AWS Blog-Generator Project Report

ARNAV SHARMA - 22BIT0496 / YASH BEHRA – 22BIT0304 / YASHA LALL – 22BIT0156

Introduction

This document outlines a project that uses various AWS services to create an automated blog generator. Leveraging AWS Bedrock, Lambda, S3, API Gateway, and CloudWatch, the project generates a blog based on a specified topic, saves it in an S3 bucket, and tracks the process through CloudWatch. This setup enables scalable and automated blog creation with easy access and monitoring via AWS.

Project Components

- **AWS Bedrock Meta Llama 3.2 70B Vision Instruct**: Provides the generative model to create blog content.
- **AWS S3**: Stores the blog output as a .txt file in a structured storage bucket.
- **AWS Lambda**: Executes the main code for blog generation, storage, and logging.
- **AWS API Gateway**: Exposes a RESTful API to trigger blog creation via Lambda.
- **AWS CloudWatch**: Monitors Lambda function logs and tracks the request flow.

Step-by-Step Setup

1. AWS Bedrock - Blog Generation Model

- **Model**: The Meta Llama 3.2 70B Vision Instruct is used to generate high-quality, AI-based blog content. This model allows for natural language generation with complex instructions.
- **Prompt and Parameters**: The model's prompt-based design allows it to accept specific instructions on blog topics, output length, and response style. By adjusting temperature and top_p parameters, the content's creativity and word choice diversity are controlled, making it suitable for varied blog topics. This dynamic prompt system enables customization of the AI response to suit different content needs effectively.

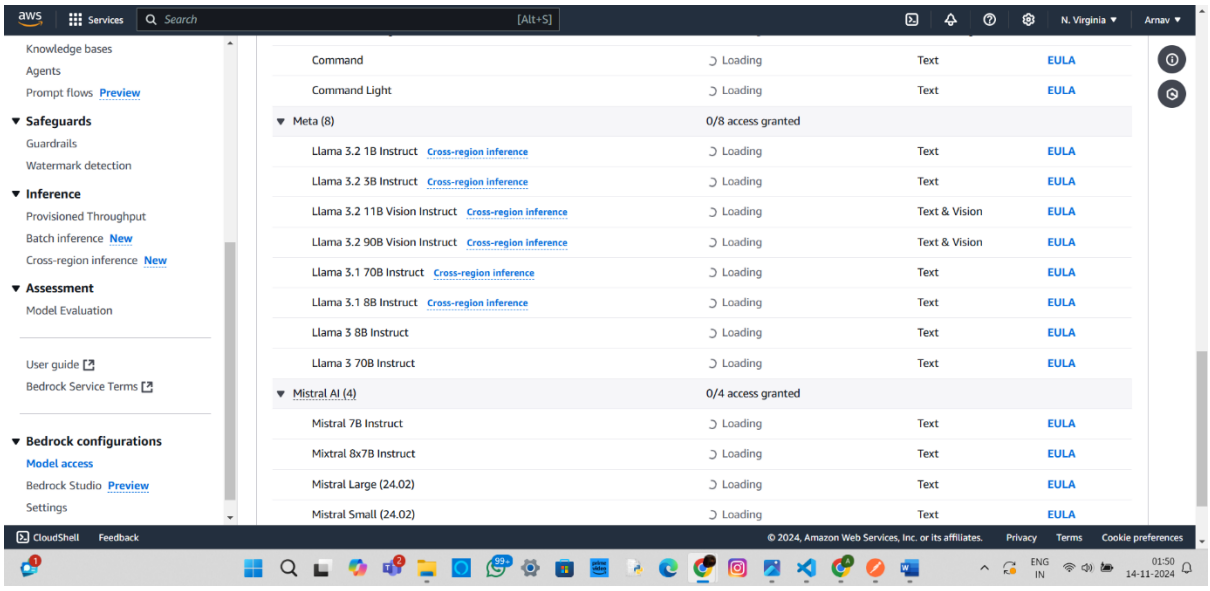
WORKING-

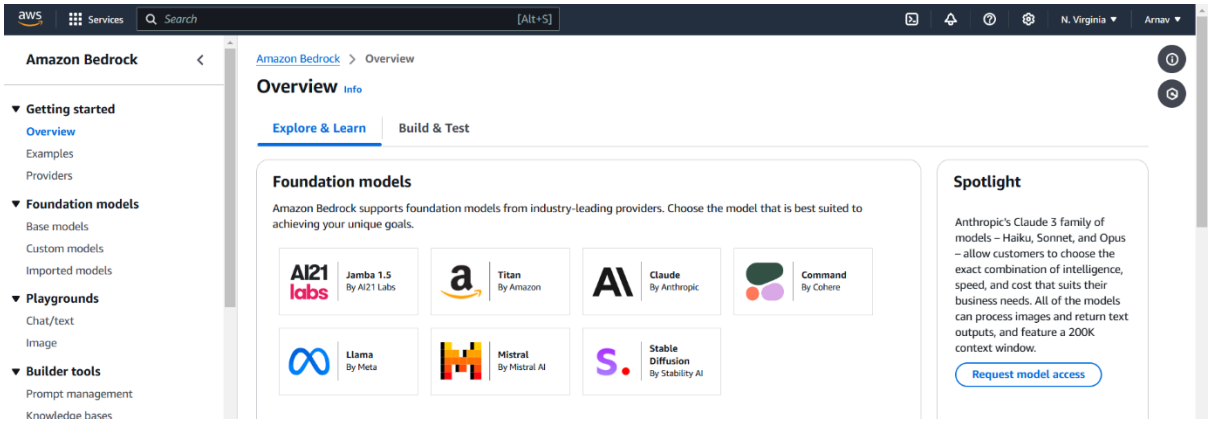
 Request Access: Access was requested and granted to AWS Bedrock.

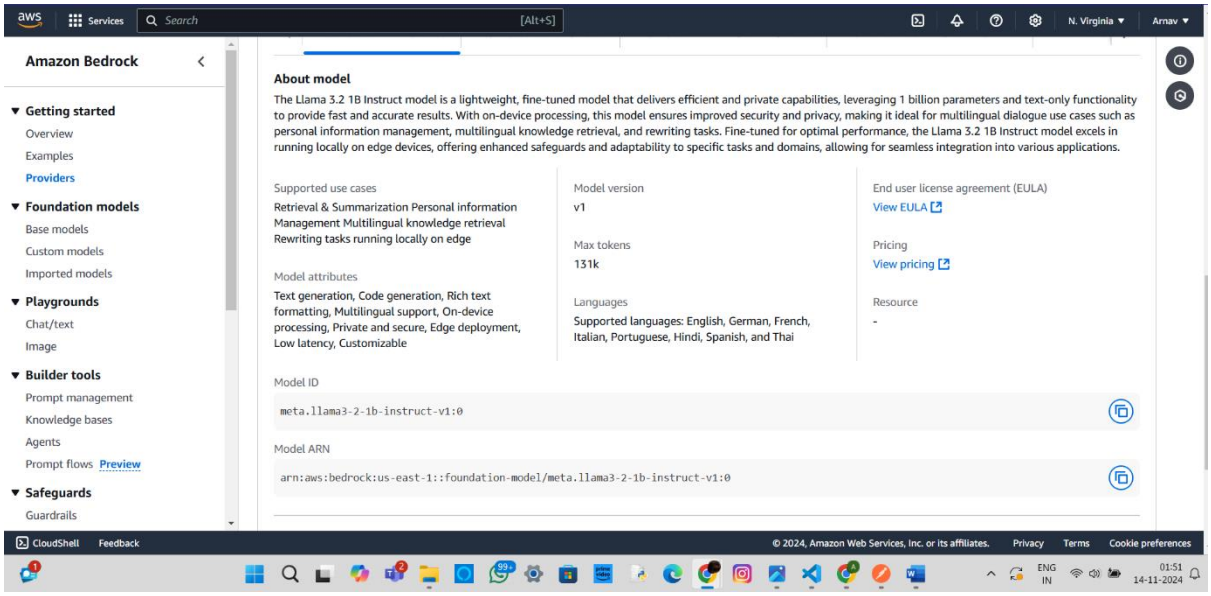
 Change Region: Switch to the North East Virginia (eu-north-1) region to utilize the Bedrock model.

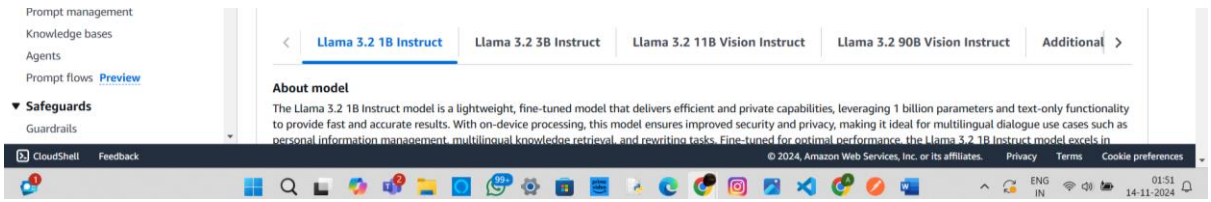
 Model ID Configuration:

- Model used: **Meta Llama 3.2 70B Vision Instruct**.
- Model ID: meta.llama3-70b-instruct-v1:0.
- The model ID was integrated into the code to use Bedrock's capabilities for blog generation.


aws
Services
Search
[Alt+S]
N. Virginia
Arnav
Knowledge bases
Agents
Prompt flows Preview
Safeguards
Guardrails
Watermark detection
Inference
Provisioned Throughput
Batch inference New
Cross-region inference New
Assessment
Model Evaluation
User guide
Bedrock Service Terms
Bedrock configurations
Model access
Bedrock Studio Preview
Settings
Command Loading Text EULA
Command Light Loading Text EULA
Meta (8) 0/8 access granted
Llama 3.2 1B Instruct Cross-region inference Loading Text EULA
Llama 3.2 3B Instruct Cross-region inference Loading Text EULA
Llama 3.2 11B Vision Instruct Cross-region inference Loading Text & Vision EULA
Llama 3.2 90B Vision Instruct Cross-region inference Loading Text & Vision EULA
Llama 3.1 70B Instruct Cross-region inference Loading Text EULA
Llama 3.1 8B Instruct Cross-region inference Loading Text EULA
Llama 3 8B Instruct Loading Text EULA
Llama 3 70B Instruct Loading Text EULA
Mistral AI (4) 0/4 access granted
Mistral 7B Instruct Loading Text EULA
Mixtral 8x7B Instruct Loading Text EULA
Mistral Large (24.02) Loading Text EULA
Mistral Small (24.02) Loading Text EULA
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
ENG IN
01:50
14-11-2024


aws
Services
Search
[Alt+S]
N. Virginia
Arnav
Amazon Bedrock
Getting started
Overview
Examples
Providers
Foundation models
Base models
Custom models
Imported models
Playgrounds
Chat/text
Image
Builder tools
Prompt management
Knowledge bases
Amazon Bedrock > Overview
Overview Info
Explore & Learn
Build & Test
Foundation models
Amazon Bedrock supports foundation models from industry-leading providers. Choose the model that is best suited to achieving your unique goals.
Jamba 1.5 By AI21 Labs
Titan By Amazon
Claude By Anthropic
Command By Cohere
Llama By Meta
Mistral By Mistral AI
Stable Diffusion By Stability AI
Spotlight
Anthropic's Claude 3 family of models – Haiku, Sonnet, and Opus – allow customers to choose the exact combination of intelligence, speed, and cost that suits their business needs. All of the models can process images and return text outputs, and feature a 200K context window.
Request model access


aws
Services
Search
[Alt+S]
N. Virginia
Arnav
Amazon Bedrock
Getting started
Overview
Examples
Providers
Foundation models
Base models
Custom models
Imported models
Playgrounds
Chat/text
Image
Builder tools
Prompt management
Knowledge bases
Agents
Prompt flows Preview
Safeguards
Guardrails
About model
The Llama 3.2 1B Instruct model is a lightweight, fine-tuned model that delivers efficient and private capabilities, leveraging 1 billion parameters and text-only functionality to provide fast and accurate results. With on-device processing, this model ensures improved security and privacy, making it ideal for multilingual dialogue use cases such as personal information management, multilingual knowledge retrieval, and rewriting tasks. Fine-tuned for optimal performance, the Llama 3.2 1B Instruct model excels in running locally on edge devices, offering enhanced safeguards and adaptability to specific tasks and domains, allowing for seamless integration into various applications.
Supported use cases
Retrieval & Summarization Personal information Management Multilingual knowledge retrieval Rewriting tasks running locally on edge
Model attributes
Text generation, Code generation, Rich text formatting, Multilingual support, On-device processing, Private and secure, Edge deployment, Low latency, Customizable
Model version
v1
Max tokens
131k
Languages
Supported languages: English, German, French, Italian, Portuguese, Hindi, Spanish, and Thai
End user license agreement (EULA)
View EULA
Pricing
View pricing
Resource
-
Model ID
meta.llama3-2-1b-instruct-v1:0
Model ARN
arn:aws:bedrock:us-east-1::foundation-model/meta.llama3-2-1b-instruct-v1:0
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
ENG IN
01:51
14-11-2024



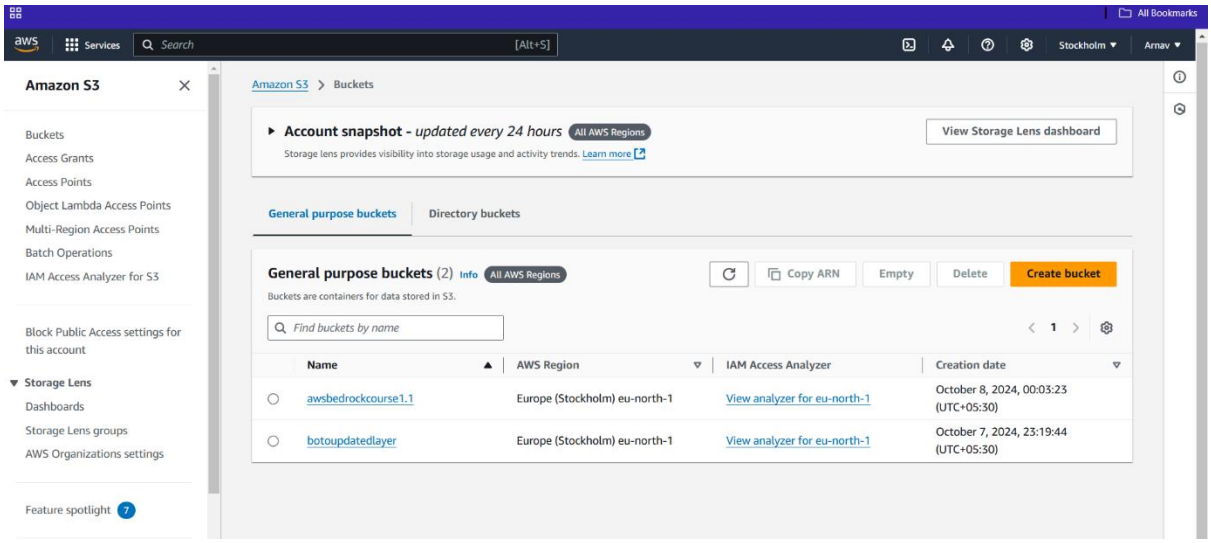
2. AWS S3 - Blog Storage

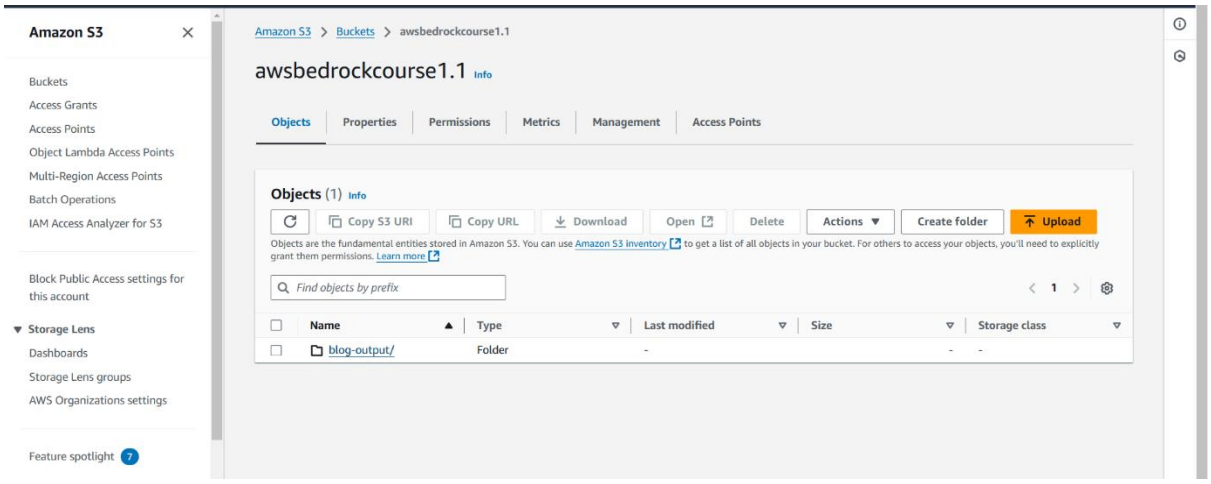
- **Bucket Setup**: An S3 bucket named **awsbedrockcourse1.1** was created to securely store the generated blogs.
- **Usage**: This bucket acts as a persistent storage solution where each blog is saved with a unique key based on the current timestamp. S3 provides highly scalable and durable storage, ensuring that each generated blog is saved reliably and can be accessed, downloaded, or processed further. By saving blogs as .txt files in a specified path within the bucket, S3 allows organized storage and easy retrieval of content.

WORKING-

 Bucket Creation: Created an AWS S3 bucket named **awsbedrockcourse1.1**.

 Purpose: Stores the blog output as a .txt file named output-blog.txt.





3. AWS Lambda - Blog Generation Logic

- **Lambda Function**: A Lambda function, named awsbedrockapi, was created to manage the core functionality of blog generation, storage in S3, and logging.
    - **Layer and Permissions**: Since the default Lambda environment didn't include the boto3 library, a custom layer boto3updatedlayer was added. This layer allowed Lambda to interact with other AWS services, like Bedrock and S3. The function was also assigned **Administrator Access**, ensuring full permissions to create, update, and delete resources as needed for seamless blog creation and storage.
    - **Deployment**: The Lambda function, once coded, was deployed and triggered by API requests, linking the generation, saving, and logging tasks in one pipeline.

WORKING-

 Lambda Function Creation:

- Created a Lambda function named **awsbedrockapi**.
- Configured to use Python 3.13/3.12/3.11 runtime.
- Deployed the function to process requests and handle blog generation logic.

 Code Implementation:

- Implemented the code in the Lambda function.
- Configured to accept a topic input, generate a blog, and store the result in S3.

 Adding Boto3 Library:

- To work with S3 and other AWS services, the Boto3 library was required.
- Added a Lambda layer named **boto3updatedlayer**, containing the Boto3 package in a zip file.

 Permissions and Access:

- Gave the Lambda function **Administrator Access** for full permissions on AWS services.

 Deployment:

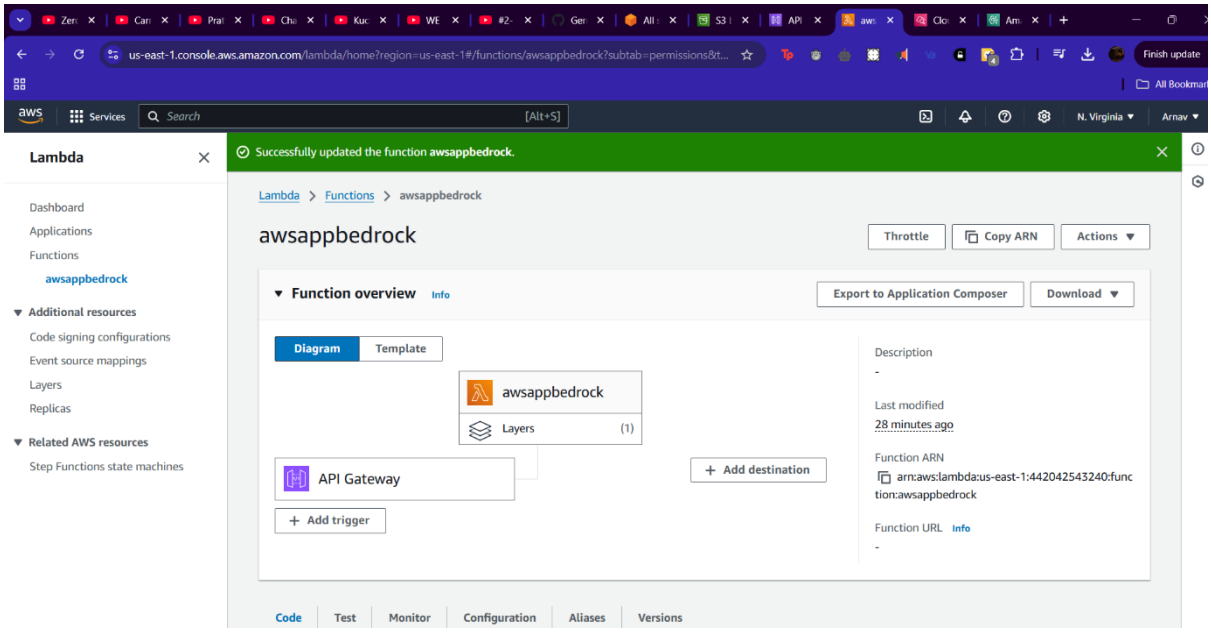
- The code, along with layers and permissions, was deployed for production.

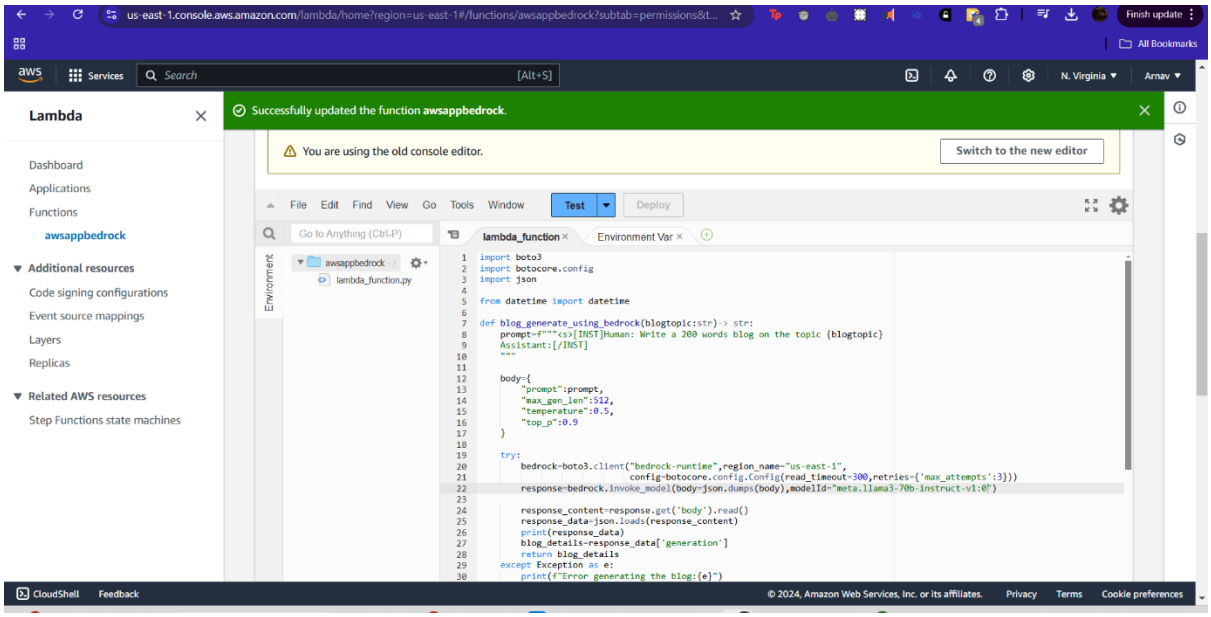
CODE-

```
import boto3
import botocore.config
import json

from datetime import datetime
```

```
def blog_generate_using_bedrock(blogtopic:str)-> str:
    prompt=f"""<s>[INST]Human: Write a 200 words blog on the topic {blogtopic}
    Assistant:[/INST]
    """

    body={
        "prompt":prompt,
        "max_gen_len":512,
        "temperature":0.5,
        "top_p":0.9
    }

    try:
        bedrock=boto3.client("bedrock-runtime",region_name="us-east-1",
                                config=botocore.config.Config(read_timeout=300,re
tries={'max_attempts':3}))
        response=bedrock.invoke_model(body=json.dumps(body),modelId="meta.llam
a2-13b-chat-v1")

        response_content=response.get('body').read()
        response_data=json.loads(response_content)
        print(response_data)
        blog_details=response_data['generation']
        return blog_details
    except Exception as e:
        print(f"Error generating the blog:{e}")
        return ""

def save_blog_details_s3(s3_key,s3_bucket,generate_blog):
    s3=boto3.client('s3')

    try:
        s3.put_object(Bucket = s3_bucket, Key = s3_key, Body =generate_blog )
        print("Code saved to s3")

    except Exception as e:
        print("Error when saving the code to s3")



def lambda_handler(event, context):
    # TODO implement
    event=json.loads(event['body'])
    blogtopic=event['blog_topic']

    generate_blog=blog_generate_using_bedrock(blogtopic=blogtopic)

    if generate_blog:
        current_time=datetime.now().strftime('%H%M%S')
```

```
        s3_key=f"blog-output/{current_time}.txt"
        s3_bucket='aws_bedrock_course1'
        save_blog_details_s3(s3_key,s3_bucket,generate_blog)


    else:
        print("No blog was generated")

    return{
        'statusCode':200,
        'body':json.dumps('Blog Generation is completed')
    }
```





us-east-1.console.aws.amazon.com/lambda/home?region=us-east-1#/functions/awsappbedrock?subtab=permissions&t...

Lambda

Dashboard
Applications
Functions
awsappbedrock
Additional resources
Code signing configurations
Event source mappings
Layers
Replicas
Related AWS resources
Step Functions state machines

Successfully updated the function **awsappbedrock**.

Encryption with AWS KMS customer managed KMS key Info

To edit customer managed key encryption, you must upload a new .zip deployment package. Learn more

AWS KMS key ARN	Key alias	Status
-	-	-

Runtime settings Info — Edit | Edit runtime management configuration

Runtime	Handler Info	Architecture Info
Python 3.12	lambda_function.lambda_handler	x86_64

Runtime management configuration

Layers Info — Edit | Add a layer

Merge order	Name	Layer version	Compatible runtimes	Compatible architectures	Version ARN
1	Boto3UpdatedLayer	1	python3.13, python3.11, python3.12, python3.10	-	arn:aws:lambda:us-

us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/details/awsappbedrock-role-kvon74td?section=...

Identity and Access Management (IAM)

Search IAM

Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings
Access reports
Access Analyzer
External access
Unused access
Analyzer settings
Credential report

Policy was successfully attached to role.

Last activity	Maximum session duration
-	1 hour

Permissions | Trust relationships | Tags | Last Accessed | Revoke sessions

Permissions policies (2) Info

You can attach up to 10 managed policies.

Simulate | Remove | Add permissions

Filter by Type: All types

Policy name	Type	Attached entities
AdministratorAccess	AWS managed - job function	2
AWSLambdaBasicExecutionRole-764cdf3...	Customer managed	1

Permissions boundary (not set)

Destinations
Function URL
Environment variables
Tags
VPC
RDS databases

awsappbedrock-role-03mtpy97

Resource summary

To view the resources and actions that your function has permission to access, choose a service.

All
1 action, 1 resource

By action | By resource

4. AWS API Gateway - Creating API for Lambda

- **API Name**: bedrockldemoapi
    - **Route and Stage Setup**: A POST route /blog-generation was created, providing a RESTful interface for accessing the blog generator. By creating a dev stage and deploying it, the API Gateway enabled direct access to the Lambda function.
    - **Endpoint URL**: This URL allows users to send HTTP POST requests to initiate blog generation, making the service accessible externally. API Gateway streamlines the process of calling the Lambda function while providing secure, scalable API management, handling authentication, throttling, and traffic monitoring automatically.

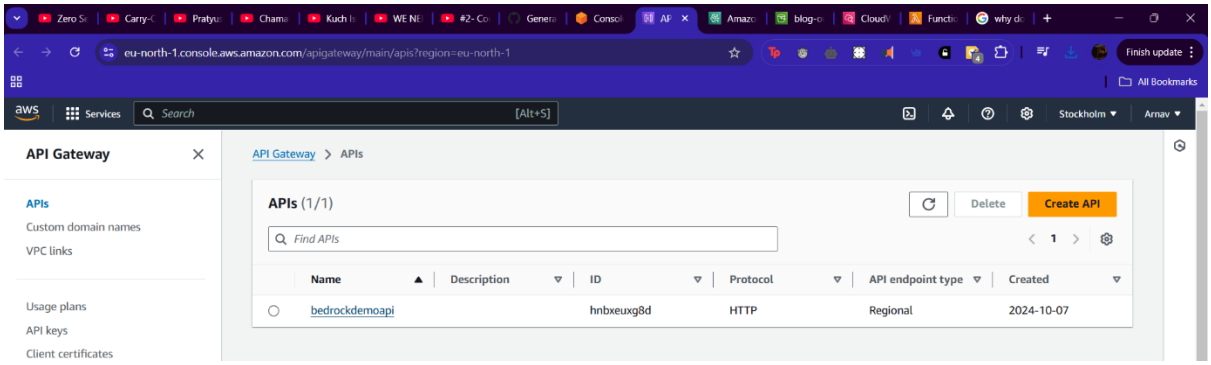
WORKING-

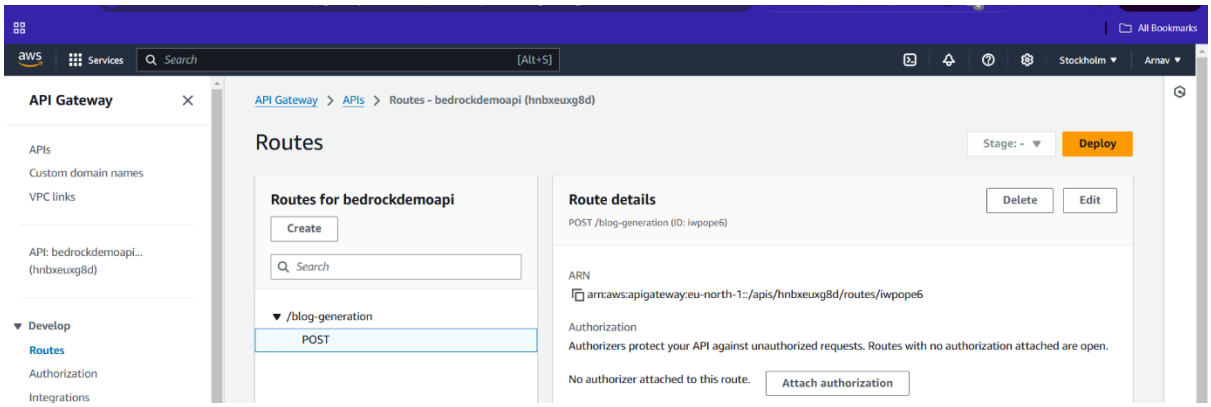
 API Creation:

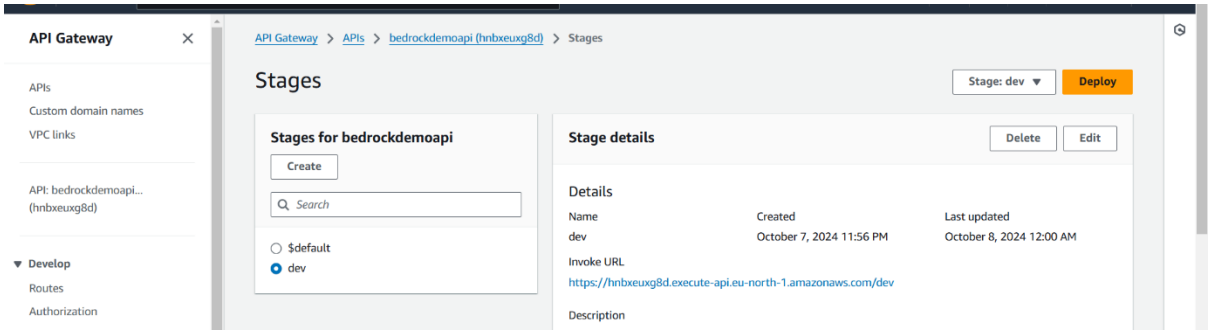
- Created an API Gateway named **bedrockldemoapi**.
- Configured a POST route called /blog-generation for API requests.

 Stage Creation and Deployment:

- Created a stage named dev and deployed the API to this stage.
- Copied the generated URL to use in external requests.







5. Testing with Postman

- **HTTP Method**: Using POST in Postman, the API endpoint is triggered to generate a blog on a given topic.
    - **Body**: The JSON payload specifies the blog_topic, such as "MS Dhoni," which guides the content generation in Bedrock.
    - **S3 Verification**: The generated blog is saved in the specified S3 bucket path. Postman serves as a testing tool to validate that the API, Lambda function, and S3 storage work seamlessly together.
    - **CloudWatch Check**: In AWS CloudWatch, logs of the Lambda function's execution provide insights into any errors, response times, and request handling. This step ensures that all processes function as expected and helps debug any issues.

WORKING-

 Configuration:

- Selected POST method.
- Entered the API URL: https://6vcio72x2g.execute-api.us-east-1.amazonaws.com/dev/blog-generation.

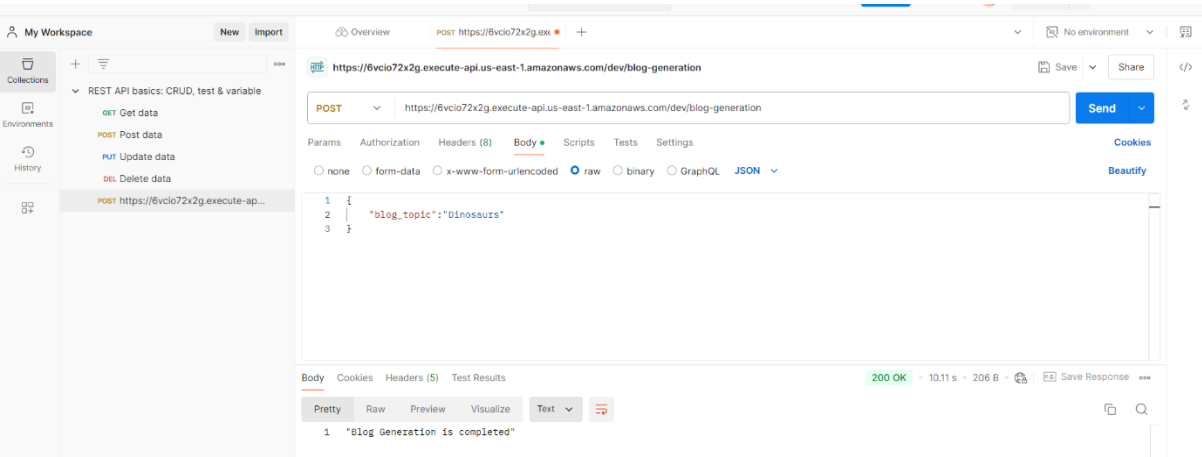
 Request Body:

- JSON format with the topic:

```
{
  "blog_topic": "Dinosaurs"
}
```

 Execution:

- Sent the request to the API.
- Monitored AWS CloudWatch and verified the generated blog output in S3.

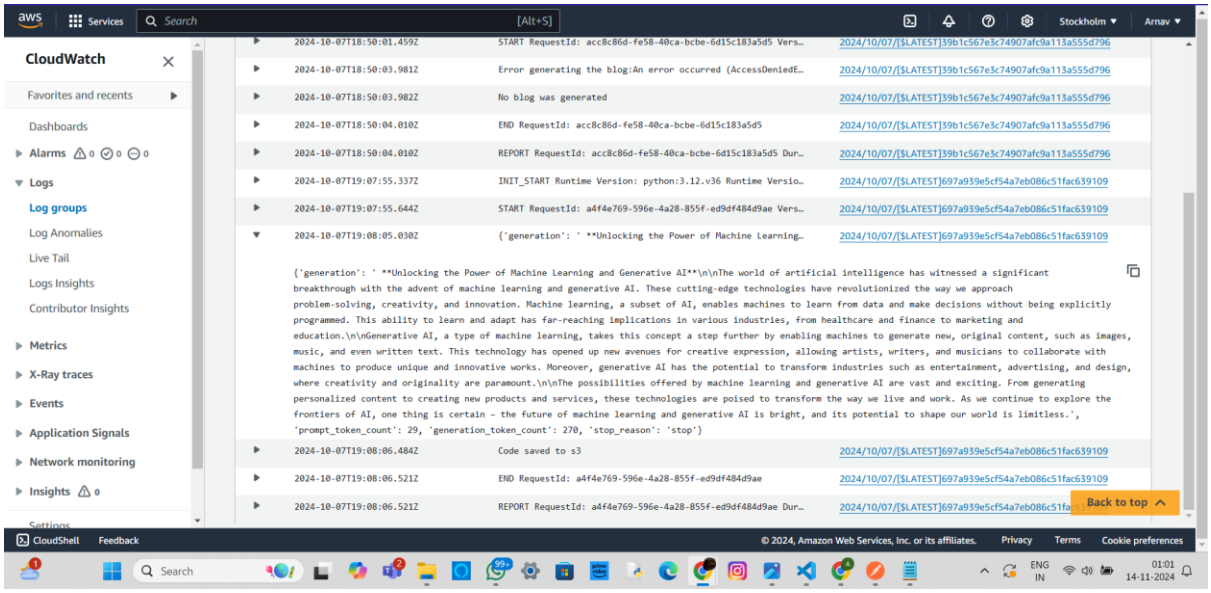


6. AWS CloudWatch - Log Monitoring

AWS CloudWatch is utilized to monitor logs from the Lambda function, allowing real-time tracking of the blog generation process. By observing these logs, it is possible to confirm that each request was processed correctly, and that the generated blog content was successfully stored in S3. CloudWatch provides crucial insights for debugging, tracking performance, and ensuring reliable operation, helping identify any errors in API responses or Lambda execution.

WORKING-

- Used AWS CloudWatch to monitor the Lambda function's logs.
- Confirmed that the blog was generated and stored in the S3 bucket as a .txt file.



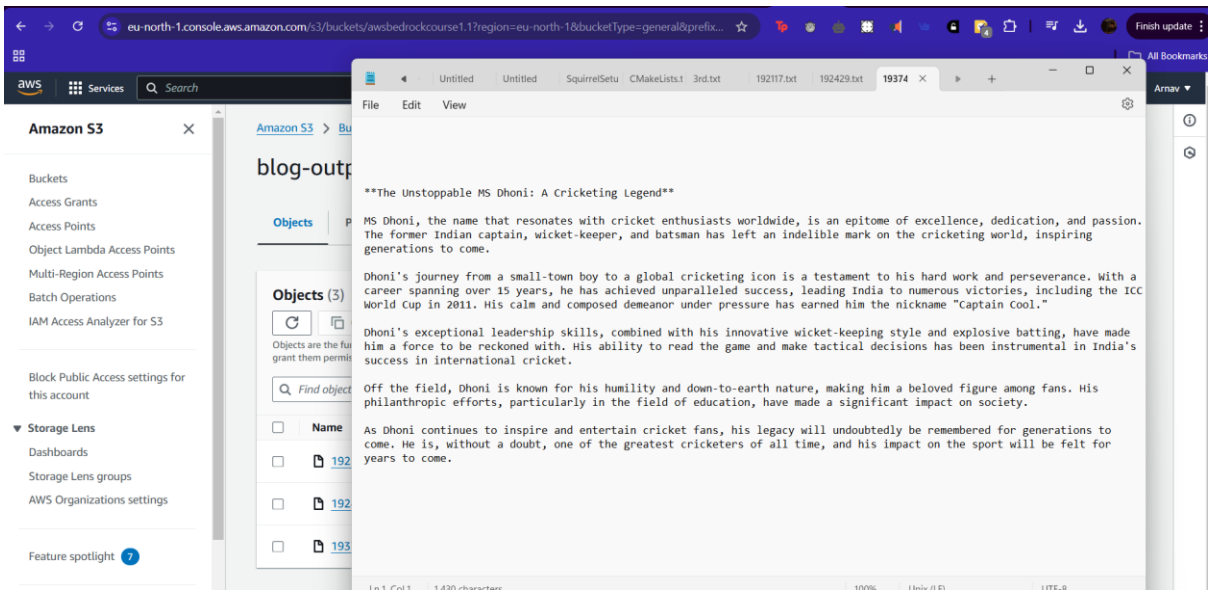
7. AWS CS3 Bucket – Output File

AWS S3 securely stores the generated blog as a .txt file in a structured bucket, allowing organized storage and retrieval of blog content. The bucket setup for this project enables quick verification of output files and ensures persistent access to each generated blog. S3's scalable and durable storage

infrastructure makes it an ideal solution for archiving content, supporting seamless access, sharing, or further processing of saved blog files.

WORKING-

- Used AWS S3 to view the output text file, which contains the blog on our given topic
- Confirmed that the blog was generated and stored in the S3 bucket as a .txt file.



Summary

This AWS Blog Generator project demonstrates a scalable and efficient way to automate blog generation using AWS services. By leveraging Bedrock's powerful language model, S3 for secure storage, Lambda for processing, API Gateway for user interaction, and CloudWatch for monitoring, this setup is both powerful and adaptable, supporting a wide range of topics and storage requirements.