**CHAPTER 1**

**INTRODUCTION**

In today’s dynamic financial markets, staying updated on stock price movements is essential for making informed investment decisions. Investors need to act quickly when stock prices hit certain levels, but constantly monitoring the market manually can be time-consuming and impractical. This is where a **Real-Time Stock Price Alert System** becomes useful, providing instant updates and notifications based on stock price changes, allowing users to make timely decisions.

The **Real-Time Stock Price Alert System** is designed to automatically track live stock prices and alert users when these prices hit user-specified thresholds. By doing so, it helps investors stay informed and make timely decisions, such as buying or selling stock based on market trends. The system continuously monitors stock prices and sends alerts in real time through convenient channels like email or SMS, enabling users to react promptly without having to constantly watch the market.

In the real world, stock prices fluctuate throughout the trading day due to various factors such as market trends, company performance, economic data, or geopolitical events. Missing out on a critical price point can lead to lost profits or potential losses. The real-time alerts provided by this system help investors and traders stay informed about stock movements and make decisions, such as buying or selling stocks, when certain price conditions are met.

**Need for a Real-Time Stock Market Alert System**

In today's data-driven world, A Real-Time Stock Market Alert System provides crucial benefits for several reasons:

**1. Timely Decision-Making:**

* Instant Notifications: Real-time alerts when stock prices hit specific thresholds.
* Market Volatility: Quick reactions to sudden market changes.

**2. Risk Management:**

* Limit Losses: By setting predefined price alerts, users can be notified when a stock price drops below a certain value, allowing them to act quickly and prevent further losses.
* Profit Booking: Alerts for price increases allow investors to lock in profits by selling at the right time.

**3. Efficiency and Automation:**

* Eliminating Manual Monitoring: Automation of stock price tracking takes place.
* Focus on Analysis: More time for strategic decision-making.

**4. Customization and Flexibility:**

* User-Specific Alerts: Personalized notifications based on user preferences.
* Scalability: Capable of handling multiple users and stocks.

**5. Data-Driven Insights**:

* Historical Data Analysis: Combines real-time alerts with historical data trends for better decision-making.
* Sentiment Alerts: Notifications based on market news that could impact stock prices.

# CHAPTER 2

**EXECUTIVE SUMMARY**

The **Real-Time Stock Price Monitoring and Alert System** is an innovative solution tailored for investors, traders, and businesses looking to stay informed of stock market movements in real time. By integrating live stock price APIs, the system continuously tracks stock performance and triggers alerts when predefined thresholds are met. With markets fluctuating rapidly, the system ensures that users can make well-informed, timely decisions based on accurate and real-time data, empowering them to respond to market opportunities or risks promptly. The integration with AWS ensures that the system can scale according to user demands, handling increased traffic during periods of high market activity. With secure, reliable, and high-performance cloud services, users can trust the system’s ability to provide timely and accurate alerts consistently.

**Key Components:**

**1. Overview of Stock Price Alert System:**

The **Real-Time Stock Price Alert System** is a powerful tool that keeps users informed about critical stock price movements, enabling timely decisions in the fast-paced financial market. With the ability to monitor stock prices continuously and deliver alerts in real time, this system empowers investors, traders, and analysts to stay ahead of market fluctuations.

**2. Benefits of Stock – Time Alerts:**

* Automation
* Real – Time Updates
* Customizability
* High performance of Cloud Services
* Reliability
* Security

**3. Workflow and Alert Triggering Mechanism:**

The system continuously monitors stock prices using an API. Users can set their desired alert conditions, such as specific price points or percentage changes. When the set condition is met, the system instantly triggers a notification via email or SMS, allowing users to take timely actions without the need for constant market oversight.

**4. Security Measures:**

The system places a strong emphasis on data security. User information and alert configurations are protected by encryption, ensuring privacy and data integrity. In addition, AWS Identity and Access Management (IAM) policies control access, ensuring only authorized users can modify or view sensitive data.

**5. AWS SNS:**

AWS SNS helps send messages to many people at once. You can use it to notify users when a stock price changes or reaches a certain level. Users can choose to receive alerts through email, SMS, or app notifications. This service is easy to set up and manage.

It ensures users get timely updates about their favorite stocks. SNS makes communication fast and efficient.

**6. Cloud9:**

**AWS Cloud9** is an online code editor for writing and testing code. You can install necessary libraries and dependencies to ensure your code runs smoothly. AWS Lambda does not support installation commands like pip install so, we use cloud9 to install all the required packages and make the python file to zip file.

**7. AWS Lambda:**

AWS Lambda runs code automatically when events occur, like stock price updates. It fetches stock prices from APIs and sends alerts based on conditions. This serverless service is cost-effective, charging only for usage.

**8. AWS Identity and Access Management (IAM):**

IAM controls access to AWS resources by setting user permissions. It ensures only authorized individuals can send alerts or fetch data. This keeps sensitive information secure in your stock alert system.

**9. API Gateway:**

API Gateway creates APIs that connect users to your services. It acts as a bridge between clients and the backend, allowing users to subscribe to alerts or check prices

**10. Amazon CloudWatch:**

Amazon CloudWatch monitors your AWS resources and performance metrics. It helps track Lambda function usage and response times. You can set alerts for issues, ensuring your system runs smoothly.

**11. Use Cases**

The **Real-Time Stock Price Alert System** caters to various **financial professionals** with unique needs. **Day traders** receive **instant notifications** to seize **short-term price changes**. **Long-term investors** monitor specific stock prices for **strategic adjustments**. **Risk-averse investors** get timely alerts on **price drops** to protect their portfolios. **Portfolio managers** can track multiple stocks with **personalized alerts** to optimize performance. Finally, **financial analysts** automate market tracking, allowing them to focus on **in-depth analysis** and insights.

**CHAPTER 3**

**REAL TIME STOCK PRICE ALERT SYSTEM**

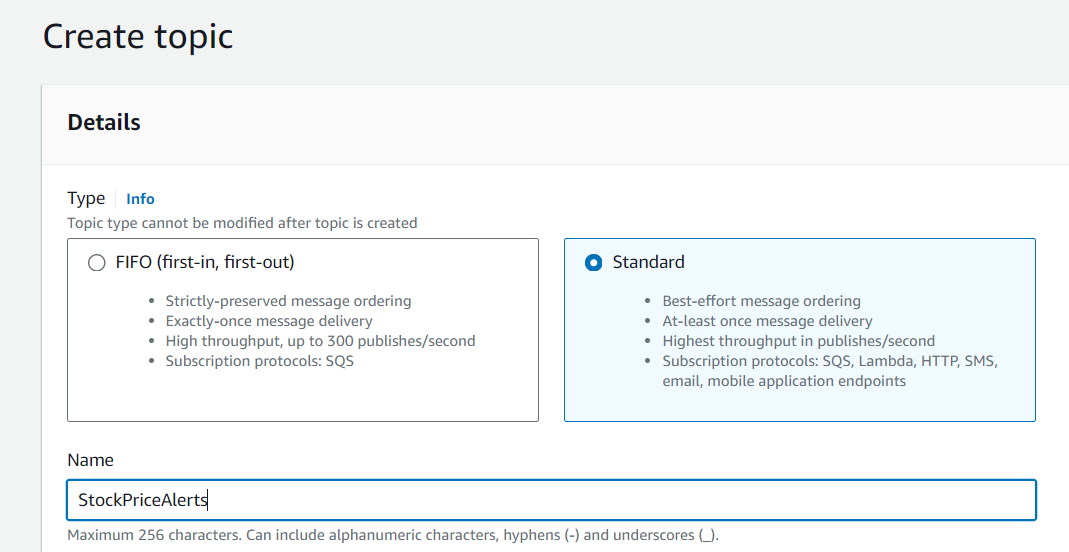
The goal of the **Real-Time Stock Price Alert System** is to watch stock prices and send alerts to users when a stock reaches a certain price. This helps investors stay informed and act quickly, without having to check the market all the time, so they don’t miss any important changes.

**Notable Factors:**

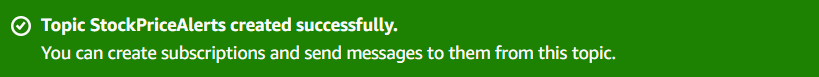
* In lambda function the installations are not possible.
* In cloud9 bash the installations takes place and the file can also be directly, convert into the .zip file. Install dependencies and package code in Cloud9 for Lambda.
* In order to connect the cloud9 and lambda we need to use the s3 and other options.
* But here we used the local system download option and directly connected to the lambda.
* Lambda’s pay-per-use model is ideal for real-time alerts which is cost efficient.

**STEPS - 1: AWS SNS**

* Login to the “AWS Management Console”, select the **SNS** (Simple Notification Service) Service, navigate to the Topic and Click **Create Topic**. We need to choose the topic type as **Standard** and enter the topic name: **Stock Price Alerts.**

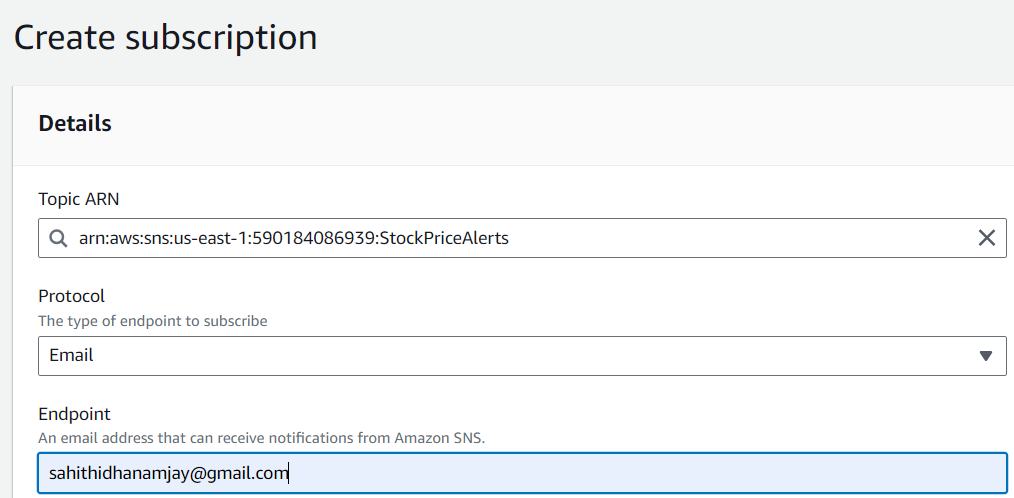


**Figure 3.1 AWS SNS Topic Creation**

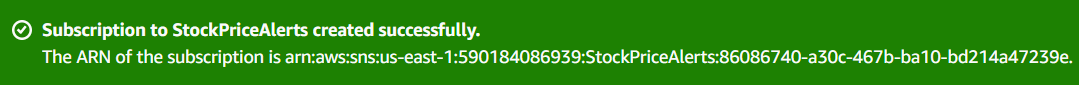


**Figure 3.2 SNS Topic Created**

* Navigate to the Subscriptions section and create a new subscription. Enter the ARN of the topic you created earlier and specify the required protocol like for the email protocol, make sure to enter the recipient's email address.

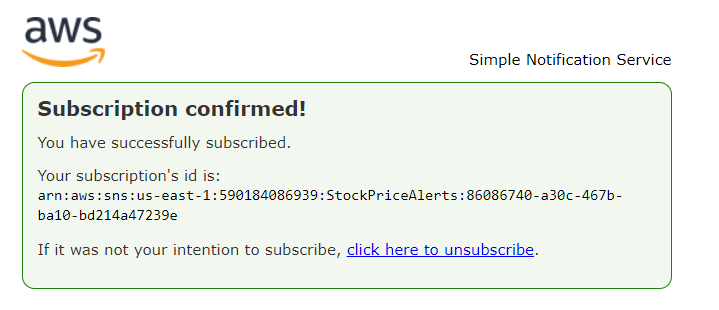


**Figure 3.3 Subscription for Topic**

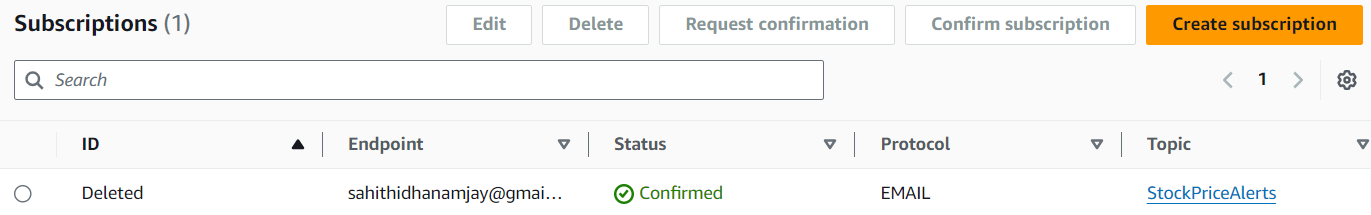


**Figure 3.4 Subscription Created**

* As you have selected the email protocol, a confirmation email will be sent to the specified address. Please ensure to confirm the subscription by clicking the link in that email.



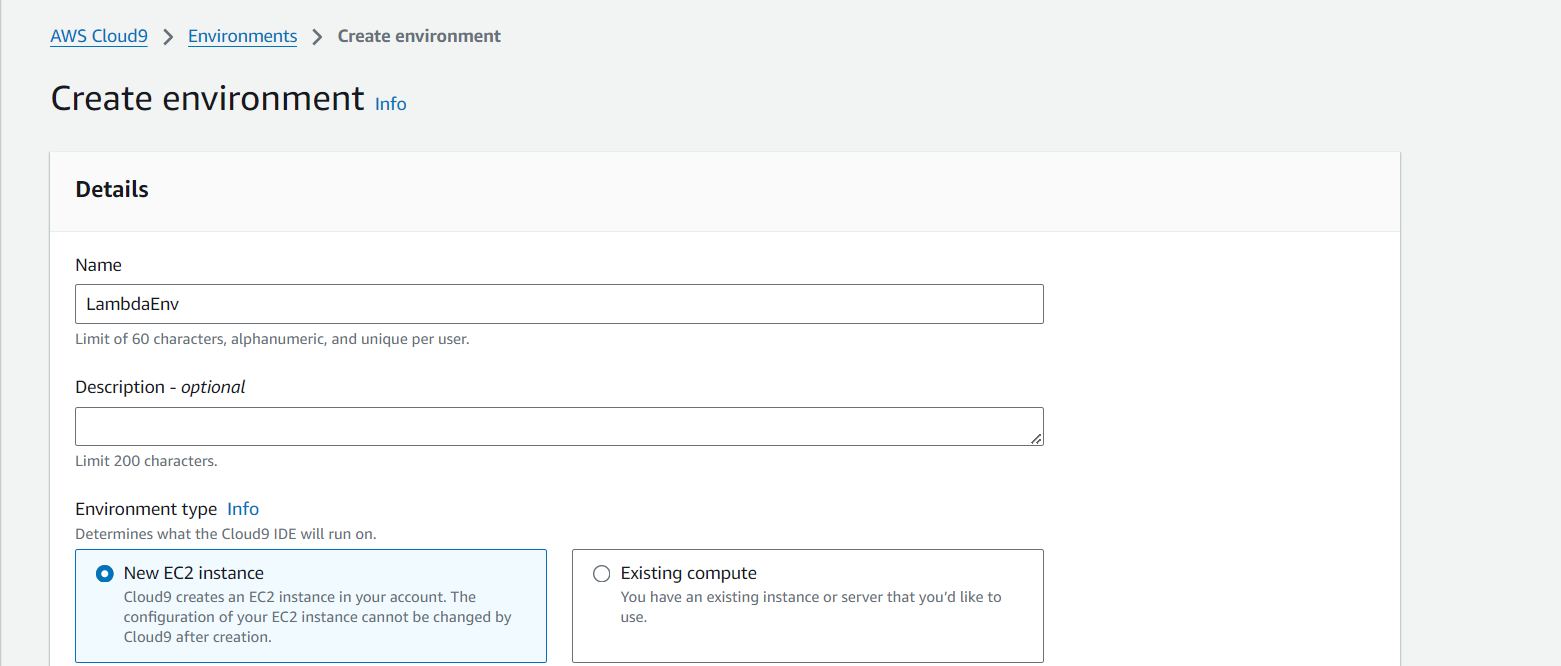
**Figure 3.5 SNS Subscription Conformation**



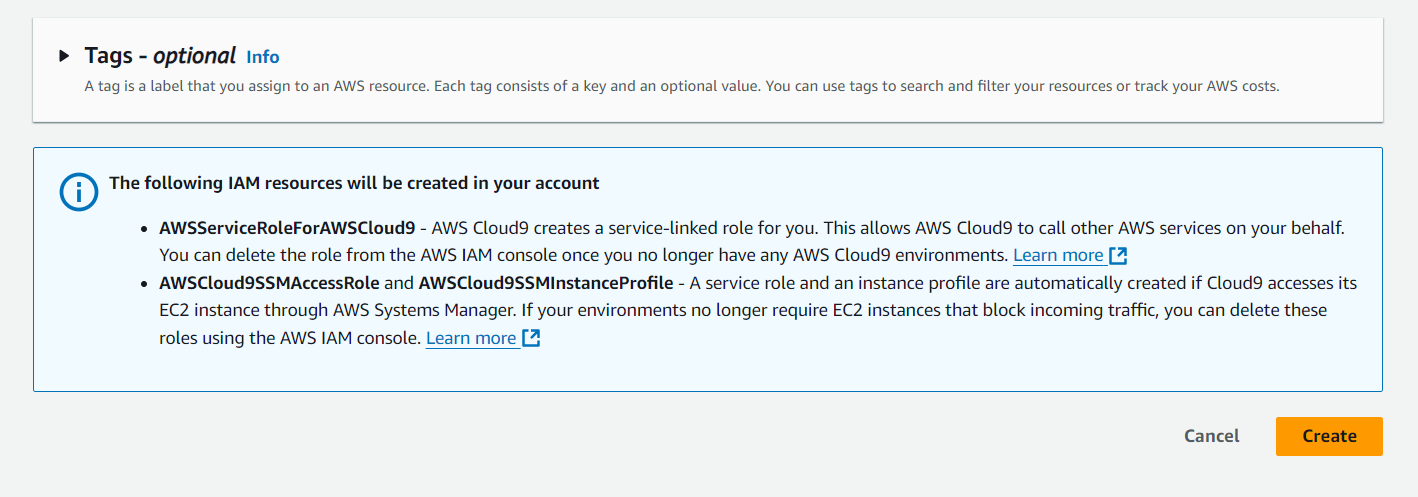
**Figure 3.6 Mail Conformation in SNS Subscription**

**STEP – 2: Cloud9**

* Navigate to the Cloud9 Service and create an environment named LambdaEnv.



**Figure 3.7 Cloud9 Environment**



**Figure 3.8 Creation of environment**

* Once your Cloud9 environment is set up, open a new terminal inside Cloud9, create a new directory for your Lambda function named: stock\_price\_lambda by below command.

**Command:** mkdir stock\_price\_lambda

* Change to the created directory by below command.

**Command:** cd stock\_price\_lambda

* Install the request library by using the following commands in the current directory. This will install the “requests” module and all its dependencies into the stock\_price\_lambda directory.

**Command:** pip install requests -t .

* In the Cloud9 file explorer, create a new Python file named **lambda\_function.py** in the stock\_price\_lambda directory. Add your Lambda function code to this file.

**Path to create a file:** file -> new file -> name it as: lambda\_function.py.

* **CODE in lambda\_function.py:**

import json

import requests

import boto3

import os

# Set up SNS client

sns = boto3.client('sns')

# Set up stock API details

STOCK\_API\_KEY = os.getenv("STOCK\_API\_KEY", "demo")

STOCK\_SYMBOL = os.getenv("STOCK\_SYMBOL", "IBM")

# SNS topic ARN (replace with your actual ARN)

SNS\_TOPIC\_ARN = os.getenv("SNS\_TOPIC\_ARN", "arn:aws:sns:us-east-

1:590184086939:StockPriceAlerts")

# Threshold values

UPPER\_THRESHOLD = float(os.getenv("UPPER\_THRESHOLD", 150.00))

LOWER\_THRESHOLD = float(os.getenv("LOWER\_THRESHOLD", 100.00))

def get\_stock\_price(symbol):

# Alpha Vantage API URL

url = f"https://www.alphavantage.co/query?function=TIME\_SERIE

S\_INTRADAY&symbol={STOCK\_API\_KEY}&interval=1min&apike

y={STOCK\_TOPIC\_ARN}"

response = requests.get(url)

data = response.json()

# Extract the latest stock price

time\_series = data.get("Time Series (1min)", {})

if not time\_series:

return None

latest\_time = sorted(time\_series.keys())[0]

latest\_price = float(time\_series[latest\_time]["1. open"])

return latest\_price

def lambda\_handler(event, context):

# Get the current stock price

stock\_price = get\_stock\_price(STOCK\_SYMBOL)

if stock\_price is None:

print("Could not fetch stock price")

return {"statusCode": 500, "body": json.dumps("Stock

price fetch failed")}

print(f"Current {STOCK\_SYMBOL} price: {stock\_price}")

# Check if the stock price exceeds the upper threshold

if stock\_price > UPPER\_THRESHOLD:

message = f"The stock price of {STOCK\_SYMBOL} is

${stock\_price}, which exceeds your upper

threshold of ${UPPER\_THRESHOLD}!"

# Send an SNS notification for upper threshold

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=message,

Subject=f"{STOCK\_SYMBOL} Stock Price Alert: Above

Threshold")

print("Upper threshold alert sent!")

# Check if the stock price falls below the lower threshold

elif stock\_price < LOWER\_THRESHOLD:

message = f"The stock price of {STOCK\_SYMBOL} is

${stock\_price}, which is below your lower

threshold of ${LOWER\_THRESHOLD}!"

# Send an SNS notification for lower threshold

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=message,

Subject=f"{STOCK\_SYMBOL} Stock Price Alert: Below

Threshold")

print("Lower threshold alert sent!")

else:

print(f"The stock price of {STOCK\_SYMBOL} is within the

threshold range.")

return {"statusCode": 200, "body": json.dumps(f"Checked stock

price: ${stock\_price}")}

* After you've added your code, zip the contents of the directory by below command. This will create a file named **lambda\_function.zip** that contains both the **requests** library and your **lambda\_function.py** code.

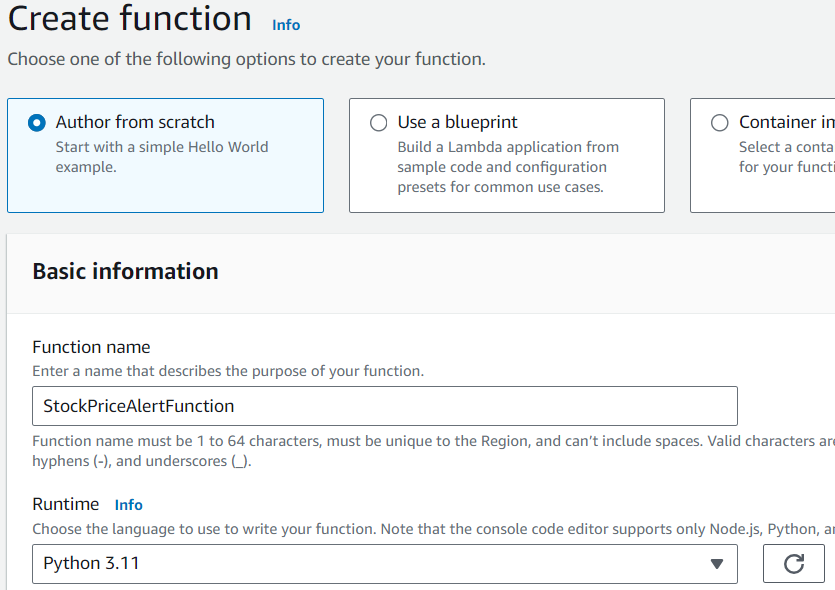
**Command:** zip -r lambda\_function.zip .

* Now download the project file to the local system and right click on the .zip file folder and choose extract all option.

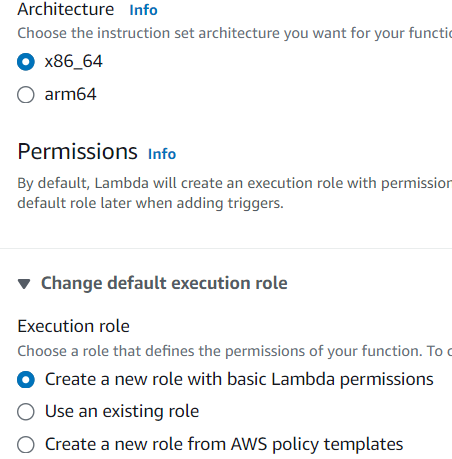
**Path to extract zip file:** downloads -> extract all

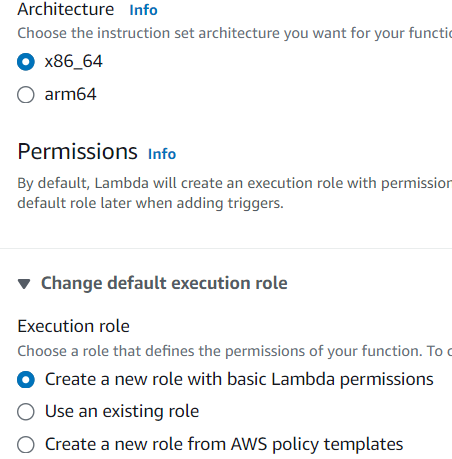
**STEPS - 3: Working with lambda function**

* In the AWS Management Console, navigate to Lambda and create a function by selecting Author from scratch, naming it **StockPriceAlertFunction**, choosing **Python 3.11** as the runtime, **x86\_64** for architecture, and selecting Create a **new role** with basic **Lambda permissions** for the execution role.



**Figure 3.9 Lambda function creation**





**Figure 3.10 Architecture and Execution role**



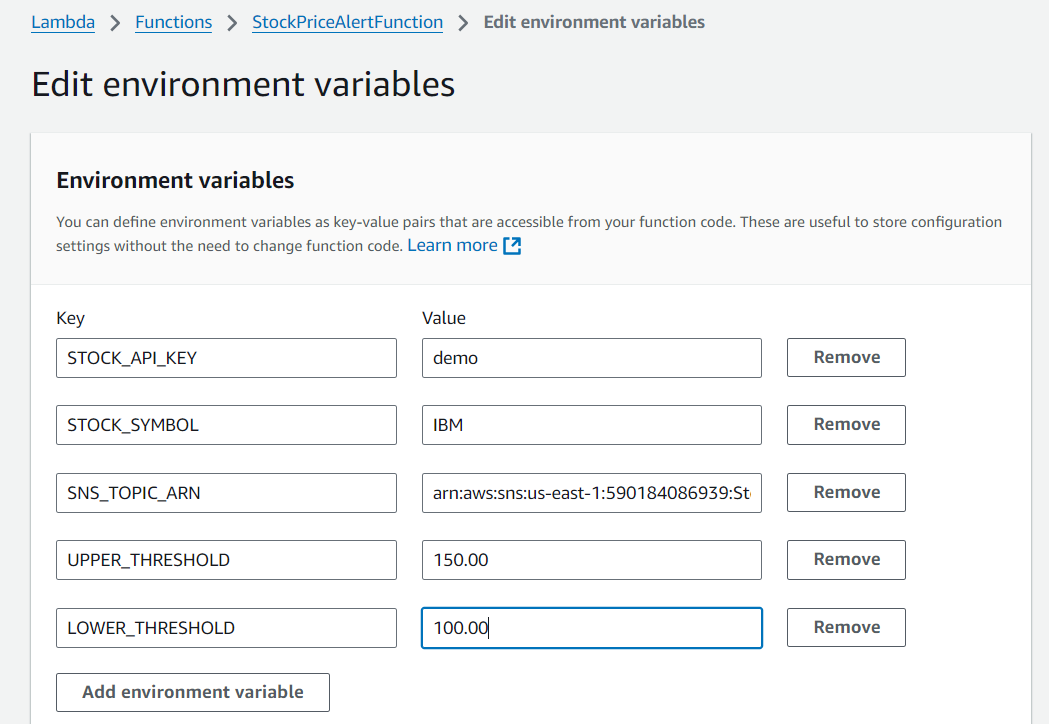
**Figure 3.11 Lambda function created**

* Upload the code file from the local system the lambda function for that we choose the **Upload** option in the function and choose **.zip file** and select the lambda\_function.py file in the extracted .zip file.



**Figure 3.12 Lambda function Updated**

* In the Lambda function settings, you need to add **environment variables** to store sensitive information and configurations.



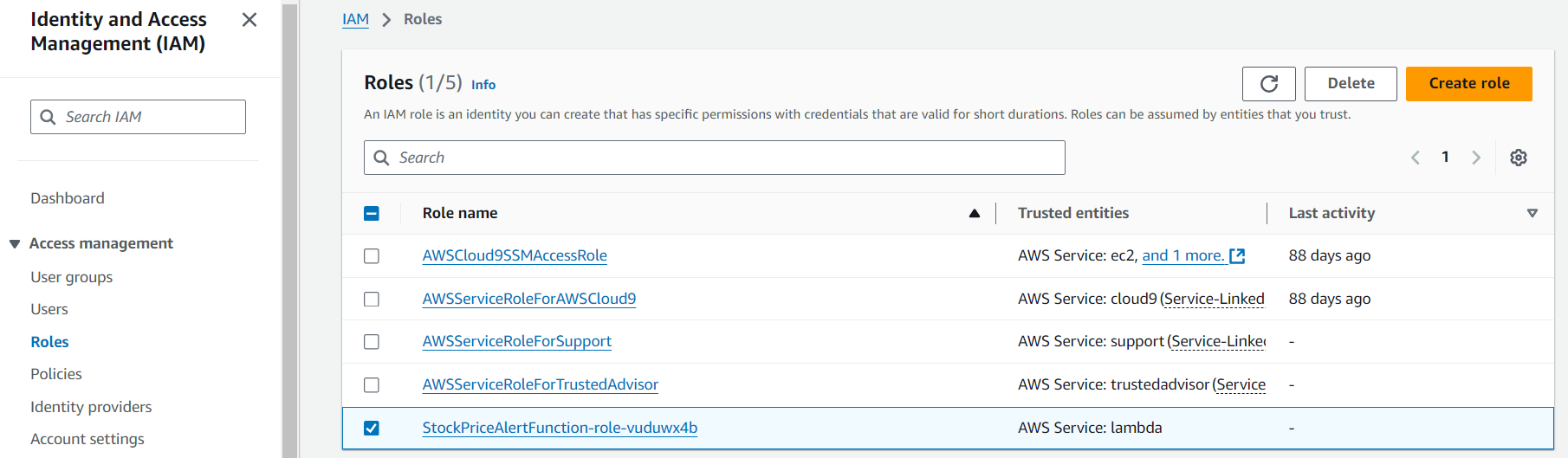
**Figure 3.13 Environment Variables in lambda function**



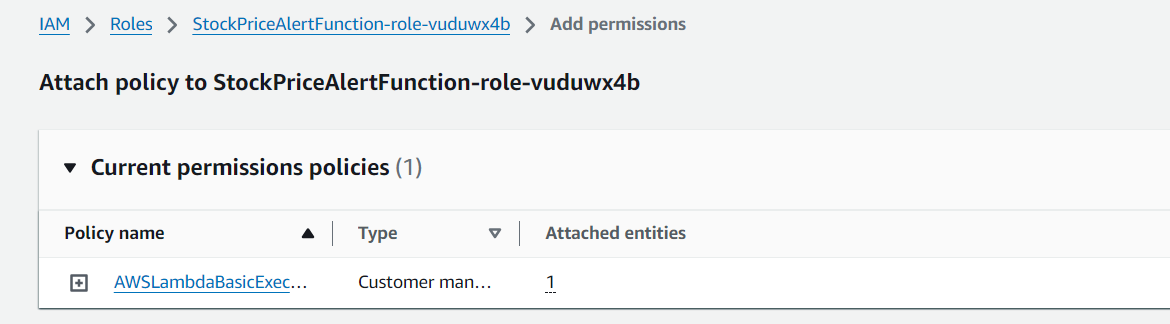
**Figure 3.14 Function updated**

**STEP – 4: Set Lambda Permissions for SNS**

* For Lambda to send messages via SNS, you need to update the Lambda execution role to allow SNS access.
* Navigate to IAM Service AWS Management services and choose a role that is already created with the function name. Add the permissions and add the below policy code in the policy editor.



**Figure 3.15 IAM role**



**Figure 3.16 Add permissions to IAM role**

* **CODE in policy editor:**

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": "logs:CreateLogGroup",

"Resource":"arn:aws:logs:us-east-1:590184086939:\*"

},

{

"Effect": "Allow",

"Action": [

"logs:CreateLogStream",

"logs:PutLogEvents" ],

"Resource": [

"arn:aws:logs:us-east-1:590184086939:log-

group:/aws/lambda/StockPriceAlertFunction:\*"

]

},

{

"Effect": "Allow",

"Action": "sns:Publish",

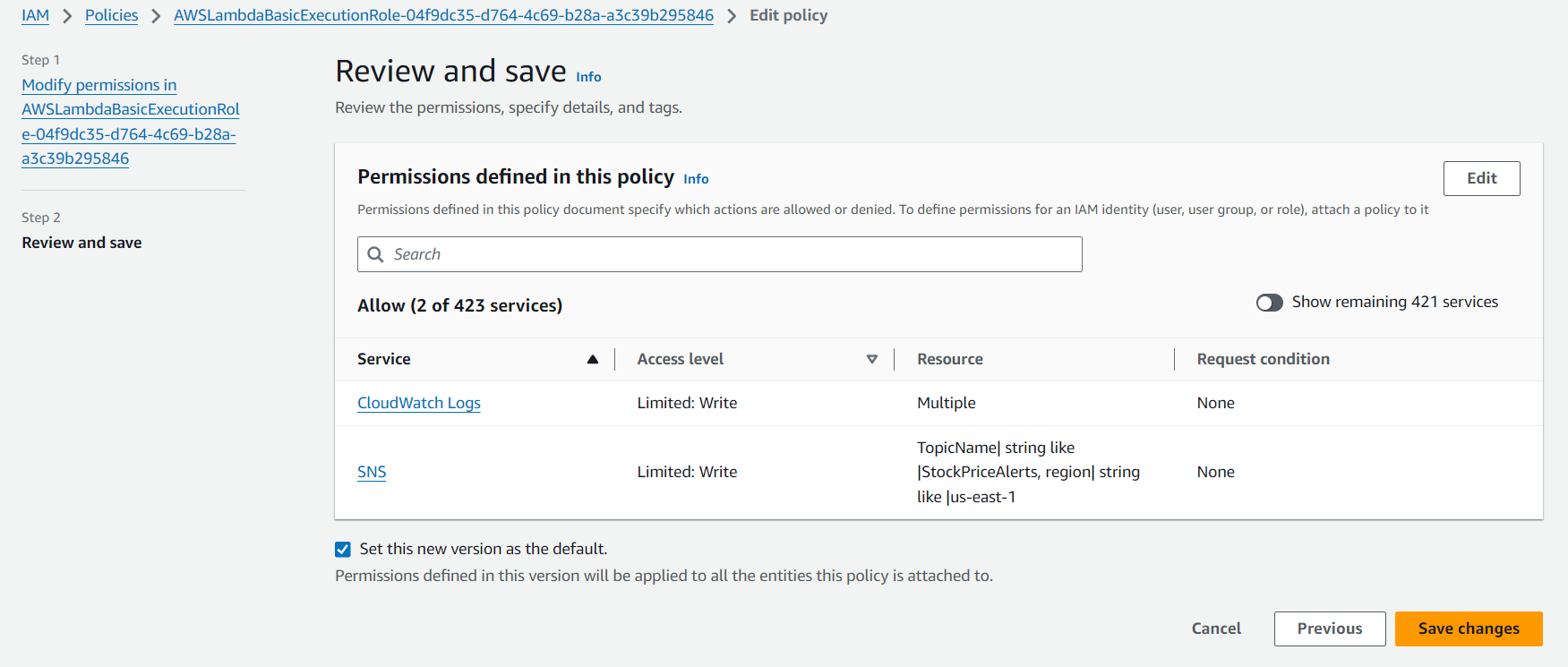
"Resource":"arn:aws:sns:us-east-1:590184086939:Stock

PriceAlerts"

}

]

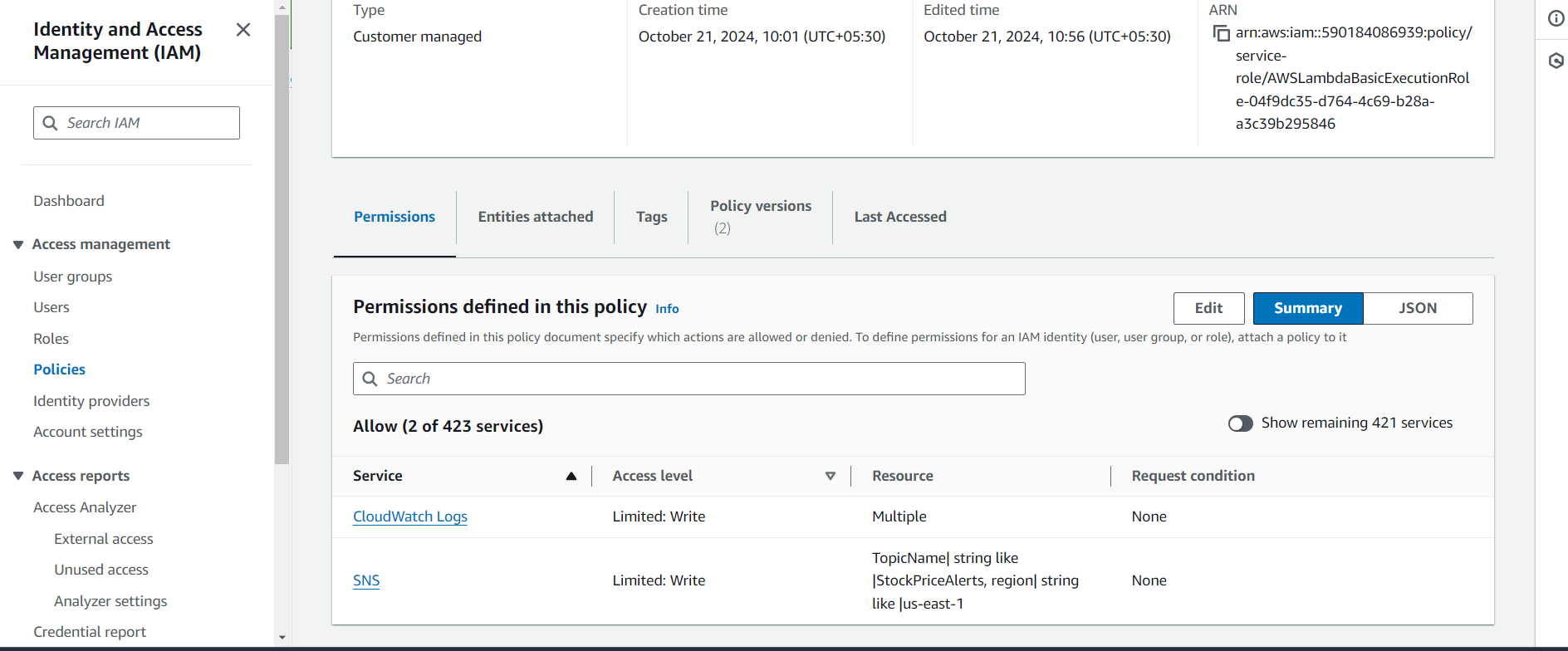
}



**Figure 3.17 IAM policy review**



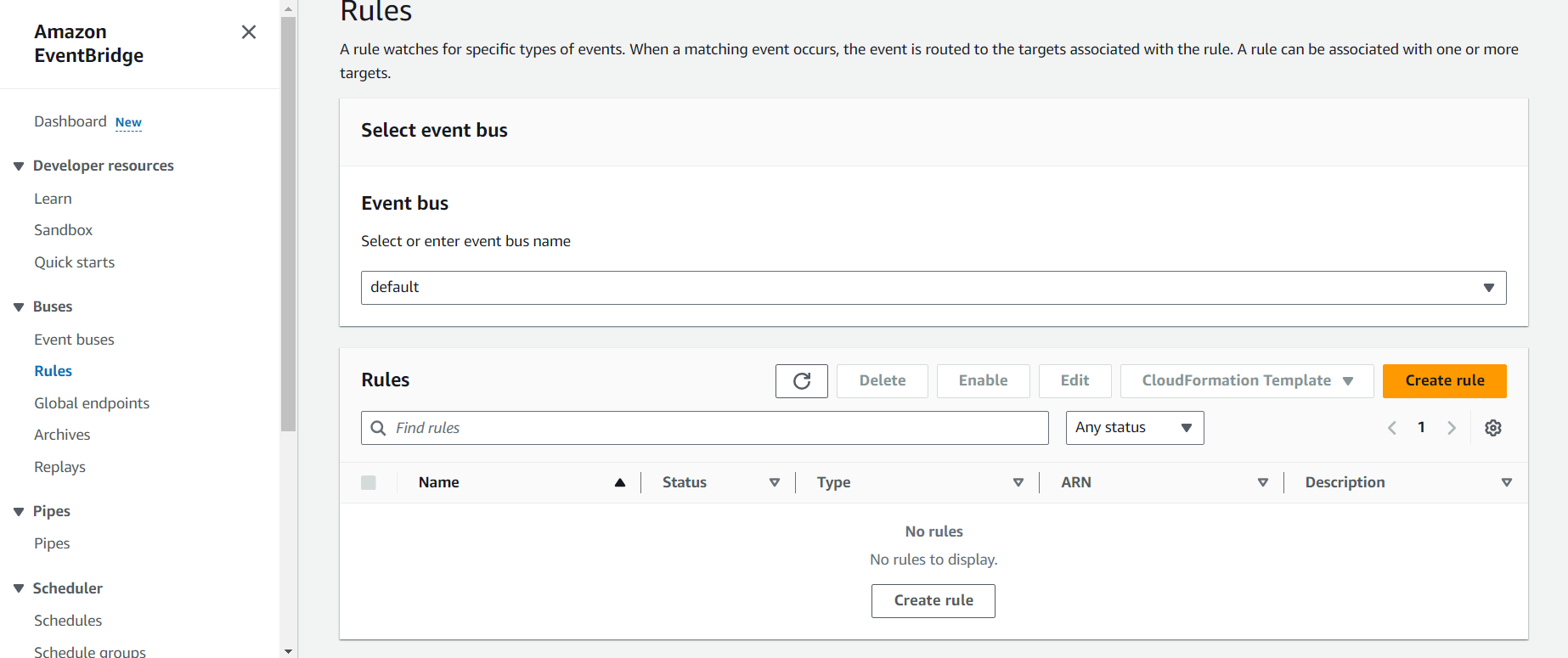
**Figure 3.18 Lambda Policy Updated**

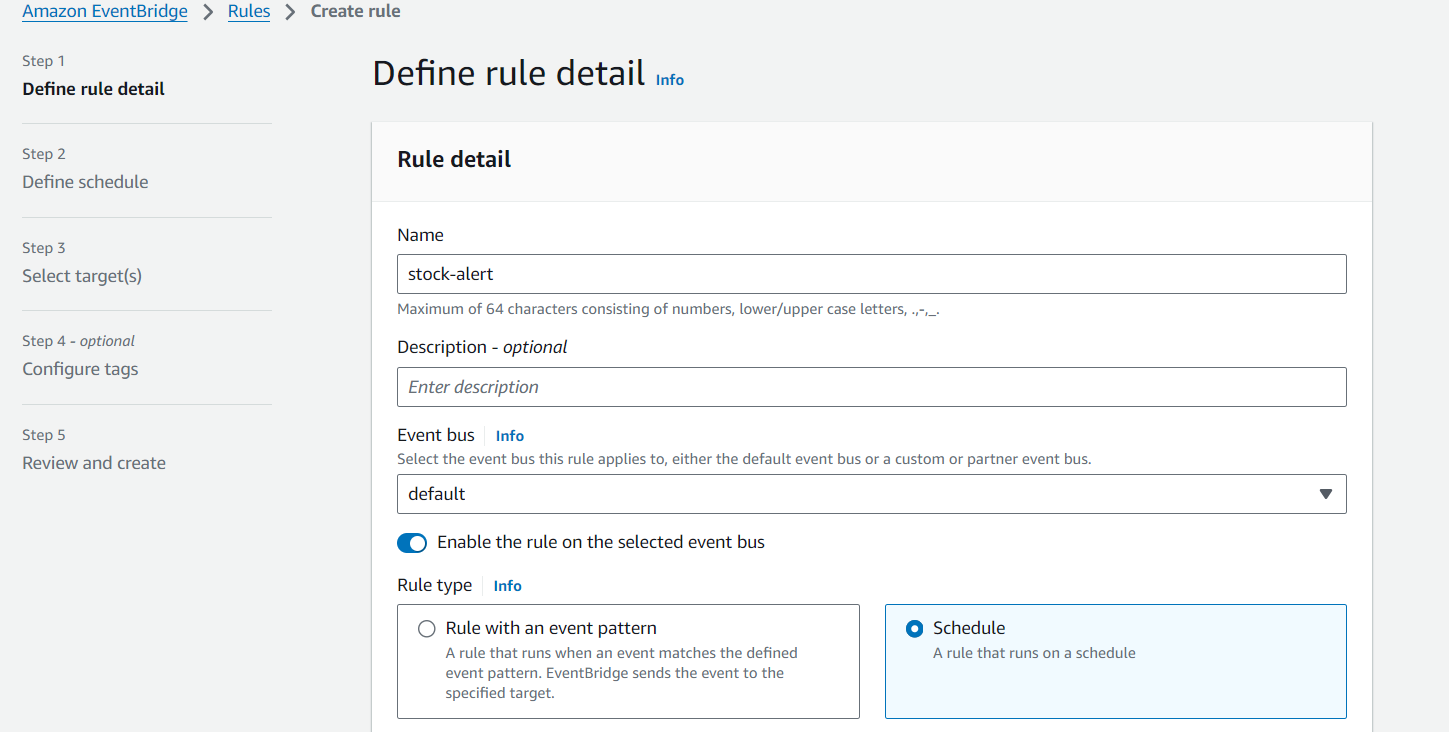


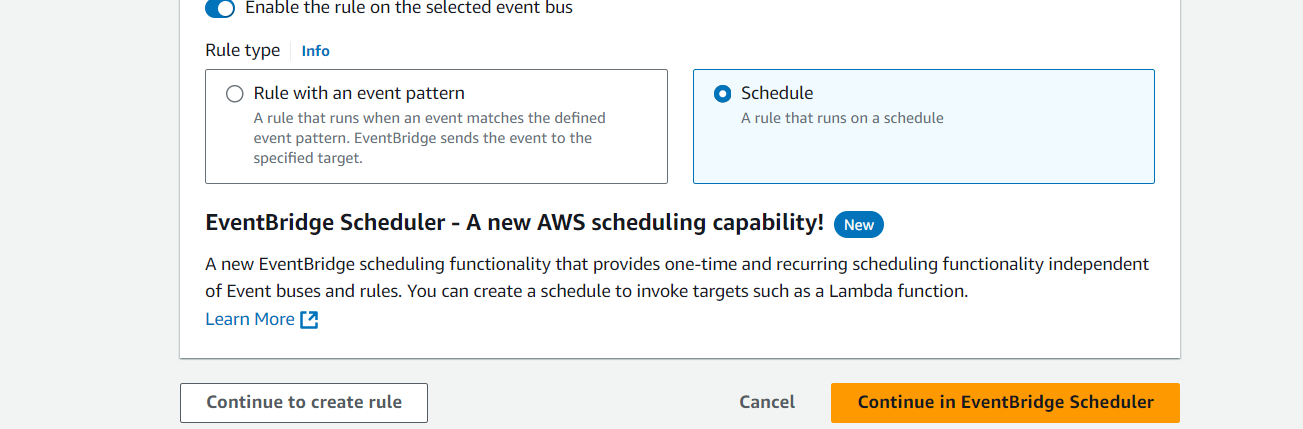
**Figure 3.19 IAM**

**STEP – 5: Schedule the Lambda Function with CloudWatch**

* To run the Lambda function at regular intervals (e.g., every 15 minutes), schedule it using Amazon CloudWatch Events (or EventBridge).
* Navigate to the CloudWatch Console and select **Rule** and then **create rule** with name **stock-alert**, RuleType as Schedule.

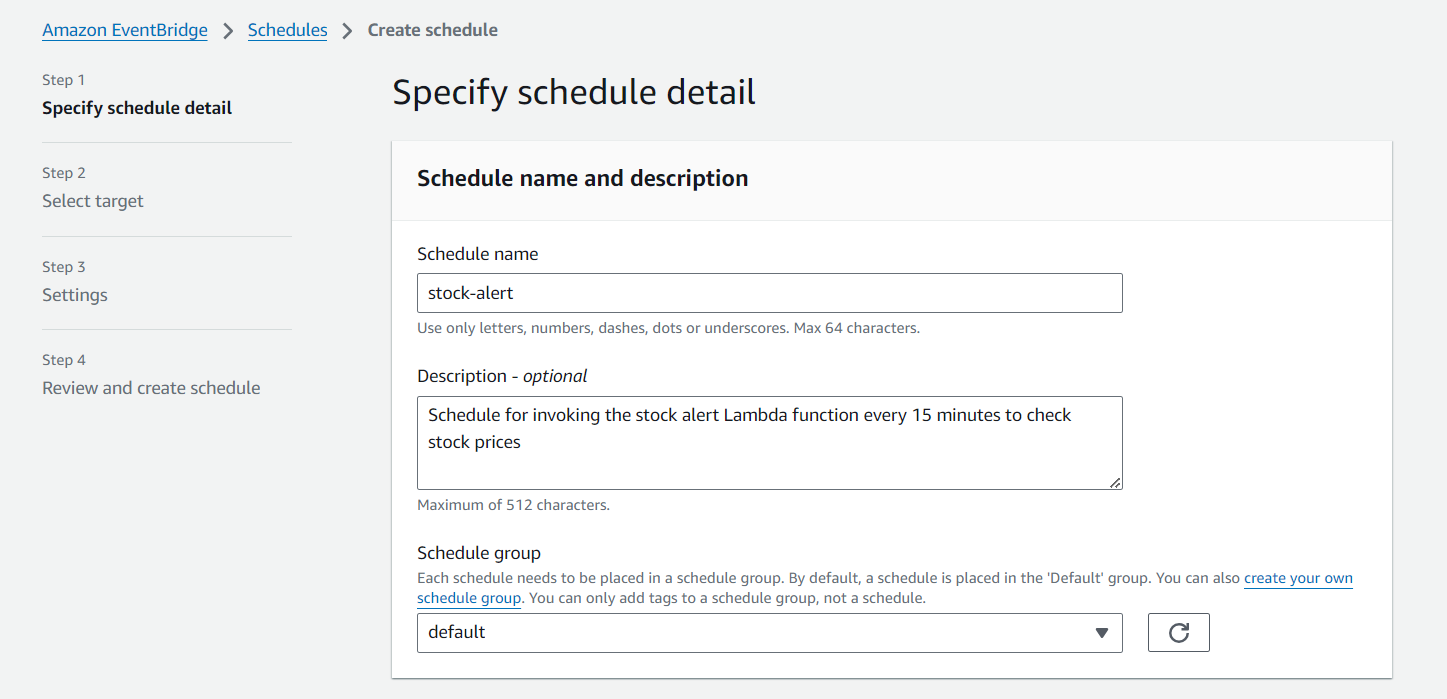


**Figure 3.20 CloudWatch Rule**



**Figure 3.21 Cloud Watch Rule creation**

* Create schedule with name **stock-alert** and can add description



**Figure 3.22 Event Bridge Schedule**

* Moving to the next step choose **Recuring Schedule** in occurrence and set up the time zone as UTC+05:30 Asia/Calcutta and schedule type is **Cron – based schedule,** fill the Cron expression.

cron(\*/5 \* \* \* \* ? \*)

This cron expression breakdown:

\*/5 - Every 5 minutes

\* - Every hour

\* - Every day of the month

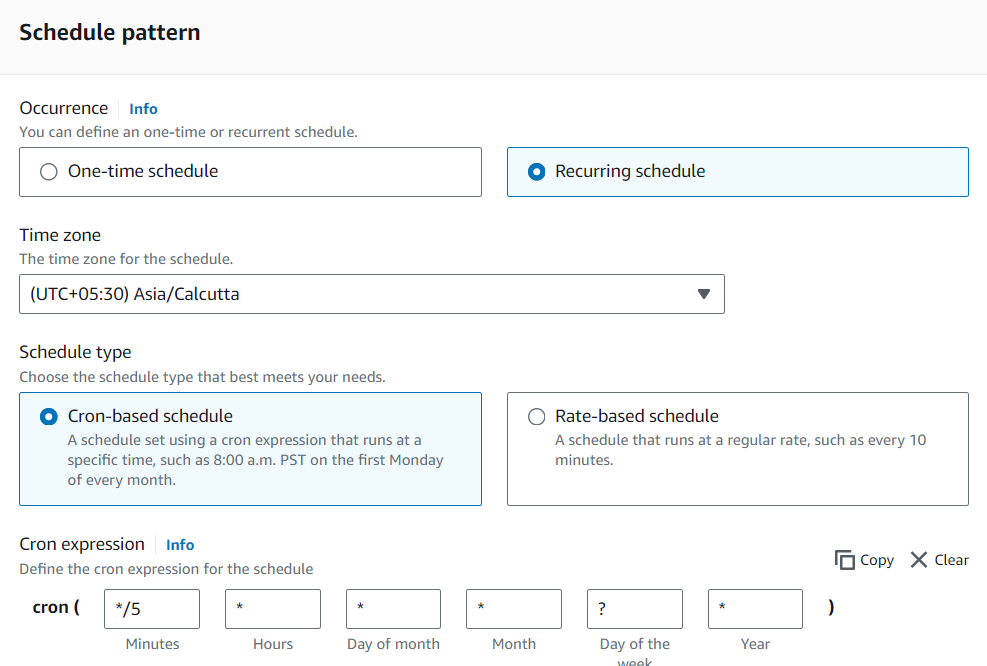
\* - Every month

? - No specific day of the week (because we are using \* for day of the month)

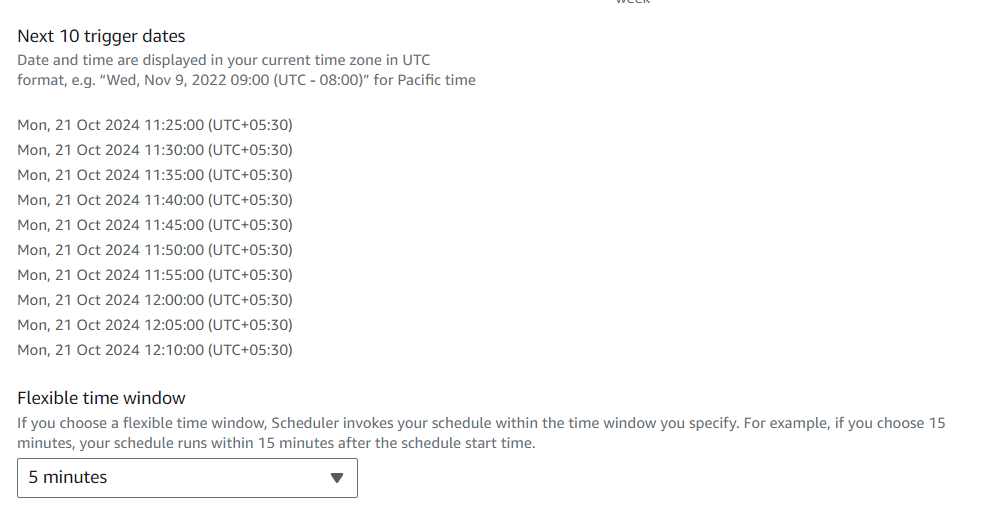
\* - Every year

This expression schedules a task to run every 5 minutes, regardless of the day, hour / month. This will run the Lambda function every 5 minutes as scheduled.

Mostly this will be useful for the timely alerts if the stock price raise above the threshold value or sit below the threshold value.

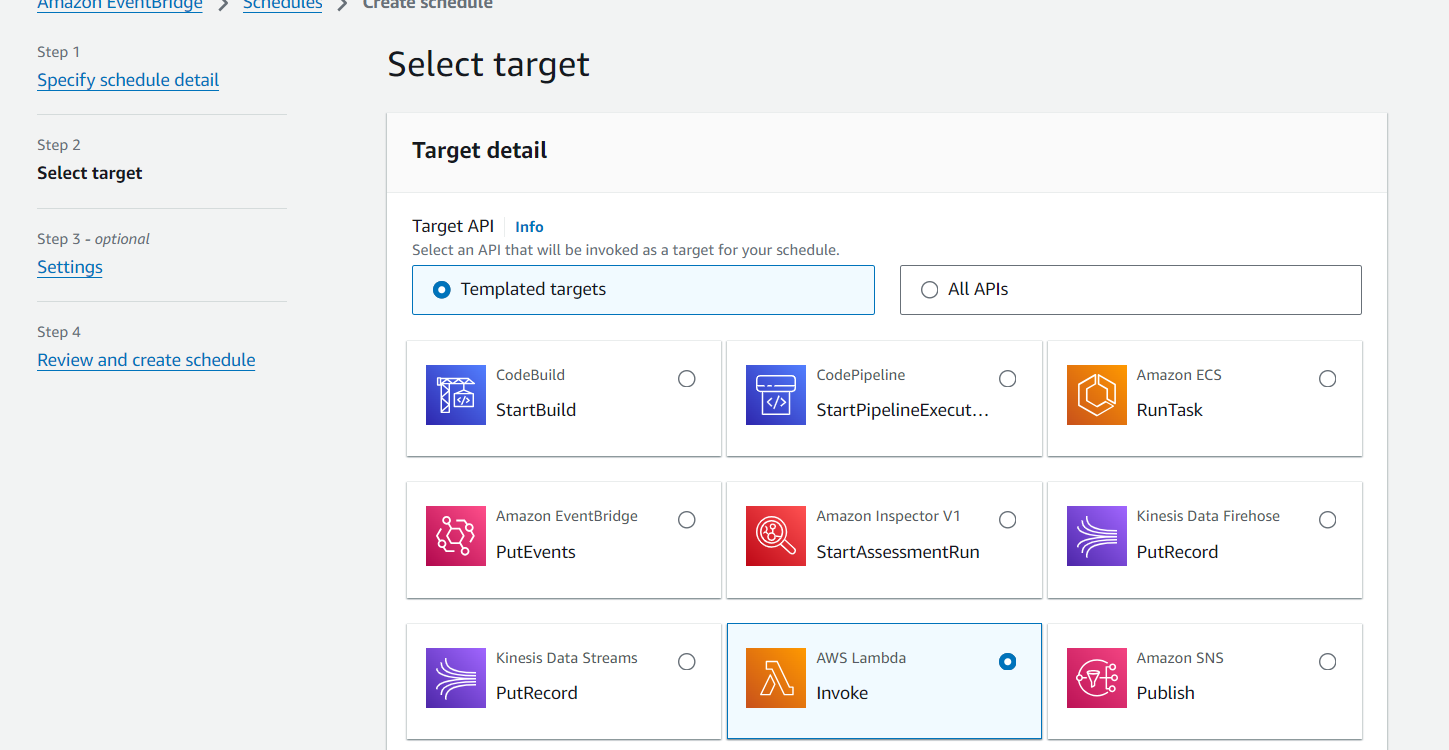


**Figure 3.23 Schedule properties**

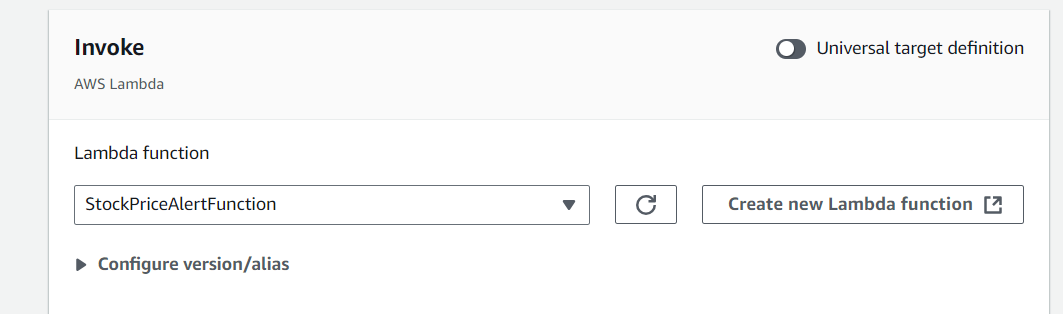


**Figure 3.24 Trigger time and date**

* Select the target as Lambda function and invoke the lambda function.

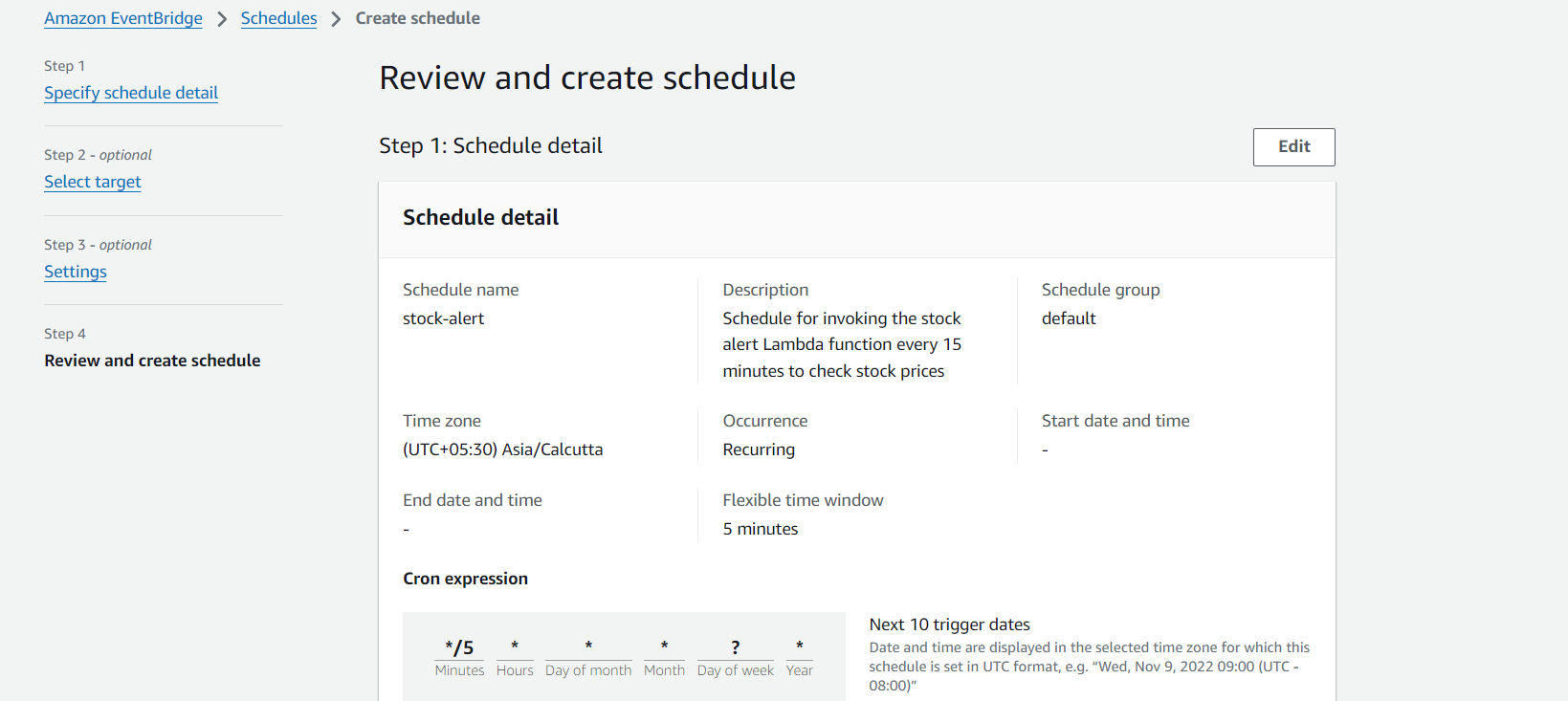


**Figure 3.25 Target selection in lambda**

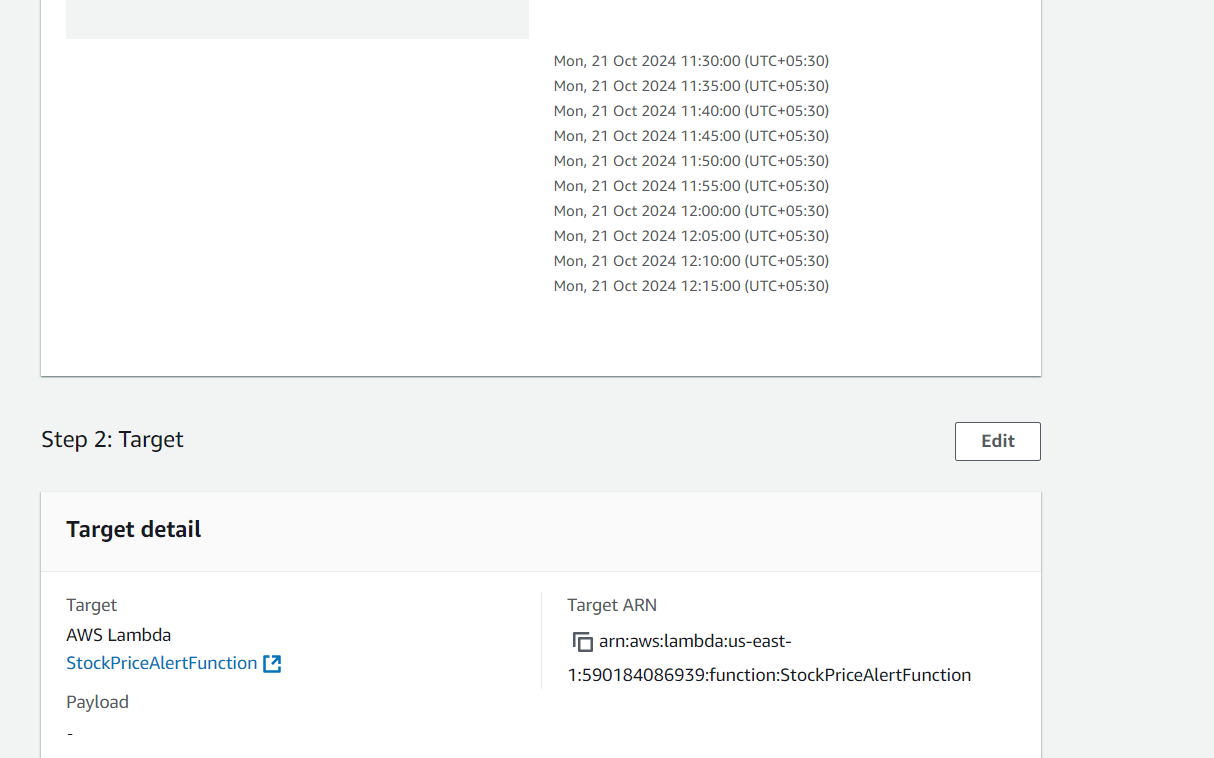


**Figure 3.26 AWS Lambda function invoke in schedule**

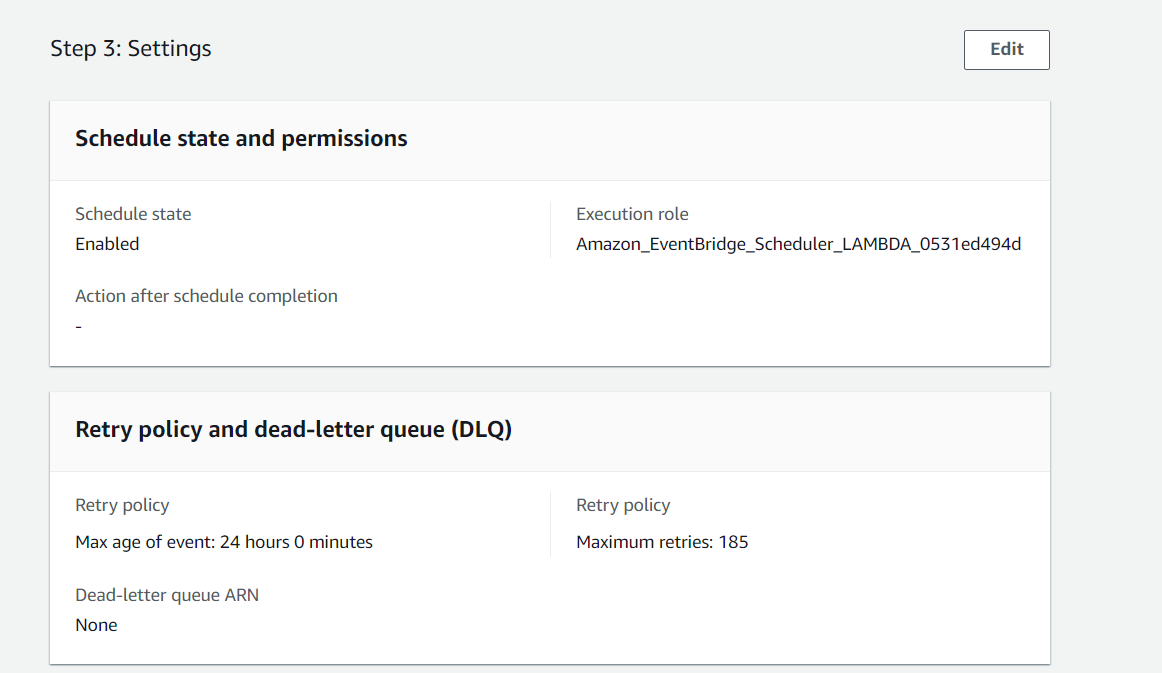
* So, this review of the data will be help to verify the specification and chosen action meets the requirement or not. More over if we found any anomaly we can easily modify it by the edit option present at top right corner of very review step.



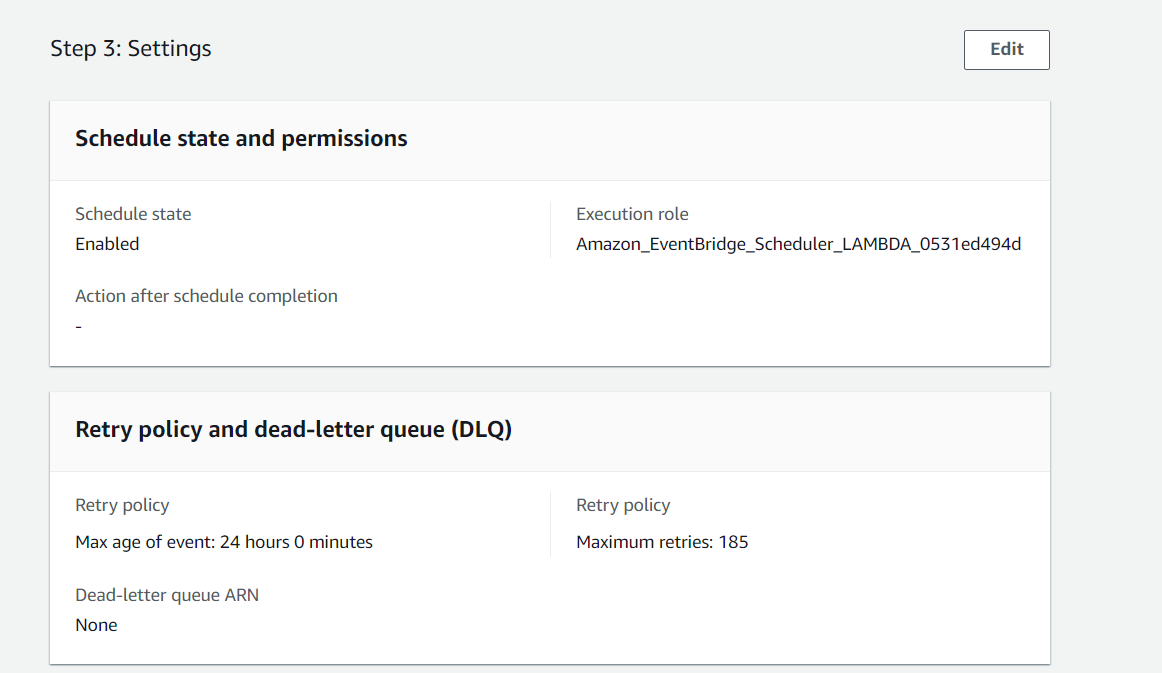
**Figure 3.27 Review of schedule details**



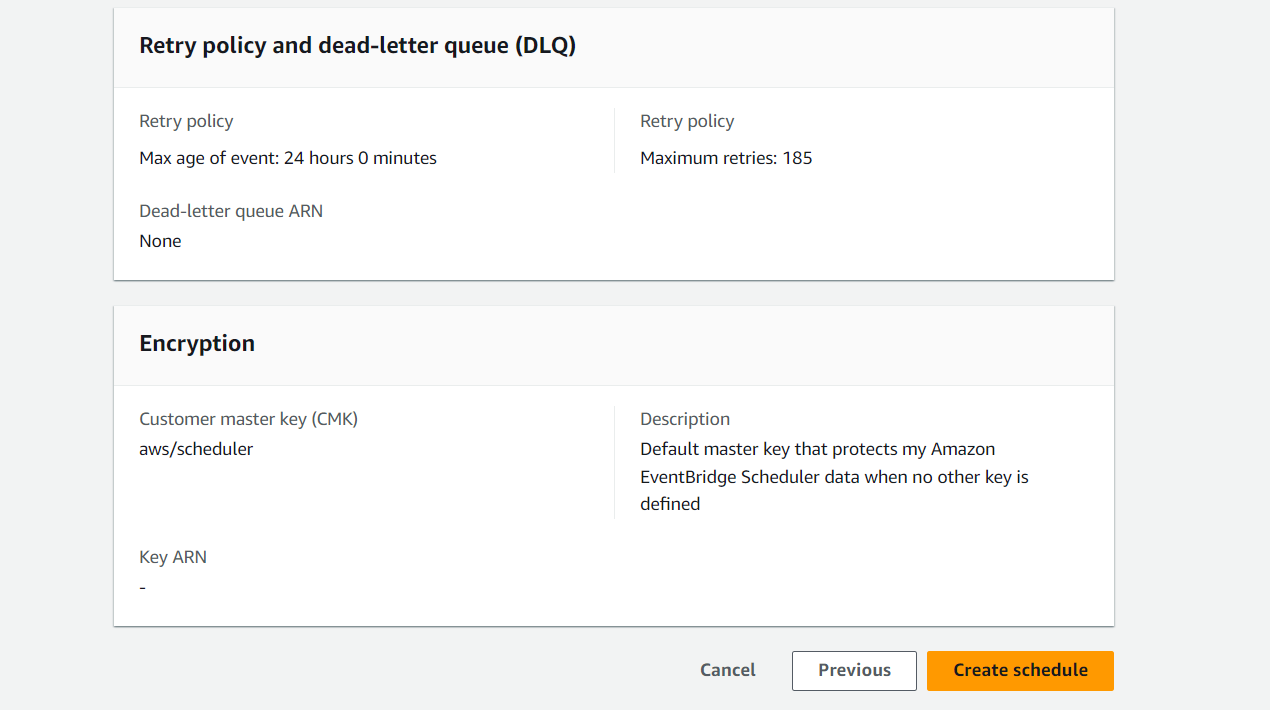
**Figure 3.28 Review of Target details**



**Figure 3.29 Schedule state and permission**



**Figure 3.30 Retry policy and dead letter queue**



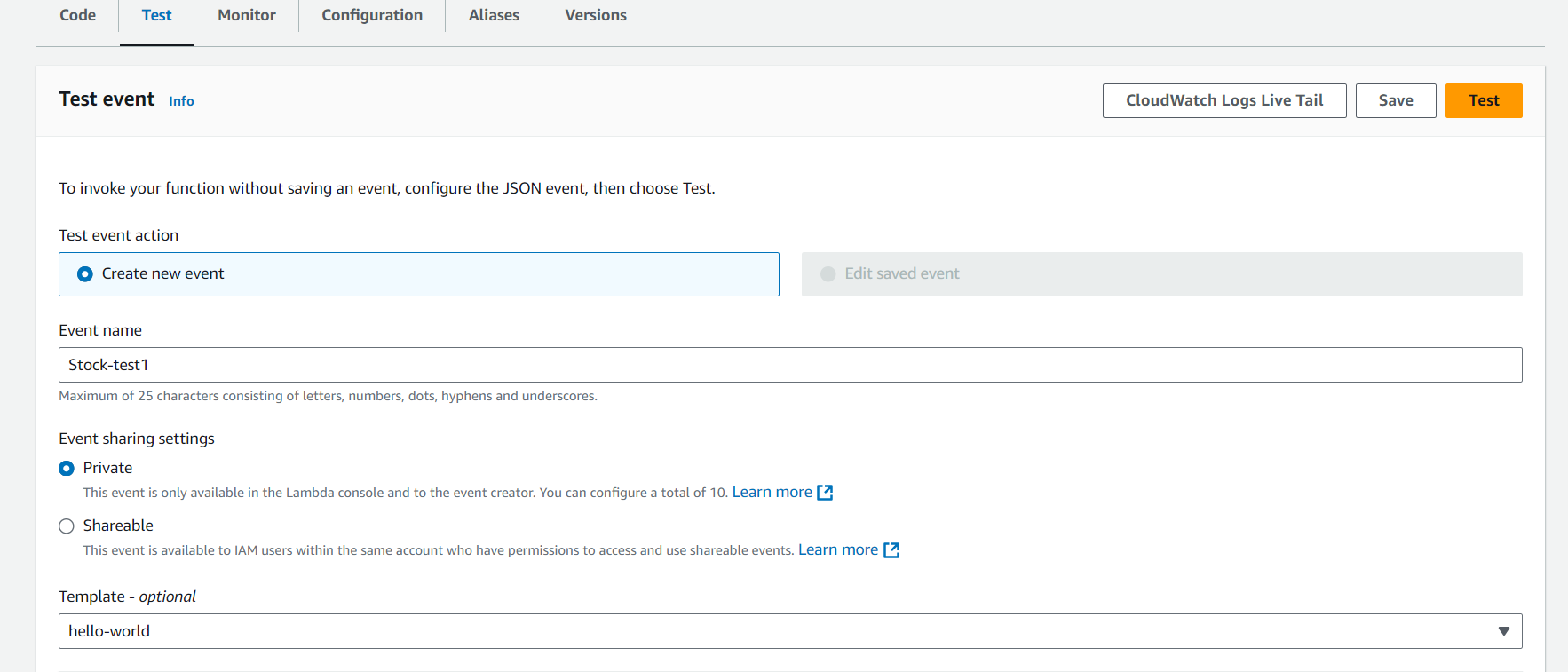
**Figure 3.31 Review of Encryption**



**Figure 3.32 Schedule is created**

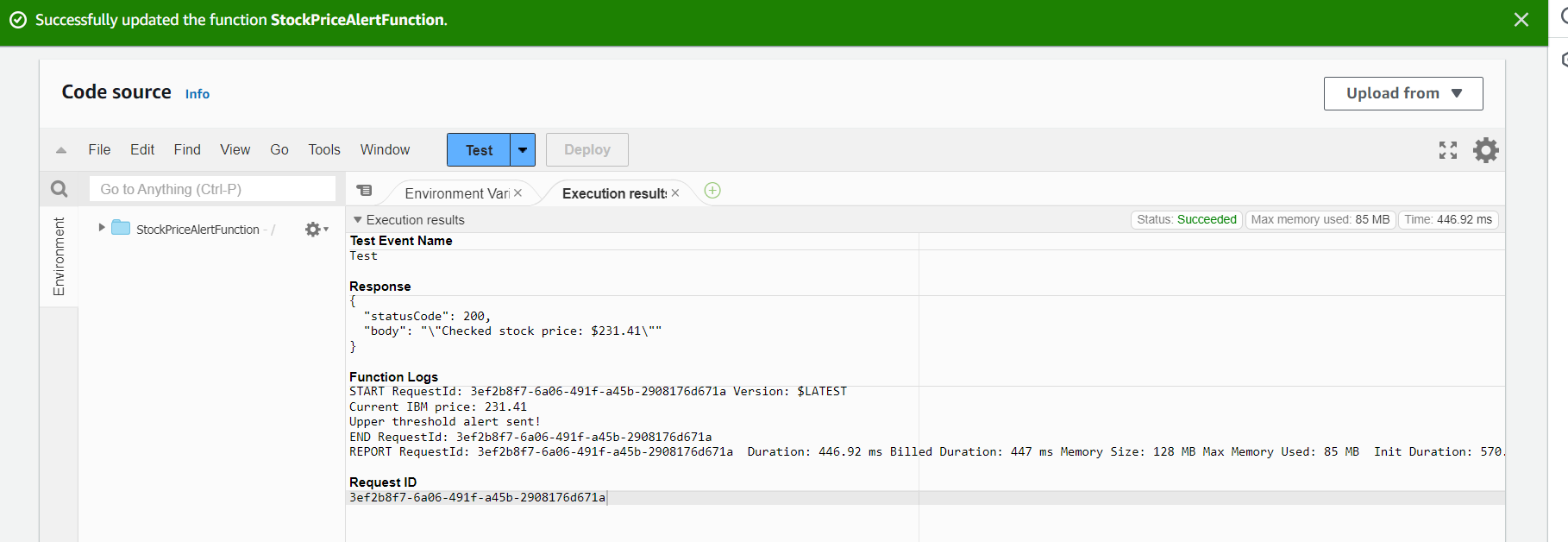
**STEP – 6: Test the System**

* In lambda console click on the Test button located at the top right corner of the function's detail page. If prompted, choose Configure test events. Select Create new test event. Fill a name and .Json file has to be empty.



**Figure 3.33 Test Event**

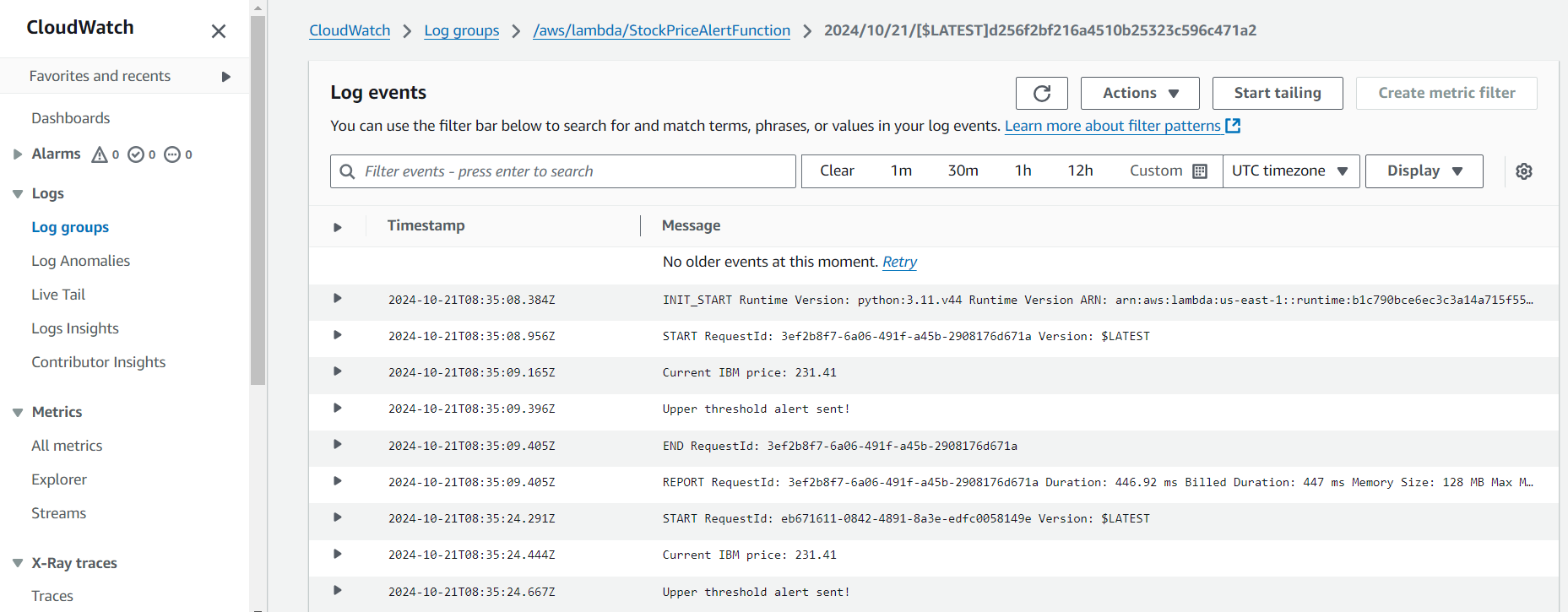
* Test the uploaded file in the lambda function by choosing the test option. So as a result, you will get the response code as 200 OK.



**Figure 3.34 Test result**

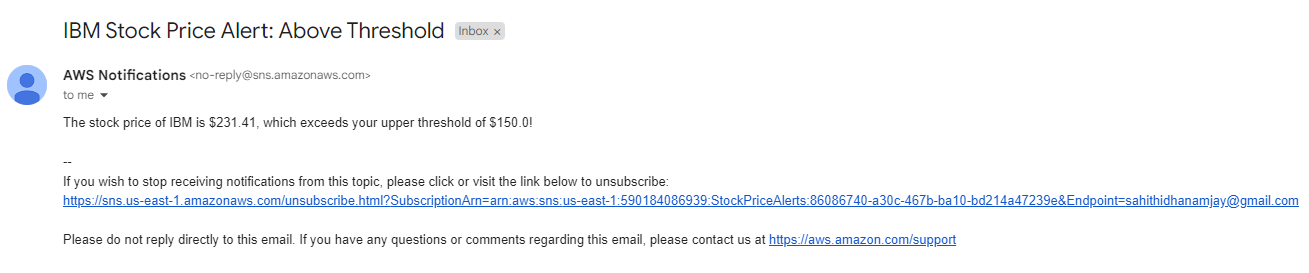
**STEP – 7: Monitoring in the Cloud Watch**

* Navigate to the **CloudWatch** Console and open the **Log** sections and choose the **log groups**. So here we can see the details of all the execution steps.

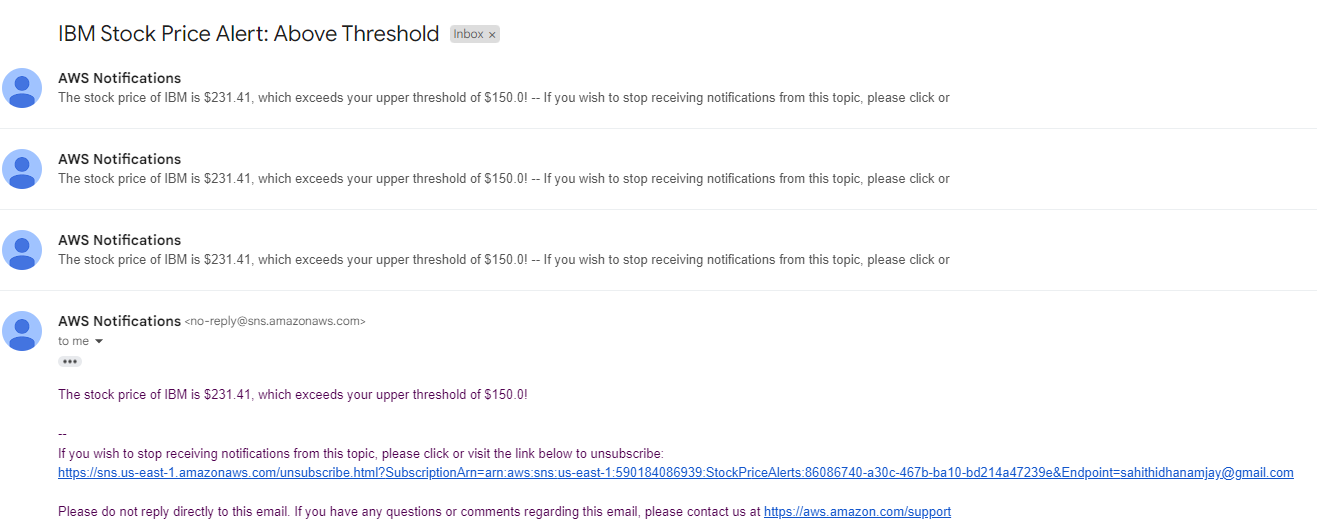


**Figure 3.35 Monitoring in Cloud Watch**

* As we have setup the time in cron expression as 5 minutes, after that file minutes we get the alert to your respective emails.



**Figure 3.36 Stock Price Alert**



**Figure 3.37 Stock Price Alert**

**CHAPTER 4**

**SOURCE CODE AND OUTPUT**

**LAMBDA FUNCTION CODE**

import json

import requests

import boto3

import os

# Set up SNS client

sns = boto3.client('sns')

# Set up stock API details

STOCK\_API\_KEY = os.getenv("STOCK\_API\_KEY", "demo")

STOCK\_SYMBOL = os.getenv("STOCK\_SYMBOL", "IBM")

# SNS topic ARN (replace with your actual ARN)

SNS\_TOPIC\_ARN = os.getenv("SNS\_TOPIC\_ARN", "arn:aws:sns:us-east-

1:590184086939:StockPriceAlerts")

# Threshold values

UPPER\_THRESHOLD = float(os.getenv("UPPER\_THRESHOLD", 150.00))

LOWER\_THRESHOLD = float(os.getenv("LOWER\_THRESHOLD", 100.00))

def get\_stock\_price(symbol):

# Alpha Vantage API URL

url = f"https://www.alphavantage.co/query?function=TIME\_SERIE

S\_INTRADAY&symbol={STOCK\_API\_KEY}&interval=1min&apikey={STOCK\_TOPIC

\_ARN}"

response = requests.get(url)

data = response.json()

# Extract the latest stock price

time\_series = data.get("Time Series (1min)", {})

if not time\_series:

return None

latest\_time = sorted(time\_series.keys())[0]

latest\_price = float(time\_series[latest\_time]["1. open"])

return latest\_price

def lambda\_handler(event, context):

# Get the current stock price

stock\_price = get\_stock\_price(STOCK\_SYMBOL)

if stock\_price is None:

print("Could not fetch stock price")

return {"statusCode": 500, "body": json.dumps("Stock price fetch

failed")}

print(f"Current {STOCK\_SYMBOL} price: {stock\_price}")

# Check if the stock price exceeds the upper threshold

if stock\_price > UPPER\_THRESHOLD:

message = f"The stock price of {STOCK\_SYMBOL} is ${stock\_price},

which exceeds your upper threshold of ${UPPER\_THRESHOLD}!"

# Send an SNS notification for upper threshold

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=message,

Subject=f"{STOCK\_SYMBOL} Stock Price Alert: Above Threshold")

print("Upper threshold alert sent!")

# Check if the stock price falls below the lower threshold

elif stock\_price < LOWER\_THRESHOLD:

message = f"The stock price of {STOCK\_SYMBOL} is

${stock\_price}, which is below your lower threshold of ${LOWER\_

THRESHOLD}!"

# Send an SNS notification for lower threshold

sns.publish(

TopicArn=SNS\_TOPIC\_ARN,

Message=message,

Subject=f"{STOCK\_SYMBOL} Stock Price Alert: Below

Threshold")

print("Lower threshold alert sent!")

else:

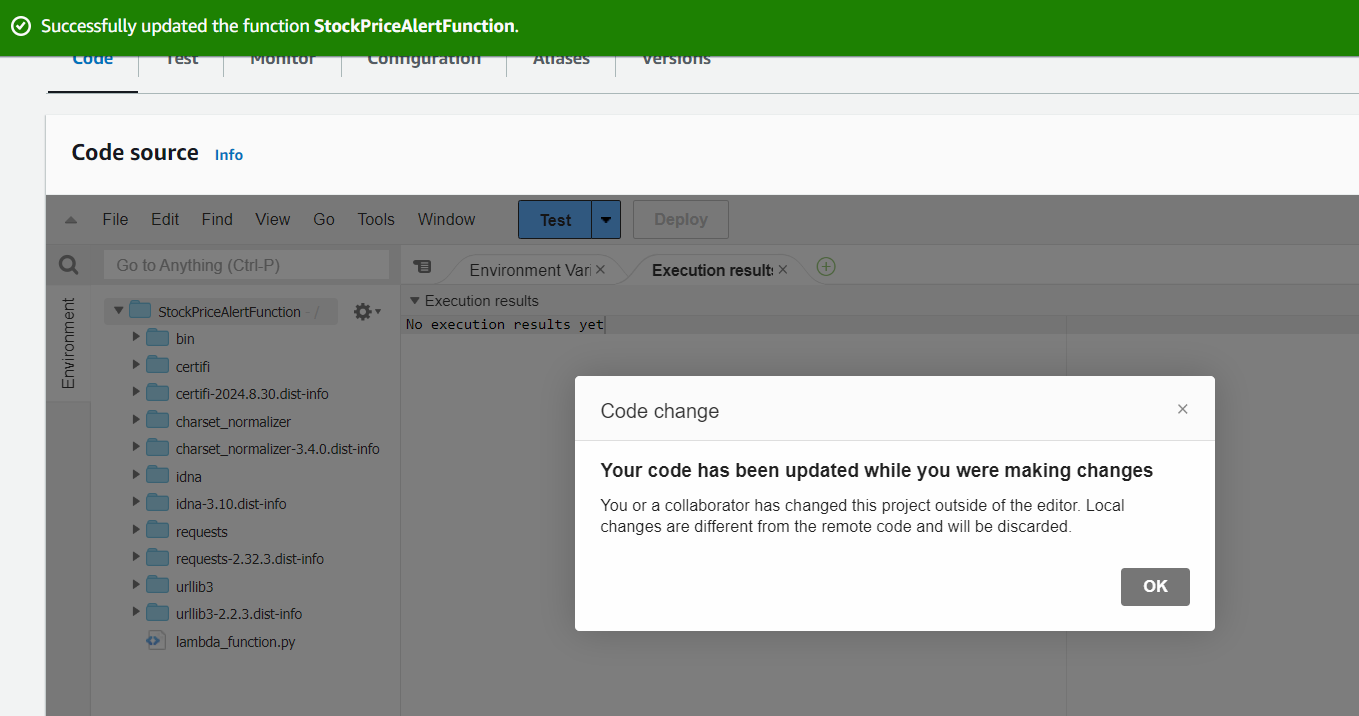
print(f"The stock price of {STOCK\_SYMBOL} is within the threshold

range.")

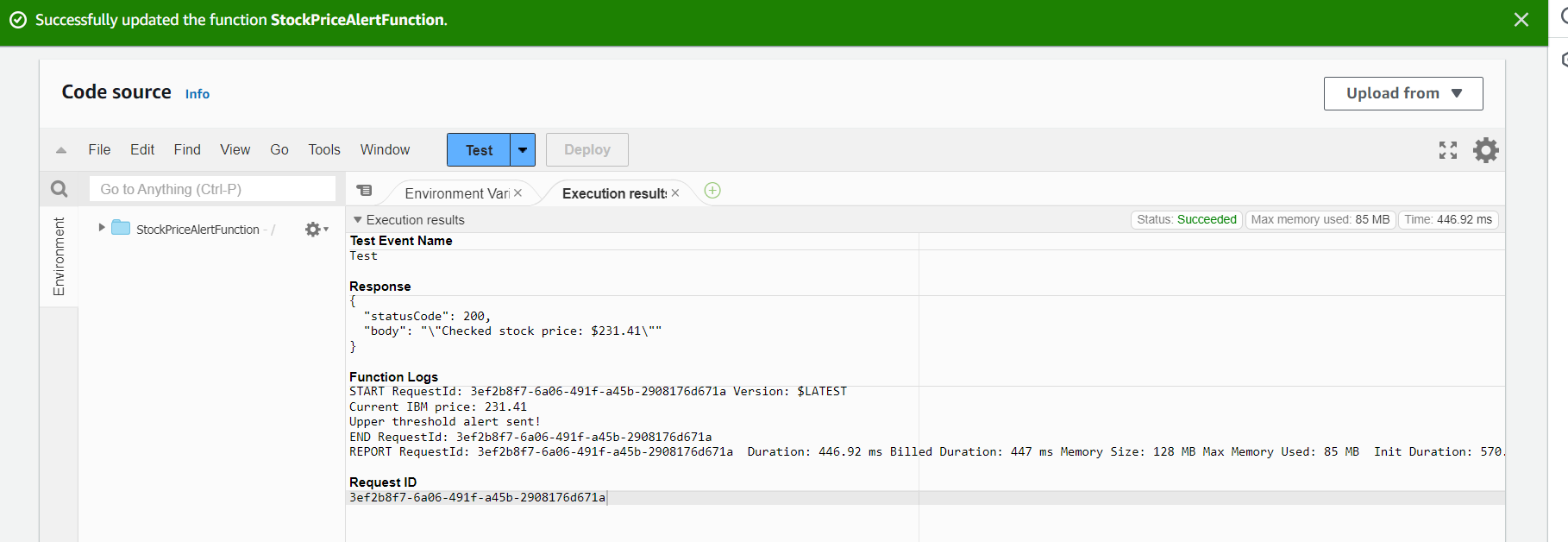
return {"statusCode": 200, "body": json.dumps(f"Checked stock

price: ${stock\_price}")}

**OUTPUT:**



**Figure 38 Code uploading**

****

**Figure 39 Result status Code**

**CHAPTER 5**

**CHALLENEGES FACED**

* **At the start of the project**, understanding the intricacies of integrating APIs like Alpha Vantage and using AWS SNS for stock price alerts can be quite challenging.
* **Managing multiple stocks** and keeping track of the alert thresholds (upper and lower limits) can be difficult. It’s essential to develop a well-structured strategy to handle multiple stocks and ensure the alerts are properly labelled and categorized for easy management.
* I encountered issues in the lambda function where the code written in the python environment of local system can’t identify some modules, so upload from local system lead for an error in the testing of the lambda function.
* **Planning the stock price checks at regular intervals** while minimizing API call limits and maintaining performance was a challenge.
* **Regularly testing the alerts** to ensure they trigger at the right stock price and send notifications in a timely manner proved to be a key challenge.
* **Automating the process of checking stock prices**, sending alerts, and managing notifications according to predefined conditions required overcoming challenges related to scheduling and retention of alerts.
* **There were challenges with network connections** and integrating with external services, such as API calls.
* **Occasional internet or network connectivity issues** were part of the project, sometimes affecting the reliability of the stock price alert system.
* Despite these challenges, gaining **hands-on experience** with real-time stock price alerts, troubleshooting API integrations, and optimizing the system for performance provided valuable learning and contributed significantly to the overall development of my technical skills.