Developing Generative AI Applications with AWS- Labs

Lab 1a Perform Text Generation

```
# Objective: Create an email to be sent to a client who provided negative feedback on the service
# provided by our customer support engineer. Invoke the model using bedrock API directly. Not using
# bedrock agent or langchain. Prompt is given, but no context info.
# Model : amazon.titan-text-express

#Create a service client by name using the default session.
import json # need this module to work with JSON data
import os # functions for interacting with the operating system, such as environment variables.
import sys # access to system-specific parameters and functions

import boto3
import botocore

module_path = ".." # Sets the module_path variable to the parent directory of the current directory.

sys.path.append(os.path.abspath(module_path)) #Adds the absolute path of the module_path to the
sys.path list. This allows Python to find modules in the specified directory.

bedrock_client = boto3.client('bedrock-runtime', region_name=os.environ.get
("AWS_DEFAULT_REGION", None))  #Creates a service client for the Amazon Bedrock Runtime service
using the default session.


# create the prompt

prompt_data = """
Command: Write an email from Bob, Customer Service Manager, AnyCompany to the customer
"John Doe" who provided negative feedback on the service provided by our customer support
engineer"""
# The above code constructs a JSON payload that can be used to invoke an Amazon Bedrock model
# for generating an email. The prompt data specifies the desired content and style of the email.


# Creates a JSON string representing the input data for the Bedrock model.
body = json.dumps({
   "inputText": prompt_data,
   "textGenerationConfig":{     # Defines the configuration settings for the text generation process:
      "maxTokenCount":8192,
      "stopSequences":[],
      "temperature":0,
      "topP":0.9
      }
   })


#invoke model # This code snippet invokes a specified Amazon Bedrock model
```

```
modelId = 'amazon.titan-text-express-v1' # change this for a different version from the model
provider
accept = 'application/json' # response body type
contentType = 'application/json' # request body type
outputText = "\n"

try:
   # Attempts to invoke the Bedrock model using the bedrock_client object.
   # body is the input data.
   response = bedrock_client.invoke_model(body=body, modelId=modelId, accept=accept,
contentType=contentType)

   # Decodes the JSON response body received from the model.
   response_body = json.loads(response.get('body').read())
   outputText = response_body.get('results')[0].get('outputText')

except botocore.exceptions.ClientError as error:

   if error.response['Error']['Code'] == 'AccessDeniedException':
         print(f"\x1b[41m{error.response['Error']['Message']}\
            \nTo troubeshoot this issue please refer to the following resources.\
             \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
             \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")

   else:
      raise error


# The relevant portion of the response begins after the first newline character
# Below we print the response beginning after the first occurrence of '\n'.

email = outputText[outputText.index('\n')+1:] # extracts the email body
print(email)
```

Lab 1b Perform text generation using a prompt that includes context

```
# Objective: Create an email to be sent to a client who provided negative feedback on the service
# provided by our customer support engineer. Use LangChain. Use LangChain to provide
# PromptTemplate with variables to include customer name, original mail from customer as
# context etc. Model - llama3-8b-instruct

# Model configuration
from langchain_aws import ChatBedrock  # Imports the ChatBedrock class from the langchain_aws
module
from langchain_core.output_parsers import StrOutputParser  # which handles parsing text output.

model_id = "meta.llama3-8b-instruct-v1:0"  # bedrock model to be used
model_kwargs =  {
      "max_gen_len": 512,  # max no of tokens (words /subwords) to generate in the response.
      "temperature": 0,
```

```
    "top_p": 1,
}

# LangChain class for chat
chat_model = ChatBedrock(  # Creates an instance of the ChatBedrock class, which represents the
                           # Bedrock model and provides an interface for interacting with it.
  client=bedrock_client, # bedrock_client object, is an instance of the
                         # boto3 client for Amazon Bedrock Runtime.
  model_id=model_id,
  model_kwargs=model_kwargs,
)


# Create a prompt template that has multiple input variables
# Imports the PromptTemplate class from the langchain.prompts module, which is used to create
# parameterized prompts for language models.
from langchain.prompts import PromptTemplate

# Creates an instance of the PromptTemplate class, defining a prompt template with multiple
# input variables.
multi_var_prompt = PromptTemplate(
  input_variables=["customerServiceManager", "customerName", "feedbackFromCustomer"],
  template="""
        # 'template' defines the template string for the prompt, including placeholders
        # for the input variables.

Human: Create an apology email from the Service Manager {customerServiceManager} at
AnyCompany to {customerName} in response to the following feedback that was received from the
customer:
<customer_feedback>
{feedbackFromCustomer}
</customer_feedback>

Assistant:"""
)

# Pass in values to the input variables
# Replaces the placeholders in the prompt template with the specified values for the input variables:
prompt = multi_var_prompt.format(customerServiceManager="Bob Smith",
                    customerName="John Doe",
                    feedbackFromCustomer="""Hello Bob,
  I am very disappointed with the recent experience I had when I called your customer support.
  I was expecting an immediate call back but it took three days for us to get a call back.
  The first suggestion to fix the problem was incorrect. Ultimately the problem was fixed after three
days.
  We are very unhappy with the response provided and may consider taking our business
elsewhere.
  """
  )


# get number of tokens
```

```
num_tokens = chat_model.get_num_tokens(prompt) # finds no of tokens in the given prompt
print(f"Our prompt has {num_tokens} tokens") # Prints a message indicating the number of tokens in
the prompt.

# Invokes the chat_model with the provided prompt. The response from the model is stored in
# the response variable.
response = chat_model.invoke(prompt)

# Configure a Chain to parse output
chain = StrOutputParser() # This class is used to parse the output from the model.

# Calls the invoke method on the chain object, passing the response from the model as input.
# The parsed output is stored in the formatted_response variable.
formatted_response=chain.invoke(response)
print(formatted_response)
```

Lab 2a Text summarization with small files

```
# Objective: Summarize the given text which is entered as part of prompt. No LangChain and Bedrock
# agent.  Only direct SDK/APIs. In the second part of the exercise, the output is printed out as a
# stream.  Model - amazon.titan-text-premier

# Create a service client by name using the default session.
import json
import os
import sys
import warnings # Used to filter warnings and suppress unwanted messages.

import boto3
import botocore

# Suppresses all warnings that might be raised during the execution of the code
warnings.filterwarnings('ignore')
module_path = ".."
sys.path.append(os.path.abspath(module_path))

bedrock_client = boto3.client('bedrock-
runtime',region_name=os.environ.get("AWS_DEFAULT_REGION", None))


prompt_data = """

Please provide a summary of the following text:
AWS took all of that feedback from customers, and today we are excited to announce Amazon Bedrock, \
a new service that makes FMs from AI21 Labs, Anthropic, Stability AI, and Amazon accessible via an API. \
Bedrock is the easiest way for customers to build and scale generative AI-based applications using FMs, \
democratizing access for all builders. Bedrock will offer the ability to access a range of powerful FMs \
for text and images—including Amazons Titan FMs, which consist of two new LLMs we're also announcing \
today—through a scalable, reliable, and secure AWS managed service. With Bedrock's serverless experience, \
customers can easily find the right model for what they're trying to get done, get started quickly, privately \
customize FMs with their own data, and easily integrate and deploy them into their applications using the AWS \
```

tools and capabilities they are familiar with, without having to manage any infrastructure (including integrations \
with Amazon SageMaker ML features like Experiments to test different models and Pipelines to manage their FMs at scale).
"""

```
# request body
# Creates a JSON string representing the input data for the Bedrock model.
body = json.dumps({
   "inputText": prompt_data,
   "textGenerationConfig":{
      "maxTokenCount":2048,
      "stopSequences":[],
      "temperature":0,  # Controls the randomness of the generated text.
      "topP":0.9  # Controls the diversity of the generated text.
      }
   })


#model configuration and invoke the model
modelId = 'amazon.titan-text-premier-v1:0' # change this to use a different vers/model provider
accept = 'application/json'
contentType = 'application/json'
outputText = "\n"

try:
   response = bedrock_client.invoke_model (body=body, modelId=modelId, accept=accept,
contentType=contentType) # invoke the specified bedrock model

   #Decodes the JSON response body received from the model.
   response_body = json.loads(response.get('body').read())

   # Extracts the generated text from the response
   outputText = response_body.get('results')[0].get('outputText')

except botocore.exceptions.ClientError as error:

   if error.response['Error']['Code'] == 'AccessDeniedException':
         print(f"\x1b[41m{error.response['Error']['Message']}\
            \nTo troubeshoot this issue please refer to the following resources.\
             \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
             \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")

   else:
      raise error

# Prints the extracted generated text
print(outputText)
```

Streaming Output Generation

```
#invoke model with response stream
modelId = 'amazon.titan-text-premier-v1:0'
response = bedrock_client.invoke_model_with_response_stream(body=body, modelId=modelId,
accept=accept, contentType=contentType)

# Extracts the response stream from the response object.
stream = response.get('body')

# Converts the response stream into a list of chunks.
output = list(stream)
output

# Imports necessary modules for displaying Markdown content and clearing the output.
from IPython.display import display_markdown, Markdown, clear_output

modelId = 'amazon.titan-text-premier-v1:0'

# Invokes the model with the response stream.
response = bedrock_client.invoke_model_with_response_stream (body=body, modelId=modelId,
accept=accept, contentType=contentType)

# Extracts the response stream.
stream = response.get('body')
output = []  # Initializes an empty list to store the output chunks.
i = 1  # Initializes a counter for tracking the number of chunks processed.
if stream:   # if response stream is not empty
   for event in stream:
      chunk = event.get('chunk')  # Extracts the chunk from the event.
      if chunk:  # Checks if the chunk is not empty
         chunk_obj = json.loads(chunk.get('bytes').decode()) # decodes JSON data in the chunk
         text = chunk_obj['outputText'] # Extracts the generated text from the chunk.
         clear_output(wait=True) # Clears the output in Jupyter Notebook to prevent overlapping text.
         output.append(text) # Appends the generated text to the output list.
         display_markdown(Markdown(''.join(output))) # Displays the concatenated output as
Markdown in Jupyter Notebook.
         i+=1
```

Lab 2b Abstractive Text Summarization

```
# Objective: Summarize the given document. Use LangChain utilities. Divide the doc into chunks
# and process one by one and aggregate the results.  Model - meta.llama3-8b-instruct

#Create a service client by name using the default session.
import json
import os
import sys

import boto3

module_path = ".."
```

```
sys.path.append(os.path.abspath(module_path))
bedrock_client = boto3.client('bedrock-runtime',
region_name=os.environ.get("AWS_DEFAULT_REGION", None))


# model configuration
from langchain_aws import BedrockLLM
modelId = "meta.llama3-8b-instruct-v1:0"
llm = BedrockLLM(
    model_id=modelId,
    model_kwargs={
        "max_gen_len": 2048,
        "temperature": 0,
        "top_p": 1
    },
    client=bedrock_client
)

# Find a text file of Amazon's CEO letter to shareholders in 2022 in the letters directory.
shareholder_letter = "./letters/2022-letter.txt"

# Loads the text file into 'letter'.
with open(shareholder_letter, "r") as file:
    letter = file.read()

# Count the tokens
llm.get_num_tokens(letter)

# You will see a warning indicating the number of tokens in the text file exceeds the
# maximum number of tokens for this model.

# (Deep learning frameworks are not present)
# None of PyTorch, TensorFlow >= 2.0, or Flax have been found.
# Models won't be available and only tokenizers, configuration and file/data utilities can be used.

# tokenizer configuration file is downloaded and processed.
tokenizer_config.json:  100%  26.0/26.0  [00:00<00:00,  875B/s]

# vocabulary file is being downloaded and processed.
vocab.json:  100%  1.04M/1.04M  [00:00<00:00,  31.9MB/s]

# merges file (used for subword tokenization) is downloaded and processed.
merges.txt:  100%  456k/456k  [00:00<00:00,  26.0MB/s]

tokenizer.json:  100%  1.36M/1.36M  [00:00<00:00,  44.5MB/s]

# configuration file for the model is downloaded and processed.
config.json:  100%  665/665  [00:00<00:00,  34.7kB/s]
```

```
# Split the text into smaller chunks because it is too long to fit in the prompt.
# RecursiveCharacterTextSplitter in LangChain supports splitting long text into chunks recursively
# until the size of each chunk becomes smaller than chunk_size. A text is separated with
# separators=["\n\n", "\n"] into chunks, which avoids splitting each paragraph into multiple chunks.

from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n"], chunk_size=4000, chunk_overlap=100
)
# This class RecursiveCharacterTextSplitter is used to split a large text into smaller chunks.
# chunk_size – max no of characters in each chunk
# chunk_overlap - number of characters that should overlap between adjacent chunks.

# Calls the create_documents method, passing a list containing the
# text to be split (letter). The method returns a list of Document objects, where each
# document represents a chunk of the original text.
docs = text_splitter.create_documents([letter])

# get the number of docs
num_docs = len(docs)

# get no of tokens in first doc
num_tokens_first_doc = llm.get_num_tokens(docs[0].page_content)

print(
    f"Now we have {num_docs} documents and the first one has {num_tokens_first_doc} tokens"
)


# Set verbose=True if you want to see the prompts being used
# load_summarize_chain function helps create a "SummarizeChain" object for
# summarizing text using a language model.
from langchain.chains.summarize import load_summarize_chain

# map_reduce means the model will summarize each document individually ("map")
# and then combine the results ("reduce") into a final summary.
summary_chain = load_summarize_chain(llm=llm, chain_type="map_reduce", verbose=False)
# verbose=False means it will suppress the messages during summarizations

#invoke chain
output = "" # initialize empty string
try:
    # generates a summary of given docs and stores it in the output variable.
    output = summary_chain.invoke(docs)

except ValueError as error:
    if  "AccessDeniedException" in str(error):
        print(f"\x1b[41m{error}\
        \nTo troubeshoot this issue please refer to the following resources.\
         \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
         \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")
```

```
        class StopExecution(ValueError):
            def _render_traceback_(self):
                pass
        raise StopExecution

    else:
        raise error

# print output
print(output['output_text'])
```

To view the original text, Fine > New > Terminal
cd LabRepository/letters
cat 2022-letter.txt

Lab 3 Use Bedrock for Q&A

```
# Objective: Answer a question regarding car tyre pressure. The answer is based on the base data in
# foundation model. Generic, sometimes hallucination. In second part, you input a document and
# ask the model to answer based only on that doc.
# Model - amazon.titan-text-express

# Ignore warnings and create a service client by name using the default session.
import json
import os
import sys
import warnings

import boto3
import botocore

# Suppresses all warnings that might be raised during the execution of the code
warnings.filterwarnings('ignore')
module_path = ".."
sys.path.append(os.path.abspath(module_path))
bedrock_client = boto3.client('bedrock-runtime',
region_name=os.environ.get("AWS_DEFAULT_REGION", None))


prompt_data = """You are a helpful assistant. Answer questions in a concise way. If you are unsure
about the answer say 'I am unsure'

Question: How can I fix a flat tire on my AnyCompany AC8?
Answer:"""
parameters = {
    "maxTokenCount":512,
    "stopSequences":[],
    "temperature":0,
```

```
    "topP":0.9
    }


#model configuration
# Creates a JSON string representing the input data for the Bedrock model.
body = json.dumps({"inputText": prompt_data, "textGenerationConfig": parameters})

modelId = "amazon.titan-text-express-v1"  # Id of the bedrock model used
accept = "application/json"  # response body type
contentType = "application/json"  # request body type

try:
    # invoke the specified Bedrock model using the bedrock_client object
    response = bedrock_client.invoke_model(
        body=body, modelId=modelId, accept=accept, contentType=contentType
    )

    # Decodes the JSON response body received from the model.
    response_body = json.loads(response.get("body").read())
    answer = response_body.get("results")[0].get("outputText")
    print(answer.strip()) # remove whitespaces at the start and end

except botocore.exceptions.ClientError as error:
    if  error.response['Error']['Code'] == 'AccessDeniedException':
        print(f"\x1b[41m{error.response['Error']['Message']}\
        \nTo troubeshoot this issue please refer to the following resources.\
         \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
         \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")
        class StopExecution(ValueError):
            def _render_traceback_(self):
                pass
        raise StopExecution
    else:
        raise error


# query regarding a fake brand
prompt_data = "How can I fix a flat tire on my Amazon Tirana?"
body = json.dumps({"inputText": prompt_data,
                   "textGenerationConfig": parameters})
modelId = "amazon.titan-text-express-v1"  # change this to use a different version from the model
provider
accept = "application/json"
contentType = "application/json"

response = bedrock_client.invoke_model(
    body=body, modelId=modelId, accept=accept, contentType=contentType
)
response_body = json.loads(response.get("body").read())
answer = response_body.get("results")[0].get("outputText")
```

```
print(answer.strip()) # leading/trailing whitespace removed using the strip() method.


# Enter the below as context

context = """Tires and tire pressure:

Tires are made of black rubber and are mounted on the wheels of your vehicle. They provide the necessary grip for driving, cornering, and braking. Two important factors to consider are tire pressure and tire wear, as they can affect the performance and handling of your car.

Where to find recommended tire pressure:

You can find the recommended tire pressure specifications on the inflation label located on the driver's side B-pillar of your vehicle. Alternatively, you can refer to your vehicle's manual for this information. The recommended tire pressure may vary depending on the speed and the number of occupants or maximum load in the vehicle.

Reinflating the tires:

When checking tire pressure, it is important to do so when the tires are cold. This means allowing the vehicle to sit for at least three hours to ensure the tires are at the same temperature as the ambient temperature.

To reinflate the tires:

    Check the recommended tire pressure for your vehicle.
    Follow the instructions provided on the air pump and inflate the tire(s) to the correct pressure.
    In the center display of your vehicle, open the "Car status" app.
    Navigate to the "Tire pressure" tab.
    Press the "Calibrate pressure" option and confirm the action.
    Drive the car for a few minutes at a speed above 30 km/h to calibrate the tire pressure.

Note: In some cases, it may be necessary to drive for more than 15 minutes to clear any warning symbols or messages related to tire pressure. If the warnings persist, allow the tires to cool down and repeat the above steps.

Flat Tire:

If you encounter a flat tire while driving, you can temporarily seal the puncture and reinflate the tire using a tire mobility kit. This kit is typically stored under the lining of the luggage area in your vehicle.

Instructions for using the tire mobility kit:

    Open the tailgate or trunk of your vehicle.
    Lift up the lining of the luggage area to access the tire mobility kit.
    Follow the instructions provided with the tire mobility kit to seal the puncture in the tire.
    After using the kit, make sure to securely put it back in its original location.
    Contact AnyCompany or an appropriate service for assistance with disposing of and replacing the used sealant bottle.
```

```
Please note that the tire mobility kit is a temporary solution and is designed to allow you to drive for a maximum of 10 minutes or 8 km (whichever comes first) at a maximum speed of 80 km/h. It is advisable to replace the punctured tire or have it repaired by a professional as soon as possible."""



question = "How can I fix a flat tire on my AnyCompany AC8?"
prompt_data = f"""Answer the question based only on the information provided between ## and give step by step guide.
#
{context}
#

Question: {question}
Answer:"""


body = json.dumps({"inputText": prompt_data, "textGenerationConfig": parameters})
modelId = "amazon.titan-text-express-v1"
accept = "application/json"
contentType = "application/json"

response = bedrock_client.invoke_model(
    body=body, modelId=modelId, accept=accept, contentType=contentType
)
response_body = json.loads(response.get("body").read())
answer = response_body.get("results")[0].get("outputText")
print(answer.strip())
```

with streaming output

```
# Imports modules for displaying Markdown content and clearing the output in Jupyter Notebook.
from IPython.display import display_markdown,Markdown,clear_output

# response with stream
response = bedrock_client.invoke_model_with_response_stream(body=body, modelId=modelId, accept=accept, contentType=contentType)

stream = response.get('body') # Extracts the response stream from the response object.
output = [] # Initializes an empty list to store the output chunks.
i = 1 # Initializes a counter for tracking the number of chunks processed.
if stream: # if stream is not empty
    for event in stream: # iterates over events in stream
        chunk = event.get('chunk') # extract a chunk from event
        if chunk:  # if chunk is not empty
            # Decodes the JSON data in the chunk.
            chunk_obj = json.loads(chunk.get('bytes').decode())
            text = chunk_obj['outputText']
            clear_output(wait=True)  # Clears the output in Jupyter Notebook to prevent overlapping text.
            output.append(text) # Appends the generated text to the output list.
```

```
        # Displays the concatenated output as Markdown in Jupyter Notebook.
        display_markdown(Markdown(''.join(output)))
        i+=1 # go to next chunk
```

Lab 4 Conversational Interface- Chat with Llama 3 and Titan Premier LLMs

```
# Objective: In this, we create chatbots with different configurations 1) Using chat history from
# LangChain to start the conversation 2) Chatbot using prompt template (LangChain)
# 3) Chat with persona – an AI assistant which takes role of a career coach.   4) Chatbot with context
# where you pass enterprise docs
# Model - amazon.titan-text-premier

#ignore warnings and create a service client by name using the default session.
import json
import os
import sys
import warnings

import boto3

# Suppresses all warnings that might be raised during the execution of the code
warnings.filterwarnings('ignore')
module_path = ".."
sys.path.append(os.path.abspath(module_path))
bedrock_client = boto3.client('bedrock-runtime',
region_name=os.environ.get("AWS_DEFAULT_REGION", None))
# format instructions into a conversational prompt
# Imports the Dict and List types from the typing module. These are used for type hints
# and annotations, improving code readability and maintainability.
from typing import Dict, List

# Defines a function format_instructions that takes a list of dictionaries as input
# and returns a list of strings as output.
def format_instructions (instructions: List[Dict[str, str]]) -> List[str]:
    """Format instructions where conversation roles must alternate
system/user/assistant/user/assistant/..."""

    # Creates an empty list named prompt to store the formatted instructions.
    prompt: List[str] = []

    # Iterates over each instruction in the provided instructions list.
    for instruction in instructions:

        # if instruction is from the system, then append your prompt as below
        # special marker indicating start of a system message.
        # strip leading/trailing white spaces from the content of the instruction
        # <|eot_id|>: This is a special marker indicating the end of a text unit.
        if instruction["role"] == "system":
```

```
            prompt.extend(["<|begin_of_text|><|start_header_id|>system<|end_header_id|>\n",
(instruction["content"]).strip(), " <|eot_id|>"])

        # if instruction is from user then add this special marker indicating the start of a user message.
        elif instruction["role"] == "user":
            prompt.extend(["<|start_header_id|>user<|end_header_id|>\n",
(instruction["content"]).strip(), " <|eot_id|>"])
        else:
            raise ValueError(f"Invalid role: {instruction['role']}. Role must be either 'user' or 'system'.")
    prompt.extend(["<|start_header_id|>assistant<|end_header_id|>\n"])
    return "".join(prompt)


# In this task, you enable the chatbot to carry conversational context across multiple interactions
# with users. Having a conversational memory is crucial for Chatbots to hold meaningful, coherent
# dialogues over time.
# You implement conversational memory capabilities by building on top of LangChain's
# ConversationChain class. The ConversationChain acts as a persistent memory module, allowing
# to store and retrieve historical statements made by both the user and the chatbot agent.

# ConversationChain class used to create conversational models.
from langchain.chains import ConversationChain

# BedrockLLM class for interacting with Amazon Bedrock models.
from langchain_aws import BedrockLLM

# ConversationBufferMemory class for storing conversation history.
from langchain.memory import ConversationBufferMemory


# Creates a BedrockLLM object for the "amazon.titan-text-premier-v1:0" model.
titan_llm = BedrockLLM(model_id="amazon.titan-text-premier-v1:0", client=bedrock_client,
model_kwargs={
    "maxTokenCount":512,
    "stopSequences":[],
    "temperature":0,
    "topP":0.9
    })

# Creates a BedrockLLM object for the "meta.llama3-8b-instruct-v1:0" model
llama3_llm = BedrockLLM(model_id="meta.llama3-8b-instruct-v1:0", client=bedrock_client,
model_kwargs={
        "max_gen_len": 512,
        "temperature": 0,
        "top_p": 1,
    })

# Creates a memory object to store past conversation history.
memory = ConversationBufferMemory()

# Creates a ConversationChain object (the core conversational model).
```

```
conversation = ConversationChain(
    llm=llama3_llm, verbose=True, memory=memory
)
# verbose=True enables detailed logging of the conversation process.

# Creates a list of instructions defining the initial user input for the conversation.
instructions = [{"role": "user", "content": "Hi there!"}]

try:
    # Attempts to predict the response using the predict method of the ConversationChain object.
    # By using the predict method, you ensure that the model takes into account the previous
    # interactions and provides a coherent response within the ongoing conversation.
    print(conversation.predict(input=format_instructions(instructions)))

except ValueError as error:
    if  "AccessDeniedException" in str(error):
        print(f"\x1b[41m{error}\
        \nTo troubeshoot this issue please refer to the following resources.\
         \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
         \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")
        class StopExecution(ValueError):
            def _render_traceback_(self):
                pass
        raise StopExecution
    else:
        raise error


# The model has responded with an initial message. Now, you ask it a few questions.
instructions = [{"role": "user", "content": "Give me a few tips on how to start a new garden."}]

# Calls the predict method on the conversation object to generate a response based on the
# provided instructions. The format_instructions function (assumed to be defined elsewhere)
# formats the instructions into a suitable input format for the model.
print(conversation.predict(input=format_instructions(instructions)))

# Build on the questions. Ask a question without mentioning the word garden to see if
# the model can understand the previous conversation.
instructions = [{"role": "user", "content": "bugs"}]
print(conversation.predict(input=format_instructions(instructions)))

# finishing the conversation
instructions = [{"role": "user", "content": "That's all, thank you!"}]
print(conversation.predict(input=format_instructions(instructions)))
```

ChatBot using Prompt Template

```
#default prompt template
# ConversationBufferMemory class is used to store conversation history.
from langchain.memory import ConversationBufferMemory
```

```
# PromptTemplate class is used to define prompt templates for language models.
from langchain.prompts import PromptTemplate

# initialize chat history
chat_history = []

# turn verbose to true to see the full logs and documents
# Creates an instance of the ConversationChain class, which represents a conversational model.
qa= ConversationChain(
    llm=llama3_llm, verbose=False, memory=ConversationBufferMemory() #memory_chain
)

# Prints a message indicating the default prompt template used by the ConversationChain object.
print(f"ChatBot:DEFAULT:PROMPT:TEMPLATE: is ={qa.prompt.template}")


#  prompt for a conversational agent
# This function returns a list of strings by adding formatting text based on if the conversation
# is from user or system
def format_prompt(actor:str, input:str):
    formatted_prompt: List[str] = []
    if actor == "system":
        prompt_template="""<|begin_of_text|><|start_header_id|>{actor}<|end_header_id|>\n{inp
        ut}<|eot_id|>"""
    elif actor == "user":
        prompt_template="""<|start_header_id|>{actor}<|end_header_id|>\n{input}<|eot_id|>"""
    else:
        raise ValueError(f"Invalid role: {actor}. Role must be either 'user' or 'system'.")

    # Creates a PromptTemplate object from the specified prompt_template.
    prompt = PromptTemplate.from_template(prompt_template)

    # Replace placeholders with the actual actor and input values.
    formatted_prompt.extend(prompt.format(actor=actor, input=input))

    # Adds a marker indicating the start of the assistant's response.
    formatted_prompt.extend(["<|start_header_id|>assistant<|end_header_id|>\n"])

    return "".join(formatted_prompt)


# chat user experience
# Imports the ipywidgets library for creating interactive widgets.
import ipywidgets as ipw
# Imports functions for displaying and clearing output in Jupyter Notebook.
from IPython.display import display, clear_output

class ChatUX:
    """ A chat UX using IPWidgets
    """
```

```
    # Initializes a new ChatUX object.
    def __init__(self, qa, retrievalChain = False):
        self.qa = qa # An instance of the ConversationChain object
        self.name = None
        self.b=None

        # retrievalChain is an optional boolean parameter (default: False). This is likely used
        # for a specific type of conversation chain with retrieval capabilities
        self.retrievalChain = retrievalChain
        self.out = ipw.Output()

        # self.qa: Stores the ConversationChain object.
        # self.name: Stores the user input text field (initially None).
        # self.b: Stores the "Send" button (initially None).
        # self.retrievalChain: Stores the retrievalChain value.
        # self.out: Stores an ipw.Output widget to capture output within the chat window.


    def start_chat (self): # Starts the chat session
        print("Starting chat bot")
        display(self.out) # Displays the output widget using display.
        self.chat(None) # Calls the chat method to initiate the first chat interaction.


    def chat (self, _): # Handles the chat interaction loop.
                        # Underscore is a dummy argument for the callback function
        if self.name is None: # If none, it means user hasn't entered any text yet
            prompt = ""    # sets the prompt to an empty string.
        else:
            prompt = self.name.value  # retrieves the user input from self.name.value

        # If the user input is 'q', 'quit', or 'Q', it ends the chat
        if 'q' == prompt or 'quit' == prompt or 'Q' == prompt:
            with self.out:
                print ("Thank you, that was a nice chat !!")
            return
        elif len(prompt) > 0:
            with self.out:
                thinking = ipw.Label(value="Thinking...")  # create thinking label
                display(thinking)
                try:
                    if self.retrievalChain:
                        result = self.qa.invoke({'question': prompt})
                    else:
                        result = self.qa.invoke({'input': format_prompt("user",prompt)}) #, 'history':
chat_history})
                except:
                    result = "No answer"
                thinking.value=""
                print(f"AI:{result}")
                self.name.disabled = True
                self.b.disabled = True
```

```
            self.name = None

        if self.name is None:
            # Creates an ipw.Text widget (text input field) for the user to enter their message.
            # Creates an ipw.Button widget labeled "Send".
            # Connects the button's on_click event to the chat method, triggering the chat loop again
            # when the user clicks "Send".
            # Displays the user input field and send button together in an ipw.Box.
            with self.out:
                self.name = ipw.Text(description="You:", placeholder='q to quit')
                self.b = ipw.Button(description="Send")
                self.b.on_click(self.chat)
                display(ipw.Box(children=(self.name, self.b)))


# start chat
chat = ChatUX(qa)
chat.start_chat()  # Initiate chat session
```

ChatBot with Persona

```
# In this task, Artificial Intelligence(AI) assistant plays the role of a career coach. Role play
# dialogue requires pre-populating the conversation with a user message before starting the
# chat. A ConversationBufferMemory is used to pre-populate the dialog.

# conversation context as a career coach
# This class is used to store conversation history and context
memory = ConversationBufferMemory()
# Adds an initial user message to the conversation history. The message defines the context for
# the conversation, informing the model that it will be acting as a career coach.
memory.chat_memory.add_user_message(format_prompt("user","Context: You will be acting as a
career coach. Your goal is to give career advice to users. For questions that are not career related,
don't provide advice. Say, I don't know."))

# Creates a ConversationChain object, which represents the conversational model.
conversation = ConversationChain(
    llm=llama3_llm, verbose=True, memory=memory
)

# Calls the predict method on the conversation object to generate a response based on the
# provided instructions.
print(conversation.predict(input=format_prompt("user","What are the career options in AI?")))

# question with different context
conversation.verbose = False
print (conversation.predict (input=format_prompt ("user","How to fix my car?")))
```

Chatbot with Context

```
# In this task, you ask the chatbot to answer questions based on context that was passed to it.

# This class provides an interface for using Amazon Bedrock embedding models within
# LangChain applications.
from langchain_aws.embeddings import BedrockEmbeddings

# FAISS is a Facebook AI Similarity Search library that can be used to efficiently store
# and search embeddings.
from langchain.vectorstores import FAISS

# This class is used to define prompt templates for language models.
from langchain.prompts import PromptTemplate

# Creates an instance of the BedrockEmbeddings class
# The BedrockEmbeddings object will be used to generate embeddings for text data,
# which can then be stored and searched using the FAISS library.
br_embeddings = BedrockEmbeddings(model_id="amazon.titan-embed-text-v1",
client=bedrock_client)


# In order to use embeddings for search, you need a store that can efficiently perform
# vector similarity searches. In this notebook, you use FAISS, which is an in-memory store.
# To permanently store vectors, you can use Knowledge Bases for Amazon Bedrock, pgVector,
# Pinecone, Weaviate, or Chroma.

# This code snippet sets up a vector store using the FAISS library in LangChain to store and
# search embeddings generated by the Bedrock embedding model.
from langchain.document_loaders import CSVLoader
from langchain.text_splitter import CharacterTextSplitter # Splits text documents into smaller chunks

# Provides an abstraction layer for the vector store.
from langchain.indexes.vectorstore import VectorStoreIndexWrapper

# Creates CSVLoader instance to load data from the CSV file
loader = CSVLoader("../rag_data/Amazon_SageMaker_FAQs.csv") # --- > 219 docs with 400 chars

# Loads the documents from the CSV file and stores them in the documents_aws variable.
documents_aws = loader.load() #

# Prints a message indicating the number of documents loaded
print(f"documents:loaded:size={len(documents_aws)}")

# Splits the documents into smaller chunks based on characters.  Chunk size in terms of characters
docs = CharacterTextSplitter(chunk_size=2000, chunk_overlap=400,
separator=",").split_documents(documents_aws)

# Prints the number of documents after splitting and chunking.
print(f"Documents:after split and chunking size={len(docs)}")
vectorstore_faiss_aws = None
try:
    # Attempts to create the vector store using FAISS.from_documents
```

```
    vectorstore_faiss_aws = FAISS.from_documents(
        documents=docs,
        embedding = br_embeddings,
        #**k_args
    )

    # Prints a message if the vector store is created successfully.
    print(f"vectorstore_faiss_aws:created={vectorstore_faiss_aws}::")

except ValueError as error:
    if  "AccessDeniedException" in str(error):
        print(f"\x1b[41m{error}\
        \nTo troubeshoot this issue please refer to the following resources.\
         \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
         \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")
        class StopExecution(ValueError):
            def _render_traceback_(self):
                pass
        raise StopExecution
    else:
        raise error

# You can use a Wrapper class provided by LangChain to query the vector database store
# and return the relevant documents. This runs a QA Chain with all default values.

chat_llm=ChatBedrock(
    model_id="amazon.titan-text-premier-v1:0" ,
    client=bedrock_client)
# Creates an instance of the VectorStoreIndexWrapper class. This wrapper class provides a
# unified interface for interacting with different vector store implementations.
wrapper_store_faiss = VectorStoreIndexWrapper(vectorstore=vectorstore_faiss_aws)

# Queries the vector store using the query method.
print(wrapper_store_faiss.query("R in SageMaker", llm=chat_llm))


# For the chatbot, you need context management, history, vector stores, and many
# other components. You start by building a Retrieval Augmented Generation (RAG)
# chain that supports context.

# function is used to create a retrieval chain
from langchain.chains import create_retrieval_chain

# This function creates a chain that combines multiple documents into a single prompt
# and then passes it to a language model for processing.
from langchain.chains.combine_documents import create_stuff_documents_chain

# This class is used to define prompt templates for conversational interactions.
from langchain_core.prompts import ChatPromptTemplate

# Defines the system prompt that will be used in the conversation chain.
```

```
system_prompt = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer "
    "the question. If you don't know the answer, say that you "
    "don't know. Use three sentences maximum and keep the "
    "answer concise."
    "\n\n"
    "{context}"
)

# Creates a ChatPromptTemplate object using the defined system prompt and a placeholder for the
user input.
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),
        ("human", "{input}"),
    ]
)

# Creates a retriever object from the vectorstore_faiss_aws vector store. This retriever
# will be used to retrieve relevant documents based on the user's query.
retriever=vectorstore_faiss_aws.as_retriever()

# This chain will be used to generate responses based on the retrieved documents and
# the user's query.
question_answer_chain = create_stuff_documents_chain(chat_llm, prompt)

# This chain will be used to retrieve relevant documents from the vector store and then
# pass them to the question-answering chain for processing.
rag_chain = create_retrieval_chain (retriever, question_answer_chain)

# Invokes the rag_chain with the query "What is sagemaker?"
response = rag_chain.invoke ({"input": "What is sagemaker?"})
print (response)

# Creates a ChatUX object that provides a chat interface
chat = ChatUX (rag_chain, retrievalChain=True)

# Start the chat
chat.start_chat()  # Only answers will be shown here, and not the citations

# Overall, this code snippet sets up a retrieval augmented generation (RAG) chain that
# combines a vector store for document retrieval with a language model for question answering
```

Lab 5 Bedrock for code generation

```
# Objective: In this, we use LLM to generate code. Use PromptTemplate from LangChain.
# create chatbots with different configurations 1) Using chat history from
# Model - meta.llama3-8b-instruct

# The prompt describes the task of analysing sales data from a CSV file and outlines the
# specific requirements for the Python code.

# Create a service client by name using the default session.
import json
import os
import sys

import boto3

module_path = ".."
sys.path.append(os.path.abspath(module_path))
bedrock_client = boto3.client('bedrock-runtime',
region_name=os.environ.get("AWS_DEFAULT_REGION", None))

# Lab setup - create sample sales.csv data for this lab.

# create sales.csv file
import csv

# Creates a list of lists containing the data to be written to the CSV file.
data = [
    ["date", "product_id", "price", "units_sold"],
    ["2023-01-01", "P001", 50, 20],
    ["2023-01-02", "P002", 60, 15],
    ["2023-01-03", "P001", 50, 18],
    ["2023-01-04", "P003", 70, 30],
    ["2023-01-05", "P001", 50, 25],
    ["2023-01-06", "P002", 60, 22],
    ["2023-01-07", "P003", 70, 24],
    ["2023-01-08", "P001", 50, 28],
    ["2023-01-09", "P002", 60, 17],
    ["2023-01-10", "P003", 70, 29],
    ["2023-02-11", "P001", 50, 23],
    ["2023-02-12", "P002", 60, 19],
    ["2023-02-13", "P001", 50, 21],
    ["2023-02-14", "P003", 70, 31],
    ["2023-03-15", "P001", 50, 26],
    ["2023-03-16", "P002", 60, 20],
    ["2023-03-17", "P003", 70, 33],
    ["2023-04-18", "P001", 50, 27],
    ["2023-04-19", "P002", 60, 18],
    ["2023-04-20", "P003", 70, 32],
    ["2023-04-21", "P001", 50, 22],
    ["2023-04-22", "P002", 60, 16],
    ["2023-04-23", "P003", 70, 34],
```

```
    ["2023-05-24", "P001", 50, 24],
    ["2023-05-25", "P002", 60, 21]
]

# Write data to sales.csv
with open ('sales.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile) # used to write the data to the CSV file.
    writer.writerows(data)

print ("sales.csv has been created!")


# define prompt template
# This class is used to define prompt templates for language models.
from langchain_core.prompts import PromptTemplate

def format_prompt(actor:str, input:str):
    match actor:  # pattern matching construct
        case "user":
            # Sets the prompt template for the user role.
            prompt_template =
"""<|begin_of_text|><|start_header_id|>{actor}<|end_header_id|>\n\n{input}<|eot_id|><|start_h
eader_id|>assistant<|end_header_id|>\n\n
"""
            # create prompt template object
            prompt = PromptTemplate.from_template(prompt_template)
            # Returns the formatted prompt. Replace with actual values of actor and input.
            return prompt.format(actor=actor, input=input)
        case _:
            print("requested actor >" + actor + "< is not supported")
            return "" # return empty string

# Create the prompt
# Analyzing sales

prompt_data = """

You have a CSV, sales.csv, with columns:
- date (YYYY-MM-DD)
- product_id
- price
- units_sold

Create a python program to analyze the sales data from a CSV file. The program should be able to
read the data, and determine below:

- Total revenue for the year
- Total revenue by product
- The product with the highest revenue
- The date with the highest revenue and the revenue achieved on that date
- Visualize monthly sales using a bar chart
```

Ensure the code is syntactically correct, bug-free, optimized, not span multiple lines unnecessarily, and prefer to use standard libraries. Return only python code without any surrounding text, explanation or context.

"""

```
# format the prompt for the language model
prompt=format_prompt("user",prompt_data)

# creates a JSON object containing parameters for a language model generation task.
body = json.dumps({
   "prompt": prompt,
   "max_gen_len": 2048,
   "temperature": 0,
   "top_p": 1,
   })


modelId = "meta.llama3-8b-instruct-v1:0" # sets the model id
response = bedrock_client.invoke_model(body=body, modelId=modelId)

# Loads the response body as a JSON object.
response_body = json.loads(response.get('body').read())

# Extracts the "generation" key from the response body. If the key exists, it returns
# the corresponding value (a list of generated text outputs).
output_list = response_body.get("generation", [])
print(output_list)

# generated code below
import csv
import datetime
import matplotlib.pyplot as plt
from collections import defaultdict

def analyze_sales(file_name):
   sales_data = []
   with open(file_name, 'r') as file:
      reader = csv.DictReader(file)
      for row in reader:
         sales_data.append({
            'date': datetime.datetime.strptime(row['date'], '%Y-%m-%d').date(),
            'product_id': row['product_id'],
            'price': float(row['price']),
            'units_sold': int(row['units_sold'])
         })

   total_revenue = sum(sale['price'] * sale['units_sold'] for sale in sales_data)
   print(f'Total revenue for the year: {total_revenue}')

   revenue_by_product = defaultdict(int)
```

```
    for sale in sales_data:
        revenue_by_product[sale['product_id']] += sale['price'] * sale['units_sold']
    print('Total revenue by product:')
    for product, revenue in revenue_by_product.items():
        print(f'Product {product}: {revenue}')

    max_revenue_product = max(revenue_by_product, key=revenue_by_product.get)
    print(f'The product with the highest revenue: {max_revenue_product}')

    max_revenue_date = max(sales_data, key=lambda x: x['price'] * x['units_sold'])
    print(f'The date with the highest revenue: {max_revenue_date["date"]}, Revenue:
{max_revenue_date["price"] * max_revenue_date["units_sold"]}')

    monthly_sales = defaultdict(int)
    for sale in sales_data:
        monthly_sales[sale['date'].strftime('%Y-%m')] += sale['price'] * sale['units_sold']
    months = list(monthly_sales.keys())
    months.sort()
    plt.bar(months, [monthly_sales[month] for month in months])
    plt.xlabel('Month')
    plt.ylabel('Revenue')
    plt.title('Monthly Sales')
    plt.show()

analyze_sales('sales.csv')
```

Lab 6 Bedrock model integration with Langchain Agents¶

```
# This is an implementation of ReAct framework. A state machine with multiple states.
# Execution guided by the reasoning by LLM. It access external tools for certain stages.

# You create an agent graph, which is a state machine that defines the flow of the conversation
# and the interaction with the tools. You define nodes with associated functions that update the
# state based on input.  The LangChain Agent can utilize the output of one tool as the input for
# the next step.    Model: anthropic.claude-3-sonnet


# Use a plan-and-execute agent that determines the order of actions and implements
# them using the tools available to the agents.
```

Environment Setup

```
# Create a service client by name using the default session.
import math
import numexpr # efficient numerical computations
import json
import datetime
import sys
import os
```

```
import boto3 # library for interacting with AWS services.

module_path = ".."
# Adds the module_path to the Python path, allowing modules from that directory to be imported.
sys.path.append(os.path.abspath(module_path))

# Creates a client for the Amazon Bedrock Runtime service.
bedrock_client = boto3.client('bedrock-runtime',
region_name=os.environ.get("AWS_DEFAULT_REGION", None))
model_id = "anthropic.claude-3-sonnet-20240229-v1:0"

# Now, create an instance of LangChain's ChatBedrock class, which allows you to
# interact with a conversational AI model hosted on Amazon Bedrock

# create an instance of the ChatBedrock.
# This class provides an interface for interacting with Amazon Bedrock language models.
from langchain_aws import ChatBedrock

# Creates an instance of the ChatBedrock class
chat_model=ChatBedrock(
   model_id=model_id ,
   client=bedrock_client)

# Invoke model
chat_model.invoke("What is AWS? Answer in a single sentence")
```

Synergize Reasoning and Acting

```
# The ReAct framework enables large language models to interact with external tools to
# obtain additional information that results in more accurate and fact-based responses.
# Large language models can generate both explanations for their reasoning and task-specific
# responses in an alternating fashion.
# Producing reasoning explanations enables the models to infer, monitor, and revise action
# plans, and even handle unexpected scenarios. The action step allows the models to
# interface with and obtain information from external sources such as knowledge bases
# or environments.

# This decorator is used to define functions as tools within the LangChain framework.
from langchain_core.tools import tool

# You define a function get_product_price that serves as a tool within the Langchain
# framework and retrieves the price of the product specified in the query from sales.csv file

@tool # Mark this function as a LangChain tool
def get_product_price(query:str):
   "Useful when you need to lookup product price"
   import csv # csv module
   prices = {} # Initializes an empty dictionary
   try:
      file=open('sales.csv', 'r')
```

```
    except Exception as e:
        return ("Unable to look up the price for " + query)

    # Creates a DictReader object from the csv module, which allows reading CSV data into
    #  dictionaries, where each row becomes a dictionary with column names as keys and
    # corresponding values as entries.
    reader = csv.DictReader(file)
    for row in reader:
        prices[row['product_id']] = row['price']
    file.close()
    # Extracts the first line from the query, removes leading/trailing whitespace, and stores it in qstr.
    qstr=query.split("\n")[0].strip()
    try:
            return ("Price of product "+qstr+" is "+prices.get(qstr)+"\n")
    except:
            return ("Price for product "+qstr+" is not available"+"\n")

# In the next cell, you define a function calculator that serves as a tool within the Langchain
# framework. This tool enables a language model to perform mathematical calculations
# by evaluating a given expression using Python's numexpr library.

@tool
# Calculator function as Langchain tool
def calculator(expression: str) -> str:
    """Use this tool to solve a math problems that involve a single line mathematical expression.
    Use math notation and not words.
    Examples:
        "5*4" for "5 times 4"
        "5/4" for "5 divided by 4"
    """
    try:
        return str(
            numexpr.evaluate(
            expression.strip(), # Removes leading/trailing whitespace
            global_dict={},
            local_dict={} # add math constants, if needed
            )
        )
    except Exception as e:
        return "Rethink your approach to this calculation"

# define list of tools
tools = [get_product_price, calculator]


# Run helper functions to print trace output to a file. These functions provide logging
# and message manipulation mechanisms within the LangChain framework.
# Imports the various message classes used in LangChain
from langchain_core.messages import HumanMessage, SystemMessage, AIMessage, ToolMessage
def output_trace (element:str, trace, node=True):
    global trace_handle
```

```
    # code to manage when trace handle enabled/disabled
    if trace_enabled:
        print (datetime.datetime.now(), file=trace_handle)
        print(("Node: " if node else "Edge: ")+ element, file=trace_handle)
        if element == "ask_model_to_reason (entry)":
            for single_trace in trace:
                print(single_trace, file=trace_handle)
        else:
            print(trace, file=trace_handle)
        print('----', file=trace_handle) # separator line to separate trace entries

def consolidate_tool_messages(message):
    tool_messages=[]
    for msg in message: # iterates thru each element in the msg list
        if isinstance(msg, ToolMessage):
            tool_messages.append(msg)
    # Returns the list of extracted tool messages. This allows functionalities in the
    # framework to work specifically with tool messages.
    return tool_messages
```

Building an Agent Graph

```
# In this task, you will be creating an agent graph for a conversational AI system that can
# interact with external tools. The agent graph is a state machine that defines the flow of
# the conversation and the interaction with the tools.

# Below, you define nodes with associated functions that update the state based on input.
# Connect nodes using edges, where the graph transitions from one node to the next.
# Incorporate conditional edges to route the graph to different nodes based on specific
# conditions. Finally, compile the agent graph to prepare it for execution, handling
# transitions and state updates as defined.

from typing import Literal  # class used for type annotations.
from langgraph.graph import StateGraph, MessagesState # used to define and manage the state
graph for the agent.
from langgraph.prebuilt import ToolNode # prebuilt component used to represent tools within the
agent's graph.

# ToolNode is a prebuilt component that runs the tool and appends the tool result to the messages
tool_node = ToolNode(tools) # This tool node will be used to represent the available tools within the
agent's graph.

# let the model know the tools it can access
# Binds the tools list to the chat_model
model_with_tools = chat_model.bind_tools(tools)

# The following function acts as the conditional edge in the graph.
# The next node could be the tools node or the end of the chain.
# This function takes a MessagesState object as input and returns a string
# indicating the next node in the graph.
def next_step (state: MessagesState) -> Literal["tools", "__end__"]:
```

```
    messages = state["messages"] # extracts list of msgs
    last_message = messages[-1]
    if last_message.tool_calls: # check if last msg include tool calls
        output_trace("next_step: Proceed to tools", last_message, node=False)
        return "tools" # Returns the string "tools" to indicate that the next node in the graph should be
the "tools" node.
    output_trace("next_step: Proceed to end", last_message, node=False)
    return "__end__"  # chain should terminate

#The following node function invokes the model that has information about the available tools
def ask_model_to_reason(state: MessagesState):
    messages = state["messages"]
    output_trace("ask_model_to_reason (entry)", consolidate_tool_messages(messages))
    try:
        response = model_with_tools.invoke(messages) # invokes the model
    except Exception as e:
        output_trace("ask_model_to_reason", messages)
        output_trace("ask_model_to_reason", "Exception: "+str(e))
        return {"messages": [messages.append("Unable to invoke the model")]}
    output_trace("ask_model_to_reason (exit)", response)
    return {"messages": [response]} # returns the response from the model

# This graph will represent the flow of execution within the agent.
agent_graph = StateGraph(MessagesState)

# Describe the nodes.
# The first argument is the unique node name, and the second argument is the
# function or object that will be called when the node is reached
# Adds a node named "agent" to the graph, associating it with the ask_model_to_reason function.
agent_graph.add_node("agent", ask_model_to_reason)
# Adds a node named "tools" to the graph, associating it with the tool_node object.
agent_graph.add_node("tools", tool_node)

# Connect the entry node to the agent for the graph to start running
agent_graph.add_edge("__start__", "agent")

# Once the graph transitions to the tools node, the graph will transition to the agent node
agent_graph.add_edge("tools", "agent")

# The transition out of the agent node is conditional.
# If the output from ask_model_to_reason function included a call to the tools, call the tool;
# otherwise end the chain
agent_graph.add_conditional_edges (
    "agent",
    next_step,
)

# Compile the graph definition so that it can run
react_agent = agent_graph.compile()
```

```
# Next, you visualize the compiled graph.

# Imports the Image and display classes from the IPython.display module, which are
# used for displaying images in Jupyter notebooks.
from IPython.display import Image, display

try:
    # Generates a Mermaid diagram representation of the graph and converts it to a PNG image.
    display(Image(react_agent.get_graph().draw_mermaid_png()))
except Exception:
    # This requires some extra dependencies and is optional
    Pass

# Now, you run helper function to print the graph output
# This function takes a stream object as input.
def print_stream(stream):
    for s in stream:
        message = s["messages"][-1]
        if isinstance(message, tuple): # If the message is a tuple, prints it directly.
            print(message)
        else:
            message.pretty_print()  # Calls the pretty_print method on the message object to display it in
a formatted way.


# add one or more questions you want to ask the agent about product pricing from the sales.csv

#list of questions
questions=[]  # create empty list
# Appends the first question to the list.
questions.append("How much will it cost to buy 3 units of P002 and 5 units of P003?")
# add second question (commented out)
#questions.append("How many units of P010 can I buy with $200?")
# third question – commented out
#questions.append("Can I buy three units of P003 with $200? If not, how much more should I spend
to get three units?")
#questions.append("Prices have gone up by 8%. How many units of P003 could I have purchased
before the price increase with $140? How many can I buy after the price increase? Fractional units
are not possible.")


# To understand the steps involved in reasoning, enable trace. However, keep the trace
# output manageable by commenting out all but one question in the list above. Alternatively,
# you can disable trace and run all the questions.
trace_enabled=True # This variable controls whether tracing information is logged during the agent's
execution.

if trace_enabled:
    file_name="trace_"+str(datetime.datetime.now())+".txt"
    trace_handle=open(file_name, 'w')
```

```
# Now, invoke the agent with the questions from the list above
# This defines a system message that will be sent to the agent before the questions. It
# instructs the agent on how to approach the task and emphasizes using the calculator tool provided.
system_message="Answer the following questions as best you can. Do not make up an answer. Think
step by step. Do not perform intermediate math calculations on your own. Use the calculator tool
provided for math calculations."

for q in questions:  # iterates thru each question in the list
    # Creates an input dictionary named inputs containing two messages:  1) First message is from
   # the system with system_message instructing the agent. 2) Second message is from the user
   # with the actual question (q) from the list.
   inputs = {"messages": [("system", system_message), ("user", q)]}
   config={"recursion_limit": 15} # configuration dictionary to set parameters

   # Calls the stream method of the react_agent object.
   print_stream(react_agent.stream(inputs, config, stream_mode="values"))
   print("\n"+"================================ Answer complete
================================="+"\n")  # Prints separator line

if trace_enabled:
   trace_handle.close()
```

The End