CSC4160 Assignment 3

Report

Name: Xu Boshi

Student ID: 122040075

Introduction:

This assignment focuses on deploying a machine learning model using AWS Lambda and API Gateway, leveraging Docker, Amazon ECR, and load testing, while using the IRIS dataset to explore the serverless deployment process and analyze cold starts.

Requirements:

- `README.md`
- `lambda_function.py` (your implementation of the Lambda function). here
  - Extracts the `values`
  - Calls the predict function
  - Return the prediction result
- Provide the CloudWatch log event for one of your Lambda function invocations. here
- A screenshot of a successful request and response using `curl` or `Invoke-WebRequest`, showing the correct prediction output. here
- `performance_analysis.ipynb`, including: here
  - Figure of the line graph, cold start histogram, and warm request histogram. (0.5 point)
- `test_locust_logs.txt` (Locust load test logs) [In the zip file]

1. Environment Setup

Windows Environment:

> OS: Windows 10 Professional 22H2
> CPU: AMD Ryzen 7 7840HS with Radeon 780M Graphics
> Memory: 32 GB

Install docker by running `Docker Desktop Installer.exe` and check the version

```
docker --version
```

Change settings of docker by unselecting `Use the WSL 2 based engine`.
Install ASW CLI by running `AWSCLIV2.msi`

In the folder with Dockerfile, run the command to create a Docker image:

```
docker build -t iris_image .
```

Run the Docker container locally

```
docker run -it --rm -p 8080:8080 iris_image:latest
```

Run test.py

```
Json output from test.py

# Result
{'statusCode': 200, 'body': '{"prediction": [0]}'}
{'statusCode': 200, 'body': '{"prediction": [2]}'}
```

It should be able to prove that the environment is running properly.

2. AWS Setup

Press `Start Lab`, find `AWS CLI` from `AWS Details` and click `Show`, information is storage in `AWS CLI`.

Create key file at:

```
C:\Users\Admin\.aws\credentials

# credentials
[default]
region=us-east-1
aws_access_key_id=ASIASCF2RHR2CISNNZ2K
aws_secret_access_key=9hRyz1P10fFpV4/runABow2Lxm8cIOjengbV1qRp
aws_session_token=IQoJb3JpZ2luX2VjEA4aCXVzLXdl....
```

3. lambda handler function

```
import pickle
import json

# load file
filename = 'iris_model.sav'
model = pickle.load(open(filename, 'rb'))

def predict(features):
    return model.predict(features).tolist()
```

```
def lambda_handler(event, context):
    try:
        # check body
        if 'body' in event:
            body = json.loads(event['body'])
            values = body.get('values', None)
        else:
            values = event.get('values', None)

        if values is None:
            return {
                'statusCode': 400,
                'body': json.dumps('Error: Missing input values.')
            }

        # get proper result
        features = values

        # run predict func
        prediction = predict(features)

        # Return result
        return {
            'statusCode': 200,
            'body': json.dumps({'prediction': prediction})
        }

    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps(f'Error: {str(e)}')
        }
```

4. ECR repository setup

Locally: Create a container image repository

```
aws ecr create-repository --repository-name iris-registry
```

Configure Authentication

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 142125120628.dkr.ecr.us-east-1.amazonaws.com
```

Get the image ID

```
docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
iris_image   latest    1b3863e723a8   2 hours ago   838MB
```

Tag and push the image to ECR

```
docker tag 1b3863e723a8 142125120628.dkr.ecr.us-east-1.amazonaws.com/iris-
registry:latest
docker push 142125120628.dkr.ecr.us-east-1.amazonaws.com/iris-registry:latest
```

5. Lambda function creation

In the AWS console, create a Lambda function using the container image you just built and select LabRole as the execution role

Create a test and use the above events:

```
{
  "values": [[5.9, 3.0, 5.1, 2.3]]
}
```

Successful, Output:

```
{
  "statusCode": 200,
  "body": "{\"prediction\": [2]}"
}
```

6. API Gateway Configuration

(1) Use the API Gateway to create a REST API for the Lambda function through the AWS console All defaults Output information:

INFO	Content
API name	as3_api
API ID	u3nt600wp3
Region	us-east-1

(2). Create a Resource and POST Method In the left-hand navigation pane, find `Resources` which is used to define the resource path for the API.
Click the `Actions` button and select `Create Resource`
Name the resource, for example, `/predict`
Check `Configure as a proxy resource`, then click `Create Resource`
Select the newly created `/predict` resource.
Click `Actions` and choose `Create Method`
Select `POST` as the method type, then choose `Lambda Function`

(3). Select the Lambda Function
Select the correct region and the Lambda function to be tested.
Leave other options as default and create the method.
(4). Test the API
Select the `POST` method and click the `Test` button. Input the test data:

```
{
  "values": [[5.9, 3.0, 5.1, 2.3]]
}
```

You should see a return result similar to before.
(5). Deploy the API
Click `Actions` and select `Deploy API`
Create a new Deployment Stage, name it `dev`, and `Deploy`
After deployment, API Gateway will generate a URL. In the left navigation pane, select `Stages`, then choose the recently deployed `dev` stage and `Deploy`
You can find the API URL under `Invoke URL`

```
https://u3nt600wp3.execute-api.us-east-1.amazonaws.com/dev/predict
```

Test the API on the local machine by triggering the Lambda function through this URL.

```
Invoke-WebRequest -Method Post -Uri "https://u3nt600wp3.execute-api.us-east-
1.amazonaws.com/dev/predict" `
 -Headers @{ "Content-Type" = "application/json" } `
 -Body '{"values": [[5.9, 3.0, 5.1, 2.3]]}'
```

Return correct information:

```
PS D:\Python_code\CSC4160\Assignment3> Invoke-WebRequest -Method Post -Uri "https://u3nt600wp3.execute-api.us-east-1.ama
zonaws.com/dev/predict" `
>>  -Headers @{ "Content-Type" = "application/json" } `
>>  -Body '{"values": [[5.9, 3.0, 5.1, 2.3]]}'


StatusCode        : 200
StatusDescription : OK
Content           : {"statusCode": 200, "body": "{\"prediction\": [2]}"}
RawContent        : HTTP/1.1 200 OK
                    Connection: keep-alive
                    x-amzn-RequestId: 57dbbf86-5e28-41b0-b02b-dbc1716e3802
                    x-amz-apigw-id: f9CpDGihoAMEN4A=
                    X-Amzn-Trace-Id: Root=1-671517d3-556f7a9d6f194ccc1ae80b8c;Parent=2e4f...
Forms             : {}
Headers           : {[Connection, keep-alive], [x-amzn-RequestId, 57dbbf86-5e28-41b0-b02b-dbc1716e3802], [x-amz-apigw-i
                    d, f9CpDGihoAMEN4A=], [X-Amzn-Trace-Id, Root=1-671517d3-556f7a9d6f194ccc1ae80b8c;Parent=2e4fabbebc5
                    0fe25;Sampled=0;Lineage=1:97dc3861:0]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : System.__ComObject
RawContentLength  : 52
```

7. Load Testing

Environment Setup Run the following to install the required package:

```
mkdir locust_logs
```

In the root directory of the assignment folder, run:

```
locust -f locustfile.py --host https://u3nt600wp3.execute-api.us-east-
1.amazonaws.com --users 10 --spawn-rate 5 --run-time 60s --csv "locust_logs/test"
--csv-full-history --html "locust_logs/test_locust_report.html" --logfile
"locust_logs/test_locust_logs.txt" --headless
```

The data will output in locust_logs folder

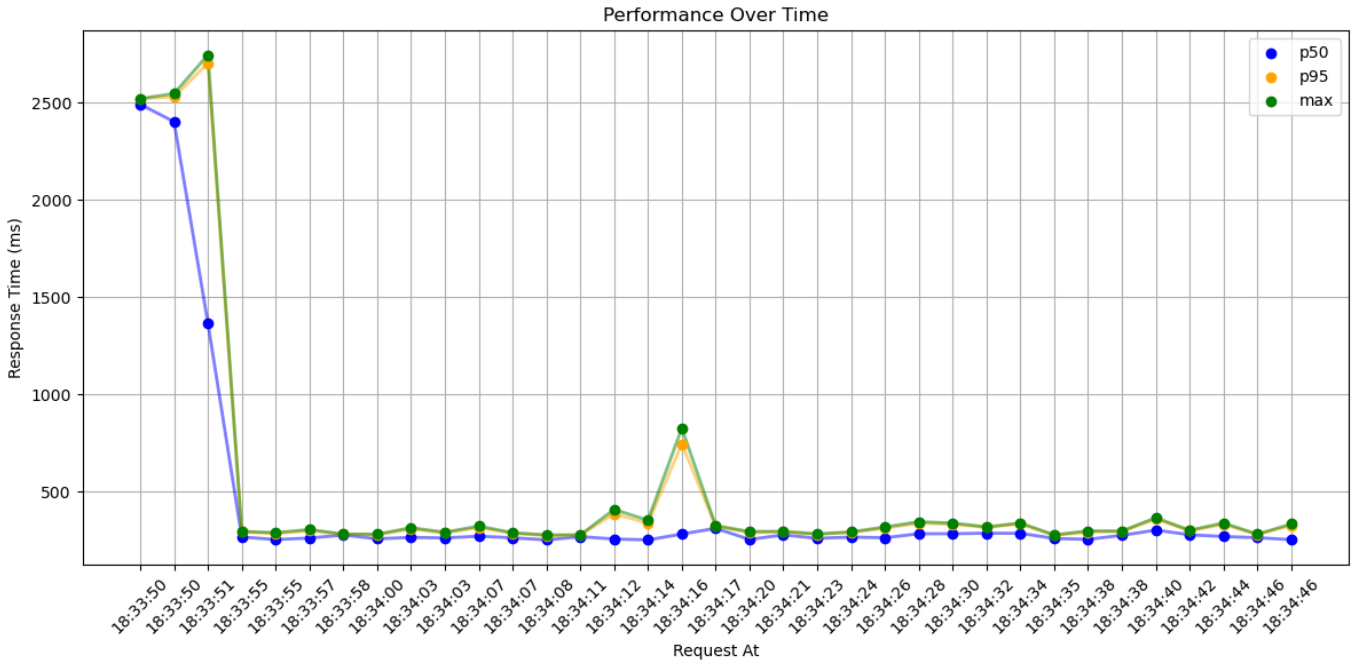
Type	Name	# reqs	# fails	Avg	Min	Max	Med	req/s	failures/s
POST	/dev/predict	135	0(0.00%)	571	328	2770	340	2.29	0.00
	Aggregated	135	0(0.00%)	571	328	2770	340	2.29	0.00

Response time percentiles (approximated)

Type	Name	50%	66%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%	# reqs
POST	/dev/predict	340	340	920	960	1100	1600	2100	2300	2800	2800	2800	135
	Aggregated	340	340	920	960	1100	1600	2100	2300	2800	2800	2800	135

8. Analysis

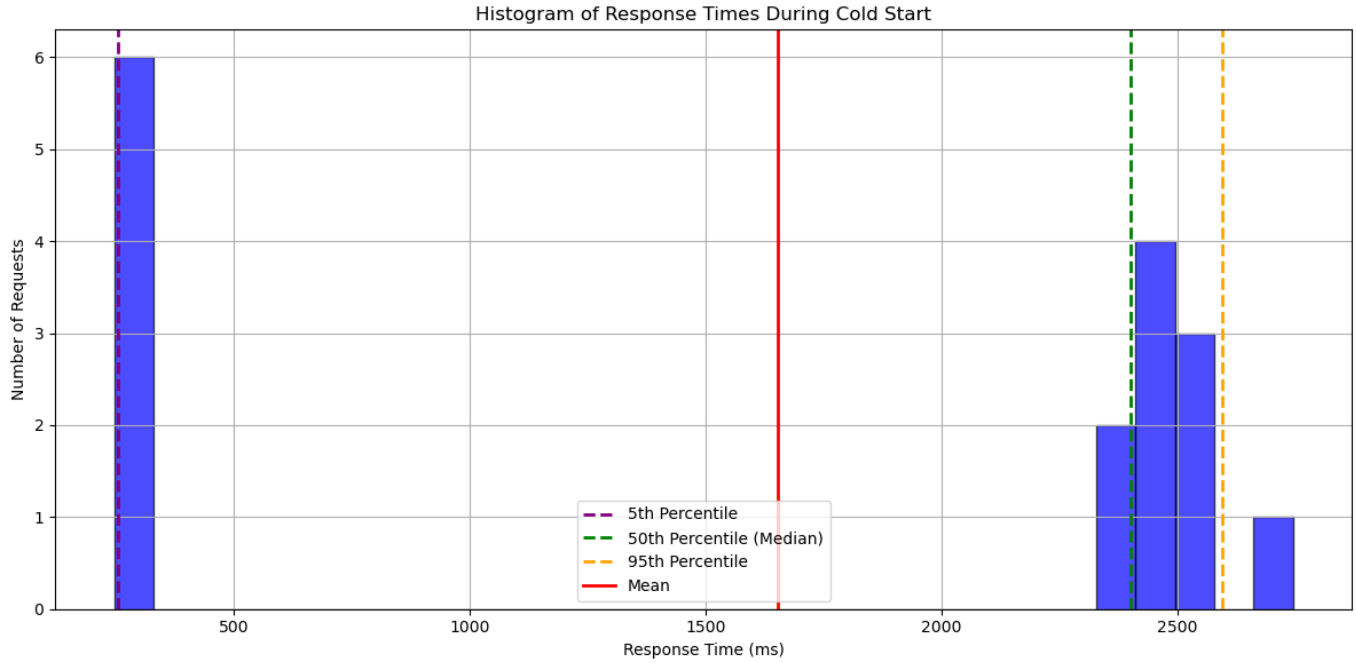
Run charts of `performance_analysis.ipynb` in Jupyter Notebook Results: First Chart: Performance Over Time (P50, P95, Max)



This chart shows the changes in request response time. The three indicators in the chart are p50 (50th percentile), p95 (95th percentile), and max (maximum response time). At the beginning of the chart, the response time is very high, with the maximum value reaching around 2500 ms. This is usually due to cold start delays, where the first request often takes longer to complete. Then, the response time drops rapidly to around 300 ms and remains stable for most of the time. This indicates that after the cold start, the system has entered a "warm state," and the response time has significantly reduced. There is a small spike in the middle

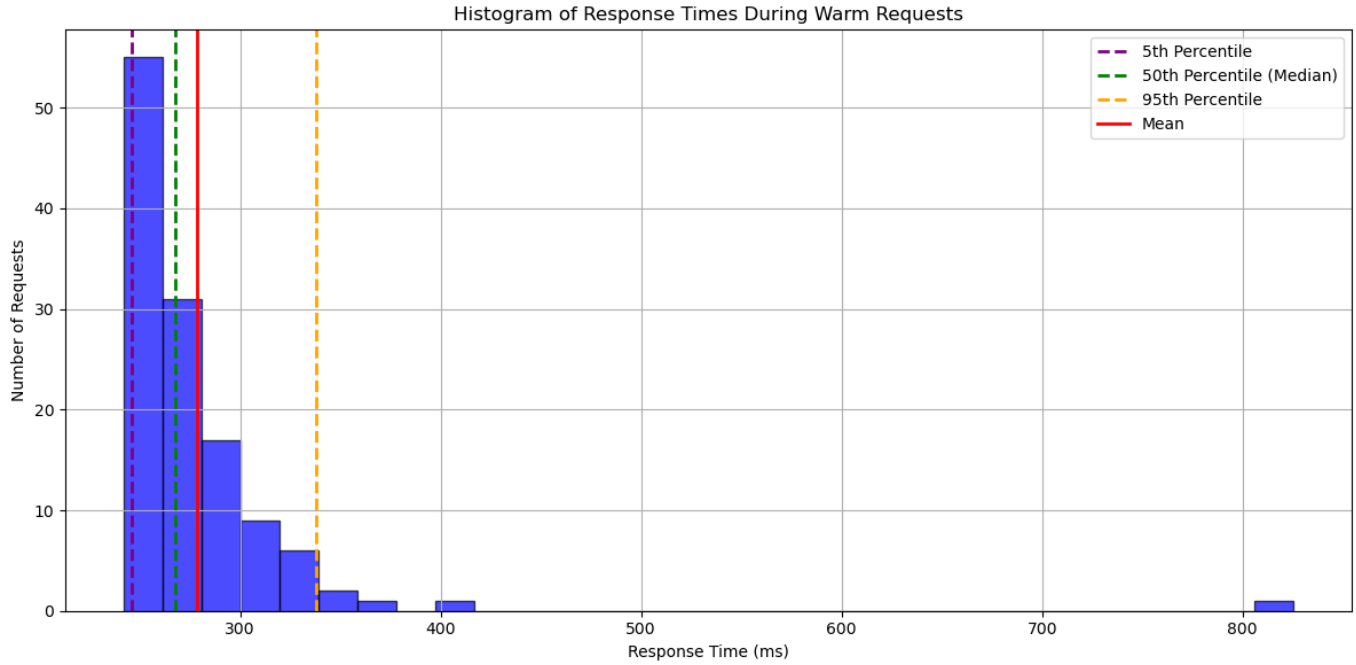
where the maximum response time slightly increases, possibly due to resource scheduling or other temporary factors. Overall, this chart shows the performance of the system from cold start to stable operation, with response times dropping sharply after the cold start and remaining at a low level.

Second Chart: Histogram of Response Times During Cold Start



This chart shows the distribution of request response times during a cold start. It is evident that the majority of requests have response times clustered around 2500 ms. Due to the nature of cold starts, the first invocation requires initializing resources, which leads to longer response times. The delay mainly comes from starting the function environment, loading dependencies, and initialization steps. As a result, there is a clear peak in the graph, indicating that all requests experience longer response times, typical of the cold start phase. The p50 (50th percentile) and p95 (95th percentile) are both in the higher range, meaning that most requests have response times far exceeding the expected values during regular operation. The mean is close to these percentiles, indicating that performance during a cold start is inconsistent, with a wide variation in response times.

Third Chart: Histogram of Response Times During Warm Requests



In contrast, the response times during a warm start are significantly improved. Most requests have response times clustered around 300 ms, with a distinct peak in the graph, showing that when the system is in a warm state, response times are stable. Unlike a cold start, a warm start does not require re-initializing resources, allowing the system to quickly process requests, resulting in much lower response times. Both the p50 and p95 values are substantially lower and close to the mean, indicating that the response times during a warm start are much more consistent, with almost no major fluctuations. This chart illustrates the system's high efficiency in a warm state, showing that once the system reaches steady operation, user experience improves significantly.

Overall Summary: During a cold start, performance is poor, with a significant delay in response times. In a warm start, performance improves considerably, with shorter and more stable response times. Overall, the system shows a marked improvement in response times as it transitions from a cold start to stable operation, with much higher efficiency once in a warm state.

9. Q&A

AWS Lambda Function (0.5 point):

What is the role of a Lambda function in serverless deployment? How does the lambda_handler function work to process requests?

> Answer: A Lambda function is the core component of AWS's serverless architecture, allowing users to execute code without provisioning or managing servers. It runs in response to various triggers, such as HTTP requests or S3 bucket events. The lambda_handler function serves as the entry point for the code execution. It receives event data and context information, processes the request based on the logic defined, and returns a response to the invoking service or user.

API Gateway and Lambda Integration (0.5 point):

Explain the purpose of API Gateway in this deployment process. How does it route requests to the Lambda function?

> Answer: API Gateway acts as a front-door interface that allows external clients to interact with backend services such as Lambda. It processes incoming HTTP requests, routes them to the appropriate Lambda function, and returns the function's output to the client. API Gateway maps HTTP methods (e.g., GET, POST) to specific Lambda functions, handling request transformations, authorization, and error management.

ECR Role (0.5 point):

What is the role of ECR in this deployment? How does it integrate with Lambda for managing containerized applications?

> Answer: Amazon Elastic Container Registry (ECR) stores container images, which Lambda can execute when deployed as a containerized application. Instead of uploading code directly to Lambda, ECR allows developers to build, push, and manage Docker images. Lambda integrates with ECR by pulling the specified container image from the repository during function execution, enabling more complex application dependencies and environments.

Cold Start Versus Warm Requests (1 Point):

Provide your analysis comparing the performance of requests during cold starts versus warm requests (based on the line graph and histograms you obtained in performance_analysis.ipynb). Discuss the differences in response times and any notable patterns observed during your load testing.

> Answer: Cold start refers to the time it takes for AWS Lambda to initialize a new instance of the function when no previous instance is available. During cold starts, there is a noticeable delay in response times due to the overhead of initializing the environment (e.g., loading the container, setting up the execution context). Warm requests, on the other hand, occur when a previously initialized Lambda instance is reused, resulting in faster response times. In the graphs from the performance analysis, cold start response times are significantly higher, with clear spikes compared to the more consistent and lower times seen with warm requests. This pattern is especially noticeable under high traffic when the system frequently initiates new cold starts.

Implications and Strategies (0.5 Point):

Discuss the implications of cold starts on serverless applications and how they affect performance. What strategies can be employed to mitigate these effects?

> Answer: Cold starts can lead to inconsistent response times, which can impact user experience in latency-sensitive applications. They can be especially problematic for high-throughput or real-time systems. Strategies to mitigate cold starts include using provisioned concurrency, which keeps pre-warmed instances ready to handle requests, reducing initialization delays. Additionally, optimizing the Lambda function's package size and runtime initialization can reduce cold start latency. Another approach is to implement a warm-up process, periodically invoking the function to ensure instances remain active.

10. Conclusion

In conclusion, this report demonstrates the successful deployment of a machine learning model using AWS Lambda and API Gateway, focusing on Docker integration and load testing with the IRIS dataset. The environment setup, from Docker installation to AWS configuration, highlights key steps for running and testing serverless applications. The experiment reveals the performance implications of cold starts, where initial response times can be significantly delayed compared to warm starts, leading to performance variability.

Through visual data analysis, the report confirms that once Lambda functions are initialized, response times stabilize, ensuring efficient and fast processing for warm requests. This report emphasizes the importance of optimizing cold start performance for real-world serverless applications and provides insights into the benefits of serverless deployment. By analyzing both cold and warm requests, it is clear that serverless architecture, while offering convenience and scalability, requires careful consideration of its trade-offs, particularly when dealing with time-sensitive applications.

========== This is the End of Assignment 3 Report ==========