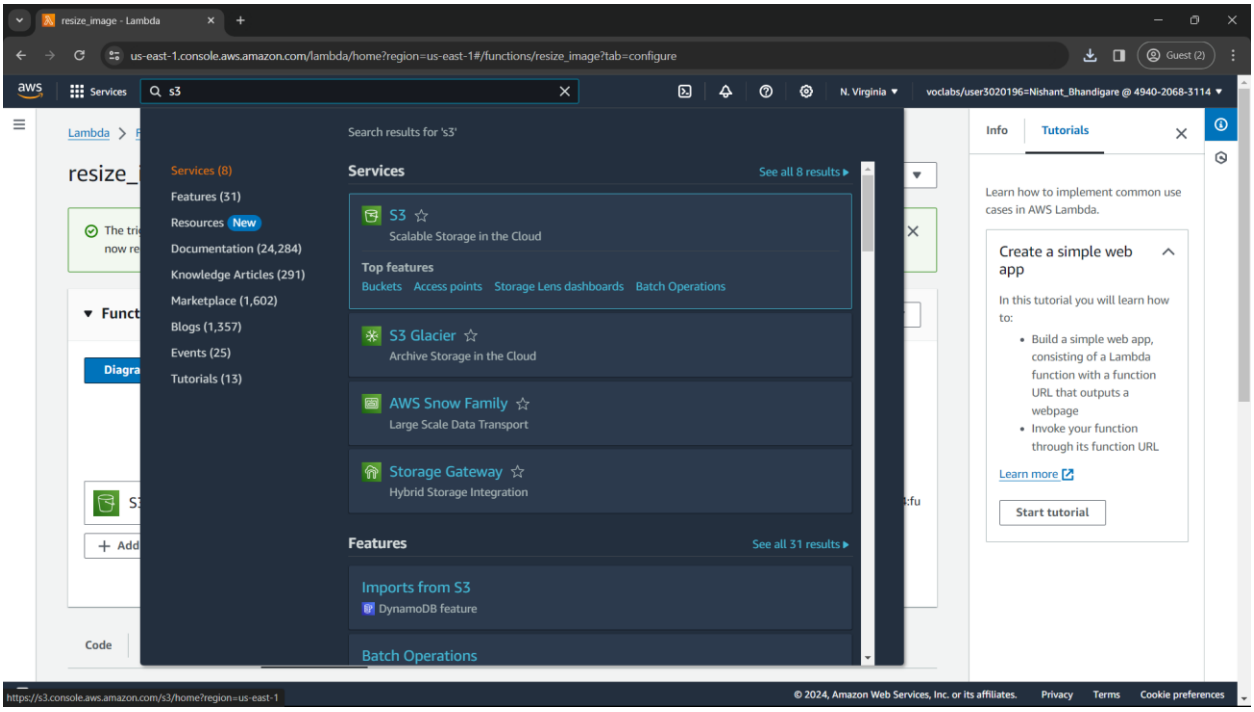
In this task, you upload an image file to your bucket.

29. Open the context (right-click) menu for the following link, and download the file to your computer:

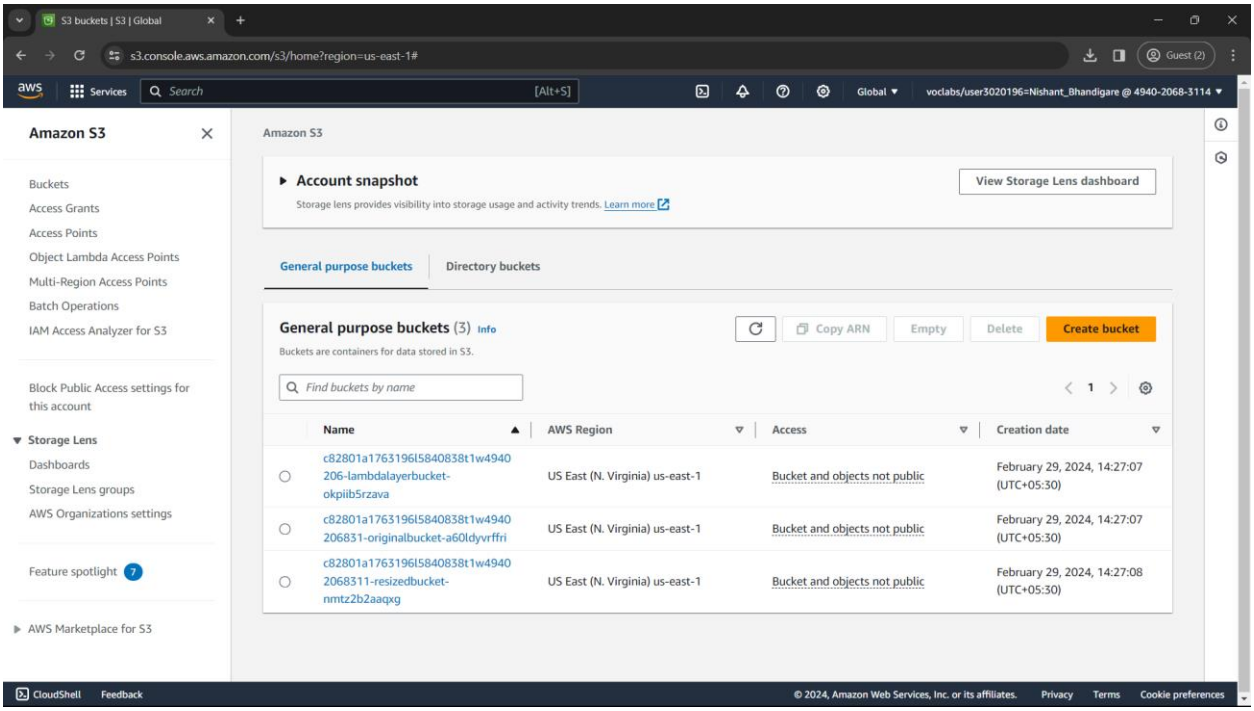
- large-image.jpg



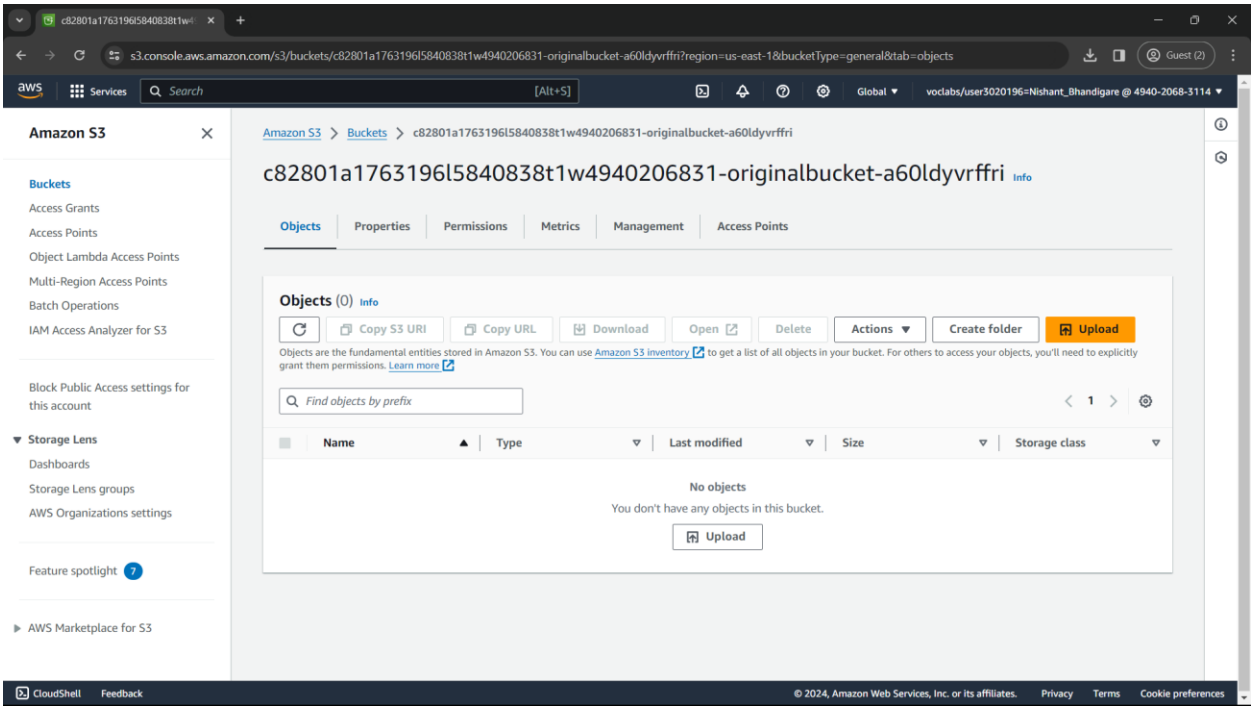
30. In the **AWS Management Console**, on the **Services** menu, enter **S3**. From the search results, choose **S3**.



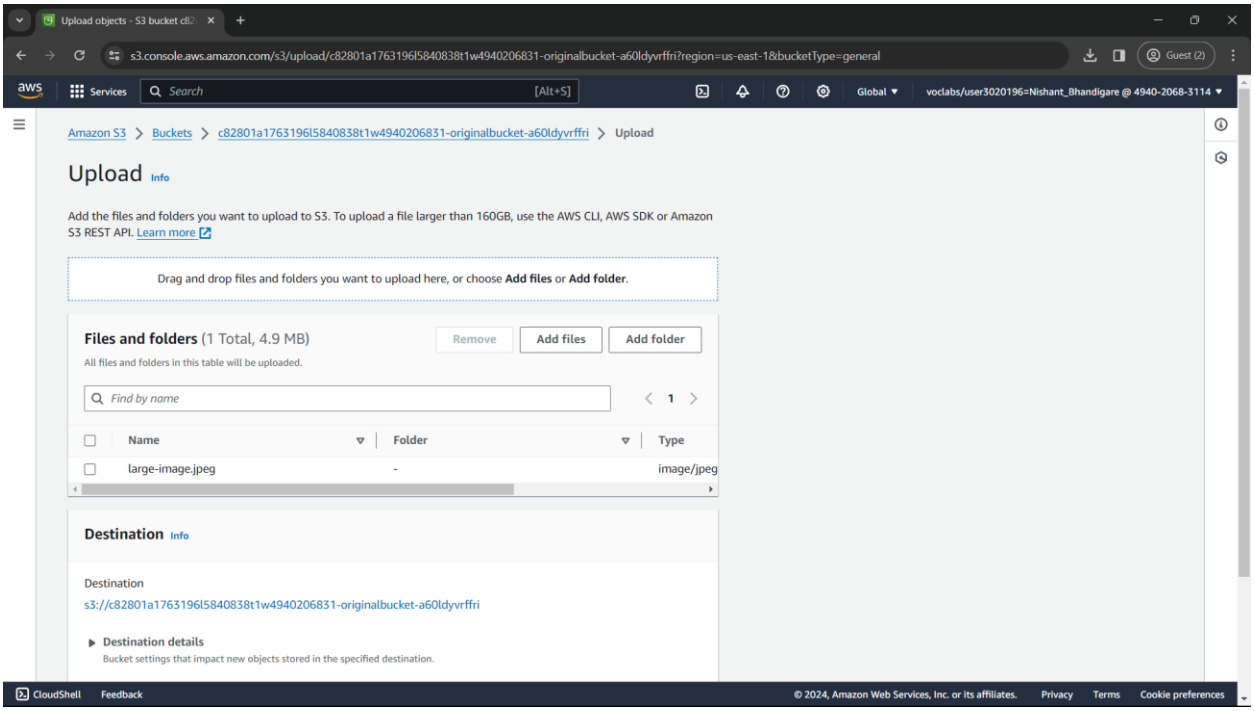
31. Choose the link for the bucket that has **original** in the name.



32. Choose **Upload**.

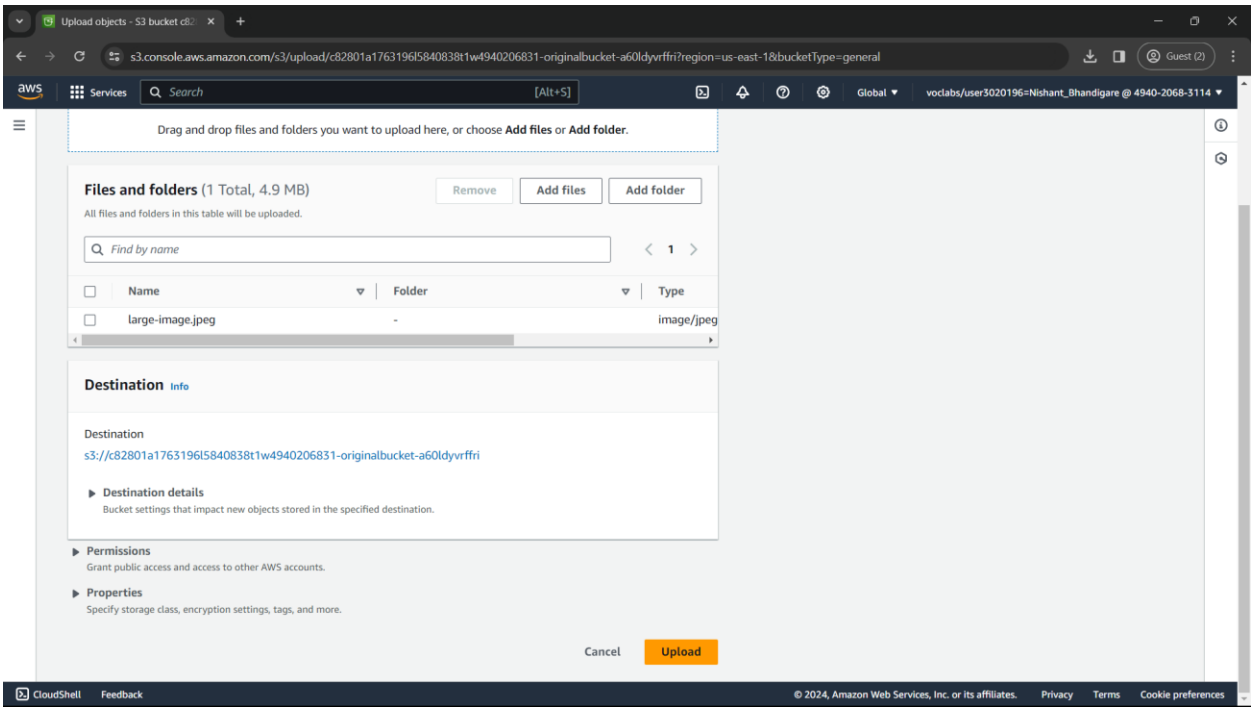


33. Choose **Add files**.
34. Choose the file that you downloaded.

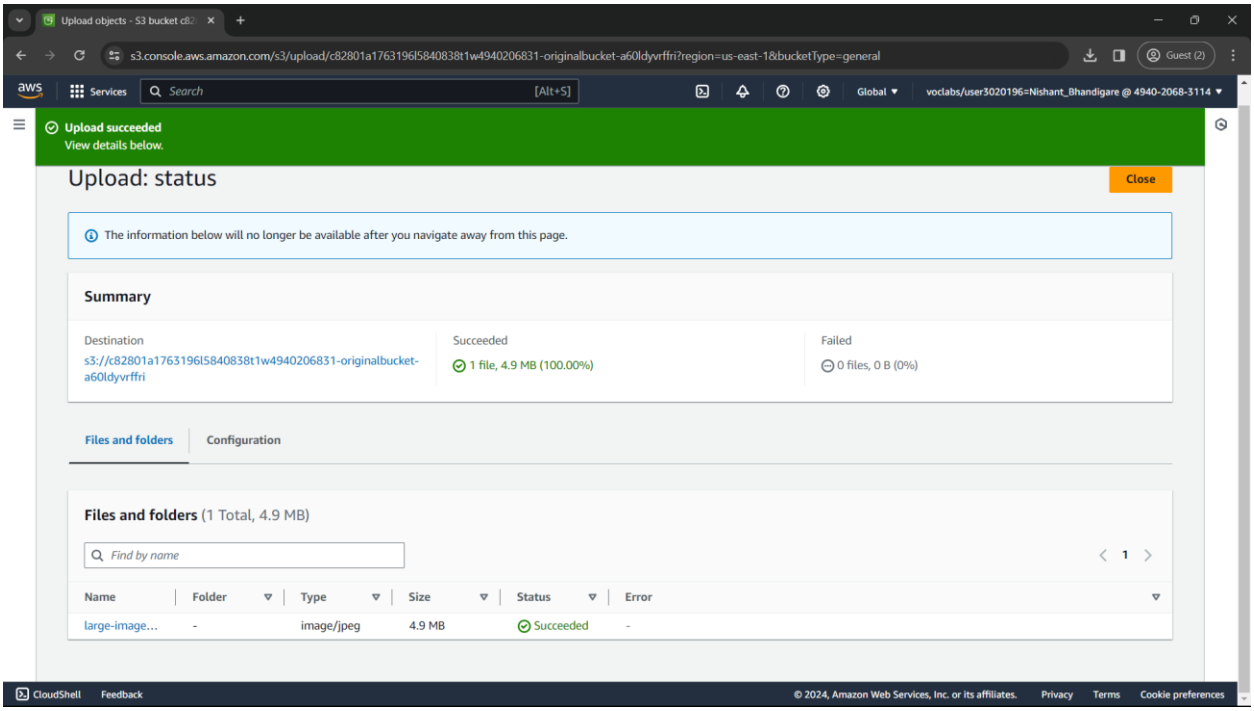


Choose **Upload**.

35. Your file is uploaded to the bucket.



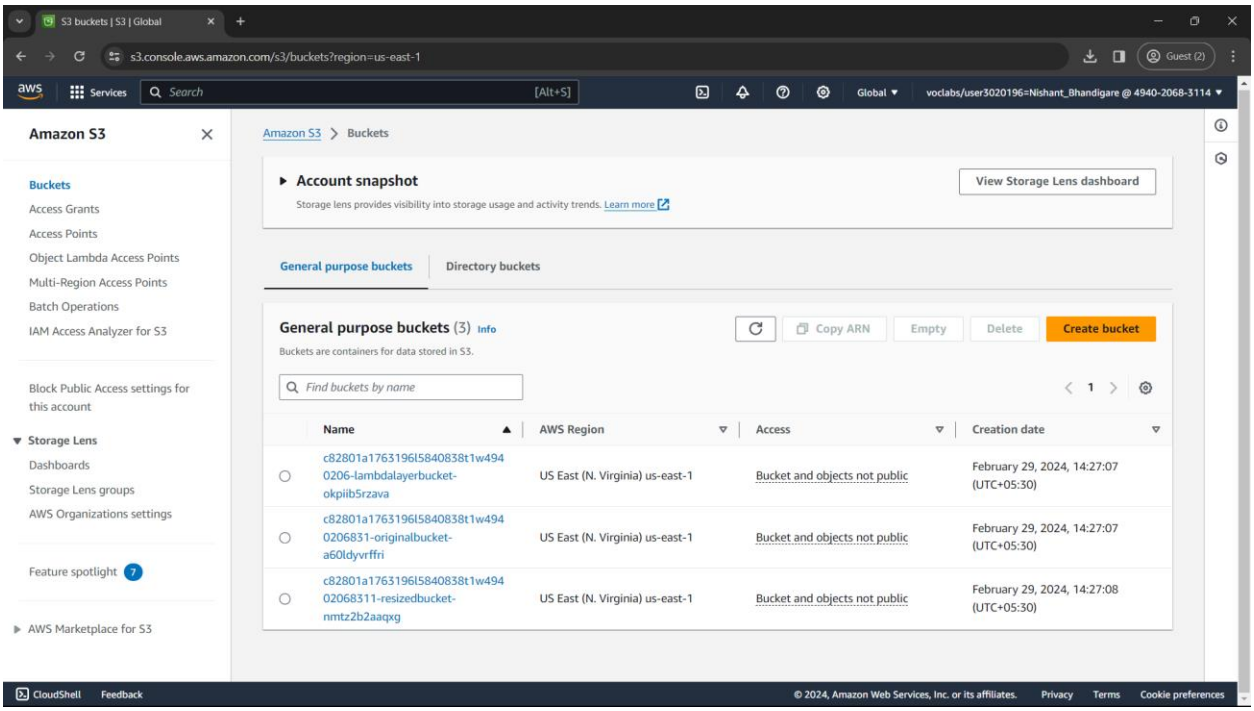
36. Choose **Close**.



If you are working on Task 4, you can now stop.

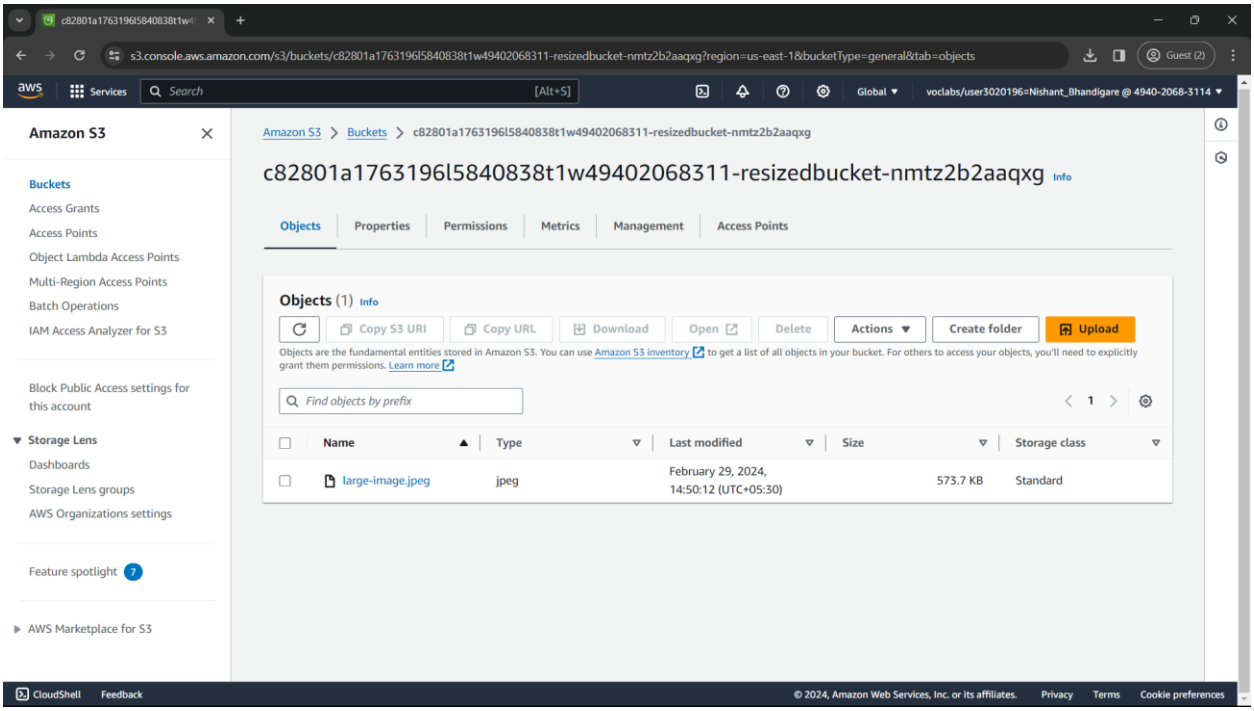
When your Lambda function runs correctly, the image file that you uploaded is reduced in size and placed in the S3 bucket that you specified when you set the environment variable for **RESIZED_BUCKET**.

37. Return to the **Buckets** section in the S3 Console.



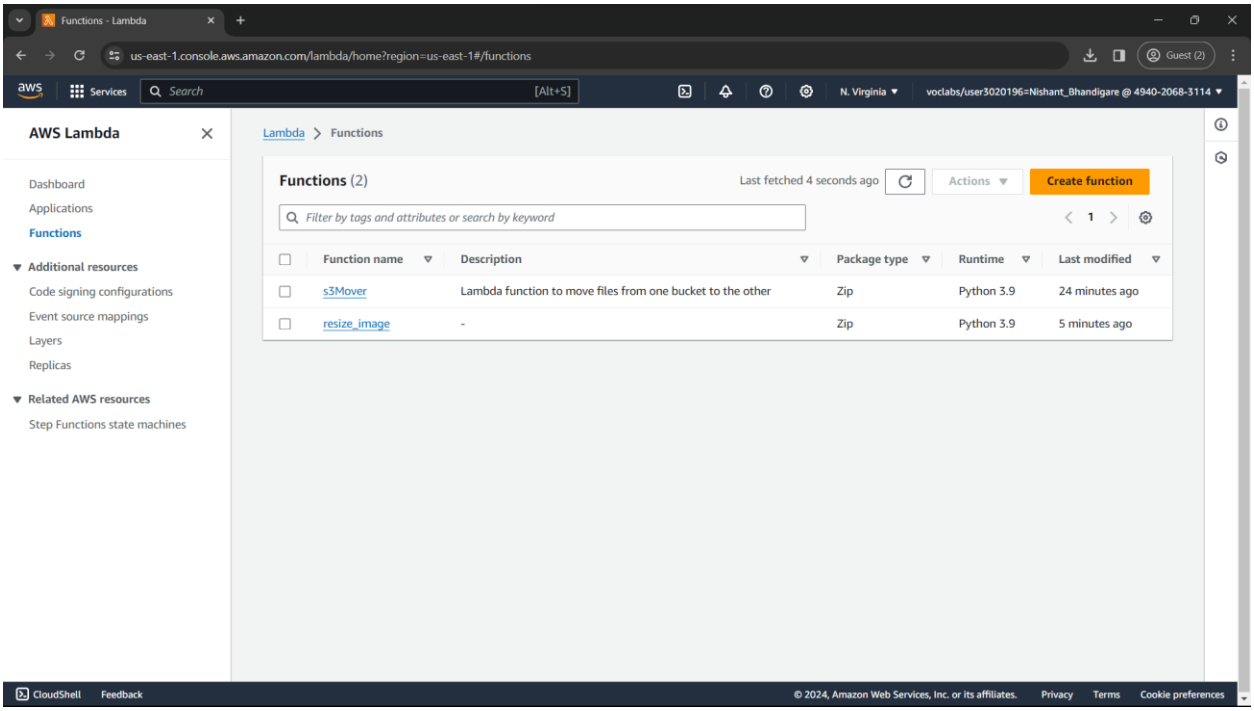
38. Choose the link for the bucket that has **resized** in the name.

Notice the file size. It's significantly reduced from the original size of 4.9 MB.

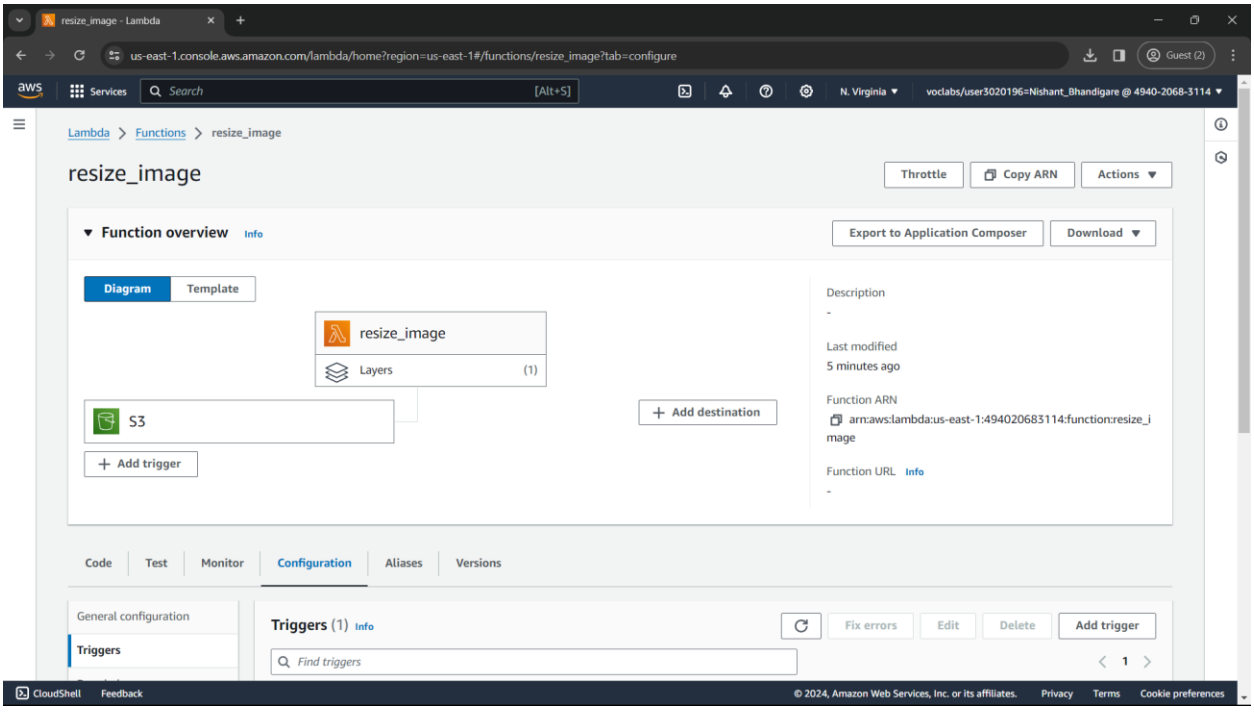


When you uploaded the image file, S3 initiated the **resize_image** Lambda function. Review the logs to see how your function performed.

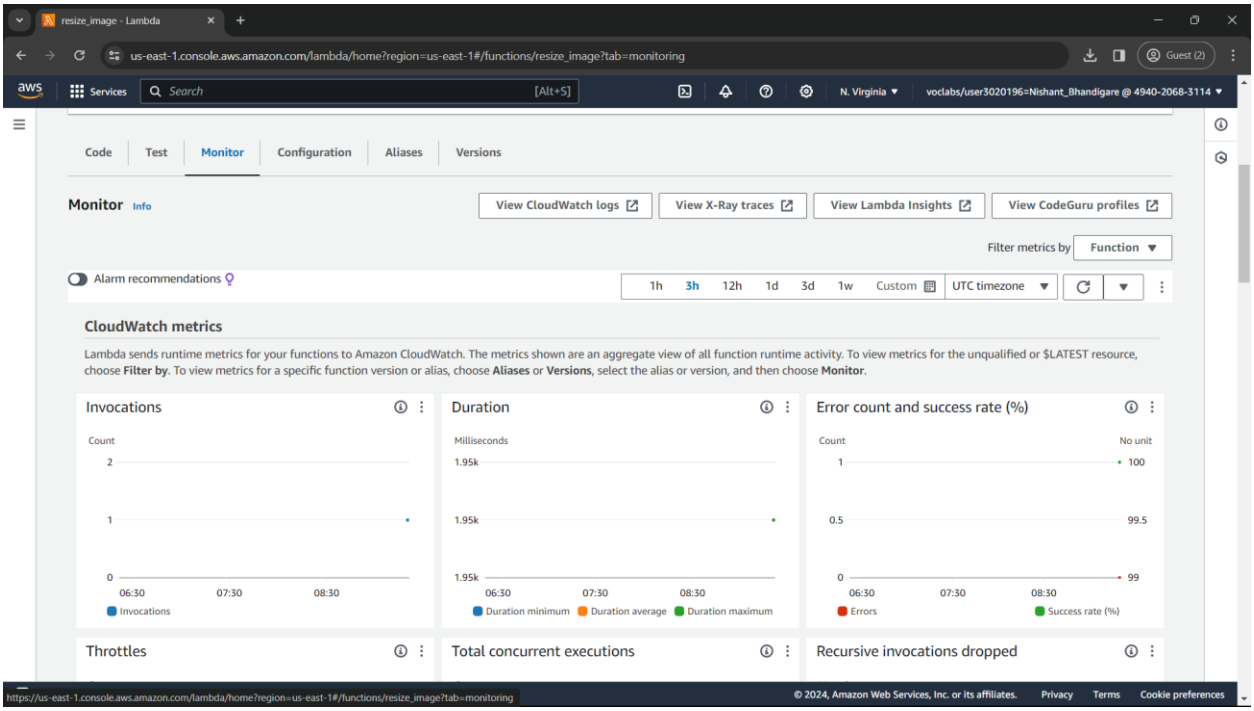
39. Navigate back to the Lambda console.



40. Choose the link for the **resize_image** function.



41. Choose the **Monitor** tab.



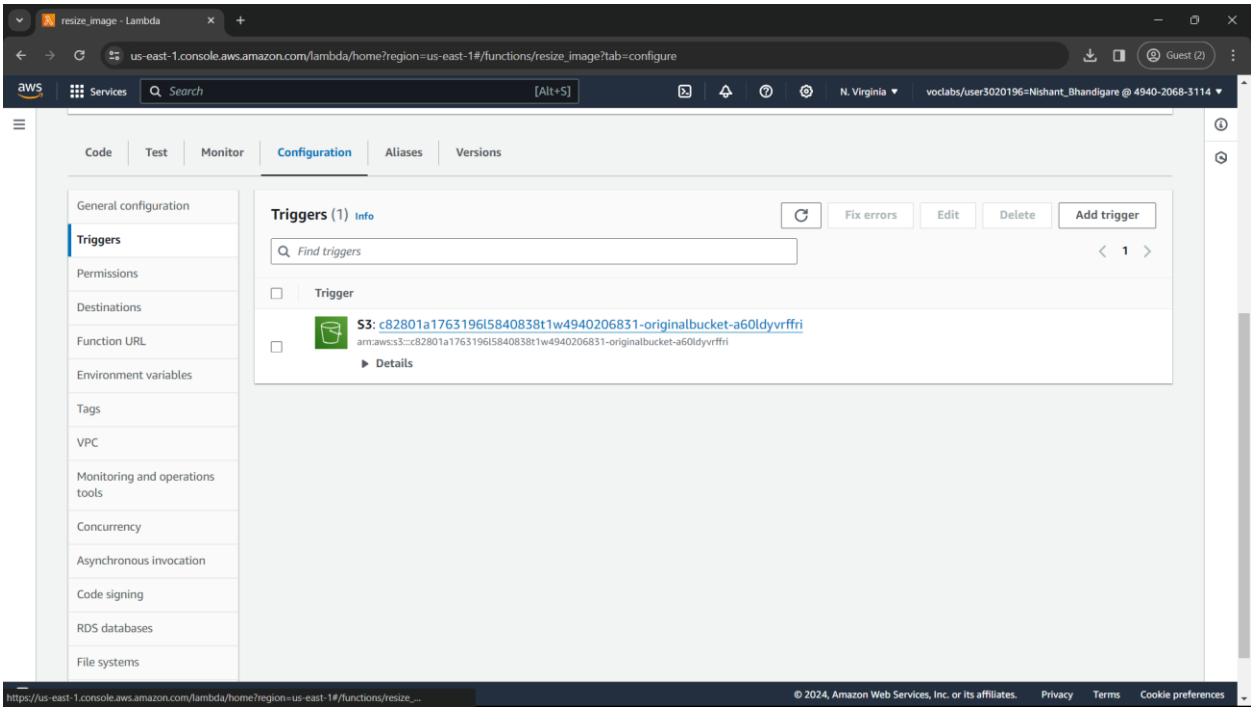
42. Choose **Logs**.

In the **Recent Invocations** table, choose a row to expand the details.

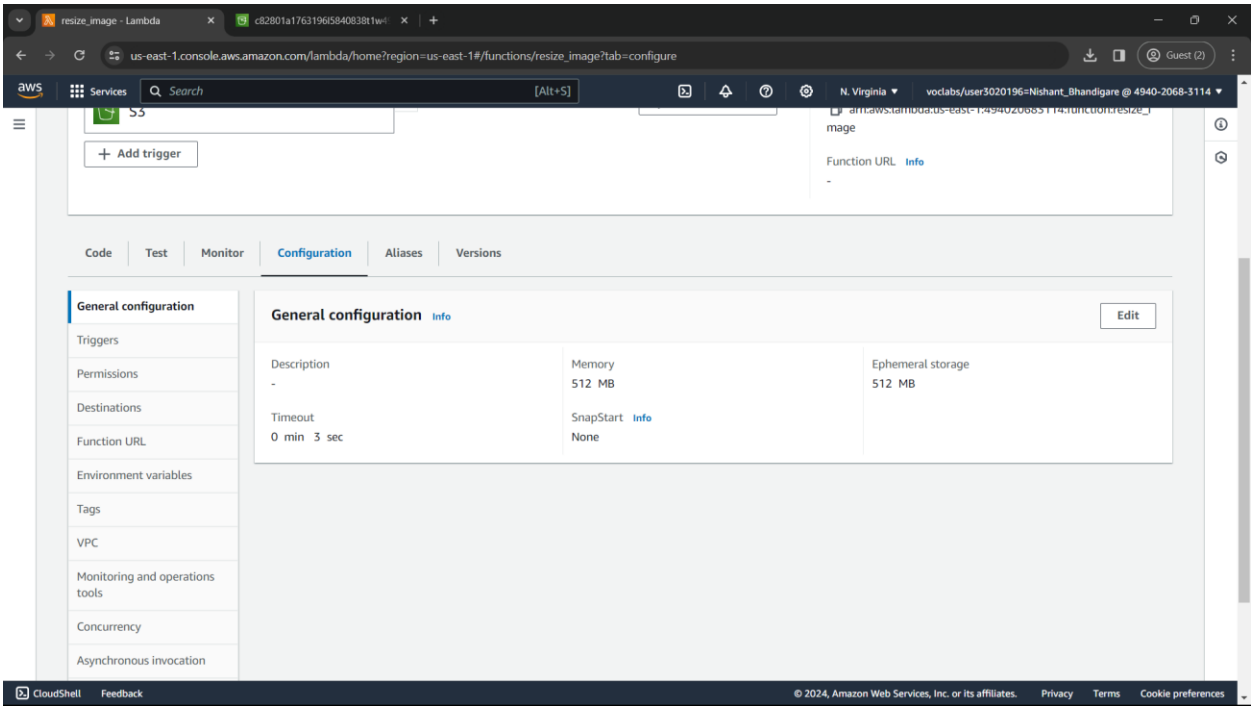
43. Notice the metrics that are recorded for each function invocation. You might have to wait for up to one minute for the data to be updated.
    The **DurationInMS** column tells you how long your function ran for this invocation.
    The first time that your Lambda function is invoked, the Lambda execution environment has to download your code and start a new execution environment. This process is called a cold start. The **@initDuration** metric in the **Recent invocation** details signifies the cold start time.
    Note the **MemorySetInMB** column. The amount of memory that's available to your Lambda function can be adjusted to affect the performance of your Lambda function. The amount of memory also determines the amount of virtual CPU available to a function. Adding more memory proportionally increases the amount of CPU, which increases the overall computational power available. If a function is CPU-, network- or memory-bound, then changing the memory setting can dramatically improve its performance.
    Notice how long it took your function to run. It could be faster. Adjust your Lambda function so that it runs faster.

Task 4: Optimizing Lambda function memory for performance

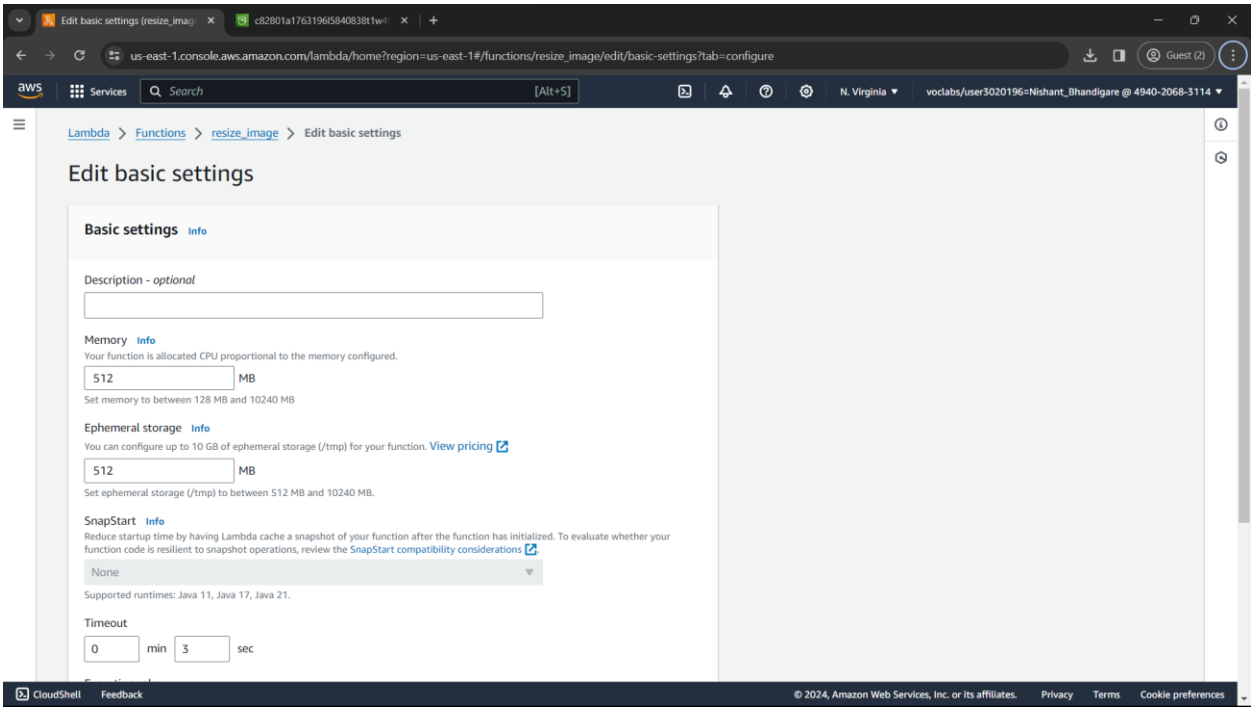
44. Choose the **Configuration** tab.



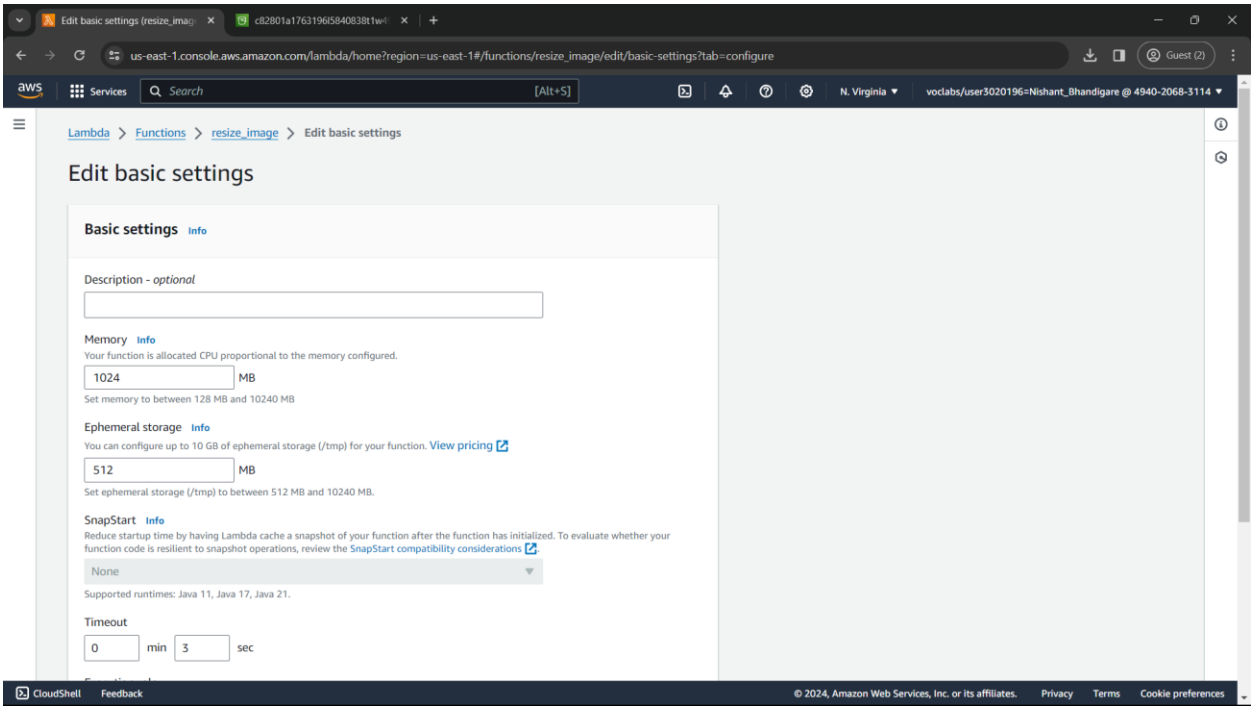
45. Choose **General Configuration**.



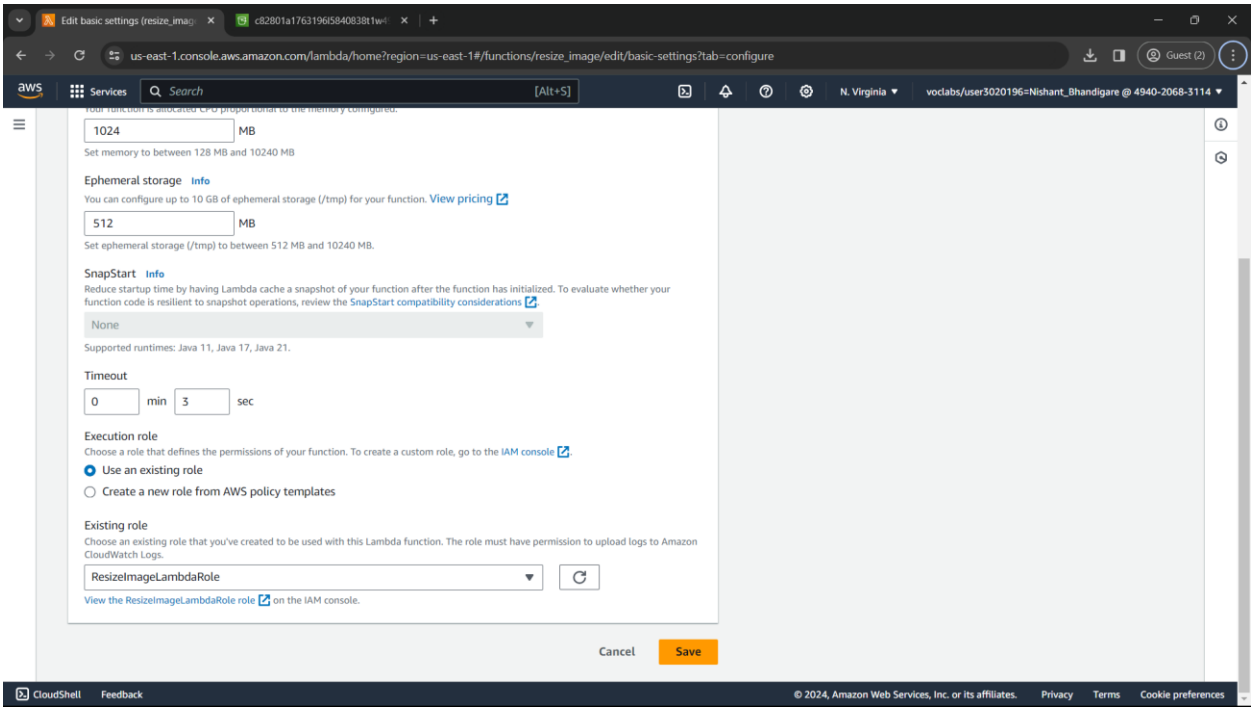
46. Choose **Edit**.



47. Adjust **Memory** to 1024 MB.

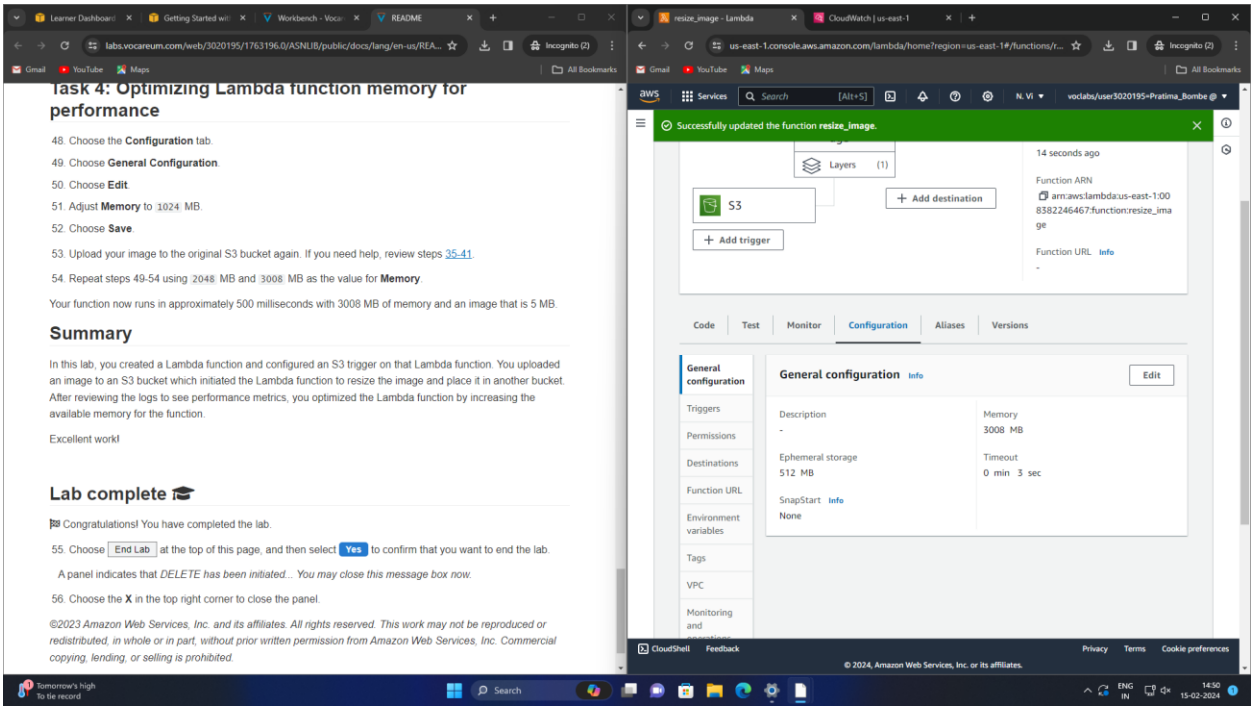


48. Choose **Save**.



49. Upload your image to the original S3 bucket again. If you need help, review steps 31-37.
50. Repeat steps 45-50 using 2048 MB and 3008 MB as the value for **Memory**.

Your function now runs in approximately 500 milliseconds with 3008 MB of memory and an image that is 5 MB.



Conclusion: In conclusion, the journey to study and implement Platform as a Service (PaaS) using AWS or Microsoft Azure has provided valuable insights into the capabilities and advantages of cloud-native application development. Throughout this endeavor, we have delved into the foundational principles of PaaS offerings provided by AWS and Azure, understanding their role in simplifying application development, deployment, and management. By leveraging services like AWS Elastic Beanstalk, AWS Lambda, Azure App Service, and Azure Functions, we have experienced firsthand the agility and scalability offered by PaaS solutions, enabling rapid iteration and innovation in application development. Furthermore, the experiment has underscored the importance of selecting the appropriate PaaS services based on application requirements, as well as the significance of seamless integration with other cloud services for enhanced functionality. Overall, the study and implementation of PaaS using AWS or Microsoft Azure have equipped us with the knowledge and skills to streamline the application development lifecycle, accelerate time-to-market, and drive business growth in the cloud era.