Amazon SageMaker AI

Problem Statement

Deploy a foundation model on Amazon SageMaker AI to build a robust AI solution that uses different prompting techniques.

Summary

This solution uses Amazon SageMaker AI to deploy and manage sophisticated AI models.
You will

- Set up the necessary configurations for model deployment in Amazon SageMaker Studio.
- Deploy a model endpoint with the SageMaker Python SDK.
- Use a SageMaker Studio notebook to experiment with different prompt techniques.

Concept

Amazon SageMaker Studio is a web-based, integrated development environment (IDE) for machine learning that helps you build, train, debug, deploy, and monitor your ML models.

Amazon SageMaker JumpStart is a comprehensive repository within SageMaker AI that provides prebuilt solutions, algorithms, and models to accelerate ML project development.

SageMaker JumpStart offers a variety of ready-to-deploy ML solutions across different domains, such as computer vision, natural language processing, and time-series forecasting.

SageMaker JumpStart includes prebuilt models, notebook examples, deployment templates, and algorithms.

A JupyterLab space is a private or shared space within SageMaker Studio that manages the storage and compute resources needed to run the JupyterLab application.

A data scientist can use JupyterLab to explore data and tune models. In addition, a machine learning operations (MLOps) engineer can use Code Editor with the pipelines tool in SageMaker Studio to deploy and monitor models in production.

The new web-based UI in SageMaker Studio is faster and provides access to all SageMaker AI resources, including jobs and endpoints, in one interface. ML practitioners can also choose their preferred IDE to accelerate ML development.

SageMaker Studio notebooks are collaborative notebooks that you can launch quickly because you don't need to set up compute instances and file storage beforehand. A set of instance types, known as fast launch types, are designed to launch in under two minutes. SageMaker Studio notebooks provide persistent storage, so you can view and share notebooks even if the instances that the notebooks run on are shut down.

Generative AI is a type of AI that can create new content and ideas, including conversations, stories, images, videos, and music. Generative AI is powered by large models, commonly referred to as foundation models (FMs), that are pre-trained on vast amounts of data.

Using SageMaker AI, you can deploy your ML models to make predictions, also known as inference. SageMaker AI provides a broad selection of ML infrastructure and model deployment options to help meet all your ML inference needs.

The SageMaker Python SDK is a powerful tool that streamlines the process of building, training, and deploying ML models by using SageMaker AI.

Parameters in API calls to large language models (LLMs) are configurations that control various aspects of a model’s behavior and generated output. Using these parameters, developers can fine-tune an LLM to meet specific requirements.

Prompt engineering is the process of designing and refining input prompts to elicit desired responses from ML models, particularly LLMs. Effective prompt engineering can significantly enhance the quality and relevance of model outputs.

Zero-shot prompting is a technique in generative AI where the model generates outputs based on instructions or queries it has never encountered during its training phase. This approach relies on the model's inherent understanding and generalization capabilities, so it can perform tasks or answer questions without prior specific training on those exact inputs.

Chain of thought (CoT) prompting is a technique that encourages LLMs to generate more logical and coherent responses by explaining their reasoning process step-by-step.

In advanced reasoning models such as DeepSeek, Llama 3.2 and Claude 3.7 Sonnet, the CoT technique is incorporated to enhance a model’s ability to perform multi-step reasoning and provide more accurate and justifiable answers.

One-shot prompting is a technique used in generative AI where the model is provided with a single example input-output pair to guide its understanding and generation of subsequent outputs.

This technique uses the model's ability to learn from minimal context, so it can infer patterns and apply them to new, similar tasks.

Few-shot prompting is a generative AI technique where the model is provided with a small number of example input-output pairs to guide its understanding and generation of subsequent outputs.

This method sits between one-shot and traditional large-scale training approaches, offering a balance by requiring only a minimal set of examples to effectively infer patterns and apply them to new tasks.

Prompt engineering techniques—such as zero-shot, one-shot, and few-shot approaches—offer different ways to guide AI models to produce desired outputs.

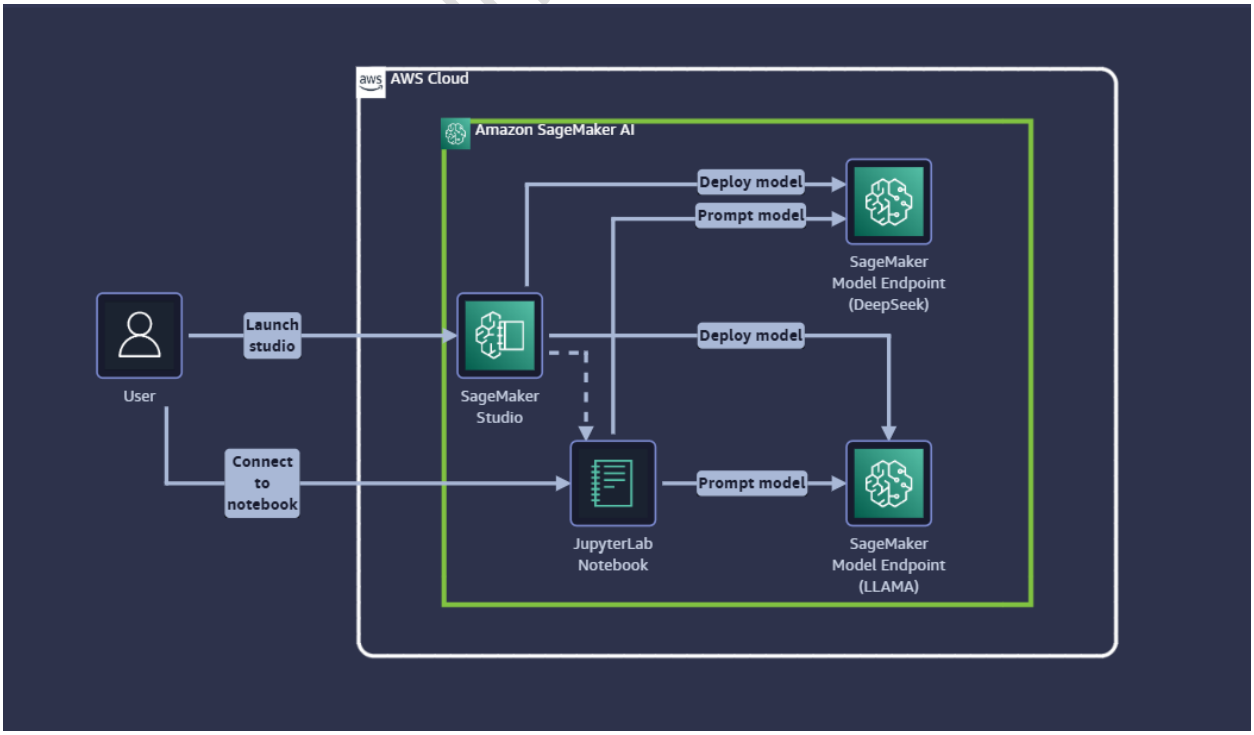
Clear instructions are crucial across all these techniques because they reduce ambiguity and help the model understand exactly what's expected.

Temperature in prompt engineering controls the randomness or predictability of a model's responses.

A lower temperature (closer to 0) makes responses more deterministic and focused, which is better for factual or technical tasks where accuracy is crucial.

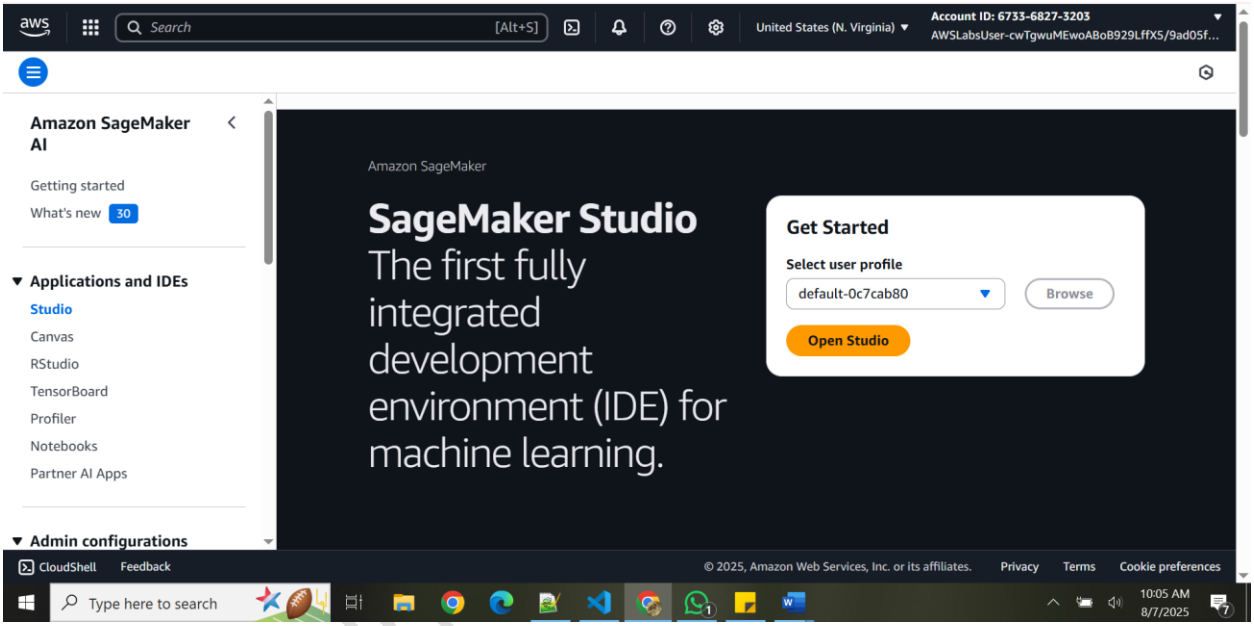
A higher temperature (closer to 1 or above) introduces more randomness, leading to more creative, diverse, and surprising outputs, which works well for creative writing or brainstorming.

Architecture

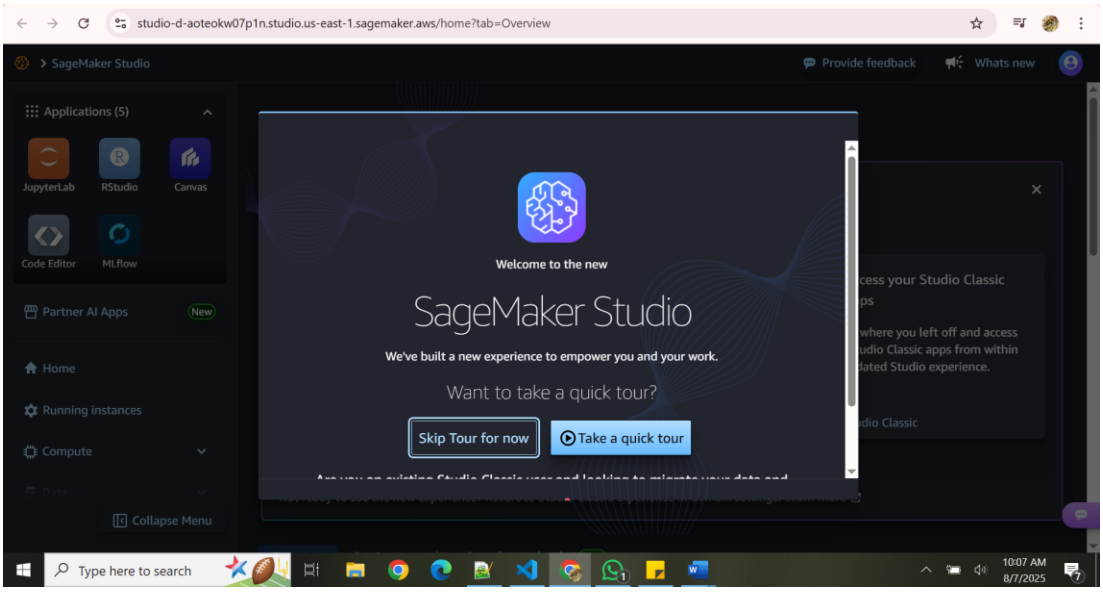


Implementation Steps

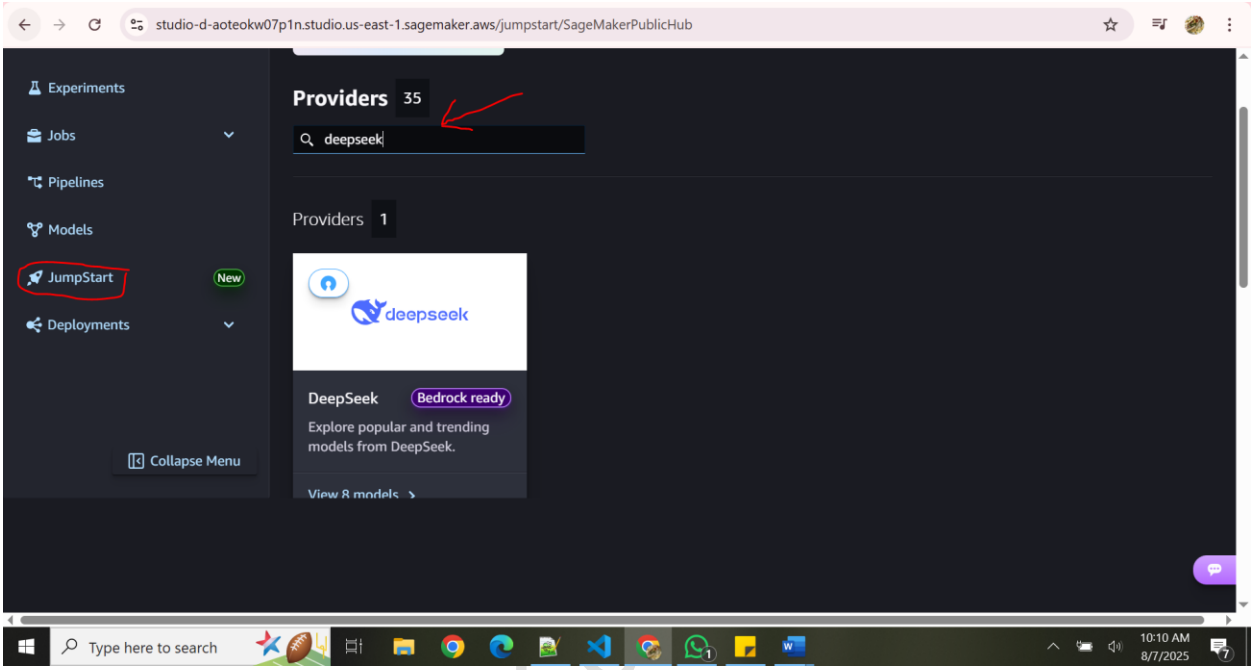
1. In the top navigation bar search box, type: sagemaker ai
2. In the search results, under Services, click Amazon SageMaker AI.
3. On the top navigation bar, review the Region selector to confirm that the Region is set to United States (N. Virginia).
4. In the left navigation pane, under Applications and IDEs, click Studio. - You might need to click the menu icon (three lines) in the left side panel to expand the navigation pane.



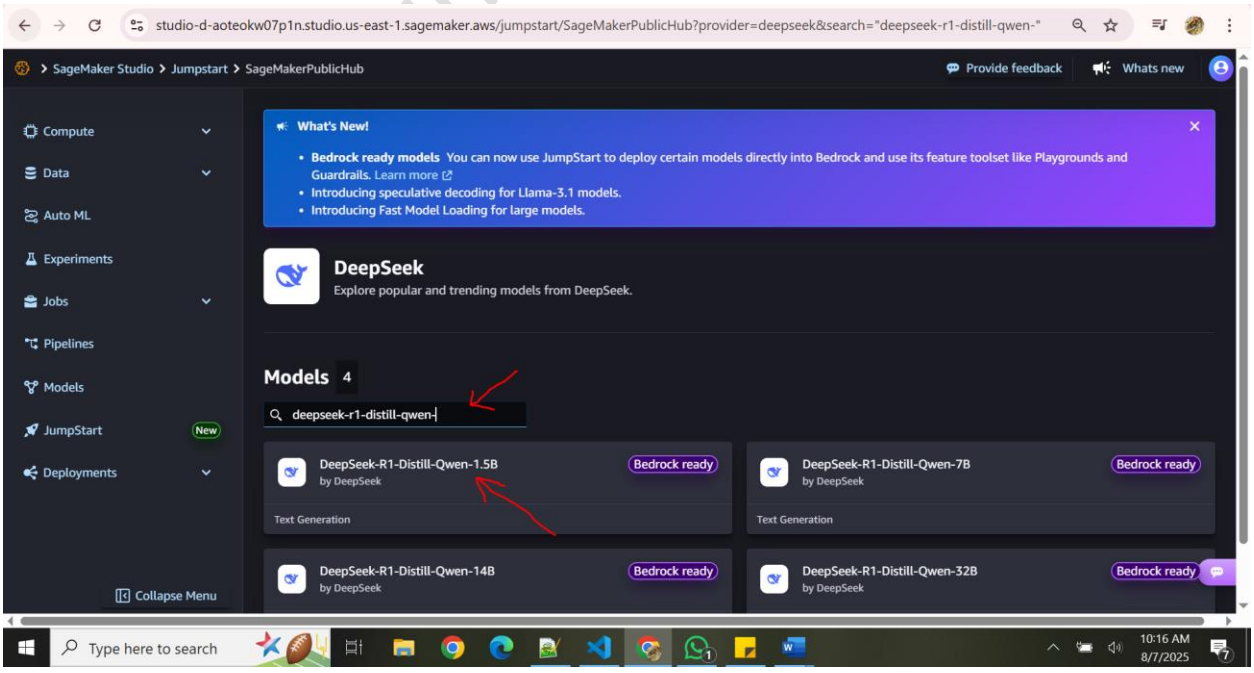
5. On the SageMaker Studio home page, click Open Studio. - SageMaker Studio opens in a new browser tab (or window). Keep the current browser tab open. You return to the SageMaker AI console in a later step.



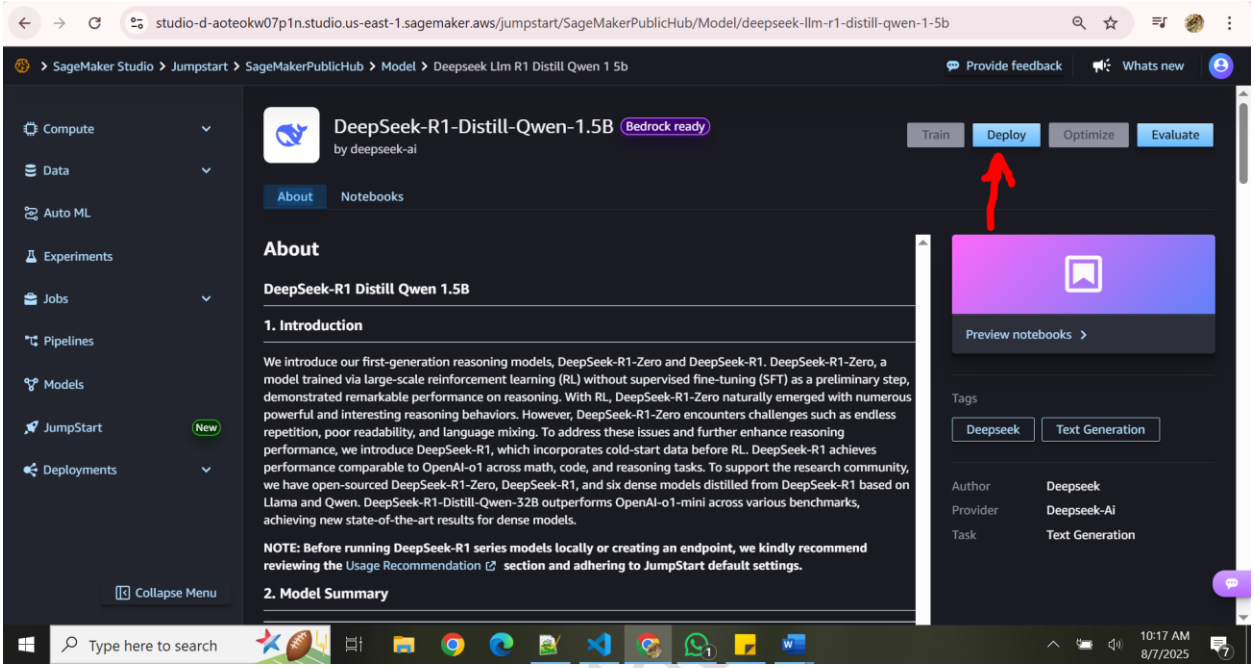
6. If a welcome pop-up box opens, click Skip Tour for now. 1. In the left navigation pane, click Jumpstart.
7. In the Providers search box, type DeepSeek. Click DeepSeek.



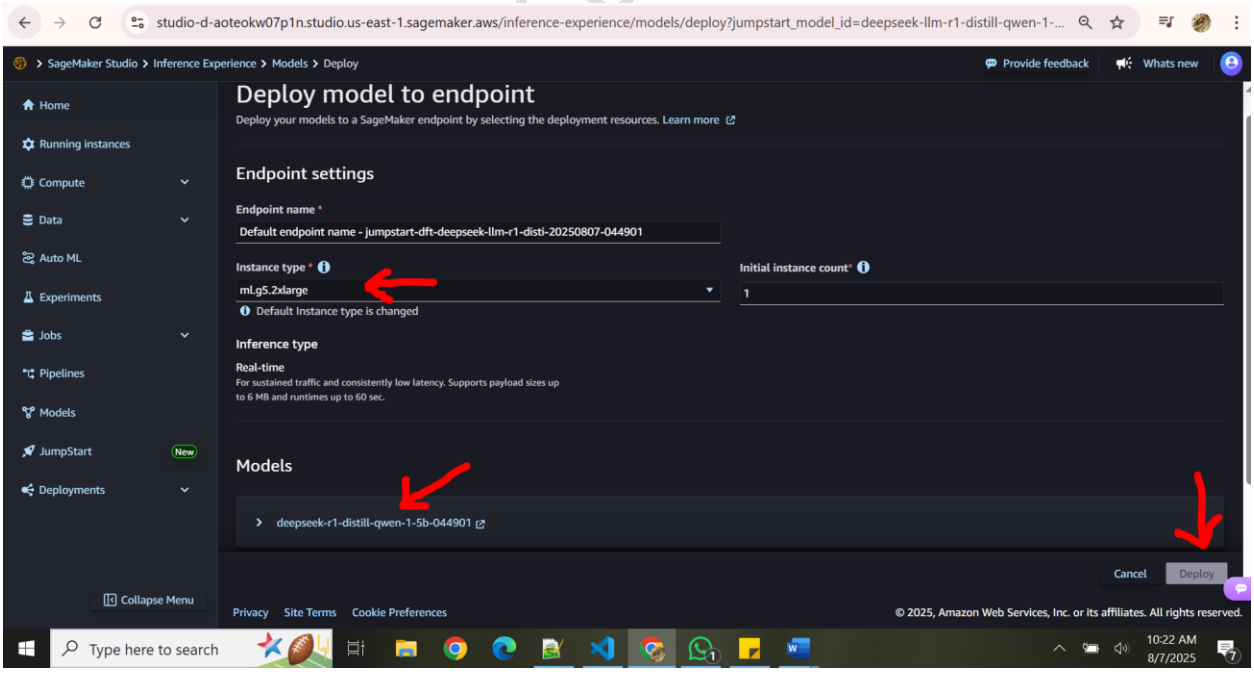
8. Under Models, find and click DeepSeek-R1-Distill-Qwen-1.5B. 2. Go to the next step.



9. Under Model information, review the information. 2. Click Deploy.

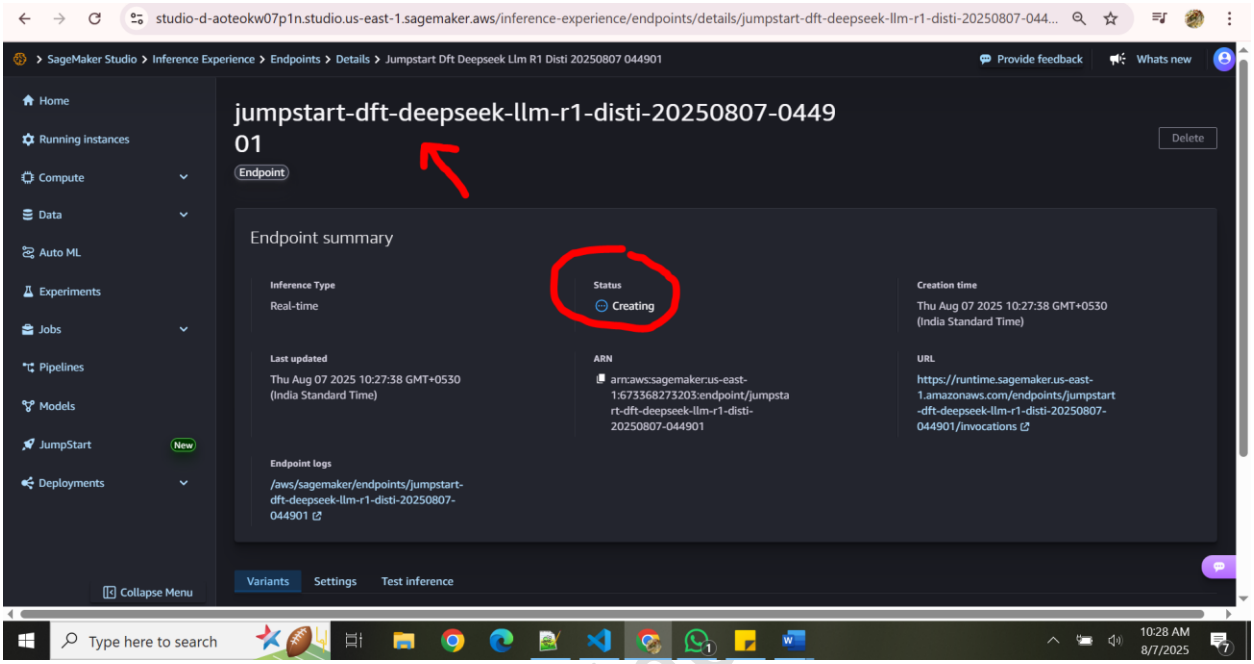


10. For Instance type, on the dropdown list, choose **ml.g5.2xlarge**. 2. Review the model name. Click Deploy.

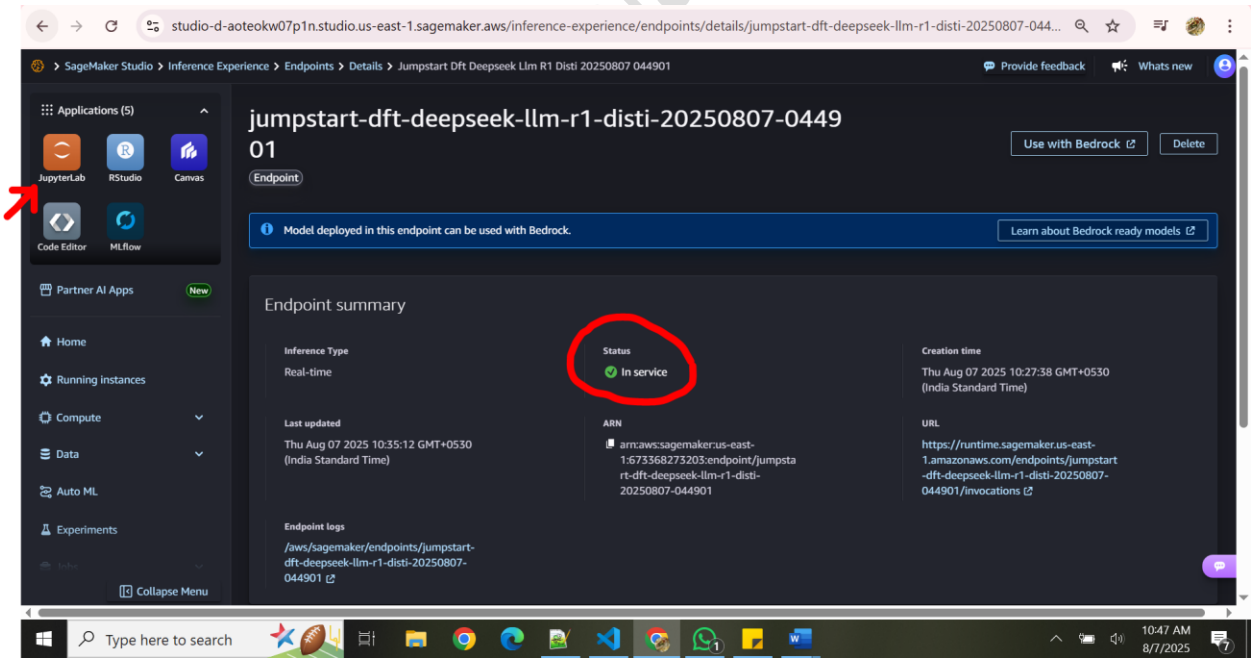


11. At the top of the page, select (highlight) and copy the full endpoint name, and then paste it in the text editor of your choice on your device. - You use this name in the upcoming steps. 2. Under Status, review the deployment status and wait for it to change

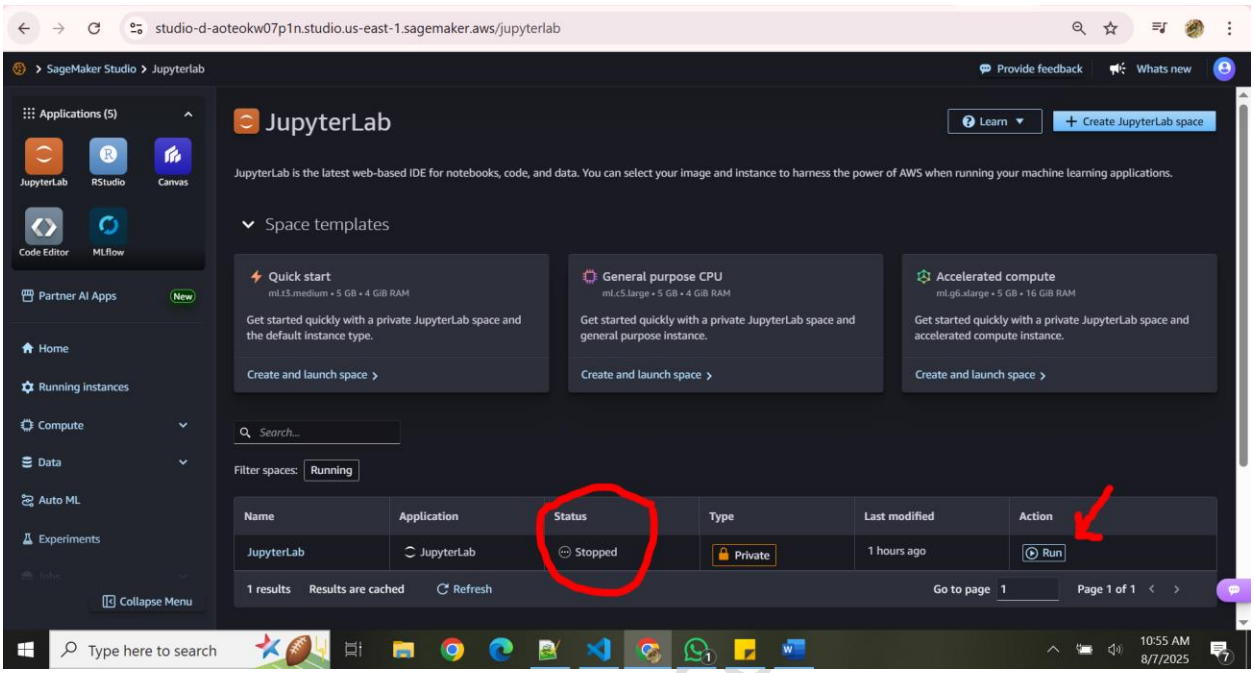
to In service. - The model can take up to 10 minutes to be deployed.



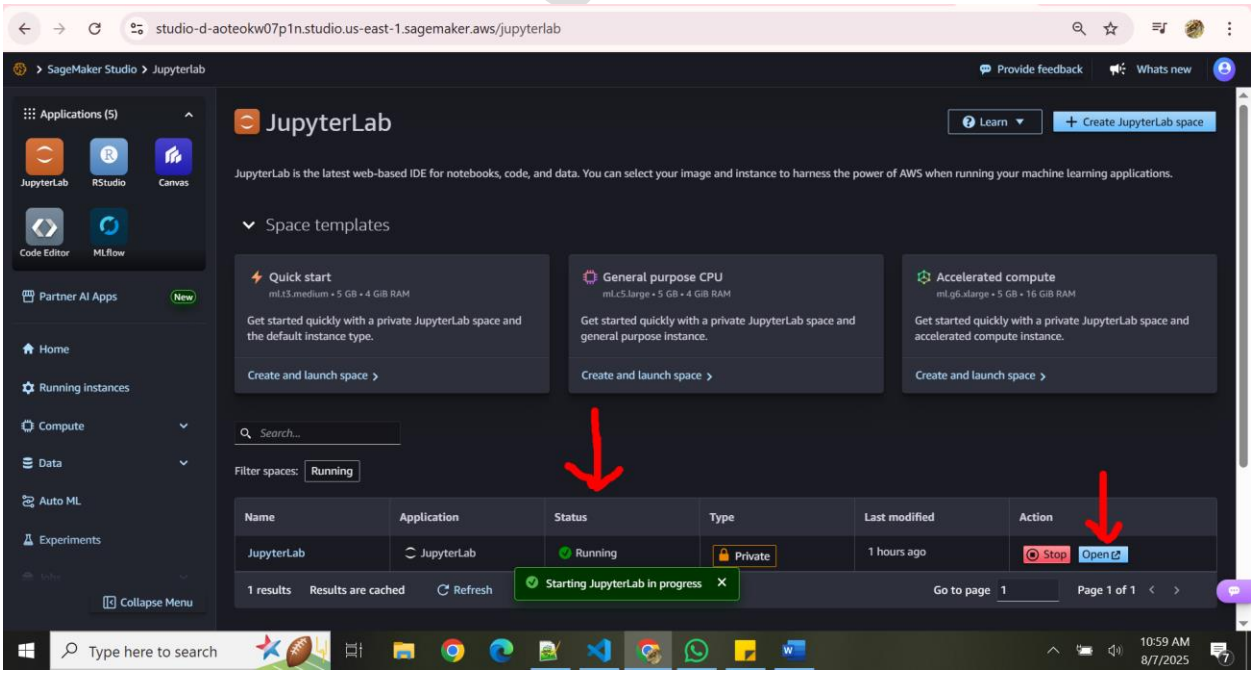
12. After the status changes, in the Applications pane, click JupyterLab.



13. On the JupyterLab page, under Status, review to confirm that the status is Stopped. 2. Under Action, click Run. - The status of the JupyterLab application can take 3–4 minutes to change to Running.

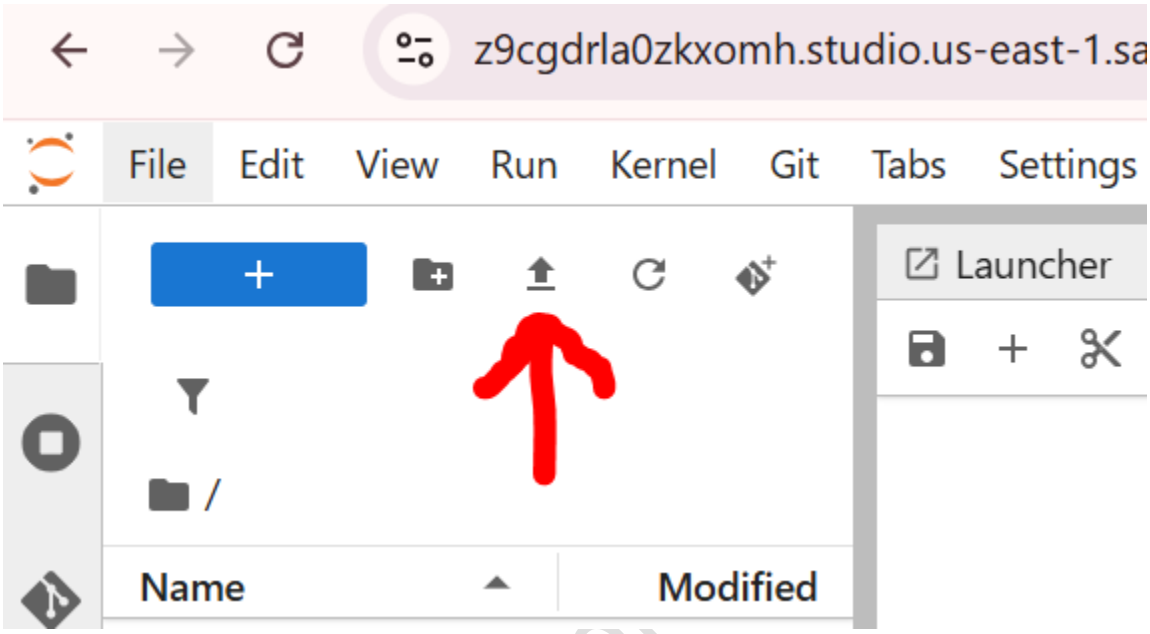


14. Under Status, review to confirm that the status is Running. 2. Under Action, click Open. - The JupyterLab application opens in a new browser tab (or window). - Keep the SageMaker Studio browser tab open. You return to it in later steps.

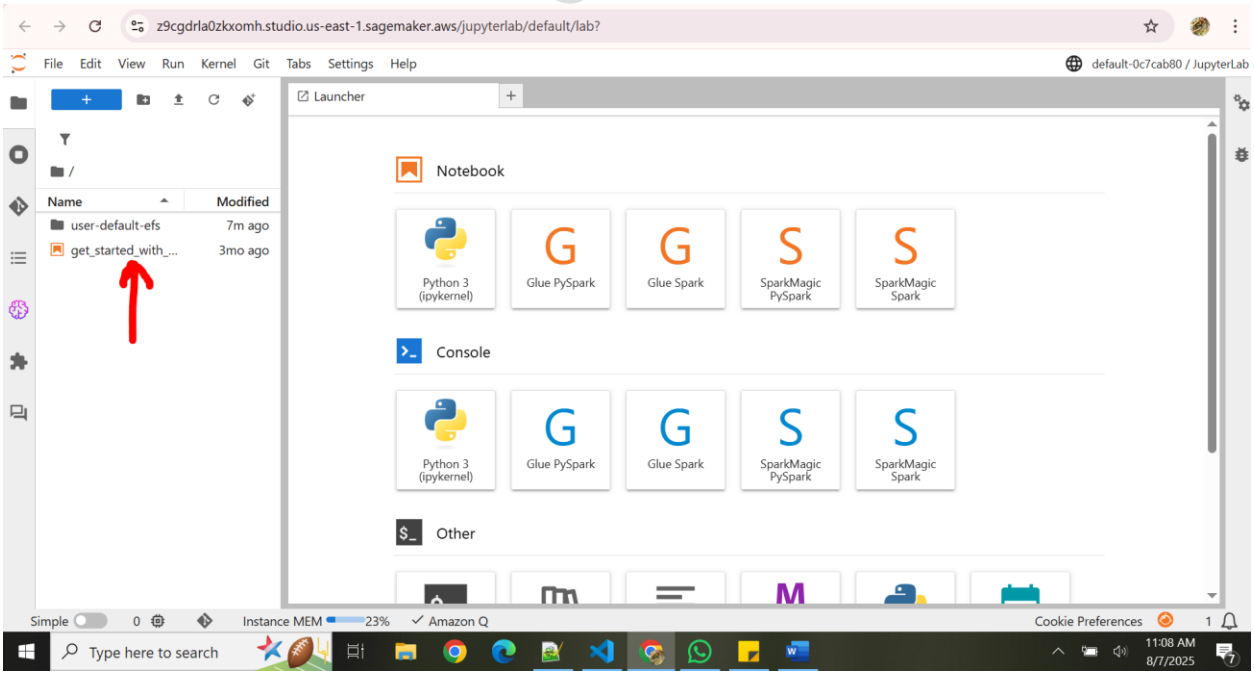


15. In the file browser, upload the attached file(get_started_with_generative_ai.ipynb). into the JupyterLab environment.

get_started_with_generative_ai.ipynb

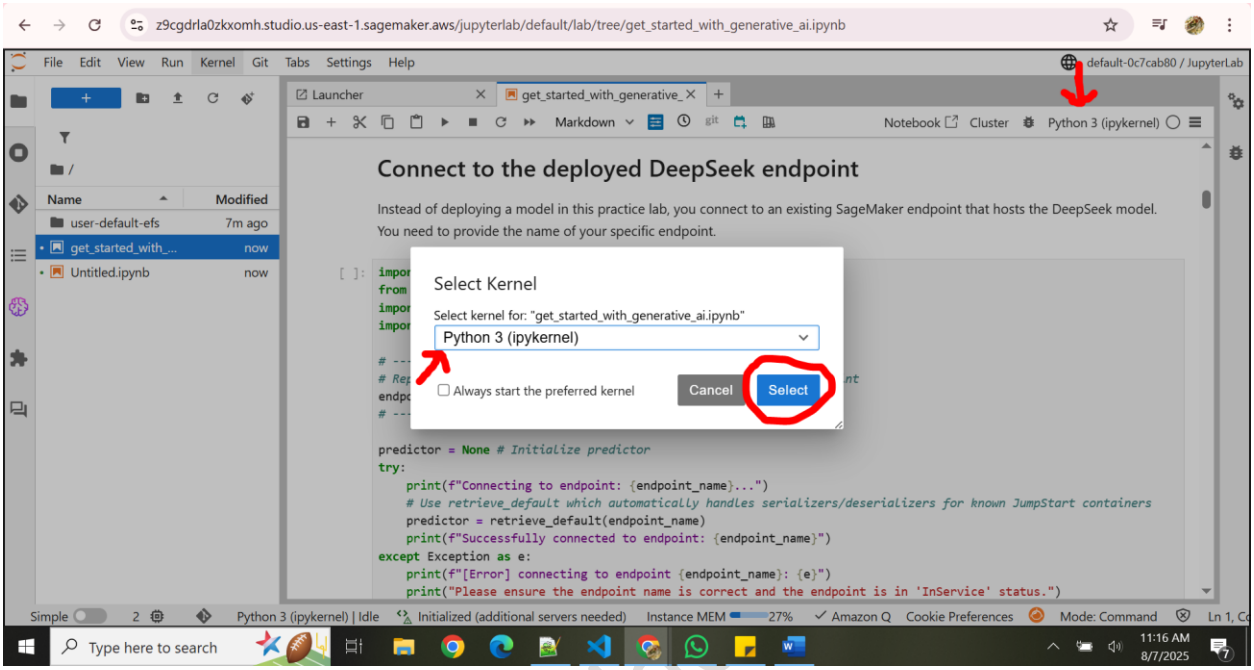


16.

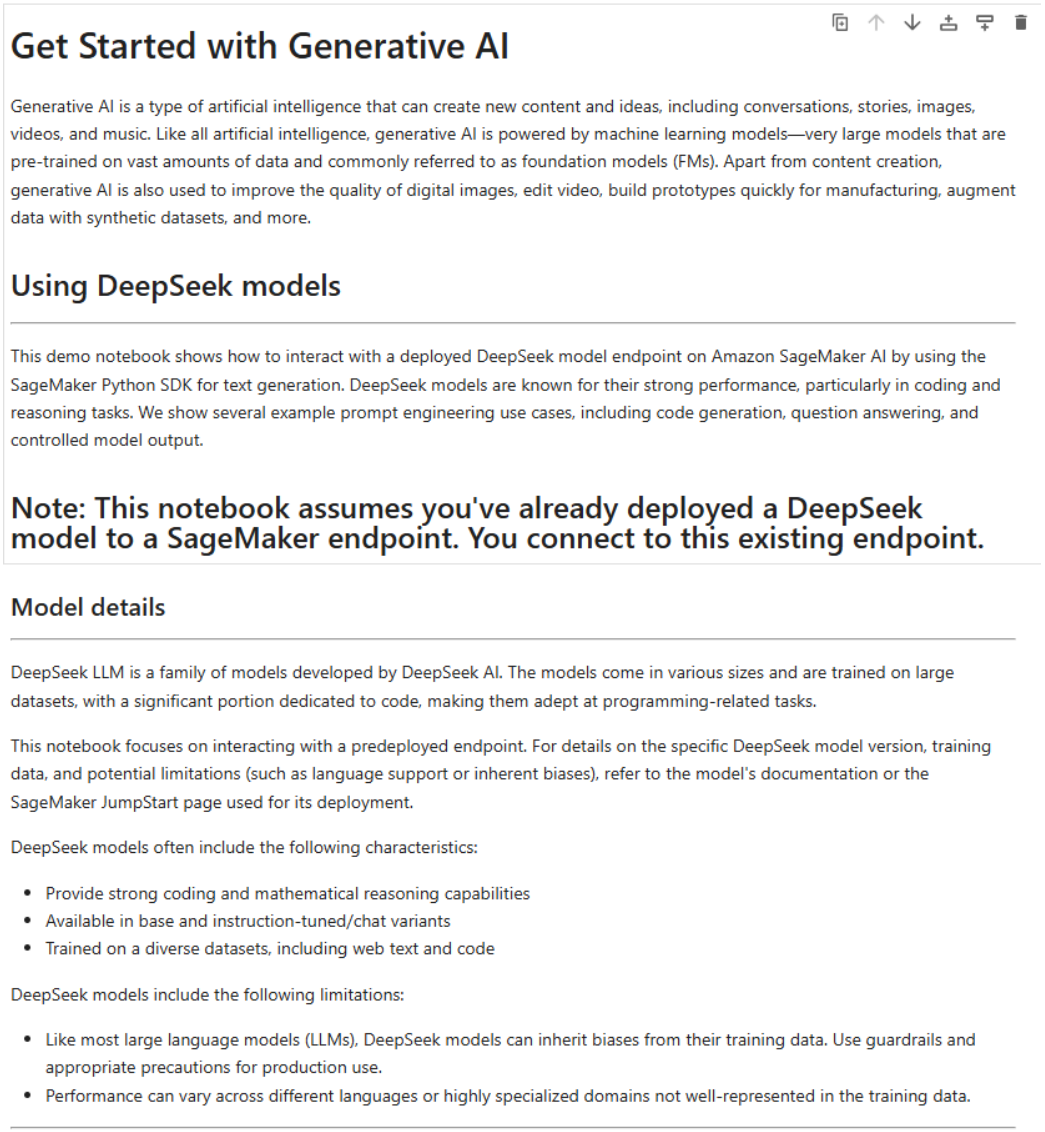


17. Open (double-click) the SageMaker Studio notebook file, get_started_with_generative_ai.ipynb. 2. In the pop-up box, choose Python 3 (ipykernel). - You might not see this pop-up box. 3. Click Select. 4. In the left sidebar, click

the File Browser (folder) icon to collapse the file browser.



18. If you did not choose Python 3 (ipykernel) in the previous step, at the top of the notebook window, review to confirm that it loaded automatically. - If needed, you can click the kernel name to open the pop-up box and select a Python 3. 2. In the notebook window, review the two introductory sections.
19. Review the Model details section.



Get Started with Generative AI

Generative AI is a type of artificial intelligence that can create new content and ideas, including conversations, stories, images, videos, and music. Like all artificial intelligence, generative AI is powered by machine learning models—very large models that are pre-trained on vast amounts of data and commonly referred to as foundation models (FMs). Apart from content creation, generative AI is also used to improve the quality of digital images, edit video, build prototypes quickly for manufacturing, augment data with synthetic datasets, and more.

Using DeepSeek models

This demo notebook shows how to interact with a deployed DeepSeek model endpoint on Amazon SageMaker AI by using the SageMaker Python SDK for text generation. DeepSeek models are known for their strong performance, particularly in coding and reasoning tasks. We show several example prompt engineering use cases, including code generation, question answering, and controlled model output.

Note: This notebook assumes you've already deployed a DeepSeek model to a SageMaker endpoint. You connect to this existing endpoint.

Model details

DeepSeek LLM is a family of models developed by DeepSeek AI. The models come in various sizes and are trained on large datasets, with a significant portion dedicated to code, making them adept at programming-related tasks.

This notebook focuses on interacting with a predeployed endpoint. For details on the specific DeepSeek model version, training data, and potential limitations (such as language support or inherent biases), refer to the model's documentation or the SageMaker JumpStart page used for its deployment.

DeepSeek models often include the following characteristics:

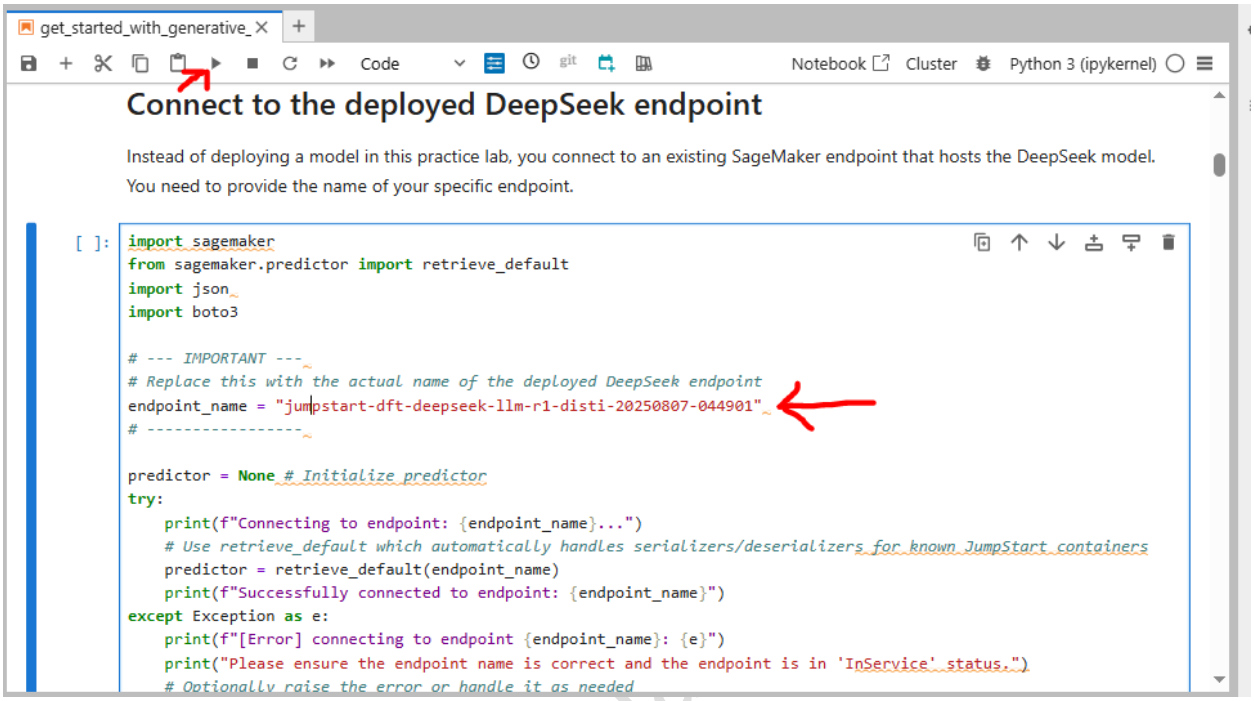
- Provide strong coding and mathematical reasoning capabilities
- Available in base and instruction-tuned/chat variants
- Trained on a diverse datasets, including web text and code

DeepSeek models include the following limitations:

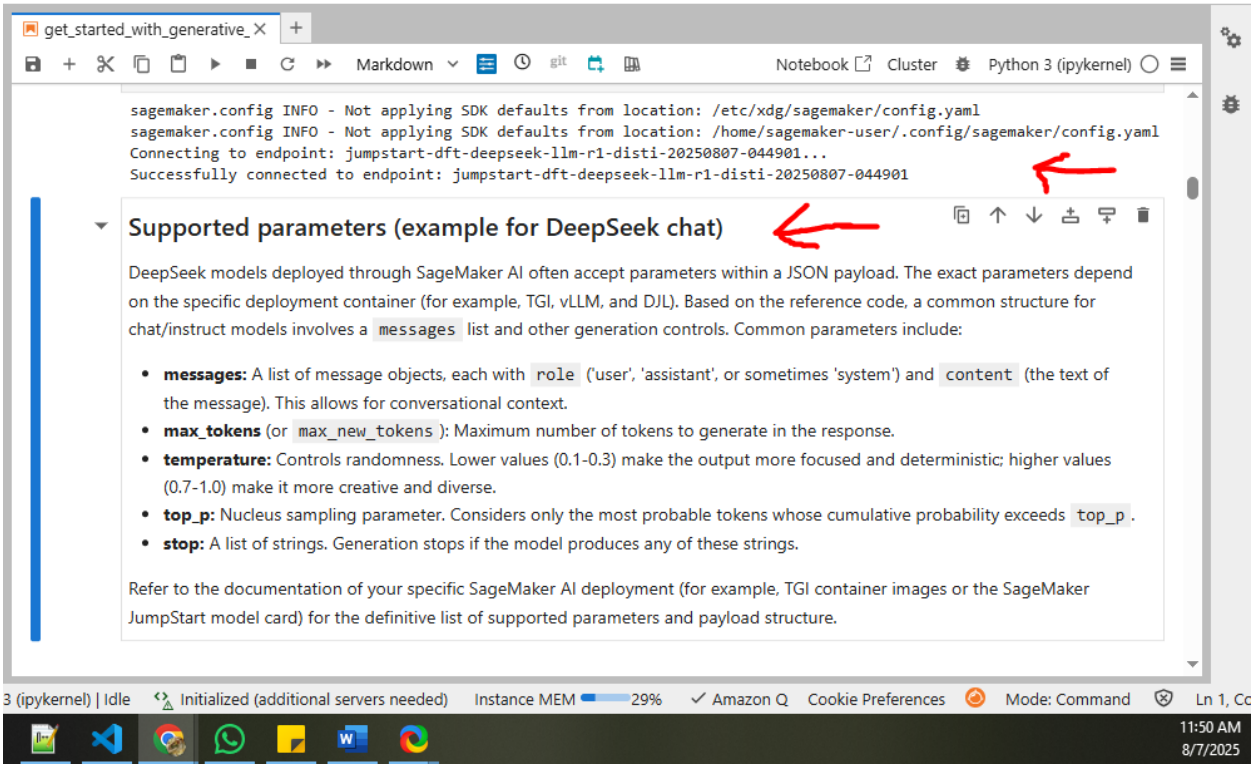
- Like most large language models (LLMs), DeepSeek models can inherit biases from their training data. Use guardrails and appropriate precautions for production use.
- Performance can vary across different languages or highly specialized domains not well-represented in the training data.

20. Under Connect to the deployed DeepSeek endpoint, in the code cell, to replace the <ENDPOINT_NAME> placeholder, paste the endpoint name that you copied in an earlier step. 2. Run the code cell. - In a Jupyter notebook, to run code in a cell, you can click in the cell, and then click the play icon on the toolbar. Or, you can press the [Shift]+[Enter] key combination. When the code cell finishes running, the text to the left of it changes from an asterisk [*] to a number. - You can safely ignore any warning alerts in the

output.

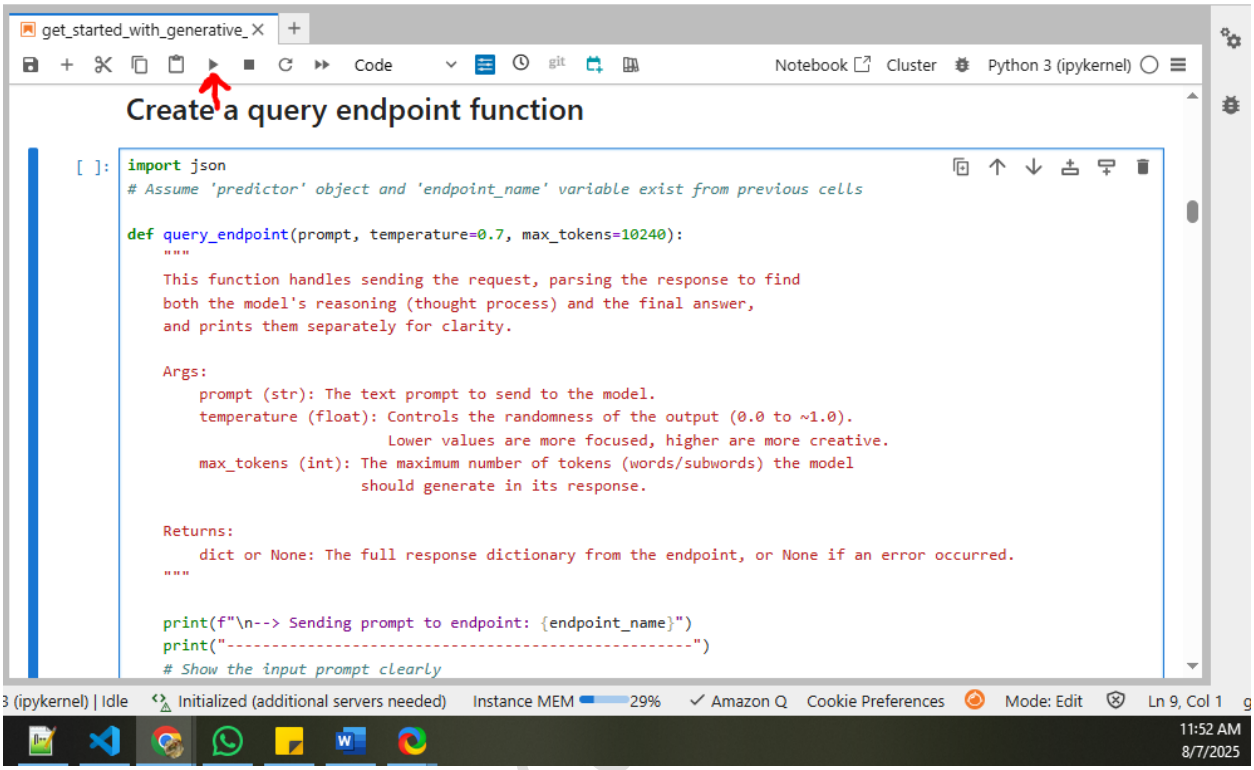


21. Review the code cell output. 2. Under Supported parameters, review the model parameters.

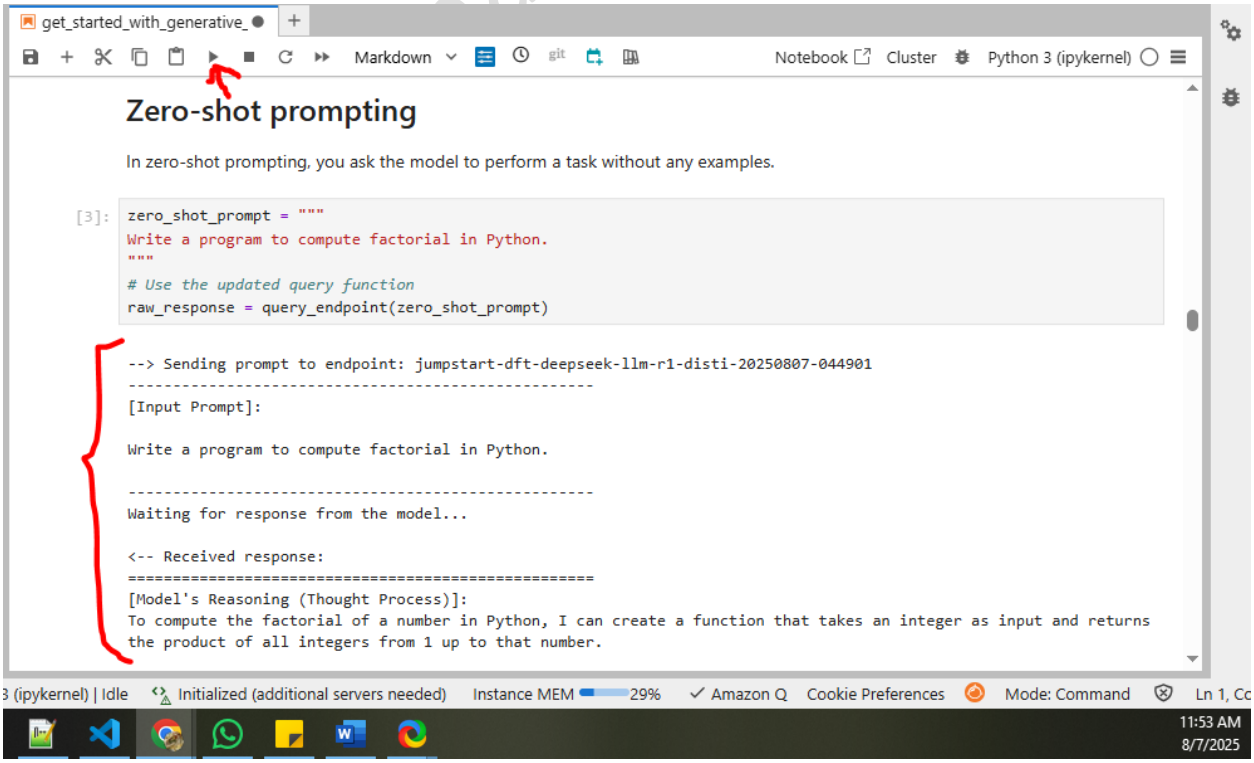


22. Under Create a query endpoint function, run the code cell. - This code creates a function to query the SageMaker endpoint. - This function facilitates the various prompts that are

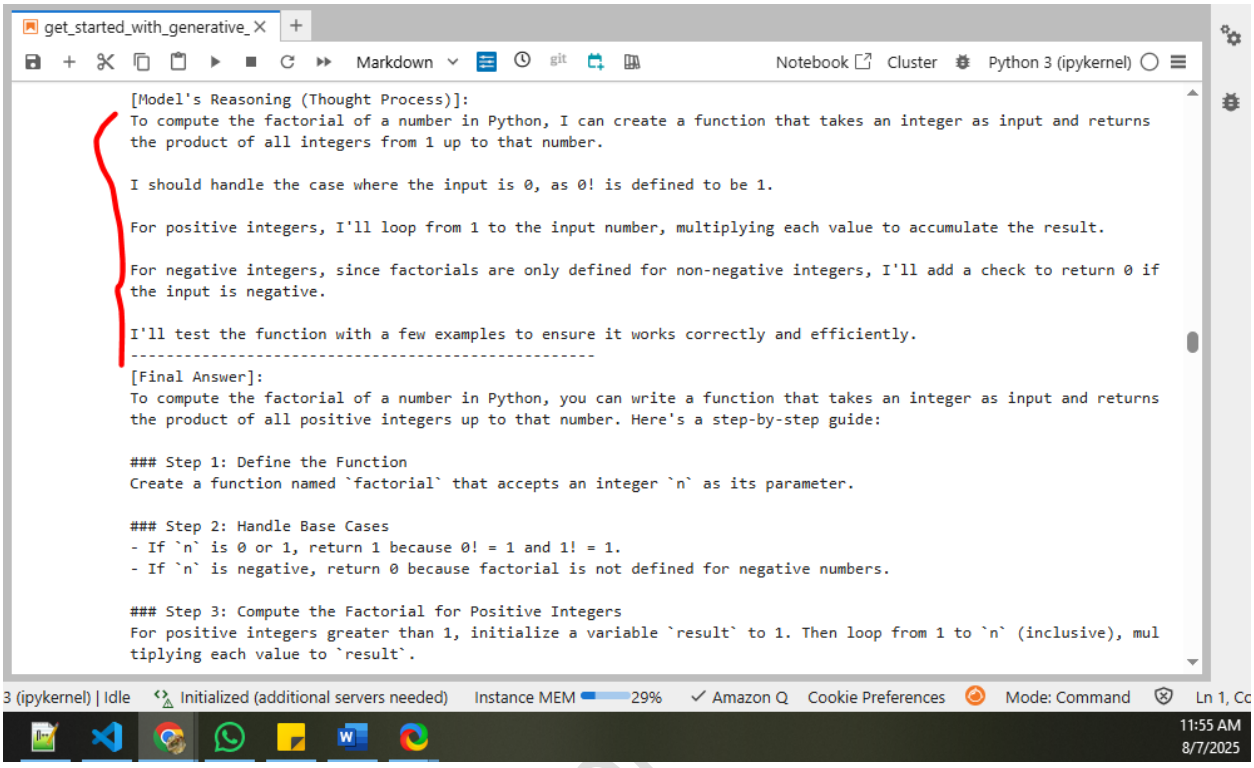
executed throughout the notebook.



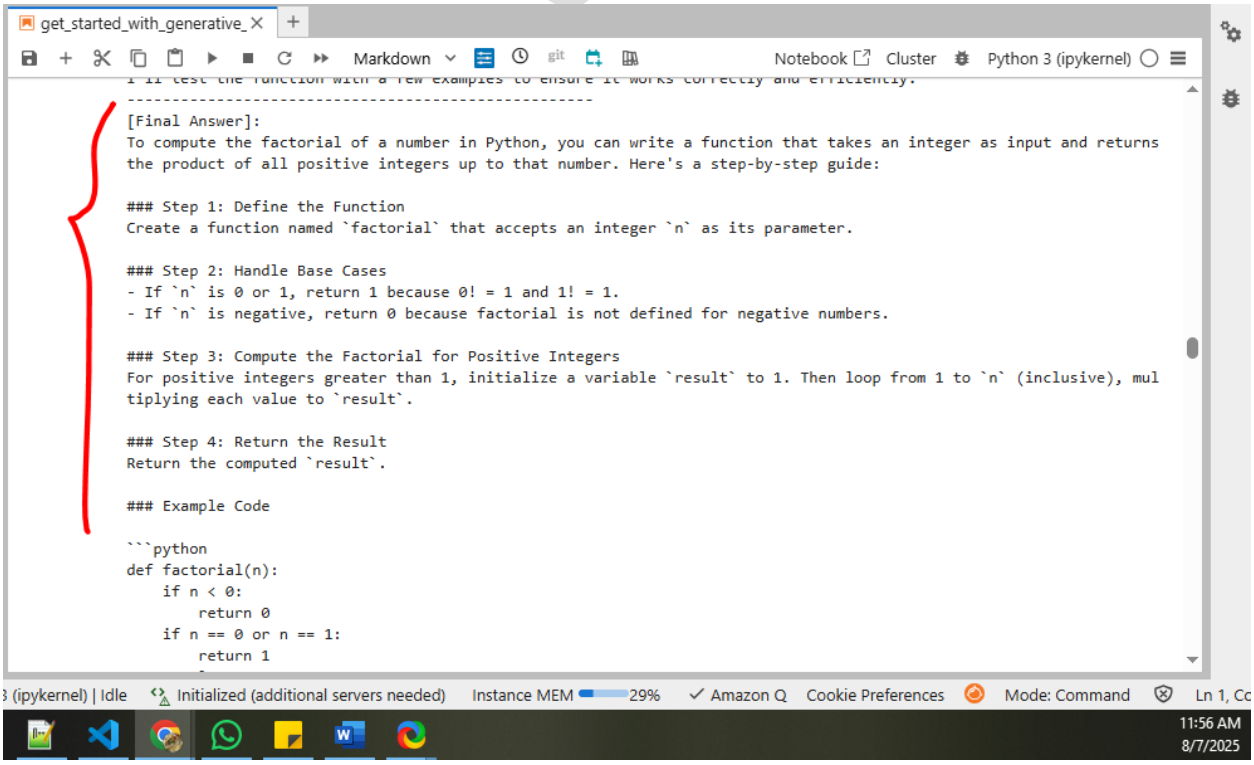
23. Under Zero-shot prompting, run the code cell. 2. Below the code cell, review the response. - The first part of the response is the model reasoning.

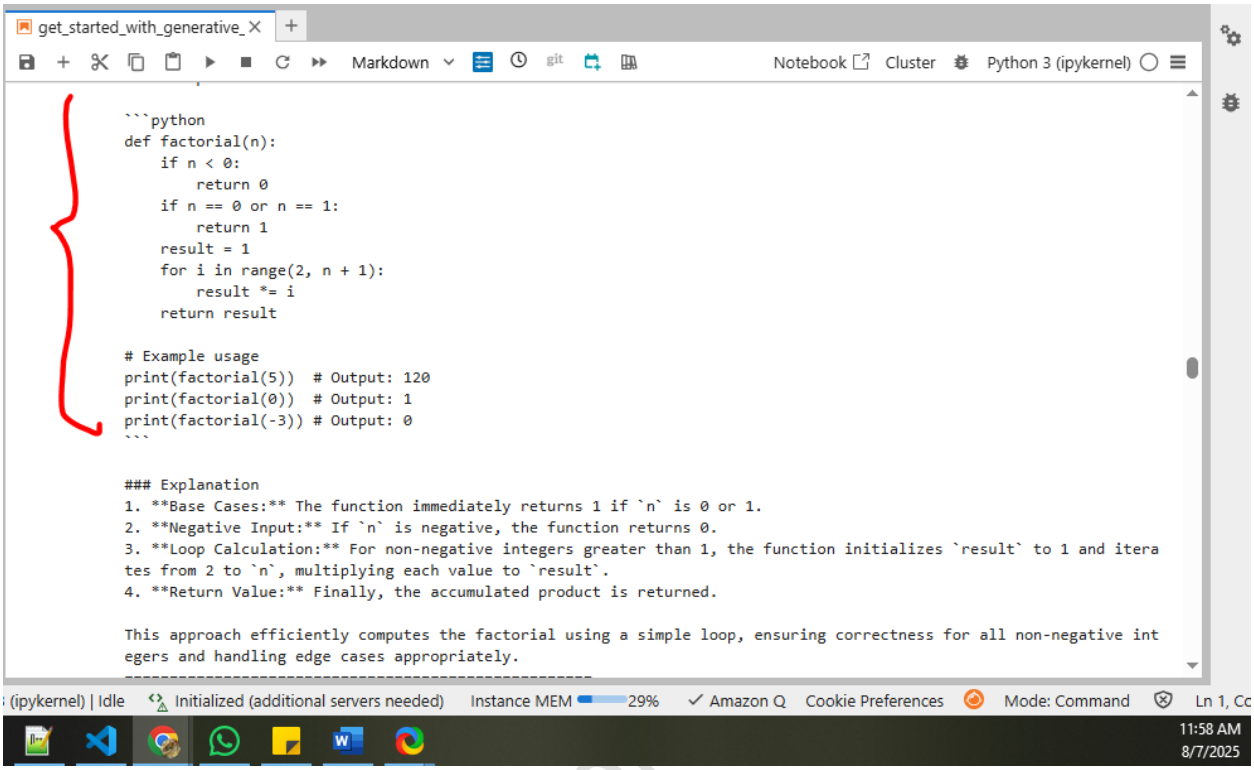


24. Review the model's reasoning.

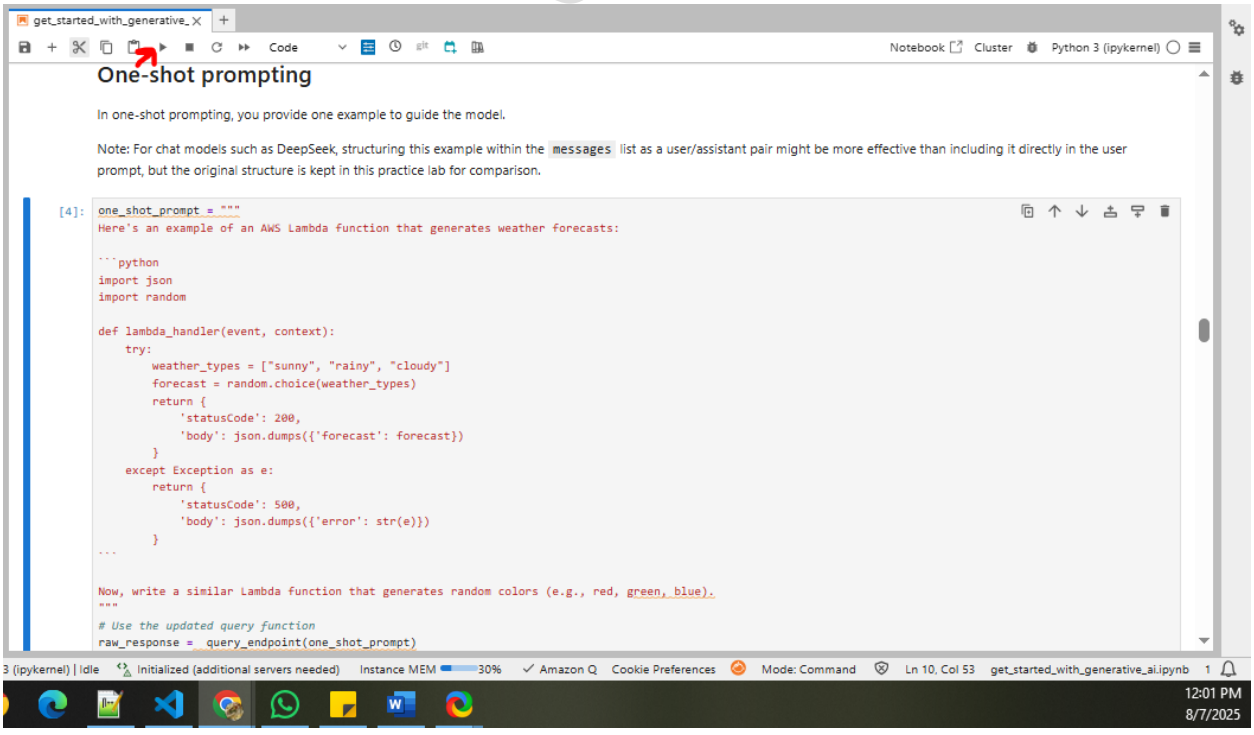


25. Under Final Answer, review the final answer.

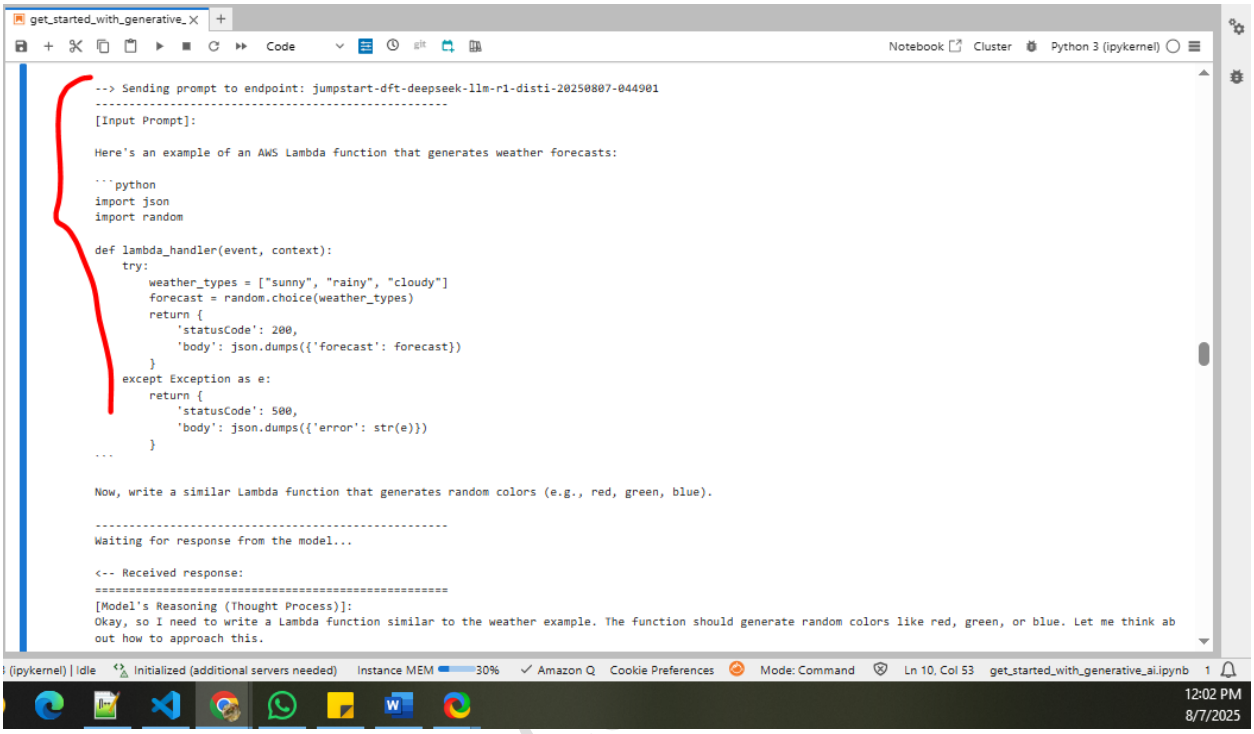




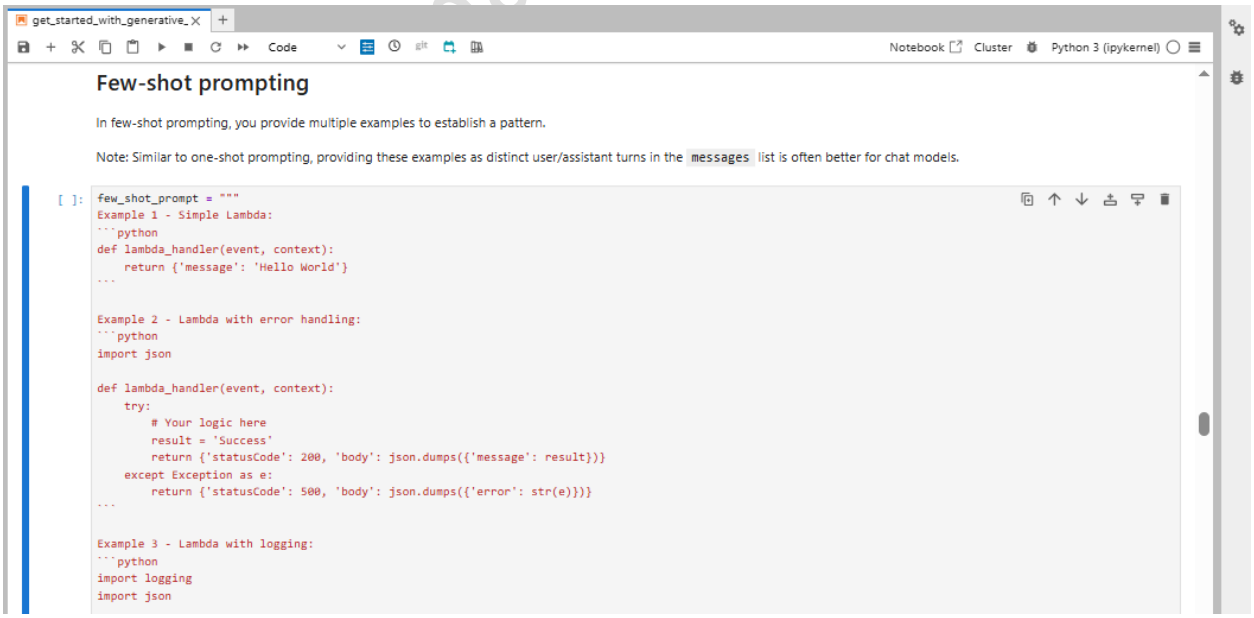
26. Under One-shot prompting, run the code cell.



27. Below the code cell, review the output. - Review both the reasoning section and final answers.



28. Under Few-shot prompting, run the code cell.



```
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    logger.info(f'Processing event: {event}')
    try:
        # Your logic here
        result = 'Logged and Processed'
        return {'statusCode': 200, 'body': json.dumps({'message': result})}
    except Exception as e:
        logger.error(f'Error processing event: {e}')
        return {'statusCode': 500, 'body': json.dumps({'error': str(e)})}
```

```
Now, create a Lambda function that:
1. Imports necessary libraries (random, logging, json)
2. Includes proper error handling (try/except block)
3. Implements logging for requests and errors
4. Defines a list of quotes from Greek philosophers (e.g., Socrates, Plato, Aristotle)
5. Randomly selects and returns one quote in the JSON body
6. Includes basic docstrings and comments
7. Returns a 200 status code on success and 500 on error.
"""

# Use the updated query function, keeping max_tokens high for longer code
# Increased max_tokens slightly more just in case
raw_response = query_endpoint(few_shot_prompt, max_tokens=1600)
```

29. Review the response.

```
--> Sending prompt to endpoint: jumpstart-dft-deepseek-llm-r1-disti-20250807-044901
------------------------------------------------------
[Input Prompt]:

Example 1 - Simple Lambda:
```python
def lambda_handler(event, context):
    return {'message': 'Hello World'}
```

Example 2 - Lambda with error handling:
```python
import json

def lambda_handler(event, context):
    try:
        # Your logic here
        result = 'Success'
        return {'statusCode': 200, 'body': json.dumps({'message': result})}
    except Exception as e:
        return {'statusCode': 500, 'body': json.dumps({'error': str(e)})}
```

Example 3 - Lambda with logging:
```python
import logging
import json

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    logger.info(f'Processing event: {event}')
    try:
        # Your logic here
        result = 'Logged and Processed'
        return {'statusCode': 200, 'body': json.dumps({'message': result})}
    except Exception as e:
        logger.error(f'Error processing event: {e}')
        return {'statusCode': 500, 'body': json.dumps({'error': str(e)})}
```

Now, create a Lambda function that:
1. Imports necessary libraries (random, logging, json)
2. Includes proper error handling (try/except block)
3. Implements logging for requests and errors
4. Defines a list of quotes from Greek philosophers (e.g., Socrates, Plato, Aristotle)
5. Randomly selects and returns one quote in the JSON body
6. Includes basic docstrings and comments
7. Returns a 200 status code on success and 500 on error.

Waiting for response from the model...

<-- Received response:
==
[Model's Reasoning (Thought Process)]:
Okay, I need to create a Lambda function that meets several requirements. Let me break down each requirement step by step.

First, I need to import necessary libraries: random, logging, and json. That's straightforward—just import them at the top.

Next, error handling with a try/except block. So, the function should have a try block where it performs its logic and an except block to catch any exceptions and return an error response.

Logging is another requirement. Logging for requests and errors is important for monitoring. I'll use the logging module, setting the level to INFO. In the try block, I'll add a logger.info to report when a quote is selected. In the except block, I'll log an error using logger.error.
```

I need a list of quotes from Greek philosophers. I'll create a list containing several quotes. Maybe include some famous ones like Socrates, Plato, and Aristotle.

The function should randomly select and return one quote. Using the random module, I can shuffle the list and pick the first element. I'll make sure to convert it to a string so it can be included in the JSON body.

Then, I need to add basic docstrings and comments. The docstring should explain what the function does, and each comment will describe a specific task like logging and error handling.

The function should return a 200 status code on success and 500 on error. So, in the success case, I'll return a JSON object with 'status' 200 and a message. In the error case, I'll return a JSON with 'status' 500.

Putting it all together, I'll structure the function with these elements: imports, try/except, logging, quotes, random selection, and error handling. I'll make sure the code is clean and well-commented for readability.

Wait, I should also consider any potential issues. For example, how will the quotes list be initialized? I'll define it once before the function. Also, the random selection should be done within the try block and then return the quote, else return the error response.

Let me outline the steps again:

1. Import required modules.
2. Define the function.
3. Inside try, select a random quote and log it.
4. If that's successful, return success response.
5. If not, enter except, log the error, and return the error response.
6. Ensure the docstring and comments are present.

I think that covers all the requirements. Now, I'll write the code accordingly.

[Final Answer]:
To solve this problem, we need to create a Lambda function that selects a random quote from a predefined list of Greek philosophers, logs the process, and handles errors appropriately. The function should return a success response with the selected quote and an error response with an appropriate message and status code.

Approach
1. **Imports**: Begin by importing the necessary libraries: `random`, `logging`, and `json`.
2. **Function Definition**: Define a function named `select_philosopher_quote` that takes no parameters and returns a response.
3. **Error Handling**: Use a try-except block to handle any exceptions that occur during the process.
4. **Logging**: Use logging to log the request and any error that occurs during the process.
5. **Quote Selection**: Use the `random.choice` function to randomly select a quote from the list and log the process.
6. **Response**: Return a JSON response indicating the success of the request. If an error occurs, return an error JSON response with the appropriate status code.
7. **Docstrings and Comments**: Add a docstring and comments to explain the purpose of the function, the imports used, and the key steps involved.

Solution Code

```python
import random
import logging
import json

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

def select_philosopher_quote():
    """Selects a random quote from a list of Greek philosophers and returns it in JSON format.

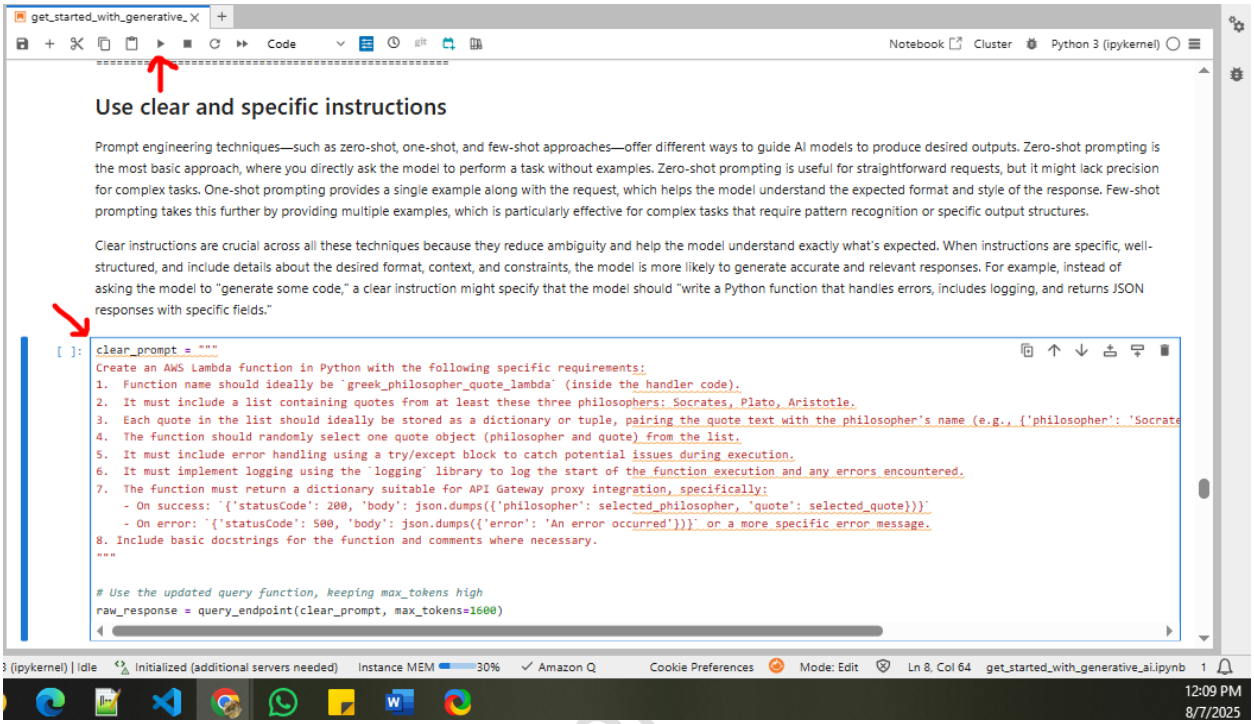
    This function uses a try-except block to handle any exceptions that occur during the
    request and error logging. It returns a success response with the selected quote or
    an error response with the appropriate status code."""
    quotes = [
        "Socrates was one of the first to give a general opinion on the value of living a
        life worth living.",
        "Plato believed that knowledge should be distributed equally among all,
        regardless of social status.",
        "Aristotle believed that human potential lies in our potential to seek knowledge."
    ]

    try:
        randomly_selected_quote = random.choice(quotes)
        logger.info(f"Selected quote: {randomly_selected_quote}")
        return {"status": 200, "body": json.dumps({"message": randomly_selected_quote})}
    except Exception as e:
        logger.error(f"Error processing request: {e}")
        return {"status": 500, "body": json.dumps({"error": str(e)})}
```

Explanation
- **Imports**: The necessary libraries are imported to ensure compatibility with the Lambda function.
- **Function Definition**: The function is defined to be self-contained and does not require any parameters.
- **Error Handling**: The function uses a try-except block to handle any exceptions that might occur during the selection of a quote or logging.
- **Logging**: The `logger` instance is configured to log messages with an INFO level, providing a clear indication of the request's state.
- **Quote Selection**: The `random.choice` function is used to randomly select a quote from the predefined list, ensuring that the selection is unbiased.
- **Response**: The function returns a JSON response indicating the success of the request with the selected quote in the body. If an error occurs, it returns an error JSON response with the appropriate status code and message.
- **Docstrings and Comments**: The function includes a docstring explaining its purpose, the imports used, and key steps involved, along with comments for clarity.

This approach ensures that the Lambda function is robust, handles errors gracefully, and provides clear logging and documentation for better maintainability.
==========

30. Review the Use clear and specific instructions description. 2. Run the code cell.



31. Review the response.

```
--> Sending prompt to endpoint: jumpstart-dft-deepseek-llm-r1-distl-20250807-044901
------------------------------------------------------
[Input Prompt]:

Create an AWS Lambda function in Python with the following specific requirements:
1.  Function name should ideally be `greek_philosopher_quote_lambda` (inside the handler code).
2.  It must include a list containing quotes from at least these three philosophers: Socrates, Plato, Aristotle.
3.  Each quote in the list should ideally be stored as a dictionary or tuple, pairing the quote text with the philosopher's name (e.g., {'philosopher': 'Socrat
es', 'quote': 'An unexamined life is not worth living.'}).
4.  The function should randomly select one quote object (philosopher and quote) from the list.
5.  It must include error handling using a try/except block to catch potential issues during execution.
6.  It must implement logging using the `logging` library to log the start of the function execution and any errors encountered.
7.  The function must return a dictionary suitable for API Gateway proxy integration, specifically:
    - On success: `{'statusCode': 200, 'body': json.dumps({'philosopher': selected_philosopher, 'quote': selected_quote})}`
    - On error: `{'statusCode': 500, 'body': json.dumps({'error': 'An error occurred'})}` or a more specific error message.
8. Include basic docstrings for the function and comments where necessary.

------------------------------------------------------
Waiting for response from the model...

<-- Received response:
======================================================
[Model's Reasoning (Thought Process)]:
Okay, so I have this task where I need to create an AWS Lambda function in Python based on the user's requirements. Let me go through each point step by step t
o make sure I understand what's needed.

First, the function name is specified as `greek_philosopher_quote_lambda`, which is a bit of a name, but I guess it's just a placeholder. The main job of this
function is to retrieve quotes from three famous Greek philosophers: Socrates, Plato, and Aristotle. Each quote should be stored as a dictionary with the philo
sopher's name and the quote text.

I also need to import the necessary libraries. The user mentioned logging, so I'll need to import the logging module. Plus, using the requests library for HTTP
requests and json for JSON operations. So I'll include those imports at the top.

Next, the function should randomly select one quote object from the list. Each quote is a dictionary of the form {'philosopher': 'Socrates', 'quote': 'An unexa
mined life is not worth living.'}. So I'll need to store the list of these quote dictionaries somewhere, probably in a variable.

Error handling is a must. I'll use try/except blocks to catch any exceptions that might occur during the function's execution, such as network errors or reques
ts failing. This will log the error and return a 500 status code with an appropriate message.

Logging is another key point. I'll import the logging library and set up the function to log the start of execution using logging.info. Also, during errors, lo
gging.error will be useful.

The success outcome should return a JSON-dumped dictionary with the selected philosopher and quote. For errors, it should log that an error occurred and return
a 500 status with the error message.

I should include comments where necessary to explain the code, but since the user asked for basic docstrings and comments, I'll write a docstring explaining th
e function's purpose, parameters, return values, and note about including the .env file and AWS setup in production.

Now, thinking about the structure of the function. It will start by logging the function's start. Then, in a try block, it will generate a list of three random
quote dictionaries. It'll use a for loop iterating three times, each time selecting a quote with random selection. After that, it will randomly select one quot
e from the list.

If everything goes well, it logs the selected quotes and returns the JSON-dumped dictionary. If there's an error, it catches the exception, logs the error, and
returns the 500 status with the message.

I should also consider edge cases. For example, what if the API endpoints aren't correctly set up? The function should handle that gracefully but inform the us
er that it didn't work.

Wait, the user mentioned the .env file and AWS setup, but since I'm writing this as a script, maybe I should note that in a comment. They also asked for basic
docstrings and comments, which I've done.

Let me structure the code. Start with the imports. Then, the function definition. Inside the function, log the start. Generate the quote list, select one, chec
k for errors. Handle exceptions, log errors, and return the appropriate response.

I think that covers all the points. Now, I'll write the code accordingly, making sure to handle all the requirements mentioned.
------------------------------------------------------
[Final Answer]:
```python
import logging
import requests
import json

logging.basicConfig(level=logging.INFO)
logging.getLogger(__name__)..setLevel(logging.DEBUG)

def greek_philosopher_quote_lambda():
    """
    Function to retrieve and random select Greek philosophical quotes.

    Returns:
        {success: {status: 200, body: {philosopher: "Socrates", quote: ...}}}
        {error: {status: 500, body: {error: ...}}}

    Raises:
        Exception: If network errors occur or requests fail to authenticate.
    """
    # Initialize logging
    logging.info("Starting Greek philosophical quotes function")

    # List of philosopher quote pairs
    quote_list = [
        {'philosopher': 'Socrates', 'quote': 'An unexamined life is not worth living.'},
        {'philosopher': 'Plato', 'quote': 'Let no one who is without a vision of an ideal society enter my door.'},
        {'philosopher': 'Aristotle', 'quote': 'The unexamined life is not worth considering: Only the examined life is
worth living.'}
    ]

    try:
        # Randomly select one quote
        selected_quote = random.choice(quote_list)

        # Generate three random quotes
        random_quotes = [random.choice(quote_list) for _ in range(3)]
        selected_quotes = random.sample(random_quotes, k=len(quote_list))[:3]

        # Select one quote randomly
        selected_quote = random.choice(selected_quotes)

        logging.info(f"Successfully retrieved and selected a quote: {selected_quote['philosopher']} - {selected_quote['quote']}")
        return json.dumps(selected_quote)

    except Exception as e:
        logging.error(f"Error occurred during function execution: {str(e)}")
        return json.dumps({
            'error': f"An error occurred during function execution: {str(e)}"
        })

# Example usage:
# lambda: greek_philosopher_quote_lambda()
# ```

This implementation meets all the requirements:
- Uses a try/except block to handle errors
- Implements logging for both success and error cases
- Randomly selects quotes from the list
- Returns appropriate responses for success and errors
- Includes error handling using the logging and try/except blocks
- Uses proper docstrings and comments
- Handles potential API errors gracefully

To use this in an AWS environment, you'll need to:
1. Make sure you have the necessary AWS services (EC2, S3) set up
2. Include credentials in your .env file
3. Set up the AWS service account
4. Set up proper logging setup
5. Set up AWS Lambda with appropriate endpoint configurations
======================================================
```
```

32. Review the Temperature and creativity control description. 2. Run the first code cell. 3. Review the response.
33. Run the second code cell. 2. Review the response.

Temperature and creativity control

Temperature in prompt engineering controls the randomness or predictability of a model's responses. A lower temperature (closer to 0) makes responses more deterministic and focused, which is better for factual or technical tasks where accuracy is crucial. A higher temperature (closer to 1 or above) introduces more randomness, leading to more creative, diverse, and surprising outputs, which works well for creative writing or brainstorming. The optimal temperature setting depends on your specific use case; use lower values when you need consistency and precision, and use higher values when you want variety and creativity.

Controlling model creativity

[7]:
```
prompt="What is AWS Lambda?"

# Low temperature (more focused, deterministic)
print("\n--- Low Temperature (Focused, temp=0.2) Response --- ")
raw_response = query_endpoint(prompt, temperature=0.2, max_tokens=32000)
```

```
--- Low Temperature (Focused, temp=0.2) Response ---

--> Sending prompt to endpoint: jumpstart-dft-deepseek-llm-r1-disti-20250807-044901
------------------------------------------------------------
[Input Prompt]:
What is AWS Lambda?
------------------------------------------------------------
Waiting for response from the model...

<-- Received response:
============================================================
[Model's Reasoning (Thought Process)]:
Okay, so I'm trying to understand what AWS Lambda is. I've heard the term before, maybe in the context of cloud computing or something related to serverless fu
nctions. Let me start by breaking down the name. AWS is a big company, right? They make a lot of cloud services. Lambda must be one of them.

I remember hearing about serverless functions before. They're something like functions that run on a serverless platform, which I think is a type of cloud comp
uting service. So maybe Lambda is a specific service within that category. But I'm not entirely sure how it works.

Let me think about what I know. Lambda functions are typically written in Python or JavaScript and called when a request comes in. They can handle data process
ing, route requests, maybe even do some database operations. But I've also heard that Lambda can run on a serverless platform, which means it doesn't need its
own server. That sounds efficient because you don't have to worry about hosting your own server.

So, AWS Lambda is probably a service that allows you to create and run these serverless functions. It's part of the AWS Lambda Function API, which I think is a
web service that lets you create these functions. But wait, isn't there also a service that actually runs these functions? Maybe it's Lambda Function itself, w
hich is the API, and then there's something like Lambda Function Container or Lambda Function Services.

I'm a bit confused about the differences between these. Let me try to outline what I know:

1. **Lambda Function API**: This is the web service where you can create, manage, and run serverless functions. It's built on top of Python and can handle vari
ous tasks like processing data, routing, and even database operations.

2. **Lambda Function Container**: I think this is a service that runs Lambda functions on a serverless platform. It's probably a separate service that you can
use to run Lambda functions without needing to interact with the Lambda API directly.

3. **Lambda Function Services**: These are pre-configured services that provide a set of Lambda functions. They might include common tasks like processing, rou
ting, caching, and more. These services are probably easier to use because they're pre-built and ready to use.

So, the Lambda Function API is the main tool for creating and running these functions, while the container and services are the actual instances where these fu
nctions run. That makes sense because you don't want to build your own serverless functions from scratch; you can leverage existing tools.

I also recall that Lambda can be used in different environments. For example, you can run it on a local machine, on a serverless platform like AWS S3, AWS Lamb
da Function Container, or even on a serverless instance like AWS Lambda. This flexibility is important because it allows you to choose the best fit for your ne
eds.

Another thing I'm thinking about is how Lambda functions handle data. They can process data in various ways, like mapping, reducing, filtering, and so on. They
can also handle asynchronous data, which is useful for real-time applications. Plus, they can perform database operations, which is great for data processing t
asks.

I'm also curious about the benefits of using Lambda. Since it's serverless, it's more efficient because you don't have to manage your own servers. It's scalabl
e, which means you can scale up or down based on demand. Plus, it's easier to deploy and maintain because you don't have to worry about infrastructure manageme
nt.

But I'm not entirely sure how Lambda functions are integrated into the AWS ecosystem. Do they work with other services like S3, EKS, or Lambda Function Service
s? I think they can, but I'm not certain. I should probably look into how Lambda functions interact with these services to see how they can be used in differen
t scenarios.

I also wonder about the differences between Lambda Function API and Lambda Function Services. The former is the tool for creating and running functions, while
the latter is the pre-configured instances. So, if I want to run a function, I'd use the API, but if I need a set of functions, I'd use the services.

Another point is the use of AWS services like S3 and EKS. Lambda can run on these services, which means you can have your Lambda functions deployed in differen
t environments. For example, you can deploy a Lambda function on a local serverless instance, on a serverless platform, or even on a managed service like AWS S
3.

I'm also thinking about the types of tasks Lambda functions can perform. They can handle data processing, route requests, perform database operations, and even
handle events. This versatility makes Lambda a powerful tool for various applications, from backend services to frontend components.

In summary, AWS Lambda is a service that allows you to create and run serverless functions using the AWS Lambda Function API. These functions can process data,
route requests, and perform database operations. They can be deployed on various AWS services like local machines, serverless platforms, or managed instances.
The Lambda Function API is the main tool, while Lambda Function Services are pre-configured instances that provide common tasks. Lambda functions are efficien
t, scalable, and easier to deploy compared to traditional serverless solutions.

I think I have a basic understanding now, but I should probably look up some examples or documentation to solidify my knowledge. Maybe checking out AWS documen
tation or tutorials on AWS Lambda would help clarify any gaps in my understanding.
------------------------------------------------------------
[Final Answer]:
AWS Lambda is a cloud-based service that enables the creation and execution of serverless functions, allowing you to handle tasks like data processing, route r
equests, and database operations without managing your own servers. Here's a structured overview of AWS Lambda:

1. **Introduction to Lambda**:
   - Lambda is part of the AWS Lambda Function API, which is a web service for creating and managing serverless functions.

2. **Key Components**:
   - **Lambda Function API**: The main tool for creating, managing, and running serverless functions. It supports Python and JavaScript, handling tasks like da
ta processing and routing.
   - **Lambda Function Container**: A service that runs Lambda functions on a serverless platform, providing a platform for Lambda functions to execute.
   - **Lambda Function Services**: Pre-configured instances offering common Lambda functions, such as processing, routing, and caching, making them easier to u
se.

3. **Functionality**:
   - Lambda functions can process data using tasks like mapping, reducing, filtering, and handling asynchronous data.
   - They support database operations, making them versatile for various applications.

4. **Deployment and Use**:
   - Lambda functions can be deployed on local machines, serverless platforms, or managed instances like AWS Lambda.
   - They are efficient, scalable, and easier to deploy compared to traditional solutions.

5. **Integration with AWS Services**:
   - Lambda functions can run on AWS S3, EKS, or managed instances, allowing flexibility in deployment environments.

6. **Benefits**:
   - Serverless efficiency, scalability, and ease of deployment make Lambda a powerful tool for modern applications.

In summary, AWS Lambda is a versatile and efficient service for creating serverless functions, offering a range of functionalities and deployment options to su
it various needs.
============================================================
```

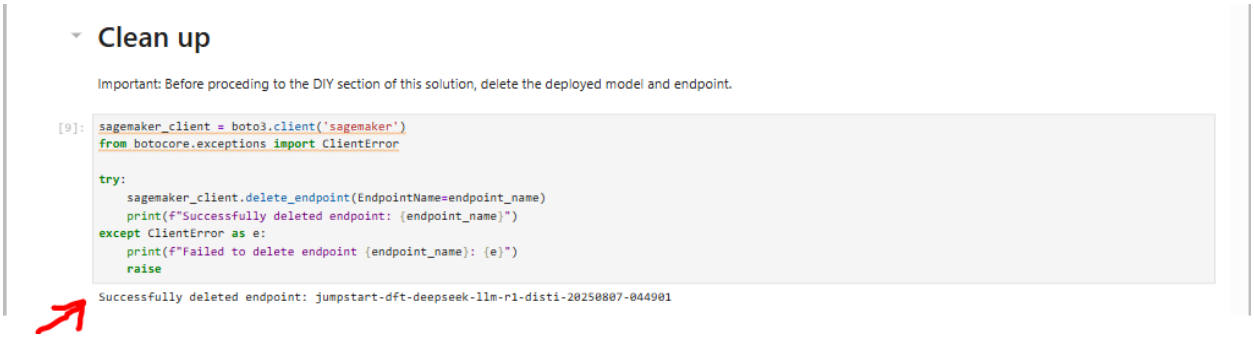
[*]:
```
prompt="Write a haiku about the challenges of software development and code maintenance."

# Higher temperature (more creative, diverse)
print("\n--- High Temperature (Creative, temp=0.9) Response --- ")
raw_response = query_endpoint(prompt, temperature=0.9, max_tokens=32000)
```

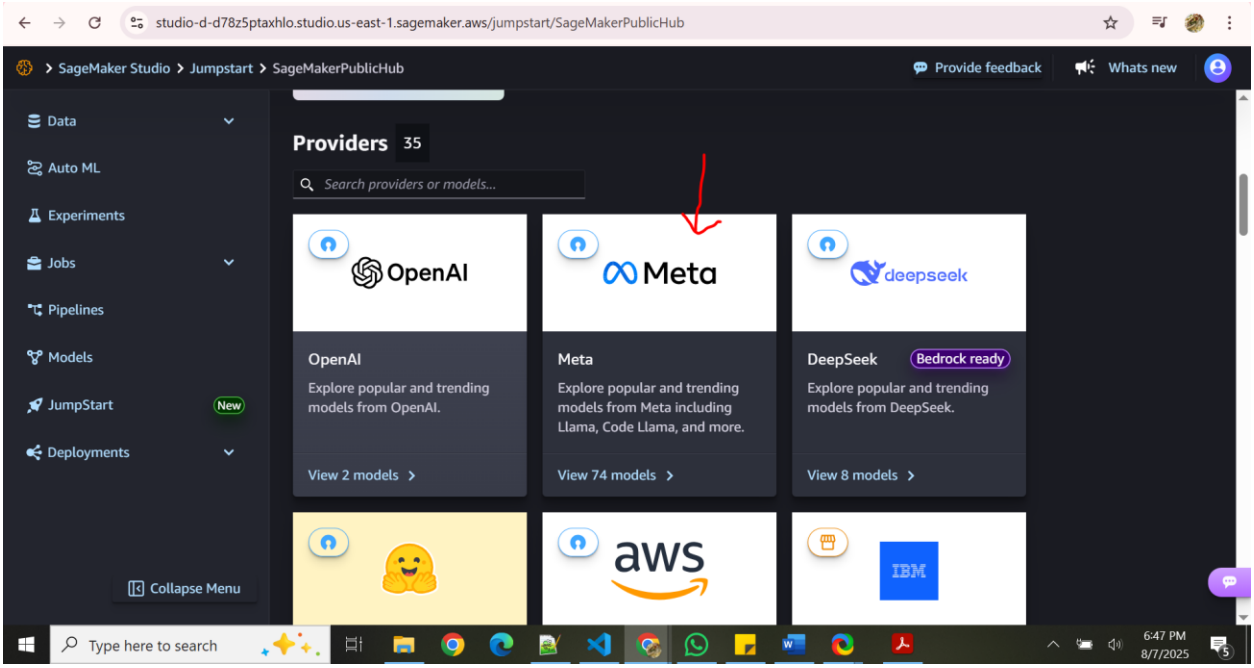
```
--- High Temperature (Creative, temp=0.9) Response ---

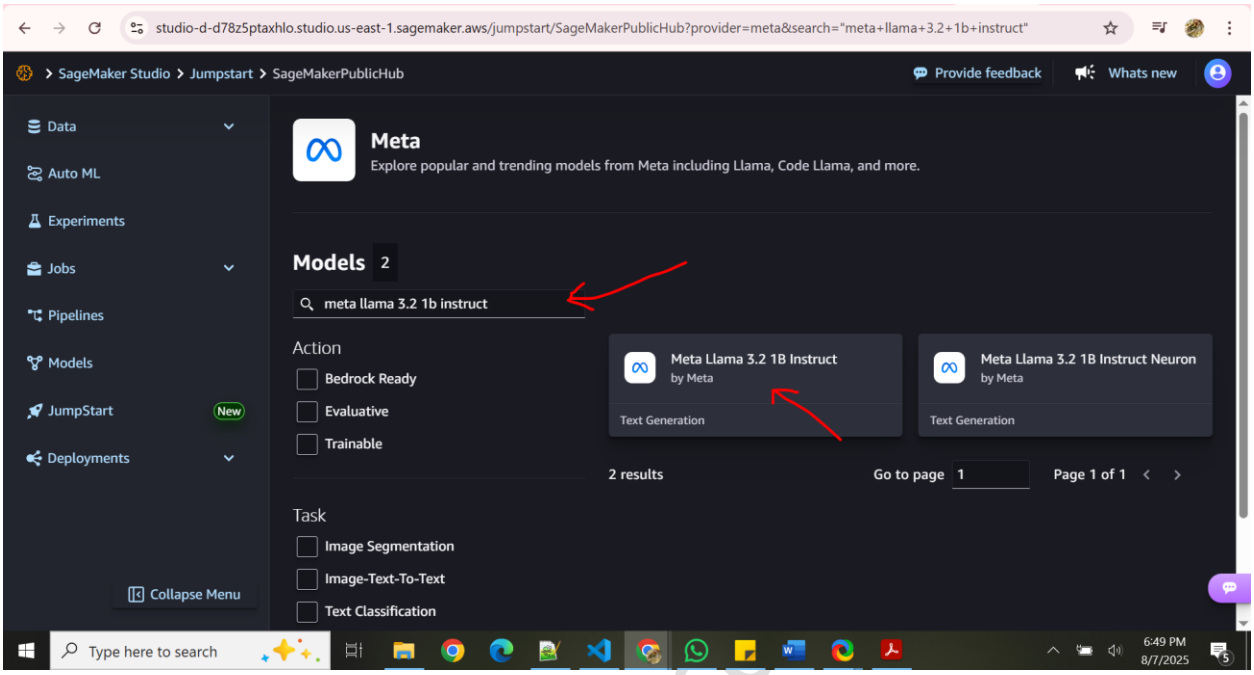
--> Sending prompt to endpoint: jumpstart-dft-deepseek-llm-r1-disti-20250807-044901
------------------------------------------------------------
[Input Prompt]:
Write a haiku about the challenges of software development and code maintenance.
------------------------------------------------------------
Waiting for response from the model...
```

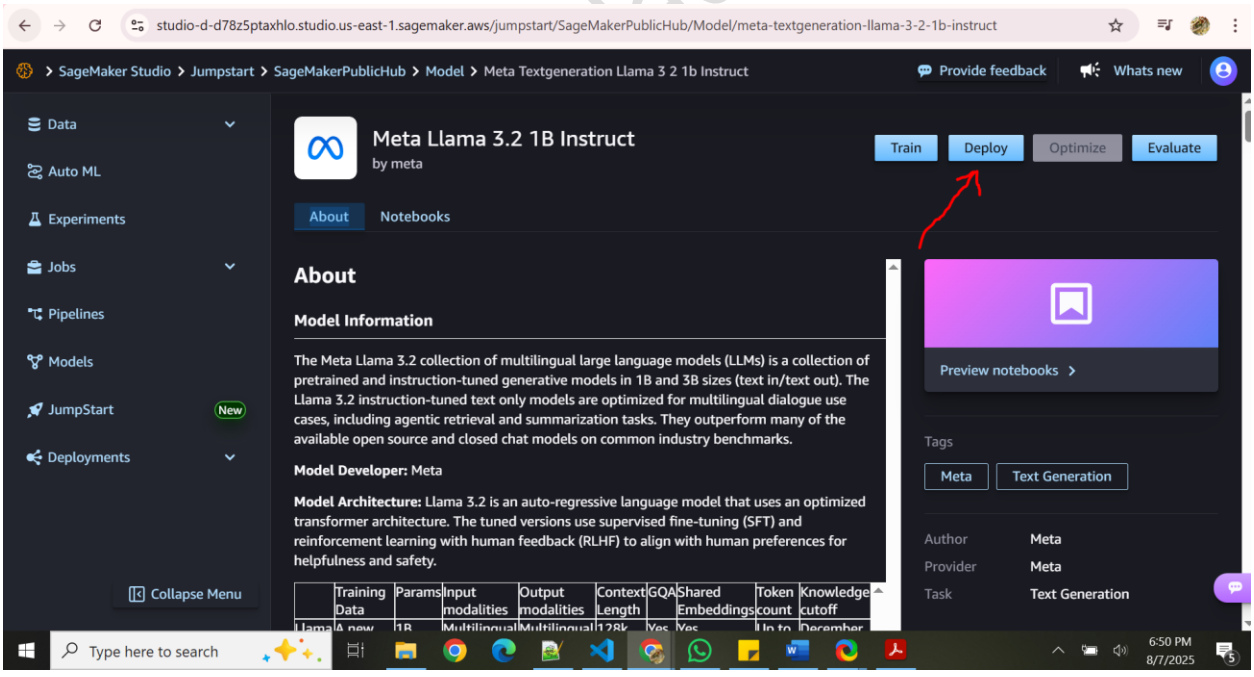
34. Under Clean up, run the code cell.



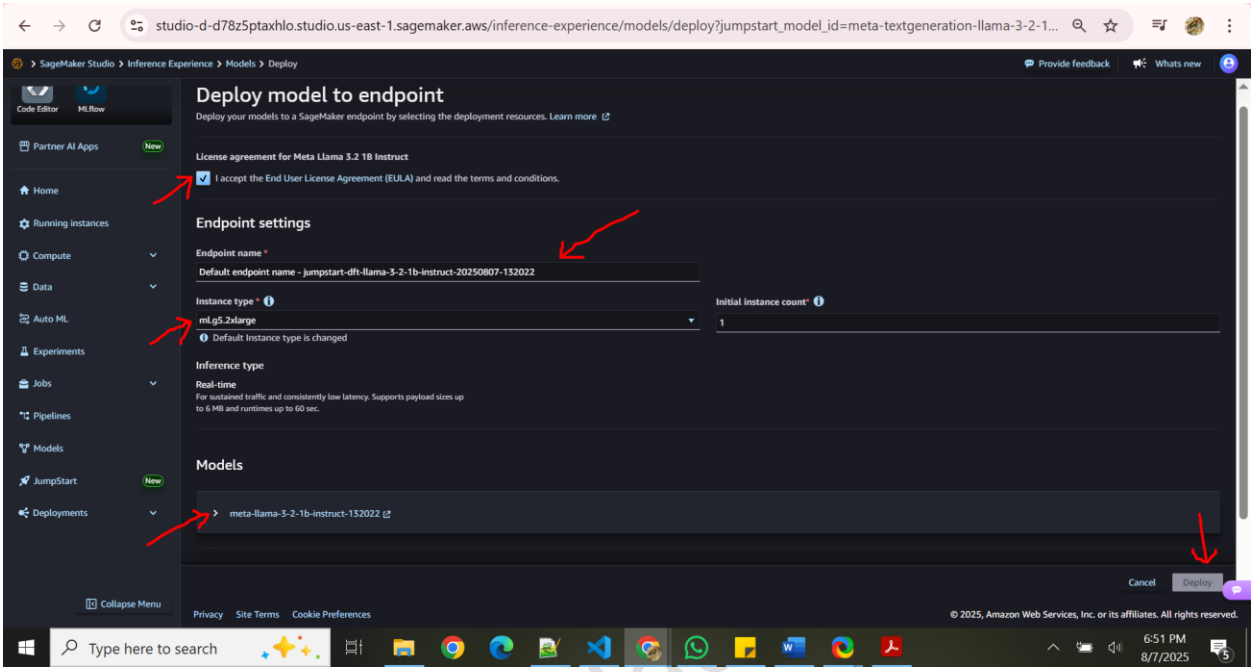
35. Under Do it yourself, review the code cell. - In the next steps, you deploy a new model by using the console. You then return to run the remaining code cells.
36. At the bottom of the notebook, review the Your turn... instructions. - After deploying the model for the DIY section, return here to test your prompt.
37. Deploy the Meta Llama 3.2 1B Instruct model in SageMaker Studio.

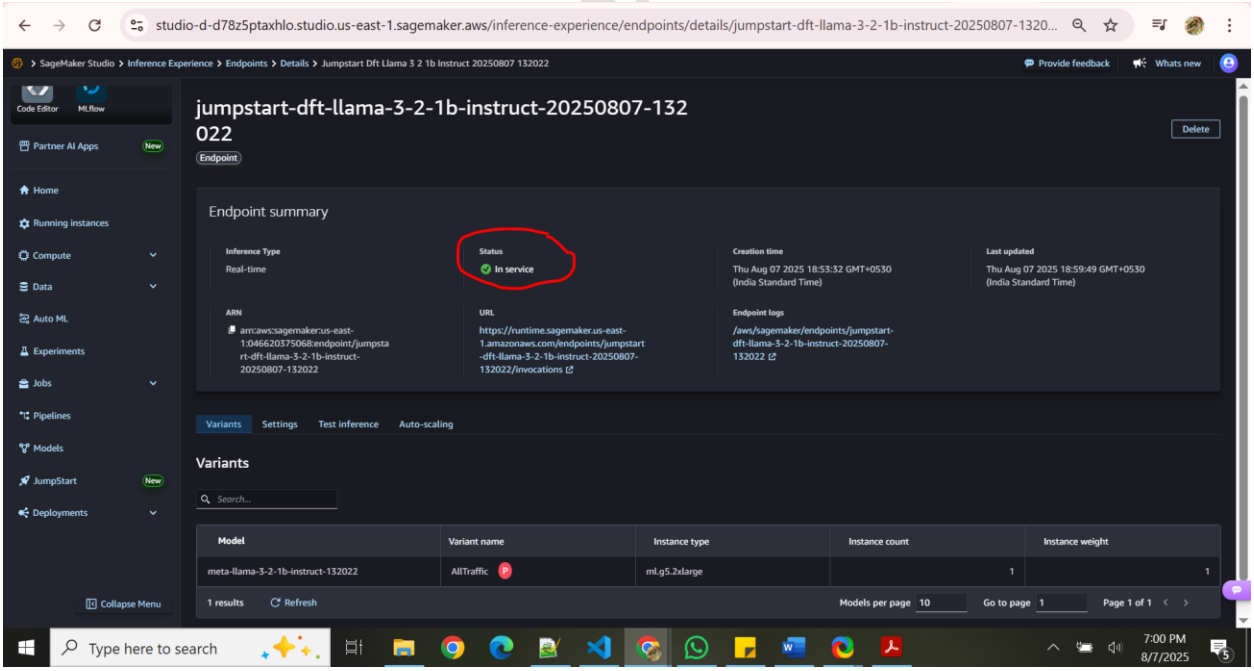



studio-d-d78z5ptaxhlo.studio.us-east-1.sagemaker.aws/jumpstart/SageMakerPublicHub?provider=meta&search="meta+llama+3.2+1b+instruct"
SageMaker Studio > Jumpstart > SageMakerPublicHub
Provide feedback
Whats new
Data
Auto ML
Experiments
Jobs
Pipelines
Models
JumpStart
New
Deployments
Collapse Menu
Meta
Explore popular and trending models from Meta including Llama, Code Llama, and more.
Models 2
meta llama 3.2 1b instruct
Action
Bedrock Ready
Evaluative
Trainable
Meta Llama 3.2 1B Instruct
by Meta
Text Generation
Meta Llama 3.2 1B Instruct Neuron
by Meta
Text Generation
2 results
Go to page 1
Page 1 of 1
Task
Image Segmentation
Image-Text-To-Text
Text Classification
Type here to search
6:49 PM
8/7/2025


studio-d-d78z5ptaxhlo.studio.us-east-1.sagemaker.aws/jumpstart/SageMakerPublicHub/Model/meta-textgeneration-llama-3-2-1b-instruct
SageMaker Studio > Jumpstart > SageMakerPublicHub > Model > Meta Textgeneration Llama 3 2 1b Instruct
Provide feedback
Whats new
Data
Auto ML
Experiments
Jobs
Pipelines
Models
JumpStart
New
Deployments
Collapse Menu
Meta Llama 3.2 1B Instruct
by meta
Train
Deploy
Optimize
Evaluate
About
Notebooks
About
Model Information
The Meta Llama 3.2 collection of multilingual large language models (LLMs) is a collection of pretrained and instruction-tuned generative models in 1B and 3B sizes (text in/text out). The Llama 3.2 instruction-tuned text only models are optimized for multilingual dialogue use cases, including agentic retrieval and summarization tasks. They outperform many of the available open source and closed chat models on common industry benchmarks.
Model Developer: Meta
Model Architecture: Llama 3.2 is an auto-regressive language model that uses an optimized transformer architecture. The tuned versions use supervised fine-tuning (SFT) and reinforcement learning with human feedback (RLHF) to align with human preferences for helpfulness and safety.
Training Data
Params
Input modalities
Output modalities
Context Length
GQA
Shared Embeddings
Token count
Knowledge cutoff
Preview notebooks
Tags
Meta
Text Generation
Author
Meta
Provider
Meta
Task
Text Generation
Type here to search
6:50 PM
8/7/2025

Follow the same process used in the practice section to deploy the LLAMA model.





38. Use the Predictor class in the SageMaker Python SDK with the new endpoint **as listed in steps 20 - 35**.
    Test the model with the same prompting techniques to evaluate the differences.