CSC4160 Assignment2

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

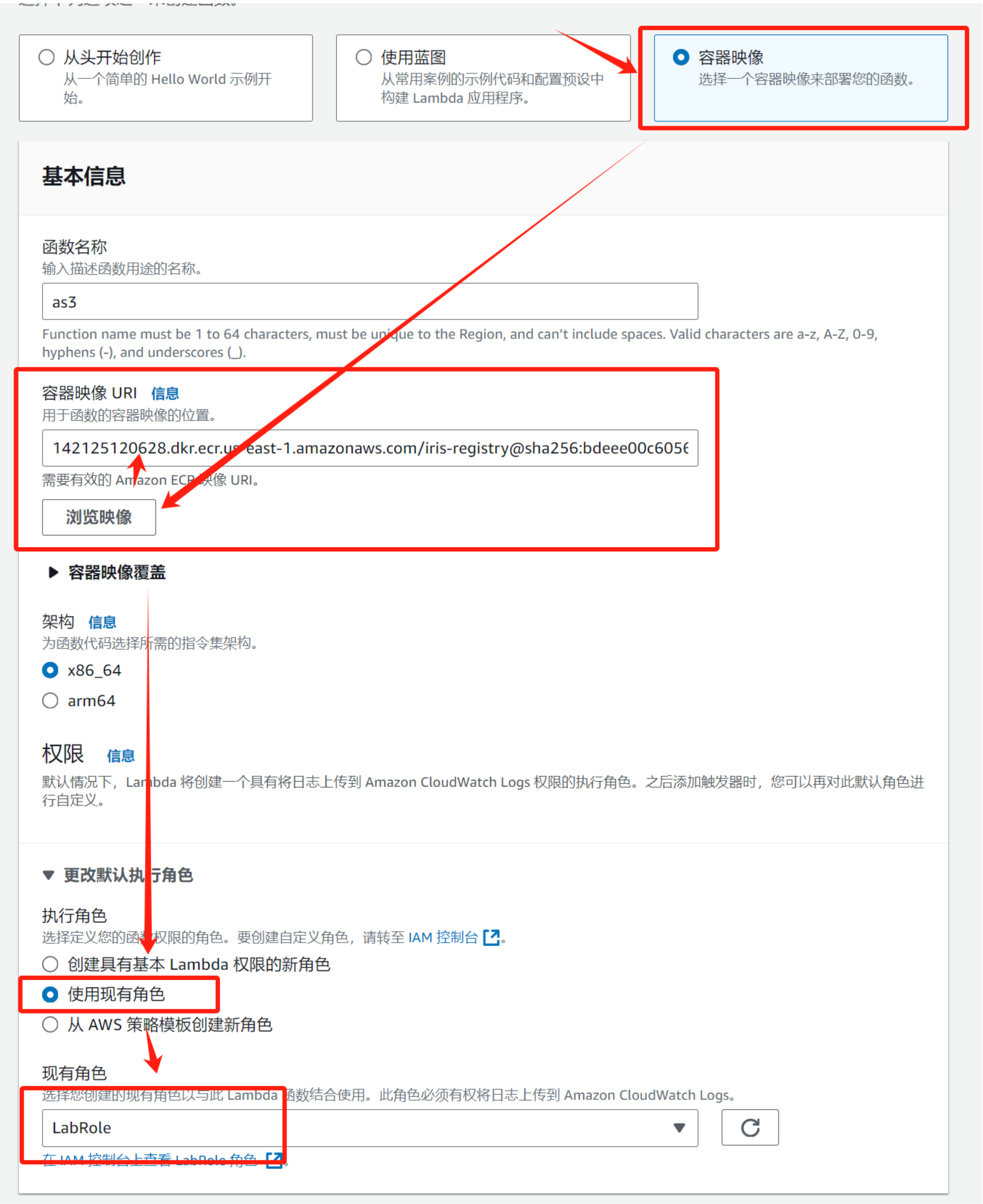
```
docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
iris_image   latest    1b3863e723a8   2 hours ago   838MB
```

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



○ 从头开始创作
从一个简单的 Hello World 示例开始。

○ 使用蓝图
从常用案例的示例代码和配置预设中构建 Lambda 应用程序。

● 容器映像
选择一个容器映像来部署您的函数。

基本信息

函数名称
输入描述函数用途的名称。
as3
Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

容器映像 URI 信息
用于函数的容器映像的位置。
142125120628.dkr.ecr.us-east-1.amazonaws.com/iris-registry@sha256:bdeee00c605
需要有效的 Amazon ECR 映像 URI。

架构 信息
为函数代码选择所需的指令集架构。
● x86_64
○ arm64

权限 信息
默认情况下，Lambda 将创建一个具有将日志上传到 Amazon CloudWatch Logs 权限的执行角色。之后添加触发器时，您可以再对此默认角色进行自定义。

▼ 更改默认执行角色

执行角色
选择定义您的函数权限的角色。要创建自定义角色，请转至 IAM 控制台。
○ 创建具有基本 Lambda 权限的新角色
● 使用现有角色
○ 从 AWS 策略模板创建新角色

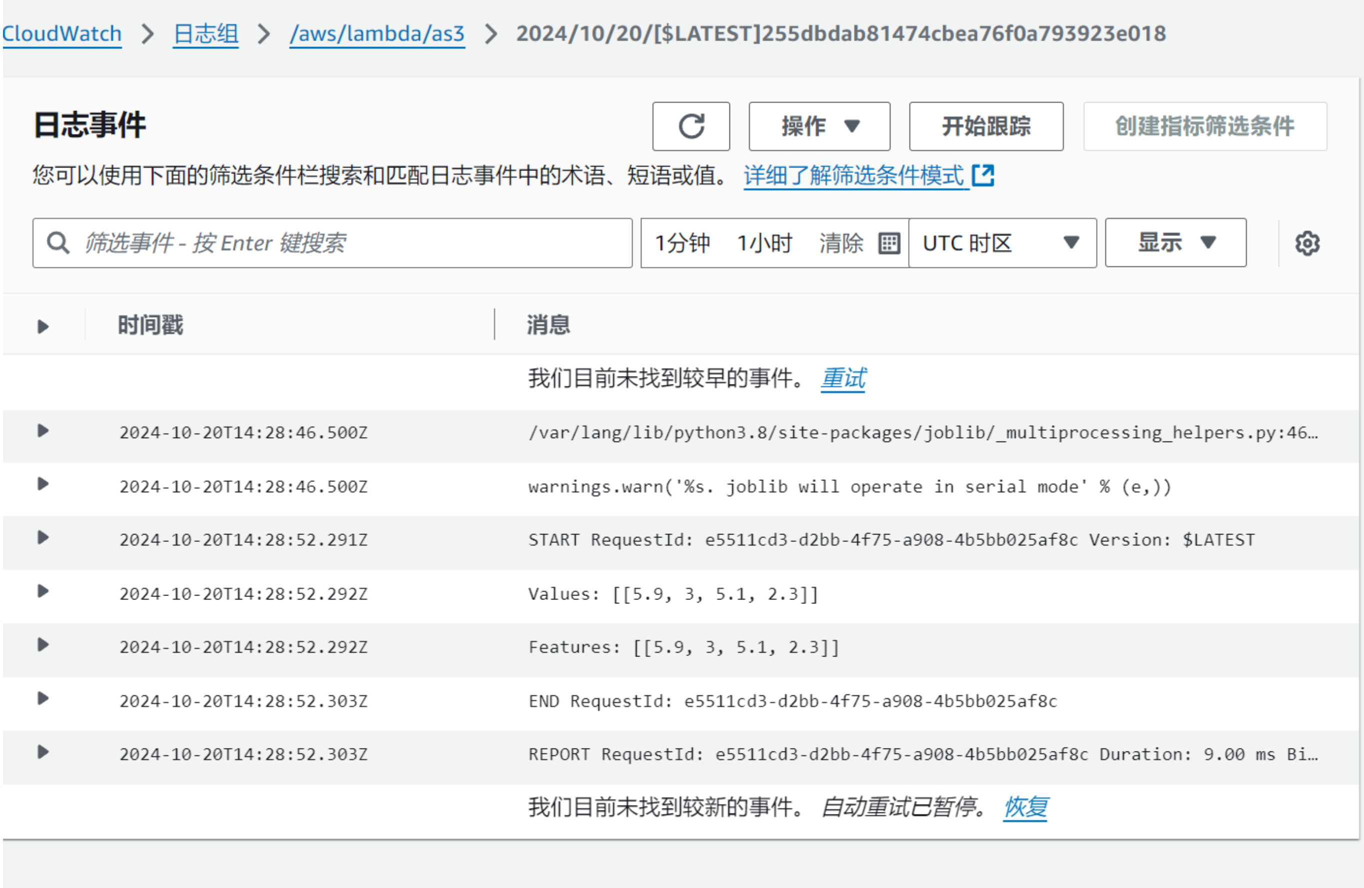
现有角色
选择您创建的现有角色以与此 Lambda 函数结合使用。此角色必须有权将日志上传到 Amazon CloudWatch Logs。
LabRole
在 IAM 控制台上查看 LabRole 角色。

Create a test and use the above events:

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

Successful, Output:

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



CloudWatch > 日志组 > /aws/lambda/as3 > 2024/10/20/[$LATEST]255dbdab81474cbea76f0a793923e018

日志事件

您可以使用下面的筛选条件栏搜索和匹配日志事件中的术语、短语或值。 详细了解筛选条件模式

时间戳	消息
	我们目前未找到较早的事件。 重试
2024-10-20T14:28:46.500Z	/var/lang/lib/python3.8/site-packages/joblib/_multiprocessing_helpers.py:46…
2024-10-20T14:28:46.500Z	warnings.warn('%s. joblib will operate in serial mode' % (e,))
2024-10-20T14:28:52.291Z	START RequestId: e5511cd3-d2bb-4f75-a908-4b5bb025af8c Version: $LATEST
2024-10-20T14:28:52.292Z	Values: [[5.9, 3, 5.1, 2.3]]
2024-10-20T14:28:52.292Z	Features: [[5.9, 3, 5.1, 2.3]]
2024-10-20T14:28:52.303Z	END RequestId: e5511cd3-d2bb-4f75-a908-4b5bb025af8c
2024-10-20T14:28:52.303Z	REPORT RequestId: e5511cd3-d2bb-4f75-a908-4b5bb025af8c Duration: 9.00 ms Bi…
	我们目前未找到较新的事件。 自动重试已暂停。 恢复

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`

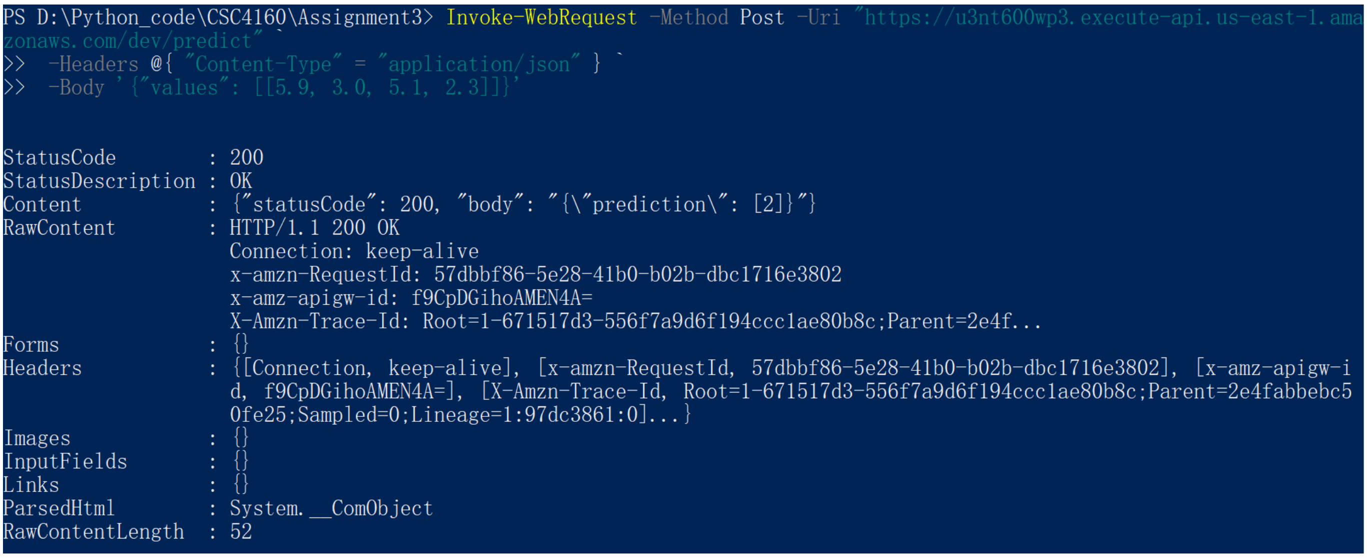
You can find the API URL under `Invoke URL`

```
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

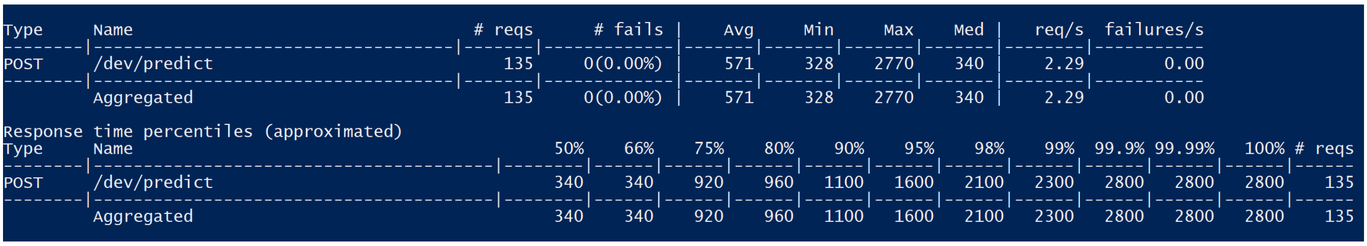
Environment Setup Run the following to install the required package:

```
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

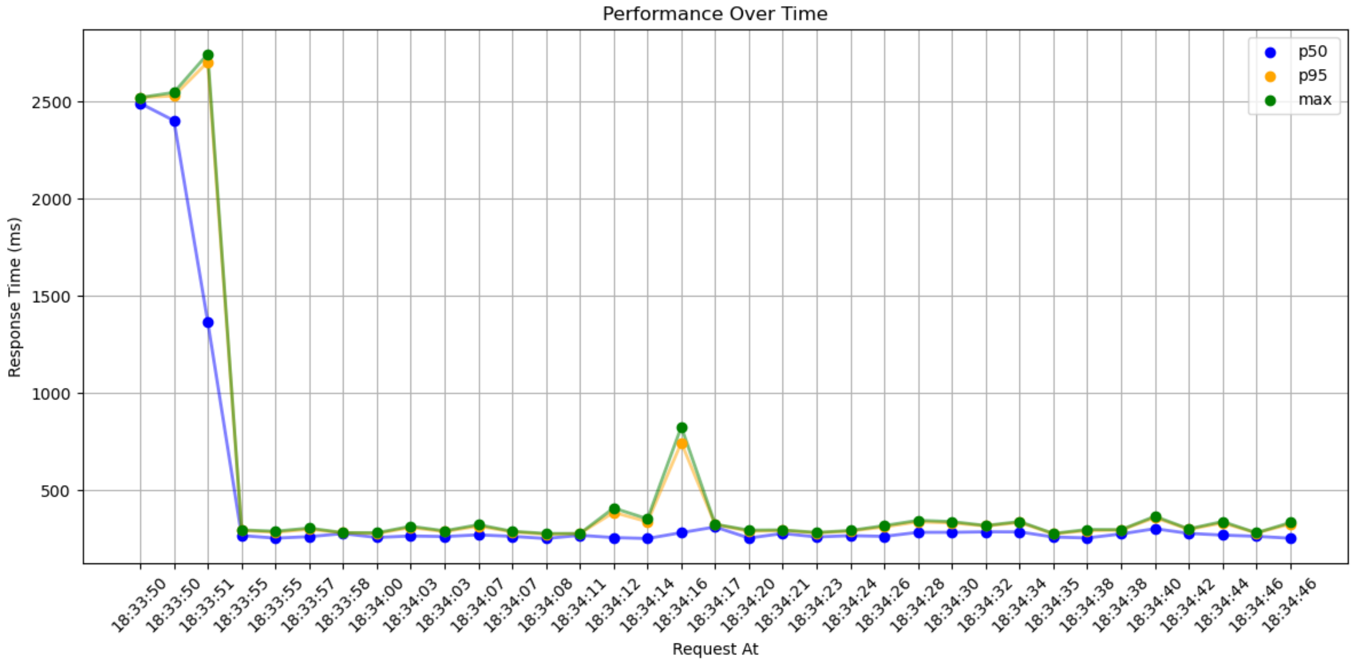


```
Type     Name                                 # reqs      # fails |    Avg     Min     Max    Med |   req/s  failures/s
--------|------------------------------------|-------|-------------|-------|-------|-------|-------|--------|-----------
POST     /dev/predict                            135     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
```

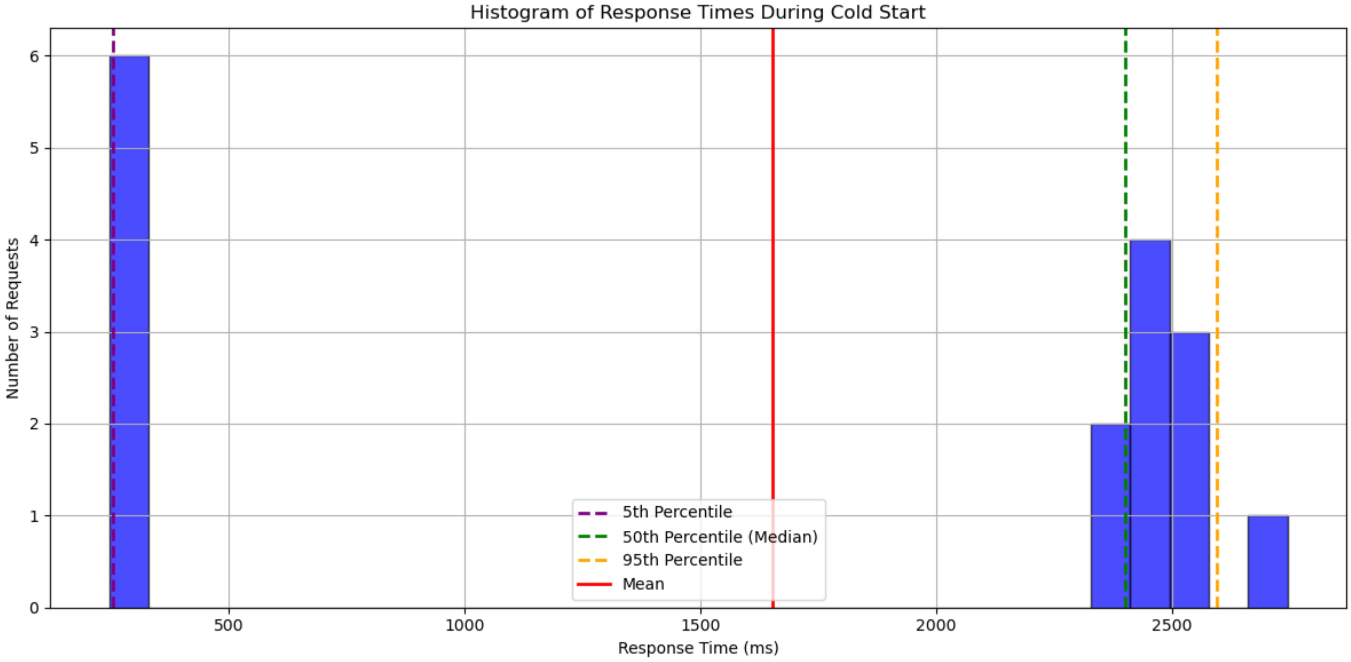
8. Analysis

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



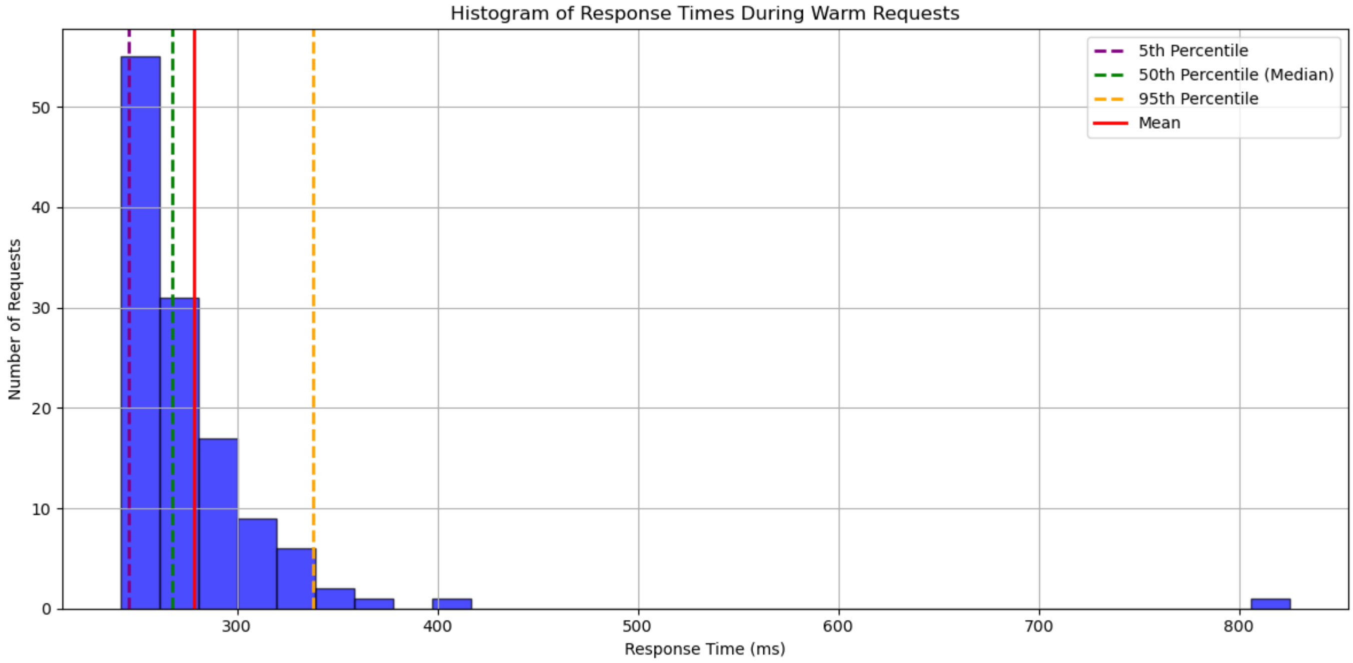
P50 (Median): The blue line represents the 50th percentile (median) of the response time. It shows that 50% of requests have a response time less than or equal to this value. P95 (95th Percentile): The orange line represents the 95th percentile, indicating that 95% of requests have a response time below this value, while 5% are slower. This helps identify occasional slow requests. Max (Maximum): The green line shows the maximum response time for each time period. This highlights extreme cases, such as cold starts or network delays, where the response time is much higher than usual. Insights: The chart shows variability in response times over time. While P50 remains fairly stable, P95 and Max experience spikes, indicating occasional slow requests, likely caused by cold starts or increased load. The Max values are particularly high during some periods, possibly due to cold starts or other transient conditions.

Second Chart: Histogram of Response Times During Cold Start



This histogram shows the distribution of response times during cold starts. P50 (50th Percentile): The green dashed line represents the median response time, indicating that 50% of requests are completed within this time. P95 (95th Percentile): The yellow dashed line represents the 95th percentile, showing that 95% of the requests are completed within this time. Max (Maximum): The highest value, likely to the right of the graph, shows the longest response times during cold starts. Mean (Average): The red solid line shows the average response time, which can be higher than the median due to the influence of slower outlier requests. Insights: Most requests during cold starts have response times concentrated around a lower range (likely below 500ms), but a few requests take much longer due to cold start delays. The 95th percentile is relatively high, indicating that cold starts introduce some significant variance in performance, causing occasional slow requests.

Third Chart: Histogram of Response Times During Warm Requests



This histogram shows the distribution of response times for warm requests (i.e., after the Lambda function has been initialized). Again, P50, P95, and Mean lines are indicated, showing how the response times are distributed. Warm requests generally show much lower response times compared to cold starts, with more consistent performance. Insights: During warm starts, most requests have shorter response times, generally below 500ms, indicating stable and fast performance once the system is warmed up. Although there are a few outlier requests with higher response times, overall performance is much better than during cold starts. Overall Summary: The first chart highlights performance fluctuations over time, with P95 and Max values showing occasional slow requests. The second chart shows response time distribution during cold starts, with a significant spread in response times due to the cold start effect. The third chart demonstrates much more stable performance during warm requests, with most requests completing quickly and consistently. This indicates that your system experiences noticeable performance degradation during cold starts, but once warmed up, it provides much more consistent and faster responses.

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 you 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 ===============