```

c2m-generative-ai-marketing-portal

├─ .flake8

├─ .gitignore

├─ .mypy\_cache

├─ .swm

├─ CODE\_OF\_CONDUCT.md

├─ CONTRIBUTING.md

├─ LICENSE

***├─ app.py***

This code is a CDK (AWS Cloud Development Kit) application that defines and deploys an AWS infrastructure stack. It leverages a YAML configuration file to configure the stack and applies best-practice checks (via CDK Nag) to ensure compliance with AWS security and operational recommendations. Let's break down each section of the code:

**1. Imports**

* **os**: Used to retrieve environment variables, such as AWS account and region information.
* **Path**: Used to handle file paths in a cross-platform manner.
* **aws\_cdk as cdk**: The main library for defining and deploying AWS resources using CDK.
* **yaml**: Used to load configuration data from a YAML file.
* **NagSuppressions**: A part of CDK-Nag, a tool that checks your CDK stack for AWS security best practices and compliance. This helps identify issues like overly permissive IAM roles or lack of encryption.
* **SafeLoader**: The YAML loader to safely parse YAML files and avoid arbitrary code execution.
* **CDSGenAIStack**: A custom class (presumably defined in the infra/cdsgenai\_stack.py file) representing the infrastructure stack.

**2. Load Stack Configuration**

The following block loads configuration values from a YAML file (config.yml), which is located in the same directory as the current script:

python

Copy code

with open(os.path.join(Path(\_\_file\_\_).parent, "config.yml"), "r", encoding="utf-8") as ymlfile:

stack\_config = yaml.load(ymlfile, Loader=SafeLoader)

* **os.path.join** and **Path(\_\_file\_\_).parent**: These functions construct the path to the config.yml file relative to the current file.
* **yaml.load**: This loads the contents of the YAML configuration file into a Python dictionary, using SafeLoader for safety.

**3. Initialize CDK App and Environment**

python

Copy code

app = cdk.App()

env = cdk.Environment(account=os.getenv("CDK\_DEFAULT\_ACCOUNT"), region=os.getenv("CDK\_DEFAULT\_REGION"))

* **cdk.App()**: This initializes the CDK application, which serves as the root of the CDK construct tree.
* **cdk.Environment()**: Specifies the target AWS environment (account and region). The values for account and region are fetched from environment variables (CDK\_DEFAULT\_ACCOUNT and CDK\_DEFAULT\_REGION), which are typically provided during the deployment process.

**4. Instantiate the Stack**

python

Copy code

stack = CDSGenAIStack(scope=app, stack\_name=stack\_config["stack\_name"], config=stack\_config, env=env)

* **CDSGenAIStack**: This is a custom stack class defined elsewhere (presumably in the infra/cdsgenai\_stack.py file). It encapsulates the AWS infrastructure (resources like Lambda functions, S3 buckets, etc.).
  + **scope=app**: The stack is added to the app (root of the CDK app).
  + **stack\_name=stack\_config["stack\_name"]**: The stack's name is pulled from the config.yml file.
  + **config=stack\_config**: The stack is passed the entire configuration data loaded from the YAML file.
  + **env=env**: The stack is deployed in the specified environment (account and region).

**5. CDK Nag Setup (Optional)**

The code includes an option to enable CDK-Nag, which runs a set of compliance and best practice checks for the infrastructure:

python

Copy code

# Aspects.of(app).add(AwsSolutionsChecks())

* **CDK Nag**: This tool can be used to inspect the AWS infrastructure for best practices and compliance with AWS Solutions Architect guidelines (e.g., secure IAM policies, resource encryption). This line is commented out but can be uncommented to enable these checks.

**6. Apply Nag Suppressions**

python

Copy code

NagSuppressions.add\_stack\_suppressions(

stack,

[

{

"id": "AwsSolutions-IAM5",

"reason": "BucketDeployment only used to deploy sample data to s3",

},

{

"id": "AwsSolutions-IAM4",

"reason": "BucketDeployment only used to deploy sample data to s3",

},

{

"id": "AwsSolutions-L1",

"reason": "BucketDeployment only used to deploy sample data to s3",

},

],

apply\_to\_nested\_stacks=True,

)

* **NagSuppressions.add\_stack\_suppressions**: This function applies exceptions (suppressions) for specific AWS Solutions checks that CDK Nag would otherwise flag.
* **Common Suppression IDs**:
  + **AwsSolutions-IAM5**: Relates to overly permissive IAM policies (e.g., allowing wildcard actions or resources). This is suppressed here because the permissions are limited to deploying sample data to S3.
  + **AwsSolutions-IAM4**: Similar to IAM5, this ID is related to identity and access management concerns.
  + **AwsSolutions-L1**: Refers to the use of certain AWS Lambda best practices, likely relaxed in this context due to non-production data deployment.
* **apply\_to\_nested\_stacks=True**: This setting ensures that the suppressions are applied not only to the root stack but also to any nested stacks (if applicable).

**7. Synthesize the CDK App**

python

Copy code

app.synth()

* **app.synth()**: This command synthesizes the CDK app, generating the AWS CloudFormation template that represents the infrastructure. It prepares the stack for deployment by translating the high-level CDK constructs into the lower-level CloudFormation template.

**Summary**

This code:

1. Loads configuration details from a YAML file.
2. Initializes a CDK app and environment, using AWS account and region from environment variables.
3. Instantiates a custom infrastructure stack (CDSGenAIStack) with configuration values.
4. Optionally enables CDK Nag for compliance checks.
5. Applies specific suppressions to CDK Nag to bypass warnings for specific non-production cases.
6. Synthesizes the app to produce CloudFormation templates for deployment.

The overall goal is to define and deploy a GenAI stack in AWS, using best practices and suppressing certain warnings where justified.

***assets***

.***dockerignore***

The .dockerignore file is used to specify files and directories that should be excluded from the Docker build context when building Docker images. Its primary purpose is to reduce the size of the build context, improve build performance, and prevent unnecessary or sensitive files from being copied into the Docker image.

Here are the key benefits of using a .dockerignore file:

1. **Improved build speed**: By excluding unnecessary files (e.g., log files, local development files, etc.), the size of the build context is reduced, speeding up the Docker build process, especially when dealing with large projects.
2. **Smaller image size**: Avoiding the inclusion of irrelevant files in the image ensures that the resulting Docker image is smaller and contains only the essential files needed for running the container.
3. **Security**: You can exclude sensitive files such as configuration files, secrets, or credentials that should not be included in the final Docker image or shared with others.
4. **Preventing unintended behavior**: Sometimes, including certain files (like temporary files or editor-specific files) can cause issues in the Docker container. The .dockerignore file prevents such files from being copied into the image.

The syntax of the .dockerignore file is similar to .gitignore, where you can define patterns to include or exclude files and directories.

│ ├─ ***Dockerfile***

The .dockerignore file is used to specify files and directories that should be excluded from the Docker build context when building Docker images. Its primary purpose is to reduce the size of the build context, improve build performance, and prevent unnecessary or sensitive files from being copied into the Docker image.

Here are the key benefits of using a .dockerignore file:

1. **Improved build speed**: By excluding unnecessary files (e.g., log files, local development files, etc.), the size of the build context is reduced, speeding up the Docker build process, especially when dealing with large projects.
2. **Smaller image size**: Avoiding the inclusion of irrelevant files in the image ensures that the resulting Docker image is smaller and contains only the essential files needed for running the container.
3. **Security**: You can exclude sensitive files such as configuration files, secrets, or credentials that should not be included in the final Docker image or shared with others.
4. **Preventing unintended behavior**: Sometimes, including certain files (like temporary files or editor-specific files) can cause issues in the Docker container. The .dockerignore file prevents such files from being copied into the image.

The syntax of the .dockerignore file is similar to .gitignore, where you can define patterns to include or exclude files and directories.

***│ ├─ build\_docker.sh***

explain each part of your build\_docker.sh script. This script is designed to build a Docker image and push it to Amazon Elastic Container Registry (ECR), both public and private repositories. Here's a detailed explanation of each section and line:

### ****1. Script Overview****

The primary purpose of this script is to automate the process of:

1. **Building a Docker image** from the current directory.
2. **Tagging** the image appropriately.
3. **Logging into AWS ECR** (both public and private, based on the commands).
4. **Pushing** the Docker image to the specified ECR repository.

### ****2. Script Breakdown****

Script line by line:

bash

Copy code

AWS\_ACCOUNT\_ID=$1

AWS\_REGION='us-east-1'

IMAGE\_TAG='v1.0' # Use versioned tags

# ECR\_REPOSITORY='public.ecr.aws/v1a3q6c0/streamlit-temp-stack:latest'

ECR\_REPOSITORY='cdk-hnb659fds-container-assets-454674044397-us-east-1'

#ECR\_REPOSITORY='cdk-hnb659fds-container-assets-562860900886-us-east-1

aws ecr-public get-login-password --region us-east-1 | docker login --username AWS --password-stdin public.ecr.aws/v1a3q6c0

docker build . --tag $IMAGE\_TAG

docker tag $IMAGE\_TAG $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$ECR\_REPOSITORY:$IMAGE\_TAG

eval $(aws ecr get-login --no-include-email)

docker push $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$ECR\_REPOSITORY:$IMAGE\_TAG

#### **a. Setting Up Variables**

bash

Copy code

AWS\_ACCOUNT\_ID=$1

AWS\_REGION='us-east-1'

IMAGE\_TAG='v1.0' # Use versioned tags

* **AWS\_ACCOUNT\_ID=$1**:
  + **Purpose**: Assigns the first argument passed to the script to the variable AWS\_ACCOUNT\_ID.
  + **Usage**: This should be your AWS account number (e.g., 123456789012).
  + **Example**: If you run `./build\_d

│ ├─ demo-data

│ │ ├─ df\_interactions.csv

│ │ ├─ df\_item\_banking.csv

│ │ ├─ df\_item\_deduplicated.csv

│ │ ├─ df\_item\_deduplicated.csv.orig

│ │ ├─ df\_segment\_data.csv

│ │ ├─ df\_segment\_data.csv.orig

│ │ ├─ df\_segment\_data\_v2.csv

│ │ ├─ df\_segment\_data\_v3.csv

│ │ ├─ df\_segment\_data\_v4.csv

│ │ ├─ df\_users\_deduplicated.csv

│ │ └─ products.json

***│ ├─ lambda***

***│ │ ├─ bedrock\_content\_generation\_lambda***

│ ***│ │ ├─ bedrock\_content\_generation\_lambda.py***

This code is a Python script designed for an AWS Lambda function that interacts with AWS Bedrock to generate text using various models such as Amazon Titan, Anthropic Claude, and AI21's models. Here’s an explanation of the different sections:

**Imports and Setup**

python

Copy code

import json

import logging

import os

import sys

from datetime import datetime, timezone

import boto3

from botocore.config import Config

* **Imports**: The script imports several standard libraries (json, logging, os, sys, datetime) and AWS SDK libraries (boto3, botocore) to manage Bedrock API and logging.

**Logger Configuration**

python

Copy code

LOGGER = logging.Logger("Content-generation", level=logging.DEBUG)

HANDLER = logging.StreamHandler(sys.stdout)

HANDLER.setFormatter(logging.Formatter("%(levelname)s | %(name)s | %(message)s"))

LOGGER.addHandler(HANDLER)

* This section sets up a logger named "Content-generation" to log events to the standard output stream (sys.stdout) in a structured format (showing log level, name, and message).

**Global Variables**

python

Copy code

BEDROCK\_ROLE\_ARN = os.environ["BEDROCK\_ROLE\_ARN"]

BEDROCK\_CONFIG = Config(connect\_timeout=60, read\_timeout=60, retries={"max\_attempts": 10})

* **BEDROCK\_ROLE\_ARN**: This environment variable is used to determine whether cross-account access is required for Bedrock. If it’s present, the script assumes a cross-account role; otherwise, it uses the current account.
* **BEDROCK\_CONFIG**: This sets connection and read timeouts along with retry logic for the AWS SDK requests.

**Model Mapping**

python

Copy code

MODELS\_MAPPING = {

"Bedrock: Amazon Titan": "amazon.titan-tg1-large",

"Bedrock: Claude": "anthropic.claude-v1",

"Bedrock: Claude V2": "anthropic.claude-v2",

"Bedrock: Claude Instant": "anthropic.claude-instant-v1",

"Bedrock: J2 Grande Instruct": "ai21.j2-grande-instruct",

"Bedrock: J2 Jumbo Instruct": "ai21.j2-jumbo-instruct",

"Bedrock: Claude Haiku": "anthropic.claude-3-haiku-20240307-v1:0",

}

* This dictionary maps human-readable model names to their corresponding internal IDs in AWS Bedrock.

**create\_bedrock\_client Function**

This function creates an AWS Bedrock client, either using the same account or cross-account access via AWS Security Token Service (STS).

python

Copy code

def create\_bedrock\_client():

if BEDROCK\_ROLE\_ARN != "None":

LOGGER.info("Using cross-account bedrock client.")

# Use STS to assume the role and create the Bedrock client

...

else:

LOGGER.info("Using bedrock client from same account.")

# Create a Bedrock client without role assumption

...

* **Cross-account Access**: If BEDROCK\_ROLE\_ARN is set, the script assumes a cross-account role using sts.assume\_role, obtains temporary credentials, and uses them to create a Bedrock client.
* **Same-account Access**: If no cross-account role is needed, the script creates a Bedrock client directly.

The function returns the client and the token expiration time (if applicable).

**verify\_bedrock\_client Function**

python

Copy code

def verify\_bedrock\_client():

# Check if the token has expired

...

This function verifies whether the Bedrock client’s session token has expired. If the expiration time is near (less than 60 seconds), it will return False, signaling that the token needs to be refreshed.

**generate\_message Function**

python

Copy code

def generate\_message(bedrock\_runtime, model\_id, system\_prompt, messages, max\_tokens):

body = json.dumps({

"anthropic\_version": "bedrock-2023-05-31",

"max\_tokens": max\_tokens,

"system": system\_prompt,

"messages": messages,

})

return bedrock\_runtime.invoke\_model(body=body, modelId=model\_id)

This function constructs a payload for generating text using AWS Bedrock’s API. It passes a system\_prompt, messages, and token limit to the Bedrock model. The Bedrock client then invokes the specified model with these inputs.

**lambda\_handler Function**

This is the main entry point for the AWS Lambda function, which is executed when the Lambda is triggered.

python

Copy code

def lambda\_handler(event, context):

LOGGER.info("Starting execution of lambda\_handler()")

...

1. **Input Parsing**:
   * Extracts the query and model parameters from the event payload (JSON format).
   * Maps the human-readable model name to its internal ID using the MODELS\_MAPPING dictionary.
2. **Model Parameter Setup**:
   * Based on the model selected (Amazon, Anthropic, or AI21), it sets up specific parameters such as token count, stop sequences, and temperature.
3. **Verify Bedrock Client**:
   * Calls verify\_bedrock\_client() to ensure the session token is still valid. If expired, it refreshes the token.
4. **Model Invocation**:
   * Depending on the selected model (Amazon, Claude, or AI21), the appropriate input structure is sent to the Bedrock model via the invoke\_model method.
5. **Response Parsing**:
   * The function extracts and returns the model’s response, such as generated text, from the response body.

**Conclusion**

This script is designed to integrate with AWS Bedrock, allowing the Lambda function to invoke various AI models for text generation. It handles both cross-account access, if needed, and same-account access. The use of the boto3 SDK allows seamless interaction with AWS services, and the Bedrock client is refreshed if necessary to avoid token expiration issues.

***│ │ │ └─ model\_configs***

***│ │ │ ├─ ai21.j2-grande-instruct.json***

This JSON object defines the parameters for a request to generate a summary of a document using a text generation model. Here's what each key-value pair represents:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Description**: This is the prompt or command provided to the model, instructing it to generate a summary of the input document in a concise, one-paragraph format.
2. **EXAMPLES**:
   * **Value**: null
   * **Description**: This field is reserved for examples to guide the model, but in this case, no examples are provided (hence, it’s set to null).
3. **STOP\_WORDS**:
   * **Value**: []
   * **Description**: This is an empty list, meaning no stop words are defined. Stop words, if provided, would be words or phrases that the model should avoid using in the generated summary.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Description**: This value refers to **nucleus sampling** (also called **Top-P sampling**), where the model selects from the top 90% of possible token outputs based on their probability distribution. It controls the randomness of the generated text, with higher values allowing more diverse choices while maintaining coherence.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Description**: This defines the maximum length of the input document (in tokens or characters) that the model will process. In this case, it’s set to 10,000, indicating the model can handle long documents.

In summary, this configuration instructs a text generation model to summarize a document of up to 10,000 characters or tokens into one paragraph, using Top-P sampling for controlled randomness and without any specific stop words or example inputs.

***│ │ │ ├─ ai21.j2-jumbo-instruct.json***

This JSON object provides the configuration for a text generation task, specifically summarizing a document in one paragraph. Here's a detailed breakdown:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Purpose**: This is the directive given to the model, instructing it to summarize the input document into a single paragraph.
2. **EXAMPLES**:
   * **Value**: null
   * **Purpose**: This field is left empty (null), indicating that no specific examples are provided to guide the model. Typically, examples might be used to help the model understand how to format or approach the task, but here it's unnecessary.
3. **STOP\_WORDS**:
   * **Value**: []
   * **Purpose**: This is an empty list, meaning there are no words the model is explicitly told to avoid. If specific words were to be excluded from the output, they would be listed here.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Purpose**: This controls **nucleus sampling**, a method of text generation. By setting TOP\_P to 0.9, the model is instructed to choose from the top 90% of predicted words, ensuring the output remains both coherent and somewhat varied. This helps balance randomness and accuracy in the generated summary.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Purpose**: This sets the maximum length (in tokens or characters) of the input document that the model will consider. In this case, it can handle documents up to 10,000 tokens or characters.

**Summary:**

This configuration is designed to have a model summarize a document of up to 10,000 characters/tokens into a single paragraph. It uses Top-P sampling (0.9) to guide the model's word choices, ensuring a balance of coherence and variety, without any specific stop words or example inputs to guide the process.

***│ │ │ ├─ amazon.titan-tg1-large.json***

This JSON object specifies a configuration for generating a summary of a document. Here's what each field represents:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Explanation**: This is the command given to the model. It instructs the model to summarize the document provided in a concise, single-paragraph format.
2. **EXAMPLES**:
   * **Value**: null
   * **Explanation**: No examples are provided to guide the model on how to perform the task. This field could have contained examples of input-output pairs, but it's left empty (null) here.
3. **STOP\_WORDS**:
   * **Value**: []
   * **Explanation**: This is an empty list, meaning that no specific words are restricted or omitted from the generated summary. If any specific words were to be avoided in the output, they would be listed here.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Explanation**: This refers to the **nucleus sampling** technique, where the model considers the top 90% of probability-weighted word choices when generating the output. This parameter controls the randomness, ensuring the summary is coherent yet slightly varied.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Explanation**: This defines the maximum length of the input document that can be summarized. The document can be up to 10,000 characters or tokens in length.

**Summary:**

This configuration sets the parameters for summarizing a document into one paragraph, allowing for up to 10,000 characters or tokens, with some flexibility in word choice due to the Top-P sampling value of 0.9. No specific stop words or examples are defined for the task.

***│ │ │ ├─ anthropic.claude-3-haiku-20240307-v1:0.json***

This JSON object defines the parameters for a text generation task, specifically summarizing a document. Here's what each part represents:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Purpose**: This is the directive given to the model, instructing it to generate a summary of the input document in a single paragraph format.
2. **EXAMPLES**:
   * **Value**: null
   * **Purpose**: No examples are provided to guide the model. Typically, examples could show input-output pairs to help the model understand the format of the task, but here, it's unnecessary.
3. **STOP\_WORDS**:
   * **Value**: ["\n\nHuman:"]
   * **Purpose**: This is a list of words or phrases the model should avoid in its output. In this case, the model is instructed to exclude the phrase "\n\nHuman:" from the summary. This is often used in dialogue generation settings, and its exclusion here ensures that the summary is focused on the content rather than replicating conversational cues.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Purpose**: This controls **nucleus sampling**, where the model considers the top 90% of likely word choices. It ensures the summary is both coherent and diverse by sampling from the most probable words while allowing for some randomness.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Purpose**: This sets the maximum length (in tokens or characters) of the input document that the model will process. It can handle documents of up to 10,000 tokens or characters.

**Summary:**

This configuration is designed to have the model summarize a document up to 10,000 characters or tokens in one paragraph. It instructs the model to exclude the phrase "\n\nHuman:" from the output, ensures controlled randomness through Top-P sampling (0.9), and doesn't rely on any specific examples to guide the model.

***│ │ │ ├─ anthropic.claude-instant-v1.json***

This JSON object defines a configuration for summarizing a document using a text generation model. Here's what each part of the object represents:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Explanation**: This is the command provided to the model. It instructs the model to generate a concise summary of the input document, specifically limited to one paragraph.
2. **EXAMPLES**:
   * **Value**: null
   * **Explanation**: No examples are provided to guide the model's behavior. This field is used to give specific input-output examples to the model, but it is not needed here, so it's set to null.
3. **STOP\_WORDS**:
   * **Value**: ["\n\nHuman:"]
   * **Explanation**: This is a list of words or phrases that the model should avoid using in the generated summary. In this case, the phrase "\n\nHuman:" is excluded from the output. This is likely used in a scenario where the model generates conversational-style output, and you don't want certain dialogue markers in the summary.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Explanation**: This controls **nucleus sampling**, where the model selects from the top 90% of probable word choices. The value of 0.9 allows for diversity in word selection, ensuring some randomness while maintaining coherence in the generated text.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Explanation**: This sets the maximum length (in tokens or characters) for the input document. The model can process documents up to 10,000 tokens or characters in length for summarization.

**Summary:**

This configuration directs the model to summarize a document (up to 10,000 characters) into one paragraph, excluding the phrase "\n\nHuman:" from the generated output. The model will use Top-P sampling (set to 0.9) to balance coherence and variety, and no specific examples are given to guide the summarization.

***│ │ │ ├─ anthropic.claude-v1.json***

This JSON object specifies the parameters for a text summarization task. Here's what each field means:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Explanation**: This is the directive for the model, instructing it to produce a one-paragraph summary of the provided document.
2. **EXAMPLES**:
   * **Value**: null
   * **Explanation**: No examples are provided to guide the model. If examples were given, they would help the model understand the format or style of the expected summary, but in this case, it's not needed.
3. **STOP\_WORDS**:
   * **Value**: ["\n\nHuman:"]
   * **Explanation**: This list contains words or phrases that the model should avoid using in the summary. In this case, the phrase "\n\nHuman:" is excluded, which is likely used in conversational settings where the model generates dialogue and you want to remove specific markers.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Explanation**: This sets the **nucleus sampling** value, which determines how the model selects the next word. By setting TOP\_P to 0.9, the model will consider the top 90% of predicted words, providing a balance between coherence and variety in the summary. It prevents the model from selecting less probable or nonsensical words while maintaining some diversity.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Explanation**: This specifies the maximum length of the input document that the model can process for summarization. The model will handle documents up to 10,000 tokens or characters in length.

**Summary:**

This configuration is designed for a model to summarize a document up to 10,000 characters or tokens in length into a single paragraph. It uses Top-P sampling to maintain coherence while allowing some variation in the output and excludes the specific phrase "\n\nHuman:" from the summary. No specific examples are provided to guide the model’s behavior.

***│ │ │ └─ anthropic.claude-v2.json***

This JSON object contains configuration settings for generating a summary of a document. Here's what each field represents:

1. **INSTRUCTION**:
   * **Value**: "Summarize the following document in one paragraph."
   * **Explanation**: This provides the directive to the model, instructing it to summarize the given document into a single paragraph.
2. **EXAMPLES**:
   * **Value**: null
   * **Explanation**: No specific examples are provided. If examples were given, they would show how input should be summarized to guide the model, but it's not used here.
3. **STOP\_WORDS**:
   * **Value**: ["\n\nHuman:"]
   * **Explanation**: This list specifies words or phrases that the model should avoid including in the generated summary. In this case, "\n\nHuman:" is to be excluded, which suggests that this phrase is likely part of the input structure (perhaps from a dialogue-based context) but should not be in the output.
4. **TOP\_P**:
   * **Value**: 0.9
   * **Explanation**: This controls **nucleus sampling**, a method where the model considers the top 90% of likely words when generating text. It ensures that the output is coherent while allowing some diversity by limiting randomness to the most probable words.
5. **INPUT\_DOCUMENT\_LENGTH**:
   * **Value**: 10000
   * **Explanation**: This defines the maximum number of tokens or characters the input document can have. The model will process up to 10,000 tokens or characters for the summarization task.

**Summary:**

This configuration instructs a model to summarize a document (up to 10,000 tokens or characters) in one paragraph. It uses Top-P sampling with a value of 0.9 for controlled randomness and coherence, and excludes the specific phrase "\n\nHuman:" from the output, which may come from a dialogue context. No examples are provided to guide the summarization process further.

Top of Form

Bottom of Form

***│ │ ├─ genai\_personalize\_batch\_segment\_job***

***│ │ │ └─ personalize\_batch\_segment\_job.py***

This Python code is an AWS Lambda function designed to handle both **POST** and **GET** requests for Amazon Personalize batch segment jobs. It interacts with AWS services such as **S3** and **Personalize** to upload item IDs, initiate batch jobs, and retrieve job details. Here's a breakdown of the code:

### ****LIBRARIES & LOGGER****

python

Copy code

import json

import logging

import os

import sys

import tempfile

import uuid

from datetime import datetime

import boto3

from botocore.exceptions import ClientError

* **Imports**:
  + Common Python libraries for handling JSON, logging, environment variables (os), temporary file creation, and UUID generation.
  + **boto3**: The AWS SDK for Python, used to interact with AWS services (S3, Personalize, STS).
  + **ClientError**: Used to handle AWS API exceptions.

python

Copy code

LOGGER = logging.Logger("Content-generation", level=logging.DEBUG)

HANDLER = logging.StreamHandler(sys.stdout)

HANDLER.setFormatter(logging.Formatter("%(levelname)s | %(name)s | %(message)s"))

LOGGER.addHandler(HANDLER)

* **Logger Setup**: The logger is initialized to output messages to the console (stdout) in a structured format with log levels.

### ****HELPER: Environment Variables****

python

Copy code

bucket\_name = os.environ["BUCKET\_NAME"]

role\_arn = os.environ["PERSONALIZE\_ROLE\_ARN"]

solution\_version\_arn = os.environ["SOLUTION\_VERSION\_ARN"]

* **Environment Variables**:
  + **BUCKET\_NAME**: The S3 bucket where input/output files are stored.
  + **PERSONALIZE\_ROLE\_ARN**: The Amazon Resource Name (ARN) of the role that allows the Lambda function to interact with Personalize.
  + **SOLUTION\_VERSION\_ARN**: The ARN of the solution version in Personalize used for the batch segment job.

### ****HANDLER:**** lambda\_handler ****Function****

This is the main function that AWS Lambda invokes. It handles both POST and GET requests and interacts with Amazon Personalize and S3.

#### **Handling POST Requests**

The POST method creates a batch segment job in Amazon Personalize using a list of item IDs.

python

Copy code

if http\_method == "POST":

# Parse the event body and extract item IDs

item\_ids = event["item-ids"]

# Generate a unique job name

job\_name = str(uuid.uuid4())

# Split the item IDs, create a JSON string, and upload to S3

...

1. **Event Parsing**: The function extracts the item-ids from the POST request body.
2. **Job Name**: It generates a unique job name using uuid.
3. **Temporary File Creation**: A temporary file is created with the item IDs, formatted as JSON objects.
4. **S3 Upload**: The file is uploaded to an S3 bucket under a specific key (personalize-input/{job\_name}.json).
5. **Batch Segment Job**: The job is created in Amazon Personalize using the uploaded file as input, and the result is returned in the response.

#### **Handling GET Requests**

The GET method retrieves the details of a specific batch segment job using the job ARN.

python

Copy code

elif http\_method == "GET":

# Extract job ARN from the request body

job\_arn = event["job-arn"]

...

1. **Event Parsing**: The function expects the job-arn to be present in the GET request body.
2. **Describe Job**: It calls the describe\_batch\_segment\_job method to get the job details.
3. **Response**: The job details are returned, including status and metadata.

#### **Handling Unsupported Methods**

python

Copy code

else:

# Unsupported HTTP method

return {"statusCode": 400, "body": "Unsupported HTTP method", "headers": {"Content-Type": "application/json"}}

* If the HTTP method is neither POST nor GET, a 400 error is returned, indicating an unsupported method.

### ****Helper Functions****

#### datetime\_handler **Function**

python

Copy code

def datetime\_handler(x):

"""Convert datetime objects to ISO format string."""

if isinstance(x, datetime):

return x.isoformat()

raise TypeError("Unknown type")

* This function is used in json.dumps() to handle datetime objects by converting them to an ISO format string (a common requirement for serializing datetime values).

### ****AWS Services Integration****

1. **S3**:
   * The function uploads item IDs to an S3 bucket and retrieves input/output data from S3.
2. **Amazon Personalize**:
   * **POST**: The function creates a **batch segment job** in Personalize using the input from the uploaded file.
   * **GET**: The function fetches the details of a previously created batch segment job using the job ARN.
3. **Error Handling**:
   * The function catches AWS ClientError exceptions to handle issues like permissions or invalid input and returns appropriate error responses.

### ****Summary****:

This AWS Lambda function provides a mechanism to:

* **Create batch segment jobs** in Amazon Personalize based on a list of item IDs (via a POST request).
* **Retrieve details** of existing segment jobs (via a GET request).
* It handles interaction with AWS services like S3 (for storing inputs/outputs) and Personalize (for running recommendation jobs), and includes proper error handling.

***│ │ ├─ genai\_personalize\_batch\_segment\_jobs***

***│ │ │ └─ personalize\_batch\_segment\_jobs.py***

This Python code is an AWS Lambda function that handles GET requests to list all Amazon Personalize batch segment jobs in a specified region. The function interacts with AWS Personalize via the boto3 client and returns the list of jobs in a structured JSON format. Here's a breakdown of the key parts:

### ****Imports and Setup****

python

Copy code

import datetime

import json

import logging

import sys

import boto3

from botocore.exceptions import ClientError

* **boto3**: AWS SDK for Python, used to interact with AWS services (in this case, Amazon Personalize).
* **logging**: Used to log important events and errors.
* **json**: Used to handle JSON encoding and decoding, such as formatting the response body.
* **datetime**: Used for handling datetime objects in the response.

### ****Logging Configuration****

python

Copy code

LOGGER = logging.Logger("Content-generation", level=logging.DEBUG)

HANDLER = logging.StreamHandler(sys.stdout)

HANDLER.setFormatter(logging.Formatter("%(levelname)s | %(name)s | %(message)s"))

LOGGER.addHandler(HANDLER)

* **Logger Setup**: A logger is initialized to output log messages to standard output (sys.stdout) in a readable format. It will log all messages at the DEBUG level or higher.

### ****Lambda Handler****

python

Copy code

def lambda\_handler(event, context):

"""Handle incoming requests for Personalize batch segment jobs.

This handler responds to GET requests and returns a list of all batch segment jobs in the specified region.

"""

# Get the HTTP method from the event object

http\_method = event["requestContext"]["http"]["method"]

# Create personalize client

personalize = boto3.client(service\_name="personalize")

# Check if the request is a GET request

if http\_method == "GET":

try:

# Call the list\_batch\_segment\_jobs method

response = personalize.list\_batch\_segment\_jobs()

# Return the list of batch segment jobs as a JSON response

return {

"statusCode": 200,

"body": json.dumps(response, default=datetime\_handler),

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

except ClientError:

# Handle any errors that occur

LOGGER.exception("An error occurred while fetching the batch segment jobs")

return {

"statusCode": 500,

"body": "An error occurred while fetching the batch segment jobs",

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

else:

# Return an error response for unsupported HTTP methods

return {

"statusCode": 400,

"body": "Unsupported HTTP method",

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

#### **Key Steps**:

1. **HTTP Method Check**:
   * The function checks the http\_method from the event object to determine if the request is a GET request. The function only supports GET requests.
2. **Amazon Personalize Client**:
   * **personalize = boto3.client(service\_name="personalize")** creates a client for Amazon Personalize. This client will be used to interact with the Personalize API.
3. **Handling GET Requests**:
   * The function calls personalize.list\_batch\_segment\_jobs() to retrieve a list of all batch segment jobs in Amazon Personalize.
   * If the call succeeds, the list of jobs is returned as a JSON response with a 200 status code.
   * If there is an error (caught as ClientError), the function logs the error using LOGGER.exception and returns a 500 response with an error message.
4. **Unsupported Methods**:
   * If any method other than GET is used, the function returns a 400 response indicating that the HTTP method is unsupported.

### ****Datetime Handler****

python

Copy code

def datetime\_handler(x):

"""Convert datetime objects to ISO format string."""

if isinstance(x, datetime.datetime):

return x.isoformat()

raise TypeError("Unknown type")

* **Purpose**: This function converts datetime objects to ISO format strings, which is a common requirement when serializing data for JSON.
  + For example, 2024-10-12T15:23:01Z is an ISO format date.
* **Used in json.dumps()**: When serializing the Personalize job response, any datetime objects (like creation times) will be passed through this function to ensure they are properly formatted.

### ****Error Handling****

* **Try/Except Block**: The function uses a try/except block to handle any errors that may occur when interacting with AWS services (like connectivity or permissions issues). If an error occurs, a 500 response is returned, and the error details are logged using LOGGER.exception.

### ****Summary****

This AWS Lambda function:

* Handles GET requests to retrieve a list of Amazon Personalize batch segment jobs.
* Interacts with AWS Personalize using boto3.
* Handles errors gracefully, logging issues and returning appropriate error messages.
* Converts datetime objects to ISO format for JSON serialization.

### ****Response Format****

For successful requests (GET):

* **Status Code**: 200
* **Body**: JSON response with the list of batch segment jobs.

For errors:

* **Status Code**: 500 (for errors with AWS) or 400 (for unsupported HTTP methods).
* **Body**: A string message indicating the type of error.

### Example Use Case

This Lambda function could be part of an API that allows users to fetch the status of batch segment jobs they have initiated in Amazon Personalize. It could be connected to an API Gateway to expose a REST endpoint for querying these jobs.

Top of Form

exportselect

Bottom of Form

ChatGPT can make mistakes. Check important info.

***│ │ ├─ genai\_pinpoint\_job***

***│ │ │ └─ pinpoint\_job.py***

This AWS Lambda function handles GET and POST requests to interact with Amazon Pinpoint's export jobs. It uses boto3 to communicate with the Pinpoint API, enabling the creation and retrieval of export jobs that export segment data to an S3 bucket.

Here’s a breakdown of the code:

**LIBRARIES & LOGGER**

python

Copy code

import json

import logging

import os

import sys

import boto3

from botocore.exceptions import ClientError

CONTENT\_TYPE\_JSON = "application/json"

LOGGER = logging.Logger("Content-generation", level=logging.DEBUG)

HANDLER = logging.StreamHandler(sys.stdout)

HANDLER.setFormatter(logging.Formatter("%(levelname)s | %(name)s | %(message)s"))

LOGGER.addHandler(HANDLER)

* **Imports**: The necessary libraries are imported. boto3 is used to interact with AWS Pinpoint, and logging is configured to capture and display logs for debugging purposes.
* **Logger**: The logger is configured to output debug-level logs to the console in a readable format.

**HELPER VARIABLES**

python

Copy code

PINPOINT\_PROJECT\_ID = os.environ["PINPOINT\_PROJECT\_ID"]

PINPOINT\_EXPORT\_ROLE\_ARN = os.environ["PINPOINT\_EXPORT\_ROLE\_ARN"]

S3\_BUCKET\_NAME = os.environ["BUCKET\_NAME"]

* These environment variables store essential information:
  + **PINPOINT\_PROJECT\_ID**: The ID of the Amazon Pinpoint project (application) associated with export jobs.
  + **PINPOINT\_EXPORT\_ROLE\_ARN**: The ARN of the role that gives permission for exporting data to an S3 bucket.
  + **S3\_BUCKET\_NAME**: The name of the S3 bucket where the exported segment data will be stored.

**HANDLER: lambda\_handler**

The function handles GET and POST requests based on the HTTP method in the incoming event. It interacts with Amazon Pinpoint to either fetch an existing export job or create a new one.

python

Copy code

def lambda\_handler(event, context):

"""Handle GET and POST requests for Pinpoint export jobs."""

# Get the HTTP method from the event object

http\_method = event["requestContext"]["http"]["method"]

# Handle GET requests for fetching export jobs

if http\_method == "GET":

...

# Handle POST requests for creating new export jobs

elif http\_method == "POST":

...

else:

# Handle unsupported HTTP methods

return {

"statusCode": 400,

"body": "Unsupported HTTP method",

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

* **GET requests**: Used to fetch details of an existing export job from Pinpoint.
* **POST requests**: Used to create a new export job for a given segment in Pinpoint.
* **Unsupported methods**: If a method other than GET or POST is used, the function returns a 400 status code with an error message.

**Handling GET Requests**

python

Copy code

if http\_method == "GET":

pinpoint\_project\_id = os.environ["PINPOINT\_PROJECT\_ID"]

event = json.loads(event["body"])

export\_job\_id = event["job-id"]

client = boto3.client("pinpoint")

try:

response = client.get\_export\_job(ApplicationId=pinpoint\_project\_id, JobId=export\_job\_id)

export\_job\_response = response["ExportJobResponse"]

return {

"statusCode": 200,

"body": json.dumps(export\_job\_response),

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

except ClientError as e:

return {

"statusCode": 500,

"body": "An error occurred while fetching the export job",

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

* **Pinpoint Client**: The boto3 Pinpoint client is used to fetch an export job by calling get\_export\_job.
* **Event Parsing**: The job ID is extracted from the event body to identify which export job to retrieve.
* **Error Handling**: If the export job retrieval fails (for example, due to permissions or an invalid job ID), a 500 error response is returned.

**Handling POST Requests**

python

Copy code

elif http\_method == "POST":

client = boto3.client("pinpoint")

event = json.loads(event["body"])

segment\_id = event["segment-id"]

try:

response = client.create\_export\_job(

ApplicationId=PINPOINT\_PROJECT\_ID,

ExportJobRequest={

"RoleArn": PINPOINT\_EXPORT\_ROLE\_ARN,

"S3UrlPrefix": f"s3://{S3\_BUCKET\_NAME}/exported-segments/{segment\_id}/",

"SegmentId": segment\_id,

},

)

export\_job\_response = response["ExportJobResponse"]

return {

"statusCode": 200,

"body": json.dumps(export\_job\_response),

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

except ClientError as e:

return {

"statusCode": 500,

"body": "An error occurred while creating the export job",

"headers": {"Content-Type": CONTENT\_TYPE\_JSON},

}

* **Segment ID**: The POST request body contains the segment ID that the export job will be created for.
* **Pinpoint Client**: The create\_export\_job method is called to create an export job for the provided segment. The exported data will be saved in the S3 bucket specified in the S3UrlPrefix.
* **Error Handling**: If any error occurs (e.g., missing permissions or invalid segment ID), the function returns a 500 status code with an error message.

**Summary**

This Lambda function allows interaction with Amazon Pinpoint to manage export jobs:

1. **GET Requests**:
   * Fetch an existing export job by its job ID.
   * The response includes details such as the job's status, completion time, and output location in S3.
2. **POST Requests**:
   * Create a new export job for a specified segment.
   * The segment data is exported to an S3 bucket in a specific folder under exported-segments/{segment\_id}.
3. **Error Handling**:
   * Both GET and POST requests handle potential issues using AWS's ClientError exceptions, returning 500 status codes when errors occur.
4. **Environment Variables**:
   * The function uses environment variables to store essential AWS configuration details such as the Pinpoint project ID, the export role ARN, and the S3 bucket name.

This Lambda function can be deployed in an AWS environment and connected to API Gateway to provide an API that manages Pinpoint export jobs.

***│ │ ├─ genai\_pinpoint\_message***

***│ │ │ └─ pinpoint\_message.py***

***│ │ ├─ genai\_pinpoint\_message\_custom***

***│ │ │ └─ pinpoint\_message\_custom.py***

***│ │ ├─ genai\_pinpoint\_message\_email***

***│ │ │ └─ pinpoint\_message\_email.py***

***│ │ ├─ genai\_pinpoint\_message\_sms***

***│ │ │ └─ pinpoint\_message\_sms.py***

***│ │ ├─ genai\_pinpoint\_segment***

***│ │ │ └─ pinpoint\_segment.py***

***│ │ └─ genai\_s3***

***│ │ └─ s3\_fetch.py***

***│ ├─ layers***

***│ │ ├─ factory\_module***

***│ │ │ └─ python***

***│ │ │ ├─ c2m\_add\_credit.py***

***│ │ │ ├─ c2m\_check\_job\_status.py***

***│ │ │ ├─ c2m\_check\_tracking.py***

***│ │ │ ├─ c2m\_create\_job.py***

***│ │ │ ├─ c2m\_delete\_address\_lists.py***

***│ │ │ ├─ c2m\_submit\_job.py***

***│ │ │ ├─ c2m\_upload\_address\_list.py***

***│ │ │ ├─ c2m\_upload\_document.py***

***│ │ │ ├─ c2m\_upload\_document.py.session***

***│ │ │ └─ factory\_module.py***

***│ │ ├─ langchain***

***│ │ │ ├─ packages.txt***

***│ │ │ └─ requirements.txt***

***│ │ └─ utilities***

***│ │ └─ python***

***│ │ └─ aws\_helper.py***

***│ ├─ shared\_module***

***│ │ └─ python***

***│ │ ├─ \_\_init\_\_.py***

***│ │ ├─ channel\_states.py***

***│ │ └─ print\_response.py***

***│ └─ streamlit***

***│ ├─ .env***

***│ ├─ .streamlit***

***│ │ ├─ config.toml***

***│ │ └─ pages.toml***

***│ ├─ poetry.lock***

***│ ├─ pyproject.toml***

***│ ├─ src***

***│ │ ├─ Home.py***

***│ │ ├─ app\_pages***

***│ │ │ ├─ 00\_Prompt\_Smith.py***

***│ │ │ ├─ 01\_Segment\_Architect.py***

***│ │ │ ├─ 02\_Content\_Personalizer.py***

***│ │ │ └─ 03\_Message\_Dispatcher.py***

***│ │ ├─ components***

***│ │ │ ├─ \_\_init\_\_.py***

***│ │ │ ├─ authenticate.py***

***│ │ │ ├─ genai\_api.py***

***│ │ │ ├─ model\_specs.json***

***│ │ │ ├─ personalize\_api.py***

***│ │ │ ├─ pinpoint\_api.py***

***│ │ │ ├─ utils.py***

***│ │ │ └─ utils\_models.py***

***│ │ └─ images***

***│ │ ├─ background.jpg***

***│ │ ├─ banner2.webp***

***│ │ ├─ click2mail-header-logo.webp***

***│ │ └─ render.png***

***│ └─ temp***

***│ ├─ qrcode\_03\_10\_2024\_10\_40\_27.png***

***│ ├─ qrcode\_09\_10\_2024\_12\_14\_56.png***

***│ └─ qrcode\_30\_09\_2024\_07\_50\_58.png***

***├─ cdk.context.json***

***├─ cdk.json***

***├─ config.yml***

***├─ images***

***│ ├─ architecture.png***

***│ ├─ cognito\_create\_user.png***

***│ ├─ cognito\_mfa\_setup.png***

***│ ├─ cognito\_user\_pool.png***

***│ ├─ filter\_product.png***

***│ ├─ lambda\_solution\_ARN.png***

***│ ├─ marketing\_content\_distribution.png***

***│ ├─ marketing\_content\_generator.png***

***│ ├─ personalize\_datasetgroup.png***

***│ ├─ personalize\_import\_done.png***

***│ ├─ personalize\_import\_interaction\_data.png***

***│ ├─ personalize\_solution\_ARN.png***

***│ ├─ personalize\_solution\_creation.png***

***│ ├─ personalize\_solutions.png***

***│ ├─ prompt\_engineering.png***

***│ ├─ request\_model\_access.png***

***│ ├─ user\_segment\_batch.png***

***│ └─ user\_segment\_upload.png***

***├─ infra***

***│ ├─ \_\_init\_\_.py***

***│ ├─ cdsgenai\_stack.py***

***│ ├─ constructs***

***│ │ ├─ \_\_init\_\_.py***

***│ │ ├─ cdsai\_api.py***

***│ │ ├─ cdsai\_endpoint.py***

***│ │ ├─ cdsai\_personalize.py***

***│ │ ├─ cdsai\_pinpoint.py***

***│ │ └─ llm\_endpoints***

***│ │ ├─ \_\_init\_\_.py***

***│ │ ├─ config\_factory***

***│ │ │ ├─ \_\_init\_\_.py***

***│ │ │ ├─ base.py***

***│ │ │ ├─ jumpstart.py***

***│ │ │ ├─ marketplace.py***

***│ │ │ └─ tgi.py***

***│ │ ├─ constants.py***

***│ │ ├─ iam.py***

***│ │ └─ utils.py***

***│ └─ stacks***

***│ ├─ \_\_init\_\_.py***

***│ └─ streamlit.py***

***├─ poetry.lock***

***├─ pyproject.toml***

***├─ requirements.txt***

***├─ resize.sh***

***├─ samples***

***│ └─ roles***

***│ └─ cross\_account\_bedrock***

***│ ├─ policy\_document.json***

***│ └─ trust\_relationships.json***

***├─ scripts***

***│ ├─ add\_cognito\_user.bsh***

***│ ├─ code\_analyzer.py***

***│ ├─ format.sh***

***│ ├─ format\_precommit\_output.py***

***│ ├─ format\_precommit\_output.sh***

***│ ├─ getJWT.bsh***

***│ ├─ get\_digest.bsh***

***│ ├─ get\_digest\_and\_build.bsh***

***│ ├─ replaceSingleToDoubleQuotesAllFiles.bsh***

***│ ├─ segment-request.json***

***│ └─ update\_password.bsh***

***├─ setup\_and\_build\_commands.txt***

***├─ tests***

***│ ├─ \_\_init\_\_.py***

***│ └─ unit***

***│ └─ \_\_init\_\_.py***

***├─ tox.ini***

***```***